Introduction to OO Design GRASP Patterns

Dave Levitt

CS 2000: Systems Analysis and Design

Agenda

- Discuss E3
- Design Concepts
- GRASP Patterns
- Lab

Design Concepts

- Review of what we've done so far:
  - We've captured stakeholders requirements.
    - How? What RUP phase(s)? So now we know all the requirements. Right?
  - We've taken a look at the business entities in the problem domain.
    - What do we call this artifact? What is the RUP call this discipline? What RUP Phase? Are we done identifying all of the business entities? Why or why not?
  - We now want to create a blueprint of the implementation. This is DESIGN!
    - Last week, we learned some notation. Notation != Design

Design Concepts (cont'd)

- Design is a very creative activity:
  - Think about clothing or room designers.
    - What image(s) come to mind? What 'rules' do you think they follow? Is it art? Science? Both? Neither?
  - Now think about an OO designer.
    - What image(s) come to mind? Are they different? Why or why not?
  - Design is a little bit of art and a little bit of science.

Design Concepts (cont'd)

- Design is all about tradeoffs:
  - For a clothing designer, what might some tradeoff's be?
  - For OO design, what might some of these tradeoffs be?
    - We want to consider these tradeoffs when we design our classes, their responsibilities, our interfaces to them and their associations to others classes.
  - It would be helpful if we had some guidance, instead of having to start from scratch.
    - Guidance gained through experience.
    - Guidance gained though collaboration with team members
    - Guidance gained though research.
    - Guidance gained through patterns

Design Concepts (cont'd)

- Think of a dressmakers pattern. How might you describe it?

- Think of a design pattern. How might you describe it?
  - A solution that has been proven effective in one domain and is likely to effective in others.

Design Concepts (cont'd)

- Patterns:
  - Are not invented. They are harvested from existing solutions.
  - Are given a name to aid in communications.
  - Are documented in a rigorous fashion
  - Sometimes conflict with each other. For example: you apply a patterns to solve one problem, but by doing so, you may introduce others.
    - This is called a contradiction, or side-effect.
    - These are the tradeoffs designers have to deal with!

GRASP Patterns

- Larman introduces a set of basic patterns that he calls GRASP: ***G***eneral ***R***esponsibility ***A***ssignment ***S***oftware ***P***attern

- Five GRASP Patterns:
  - Information Expert
  - Creator
  - High Cohesion
  - Low Coupling
  - Controller

- Later chapters will introduce other GRASP Patterns which will be discussed in CS 2100.

GRASP - Information Expert

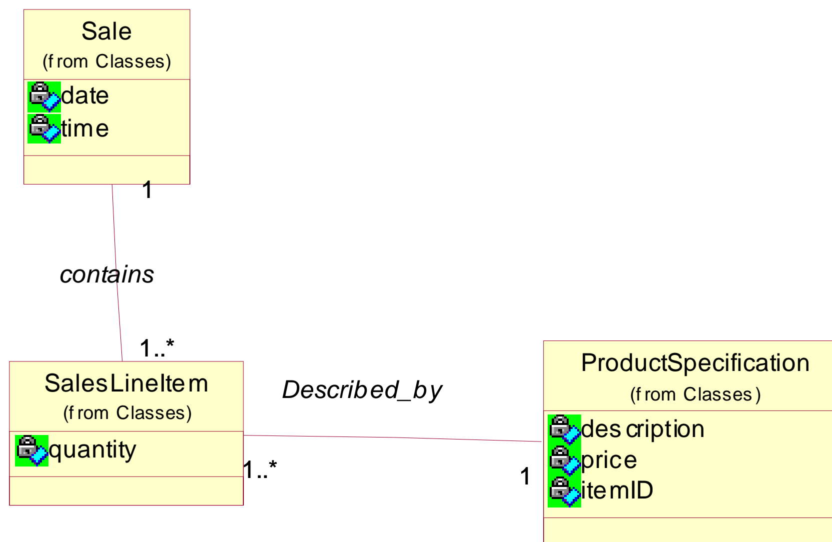
- **Problem:** A system will have hundreds of classes. How do I begin to assign responsibilities to them?
- **Solution:** Assign responsibility to the ***Information Expert*** – the class that has the information necessary to fulfill the responsibility.
- **Mechanics:**
  - Step 1: Clearly state the responsibility
  - Step 2: Look for classes that have the information we need to fulfill the responsibility.
  - Step 3: Domain Model or Design Model?
  - Step 4: Sketch out some interaction diagrams.
  - Step 5: Update the class diagram.

