Information hiding

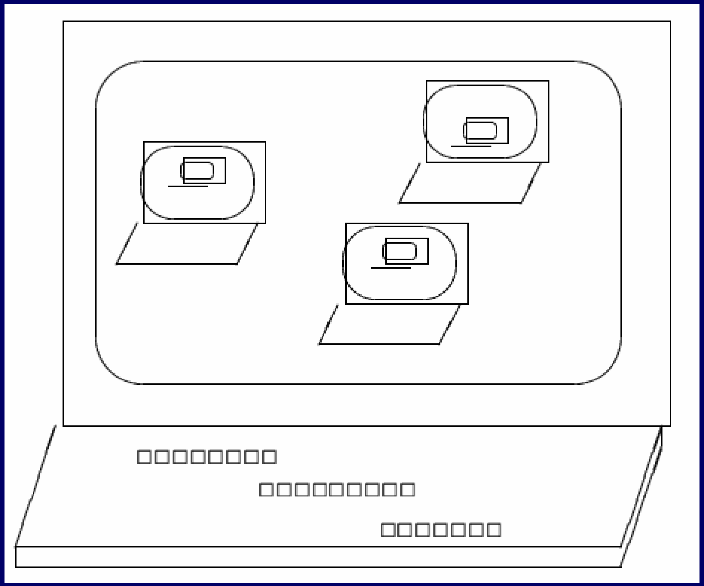
- Notice how a user of a service being provided by an object, need only know the name of the messages that the object will accept.
  - They need not have any idea how the actions performed in response to these requests will be carried out.
- Having accepted a message, an object is responsible for carrying it out.

Receivers and behavior

- Messages differ from traditional function calls in two very important respects:
  a) A designated *receiver* accepts the message
  b) The interpretation of the message may be different, depending upon the receiver
- Although different objects may accept the same message, the actions (*behavior*) the object will perform will likely be different
  – Might not even know what behavior to perform until run-time – a form of *late binding*

Elements of OOP – Recursive Design

- 3. Every object has its own memory, which consists of other objects.
  - The structure of the part mirrors the structure of the larger unit.
- Principle of non-interference: "Ask not what you can do *to* your data structures, but ask what your data structures can do *for* you." (Budd)

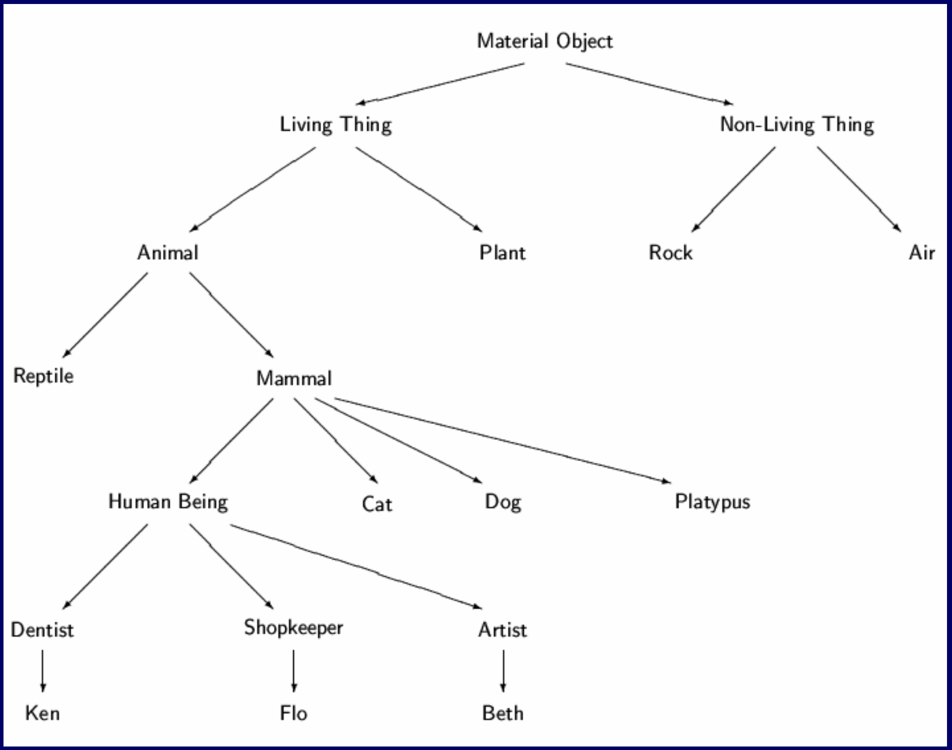


Elements of OOP - Classes

- 4. Every object is an instance of a class. A class groups similar objects.
  - Flo is an *instance* of the *class* Florist
- 5. The class is the repository for behavior associated with an object.
  - All objects that are instances of a class use the same method in response to similar messages.

