Introduction to Object Oriented Programming 2E Timothy A. Budd

Chapter 2

Responsibility Driven Design

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Responsibility Driven Design (RDD)

Design technique that:

- Focuses on system behavior (i.e., actions) initially
- Can deal with ambiguous and incomplete specifications
- Flows from Analysis to Solution in a "natural" way
- Integrates various aspects of software development

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Directed Evolution

Take specifications as they occur in nature

- imprecise
- ambiguous
- unclear

Evolve the specification in concert with the design.

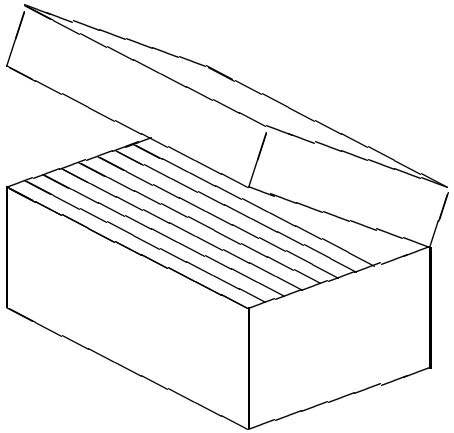
© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

The IIKH Example

Suppose you are the chief software architect in a major computing firm.

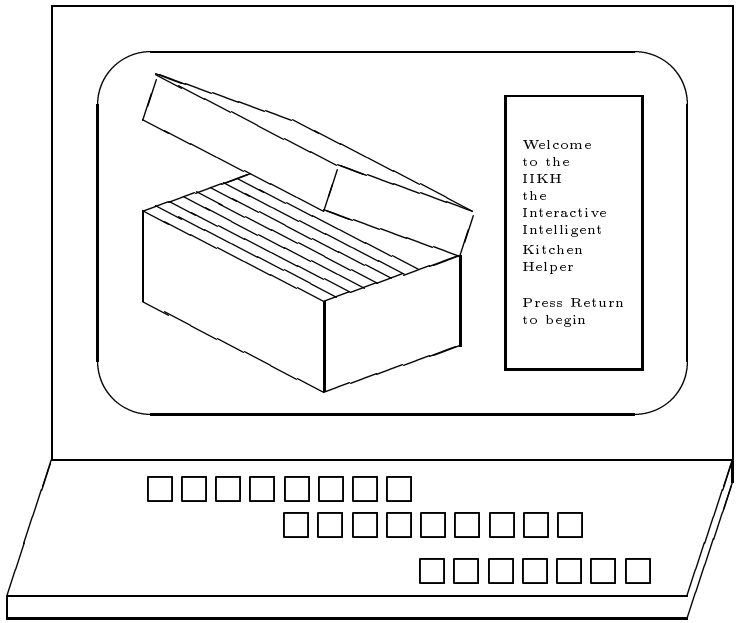
The president of the firm rushes into your office with a specification for the next PC-based product. It is drawn on the back of a dinner napkin.

Briefly, the *Intelligent Interactive Kitchen Helper* (IIKH) will replace the box of index cards of recipes in the average kitchen.



Your Job

Your job is to develop the software that will implement the IIKH.



© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Capabilities of the IIKH

With the IIKH a user can:

- Browse database of recipes
- Add new recipes to database
- Edit or annotate existing recipes
- Plan meals consisting of several courses
- Scale recipes for some number of servings
- Plan longer period, say a week
- Generate grocery list for all items in all menus for period

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Characterization by Behavior

The goal in using Responsibility Driven Design will be to first characterize the application by *behavior*:

1. Capture behavior of entire application
2. Refine into behavioral descriptions of subsystems
3. Refine behavioral descriptions into code

This is the fundamental aspect of Object-Oriented Programming.

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Working Through Scenarios

The specification may be ambiguous.

