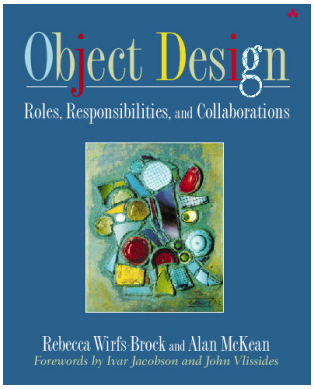
A Brief Tour of Responsibility-Driven Design

July 2006
Rebecca Wirfs-Brock
rebecca@wirfs-brock.com

What Is Responsibility-Driven Design?

A way to design software that…

- –emphasizes modeling of objects' **roles**, **responsibilities**, and **collaborations**
- –uses informal tools and techniques
- –adds responsibility concepts and thinking to any process



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

www.wirfs-brock.com for articles & presentations

The Design Process

Design is messy and iterative

Early descriptions often are imprecise

Deciding details too early can constrain your choices

Key objects and their interaction patterns have the most impact

Later descriptions add details

Different Points-of-View: Different Results

Data-Driven

Responsibility-Driven

Event-Driven

Rule-Based

Ad-Hoc

influence

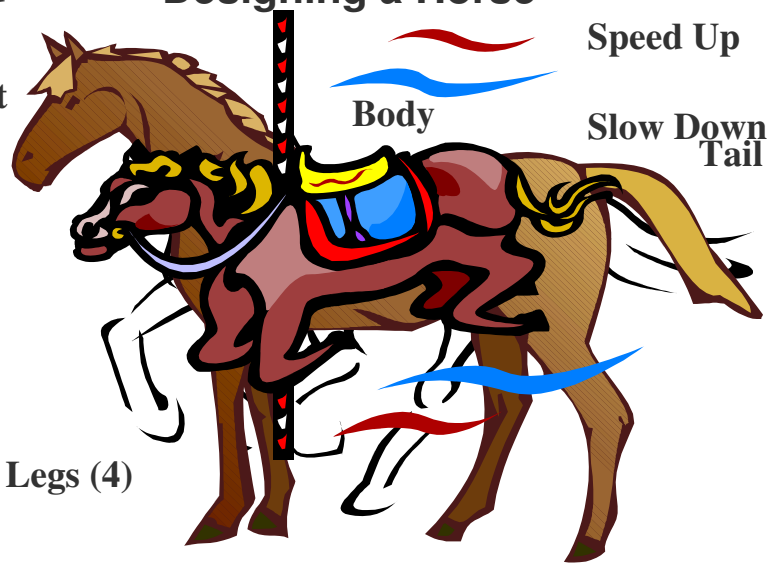
Choice of key design abstractions

Distribution of data and behavior

Patterns of collaboration

Object visibilities

Designing a Horse



Wirfs-Brock Associates www.wirfs-brock.com Copyright 2006 5

Designing a Horse Responsibly

Wirfs-Brock Associates www.wirfs-brock.com Copyright 2006 6

Responsibility-Driven Design Principles

Maximize Abstraction

- Initially hide the distinction between data and behavior. Think of objects responsibilities for “knowing”, “doing”, and “deciding”

Distribute Behavior

- Promote a delegated control architecture
- Make objects smart— have them behave intelligently, not just hold bundles of data

Preserve Flexibility

- Design objects so interior details can be readily changed

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

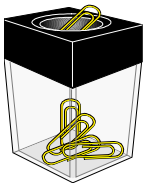
Role Stereotypes: A tool for seeing and shaping object behaviors

stereotype—A conventional, formulaic, and oversimplified conception, opinion, or image

From Responsibility-Driven Design: Object Role Stereotypes

Information holder - knows and provides information

Measurement

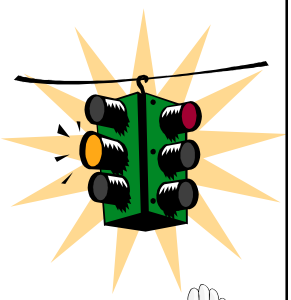


Structurer - maintains relationships between objects and information about those relationships

Sensor Repository

Object Role Stereotypes

Coordinator – mechanically reacts to events **Sensor Poller**



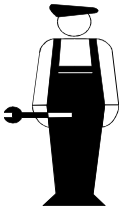
Controller - makes decisions and closely directs others' actions

Data Collector

Object Role Stereotypes

Interfacer - transforms information and requests between distinct parts of a system **Sensor**

Service provider - performs work on demand **Confidence Rater**



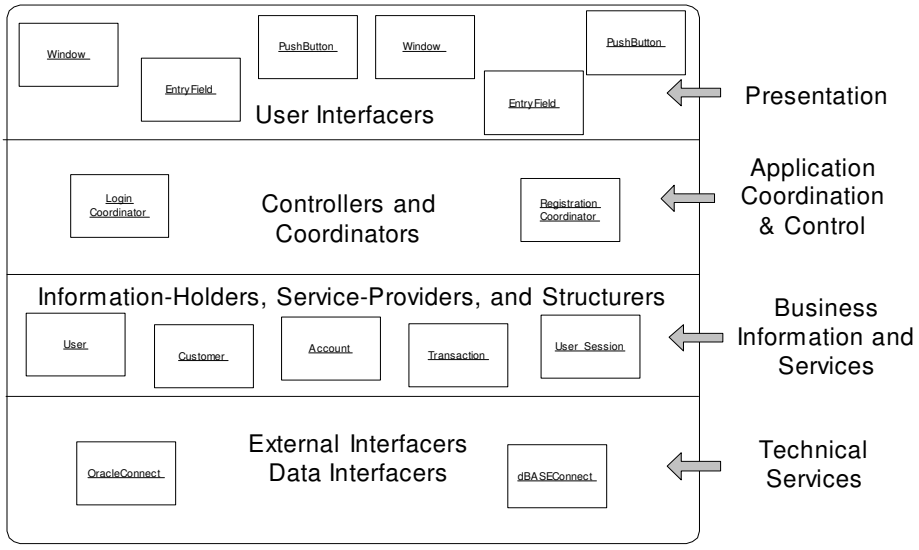
Three Uses for Object Role Stereotypes

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

Layered Architecture



Pulling up a level…to compare

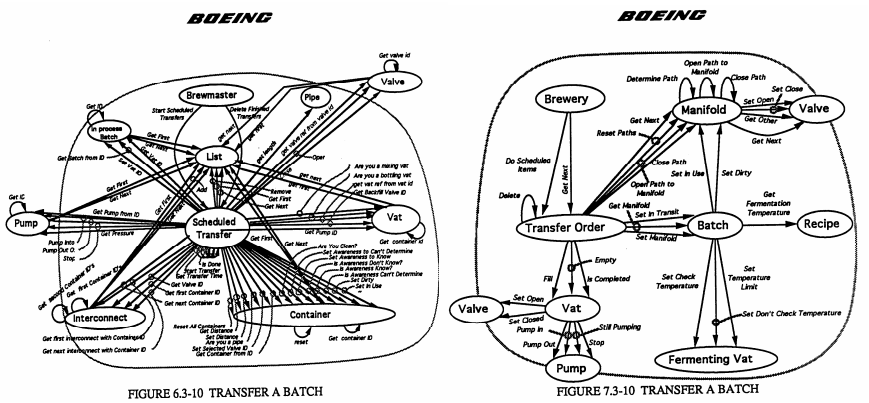


FIGURE 6.3-10 TRANSFER A BATCH

FIGURE 7.3-10 TRANSFER A BATCH

"The Object-Oriented Brewery: A Comparison of Two Object-Oriented Methods,"R. Sharble and S. Cohen, Boeing Technical Report BCS-G4059, 1992.

"How Designs Differ", R. Wirfs-Brock, Smalltalk Report, vol. 1, no. 4

…and characterize

Data-Driven Design Approach	Responsibility-Driven Design Approach
centralized control	delegated control
controllers	coordinators
inherited attributes	inherited behavior
many low-level messages	fewer, higher-level messages
lots of simplistic information holders	a few smart objects that blend role stereotypes

n-tier web applications

Layer	Functionality	Role	Technique
Client	User Interface	**Interfacer**	HTML, JavaScript
Presentation	Page Layout	**Interfacer**	JSP
Control	Command	**Coordinator**	Servlet
Business Logic	Business Delegate	**Controller**	POJO, Session EJB
Data Access	Domain Model	**Information Holder, Structurer**	JavaBean, Entity EJB
Resources	Database, Enterprise Services	**Service Provider**	RDBMS, Queues, Enterprise Service Bus

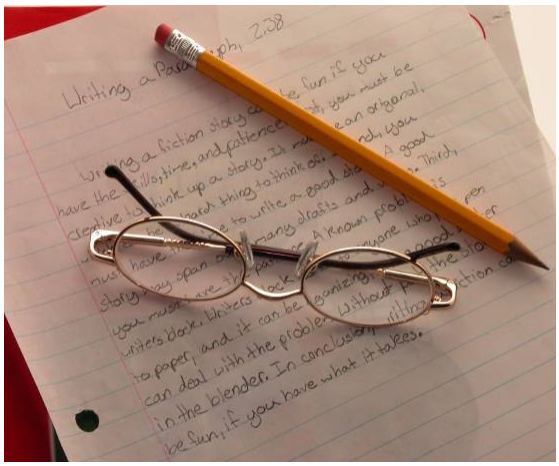
Exploratory Design

Characteristic: Formative

Goal: Produce object and interaction models

Results: Class descriptions, object collaborations

A Designer's Story: A tool for seeing what's important



Designer's story—a quickly written paragraph or two description of important ideas, what you know, and what you need to discover

Elements of a story…

What is your design supposed to do?

Is there something similar you can draw upon or emulate?

What will make it a success?

What are the most challenging parts?

Why tell a designer's story?

To put your spin on what's important

Describing the problem helps you own it

Sharing them builds understanding and a common vision

Metaphors are hard to come by…identifying themes and key responsibilities from designer stories is one alternative

Identify Story Themes

Themes are key areas of system activity and design focus

- Online Banking themes
  - modeling online banking activities
  - representing common banking functions
  - configuring system behavior
  - accessing scarce resources

They can be broad or narrow

Finding Candidates

Guidelines for Inventing Objects

An object should capture one key abstraction

Choose meaningful names

Distinguish objects by behavior differences

Fit objects into their design context

Where Do We Find Objects?

Objects that support

- System behaviors
- Architecture
- Performance requirements
- Software mechanisms and machinery

Look for

- Key concepts in the domain
- Things that represent the software's view of things outside the software in the "real world"

Objects from Key Concepts—Domain Objects

Familiar concepts to someone who knows about the kind of problem your application is solving

- In the domain of banking: account, funds, currency, financial transaction
- In the domain of railroad shipping: consist, rail yard, shipping route

The Whole Value* Pattern

Classes that represent meaningful quantities in your domain

Examples: currency, calendar periods, temperature, color, weight, brightness.

Windspeed (NNW at 20 kph)

Temperature (75 degrees Fahrenheit)

Lightreading (1000 lumens)

The name whole value means object do not have an identity of importance

*Described by Ward Cunningham in
The CHECKS Pattern Language of Information Integrity,
pages 145-156 in Pattern Languages of Program Design, volume 1
see http://c2.com/ppr/checks.html#1

Domain Objects: Entity* or Value?

Entity object—An object distinguished by *who* it is

Entities have life cycles and can change form and content, but the thread of continuity must be maintained. "You are who you are and you are unique."

Value object—An object that needn't be unique (others can share a reference to it)

It typically describes some characteristic. "I don't care which blue crayon I use, just that I have one."

*Described by Eric Evans in
Domain Driven Design, chapter on Entities

What To Look For

Look for inventions that represent:

- The ***work*** your software performs
- The ***things*** your software affects or is connected to
- The ***information*** that flows through your software
- Your software's ***decision-making***, ***control*** and ***coordination*** activities
- Ways to ***structure*** and ***manage*** groups of objects
- ***Representations*** of real world things your software needs to know something about

Characterizing Your Candidates

CRC Cards: An informal design tool

*C*andidate, *R*esponsibilities, *C*ollaborators

Sensor

Purpose: Represents what the Arbor 2000 system knows about devices that reports data that is physically sensed from the environment. Sensors can report light intensity, temperature, wind speed and direction, rainfall and other physical readings. Some kinds of sensors can sense multiple physical characteristics and are capable of reporting readings at different intervals (such as every minute, hourly, weekly, monthly) or based on a significant event (temperature rising x degrees in a period of time, x amount of rainfall, etc.).

Stereotypes: Service Provider, Interfacer

Explaining Purpose

A candidate does and knows certain things. Briefly, say what those things are. A pattern to follow:

> An object is or represents a *thing* that *knows or does certain things*. And then mention one or two interesting facts about the object, perhaps a detail about what it does or who it works with.

Purpose Matches Stereotype

A service provider does specific work. The type of work it does is important to describe:

> A compiler is a program that translates source code into machine language.
>
> A RazzmaFrazzer is a converter that accurately and speedily translates Razzmas into Frazzes. As it translates, it logs statistics on how accurate the translation is and whether any information is lost.

Role Stereotypes

Doing, Knowing and Deciding

Stereotypes are simplified views that help you characterize the roles objects play in an application

- **Service providers** do things
- **Interfacers** translate requests and convert from one level of abstraction to another
- **Information holders** know things
- **Controllers** direct activities
- **Coordinators** delegate work
- **Structurers** manage object relations or organize large numbers of similar objects

CRC Cards: An informal design tool

Candidate, Responsibilities, Collaborators

Sensor	
knows physical characteristics it can detect	Parser
knows reporting interval	Measurement
maintains configuration parameters	
knows sensor make and model	
knows location	
knows if activated	
creates measurements from reports	

Collaborators

Responsibilities

Naming Candidates

Naming Candidates

Fit a name into some naming scheme
Calendar→GregorianCalendar→JulianCalendar?
ChineseCalendar?

Give service providers "worker" names
Service providers are "workers", "doers", "movers" and "shakers": StringTokenizer, ClassLoader, and Authenticator

Choose a name that suits a role
Objects named "Manager" organize and pool collections of similar objects: AccountManager organizes Account objects

Naming Candidates

Choose names that expand an object's behavior
AccountRecord?—facts set down in writing
Account?—sounds livelier—an object that makes informed decisions on the information it represents

Choose a name that lasts a lifetime
A ninety-year old named "Junior"?
ApplicationInitializer or ApplicationCoordinator?