GRASP - Information Expert (cont'd)

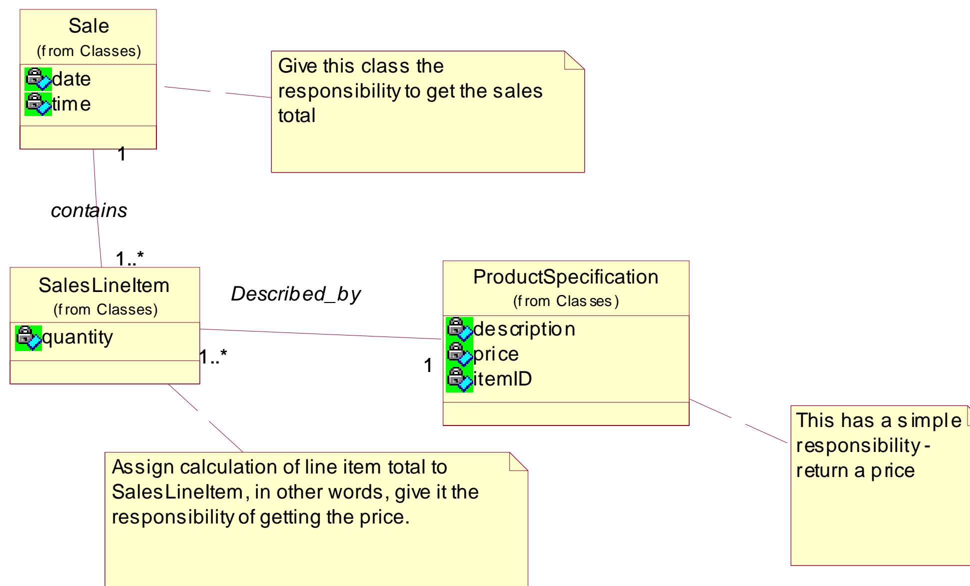
- **Discussion Points:**
  - Throughout the process of assigning a responsibility, you may discover many lower-level responsibilities. Example: Profit & Loss Statement. What should you do then?

- **Contradictions:**
  - Sometimes application of the Expert Pattern is undesirable. For example: Who should save the sale to a database?
    - ***Stay tuned….. We'll discuss problems like this in Design Models?***

