Lecture 18: Responsibility-Driven Design, Part 1

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2003

Credit where Credit is Due

- Some material presented in this lecture is taken from Object Design: Roles, Responsibilities, and Collaborations. © Addison Wesley/Pearson Education, 2003. ISBN 0-201-37943-0
- Some additional material is drawn from Craig Larman's Applying UML and Patterns. © Craig Larman, 2002 ISBN 0-13-092569-1

Background

- Responsibility-Driven Design is a design technique for transforming analysis models into a design that can be implemented by developers
- During analysis, it makes use of use cases/scenarios to capture functional requirements and CRC cards to identify an initial class model
- It then develops a design by stepping through a process of assigning responsibilities to these candidates until all functional requirements have been covered

RDD Life Cycle (I)

- Product Definition and Planning
  - Define project goals
- Responsibility-Driven Analysis
  - System Definition
    - Develop system architecture
    - Identify initial system concepts
    - Identify system responsibilities

RDD Life Cycle (II)

- Responsibility-Driven Analysis (cont.)
  - Detailed Description
    - Specify development environment
    - Write use cases/scenarios
    - Analyze/identify non-functional requirements
    - Document system dynamics
      - Use activity diagrams to show constraints between use cases
    - User Interface Design
      - Develop screen specifications and navigation model

RDD Life Cycle (III)

- Responsibility-Driven Analysis (cont.)
  - Object Analysis
    - Identify domain objects using CRC cards
    - Document additional concepts and terms
      - Such as a domain's business rules
- Exploratory Design
  - Associate domain objects with execution-oriented objects
  - Assign responsibilities to objects
    - Again use CRC cards
  - Develop initial collaboration model
    - Sequence/Collaboration Diagrams

RDD Life Cycle (IV)

- Design Refinement
  - Justify trade-offs/Document design decisions
  - Distribute application control
    - Identify control styles
    - Identify patterns of decision making and delegation in the object model
  - Refine static/dynamic associations between classes
    - Class Diagram
  - Revise model to make it more maintainable, flexible, and consistent
    - Use design patterns, simplify interfaces, etc.
  - Document design using UML
- Ready for Implementation!

Additional Background

- RDD develops an analysis model using CRC cards
- As covered previously, on the unlined side of the index card, we are supposed to specify a candidate's name, purpose, patterns, and stereotypes
  - In lecture 11, I left the exact nature of stereotypes undefined; lets define them before looking at RDD in more depth

Stereotypes (I)

- A well-defined object supports a clearly defined role
  - some roles are application-specific
    - such as an image import plug-in for Adobe Photoshop
  - but some roles are generic; RDD refers to generic roles as “role stereotypes” or just “stereotypes”
    - do not confuse this concept with UML’s stereotype extension mechanism

Stereotypes (II)

- Information Holder
  - knows and provides information
- Structurer
  - maintains relationships between objects and information about those relationships
- Service Provider
  - performs work for other objects
- Coordinator
  - reacts to events by delegating tasks to others
- Controller
  - makes decisions and closely directs others’ actions
- Interfacer
  - transforms information and requests between different parts of our system

Stereotypes (III)

- In general an object can have more than one stereotype, but…
  - the idea is to try to assign the stereotype that captures the major role of a class
    - for instance, any class that has an attribute is technically an “information holder”
    - but if that class mainly responds to events, you should classify it as a coordinator

Stereotypes (IV)

- Why are stereotypes useful?
  - Stereotypes can help to identify responsibilities for a class
    - For instance, a service provider will have responsibilities for “doing” or “performing” specific services
    - You’ll be able to classify responsibilities
      - “Oh, that is something that an interfacer must do!”
  - Thinking in terms of stereotypes can make you a better designer
    - You will become comfortable partitioning an application into objects that play these roles
      - Can be used to test the completeness of a design
        - “I haven’t created any controllers yet, I can’t possibly be done!”
    - You will learn how certain roles relate to design patterns

Responsibilities

- The core of RDD is assigning responsibilities to objects
  - So, what is a responsibility?
- Responsibilities are general statements about software objects; they include
  - The actions an object performs
  - The knowledge an object maintains
  - Major decisions an object makes that affect others

Example: Teakettle (I)