Include facts most relevant to its users
TimerAccurateWithinPlusOrMinusTwoMilliseconds?
or simply Timer?

Eliminate naming conflicts by adding description
Rename Properties to TransactionHistoryProperties

Refining Candidates

Are You Looking For Objects, Roles, or Classes?

Candidates represent important, vivid concepts, machinery and mechanisms

You can think concretely, identifying concrete classes that represent things that perform some work in your application

..or more abstractly, identify abstractions that stand in for many different variations on a theme

When finding Common Roles

Blur distinctions — Identify categories. Let go of the little details that make objects different

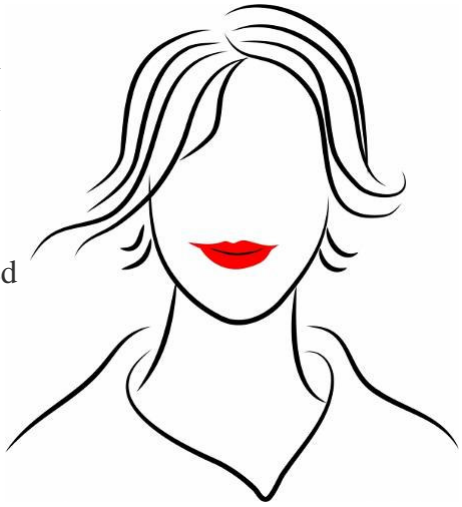
Explain Key Abstractions

If a role can be played by several different classes of objects, explain both the general characteristics and mention something about the others that will play this role:

> *An AccountingService represents a single accounting transaction performed by our online banking application. Successful transactions result in updates to or queries to a customer's accounts. Specific AccountingServices communicate with the banking systems to perform the actual work. Examples of AccountingServices are FundsTransferService, MakePaymentService, and ViewAccountBalanceService.*

Powerful abstractions simplify and give your design economy of expression

Reusable roles can be identified and shared among different classes



Keep a Candidate When You Can…

Name it

Define its purpose

Stereotype it

See it supports a particular use case

See that it is an important architectural element

Assign it one or two initial responsibilities

Understand how others view it

See how it behaves differently

Finding and Assigning Responsibilities

Where Do We Find Responsibilities?

By looking at various descriptions of system behavior and then modeling how a community of objects work:

- Object role stereotypes and purpose statements
- Use cases
- Gaps in these descriptions
- Other requirements, themes and stories
- Following "what if…then…and how" chains
- Relationships and dependencies between candidates
- Candidates' "life events"
- Technical aspects of a specific software environment
- A design perspective on how things should work

The generative power of role stereotypes

Pushing on an object's character leads to initial responsibilities

- Ask of a service provider, "what requests should it handle?" Turn around and state these as responsibilities for "doing" or "performing" specific services
- Ask what duties does an interfacer have for translating information and requests from one part of the system to another (and translating between different levels of abstraction)?
- What events does a controller handle and who does it direct?

How Do You State Responsibilities?

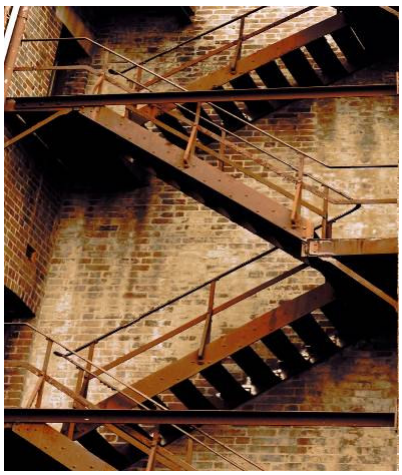
A single responsibility is larger than an operation or attribute:

- Example: A DataCollector wraps operating system resources, such as sockets or data streams, retrieves raw data and converts it (packages it) into one or more data records from a sensing device.
- Responsibilities: Receive raw data from a sensor or sensor group
- Chunks data into individual readings

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

Technique: Seeing at different abstract levels



We can see objects and behavior at different levels:

- At the *conceptual* level- a set of responsibilities
- At the *specification* level- set of methods that can be invoked
- At the *implementation* level- code and data

Technique: Pull of a Level
Reverse engineer a class into responsibilities

The Java Calendar class

Internally, Calendar keeps track of a point in time in two ways. First, a "raw" value is maintained, which is simply a count of milliseconds since midnight, January 1, 1970 GMT, or, in other words, a Date object. Second, the calendar keeps track of a number of fields, which are the values that are specific to the Calendar type. These are values such as day of the week, day of the month, and month. The raw millisecond value can be calculated from the field values, or vice versa.

Calendar also defines a number of symbolic constants. They represent either fields or values. For example, MONTH is a field constant. It can be passed to get() and set() to retrieve and adjust the month. AUGUST, on the other hand, represents a particular month value. Calling get(Calendar.MONTH) could return Calendar.AUGUST.

Calendar Methods

public int getFirstDayOfWeek()

This method returns the day that is considered the beginning of the week for this Calendar. This value is determined by the Locale of this Calendar. For example, the first day of the week in the United States is Sunday, while in France it is Monday.

public abstract int getGreatestMinimum(int field)

This method returns the highest minimum value for the given time field, if the field has a range of minimum values. If the field does not have a range of minimum values, this method is equivalent to getMinimum().

public abstract int getLeastMaximum(int field)

This method returns the lowest maximum value for the given time field, if the field has a range of maximum values. If the field does not have a range of maximum values, this method is equivalent to getMaximum(). For example, for a GregorianCalendar, the lowest maximum value of DATE_OF_MONTH is 28.

public abstract int getMaximum(int field)

This method returns the maximum value for the given time field. For example, for a GregorianCalendar, the maximum value of DATE_OF_MONTH is 31.

public final void set(int year, int month, int date)

This method sets the values of the year, month, and day-of-the-month fields of this Calendar.

public final void set(int year, int month, int date, int hour, int minute) This method sets the values of the year, month, day-of-the-month, hour, and minute fields of this Calendar.

public final void set(int year, int month, int date, int hour, int minute, int second)

This method sets the values of the year, month, day-of-the-month, hour, minute, and second fields of this Calendar.

public void setFirstDayofWeek(int value)

This method sets the day that is considered the beginning of the week for this Calendar. This value should be determined by the Locale of this Calendar. For example, the first day of the week in the United States is Sunday; in France it's Monday.

public void setLenient(boolean lenient)

This method sets the leniency of this Calendar. A value of false specifies that the Calendar throws exceptions when questionable data is passed to it, while a value of true indicates that the Calendar makes its best guess to interpret questionable data. For example, if the Calendar is being lenient, a date such as March 135, 1997 is interpreted as the 135th day after March 1, 1997.

public void setMinimalDaysInFirstWeek(int value)

This method sets the minimum number of days in the first week of the year. For example, a value of 7 indicates the first week of the year must be a full week, while a value of 1 indicates the first week of the year can contain a single day. This value should be determined by the Locale of this Calendar.

public final void setTime(Date date)

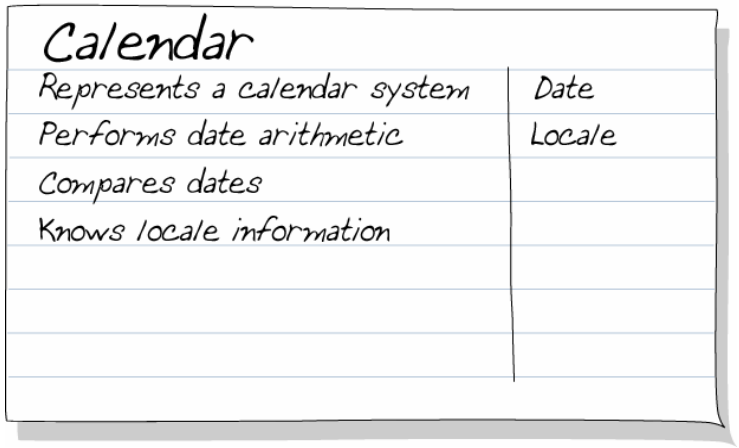
This method sets the point in time that is represented by this

Calendar.

public void setTimeZone(TimeZone value)

This method is used to set the time zone of this Calendar.

To get a general picture: Calendar revealed



What's Missing From Use Cases

Use cases are descriptive, not prescriptive

There is a gap between these descriptions and a design

Use cases rarely describe aspects of

- Control and coordination
- Error recovery
- Visual display
- Timing and synchronization

Deriving Responsibilities from Use Case Descriptions

We bridge this gap by:

- Identifying things our software does and information it needs
- Restating these as responsibilities
- Breaking down large statements into smaller parts
- Inventing control and coordination mechanisms
- Designing exceptions and exception recovery

Gathering Responsibilities from Use Cases

What you find depends on the level of use case details

Any responsibility will have to be transformed into statements of individual objects' responsibilities

Responsibilities may be broad statements (which need to be decomposed)

Use Case: Report Sensor Reading

Trigger: A reporting interval has elapsed, or a sensor has been polled

1. Physical sensor transmits report, which includes: one or more raw data values, a timestamp, sensor identification.

2. System verifies physical sensor is known to system

3. System converts raw sensor data report to measurements (which include sensor id, normalized reading value, timestamp, location, confidence rating)

4. System verifies that measurements are within prescribed manufacturer's ranges

5. System compares measurement against recent historical values and assigns each measurement a confidence rating.

6. System stores measurements

Use Case: Report Sensor Reading

Trigger: A specified interval has elapsed, or the sensor has been polled (two ways to start…different set up for each case…but either way should stimulate the same processing)

1. Physical sensor transmits report, which includes:one or more raw data values, a timestamp, sensor identification. (some controlling object will have to receive the raw data and then start the action)

2. System verifies physical sensor is known to system (need to keep track of known sensors—a structurer/repository?)

3. System converts raw sensor data report to measurements (which include sensor id, normalized reading value, timestamp, location, confidence rating) (data will then have to be converted based on sensor type and manufacturer's characteristics)

Use Case: Report Sensor Reading

4. System verifies that measurements are within prescribed manufacturer's ranges (some object will have to verify and another will have to hold onto manufacturer's values…are these two different objects or?)

5. System compares measurement against recent historical values and assigns each measurement a confidence rating.(recent values—are they stored / cached, or both? Assuming confidence thresholds can vary based on measurement type, we'll need different ways to determine confidence based on measurement type)

6. System stores measurements (some interface to an external data base is implied)

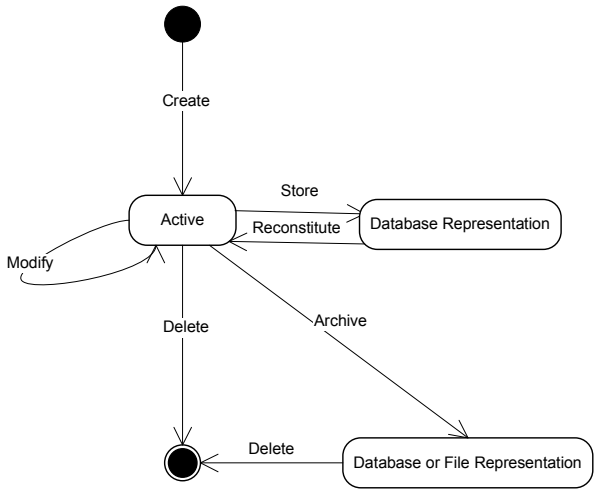
Responsibilities from Objects' "life events"

Some objects' responsibilities are largely shaped by how they react to specific events

Most of the work of a controller or coordinator is in response to events that they interpret

When an object is born and when it leaves the scene are common places to find responsibilities for gracefully entering and leaving the scene

A Typical Life Cycle of a Domain Entity



Be Opportunistic!

Assigning one responsibility leads you to think… what client responsibilities will use it … and how it will be accomplished (subresponsibilities assigned to other objects in the "neighborhood")

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

Collaboration Design

What Is A Collaboration Model?

How a group of objects work together to fulfill a specific task

It includes a description of objects, what each does, and how they interact

We use CRC cards to record each object's responsibilities and collaborations and sequence diagrams to show interactions

Early Collaboration Modeling

Concentrate on control, coordination, and important services

- Don't over specify a collaboration
- Stop designing collaborations when you can show that your small set of objects fulfills its purpose

Focus on objects you invent, not objects used from a library

- Ignore GUI details – Treat the buttons and list selections and entering data as the source of events (e.g. user indicates "Predict Fire Danger Rating" not "user clicks on button")

Running a Modeling Session

Ask which object receives the event? Then what happens?

Stay at the same (or +-1) conceptual level

- If you are exploring how to handle a Sensor reporting data, don't dive into the details of the database

Follow the logic closely, think critically

- Are things being done in the right order? Does validating the data of a sensor reading happen before or after a Measurement is created?
- Do objects really know enough to perform the responsibility you are asking them to?
- Are you considering boundary cases?

