Responsibility-Driven Design

- First described by Wirfs-Brock et. al. in *Designing Object-Oriented Software*.
- Emphasizes informal techniques for describing objects, their roles, responsibilities and interactions.
- Focus on
  - developing a model of cooperating, communicating objects
  - dynamics of an object design rather than internal state and object structure

Responsibility-Driven Design

- Three Phases
  - System description
  - Exploratory design
  - Refinement
System Design

- Informal, technology-independent phase
- Artifacts:
  - Context diagram
  - List of actors
  - Scenarios
  - Conversations
  - Action/response descriptions
  - Functional architecture

Exploration

- Steps in this process include:
  - finding classes in the system
  - determining class responsibilities and what knowledge the classes should maintain
  - determining how objects collaborate with other objects
- The process is iterative, with well-defined checkpoints; each stage produces meaningful artifacts
- Artifacts
  - Class descriptions
  - Object collaborations graph

Collaborations

- OO design models the world in terms of collaborating objects
- Collaborations are one-way interactions in which one object requests a service of another object
- The object making the request is the client; the object that receives the request and carries out the service is the server.
- The list of requests that a client can make is described by a contract.
- The system is modeled as clients and servers collaborating as specified by contracts
Client-Server Contract

- Clients and servers are roles; an object can assume either role at any time
- Objects are clients and servers within the system
- Responsibilities are contracts between the objects
- Collaborations are how we determine which clients and which servers are bound by which contracts

CRC cards

- CRC = “Class, Responsibility, Collaborator”
- CRC cards
  - technique for identifying classes, responsibilities, and collaborations between classes
  - props to talk through scenarios to figure out how objects will work together
  - originally introduced to coax experienced and non-experienced programmers into thinking in terms of objects
  - Not part of the UML

CRC cards

- 4” x 6” index cards divided into 3 parts
  - Classname
  - Responsibilities
  - Collaborators

- When one object uses another, the second object is designated as the first object’s collaborator
- A good place to introduce CRC cards is between use cases and interaction diagrams. They will help you to identify the interactions from use cases that go on the interaction diagrams.
CRC Cards

- CRC cards don't include details like attributes and methods. Can't include too much detail because an index card doesn't have much room.
- CRC cards are easy and flexible and very low-tech. Easy to spatially arrange, create and destroy. Encourages experimentation.
- CRC Cards help you to make the paradigm shift to OOP. CRC cards help you to identify objects and to think in terms of an OO solution.

CRC Cards

- CRC cards can be a bridge between the analysis phase and the design phase. They will reveal class responsibilities and collaborations.
- Can be used to discover how to implement use cases. They lead to the discovery of components and responsibilities.
- The process is to walk through use cases. Activities are identified and assigned to some component as a responsibility.
**Using CRC Cards**

- In a CRC exercise:
  - a card represents an instance of an object type
  - the responsibility of the object is identified
  - one person talks through a use case scenario while one or more persons show the objects that work together in the scenario

**Assigning Responsibilities**

- After identifying core classes, assign responsibilities to classes
  - responsibilities for exhibiting behaviors
  - responsibilities for holding knowledge
- Responsibilities (Wirfs-Brock)
  - the knowledge an object maintains
  - the actions an object can perform
- Write responsibilities on CRC cards for each core class

**Assigning Responsibilities**

- Use a combination of brainstorming and role-playing of scenarios to discover responsibilities
- Focus on the what, not the how.
- Grady Booch: "When considering the semantics of classes and objects, there is the tendency to explain how things work; the proper response is 'I don't care.'"
- False semantic distinction among responsibilities may rob us of the ability to use inheritance and polymorphism to advantage.
Assigning Responsibilities

• Assign a responsibility to the class it logically belongs to.
  – “provide” and “update” responsibilities, and responsibilities that require access to or manipulation of an attribute, will belong with the class where the attribute in question is

How do you identify responsibilities?

