A Brief Tour of Responsibility-Driven Design in 2004

Rebecca Wirfs-Brock
rebecca@wirfs-brock.com
What Is Responsibility-Driven Design?

A way to design software that…

– emphasizes behavioral modeling of objects’ roles, responsibilities, and collaborations
– uses informal tools and techniques
– enhances development processes from XP (eXtreme Programming) to RUP (Rational Unified Process)

…with responsibility concepts and thinking
Responsibility-Driven Design Resources

Our new book has more techniques and practices.

*Object Design: Roles, Responsibilities and Collaborations*, Rebecca Wirfs-Brock and Alan McKean, Addison-Wesley, 2003

www.wirfs-brock.com for articles & presentations
Responsibility-Driven Design Principles

Maximize Abstraction
   Hide the distinction between data and behavior. Think of objects responsibilities for “knowing”, “doing”, and “deciding”

Distribute Behavior
   Make objects smart—have them behave intelligently, not just hold bundles of data

Preserve Flexibility
   Design objects so they can be readily changed
Different Points-of-View: Different Results

- Data-Driven
- Responsibility-Driven
- Event-Driven
- Rule-Based
- Ad-Hoc

Choice of key design abstractions

Distribution of data and behavior

Patterns of collaboration

Object visibilities

influence
Designing a Horse

Head

Start

Stop

Body

Speed Up

Slow Down

Tail

Legs (4)
Designing a Horse Responsibly
Responsibility-Driven Design Constructs

an application = a set of interacting objects
an object = an implementation of one or more roles
a role = a set of related responsibilities
a responsibility = an obligation to perform a task or know information
a collaboration = an interaction of objects or roles (or both)
a contract = an agreement outlining the terms of a collaboration
Roles and Responsibilities
Role Stereotypes

Stereotypes are simplified views that help you understand an object or component’s purpose.

“Something conforming to a fixed or general pattern; especially a standardized mental picture held in common by members of a group and representing an oversimplified opinion.” — Webster’s Seventh New Collegiate Dictionary

Each object fits at least one stereotype. They can fit more than one. Common blends:

- service provider and information holder,
- interfacer and service provider,
- structurer and information holder.
Stereotypes
simplified views of roles

Controller—Controls application execution
Characterized by decisions it makes
Example: TransactionController

Coordinator—Coordinates actions
Characterized by actions it delegates
Example: ViewCoordinator
Stereotypes
simplified views of roles

**Interfacer**—Communicates actions and intentions between our system and others, or between layers of a system

Characterized by what it communicates with and how well it “hides” their details

Examples: UI objects, an object that “wraps” an interface to another application

**Service Provider**—Performs specific operations on demand.

Characterized by what it does (computation, calculation, transformation)

Example: CreditChecker
Stereotypes
simplified views of roles

**Information Holder**—Holds facts.
Characterized by what it knows
Example: TransactionRecord, Account

**Structurer**—Maintains relationships between others.
Characterized by who it knows and what it knows about them
Example: Order
Layered Architecture

User Interfacers

Controllers and Coordinators

Information-Holders, Service-Providers, and Structurers

External Interfacers

Presentation

Application Coordination & Control

Business Information and Services

Technical Services
Three Uses for Object Role Stereotypes

1. In early modeling, stereotypes help you think about the different kinds of objects that you need

2. You consciously blend stereotypes with a goal of making objects more responsible and intelligent
   – information holders that compute with their information
   – service providers that maintain information they need
   – structurers that interface to persistent stores, and derive new relationships
   – interfacers that transform information and hide many low-level details

3. Study a design to learn what types of roles predominate and how they interact
Informal Technique: CRC Cards

Candidate, Responsibilities, Collaborators

CRC cards are an informal way to record early design ideas about candidates

MessageBuilder
Builds message from selections
Presents guesses to user
Controls the pacing

Message
Presenter

MessageBuilder
Purpose: The MessageBuilder is a hub of activity in the application. It coordinates the timing, the presentation of guesses, the message construction. It centralizes control and is a core element of the control architecture.

Stereotype: Controller? Coordinator?
Purpose: Describing Candidate Roles

An object does and knows certain things for a reason. Briefly, say why it exists and an overview of its responsibilities. Mention one or more interesting facts about the object or a detail about what it does or knows or who it works with.

A compiler is a program that translates source code into machine language.

A FinancialTransaction controls a single accounting transaction performed by our online banking application. Successful transactions result in updates to a customer's accounts.
Look for Appropriate Abstractions

Model an abstraction if it defines responsibilities common to at least three subclasses

Do not include a lower level abstraction if it adds no significant value

Objects can always behave differently by checking and making decisions based on encapsulated state!
What are Responsibilities?

Behavior for
  knowing
  doing
  deciding

Stated at a high level

Assigned to appropriate objects
How Do You State Responsibilities?

They are larger than individual attributes or operations. A single responsibility is often realized by several methods.

