7. OBJECT-ORIENTED DESIGN

Responsibilities Implies Noninterference

- When you make an object (be it a child or a software system) responsible for specific actions, you expect a certain behavior, at least when the rules are observed. But just as important, responsibility implies a degree of independence or noninterference. If you tell a child that she is responsible for cleaning her room, you do not normally stand over her and watch while that task is being performed--that is not the nature of responsibility.
- Instead, you expect that, having issued a directive in the correct fashion, the desired outcome will be produced.
- When Chris gave the request to the Florist to deliver flowers to Robin, it was not necessary to think about how the request would be serviced. The florist, having taken on the responsibility for this service, is free to operate without interference on the part of the customer Chris.
- The difference between conventional programming and object-oriented programming is in many ways the difference between actively supervising a child while she performs a task and delegating to the child responsibility for that performance.

Responsibilities Implies Noninterference cont.

- Conventional programming proceeds largely by doing something to something else--modifying a record or updating an array, for example.
- Thus, one portion of code in a software system is often intimately tied, by control and data connections, to many other sections of the system.
- Such dependencies can come about using global variables, through use of pointer values, or simply through inappropriate use of and dependence on implementation details of other portions of code.
- A responsibility-driven design attempts to cut these links, or at least make them as unobtrusive as possible.
- Responsibility-driven design elevates information hiding from a technique to an art. This principle of information hiding becomes vitally important when one moves from programming in the small to programming in the large.
- Major benefits of OOP can be seen when using a subsystem from one projects to the other.

Programming in the Small and in the Large

- Programming in the small characterizes projects with the following attributes:
  - ✓ Code is developed by a single programmer, or perhaps by a very small collection of programmers. A single individual can understand all aspects of a project, from top to bottom, beginning to end.
  - ✓ The major problem in the software development process is the design and development of algorithms for dealing with the problem at hand.
- Programming in the large, on the other hand, characterizes software projects with features such as the following:
  - ✓ The software system is developed by a large team, often consisting of people with many different skills. There may be graphic artists, design experts, as well as programmers. Individuals involved in the specific cation or design of the system may differ from those involved in the coding of individual components, who may differ as well from those involved in the integration of various components in the final product. No single individual can be considered responsible for the entire project, or even necessarily understands all aspects of the project.
  - ✓ The major problem in the software development process is the management of details and the communication of information between diverse portions of the project.