Walk through application scenarios to uncover the desired behavior:

- Pretend already have working application
- Walk through the various uses of system
- Establish "look and feel" of system
- Make sure all intended uses are revealed
- Develop descriptive documentation
- Create the high level software design

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Software Components

Software *component*: an abstract design entity with which we can associate responsibilities for different tasks

May eventually be a class, function, module, etc.

A component:

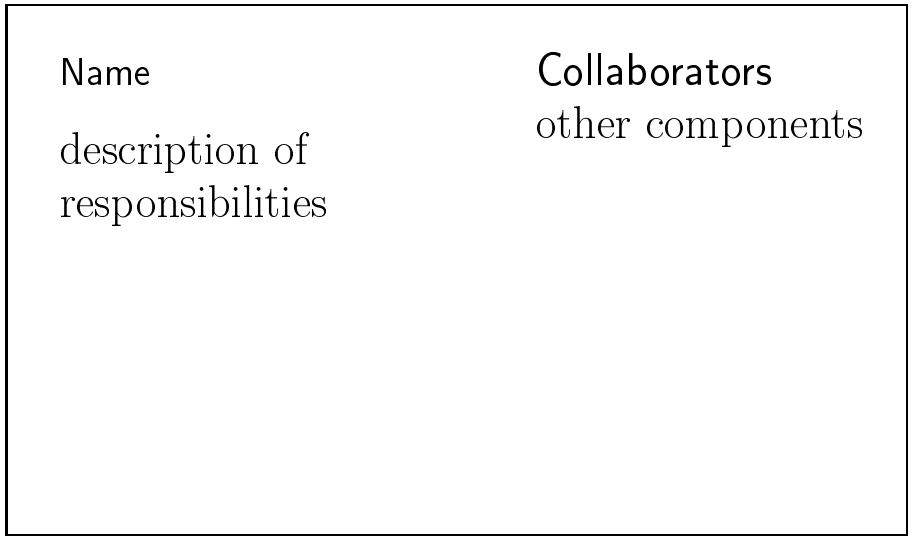
- Must have a small, well-defined, cohesive set of responsibilities
- Should interact with other components as little as possible (i.e., have minimal coupling)

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

CRC Cards

Components are easily described using CRC cards.

CRC – Component (name), Responsibilities, Collaborators



- Use index cards: inexpensive, erasable, physical, small

The first component, The Greeter

First component defined is the Greeter

Writes a welcome message on the screen at startup

Offers the user a choice of actions:

- Browse database of recipes
- Add a new recipe
- Edit or annotate a recipe
- Review a plan for several meals
- Create a plan of meals

Many details on how these are done can be ignored for now.

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

The Greeter CRC Card

Greeter	Collaborators
Display Informative Initial Message	Database Manager
Offer User Choice of Options	Planner
Pass Control to either	
Recipe Database Manager or	
Planner for processing	

The Recipe Database Component

Ignoring the planning of meals for now, your team next explores the recipe database component.

- Must maintain database of recipes
- Must permit *browsing* of database
- Must permit editing or annotating existing recipes
- Must permit adding new recipes

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Postponing Decisions

Many decisions (e.g., method of browsing) can be ignored for now – are encapsulated within the recipe database component – do not effect other components.

- Scroll bars and windows?
- Virtual "book" with thumb-holes and flipping pages?
- Keywords and phrases?

Only need to note that somehow user can manipulate database to select recipe

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Responsibilities of a Recipe

Make recipe itself an active data structure – maintains information, also performs tasks:

- Maintains list of ingredients and transformation algorithm
- Can edit these data values
- Can interactively display itself on screen
- Can print itself

We can extend to other actions later (ability to scale, produce grocery list, etc.)