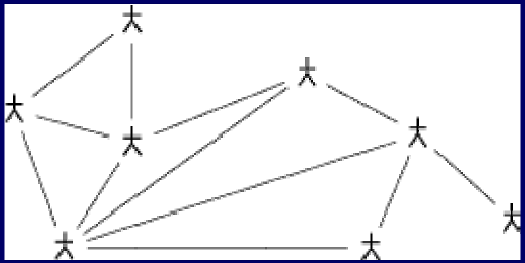
Elements of OOP - Inheritance

- 6. Classes are organized into a *singly-rooted* tree structure, called an inheritance hierarchy
- Data and *general* behavior at one abstraction level extend to lower levels
  - But can *override* behavior (a later topic)



Levels of abstraction 1

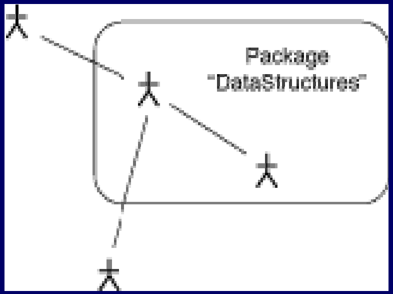
- Communities of interacting objects



  - Internally: within the program system
  - And externally: team of programmers, each responsible for different parts of the system
- Focus here is on *communication at the highest level* of abstraction
  - i.e., lines of communication between the agents

Packages and Namespaces

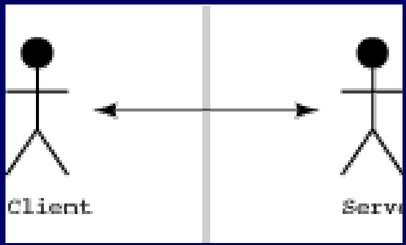
- Used to surround a collection of objects (a small community in itself) with a layer



- To control visibility from outside the module
  - A form of information hiding – promotes low coupling, and thus modifiability, reuse potential, and so on

Levels of abstraction 2

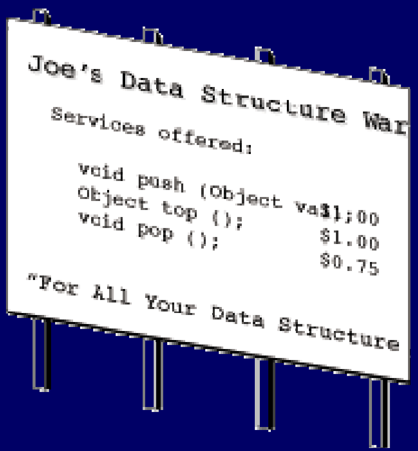
- Clients and servers – abstraction about the relationship between two individual objects



  - Typically one is providing a service, and the other is using the service
- Note: not specifically web servers/clients – a more general idea about interacting objects

Levels of abstraction 3, 4, …

- 3. Describing services
  - – Focus is on a server
  - – Independent of clients
  - – i.e., defining the *interface*



- 4. Implementing the interface – from point of serving the client(s)
- … Implementing individual functions, and other background features about which the clients have no need to know

Finding the right abstraction level

- A critical problem to solve in early stages of development – not easy, and no “right way”
  - Must determine what details are appropriate at each level of abstraction
  - And (often more importantly) must decide what details should be omitted – to be considered later
- Don’t want to ignore important information
  - But don’t want to manage too much information, or have excessive information hide critical details

On to OO design ideas

Really just an introduction (much more in CS 48)

About “programming in the large”

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)