Start with rough sketches…

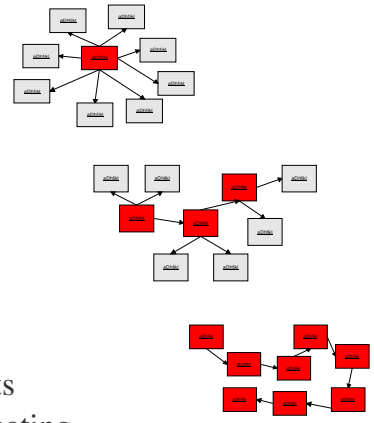
then get more precise…

by…

Showing a sequence of messages

- Label message arrows with request names
- Show arguments passed along with requests
- Show return values for important (unobvious) information returned
- Illustrate creation of key objects

Control centers and collaboration styles: Tools for shaping solutions



control center—a place where objects charged with controlling and coordinating reside

Control Centers

Deciding on and developing a consistent control style is one of the most important design decisions you can make. Not all centers the same style

- Handling web interactions
- Managing complex software processes
- Objects working together within a subsystem
- Control of external devices or external applications

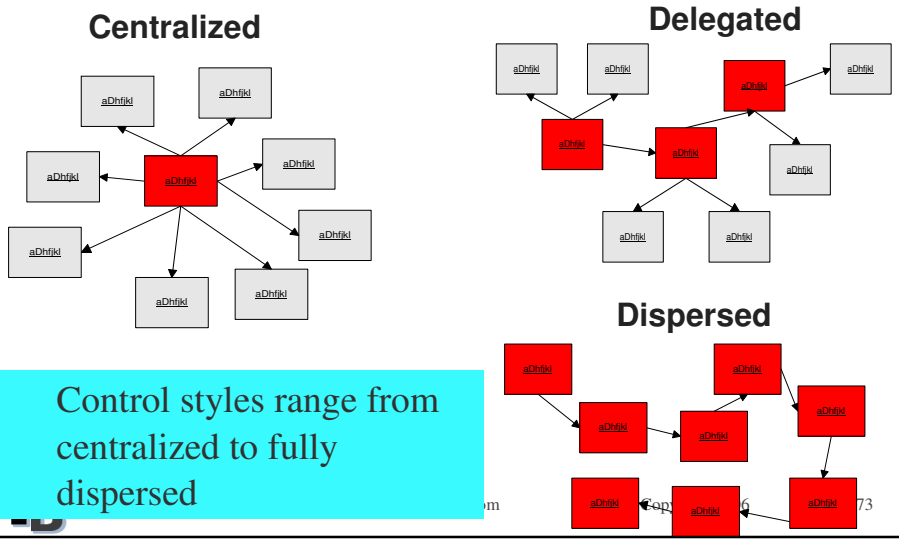
Control Design

Involves decisions about

- how to control and coordinate tasks,
- 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 patterns for distributing the flow of control and sequencing of actions among collaborating objects. Make similar parts of your system be consistent

Collaboration Styles



Control styles range from centralized to fully dispersed

Centralized Control

Generally, one object (the controller) makes most of the important decisions. Tendencies with this strategy:

- Control logic can get overly complex
- Controllers can become dependent upon information holders' contents
- Objects can become coupled indirectly through the actions of their controller
- The only interesting work is done in the controller

Drawback:
Changes can ripple among controlling and controlled objects

Delegated Control

Some decision making and much of the action passed off to objects surrounding a control center. Neighbors have significant roles:

- Coordinators tend to know about fewer objects than dominating controllers
- Messages between collaborators are higher-level

Benefits:

- Changes typically localized and simpler
- Easier to divide interesting design work among a team

5

Dispersed Control

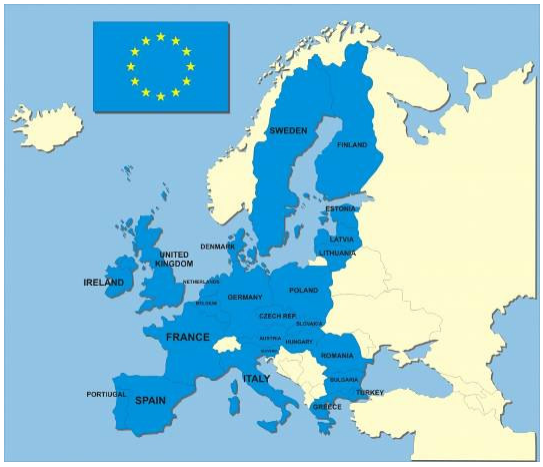
Spreads decision making and action among objects who individually do little, but collectively their work adds up. This can result in:

- Long message chains to dig information out of information holders
- Little or no value-added by those receiving a request

Drawback:

- Hardwired dependencies between objects in call chain
- May break encapsulation

Trust Regions: A tool for seeing where "defensive" behavior is or isn't needed

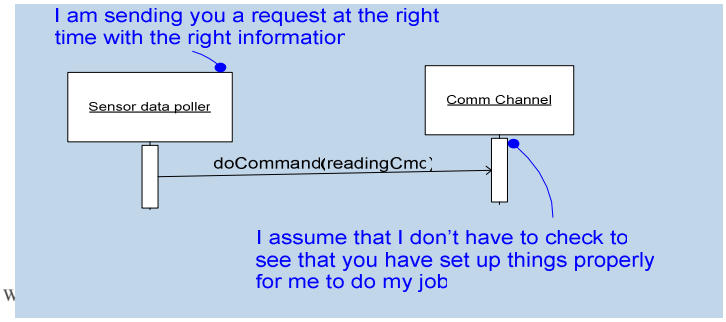


trust region—an area where trusted collaborations occur

Definition: Collaborate

To work together, especially in a joint intellectual effort

Objects or components working together toward a common goal



Definition: Collaborate

2. To cooperate treasonably, as with an enemy occupation force

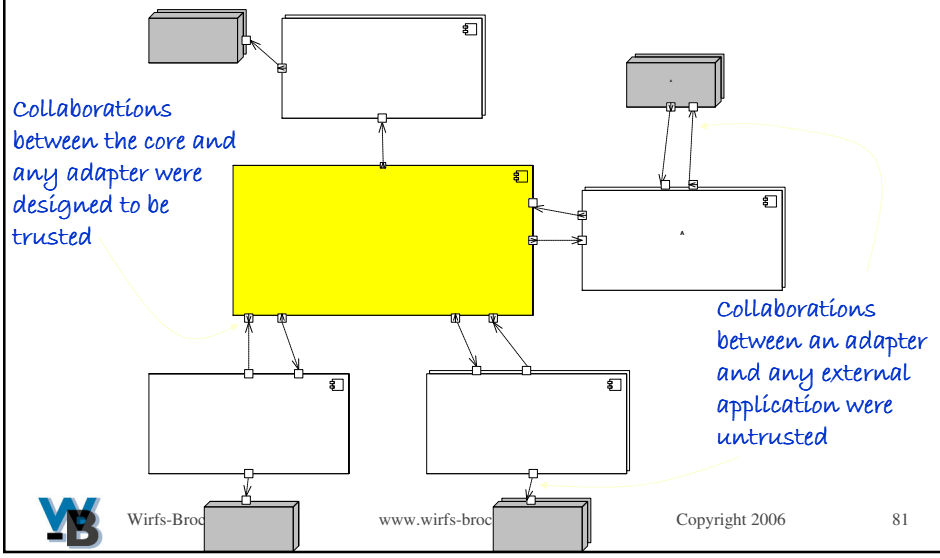
Implications of trust

Objects at the "borders" may take on extra responsibilities

Within a trust region, collaborations can be more collegial

- Requests can be assumed to be at the right time and contain the right information
- Objects deep inside a trust region can be designed to not check for well-formed or timely requests

Trust In A Telco Integration Application



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:

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

Collaboration Cases To Consider

Collaborations between objects…

- that interface to the user and the rest of the system
- in different layers or subsystems
- inside your system and objects that interface to external systems
- you design and objects designed by someone else

Requests From Untrusted Sources

The receiver is likely to check for timeliness, relevance, and correctly formed data

There are degrees of trust

Don't design every object to behave defensively

- Redundant checks are hard to keep consistent and lead to brittle code
- It leads to poor performance

Recovering From Exceptions

Reasons To Think About Exceptions Early, Often, Sooner *And* Later

Usability may be affected

> Consider software that enables a severely disabled user to construct messages and communicate with others. Shouting "stack overflow!" or "network unavailable!" isn't acceptable

The degree to which a user can or should be involved in exception handling has profound design implications

Solutions may not be obvious or "easy". Experimentation may be required

The Mismatch Between Use Case And Program Execution

A single use case step can result in thousands of requests between collaborating objects, any number of which could cause numerous object exceptions

There isn't a direct correspondence between use case and program exceptions

Exceptions for Report Sensor Reading

Sensor is unknown- Store data in "raw" form for potentially later processing

Raw data improperly formed (bad packet)- Log error

Measurement value out of expected range- Log error and do not store "suspect" measurement, signal possible physical sensor fault

Historical data not available- Assign measurement low confidence

Measurement value exceeds threshold for "expected value"- Assign low confidence and signal "abnormal change in reading" event

Database unavailable- Attempt recovery, then signal database failure

A Strategy For Handling Exceptions For A Key Collaboration

- Brainstorm most likely exception cases. Name and describe them first
- Then address how to resolve easy-to-recover from cases first
- Explore alternatives for tougher ones. Test for usability and feasibility

Handle Exceptions as Close to The Problem as You Can

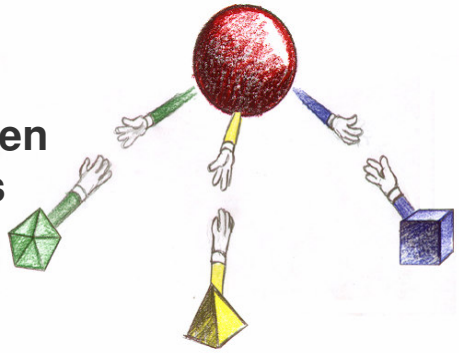
There are many different ways to "handle" an exception. It could be logged and rethrown (possibly more than once), until some object takes corrective action

Who naturally might handle exceptions?

- External interfacers often take responsibility for handling faulty conditions in other systems
- The initial requestor
- As a fallback, pass the buck to some object who takes responsibility for controlling the action

Contracts

Formal Tool: 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 Sensor contracts

1. Manage physical characteristics
2. Manage settings
3. Conversion of raw data into measurements
4. Know manufacture info

Technique: 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

Request: Get	Obligations	Benefits
Client: Sensor poller	(precondition) Keeps track of valid polling intervals and sensor polling requests	Data request honored/poller doesn't have to know comm details
Service provider: Communications Channel	(preconditions) Sensor assigned to comm channel Sensor is active (postcondition) Returns sensor data	Only needs to poll active devices, no error recovery required

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!

Exercise: Write a Designer's Story

Briefly read the problem description (pages 35-7 of the handout)

Spend 10 minutes writing a design story that identifies the particular challenges of the Data Collection Problem

Share your story with someone seated near you

(As a group, we will then identify some key themes)

Exercise: Identify Some Candidates

In a 5 minute brainstorm, come up with a list candidates:

- What work needs to be done? (Controllers, Coordinators, Service Providers)
- What information flows around the software system? (Information holders, Structurers)
- What needs to be structured and managed? (Structurers)
- What real world things does the software need to be aware of?(Information holders, blends)
- How does it connect to other systems and external devices? (Interfacers)

Exercise: Define a Candidate

Now, pick one candidate stereotype it and write a brief statement of purpose on the unlined side of a CRC card

Choose whether your candidate is an information holder, structurer, controller, coordinator, interfacer or service provider. You may list more than one stereotype if you think your candidate might be a "blend"

Use Case: Report Sensor Reading
Actor: Physical Sensing Device

Context: Either the specified reporting interval has elapsed or the sensor is being asked for its current reading…

1. Sensor transmits report, which includes:
one or more raw data values, a timestamp, sensor identification.
2. System verifies physical sensor is known to system
3. System converts raw sensor data packets to measurements.
4. System verifies that measurements are within prescribed manufacturer's ranges
5. System compares measurement against recent historical values and assigns each measurement a confidence rating.
6. System stores measurements

Exceptions:
2. Sensor is unknown- Store data in "raw" form for potentially later processing

Post condition: One or more measurements have been stored

Exercise: Identify Some Responsibilities

Can you identify an object with control and coordination responsibilities for this use case? Give it a name and list some of its responsibilities

There may also be an object that represents what our system knows about the physical sensors installed in the field. What responsibilities would you give this object?

Define some responsibilities of a Measurement object? What stereotype is it?

Tutorial Notes for A Tour of Responsibility-Driven Design

The design approach known as Responsibility-Driven Design is a way of designing complex software systems using objects and component technology. Responsibility-Driven Design was conceived in 1990 as a shift from thinking about objects as data + algorithms, to thinking about objects as roles + responsibilities. The principles behind Responsibility-Driven Design were first described in an OOPSLA paper in 1989, and the book, *Designing Object-Oriented Software*. If you are interested in reading more about Responsibility-Driven Design, you can find several introductory articles about it on www.wirfs-brock.com, or read about the latest thinking tools and techniques in *Object Design: Roles, Responsibilities, and Collaborators* by Rebecca Wirfs-Brock and Alan McKean.

Responsibility-driven design draws upon the experiences of a number of very successful and productive Smalltalk designers. The original concepts and motivation behind responsibility-driven design were formulated when several of us developed and taught a course on object-oriented design to Tektronix engineers in the late 1980s. These engineers were working on object-oriented projects that would be implemented in Smalltalk, C++ and other, non-object-oriented programming languages. Although we refined our initial thoughts and added several techniques to our design toolkit, but the underlying values remain. Responsibility-Driven Design emphasizes practical techniques and thinking tools.

In a responsibility-based model, objects play specific roles and occupy well-known positions in the application architecture. It is a smoothly-running community of objects. Each object is accountable for a specific portion of the work. Objects collaborate in clearly-defined ways, contracting with each other to fulfill the larger goals of the application. By creating such a "community of objects," assigning specific responsibilities to each, you build a collaborative model of your application.

Responsibility-Driven Design emphasizes that objects are more than simple bundles of logic and data ... they are service-providers, information-holders, structurers, coordinators, controllers, and interfacers to the outside world. Each must know and do its part. Thinking in terms of these object role stereotypes enables you to conceive of how an object should be designed to fit in and play its part. Role stereotypes, from Responsibility-Driven Design are a fundamental way of seeing objects' responsibilities. Think of them as "purposeful oversimplifications" that help

designers identify the gist of what an object should accomplish. Early on, designers use stereotypes to characterize their initial candidate objects. Later, use stereotyping to take a broader look at implemented code and discern the kind of roles various objects fulfill.

Progression of Ideas	1990	1995	2005
Specifications	Assumed pre-existing	Create or restructure into forms that guide design	Can fit into agile and more traditional development practices. Problem framing helps ask the right questions
Finding objects	Naïve-nouns	Found in analysis, design and/or concept formation	Involves modeling concepts and invention. Role stereotypes and domain objects, finding roles that may map to one or more classes
Refinement	Inheritance, streamlining communications	1990+ composition & configurable algorithms	+ hotspots and designing to support planned variations
Guidelines	General	1990+ Values and tradeoffs, control style	Assigning responsibilities according to role stereotype, keeping objects with a narrow focus
Architecture	Largely ignored, subsystems given light treatment	4 layer application architecture, interfacers to model non-oo services	+Control Centers, Trust regions. Recovery strategies and exception design
Values	Behavior is good (implying data focus is bad)	Keep abstractions at high level Focus on behaviors	Develop consistent patterns of collaboration Hide details inside objects

Table 1: Progression of Responsibility-Driven Design Ideas

Responsibilities describe what software must do to accomplish its purpose. Design work progresses from requirements definition through roughly sketched ideas and then on to more detailed descriptions and software models. At the beginning, we focus on describing our system by capturing the viewpoints of many different stakeholders. We need to consider multiple perspectives in our solutions. Responsibility-Driven Design is a clarification process. We move from initial requirements to initial descriptions and models; from initial descriptions to more detailed descriptions and models of objects; from candidate object models to detailed models of their responsibilities and patterns of collaboration.