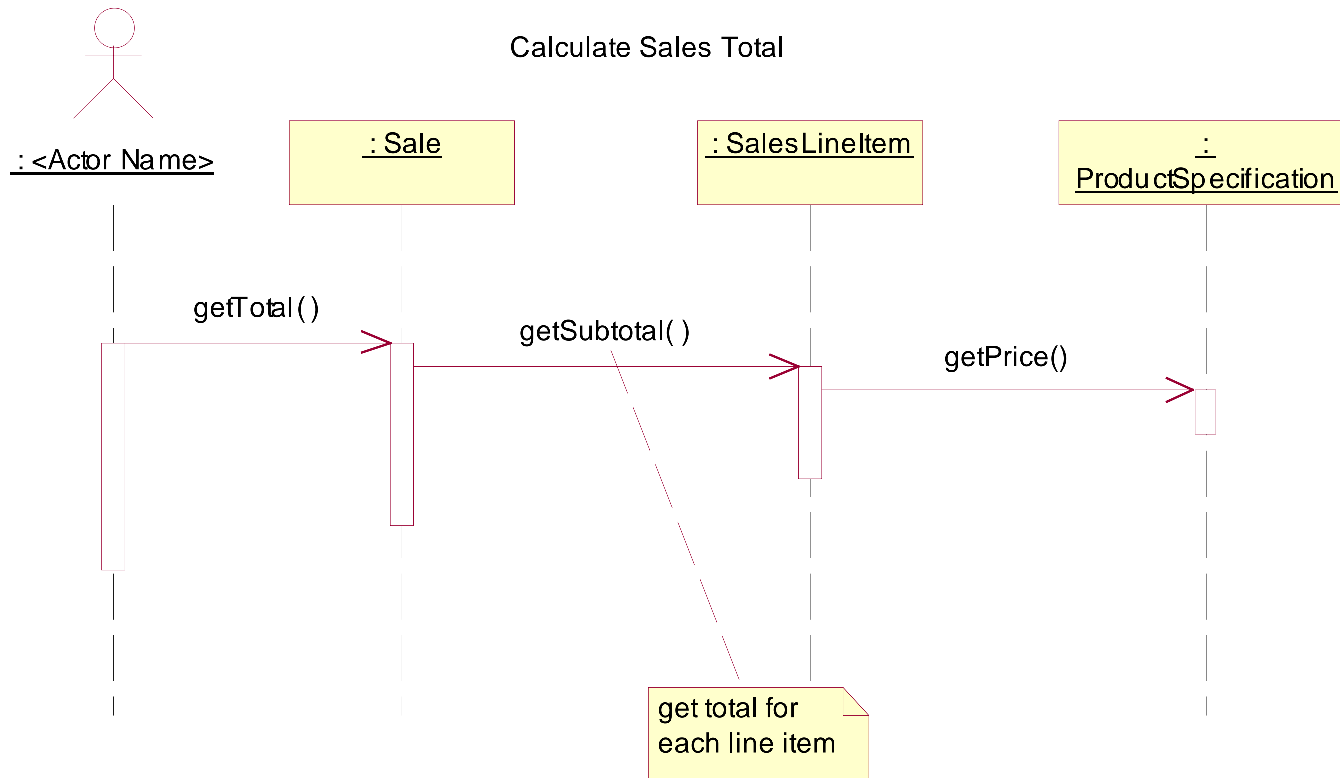
GRASP - Information Expert (cont'd)



GRASP - Information Expert (cont'd)



GRASP - Information Expert (cont'd)



GRASP – Creator Pattern

- **Problem:** Who creates new instances of some class?
- **Solution:** Let class A create an instance of class B if….
  - » **A *aggregates*** (whole-part relationship) B objects
  - » A ***contains*** B objects
  - » **A *records*** instances of B objects
  - » A ***closely uses*** B objects
  - » A has ***initialization data*** that is needed when creating B objects.
- **Mechanics:** Step 1: Look at Domain / Design model and ask: "Who should be creating these classes"?