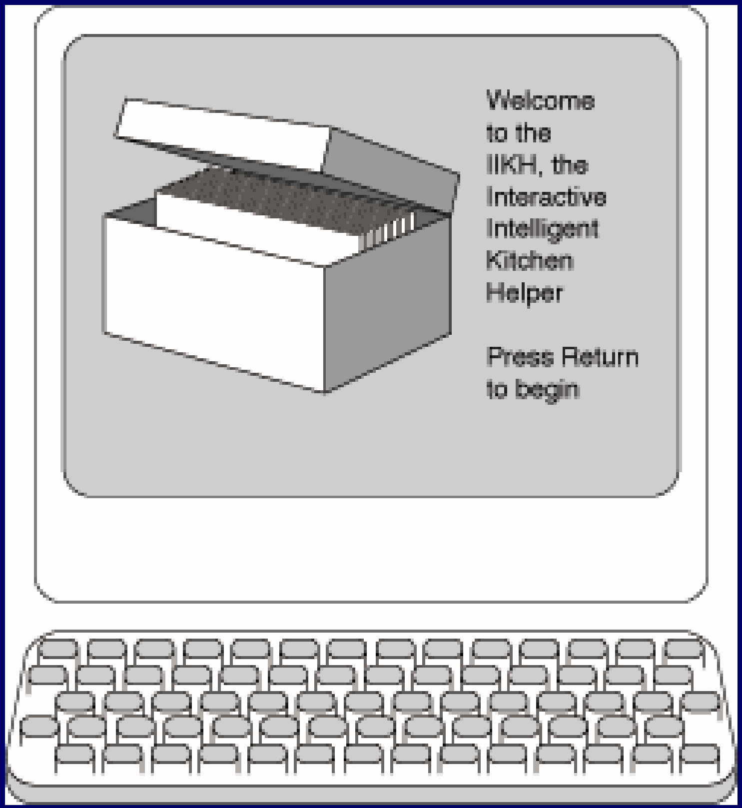
- Q. What aspects of a problem are known first?
  a) Data structures
  b) Functions
  c) Formal specifications
  d) Behavior
- A design technique based on *behavior* can be applied from the very beginning of a problem
  – Other aspects (the structural properties) necessarily require more preliminary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

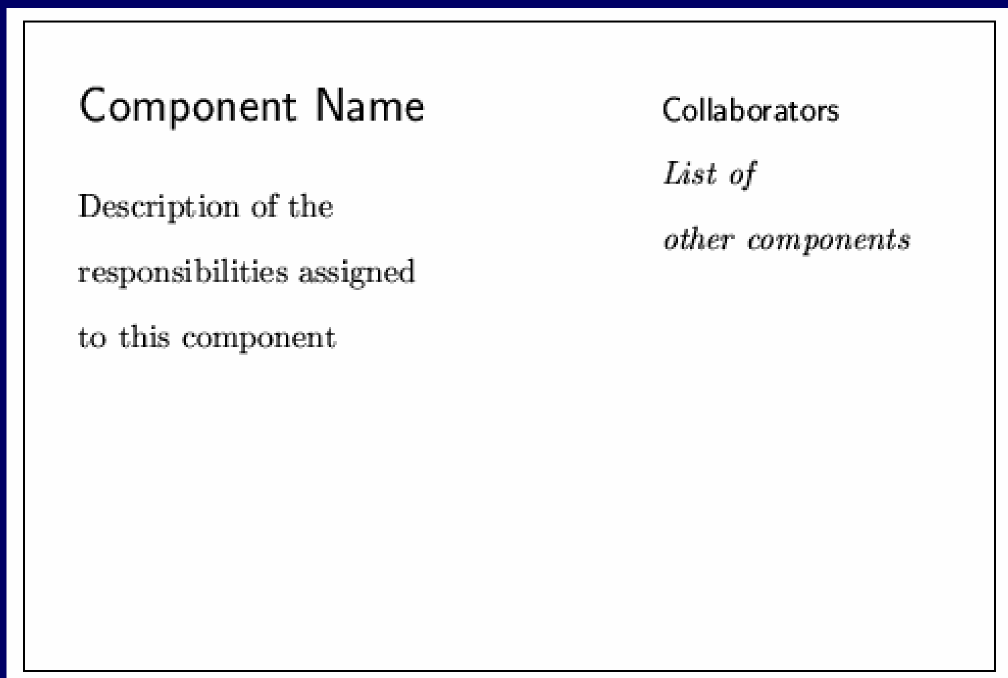
- Idea: Pretend we already had a working application - walk through the various uses of the system
- Use Case vs. Scenario:
  - A scenario is a specific use case instance
- Goal is to make sure we have uncovered all the intended uses of the system
- Also helps establish and comprehend the "look and feel"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