We wish to be very clear on one point: although we present object oriented development activities in a linear fashion, this is rarely how design proceeds in practice. Software design processes are highly fluid and opportunistic, even though the final results are firmly fixed in code.

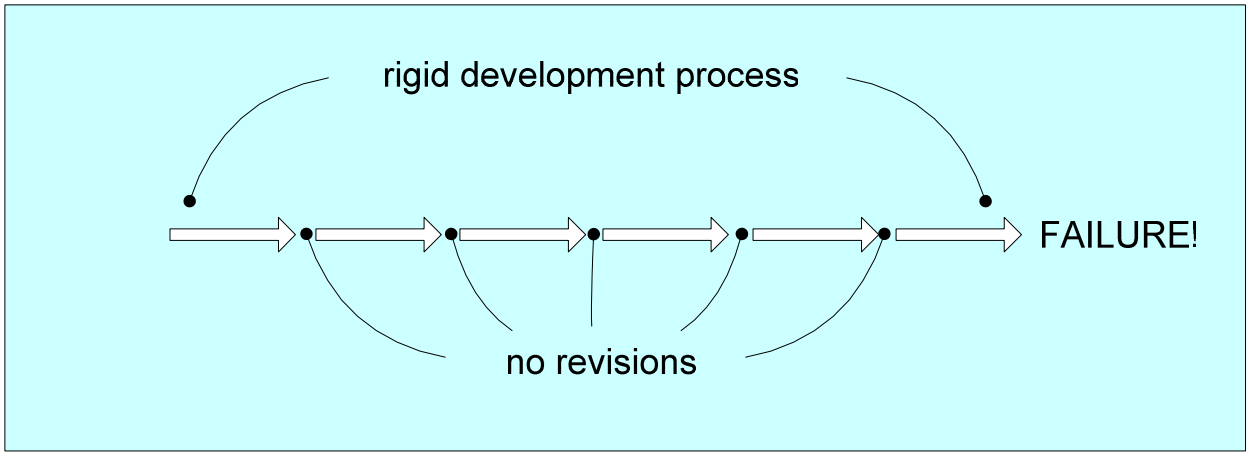


Figure 1 Rigid, tightly planned development

A design journey is filled with curves, switchbacks, and side excursions. When tracking down design solutions, you often switch among different design activities as you discover different aspects of the problem. Be opportunistic. Use a variety of tools that help gain perspective, discover information, and craft solutions. Design is fluid and malleable.

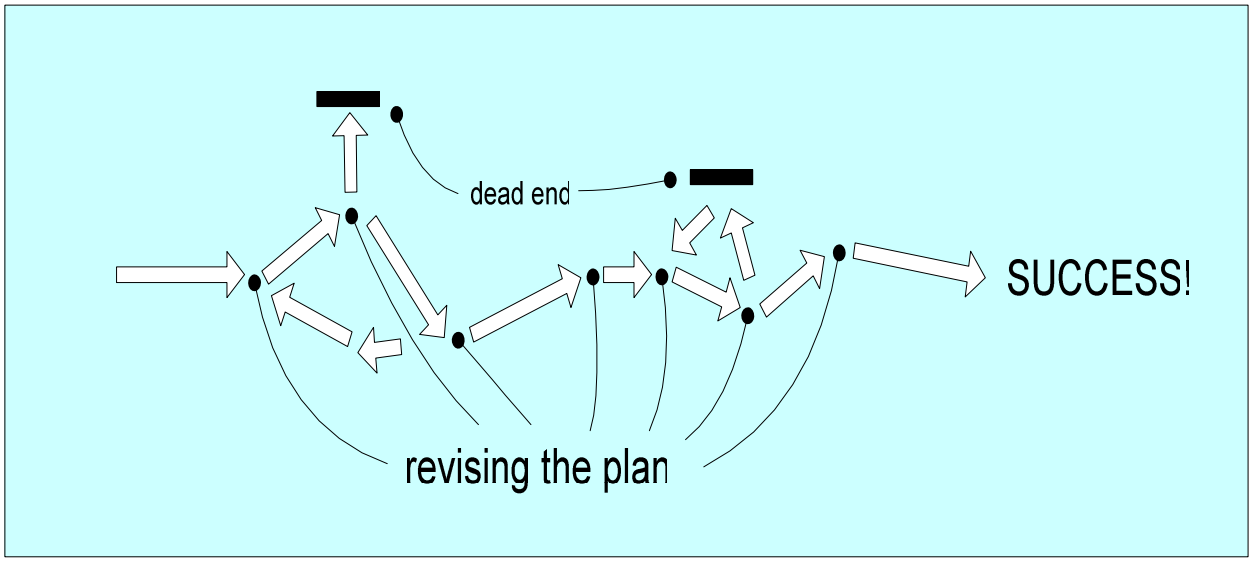


Figure 2 The Responsibility-Driven path is a flexible one

Ordering of activities and focus will, of necessity, change. Planning, adding new features, setting goals, characterizing the application via a prototype, creating an object model, identifying the hard problems—these are only some tasks. These tasks vary in their purpose, rigor, scope, emphasis, context, and applicable tools.

With all but the simplest software, you can't fathom what lies ahead. With so much complexity, you won't always make optimal decisions. Progress isn't always steady. Along the way you discover new information and constraints. Take time to breathe and smooth out these recurring wrinkles.

To address your lack of 20-20 foresight, plan pauses to reexamine, adjust, and align your work to a changing set of conditions. This allows you to incorporate your growing understanding into what is built. Bear in mind that design is iterative and incremental. As designers, we naturally think that software objects are the center of the software universe. However object-oriented we may be, though, many other participants and perspectives go into the conception, design, and construction of a successful application. Just like a theater production, software development involves much more than meets the eye during a performance. And although objects may take center stage for our work, it is important to recognize the impact that different perspectives and activities have on design.

We break the object design process into two major phases: creating an initial design (exploratory work) and then crafting more comprehensive solutions (refinement)

Activity	Results
Associate domain objects with execution-oriented ones.	**A CRC model of objects, roles, responsibilities, and collaborators**
Assign responsibilities to objects.	**Sequence or collaboration diagrams**
Develop initial collaboration model.	**Descriptions of subsystem responsibilities and collaborations**
	Preliminary class definitions
	Working prototypes

Table 2: Exploratory Design

At some point after you've developed an initial exploratory design, you want to break away from designing and start coding. This could occur after a relatively short while, especially if your design is straightforward or if you are doing XP (Extreme Programming) where design-test-implement cycles are tightly integrated. Or, perhaps you want to prove part of your design by implementing a prototype before investing energy designing other parts that rely on that proof of concept being solid. Whether you take the time to polish your design a bit before coding or plan on adjusting your design during implementation, your initial design ideas will change. Most applications are too complex to "design right" the first time. Creating a workable design means revisiting initial assumptions to make sure that your design lives up to stakeholders' expectations. It may also mean spending extra time to design a flexible solution.

Activity	Results
Justify tradeoffs	**Documentation of design decisions**
Distribute application control	**Control styles identified** **Easy-to-understand patters of decision making and delegation in the object model**
Revise model to make it more maintainable, flexible and consistent	**Creation of new object abstractions** **Revision of object roles, including stereotype blends** **Simplified, consistent interfaces and patterns of collaboration** **Specification of classes that realize roles** **Application of design patterns**
Document the design clearly	**UML diagrams describing packages, components, subsystems, classes, interaction sequences, collaborations, interfaces** **Code**
Formalize the design	**Contracts between system components and key classes**

Table 3: Design Refinement

Tools and Techniques

Responsibility-Driven Design offers techniques for honing your thinking about how to divvy an application's responsibilities into objects and coordinating their performance. Following are summaries of techniques that you may want to add to your designer's toolkit.

Tool: A Designer's Story

A **designer's story** is a way to put your own spin on the system you are work on and a substitute for Extreme Programming's elusive metaphor. Early on in any project I now write a design story. Originally, I used a design story as a private way to organize my thoughts. Lately, I've been encouraging teams to individually write design stories and then share them at the beginning of a project. This has been a good way to voice individual visions that can complement and be melded into a shared perspective. And it's a good ice-breaker for newly formed teams or in situations where some voices dominate and others' voices don't get heard.

Here are four reasons to write a design story:

- To restate any requirements from your design perspective
- To put your spin on what's important or hard or easy or similar to what you've done before
- Boiling it down helps you grasp the problem
- To own the problem

Sharing your design stories with you teammates allows you to:

- Have a voice
- Get others' perspectives
- Develop collective thoughts
- Build mutual understanding

Technique: Write a designer's story. The technique is very simple and even those who only want to write code can bang out something if they know it will be short, sweet, to the point, and only take 15 minutes. I tend to pump out lots of words when I am put in front of a word processor. Perhaps too many. So I now prefer to write my stories by hand, especially when I intend to share them with others. This makes it more personal and it looks rough and less polished which is a good thing.

Here are the basic techniques in a nutshell. Quickly write a rough story—two paragraphs or less is ideal. Write about your application's essential characteristics: the themes. Talk about important ideas such as:

- What is your application supposed to do? How will it support its users? Is it connected to a real world example that you can study or emulate? Is it similar to what you've done before?
- What will make your application a success? What are the most challenging things to design?

Tell what you know and what you need to discover.

> Here is an example for an online banking application I worked on with about 10 others. It was designed for a consortium of South American banks. After reading the spec that the technical architect wrote after he came back from South America, I sat down and wrote a story to wrap my head around the system (after all I was the project leader and had to "own" the problem and the ensuing design). I never shared this story with my teammates and I was chatty as I wrote it in a word processor. I'll only show an excerpt:
>
> This application provides internet access to banking services. It should be easily configured to work for different banks. A critical element in the design is the declaration of a common way to call into different backend banking systems. We will define a common set of banking transactions and a framework that will call into banking-specific code that "plugs into" the standard layer implementing the details. The rest of our software will only interface with the bank-independent service layer…At the heart of our system is the ability to rapidly configure our application to work for different backends and to put a different pretty face on each. This includes customizing screen layouts, messages and banner text. The online banking functions are fairly simple: customers register to use the online banking services, then log in and access their accounts to make payments, view account balances and transaction histories, and transfer funds…"

Technique: Identifying application themes. Although you could stop after merely writing and sharing design stories, I've found it useful to use themes as a source of inspiration for identifying key aspects or important areas of design focus. Design themes are what I substitute when I cannot find any elusive metaphor to guide my design.

The key themes I pulled from the online banking story were:

- Modeling online banking activities
- Representing common bank functions
- Configuring system behavior
- Accessing scarce resources

Themes can be either broad or narrow…I find that the broader they are, the more work you have to do to drill down to an appropriate level for identifying candidate objects, but if they are too narrow, there aren't many objects to harvest.

Technique: Leveraging themes to identify key areas of activity and identify initial candidates. Once you have identified major themes, you can use them as one source of inspiration. Make educated guesses about the kinds of inventions you'll need in your design based on the nature of your application and the things that are critical to it. I consider any candidates that I haves tot of this brief dip into the system and start up my design thinking.

Consider each of these perspectives for a particular theme:

- The work your system performs
- Things affected by or connected to your application (other software or physical devices)
- Information that flows through your software
- Decision-making, control and coordination activities

- Stuctures and groups of objects
- Representations of real-world things your application needs to know something about

Although I recommend you consider each perspective when you hunt for “seed corn” candidates, if you find that a particular perspective doesn’t yield any insights, move on.

For example, for the theme “online banking functions” considering the work our system performs led us to consider candidates that specifically supported performing financial transactions and queries. Things affected by our software were the accounts and backend banking transactions. Lots of information flowed through our system to accomplish these activities—information about transactions, account balances, transaction amounts, account history, payments….

Identifying candidates that support each theme is a quick brainstorming activity. Sometimes candidates leap right out when you look at a particular perspective. Often different themes and perspectives reiterate and reinforce the need for certain kinds of candidates. This is good. It builds confidence in the relevance a particular candidate has. At other times, ideas do not come so quickly and you must think more deeply to come up with potential candidates. You won’t find all important candidates in this first pass look through your system and your initial ideas will certainly change.

I use this as an ice breaker for identifying objects that may play a role in the design. In a brainstorming session a team can usually work up a candidate list in a couple of hours.

Tool: Object Role Stereotypes

Role stereotypes from Responsibility-Driven Design are a fundamental way of seeing objects' responsibilities. Think of them as purposeful oversimplifications that help you identify the gist of an object's responsibilities. You can use stereotyping early on to characterize your early candidate objects. Later, you can use stereotyping to characterize your design.

Here is a brief description of six stereotypes:

- Information holder—knows and provides information
- Structurer—maintains relationships between objects and information about those relationships
- Service provider—performs work and in general offers services
- Controller—makes decisions and closely directs others' actions
- Coordinator—reacts to events by delegating tasks to others
- Interfacer—transforms information and requests between distinct parts of a system. User interfacers translate requests from the user to the system (and vice versa). External interfacers usually "wrap" other system APIs. There are other interfacers in complex systems that serve as the "front door" to subsystems.

Technique: Stereotyping a Candidate

Can an object have more than one stereotype? Sure. Each candidate fits at least one stereotype. They often fit two. Common blends: service provider and information holder, interfacer and service provider, structurer and information holder.

I recommend that you identify the major stereotype you want to emphasize and then check your initial ideas against your current thinking from time to time.

Technique: Identifying a Candidate's Purpose

I write a purpose statement on the unlined side of a CRC (candidate-responsibilities-collaborators) card. Not surprisingly, the candidate's purpose matches its stereotype. A candidate knows and does certain things. Briefly, say what those things are. A pattern to follow:

> An object is a type of think that knows or does x. And then mention one or two interesting facts about the object, perhaps a detail about what it does or who it collaborates with.