Example: A Customer object has a name which may be comprised of a first name, surname, middle name, and there may be titles or nicknames.

A good statement of its responsibility: A customer “knows its name and preferred ways of being addressed.”

Use strong descriptions. The more explicit the action, the stronger the statement.

Stronger verbs: remove, merge, calculate, credit, activate
Weaker verbs: organize, record, process, maintain, accept
CRC - the Responsibilities

- Destination
  - Knows alias of receiver
  - Knows signature of sender
  - Knows address info
  - Sends a message

what it knows

what it does
Guidelines for Assigning Responsibilities

Keep behavior with related information. This makes objects efficient.

Don’t make any one role too big. This makes objects understandable.

Distribute intelligence. This makes objects smart.

Keep information about one thing in one place. This reduces complexity.
Options for Fulfilling a Responsibility

An object can always do the work itself:

- A single responsibility can be implemented by one or more methods
- Divide any complex behavior into two parts
  - One part that defines the sequence of major steps + helper parts that implement the steps
- Send messages to invoke these finer-grained helper methods

Delegate part of a responsibility to one or more helper objects:

- Ask them to do part of the work: make a decision or perform a service
- Ask them relevant questions
Collaborations and Trust Regions
CRC - the Collaborators

Destination

- Knows alias of receiver
- Knows signature of sender
- Knows address info
- Sends a message

Mailer

UserProfile

collaborators

These objects are sent messages by Destination as it performs its responsibilities
Guidelines for Collaborating

Delegate control if possible. Let collaborators be responsible

Look for opportunities to ask for services or direct others’ actions more intelligently

Give objects the ability to both do and know things

Look for ways to make similar things work consistently
Start with rough sketches...
...then get more precise

Show a sequence of messages between these objects

  Label message arrows with names of requests

  Show arguments passed along with requests when it is important to understanding what information (objects) pass between collaborators

  Show return values when it is important that information is returned from a request
Sequence Diagram: Adding a Guess To A Message

- /Selector
- /Presenter
- :Timer
- :Message Builder
- :Guesser
- /Guess :Letter
- :Message

- handleTimeout() → nextGuess()
- presentGuess() → guess
- handleSelection() → addTo(Message)
- addLetter(Letter)

Letters, Words, Sentences, and commands can all be guesses

Message is responsible for handling specific Guesses by name
**Definition: Collaborate**

To work together, especially in a joint intellectual effort

This definition is collegial: Objects working together toward a common goal. Both client and service provider can be designed to assume that if any conditions or values are to be validated, they need to be done only once.

I am sending you a request at the right time with the right information

I assume that I don’t have to check to see that you have set up things properly for me to do my job.
But Can Collaborators Always Be Trusted to Behave Responsibly?

Consider collaborations between objects…

that interface to the user and the rest of the system inside your system and objects that interface to external systems in different layers or subsystems you design and objects designed by someone else
Informal Tool: Technique Trust Regions

Divide your software into regions where trusted communications occur. Objects in the same trust region communicate collegially.

Give objects at the edges responsibilities for verifying correctly formed requests.

Assign objects that have control and coordination responsibilities added responsibilities for recovering from exceptions and errors.
Implications of Trust

In a large system, distinguish whether collaborations among components can be trusted.

Identify the guarantees, obligations, and responsibilities of each component.

Use contracts to specify the details.
Trust Regions in The Telco Integration Application

Even so, well-formed requests can fail

Collaborations between an adapter and any adapted system are untrusted

trust region
Objects At The “Edges” Take On Added Responsibilities

- **Selector**: Selects the user's input.
- **Presenter**: Displays the selected guess.
- **Guess**: Performs the guessing actions.
- **Message**: Contains the message to be guessed.
- **Mailer**: Sends messages to the user.
- **Guesser**: Coordinates the guessing process.
- **Guess Dictionaries**: Stores guess letters, words, and sentences.
- **Timer**: Manages the pacing of the game.

*Debounces eye blinks*

*Collaborates with Mailer that handles exceptions*
Collaborations Among Trusted Colleagues

For collaborations among objects within the same trust region, there is little need to check on the state of things before and after each request.

If an object cannot fulfill its responsibilities and it is not designed to recover from exceptional conditions, it could raise an exception or return an error, enabling its client (or someone else in the collaboration chain) to handle the problem.
When Receiving Requests From Untrusted Sources

When receiving requests untrusted sources, you are likely check for timeliness, relevance, and correctly formed data.

But don’t design every object to collaborate defensively.