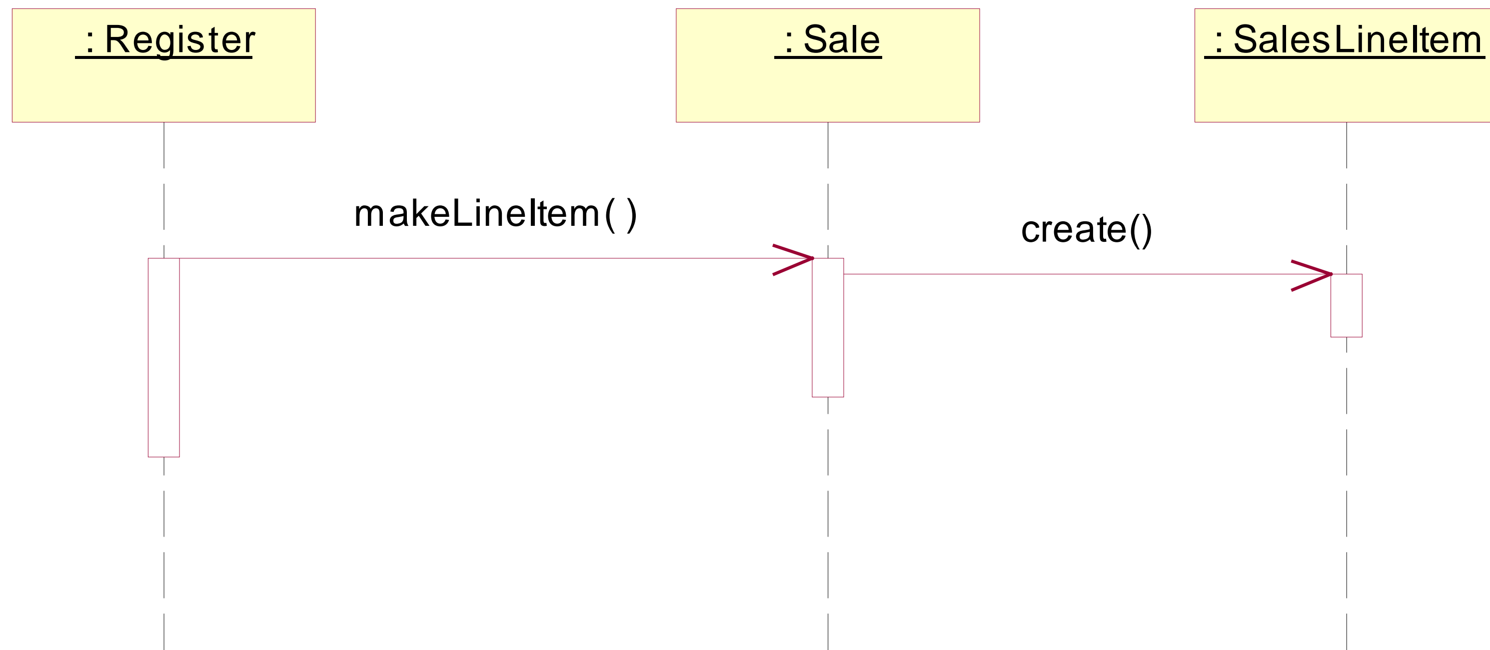
  Step 2: Look for classes that *create, aggregate, etc.*

  Step 3: Sketch or update interaction / class diagrams.

Contradictions:

Sometimes it's time consuming to create objects.

GRASP – Creator (con't)



GRASP – Low Coupling

- Concept – Coupling:
  - Coupling refers to dementedness, or connectedness.
  - Our goal is to design for low coupling, so that changes in one element (sub-system, system, class, etc.) will limit changes to other elements.
  - Low coupling supports increased reuse. Why?
  - Taken to the extreme, what if we were to design a set of classes with no coupling. Is this possible?
  - We can't avoid coupling, but we want to make sure we understand the implications of introducing it and/or the tradeoffs of reducing it.

GRASP – Low Coupling (cont'd)

- **Problem:** How do you support low dependency, low change impact, and increased reuse.
- **Solution:** Assign responsibility so responsibility remains low.
- **Mechanics:** Look for classes with many associations to other classes.

  Look for a methods that rely on a lot of other methods (or methods in other classes, I.e. dependencies.