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

The Planner Component

We return to the Greeter, start different scenario

Identify the Planner which:

- Permits selecting sequence of dates for planning
- Permits editing existing plan
- Associates with Date object

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

The Date Component

The Date component holds sequence of meals for sequence of dates

- Editing specific meals
- Annotating information about dates ("Conrad's Birthday", "Thanksgiving Dinner", etc.).
- Printing grocery list for entire set of meals

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

The Meal Component

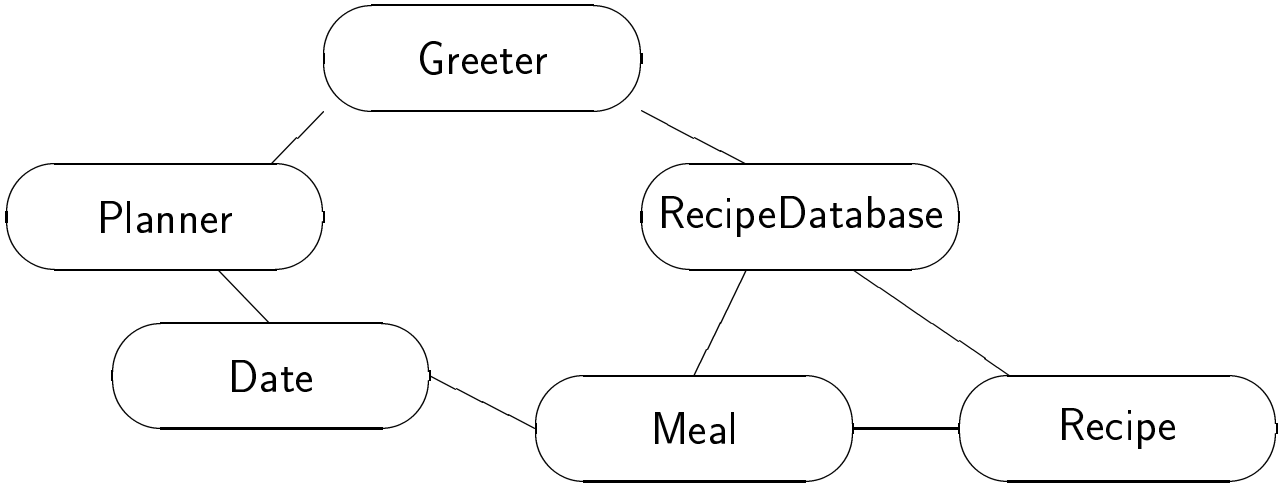
The Meal component holds information about a single meal

- Allows user to select individual recipes from database
- Automatically scales recipe to user-supplied number of servings
- Produces grocery list for meal, combining grocery lists from individual scaled recipes

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

The Six Components

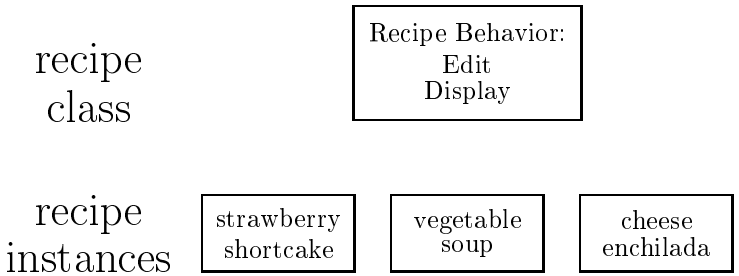
Having walked through the various scenarios, your team eventually decides everything can be accomplished using only six software components.



Instances and Classes

There are likely many *instances* of recipe, but they will all *behave* in the same way.

We say the behavior is common to the *class* of recipes.



Since earlier our goal was to identify behavior, we ignored this distinction and concentrated on prototypical objects.

Next Step – Formalize the Interface

Formalize the channels of communication between the components

- Identify general structure of each component
- Make components with only one behavior into functions
- Implement components with many behaviors as classes
- Name each responsibility (will eventually becomes a procedure)
- Assign information to each component
- Identify preconditions and postconditions for each responsibility and appropriate invariants for each class
- Replay scenarios to ensure all data is available

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Coming Up with Names

The selection of names is an important task. Give it sufficient time and thought.