You might mention one or two interesting facts about the candidate, perhaps a detail about who it works with, to provide more context:

> A FinancialTransaction represents a single accounting transaction performed by our online banking application. Successful transactions result in updates to a customer's accounts.

What do you do with a purpose statement? It can be recycled into a class comment, once you believe a candidate will stick around.

Technique: Identifying Responsibilities

Whether an object primarily "knows things", "does things", or "controls and decides" is based on its role stereotype. Exploring an object's character will lead to an initial set of responsibilities.

For example, information holders answer questions. They are responsible for maintaining certain facts that support these questions. Rather than listing out all the "attributes" of an object, or going into details about its variables, responsibilities are a higher level view of an object. Instead of talking about a customer's first name, last name, surname, nickname, etc…. you can state this general responsibility as "knows name and preferred ways of being addressed.

When designing a service provider ask "what requests should it handle"? Then, turn around and state these as general statements for "doing" or "performing" specific services. Again, responsibilities can be written at a higher-level than a single method or operation. For example, you can talk about "compares to other dates" instead of listing out ">", "<", "<=", etc.

Tool: CRC Cards

CRC cards were invented by Ward Cunningham and Kent Beck in the late 1980s as a way of teaching object concepts to newcomers. They were popularized by my first book, *Designing Object Oriented Software,* and are one technique for informally specify the role and responsibilities of an object, component or subsystem.

In my most recent design book, I've updated my thinking on CRC cards. Initially the first C stood for Class, but now I consider it to stand for a Candidate, which may end up being a component, a class, or even an interface that is shared by multiple classes of objects or components. The R stands for responsibilities, and the second C stands for collaborators or helpers that the candidate uses to accomplish its specific tasks.

On the unlined side of a CRC card is where you write a purpose statement. Not surprisingly, the candidate's purpose matches its stereotype. A candidate knows and does certain things. Briefly say what those things are. A pattern to follow:

> An object (or component) represents a thing that contains certain information or performs specific work. And then mention one or two interesting facts about the candidate, perhaps a detail about what it does or who it works with.

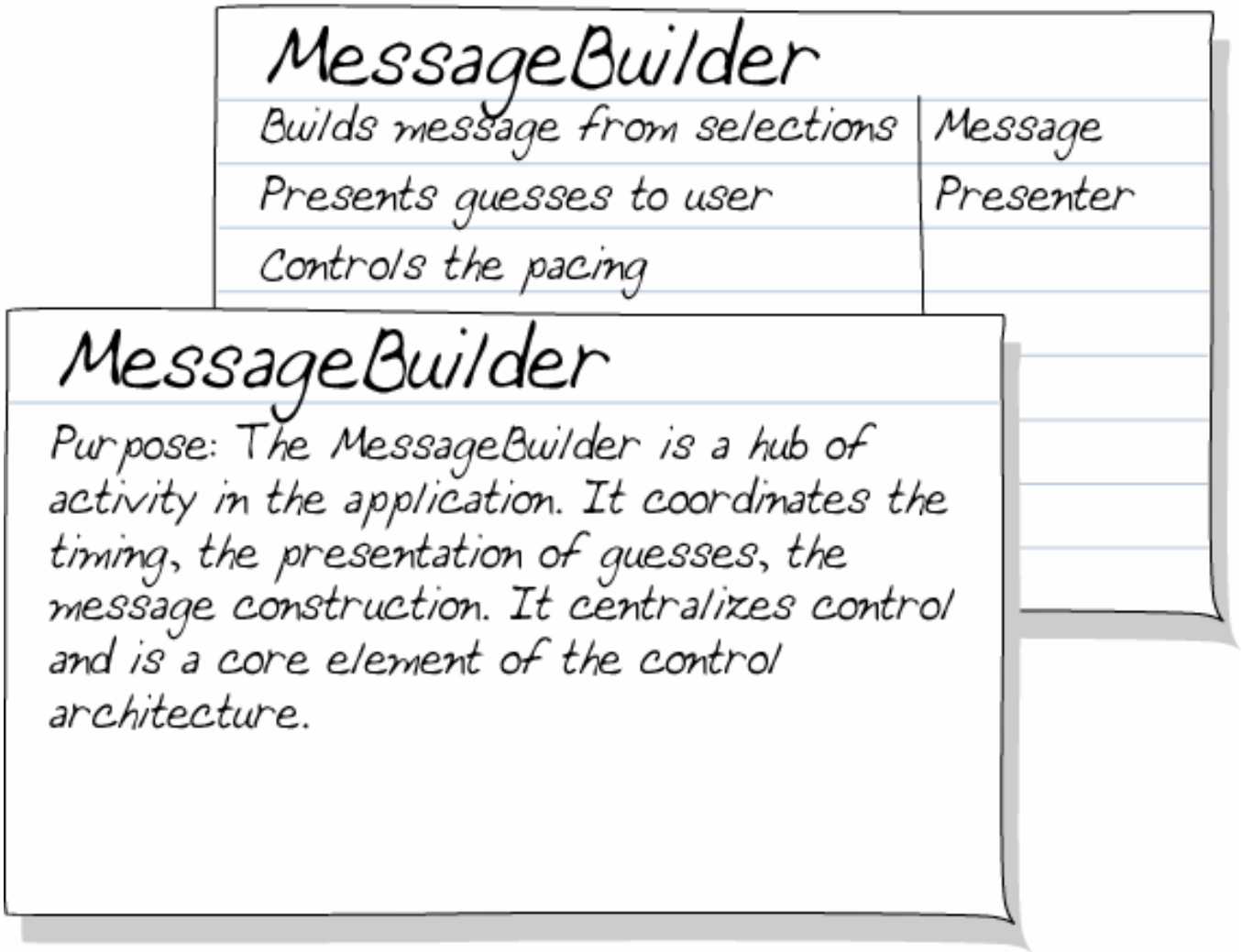
Here is a sample purpose statement:

> A Financial Transaction represents a single account transaction performed by our online banking application. Successful transactions result in updates to a customer's accounts.

On the lined side of the card are spaces for the candidate's responsibilities (on the left hand side). Any objects that it uses to accomplish its tasks are listed on the right hand side of the card. Whether an object primarily "knows things" "does things" or "controls and decides" is based on its role stereotype. Exploring an object's character will lead to an initial set of responsibilities. For example, information holders answer questions. They are responsible for maintain facts that support these questions. When assigning responsibilities to an information holder ask, "what do other objects want to ask about?" Then turn around and state these as responsibilities for

"knowing". When designing a service provider asks, "What requests should it handle?" Then restate these as responsibilities for performing specific services.

Below are the front and back sides of a CRC card.



Guidelines for Finding Objects

The abstractions you choose greatly affect your overall design. At the beginning, you have more options. As you look for candidate objects, you create and invent. Each invention colors and constrains your following choices. Initially, it's good to seek important, vivid abstractions—those that represent domain concepts, algorithms, and software mechanisms. Highlight what's important. If you invent too many abstractions, your design can get overly complex. Not enough abstraction and you'll end up with a sea of flat, lifeless objects.

Your goal is to invent and arrange objects in a pleasing fashion. Your application will be divided into neighborhoods where clusters of objects work toward a common goal. Your design will be shaped by the number and quality of abstractions and by how well they complement one another. Composition, form, and focus are everything! Here are some guidelines to help you during your discovery and invention:

Include only the most revealing and salient facts in a name.

The downside of any descriptive scheme is that names can become lengthy. Don't name every distinguishing characteristic of an object; hide details that might change or should not be known by other objects.

> Should people really have to care that they are using a MillisecondTimerAccurateWithinPlusOrMinusTwoMilliseconds, or will Timer suffice? Detailed design decisions should not be revealed unless they are unlikely to change and they have a known impact on the object's users. Exposing implementation details makes them hard to change.

Give service providers "worker" names.

Another English language naming convention is to end job titles with "er." Service provider objects are "workers," "doers," "movers," and "shakers." If you can find a "worker name" it can be a powerful clue to the object's role

> Many Java service providers follow this "worker" naming scheme. Some examples are StringTokenizer, SystemClassLoader, and AppletViewer.

If a worker-type name doesn't sound right, another convention is to append Service to a name. In the CORBA framework, this is a common convention—for example, TransactionService, NamingService, and so on.

Look for additional objects to complement an object whose name implies broad responsibilities.

Sometimes a candidate represents a broad concern; sometimes its focus is more narrow. If you come across a name that implies a large set of responsibilities, check whether you've misnamed a candidate. It could be that your candidate should have a narrower focus. Or it might mean that you have uncovered a broad concept that needs to be expanded. Looking for objects that round out or complement a broad name can lead to a family of related concepts—and a family of related candidates. Many times we need both specific and general concepts in our design. The more generic named thing will define responsibilities that each specific candidate has in common.

> An object named AccountingService likely performs some accounting function. The name AccountingService isn't specific. We cannot infer information about the kinds of

accounting services it performs by looking only at its name. Either AccountingService is responsible for performing every type of accounting function in our application, or it represents an abstraction that other concrete accounting service objects will expand upon. If this is so, we'd expect additional candidates, each with a more specific name such as BalanceInquiryService, PaymentService, or FundsTransferService. These more specifically named candidates would support specific accounting activities.

Highlight a general concept with more specific candidates. If you can think of at least three different special cases, keep both the general concept and specific ones. If later on, you find that these more specific candidates don't share any responsibilities in common, the more abstract concept can always be discarded. However, if you have simply assigned a candidate a name that is too generic, by all means rename it.

If your candidate could represent historical records of many other things, better to leave it with a more generic name, History instead. If you intend to model transaction history, rename your candidate TransactionHistory. You decide how specific you want to be.

Therein lies the art of naming: choosing names that convey enough meaning while not being overly restrictive. Leave open possibilities for giving a candidate as much responsibility as it can handle, and for using it in different situations with minor tweaks. It certainly is a more powerful design when a candidate can fit into several different situations. The alternative—having a different kind of object for each different case—is workable, but not nearly so elegant.

Choose a name that does not limit behavior.

Don't limit a candidate's potential by choosing a name that implies too narrow a range of actions.

Given the choice, pick a name that lets an object take on more responsibility.

Consider two alternatives for a candidate: Account or AccountRecord. Each could name an object that maintains customer information. From common knowledge we know one meaning of record is "information or facts set down in writing." An AccountRecord isn't likely to have more than information holding responsibilities if we fit its role to conventional usage of this name. The name Account, however, leaves open the possibility for more responsibilities. An Account object could make informed decisions on the information it represents. It sounds livelier and more active than AccountRecord.

Choose a name that lasts for a candidate's lifetime.

Just as it seems funny to hear a 90-year old called "Junior," it's a mistake to name a candidate for its earliest responsibilities, ignoring what else it may do later on. And don't be content to stay with the first name you give a candidate if its work changes.

An object that defines responsibilities for initializing an application and then monitoring for external events signaling shutdown or re-initialization, is better named ApplicationCoordinator than ApplicationInitializer. ApplicationInitializer doesn't imply having ongoing responsibilities after the application is up and running. ApplicationCoordinator is a better name because its more general meaning encompasses more responsibilities.

Choose a name that fits your current design context.

When you choose names, select ones that fit your current design surroundings. Otherwise, your candidates' names may sound strange. What sounds reasonable in an accounting application may seem jarring in an engineering application.

> A seasoned Smalltalker tried hard to set aside his biases when he started working with Java. Although he expected Java classes to have totally different responsibilities, he was surprised to find the Java Dictionary class to be abstract. In Smalltalk, Dictionary objects are created and used frequently.

Shed your past biases when they don't fit your current situation.

Do not overload names.
Unlike spoken language, where words often have multiple meanings, object names should have only one meaning. It isn't good form to have two different types of Account objects with radically different roles that coexist in the same application. Some object-oriented programming languages let you assign the same name to different classes but then force you to uniquely qualify a name when you reference a particular class in code. In Java, for example, classes from different packages can have the same name. In order to uniquely designate a specific one, its name must be qualified by the name of the package where it is defined.

Names of things that can simultaneously coexist within a single application should be given different names. Don't overload a name. Programmers have only one context—the running application—in which to interpret names. They already have enough to think about without adding yet another source of confusion. Compilers are good at automatically applying the correct qualification to a name. Humans aren't!

Eliminate name conflicts by adding an adjective.
Sometimes the best names are already chosen. Still, you need to name your candidate. By adding a descriptive phrase to a name, you can come up with a unique name.

> The synonyms for Property, a class defined in the Java libraries, include these words: characteristic, attribute, quality, feature, and trait. Although "attribute" or "feature" might work, "characteristic" seems stuffy, and "quality" seems strained.

Choose names that are readily understood.
A name shouldn't be too terse. Don't encode meaning or cut corners to save keystrokes. If you want others to get a sense of an object's role without having to dig into how it works, give it a descriptive name. A name can be descriptive without being overly long.

> "Acct" is too cryptic. "Account" is better.

If a common meaning suits a candidate, use it to form a basic definition.
Don't invent jargon for invention's sake. In the case of alternative definitions, choose one that most closely matches your application's themes. Start with a standard meaning, if it fits. Then describe what makes that object unique within your application.

The American Heritage Dictionary has six definitions for account:

1. A narrative or record of events
2. A reason given for a particular action

3. A formal banking, brokerage, or business relationship established to provide for regular services, dealings, and other financial transactions
4. A precise list or enumeration of financial transactions
5. Money deposited for checking, savings, or brokerage use
6. A customer having a business or credit relationship with a firm

It isn't much of a stretch to conceive of different candidates that reflect each of these definitions. In our online banking application, accounts most likely represent money (definition 5). Rules that govern access to and use of funds are important. Different types of accounts have different rules. Although it is conceivable that an account could also be "a precise list of financial transactions", (definition 4), we reject that usage as being too far off the mark. People in the banking business think about accounts as money, assets, or liabilities and not as a list of transactions. In the same fashion, we reject definition 6. It doesn't specifically mention assets. We easily reject definitions 1 and 2 as describing something very different from our notion of accounts in banking. In banking, accounts represent money. We choose definition 5 because it is the most central concept to the world of banking:

> An account is a record of money deposited at the bank for checking, savings, or other purposes.

Add application-specific facts to generic definitions.
The preceding definition is OK, but it is too general for online banking. In the online banking application, users can perform certain transactions and view their balances and transaction histories. We add these application specifics to our original description:

> An account is a record of money deposited at the bank for checking, savings, or other purposes. In the online banking system customers can access accounts to transfer funds, view account balances and transaction historical data, or make payments. A customer may have several bank accounts.