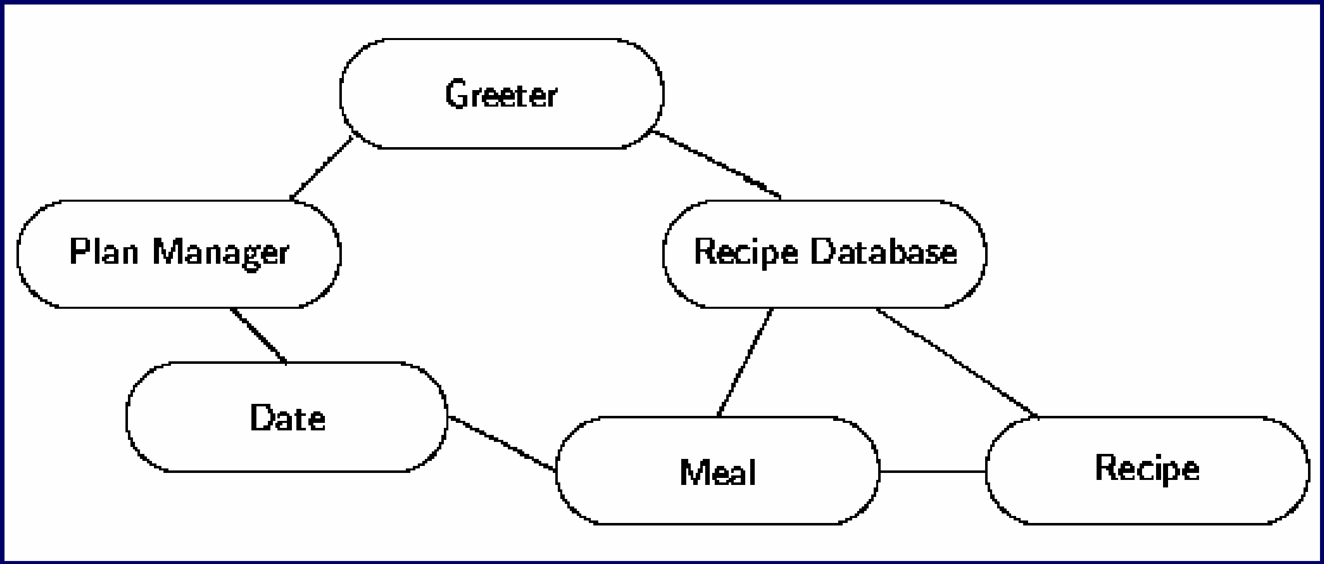
- Data: maintain list of ingredients and transformation algorithm
- Methods:
  - Ways to access and edit these data values
  - Maybe ways to display/print itself
  - Consider adding other actions later (ability to scale itself, integrate ingredients into a grocery list, and so on)
- Collaborators?

