On to OO design ideas

Really just an introduction (much more in CS 48)

About “programming in the large”

BTW: Assuming no need to lecture about Abstraction chapter of Reading #2. Right?

Small vs. large programs

- Programming in the small:
  - Usually just one programmer
  - He/she understands everything from top to bottom
  - Major problems are in the development of algorithms
- Programming in the large:
  - System is developed by large team(s) of programmers
  - Major problems are in the management of details
  - Communication is vital – between programmers, and between their respective software subsystems

Basis for Design (early stages)

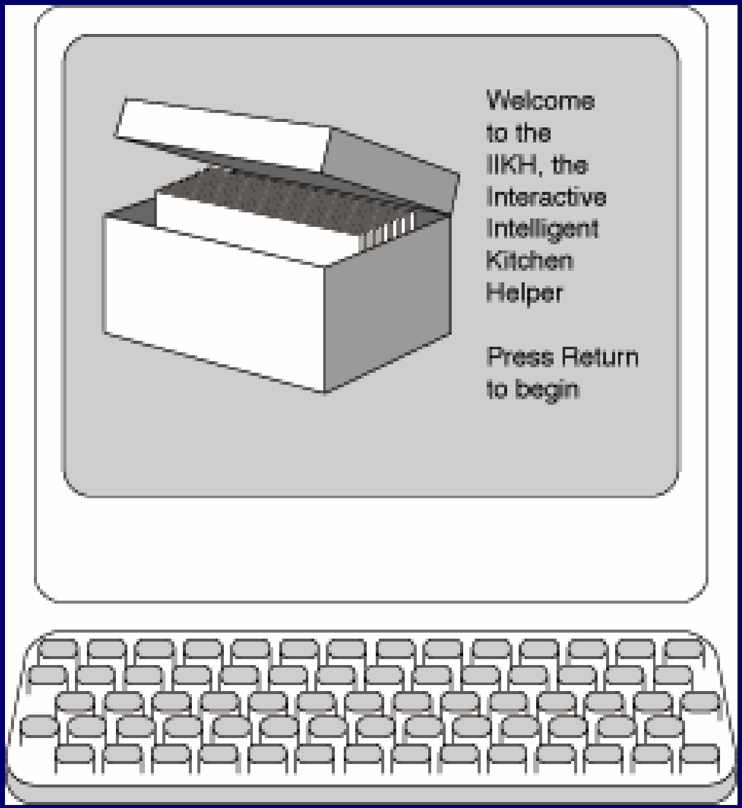
- Q. What aspects of a problem are known first?
  a) Data structures
  b) Functions
  c) Formal specifications
  d) Behavior
- A design technique based on *behavior* can be applied from the very beginning of a problem
  – Other aspects (the structural properties) necessarily require more preliminary analysis

Responsibility-Driven Design

- "Understanding responsibilities is key to good object-oriented design" (Martin Fowler)
- RDD concept: some object (and thus some class) must be responsible for every task that has to be accomplished by the system
- RDD is an Agile design technique
  - Accounts for ambiguous and incomplete specifications
  - Naturally flows from Analysis to Solution.
  - Easily integrates with various aspects of software development

Example: designing the Intelligent Interactive Kitchen Helper (IIKH)

- Imagine the boss rushes in with his specifications for your team's next project … carefully drawn on a napkin
- Briefly: the system is intended to replace that box of index cards of recipes in many kitchens



RDD activities – focus on behavior

- First identify and describe the behavior of the entire application
  - What the system must *do*
  - In what ways the system will interact with actors (users, other systems, …)
- Refine this overall behavior into behavioral descriptions for subsystems
- Translate the behavior descriptions into code

IIKH system behavior

- Browse a database of recipes
- Add a new recipe to the database
- Edit or annotate an existing recipe
- Plan a meal consisting of several courses
- Scale a recipe for some number of users
- Plan a longer period, say a week
- Generate a grocery list that includes all the items in all the menus for a period