• Look at the verbs found in the requirements specification and decide which of them represent actions that an object in the system must perform
• Perform a walkthrough of the system, looking for behavior that must occur as a result of input to the system.
• The class name suggests the role the class plays and what the objects must do to fulfill the role
• Remember, responsibilities are more general than operations

Assigning Responsibilities

• Guidelines:
  – evenly distribute system intelligence
  – state responsibilities as generally as possible
  – keep behavior with related information, if any
  – keep information about one thing in one place
  • eliminates duplication which leads to inconsistency
  – share responsibilities among related objects
“Evenly Distribute System Intelligence”

• ‘system intelligence’ means what the system knows, the actions it can perform and the impact it has on other systems
• ‘class intelligence’ means how much it knows or can do, and how many other objects it can affect.
• Centralized control with one intelligent object is a ‘god’ class
  – intelligent object is like main control module of a procedural program
  – other objects are like traditional data structures

• Several disadvantages to centralized control
  – more difficult to modify the system
  – requires more unintelligent classes to implement the same system, thus more effort
• Better approach: design all classes to be equally intelligent (as nearly as possible)
  – each object will know about fewer things
  – more flexible system, easier to modify
  – fewer classes needed to implement a system with comparable functionality

“State responsibilities in general terms”

• A drawing element object knows how to draw itself… for example,
  – a ‘rectangle’ draws an oblong shape
  – a ‘square’ draws a square shape
  – a ‘line’ draws a line
• Keeping responsibilities in general terms can lead to identification of shared common responsibilities
“Keep behavior with related information”

• If an object has to maintain information, it should also have responsibility of performing operations on the information
• If an object needs certain information to perform some of its operations, then the object should maintain that information
• If information changes, the object that needs to know will know; no update messages are required

“Share Responsibilities among Related Objects”

• Example: a drawing editor may need to have the current state of a drawing displayed at all times
  – responsibility: display the drawing being edited
• Several objects cooperate in this responsibility:
  – DrawingEditor knows when drawing has changed and needs to be redrawn
  – Drawing knows which drawing elements to display
  – DrawingElement knows how and where its visual representation should be drawn

Brainstorm, then Refine

• Think simple. Factor out complexity.
• If most the responsibilities fall to one or two classes, then the system is probably biased toward a procedural perspective -- does not take advantage of polymorphism and encapsulation. Many of the classes are reduced to “records” -- simply knowing about the information they hold.
Refine

- Give each class a distinct role in the system. Strive to make each class a well-defined, complete, cohesive abstraction. Such a class has higher probability of being reused.
- Too many responsibilities is a sign of low cohesion. If you can restate the responsibilities with a few abstract terms that still describe the individual responsibilities it replaces, then you would have high cohesion. If the responsibilities are numerous, and you can't describe them with a more general verb phrase you have low cohesion.
- If card starts to fill up
  - collapse responsibilities with a more abstract description
  - split class

Use Abstraction

- Build hierarchies of classes. Abstract the essence of related classes by identifying where they have common responsibilities -- where they do the same thing, but do it differently -- same "what", different "how".
  - That is, look for opportunities for the classes to use polymorphism to implement the same responsibility differently.
- The new parent classes may be abstract classes. That is, no actual objects may ever exist of that type. The abstract class exists to link together similar concrete types of objects.

Be Flexible

- Remember that CRC cards are inexpensive and erasable!!
- Do not hesitate to experiment with different configurations of classes or assignments of responsibilities. Changing the CRC cards is easy in the early stages of a project; changing the code later in the project is not easy.
Collaborators

• Collaborators include other classes the class interacts with to accomplish its responsibilities.
• Collaborators are listed for each responsibility. Therefore, you may have the same collaborator listed more than once on a CRC card.
• Class A collaborates with class B if class A sends a message to class B.

Assigning Collaborations

• Identify relationships among classes
  – Classes must cooperate to accomplish nontrivial tasks
• Pair collaborations with responsibilities on CRC card
• Task of finding collaborators for classes often intermixed with finding responsibilities

Assigning Collaborations

• Use scenario-based role-play to find and/or test these collaborations
  – Scenario is a system behavior and sequence of system events to realize it
  – Simulating execution enables team to discover responsibilities and collaborations and/or check correctness of those already found
• Too many collaborators is a sign of high (tight) coupling.