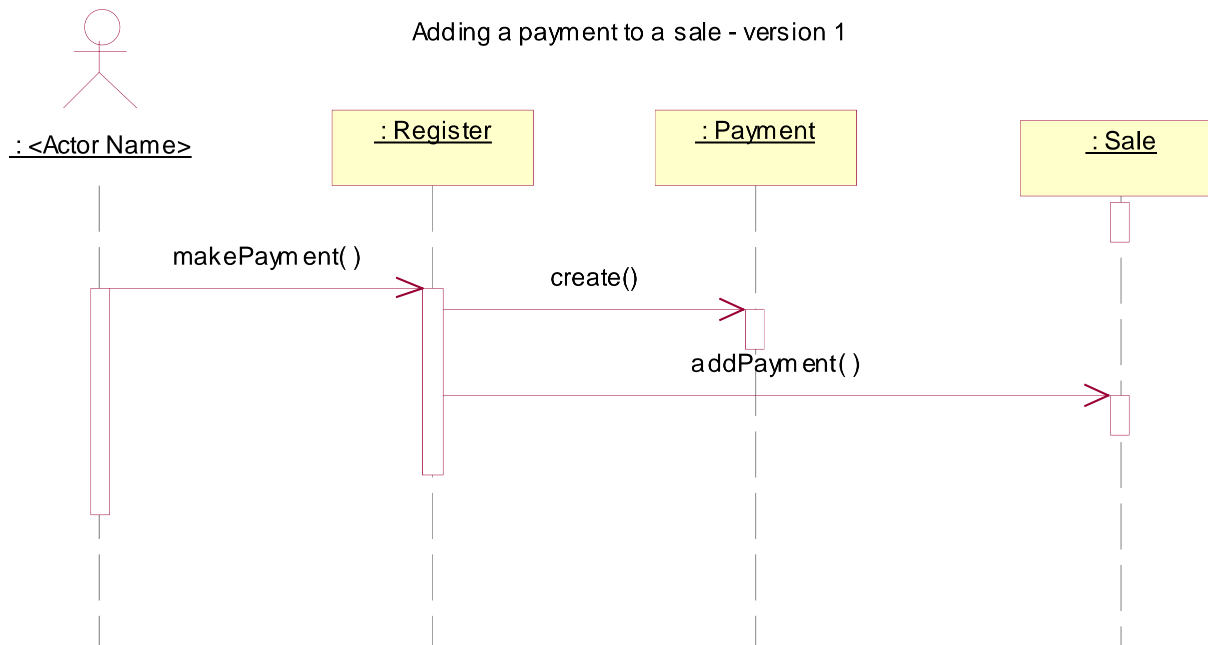
  Rework your design as needed.

GRASP – Low Coupling (cont'd)

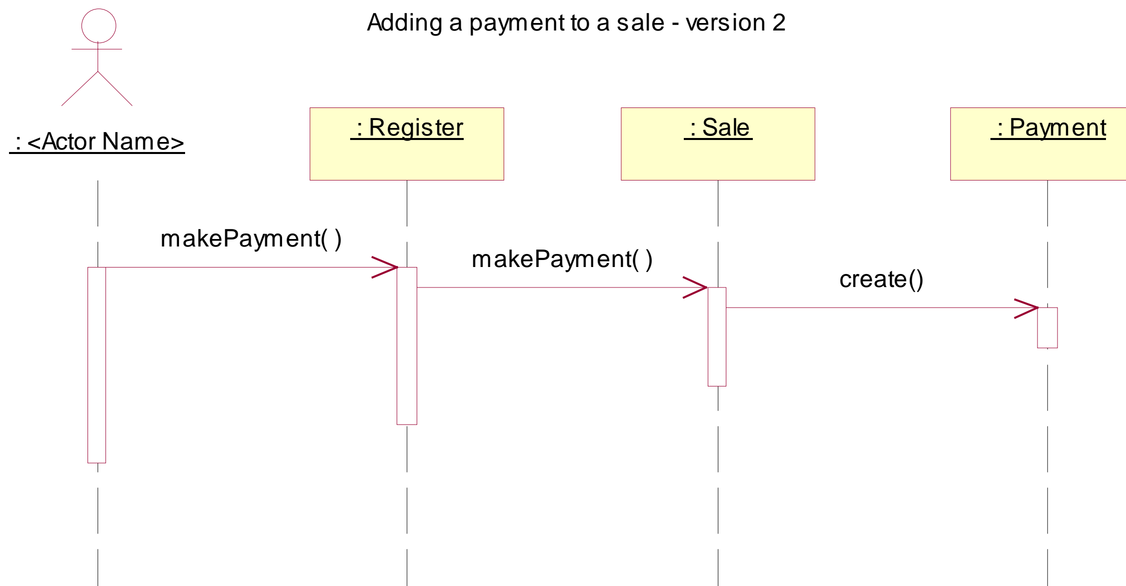
- **Discussion points:**
  - **Is it OK to couple your design to let's say a particular database?**

  - **Is it OK to couple a java application to a 3rd party java toolset? How about the java *Util*** library.