Describing use cases

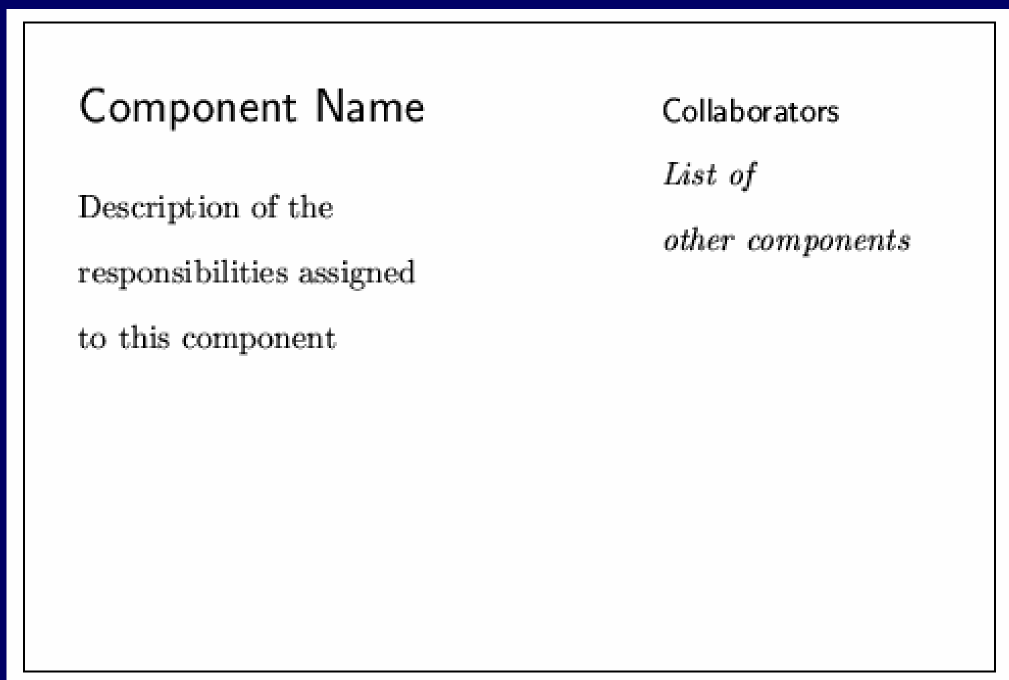
- Idea: Pretend we already had a working application - walk through the various uses of the system
- Use Case vs. Scenario:
  – A scenario is a specific use case instance
- Goal is to make sure we have uncovered all the intended uses of the system
- Also helps establish and comprehend the “look and feel” of the system

IIKH use cases?

Software components

- A software *component* is simply an abstract design entity with which we can associate responsibilities for different tasks
- May eventually be turned into a class, a function, a module, or something else
- Design principles:
  - A component must have a small, well-defined set of responsibilities
  - A component should interact with other components to the minimal extent possible

CRC cards



- Records name, responsibilities, and collaborators of a component
- Inexpensive
- Erasable
- Physical

What good are they?

Identifying components

- With OOP, mostly asking "What types of objects will make up the system?"
- Carefully study the problem (especially requirements and use cases) to find out
  - Candidate classes: *nouns* in the problem
    - Some are data – will be treated as class attributes
    - Most are participants in the solution – agents!
  - Operations: *verbs* in the problem

Component identification in RDD

- As we walk through scenarios, we go through cycles of identifying a what, followed by a who
  - *What* action needs to be performed at this moment?
  - *Who* is the component that is charged with performing the action?
- Every *what* must have a *who*, otherwise it simply will not happen.
- Postpone decisions about specific GUI details, algorithms, … – keep to *major* responsibilities

Identifying IIKH components

- The analysis team (author Budd …) decides the major responsibilities divide naturally into two groups
  - Recipe database – browsing, reviewing/editing recipes
  - Menu plans – creating/reviewing plans for meals
- Team also decides to include a component called a Greeter to present an attractive window, and allows the user to select from the various choices
  - Idea is that this component will pass on tasks to either a recipe database object or a menu planner object

Assigning responsibilities: Greeter

- Operations?
  - Greet user
  - Offer choices
  - Pass control
- Data?
- Collaborators?
  - Recipe Database
  - Planner