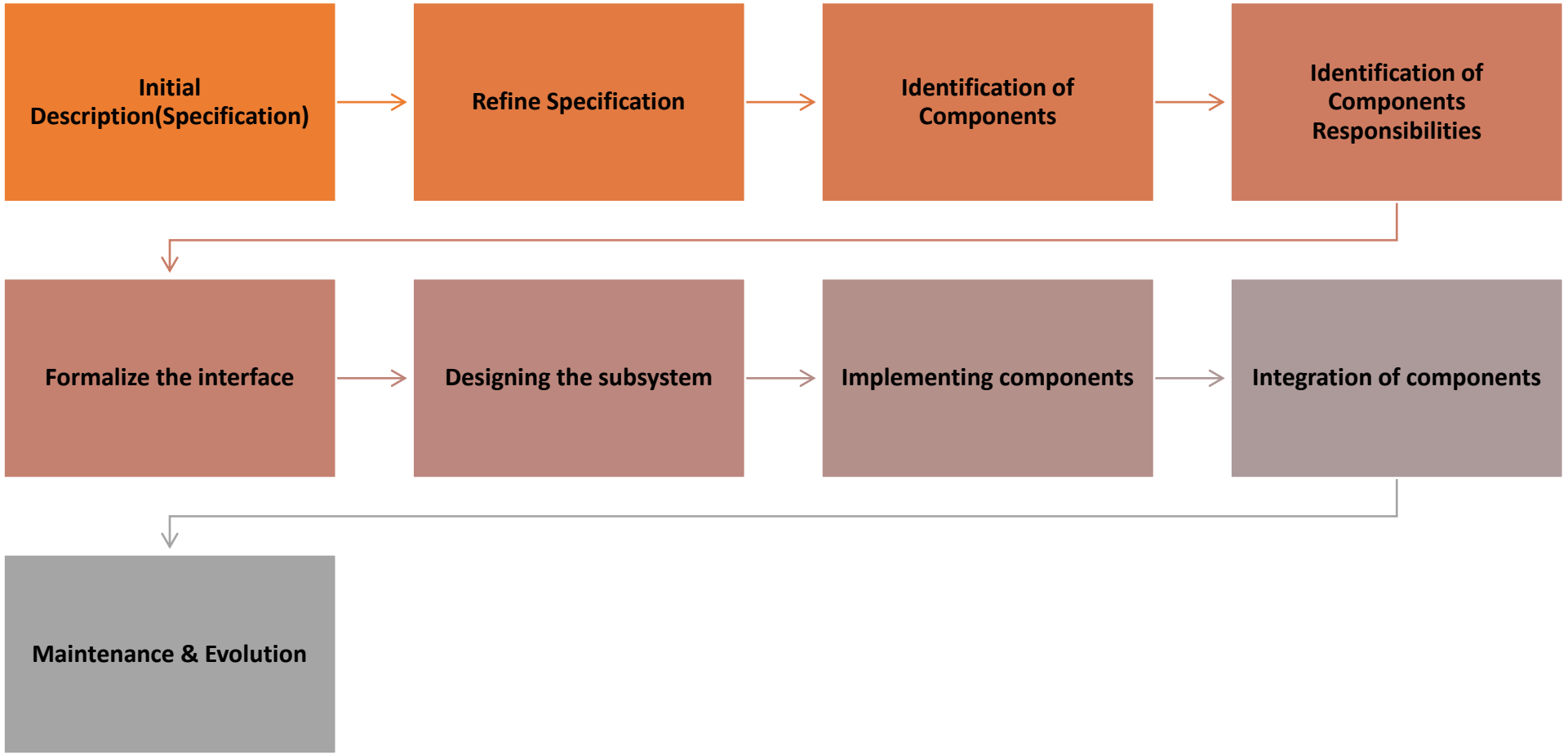
Why Begin with Behavior?