- Consider the design of a teakettle
  - What is the right form for a teakettle?
    - A teakettle holds water that can be heated until boiling
    - People can safely pick up a teakettle when it is filled with boiling water and pour a cup of tea
    - By convention, a teakettle whistles when the water boils
  - These characteristics can be restated as responsibilities
    - Pour contents without spilling or splashing
    - Hold water that can be heated until boiling
    - Is safe to hold and carry while water is hot
    - Notify when boiling occurs

Example: Teakettle (II)

- Did we get this right?
- It depends on the boundaries we have set for the problem; in conventional terms we have the bases covered
- But some designers like to redefine the problem: “Its not the teakettle that needs to be designed, but the method of heating the water!”
  - Here the teakettle becomes part of the context, rather than the “form being designed”
  - This type of thinking might lead to innovation such as an “instant hot” unit that heats tap water as it flows through it
- The trick is to know when to indulge this type of thinking; sometimes it leads to innovation but sometimes it adds unnecessary complexity and expense
  - For most people, the conventional teakettle works just fine!

Finding Responsibilities

- Use Cases
  - Identify system responsibilities stated or implied by use cases
  - plug gaps in use cases by developing lower-level responsibilities (and classes)
- Follow “what if…then…and how” chains
- Identify stereotypical responsibilities
- Identify responsibilities to support relationships between candidates
- Patterns (!)

Use Cases and Responsibilities

- Use cases describe our software from the perspective of an outside user
  - They don't tell how something is accomplished
  - We need to "bridge this gap" by transforming these descriptions into explicit statements about actions, information, or decision-making responsibilities
    - This is similar to Maciaszek's step of finding system activities after creating use cases
- Bridging the Gap
  - Identify things the system does and the information it manages
  - Restate these things as responsibilites
  - Break them down into smaller parts if necessary and assign them to appropriate objects

Example: University Enrollment

- A student can register online for classes by filling out and submitting an online registration form for approval. While filling out the registration form, a student can browse course schedules, cross-listed courses, audit degree requirements, and update personal and financial aid information. The student can also access the "waitlist class" and "drop class" functions
- The system should identify problems as courses are added, such as time conflicts, full classes, lack of prerequisites, etc.

Example: Responsibilities

- Generate and display an online registration form (something needs to know the structure of the form and how to display it)
- Provide feedback as the student enters course selections about conflicts or problems (Something needs to check that a student can sign up for a course; a component is also needed to display feedback about the results)
- Provide capabilities for browsing, auditing degree requirements, and updating personal/financial information (browsing sounds like a big responsibility, auditing sounds like a complex process, updating personal information will require specific boundary, controller, and domain classes)
- …

Example: Specific Scenario

- 1. Student logs in
- 2. System verifies that student is eligible to register and displays reg. form
- 3. Student adds courses to schedule
- 4. System verifies schedule and returns approved courses for confirmation
- 5. Student confirms schedule
- 6. System updates course rosters and confirms successful registration

Example: More responsibilities

- Check that student is eligible to register
  - From step 2
- Add student to course rosters
  - From step 6
- Display confirmation of registration
  - From step 6
- Validate each course in schedule meets constraints such as prerequisites, etc.
  - From step 4

Example: Filling in gaps

- These directly derived responsibilities have gaps; ask questions and identify additional responsibilities
  - How are prerequisites specified?
    - A relationship between course objects?
      - Possibly need structurer to handle this
  - What states does a student's schedule go through? "build/submit/confirm"
    - Who manages this life cycle? The schedule object?
  - Does registering happen in "real time"?
  - How much help should the system give to a student when things go wrong?

"What if" scenarios

- Asking "what if" questions can lead to lines of reasoning that identify additional responsibilities
  - What if the database goes down before my schedule is confirmed?
    - Is the student out of luck? Can the schedule be saved elsewhere and retrieved for later submission?
- This type of thinking will lead to new candidates with responsibilities to handle this situation

Stereotypical Responsibilities

- As mentioned before, stereotypes have common sets of responsibilities that can help generate specific responsibilities for objects that play these roles
  - Information holders "know" things
  - Service providers "do" things
  - Structurers "create" and "maintain" things
  - …

Responsibilities from Relationships

- A meeting has attendees
  - Who has the following responsibility:
    - "How many people attended this meeting?"
- Probably the meeting object
  - This responsibility was derived from the relationship between the two objects however