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

Recipe Database responsibilities

- Major responsibilities:
  - maintain the database of recipes
  - allow user to browse the database
  - permit user to edit or annotate existing recipes
  - permit the user to add a new recipe
- Who should be in charge of editing a recipe?
  - Clearly a job for a Recipe class. Okay add one!
  - Recipe becomes a collaborator of Recipe Database
- Postpone decisions about *how* user interacts, how to store recipes, and other implementation details

Responsibilities of a Recipe

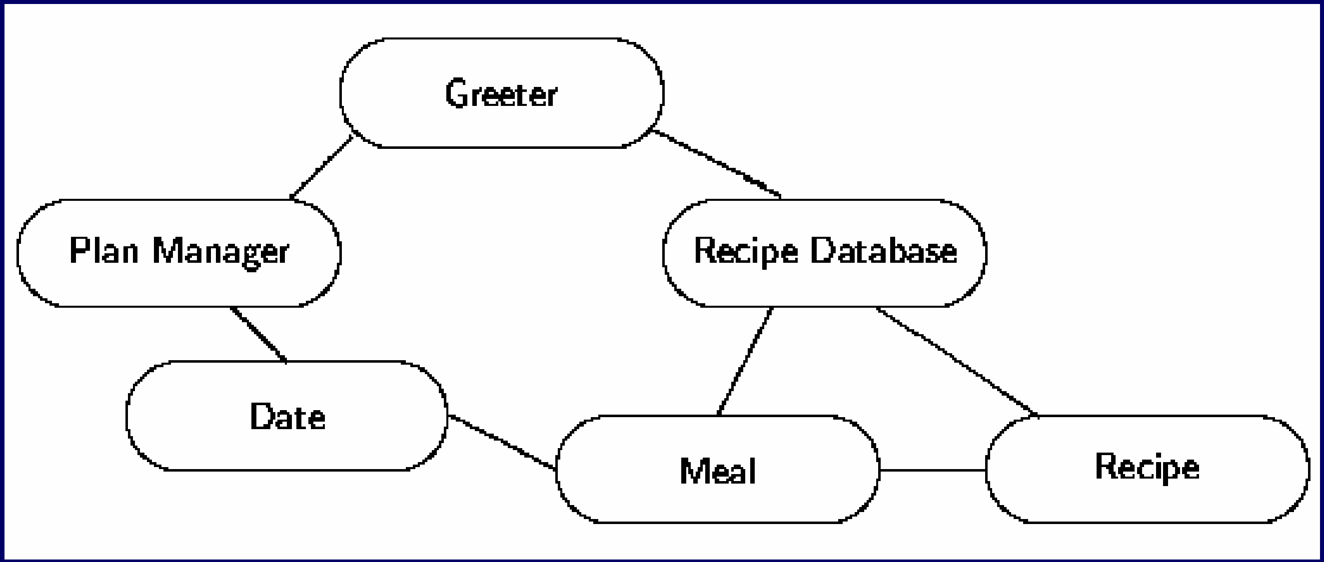
- Data: maintain list of ingredients and transformation algorithm
- Methods:
  - Ways to access and edit these data values
  - Maybe ways to display/print itself
  - Consider adding other actions later (ability to scale itself, integrate ingredients into a grocery list, and so on)
- Collaborators?

Meal planning sub-system

- Planner responsibilities:
  - Maintains a sequence of dates (for the user to plan)
    - Suggests collaboration with a *Date* object.
  - Let user select sequence of dates for planning
  - Let user create a plan or edit an existing plan
- Date responsibilities:
  - Holds a sequence of meals for a given date
    - Hmmm … probably will need *Meal* objects too!
  - Let user edit specific meals, annotate dates, print out grocery list for entire set of meals
- Meal responsibilities – data/operations for one meal

