Chapter 3: OO Design

Using an OO language is neither necessary nor sufficient

Responsibility-driven design

Responsibility/Noninterference

- Cannot be responsible with some independence or freedom of choice
- Weaker ties between system parts:
  - Better for programming in the large
  - Better support for reuse

Programming in the Small

- Code is developed by a single person/small team.
- A single person can understand all aspects of a project.
- Major problem: algorithm design and development

Programming in the Large

- Large team of programmers
- Designers may differ from implementors
- No single individual understands all aspects of the project
- Major problem:
  - Management of details
  - Communication between project groups

Why Behaviour?

- Easy to understand for both
  - The end-user
  - The programmer
- First aspect of a system that can be described naturally.
- Usually want a system that "does XYZ", I.e. exhibits some specific behavior

Case study in RDD: IIKH

- Develop the Intelligent Interactive Kitchen Helper:
  - Database of recipes
  - Assists menu-planning and shopping
- Initial specification: ambigous
- Try capturing behavior at a high level
- Can be done hierarchically

Working with components

- Initial specs are ambigous/unclear
- Understand "look and feel" better
- Early user feedback possible
- Final spec will change during the project
- High-level structuring decisions: "which components does what"

Identify components

- Like building a car: divide into smaller parts
- May become a class, method, pattern …
- Essential:
  - Must have a small well-defined set of responsibilities
  - Should interact with other components as little as possible

CRC cards

- Walk through scenarios
- Record results: Component, Responsibility, Collaborator (CRC) cards

Component Name	Collaborators
Responsibility description	List of other components

Physical representation

- Advantages of CRC cards:
  - Availability
  - Cheap
  - Throw away
  - Small
  - Limit complexity

What/Who cycle

- Find out WHAT needs to be done
- Then find out WHO (which component or agent) should do it

Documentation

- Two documents (already now):
  - User manual
  - System design docu
- Especially important: document high-level design decisions, both options considered and arguments pro and cons

Components and Behavior

- First have a "Greeter":

GREETER	
Display initial message	Colloborators:
Offer option choice	DB manager
Pass control on to:	Plan manager

- Main behaviours naturally grouped into 2 components:
  - recipe database
  - Meal planner

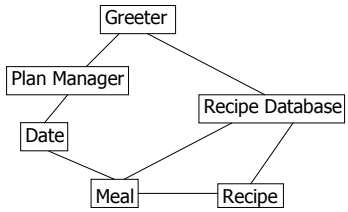
Postponing decisions

- If a decision is solely within a single component, delay it!
- E.g. how to support browsing:
  - List of categories
  - List of keywords
  - Search for full-text/keywords/…
  - Scroll-bars/thumb-nails/auto-completion
- NOT NECESSARY NOW, choose later

Preparing for change

- Change should affect few components
- Isolate effects of likely changes:
  - User interface
  - Communication formats
  - Output formats
- Isolate/reduce OS/hardware dependencies
- Reduce coupling between components
- Document design decisions

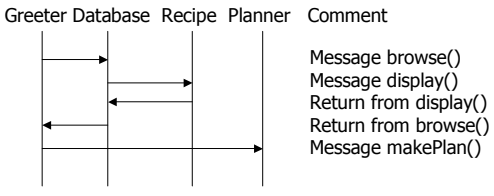
Continuing the scenario



- Components and comm-channels

Interaction Diagrams

- Makes time (flow) explicit:



Behavior and State

- Behavior: set of actions, also called protocol (e.g. editing, displaying, printing a recipe, …)
- State: information held within (e.g. recipe ingredients, preparation instruction); may change over time

Instances and Classes

- Class: describes a set of objects of similar behavior, I.e. behavior is associated with the class
- Instances: have their own, different data values ("state"), but perform the same actions

Coupling and Cohesion

- Cohesion: degree to which the responsibilities of a single component form a meaningful unit
- Coupling: describes the relationship between components
- Want:
  - High cohesion
  - Low coupling

Parnas's Principles

- Information hiding/encapsulation, distinguish:
  - What a component can perform (interface view)
  - How a component performs a task (implementation view)
- Component developer must provide the necessary information for a user, but no more.
- Developer must be provided with the necessary information for carrying out the responsibilities, but no more.

Formalize the interface

- Redo CRC cards into a more formal spec (method names, arguments, …)

DATE	
Maintain information about specific date	Collaborators
Date(year,month,day) – create new date	Plan Manager
DisplayAndEdit() – display date info in window allowing user to edit entries	Meal
BuildGroceryList(List) – add items from all menus to grocery list	

Naming

- Selecting good names is important, will be the vocabulary for your application
- Guidelines:
  - Use pronounceable names
  - Use caps/underscores: CardReader, card_reader
  - Abbreviate carefully: TermProcess?
  - Avoid ambiguity: empty?
  - Avoid digits inside names: 0/O, 1/l, 2/Z, 5/S
  - Booleans: PrinterIsReady better than PrinterStatus
  - Extra care for names of rare and costly operations

Designing the representation

- When designing a single component:
  - Choose algorithm to implement behavior
  - Choose appropriate data structure to hold state
- Wrong choice: complex, inefficient result
- Right choice: simple, efficient result

("elegance is not optional!")

Implementing components

- Straightforward given good design
- But often: notice that some behavior and/or state needs to be assigned yet another component:
  - Facilitator working "behind the scenes"
- Documentation, verification/testing important: are legal input values being dealt with appropriately?

Component integration

- Unit testing: single components
- Integration testing: several components together (may need "stubs")
- Regression testing: replay old test cases after any modifications

(more on testing in 315 next year)

Maintenance and Evolution

- Reasons / causes for change:
  - Bugs: patches, new releases
  - Requirement changes
  - Hardware changes
  - User expectation changes
  - Better documentation requested

Summary

- Emphasis on responsibility and behavior
- Key tool: scenarios
- Captured by: CRC cards
- Software life cycle:
  - Design & Implementation
  - Implementation broken into components
  - Maintenance: long-term