- It leads to poor performance
- Redundant checks are hard to keep consistent and lead to brittle code
When Using An Untrusted Collaborator

If a collaborator can’t be trusted, it doesn’t mean it is inherently more unreliable. It may require extra precautions to use:

- Pass along a copy of data instead of sharing it
- Check on conditions after the request completes
- Employ alternate strategies when a request fails
Control Styles and Control Center Design
Control Design

Involves decisions about

- how to control and coordinate application tasks (use case control design),
- where to place responsibilities for making domain-specific decisions (rules), and
- how to manage unusual conditions (the design of exception detection and recovery)

Goal: develop a dominant pattern for distributing the flow of control and sequencing of actions among collaborating objects
Control styles range from centralized to fully dispersed.
Characteristics of Centralized Control

Generally, one object (the controller) makes most of the important decisions. Decisions may be delegated, but most often the controller figures out what to do next. Tendencies with this strategy:

- Control logic can get overly complex
- Controllers surrounded by simple information holders and service providers
- These simple objects tend to have low-level, non-abstract interfaces

Drawback:
Changes can ripple among controlling and controlled objects
Characteristics of Delegated Control

A delegated control style passes some decision making and much of the work off one objects surrounding a control center. Each object has a more significant role to play:

- Coordinators know about fewer objects than dominating controllers
- Objects both know and do things—blends of stereotypes
- Higher-level communications between objects

Benefits:
- Changes typically localized and simpler
- Easier to divide detailed design work
Characteristics of Dispersed Control

A dispersed control style spreads decision making and action among objects who do very little, but collectively their work adds up. This can result in:

Little or no value-added by those receiving a message and merely “delegating” request to next in chain

Drawback:

Hardwired dependencies between objects in call chain
May break encapsulation
Control Center Design

A control center is a place in an application where a consistent pattern of collaboration needs to exist.

In all but the simplest application, you will have multiple control centers

Control center design is important to consider when:
- Handling user-initiated events (typically described by use cases)
- Managing complex software processes
- Designing how objects work together within a subsystem
- Controlling external devices and/or external applications under your software’s control
Control Style Development Guidelines

Don’t adopt the same control style everywhere. Develop a control style suited to each situation:
   - Adopt centralized control when you want to localize decisions in a single controller
   - Develop a delegated style when work can be assigned to specialized objects

Several styles can co-exist in a single application
   - Similar use cases often have a similar control style
   - Control styles within subsystems can vary widely

General design rule: Make analogous parts of your design be predictable and understandable by making them work in similar ways
Different Application Architectures

Blackboard

Layers

Presentation Layer
Application Layer
Domain Services Layer
Technical Services Layer

Pipes and Filters
Contracts
Responsibility-Driven Design Contracts

“The ways in which a given client can interact with a given server are described by a contract. A contract is the list of requests that a client can make of a server. Both must fulfill the contract: the client by making only those requests the contract specifies, and the server by responding appropriately to those requests. …For each such request, a set of signatures serves as the formal specification of the contract.”

—Wirfs-Brock, Wilkerson & Wiener
Finding and Preserving Contracts

A class that is viewed by all its clients identically, offers a single contract.

A class that inherits a contract should support it in its entirety. It should not cancel out any behavior.

A subclass may extend a superclass by adding new responsibilities and defining new contracts.

A class that is viewed differently by clients can offer multiple contracts. Organize responsibilities into contracts according to how they are used:

Example: Specify four BankAccount contracts
   1. Balance Adjustment
   2. Balance Inquiry
   3. Managing Challenge Data
   4. Maintaining Transaction History
Specifying Detailed Contracts

“Defining a precondition and a postcondition for a routine is a way to define a contract that binds the routine and its callers....”

—Bertrand Meyer, *Object-Oriented Software Construction*

Meyer’s contracts add even more details. They specify:

- Obligations required of the client
  - Conditions that must be true before the service will be requested
- Obligations required of the service provider
  - Conditions that must be true during and after the execution of the service
- Guarantees of service

Defined for each method or service call
Example: A Contract For A Request That Spans A Trust Boundary

<table>
<thead>
<tr>
<th>Request: Funds Transfer</th>
<th>Obligations</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client: Online banking app</td>
<td>(precondition) User has two accounts</td>
<td>Funds are transferred; balances adjusted</td>
</tr>
<tr>
<td>Service provider: backend banking system</td>
<td>(preconditions) Sufficient funds in the first account Honor requests only if both accounts active (postcondition) Both balances are adjusted</td>
<td>Only needs to check for sufficient funds and active accounts, need not check that user is authorized to access accounts</td>
</tr>
</tbody>
</table>
A Unified View of Contracts

A design can be viewed at different levels of abstraction

Responsibility-Driven Design Contract
  name and description
  list of clients and suppliers
  list of responsibilities defined by the contract
  method signatures

Meyer’s contracts add precision where we stopped:
  method signature
    • client obligations
    • supplier benefits
    • preconditions, postconditions, invariants
When To Use Contracts

Use them as a point of discussion when you are assigning responsibilities among collaborators.

But writing detailed contracts is a lot of work. Use them when you want to be formal and precise.

Detailed contracts are especially useful for defining collaborations between your software and external systems.
Designing Responsibly

Use the best tool for the job
  Tools for thinking, abstracting, modeling
  Tools for analyzing
  Tools for making your application flexible

Learn your tool set, and practice, practice, practice

The best designers never give up, they just know when to call it a day!