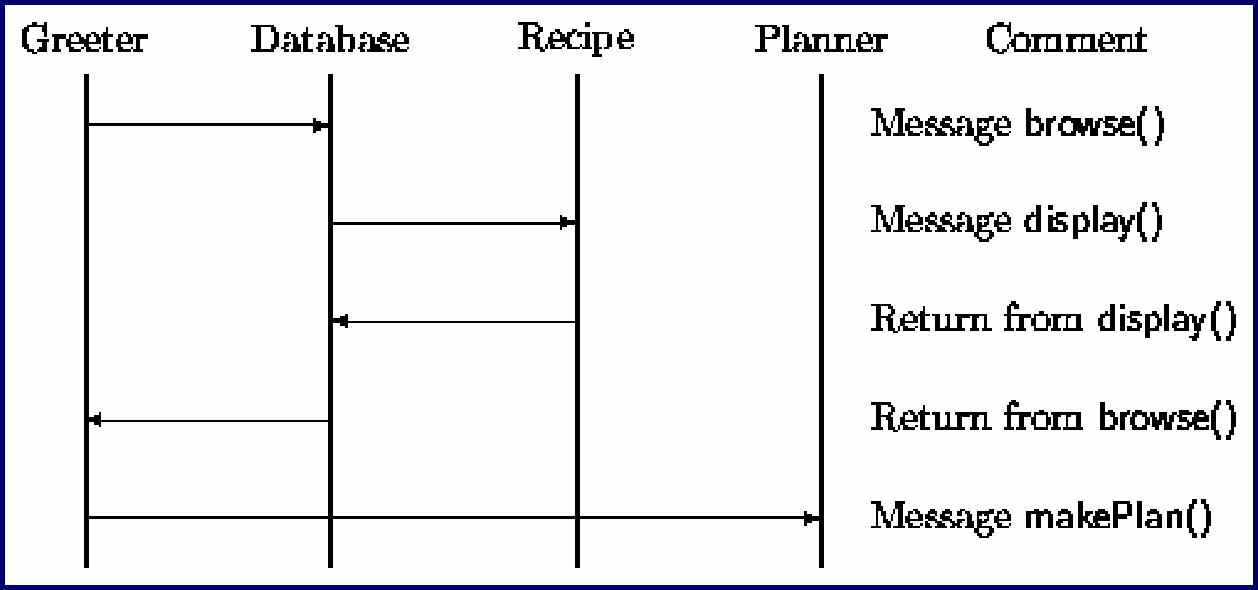
IIKH class associations

- Greeter uses 1 Plan Manager and 1 Recipe Database
- Recipe Database uses Meal and Recipe objects
- Plan Manager uses Date objects
- Date objects use Meal objects



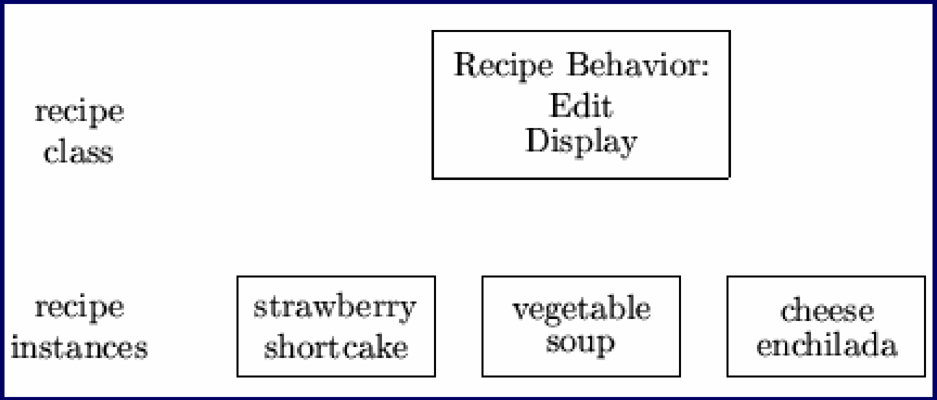
Modeling interactions

- Design *how* objects send messages to other objects while fulfilling their responsibilities
- Show messages in an interaction diagram



Behavior and state revisited

- All components are characterized by two aspects:
  - Behavior – the set of actions a component can do
  - State – all the information (data) a component holds
- Btw: it is common for behavior to change state
  - e.g., edit recipe → change preparation instructions
- Similarly: state will very likely affect behavior