The more focused a candidate is, the better. Of course, a candidate may be suited to more than one use. Objects can be designed to fit into more than one application. A framework operates in many different contexts. A utilitarian object can be used in many cases. If you want your candidate to have a broader use, make this intent clear by writing the expected usage down on the CRC card.

Distinguish candidates by how they behave in your application.
If distinctions seem blurry in the world outside your software, it is especially important to clarify your software objects' roles. Even if you can distinguish between a customer and an account, you still need to decide whether it is worth having two candidates or to have one merged idea. (Don't expect the business experts to help make this decision. It is a purely "technical" modeling one.) A candidate that reflects something meaningful in the world outside your application's borders may not be valuable to your design.

Let's look at the sixth definition of account:

> "An account is a customer having a business or credit relationship with a firm."

What is the difference between a customer and an account? Are they the same? If we had chosen this definition, would we need both customer and account objects in our banking application? When you discover overlapping candidates, refine their roles and make distinctions. Discard a candidate or merge it with another when its purpose seems too narrow (and could easily be subsumed by another candidate). When in doubt, keep both.

> For both Customer and Account to survive candidacy and stick in a design, their roles must be distinct and add value to the application. We could conceive of a Customer as a structurer that manages one or more Account objects. And, in the online banking application, one or more users can be associated with a Customer. For example, the customer "Joe's Trucking" might have four authorized users, each with different privileges and access rights to different accounts. Another option would be to give an Account responsibility for knowing the customer and users. We could then eliminate Customer. We decide to include both Customer and Account in our design because giving those responsibilities to Account objects doesn't seem appropriate—we can envision customers and users sticking around even when their accounts are closed (and perhaps new accounts are opened). So customers are somewhat independent of accounts.

During exploratory design, expect a certain degree of ambiguity. You can always weed out undistinguished candidates when you find they don't add any value. Put question marks by candidates that need more definition. A candidate is just that—a potential contributor.

Look for powerful abstractions and common roles.
Things in the real world do not directly translate to good software objects! Form candidates with an eye toward gaining some economy of expression. Carefully consider which abstractions belong in your object design.

> In our Kriegspiel game there are various actions that a player can perform: "propose a move," "ask whether a pawn can capture in a move," "suspend a game," and so on. It's a pretty safe bet that we have a different candidate for each action: ProposeAMove, SuspendAGame, and so on. Proposing a move seems quite distinct from suspending a game. A harder question is whether we should define PlayerAction as a common role shared by each of these action-oriented candidates? If we can write a good definition for PlayerAction we should do so and define a role that is shared by all player action candidates. There seem to be several things common to all actions (such as who is making the request and how long it is active). Eventually, if we find enough common behavior for PlayerAction, it will be realized in our detailed design as a common interface supported by different kinds of PlayerAction objects. We may define a superclass that defines responsibilities common to specific player action subclasses. Or common behavior might imply the need for another candidate that is the supplier of that shared behavior.

Look for the right level of abstraction to include in your design.
Finding the right level of abstraction for candidates takes practice and experimentation. You may have made too many distinctions and created too many candidates—a dull design that works but is tedious. At the end of the day, discard candidates that add no value, whether they are too abstract or too concrete. Having too many candidates with only very minor variations doesn't make a good design. Identify candidates that potentially can be used in multiple scenarios.

Certain actions affect the position of pieces on a board. Should we have different candidates for each piece's potential types of moves? Not likely. This solution is tedious and offers no design economy. If you can cover more ground with a more abstract representation of something, do so. A single candidate can always be configured to behave differently under different situations. Objects encapsulate information that they can use to decide how to behave. The ProposeAMove candidate can easily represent all moves suggested by any chess piece. This single candidate will know what piece is being moved and its proposed position.

Discard candidates if they can be replaced by a shared role

To find common ground, you need to let go of the little details that make objects different in order to find more powerful concepts that can simplify your design.

What do books, CDs, and calendars have in common? If you are a business selling these items over the Internet, they have a lot in common. Sure, they are different, too. Books likely belong to their own category of items that can be searched and browsed. But all these kind of things share much in common. They all have a description (both visual and text), a set of classifications or search categories they belong to, an author, an availability, a price, and a discounted price. It sounds as if their common aspects are more important, from the Web application's perspective, than their differences. This suggests that all these different kinds of things could be represented by a single candidate, InventoryItem, that knows what kind of thing it is and the categories it belongs to.

Purely and simply, you gloss over minor differences. You don't need to include different candidates for each category of thing. In fact, those distinctions may not be as important to your software as they are to those who buy and use the items.

When you are shopping for items, you may be thinking of how they are used—books are read, calendars hung on a wall, and CDs played—but those distinctions are not important if you are designing software to sell them. Sure, you want to allow for your software to recognize what category something belongs to. You want to list all books together. But you probably want to categorize things in the same subcategory, whether or not they are the same kind of thing. Books about jazz and jazz CDs are in the "jazz items" category.

Only if objects in different categories behave differently in your software do you need to keep different categories as distinct candidates. The real test of whether a category adds value to a design is whether it can define common responsibilities for things that belong to it.

Blur distinctions.

There are times when both concrete candidates and their shared role add value to a design. There are times when they do not. If you clearly see that candidates that share a common role have significantly different behavior, then keep them. Test whether the distinctions you have made are really necessary.

What value is there in including different kinds of bank accounts, such as checking or savings accounts, in our online banking application? Checking accounts, savings accounts, and money market accounts have different rates of interest, account numbering schemes, and daily account draw limits. But these distinctions aren't important to our online banking application. We pass transactions to the banking software to handle and

let them adjust account balances. In fact, because our application is designed to support different banks, each with its own account numbering scheme, a distinction made on account type (checking or savings) isn't meaningful. Our application doesn't calculate interest. So we choose to include only BankAccount as a candidate. If we were designing backend banking software that calculated interest, our decision would be different.

Guidelines for Assigning Responsibilities

Our strategy for assigning responsibilities to objects is very simple: Cover areas that have big impacts. Look for actions to be performed and information that needs to be maintained or created. You can glean information from several sources: Perhaps you have a specification of your software's usage; you may have written some use cases; or you may know of additional requirements or desired characteristics of your software. Responsibilities emerge from these sources and from ideas about how your software machinery should work.

You will need to reformulate demands and characteristics and software descriptions into responsibility statements. If statements seem too broad to be assigned to individual objects, create smaller duties that can be. These smaller subresponsibilities collectively add up to larger ones. Formulating and assigning responsibilities to objects involves inspiration, invention, and translation of constraints and general descriptions into specific responsibilities. Assigning responsibilities to objects gives them shape and form. Here are some guidelines for finding responsibilities and assigning them to objects in your candidate object model:

Responsibilities come from statements or implications of system behavior found in use cases.

There is a gap between use case descriptions and object responsibilities. Responsibilities are general statements about what an object knows, does, or decides. Use case descriptions are statements about our system's behavior and how actors interact with it. Use cases describe our software from the perspective of an outside observer. They don't tell how something is accomplished. Use cases provide a rough idea of how our system will work and the tasks involved. As designers we bridge this gap by transforming description found in use cases into explicit statements about actions, information, or decision-making responsibilities. This is a three-step process:

- Identify things the system does and information it manages.
- Restate these as responsibilities.
- Break them down into smaller parts if necessary, and assign them to appropriate objects.

Depending on how much detail is included in a use case, it can be more or less difficult to find statements about our software's behavior. Use cases aren't packed with actions or behaviors that are readily ascribed to individual objects. However, even from this high-level narrative we can glean responsibilities. By intent, use cases leave out design details. They are descriptive, not prescriptive. They tell a story. Use cases are descriptions that we use as general guides as we build our design. Use case scenarios describe step-by-step sequences. Supposedly they include more detail than an overview.

Additional responsibilities come from plugging inherent gaps in use case and other system descriptions.

To gain confidence in your design, you must dig deeper into the nature of the problem and ask questions. Just by looking at our list of responsibilities we can come up with questions leading to more responsibilities. The sooner you ask and get answers to specific questions that will shape your system's behavior, the better. The answers will guide your thinking as you discover more detailed software responsibilities.

Use cases rarely describe aspects of control, coordination, error detection, visual display, timing, or synchronization. Designers must figure out these details. You can push forward with assigning responsibilities, even with many questions left answered. Tag those questions that will have the biggest impact. If you envision a range of possible answers and guess at those that are most likely to have the most impact, you can know where to push for answers.

Take two approaches: Identify responsibilities as well as unresolved questions. Continue to work on what you do know. Identify questions that are most likely to significantly impact your design. Once you get answers, you undoubtedly will refine your design. You won't know how comprehensive your solution needs to be until you get some answers.

Defer the specific design of control and coordination responsibilities until you make choices about how to distribute decision-making and control responsibilities. Test your collaboration model with both "happy path" and more complicated scenarios. For now, collect and assign as many specific responsibilities as you can.

Design, and the assignment of responsibilities, is iterative. You make an initial pass at pinning down responsibilities, and then you rework your ideas as you come to know more about your objects and their interactions.

Responsibilities come from themes and design stories.
Earlier, we recommended that you write a brief story that describes the key ideas behind your software. This design story kept you focused on what's important and stimulated your thinking about appropriate candidates. You can return to this story to extract some responsibilities.

Because of the story's brevity, the responsibilities we find reflect only the highlights.

For example, from the online banking application story, we surmise that connections (and other scarce resources) must be managed. We can assign responsibilities for managing connections to specific connection managers. Financial transactions will be performed by the coordinated work of many objects; each with specific responsibilities. To assign responsibilities for performing transactions, we need to consider the details of each transaction in turn. Each transaction will require a different sequence of work steps, although some may be in common (for instance, all transactions are logged along with user-specific notes in the system's database).

Responsibilities come from following "what if... and then.. and how?" chains of reasoning.
To gain even more insight, you need to consider how various requirements may impact your design. This involves more heavy mental lifting than our other responsibility sources. In this case, you don't start with a specific task such as "make a loan payment" or specific action such as "verify credit load." Instead, you need to lay a path from a high-level goal, such as "the software should be offline only during routine maintenance," to a series of actions or activities that achieve it. Only then can you make statements about what the system needs to specifically do as a consequence. Once you've come up with these specific conclusions, you can formulate specific responsibilities.

We can think of many situations when we've chased design implications. Most involved short, solo excursions. Individuals thought through the problem and followed their instincts. As a group we might have kicked around the nature of the problem before the individuals went away and thought through the problem. Reasoning towards a solution seems to be an individual activity or one taken on by a small team of like-minded souls.

Often your initial design will not be as simple or as elegant or as complete as you'd like. You don't have time to make many wrong moves. On the online banking project, the designer followed these principles: Keep concerns separate, and don't intermix responsibilities. Each object or component should do its job simply and well. Following his initial line of reasoning led him to very specific responsibilities. His objects weren't up to his high standards, but they did the job.

Responsibilities naturally arise from an object's stereotypical roles.
Whether an object primarily "knows things," "does things," or "controls and decides" is based on its role stereotype. Exploring an object's character will lead to an initial set of responsibilities. Information holders answer questions. They are responsible for maintaining facts that support these questions. When assigning responsibilities to an information holder, ask, "What do other objects or parts of the system want to ask?" Restate these queries as responsibilities for "knowing." Look for specific information that fits each candidate's role. Each information holder should support a coherent, related set of responsibilities. Secondarily ask, "What else does this information holder need to know or do in order to carry out its public obligations?" These will be private responsibilities it undertakes to carry out its public duties.

When designing a service provider ask "What requests should it handle?" Then turn around and state these as responsibilities for "doing" or "performing" specific services. Similarly, structurers should have responsibilities for maintaining relationships between other objects and for answering questions about them. Interfacers will have specific duties for translating information and requests from one part of the system to another (and translating between different levels of abstraction). Coordinators have specific duties for managing cooperative work. Controllers should be responsible for fielding important events and also directing the work of others.

Look for private responsibilities that are necessary to support public responsibilities.
Even as you make general statements of responsibilities, you may think about how your objects might accomplish them. When should you focus on these details? As a matter of principle, concentrate first on what an object does for others. Once you've arranged these core, publicly visible responsibilities, reach for additional private responsibilities that support them.

Record responsibilities as you think of them. Make sure you are comfortable with your object's role in its community before you work out many details. If you know these details, you can record them. What's the best way to do this? Should you get more specific with your responsibility statements, or are there other options?

Earlier, we mentioned that responsibilities are recorded on CRC cards along with a statement of purpose and a list of collaborators. Given the limited space on the CRC cards, you should use this real estate wisely. Make responsibility statements as brief as possible. Convey necessary

information by reworking and revising all parts of your object's description. Don't pack everything into responsibilities. Record details in ways that let you remember them without creating clutter.

Responsibilities come from examining relationships between candidates.
Examining relationships between candidates can identify additional responsibilities. Objects can be related in complex ways: "composed of," "uses," "owns," "knows about," and "has" have very imprecise meanings in the English language. However, objects we tag as "structurers" nearly always have responsibilities for "maintaining" or "managing" objects they organize, whether we think of them as being "composed of," "owning," "knowing," or "aggregating" those objects.

When an object plays the role of a structurer, it organizes groups of objects. Because of this role, it likely has responsibilities for answering questions about the things it knows of. To make specific responsibility assessments, we need to understand why a structurer exists and how its responsibilities change as its composition changes.

Responsibilities may be associated with important events during an object's lifetime.
Some objects' responsibilities are largely shaped by how they react. These objects are spurred to action by specific events. Controllers and coordinators fit this profile: most of the work they do is in response to stimulus they interpret.

Not all objects are so externally driven. Some react to internal changes. When an object is create and when it is no longer used are common places to find responsibilities for gracefully entering and leaving the scene. In most object-oriented languages, objects are notified of their impending exit with a "finalize" notice, allowing them to release resources before leaving.

Responsibilities may be assumed when an object fits into its technical environment.
The responsibilities we have identified up to this point have been in support of required system behavior. We mention this source last because it yields responsibilities of a different nature: those required for an object to fit into its software context. As a designer, you don't invent these responsibilities but you must understand their implications. Quite simply, your objects won't function properly unless they take on these implementation-specific responsibilities. Implementation-specific responsibilities shouldn't be your first concern. But if you know where your objects are headed, plan for them.