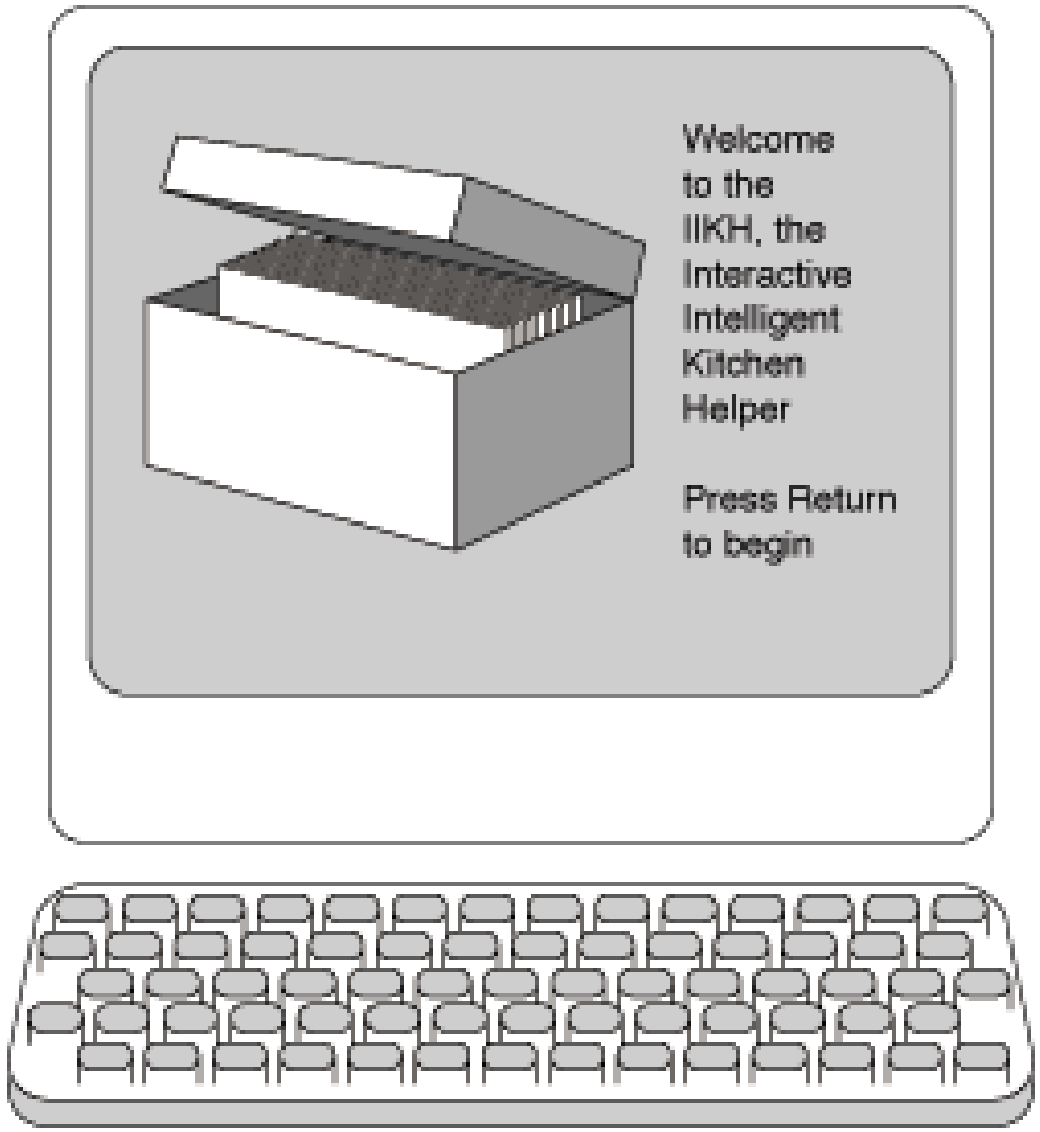
- Why begin the design process with an analysis of behavior? The simple answer is that the behavior of a system is usually understood long before any other aspect.
- Earlier software development methodologies concentrated on ideas such as characterizing the basic data structures or the overall structure of function calls, often within the creation of a formal specification of the desired application.
- But structural elements of the application can be identified only after a considerable amount of problem analysis.
- Similarly, a formal specification often ended up as a document understood by neither programmer nor client. But behavior is something that can be described almost from the moment an idea is conceived and can be described in terms meaningful to both the programmers and the client.
- Responsibility-Driven Design (RDD), developed by Rebecca Wirfs-Brock, is an object-oriented design technique that is driven by an emphasis on behavior at all levels of development. It is but one of many alternative object-oriented design techniques.

Responsibility Driven Design (RDD)

- RDD is developed by Rebecca Wirfs-Brock.
- A design technique that has the following properties:
  - ✓ Can deal with ambiguous and incomplete specifications.
  - ✓ Naturally flows from Analysis to Solution.
  - ✓ Easily integrates with various aspects of software development.

Development Process





An Example: *IIKH(Intelligent Interactive Kitchen Helper)*

- Imagine you are the chief software architect in a major computing firm.
- The president of the firm rushes into your office with a specification for IIKH, the next PC-based product. It is drawn on the back of a dinner napkin.

1. Initial Description

- Briefly, IIKH is a PC-based application that will replace the box of index cards of recipes in the average kitchen. But more than simply maintaining a database of recipes, the kitchen helper assists in the planning of meals for an extended period, say a week.
- The user of the IIKH can sit down at a terminal, browse the database of recipes, and interactively create a series of menus. The IIKH will automatically scale the recipes to any number of servings and will print out menus for the entire week, for a particular day, or for a particular meal. And it will print an integrated grocery list of all the items needed for the recipes for the entire period.
- Initial description : ***Ambiguous, Unclear***
- We need to clarify the ambiguities in the description.