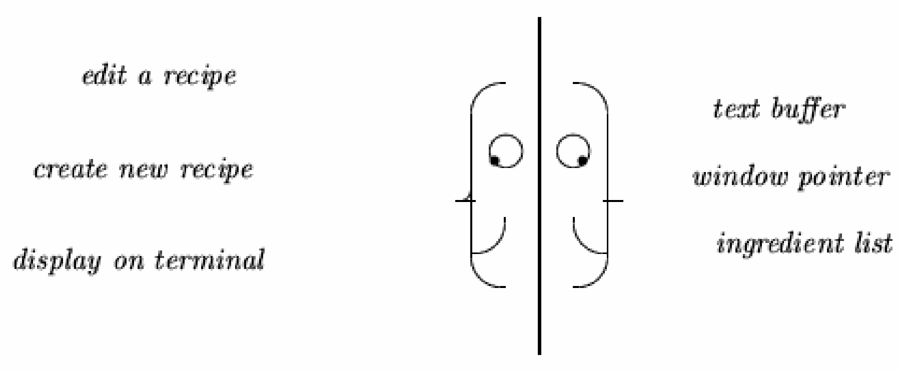


Two important design principles

- The separation of tasks into the domains of different components should be guided by the concepts of coupling and cohesion
- Cohesion is the degree to which the tasks assigned to a component seem to form a meaningful unit – should *maximize cohesion*
- Coupling is the degree to which the ability to fulfill responsibilities depends on the actions of other components – should *minimize coupling*

Interface vs. implementation

- Two views:
  - Client: public
  - Developer: private
- David Parnas:
  - The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide *no* other information.



Formalize component interfaces

- Names are given to each of the responsibilities – eventually probably mapped to procedure names
- Identify the general structure of each component
  - Information is assigned to each component and all information is accounted for
  - Components with only one behavior and no state to maintain may be made into functions
- Components with many behaviors are more properly implemented as classes
- Replay scenarios to ensure all data are available and all responsibilities are assigned

Selecting names is important

- Names should be evocative in the context of the problem – meaningful even to non-programmers
  - Nouns for classes, modules, variables
  - Verbs for operations
- Names should be short
- Names should be pronounceable (read out load)
- Names should be consistent within the project
  - Most critical for public parts though
- Avoid digits within a name – easy to misread

Detour back to shell – scripts

- In preparation for this week's lab
- Not covered in Reader (#1 just mentions)

Later: More OO design - classes.

Bourne shell programs

- Are text files with `sh` commands – e.g., `myScript`
  - To execute, can do `sh myScript`
    - The program runs in a new shell – called a child shell
  - Or `chmod u+x myScript` – then just `./myScript`
    - Requires that `sh` is the default shell (usually `bash` okay too)
- # – normally identifies a comment
  - Special case if line 1 – `#!/bin/sh` – identifies shell
    - Means use `sh` as child shell for this script – works in all shells
- Can access command line arguments: `$1` to `$#`
  - e.g., `cp $1 $2` # copies first to second (if files)
  - e.g., `echo $#` # prints number of arguments

sh variables and assignment

- `name="Jack Sprat"` # note *no spaces*
- `echo "The name is $name"` # need '$'
- `workdir=`pwd`` # use `…` to assign result of …
  - Similarly, `echo "date and time is `date`"`
- Can read from standard input and calculate too
  - `echo "enter value"`
  - `read val`
  - `doubleval=`expr $val + $val``
    - Or just: `echo "doubled: `expr $val + $val`"`

sh control structures, and FYIs

- An `if-then-elif-else-fi` statement
  - Expression is a test: `test $# -gt 0`
  - Or simpler: `[ $# -gt 0 ]` # spaces mandatory
  - Can test files too: `-d`, `-f`, `-e`, `-r`, `-w`, `-x`, …
- A `while-do-done` statement: same expressions
- A `for-do-done` statement: `for variable in list`
  - List is command line arguments if not specified
- Examples at ~`mikec/cs32/demo/scripts/`
- FYI: can program *any* shell, but different syntax
  - And other "scripting languages" (e.g., Perl, Python, …)