To start, state responsibilities generally.
Responsibilities are best stated at a level above individual attributes or operations. Don't get overly specific in your statements. A statement of responsibility, if worded generally, can encompass many specific requests. There may be 10 ways to ask for tax calculations that are covered by the statement "Calculate all taxes based on locale." There isn't enough room on a CRC card to record very many details. These lower-level details belong in an information model or some other, more precise description. Use CRC cards for high-level descriptions. If you are worried you'll forget details, jot down hints on the card that will help you remember them as you work: "knows its name and preferred ways of being addressed (e.g., title, nicknames, etc.)".

Space on cards is limited, so use it wisely.

Find the right level of description.
How many responsibilities do you need to shape an object's character? Responsibilities can be tersely worded or slightly more descriptive. It's a matter of personal and team style. You can be more or less brief, just as long as you and your teammates understand on another.

Use strong descriptions.
An object can seem ill defined if its responsibilities seem hazy. Behind a wall of vagueness can lie details that should not be ignored. Avoid weakly stated responsibilities if you can find stronger, more explicit descriptions.

Daryl Kulak and Eamonn Guiney, in their book *Use Cases: Requirements in Context* caution against giving use cases weak names. They suggest that more concrete verbs make for less vague use case names. If you use weak verbs, it may be because you are unsure of exactly what your use case should accomplish. The same principle applies to naming responsibilities for actions. The more strongly you can state a responsibility, the less you are fudging.
Strong Verbs: remove, merge, calculate, credit, register, debit, activate

Weak Verbs: organize, record, find, process, maintain, list, accept

Of course, there are always exceptions to the rule. A weak-sounding phrase may have specific meaning in a certain context. In this case, don't look for a stronger term. Listing a property has a very specific meaning in the real estate business: It means to put a property on the market for sale.

Be opportunistic.
Thinking about one object leads to thinking about others. When considering an object's public responsibilities, you think about why its clients need to call on these services and what they are ultimately responsible for accomplishing. When you look at a single responsibility, you think about how it might be accomplished. This shift of focus is good (as well as hard to avoid). You test the fit of an object to its context by looking at both its use and its effects on others. If you hop around too much, however, you might leave an object before you have a firm grasp of its responsibilities. To avoid this, take a first pass at an object's major responsibilities before moving too far away from it.

Decide how an object will divide or share the work of a large or complex responsibility.
An object has three options for fulfilling any responsibility. It can either

- Do all the work itself
- Ask others for help doing portions of the work (collaborate with others)
- Delegate the entire request to a helper object

When you're faced with a complex job, ask whether an object is up to this responsibility or whether it is taking on too much. A responsibility that is too complex to be implemented by a single object essentially introduces a new sub-design problem. You need to design a set of objects that will collaborate to implement this complex responsibility. These objects will have roles and responsibilities that contribute to the implementation of the larger responsibility.

At this point we're not asking you to make detailed decisions about how to design specific collaborations between these objects, only that you think through your options for assigning subresponsibilities. If a responsibility seems too big for one object, speculate on how you might break that responsibility into smaller logical chunks. These can be given as work assignments to other objects. Pursuing this line of thinking may lead you to new candidates with smaller, more tightly focused roles.

Make sure an object isn't doing too much.
If you find an object with a long laundry list of responsibilities, this could indicate one of two problems: either you are stating its responsibilities in too much detail, or it is taking on too much. It is easy to rewrite responsibilities at a higher level.

However, if your object is too busy, consider splitting it into several smaller ones that will work together on the problem. Expect these objects to collaborate with one another. Although it may require more study before you obtain an overall understanding of this new system of objects, distributing the work among a number of objects allows each object to know about relatively fewer things. It results in a system that is more flexible and easier to modify.

Keep behavior with related information.
If an object is responsible for maintaining certain information, it is logical to assign it responsibilities for performing any operations on that information. This makes the object smarter; not only does it know things, but it also can do things with what it knows! Conversely, if an object requires certain information to do its job, it is logical (other things being equal) to assign it the responsibility for maintaining that information. In this way, if the information changes, no update messages need to be sent between objects.

Distribute system intelligence.
A system can be thought of as having a certain amount of intelligence. The sum of a system's intelligence is what it knows, the actions it can perform, and the impact it has on other systems and its users. Given their roles within a system, some objects can be viewed as being relatively "smart," whereas others seem less so. An object incorporates more or less intelligence according to how much it knows or can do and how many other objects it affects. For example, structuring objects such as sets or arrays are usually not viewed as particularly intelligent: They store and retrieve objects but have relatively little impact on the objects they store or any other parts of the system. Other structurers can be more intelligent. They have responsibilities not only for maintaining their contents, but also for answering questions about them collectively.

Objects with responsibilities for controlling activity can be more or less intelligent, depending on how much work they delegate and how much they know about the work of those they manage. Guard against the tendency to make controllers too intelligent. We prefer to give the collaborating objects as much responsibility as they can handle. The more intelligent controllers are, the less intelligent are those that surround them. If you place too much responsibility in a controller, you lose design flexibility. Our goal isn't to evenly distribute intelligence, but to give objects those responsibilities they can handle.

Keep information about one thing in one place.
In general, meeting the responsibility for maintaining specific information is easier if that information isn't shared. Sharing implies a duplication that can lead to inconsistency. Part of making software easier to maintain is eliminating potential discrepancies. If more than one object must know the same information to perform an action, three possible solutions exist:

A new object could be created with the responsibility for being the sole repository of this information. This information holder would be shared among those who have a "need to know." It may be that the information "fits" with the existing responsibilities of one of the existing objects. In that case, it could assume the added responsibility of maintaining the information. Others could request this information when they need it.

It may be appropriate to collapse various objects that require the same information into a single object. This means encapsulating the behavior that requires the information into a single object and obliterating the distinction between the collapsed objects. Sometimes we go overboard, factoring out responsibilities into roles that are too small. In that case it is better to pull them back into a single, more responsible object.

Make an object's responsibilities coherent.
They should all relate in some way to the overall role of the object. An object as a whole should be the sum of its responsibilities. These responsibilities should complement one another. Everything an object knows or does should contribute to its purpose or fit into your design model.

Restrict an object's responsibilities to a single domain.
Meilir Page-Jones in *Fundamentals of Object-Oriented Design in UML* introduces a way of dividing a software system (and the objects that live within it) into domains. Domains are Page-Jones's way of dividing the machinery of an application into different contexts. According to Page-Jones, objects that live in lower domains shouldn't have responsibilities that tie them to objects in a higher domain. The more you tie objects in a lower domain to a higher one, the harder it is to reuse them in different contexts.

Page-Jones's divisions (from higher to lower level domains) are as follows:

- Application: objects valuable for one application
- Business: objects valuable for one industry or company
- Architectural: objects valuable for one implementation architecture
- Foundation: objects valuable across all business and architectures
- Foundation objects are further divided into three categories or subdomains:
- Fundamental: objects so basic that many programming languages include them as primitive data types, such as integers or reals
- Structural: objects that organize others, such as sets, collections, hashtables or queues
- Semantic objects: objects that represent basic concepts with specific meaning, such as date, time, or money

To test whether two different objects are in the same domain, ask, "Can one object be built without any knowledge of the other?" If so, these two objects aren't likely to be in the same

domain. But there are still places where you could tangle domains if you aren't careful—for example, when you need to convert from one type of object to another.

Avoid taking on nonessential responsibilities.
Avoid diluting an object's purpose by having it take on responsibilities that aren't central to its main purpose. Taking on responsibilities is easy to do, especially when you're deciding who should be responsible for maintaining a relationship. The obvious first answer is to make one or the other, or both, related objects be responsible.

The easy first answer isn't always the best. Each new responsibility needs to be considered carefully. It is easy to "slip one in" as an easy solution and avoid thinking through the consequences. An object that has a lot of links to others is going to be harder to maintain and move to a new context.

Consider creating a new object that is responsible for structuring the relation between people and dogs, another for people and valued property, and so on. Each of these new objects knows of a specific relationship. Instead of one big object knowing many others, the net result is a few simpler objects, each knowing some specific relationship. This is one way to "straddle" objects in separate domains. It results in a trimmer Person, unburdened with responsibilities that aren't intrinsic to its nature. Of course, this too, can be carried to extremes. Too many objects with responsibilities to "glue" others together can also make a design brittle and hard to understand.

Decide what relations are intrinsic to an object in the context of your application and which are not. Assign responsibilities for maintaining nonessential relations to new structurers.

Don't overlap responsibilities.
Sometimes you aren't sure which object should check, guarantee, or ensure that things are done the right way. Who should be ultimately responsible? If you want a robust system, you must make your objects and neighborhoods resistant to careless mistakes and errors.

Should you make the client check before it calls on the services of another? Should you give service providers responsibilities for checking that their requests are properly formed? If you're not sure whom the clients are or under what situations a responsibility will be carried out, you might be inclined to put in safety checks everywhere.

This line of reasoning leads to overly zealous objects, all of them fretting about the state of the system. It can be extremely costly to maintain such a complex system of objects. You are better off developing a simple, consistent strategy for checking and recovering, and sticking with that. Not everyone needs to be involved or "share in an important responsibility."

If you want an object to be impervious to malicious requests, give it responsibilities for detecting and deflecting them. Once you've given an object that responsibility, design its clients to be more cavalier; they need only react to bounced requests, not prevent them. We will return to this topic, when we design collaborations. But for now, consider that when you give one object a responsibility, you are potentially relieving the workload of another. It isn't necessary to build in overlapping responsibilities unless your system explicitly demands redundancy.

Problem: You have a big responsibility that doesn't seem to belong to any candidate.
Who should be responsible for solving world peace or ending world hunger? There aren't simple answers because these are extremely broad problems. If you really wanted to tackle world peace or hunger, you'd have to break these enormous problems into smaller factors that, if solved, might contribute to lessening friction or reducing hunger. Divide a big problem into smaller problems, and solve those.

Big software responsibilities can seem equally daunting to those tasked with solving them. What object should be responsible for "interacting with the user" or "performing business functions" or "managing resources" or "doing the work"? If a responsibility seems too big or too vague, break it into smaller, more specific ones that can be assigned to individual objects. Treat the "big responsibility" as a problem statement and reiterate through identifying specific objects with smaller responsibilities that add up to the larger responsibility.

Problem: You don't know what to do with a vague responsibility.
If you can't get more concrete, perhaps you are trying to add precision to a statement that is so general that you can't get any traction. You don't know enough to break it down into subparts. Before you can design a solution, you may need further definition from someone who knows more about the problem than you do. It's always fair to ask, "Can you be more specific about what you mean by performing business functions?" If you are lucky, your statement may not really be a problem at all. You may already have assigned specific responsibilities that are subsumed by a broad unapproachable statement.

Problem: You can't decide between one of several likely candidates.
Sometimes it isn't obvious which candidate should be assigned a specific responsibility. When you're choosing which of several objects to assign a responsibility, ask, "What are all my options for assignment? If I choose this possibility, what does that imply for its surrounding neighbors?" If you have trouble assigning a particular responsibility, the solution is simple: Make an arbitrary assignment and walk through the system to see how it feels. There isn't necessarily a single "right" answer. Don't get in a jam thinking that you must optimally solve the problem or that there is only one optimal assignment. There may be several, or none.

Problem: You have trouble assigning a specific responsibility.
You may get stuck on a responsibility that seems to be reasonably stated but has nowhere to go. This could mean that you are covering new territory and may need to invent a new candidate. Great! This is progress. Or it could be that even though the responsibility is specific, your existing candidates' responsibilities are stated at a higher level of detail. If so, remember that responsibilities are general statements; what you think of as a specific responsibility you have trouble assigning may actually be an implementation detail that doesn't really belong on a CRC card. If so, save it for later.

Problem: You are worried about how a responsibility is actually going to be accomplished.
You've stated responsibilities generally, but you have nagging doubts. How will each object carry out its duties? Are you concerned because you suspect that something is missing? If so, follow your instincts and figure that out. Are you a stickler for details? Until you see running

code, you never believe a design will work. If so, relax. Your design isn't finished quite yet. And it will change as you design collaborations, too. Once you are comfortable with how you've arranged responsibilities among a set of collaborators, then you can pin down responsibilities to a specific implementation. A responsibility for maintaining knowledge could mean that

- The object holds on to the fact directly.
- It could derive it from other information sources.
- When asked, it turns around and collaborates with another that can compute (and is responsible for reporting the results to others).

At this point, all your options are open. Stating that a MonetaryTransaction "knows its applicable taxes" could mean that it stores its taxes directly in variables or that, when asked, it turns around and delegates this request to a tax calculator object that does all the work. We don't have to decide these things just yet. In fact, until we know our candidates and all the dimensions of the problem better, we don't know enough to make informed decisions about how "knowing" responsibilities are best implemented.

Tool: Control Center Design

Deciding on and developing a consistent control style is one of the most important decision a designer makes. A **control center** is a place where objects charged with controlling and coordinating reside.

Developing a control style involves decisions about:

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

While it is true that many frameworks make some of these decisions for you, there is much room for judgment (and lots of options to explore). It isn't just a matter of style. Control design affects complexity and ease or difficulty of your design to change. Your goal should be to develop a dominant, simple enough pattern for distributing the flow of control and sequencing of actions among collaborating objects.

A control style can be **centralized, delegated,** or **dispersed.** But there is a continuum of solutions. One design can be said to be more centralized or delegated than another.
Your goal should be to develop a dominant pattern for distributing the flow of control and sequencing actions among collaborating objects.

If you adopt a **centralized control style,** you place major decision-making responsibilities in only a few objects—those stereotyped as controllers. These decisions can be simple or complex, but with centralized control, most objects that are used by controllers are devoid of any significant decision-making responsibilities. They do their job, but generally are told by the controller how to do it.

Choosing a **delegated control style,** you make a concerted effort to delegate decisions among objects. Decisions made by controllers will be limited to deciding what should be done and handling exceptions. Following this style, objects with control responsibilities tend to be coordinators rather than controllers controlling every action.

Choosing a **dispersed control style** means distributing decision-making across many objects involved in a task. I haven't worked on systems where I've consciously use this style, although you could consider a pipes-and-filter architecture or chain-of-responsibilities pattern to be a dispersed control style.