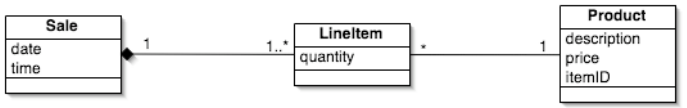
Patterns for Identifying Responsibilities

- Craig Larman has developed patterns for helping to identify responsibilities (we will review four here; he has actually developed nine such patterns)
  - Information Expert (or Expert)
  - Creator
  - Low Coupling
  - High Cohesion

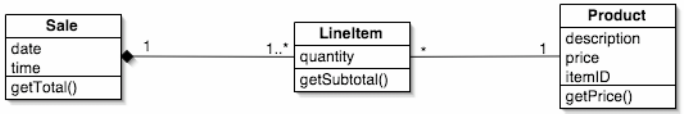
Information Expert

- Assign a responsibility to the class that has the information necessary to fulfill it
  - Consider a "cash register" domain with the following objects: Sale, LineItem, Product
  - Consider the responsibility: "Know the grand total of a Sale"
  - It seems obvious that the sale object should have this responsibility, but lets look at the implications

Sales Example: Class Diagram



If we want to get the total value of a sale, we would need to call a method like getTotal() on Sale; This method would need to call a method like getSubtotal() on LineItem, since LineItem is the "expert" for this information; But this method would need to call getPrice() on Product since only Product has this information



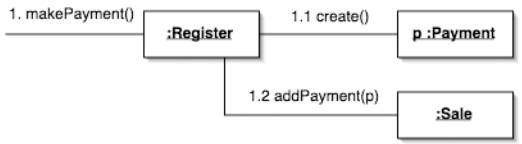
Creator

- Assign class B the responsibility of creating an instance of A if one or more of the following is true
  - B aggregates A
  - B contains A
  - B records instances of A
  - B closely uses A
  - B has the data required to initialize A
- In our previous example, Sale should be assigned the responsibility of creating LineItem objects; this means that Sale will need a method like "addLineItem()" or similar

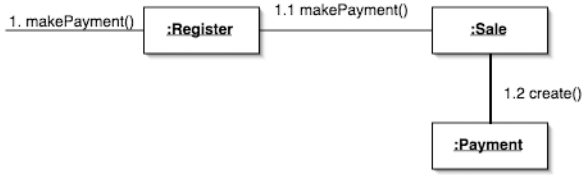
Low Coupling

- Assign a responsibility so that coupling remains low
  - coupling is a measure of how strongly a class is connected to, has knowledge of, or relies on other classes
  - Building on our "cash register" example, consider the classes Payment, Register, and Sale
  - How should we handle the "make payment" responsibility

Two Options



Note that we name the Payment object above, so we can pass it as a parameter.



Which option should we choose?

The first requires the Register object to know about two objects

The second requires Register to know about only one object

All things being equal, this pattern would choose option 2

High Cohesion

- Assign a responsibility so that cohesion remains high
  - In terms of object design, cohesion is a measure of how strongly related and focused the responsibilities are of a class
- In previous example, this pattern would pick option 2 again; The Register object is likely to have many operations that it must handle (or coordinate); if it has to know the details of handling each operation it will lack cohesion

Recording Responsibilities

- Responsibilities should be recorded on CRC cards
- If you can't find a "home" for a responsibility; record it on a post-it node and place it to one side…eventually a home will be found for it
  - or it may need to be decomposed into smaller responsibilities that are easier to assign

Additional Tips (I)

- State responsibilities generically
  - For a customer object, say
    - "Knows name and preferred ways of being addressed"
  - Don't say
    - "Knows first name"
    - "Knows last name"
    - "Knows nick name"
    - …

Additional Tips (II)

- Use strong descriptions
  - Vague responsibilities do not help
    - So use verbs like
      - remove, merge, calculate, activate
    - rather than
      - organize, record, find, process, maintain
- Avoid nonessential responsibilities
- Do not overlap responsibilities
  - For instance do not have a client verify the data it sends AND have the server verify the data it receives; have the server verify and the client be able to handle situations where data is rejected

Testing Candidate Quality

- Once responsibilities have been assigned, check to see that each candidate is well formed
  - Does it stick to its purpose?
  - Are its responsibilities clearly stated?
  - Do its responsibilities match its role?
  - Is it of value to other objects in its neighborhood?
- What's Next?
  - Designing collaborations