GRASP – Low Coupling (con't)



GRASP – Low Coupling (con't)



This version let's Sale create a Payment - as opposed to Register creating one. Which version supports lower coupling? Why?

GRASP – High Cohesion

- **Concept – Cohesion:**
  - Cohesion is a measure of “relatedness”.
  - High Cohesion says elements are strongly related to one another.
  - Low Cohesion says elements are not strongly related to one another. Ex:
    - System level: ATM with a use case (function) called “Teller Reports”.
    - Class level: A Student class with a method called “getDrivingRecord().
    - Method level: Methods with the word “And” or “Or” in them.
    - Also applies to subsystem (package) level, component level, etc.
  - Designs with low cohesion are difficult to maintain and reuse.
  - One of the fundamental goals of an effective design is to achieve high cohesion with low coupling (which we will see later).

GRASP: High Cohesion (cont'd)

- **Problem:** How do you keep complexity manageable?
- **Solution:** Assign responsibility so that cohesion remains high.
- **Mechanics:** Look for classes with too-few or disconnected methods.
  Look for methods that do to much (hint: method name)
  Rework your design as needed.
- **Contradictions:**
  - **High Cohesion and low coupling are competing forces. More later.**

GRASP – High Cohesion (con’t)

- Additional:
  - Larman describes degrees of cohesion, from very low to high. For now, just consider that in general, classes with a small number of functionally related methods is more desirables than bloated classes with disconnected methods.

- Look at the sequence diagrams for adding a payment to a sale, but this time from the perspective of cohesion. Which version supports high cohesion, and why?

- This example illustrates that:
  - High cohesion and low coupling can be competing forces.
  - Design is not so clear cut, e.g. it is not always an exact science, but more about heuristics and tradeoffs.

GRASP - Controller

- **Problem:** Who handles events from external actors, e.g. startup(), playSongs(), etc?
- **Solution:** Assign the responsibility to a controller class, such as:
  - » **A class that represents the overall system, device, or subsystem. Example: Jukebox.**
  - » **A class that represents a use case. Example: makeSaleHandler. makeSaleCoordinator, etc.**
  - » **These classes often don't do the work, but delegate it to others.**
- **Additional**: The decision to create system controllers vs. use case controllers are often driven by the dynamics of high cohesion vs. low coupling.

GRASP – Controller (cont'd)

- Additional (cont'd):
  - Watch for bloated controllers. This could be a sign of what kind of failure?

  - From an architectural perspective, systems are usually broken up onto layers, or tiers. Example 2 tier client-server, 3 tier, n tier, etc.
    - The UI objects are in the presentation layer.
    - The business objects representing the problem domain are in the application or domain layer.
    - The objects representing databases, network connections, etc. are in the Technical or infrastructure layer(s)
    - Layering and related decisions are frequently made by an Architect.

  - Controllers typically receive requests from UI (interface objects). It is not a good idea to put business logic in UI objects. Why?

Assigning Responsibilities – Other Sources

- CRC Cards:
  - Another popular technique to assigning responsibilities to classes is to use CRC cards. CRC = **C**lass: **R**esponsibility: **C**ollaboration.
  - Introduced by Kent Beck and Ward Cunnigham.
  - Popularized by Rebecca Wirfs-Brock: Designing Object Oriented Software
- Design Heuristics by Authur Riel. CS 2100 – Stay tuned!
- Web sites: Martin Fowler, Peter Coad.