Nothing is inherently good about any particular style. They all have plusses, minuses, and things to watch out for. But generally, I prefer a delegated control style as it seems to give more life (and responsibilities) to objects outside a control center and avoids what Martin Fowler calls "anemic domain models". In a nutshell, here are characteristics of each style.

Centralized Contol. 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 to watch out for with this strategy:

- Control logic getting overly complex
- Controllers becoming dependent on information holders' contents
- Objects becoming indirectly coupled as a result of the controller getting information out of one object and stuffing it into another
- Changes rippling among controller and controlled objects
- The only interesting work (and programming effort) being done in the controller

Delegated Control. A delegated control style passes some of the decision making and much of the action off to objects surrounding a control center. Each neighboring object has a significant role to play:

- Message between coordinators and the objects they collaborate tend to be higher level requests (e.g. instead of setters and getters and minute calls, there are more "Nike" requests—justDoIt()).
- Changes are typically more localized and simpler
- Easier to divide interesting work among a team

Dispersed Control. A dispersed control style spreads decision making and action among objects that individually do little, but collectively, their work adds up. This isn't an inherently bad strategy; but avoid these tendencies:

- Little or no value added by those receiving a message and merely delegating to the next object in the chain
- Hardwiring dependencies between objects in long collaboration chains

The Pipes and Filters architectural pattern exemplifies well-designed dispersed control. It divides the task of a system into several sequential processing steps. These steps are connected by the data flow through the system. The output of a step is "piped" to another processing step ("a filter"). A filter consumes and delivers data incrementally—in contrast to consuming all its input before producing any output—to enable parallel processing. The input to the system is provided by a data source such as a text file. The output flows into a data sink such as a file or display device. The data source, the filters and the data sink are connected by pipes.

Filter components are the processing units of the pipeline. A filter enriches, refines or transforms input data. It enriches data by computing and adding information, refines data by concentrating or extracting data, and transforms it by delivering it in some other format. A filter may do all three activities.

Technique: Control Center Design

Don't adopt the same control style everywhere. Develop a control style suited to each situation:

- Adopt a centralized style when you want to localize decisions in one place
- Develop a delegated style when work can be assigned to specialized objects
- Several styles can and should co-exist in a complex application
- Look at how a particular framework (or accepted style of programming, say, how a J2EE application "typically does things") impacts the control styles you adopt and whether it

injects undue complexity into your design. For example, a style that separates business rules from information holder objects results in 2x the number of classes, but arguably makes it easier to unit test information holders.

- Assess whether your ideas about control style line up with other experts or pattern authors

Control styles within subsystems vary widely. As a general design rule, make analogous parts of your design work in similar ways.

Tool: Trust Regions

One way to get a handle on where collaborations might be streamlined and simplified is to carve your software into regions where trusted communications occur. Generally, objects located with the same **trust region** communicate collegially, although they still encounter exceptions and errors as they do their work. Within a system there are several cases to consider:

- Collaborations among objects that interface to the user and the rest of the system (unless information it is verified before it is sent to the rest of the system, it shouldn't be trusted to be valid)
- Collaborations among objects within the system and objects that interface with external systems
- Collaborations among objects outside a neighborhood or subsystem and objects inside
- Collaborations among objects in different layers
- Collaboration among objects you design and objects designed by someone else
- Collaborations with library objects

Whom an object receives a request from is a good indicator of how likely it is to accept a request at face value. Whom an object calls on determines how confident it can be that the collaborator will field the request to the best of its ability. It's a matter of trust. In general, when objects are in the same layer or neighborhood, they can be more trusting of their collaborators. And they can assume that objects that use their services call on them appropriately.

If a request is from an untrusted or unknown source, extra checks may be made before a request is honored.

Technique: Identify trust regions. Carve your software into regions where "trusted" communications occur. Objects in the same trust region communicate collegially, although they may still encounter exceptions and errors. When an object uses a collaborator that is outside of its trust region it may take extra precautions, especially if it has responsibilities for making the system more reliable. It may need to:

- Pass along a copy instead of sharing data
- Check on conditions after the request completes
- Employ alternate strategies when a request fails

Objects at the "edges" of a trust region typically take on more responsibilities. For example, an object that receives a message from an "outsider" may make initial checks, then only pass along known good requests to others.

References

Responsibility-Driven Design

The most recent source on Responsibility-Driven Design can be found in the book:

R. Wirfs-Brock and A. McKean, *Object Design: Roles, Responsibilities, and Collaborations,* Addison-Wesley, 2003.
ISBN 0-20-137943-0

Errata can be found at http://www.wirfs-brock.com/PDFs/Book%20Corrections.pdf

Responsibility-Driven Design has been written about in several "old classics". The original OOPSLA paper appeared in 1989 and a book (still in print) in 1990.

R. Wirfs-Brock and B. Wilkerson, "Object-Oriented Design: A Responsibility-Driven Approach," OOPSLA '89 Conference Proceedings, SIGPLAN Notices, 24(10), pp. 71-76, 1989.

R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software,* Prentice Hall, 1990.
ISBN 0-13-629825-7

More recently, I have been writing about design in general (and responsibility-driven design principles) in IEEE Software as its Design Editor. A copy of my columns can be found on my website: www.wirfs-brock.com/resources

A variety of magazine articles and tutorial presentations also talk about certain ideas conceived since 1990. Many of the articles can be downloaded from www.wirfs-brock.com/resources

R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.
ISBN 0-13-629825-7

J. Carroll, Editor, Chapter 13, "Designing Objects and Their Interactions: A Brief Look at Responsibility-Driven Design", pp. 337-360, in *Use Case-Based Design: Envisioning Work and Technology in System Development,* John Wiley & Sons, 1995.
ISBN 0-471-07659-7

From The Report on Object Analysis and Design:
R. Wirfs-Brock, "How Designs Differ," vol. 1, no. 4.

R. Wirfs-Brock, "Adding to your Conceptual Tool Kit: What's Important about Responsibility-Driven Design", vol. 1, no. 2.

R. Wirfs-Brock, "Characterizing Your Application's Control Style"
From the Smalltalk Report:
R. Wirfs-Brock, "Designing Use Cases: Making the Case for a Use Case Framework," vol. 3, no. 3.

R. Wirfs-Brock, "The Art of Designing Meaningful Conversations," vol. 3, no. 5.

R. Wirfs-Brock, "Characterizing Your Objects," vol. 2, no. 5.

R. Wirfs-Brock, "Characterizing Object Interactions," vol. 2, no. 6.

R. Wirfs-Brock, "Object Visibility: Making the Necessary Connections," vol. 2, no. 2.
From Object Magazine:

Wirfs-Brock, “Stereotyping: A Technique for Characterizing Objects and their Interactions,” Nov/Dec. 1993.

CRC Cards

Ward Cunningham and Kent Beck introduced the idea of CRC cards in their OOPSLA paper. This OOPSLA paper was written when Ward was at Tektronix, the same year the Rebecca and Brian Wilkerson wrote about Responsibility-Driven Design. CRC cards were originally conceived as a teaching tool; they since have evolved into a widely used design aid.

K. Beck and W. Cunningham, “A Laboratory for Teaching Object Oriented Thinking,” OOPSLA ‘89 Conference Proceedings, SIGPLAN Notices, 24(10), pp. 1-6, 1989.

Nancy Wilkerson has written an entire book on using CRC cards. She has added to the ‘lore’ of CRC cards with practical tips and the notion of detailed design cards.

N. Wilkinson, *Using CRC Cards An Informal Approach to Object-Oriented Development,* SIGS Books, 1995.
ISBN 1-884842-07-0

Design Principles, Libraries and Contracts

Bertrand Meyer’s books aren’t just about Eiffel. He introduced the notion of formally specifying the contractual relationships between objects. His monumental achievements are documented in 1200+ pages of *Object-Oriented Software Construction, Second Edition*.

B. Meyer, *Object-Oriented Software Construction, Second Edition*, Prentice Hall, 1998.
ISBN 0-13-629155-4

B. Meyer “Design by Contract,” in *Advances in Object-Oriented Software Engineering,* D. Mandrioli and B. Meyer, editors, Prentice Hall, 1992, pp. 1-50.

B. Meyer, “Applying ‘Design By Contract’,” IEEE Computer, vol. 25, no. 10, pp. 40-51.

ISBN 1-884777-10-4

Design in General

Donald Norman writes about why designers’ creations are hard to use. His examples come from industrial design, but his design principles are appropriate to object and framework designers. We recommend his books to anyone interested in making designs easier to use.

D. Norman, *The Design of Everyday Things,* Bantam Doubleday, 1988
ISBN 0-38-52677-64

Reading for Pleasure

You can tell that I am a true bibliophile when I include references to a book that appears to have nothing to do with design. This book has influenced me my thinking about design, writing style, natural language, poetry, and Italian music. It is dense and beautiful. He has an intuition that it might be the best book that he will ever write. For the creator of *Godel, Escher, Bach*, that is saying something. An astounding book!

D. R. Hofstadter, *Le Ton beau de Marot,* Basic Books, 1998
ISBN 0-465-08645-4

Data Collection Problem OOPSLA DesignFest™ Problem

Background
A local forest technology company, Forests 'R' Us, wants to build and sell a system for gathering and analyzing weather information to predict forest fires and help with water table management. The Arbor2000 will be sold to National Forests, Environment Canada, the U.S. Forest Service, and large private landowners. It will consist of hardware and software both locally in the owner's office building and remotely in the forests.

The data sensors in the forest report at various intervals to the central computer via satellite, packet radio, cell phone, dial-up phone, or dedicated line. The central computer stores and analyzes the information. The users run a wide variety of reports, browsers, historical trend analysis, and future prediction algorithms over the data. Furthermore, given the inherently geographic nature of the data, many of the reports incorporate maps.

The sensors, such as temperature, sunlight intensity, wind speed and direction, rainfall, and so on, com in three basic types:

1. those that report on a regular basis (every minute, hour, day, month),
2. those that only report when a significant event occurs (a certain amount of rain has fallen, the temperature rises above a threshold), and
3. those that must be queried.

Some sensors fall in multiple groups; for example, they report events but can also be queried.

The sensors are produced by different manufacturers and return numeric values in a wide variety of units (miles/hour, km/hour, lumens, watts, calories/day, etc.) and at widely varying intervals and tolerances.

Additionally, the data links are not necessarily reliable, and yet the system must deal with all these issues while presenting both a uniform and a detailed view of the data to the user and his or her agent/analysis programs.

Desired Programs
Forests 'R' Us needs three categories of programs:

1. one to gather the sensor data as it arrives and store it in a database,
2. one to configure the field sensors, and
3. the one to provide the user interface for browsing and analyzing the data.

Gathering the sensor data is relatively simple: the field sensors send information packets to the central computer, and the central computer stores them. Each packet contains the sensor ID, the time stamp, and the numeric sensor measurement. For cost reasons, many sensors are grouped into sensing units which send their data together (e.g., wind speed, direction, humidity, and temperature) via one phone call rather than four separate calls.

Configuring the field sensors consists of telling the software where each sensor is physically located and what type of sensor it is. Additionally, many sensors have different settings for measurement units and errors, reporting intervals, etc., so these too are configured. Because this is a 7 x 24 system, sensors can be replaced at any time, usually with an upgraded model and thus with different measurement units, error tolerances, etc.

The browsing and analyzing programs are the heart of the system. The analysis algorithms provide fire danger ratings, water table estimates, flash flood warnings, and so on. The browsing interfaces provide detailed information, both tabular and geographic, from the database. For example, the temperature maps similar to those seen on the evening news are one of the possible graphical outputs. The user should be able to navigate through the information in many ways including:

1. Map browsing multiple sensor types (temperature and rainfall) or multiple time periods (temperature over the previous month).
2. Browsing the type and status of the sensors at any location or locations.
3. Browsing the reliability and age of the information for any sensor and/or location.

To provide for future expansion, each of the predicted values available for display (e.g. temperature, rainfall, fire danger, flash flood risk, etc.) should be computed via a plug-in module. (Forests 'R' Us intends to sell additional modules for other risk factors, such as earthquake prediction, in the future.)

Common Situations
The following are typical scenarios and conditions that the Arbor2000 software is expected to handle.

Situation #1
There are sixteen sensor groups, each with three or four sensors, placed in the Rumbling Range National Forest. The sensors are randomly chosen from rainfall, temperature, sunlight, wind speed, wind direction, and snowpack sensors. The sensors report from once a minute to once a day and in a variety of units.

Jane Arden, a National Park Service Ranger, wants to post the fire danger results outside the Visitor Information Center, so she uses the Arbor2000 to examine the graphical view of fire danger in the forest. Overall, the fire danger is "moderate" with one area of "low danger + high uncertainty". Looking into the uncertain area, she finds that a number of the sensors have not reported for quite a while, leading to the uncertainty. Further investigation reveals that none of the sensors in group 2 and 4 have reported, and further checking shows that groups 2 and 4 are the only two which use the 555-3473 phone modem. She dispatches a repair crew to figure out the problem with the phone line while she posts the "moderate" fire danger sign in front of the visitor's center. She also checks the fire danger last year, and finds out that it was "low" over the entire forest, so she calls the Rumbling Range Spokesman-Review and asks them to print a story about how the fire danger is higher this year due to lower than expected rainfall.

Situation #2
The Rumbling Range National Forest buys two additional sensor arrays and hires a helicopter crew to plant them in the forest. After they return with Global Positioning System confirmation of the latitude and longitude of the sensors, Jane configures the system to receive the new data. Fortunately, the Arbor2000 is clever enough to store the unidentified incoming data until Jane had time to indicate where the arrays were located and what sensor types they were.

Situation #3
Forests 'R' Us comes out with a new plug-in module that it generously gives away free over the Internet. This new module computes trend analysis of the sunlight sensors to detect premature failure. Ms. Arden downloads and runs the module against the Rumbling Range database, only to discover that sensor #372 on Bald Mountain shows signs of age—its measured output has slowly declined over the past four years. Jane decides to hike to the top of the mountain and replace the sensor.

When she reaches the top, she discovers that the problem is not the sensor, but rather a small pine tree shielding the sensor from the sun. Unwilling to cut down the only tree on Bald Mountain, she relocates the sunlight sensor 100 meters to the south. When she returns to base, she updates the database with the sensor's new location.