- Names should be evocative in the context of the problem.
- Names should be short.
- Names should be pronounceable (read them out loud).
- Names should be consistent within the project.
- Names should not have multiple interpretations.
- Use capitalization and underscores within a name; avoid digits.
- Names of booleans should indicate meaning of true value.

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Documentation

In addition to CRC cards, begin developing other documentation almost at the beginning of the project.

The two most important documents are:

- the user manual
- the system design documentation

User Manual

The user manual describes the application as seen by the user.

- Does not depend upon implementation – can be developed before implementation
- Can naturally flow from walking through scenarios.
- Can be used to make sure the users and implementors have the same ideas

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Quality

Always remember: The primary measure of quality is the degree to which your customers (clients) are satisfied with your product.

Customers are often unsure of exactly what they want.

Thus you should work with the client early in the design phase to make sure you are developing is the "right" product.

Write the user manual before the software and share it with the clients.

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

System Design Documentation

Record the decisions made during system design.

- Record arguments for/against any major decision and the rationale for final choice
- Record CRC cards for the major components
- Maintain a log or diary of the process schedule
- Produce documentation as you make decisions – don't wait until major details are forgotten
- Do not rely exclusively on code – it records final outcomes, not factors leading up to decisions
- Keep documentation up-to-date and consistent as the product evolves

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Preparing for Change

Remember that change is inevitable: user requirements change with experience, host platforms change, user organizations change, government regulations change, want to expand to new markets (perhaps global), etc.

- Try to predict the most likely sources of change and isolate the effect. Common changes include interfaces, file formats, communication protocols
- Isolate interfaces to hardware and system software
- Reduce dependency of one software component on another
- Accurately document the rationale for major decisions

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Next Step – Select Representations for Subsystems

Select the internal representation for each component:

- Use knowledge of classic data structures of computing science
- Use knowledge of classic algorithms of computing science
- Use knowledge of available libraries

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Step – Implement and Test Subsystems

If subsystem large or complex, apply RDD recursively if needed.

For smaller or less complex subsystems, use classic techniques (e.g., *stepwise refinement*) to design and implement the internal representation.

As necessary, refine the preconditions, postconditions, and invariants for the subsystem.

Validate the subsystems in isolation:

- Informally prove that the components are correct – postconditions are established and invariants preserved, preconditions of called components are satisfied
- Make sure error conditions are handled safely
- Use software testing to build confidence and test pragmatic considerations
- Test subsystem using stubs for other subsystems (This is *unit testing*.)

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Step – Integration and Testing

Carefully and incrementally integrate components into completed system (i.e., *integration testing*):

- Use stubs for unimplemented components to perform testing throughout integration
- Test partially integrated system
- When errors are discovered during integration, make appropriate changes to subsystems, revalidate each, and repeat testing of partially integrated system
- Replace a stub by another completed subsystem and repeat integration testing process

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Maintenance and Evolution

Software does not remain fixed after the first working version is released.

- Errors or bugs can be discovered. Must be corrected.
- Requirements may change. Say as a result of government regulations, or standardization among similar products.
- Hardware or operating systems may change.
- Users expectations may change. Greater functionality, more features. Often as a result of competition from similar products.
- Better documentation may be required.

A good design recognizes the inevitability of change, and plans an accommodation for these activities from the very beginning.

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Common Design Flaws

Direct modification: Classes that directly modify data in other classes violate encapsulation.

Such coupling results in inflexible designs.

Too much responsibility: Classes with too much responsibility are difficult to understand and use.

Responsibility should be broken into smaller meaningful packages and distributed.

No responsibility: Classes with no responsibility serve no purpose.

This often arises when designers equate physical existence with logical design existence.

Classes with unused responsibility: Usually the result of designing software components without thinking about how they will be used.

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)

Misleading names: Names should be short and should unambiguously indicate which responsibility is involved.

Inappropriate inheritance: This occurs when subclassing is used in situations where the concepts do not share an *is-a* relationship.

© Timothy A. Budd 1996 (Modifications © H. Conrad Cunningham 1997)