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

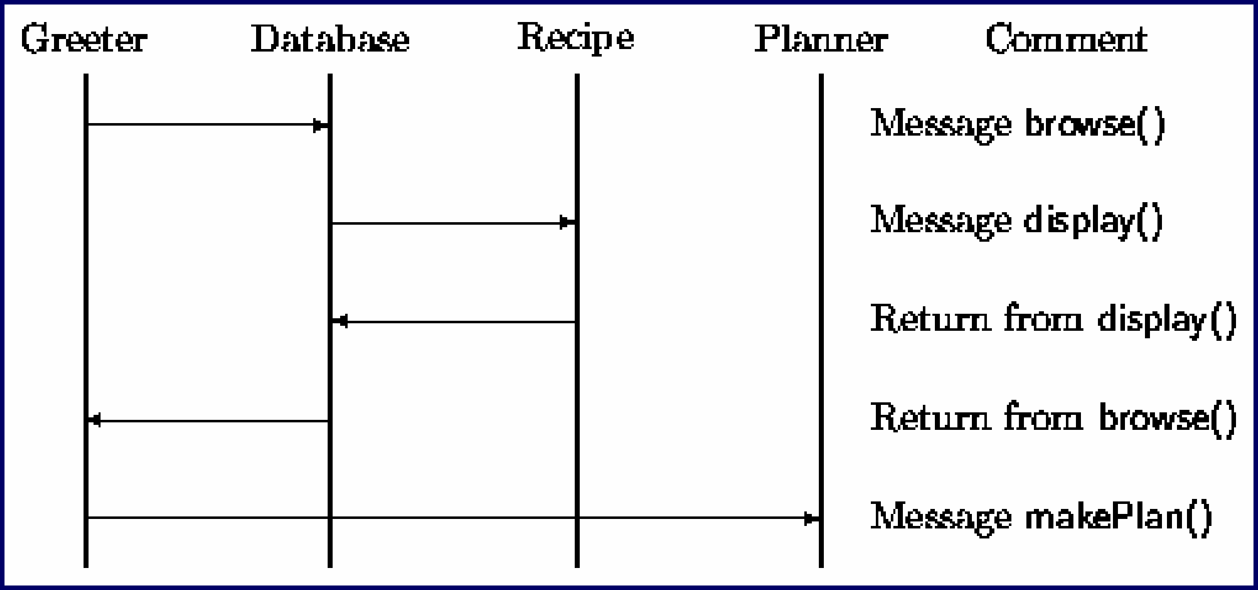
IIKH class associations

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



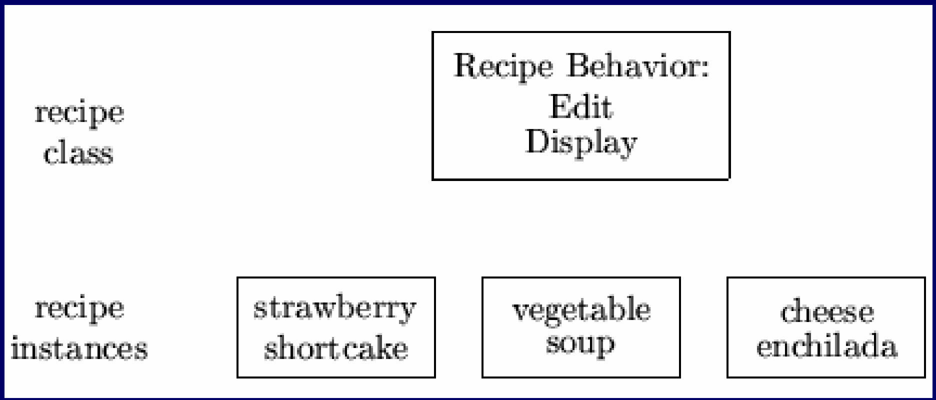
Modeling interactions

- 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

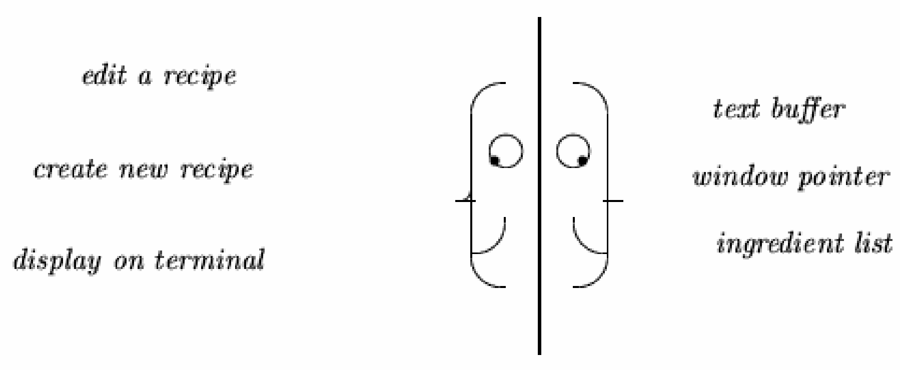


Two important design principles

- 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