2. Refine Specification (Working Through *Scenarios*)

- Because of the ambiguity in the specification, the major tool we will use to uncover the desired behavior is to walk through application *scenarios*.
- Pretend we had already a working application. Walk through the various uses of the system.
- Establish the ``*look and feel*'' of the system.
- Make sure we have uncovered all the intended uses
- Develop descriptive documentation
- Create the high-level software design
- Other authors use the term ``use-cases'' for this process of developing scenarios

Scenario Example

Simple Browsing

Alice Smith sits down at her computer and starts the IIKH. When the program begins, it displays a graphical image of a recipe box, and identifies itself as the IIKH, product of IIKH incorporated. Alice presses the return button to begin.

In response to the key press, Alice is given a choice of a number of options. She elects to browse the recipe index, looking for a recipe for Salmon that she wishes to prepare for dinner the next day. She enters the keyword Salmon, and is shown in response a list of various recipes. She remembers seeing an interesting recipe that used dill-weed as a flavoring. She refines the search, entering the words Salmon and dill-weed. This narrows the search to two recipes.

She selects the first. This brings up a new window in which an attractive picture of the finished dish is displayed, along with the list of ingredients, preparation steps, and expected preparation time. After examining the recipe, Alice decides it is not the recipe she had in mind. She returns to the search result page, and selects the second alternative.

Examining this dish, Alice decides this is the one she had in mind. She requests a printing of the recipe, and the output is spooled to her printer. Alice selects "quit" from a program menu, and the application quits.

Abilities of the IIKH

- Here are some of the things a user can do with the IIKH
  - 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

3. Identification of Components

- The engineering of software is simplified by the identification and development of software components.
- A component is simply an abstract entity that can perform tasks--that is, fulfill some responsibilities. At this point, it is not necessary to know exactly the eventual representation for a component or how a component will perform a task.
- A component may ultimately be turned into a function, a structure or class, or a collection of other components.
- At this level of development there are just two important characteristics:
  - ✓*A component must have a small well-defined set of responsibilities.*
  - ✓*A component should interact with other components to the minimal extent possible.*

4. Identification of Components Responsibilities

- Components are most easily described using CRC cards. A CRC card records the name, responsibilities, and collaborators of a component.

Component Name	Collaborators
Description of the responsibilities assigned to this component	*List of other components*

Fig: CRC card Template

- It is often useful to represent components using small index cards.

CRC Card

- Written on the face of the card is the name of the software component, the responsibilities of the component, and the names of other components with which the component must interact. Such cards are sometimes known as CRC (Component, Responsibility, Collaborator) cards, and are associated with each software component. As responsibilities for the component are discovered, they are recorded on the face of the CRC card.
- While working through scenarios, it is useful to assign CRC cards to different members of the design team.
- The member holding the card representing a component records the responsibilities of the associated software component, and acts as the “surrogate” for the software during the scenario simulation.
- He or she describes the activities of the software system, passing “control" to another member when the software system requires the services of another component.
- An advantage of CRC cards is that they are widely available, inexpensive, and erasable. This encourages experimentation, since alternative designs can be tried, explored, or abandoned with little investment.

CRC Card

- The physical separation of the cards encourages an intuitive understanding of the importance of the logical separation of the various components, helping to emphasize the cohesion and coupling.
- The constraints of an index card are also a good measure of approximate complexity{a component that is expected to perform more tasks that can easily in this space is probably too complex, and the team should and a simpler solution, perhaps by moving some responsibilities elsewhere to divide a task between two or more new components.
- The identification of components takes place during the process of imagining the execution of a working system.
- Often this proceeds as a cycle of *what/who* questions. First, the design team identifies what activity needs to be performed next. This is immediately followed by answering the question of who performs the action.
- In this manner, designing a software system is much like organizing a collection of people, such as a club. Any activity that is to be performed must be assigned as a responsibility to some component.

The Greeter

- Let us return to the development of the IIKH. The first component your team defines is the Greeter. When the application is started, the greeter puts an informative and friendly welcome window on the screen.
- Offer the user the choice of several different actions
  - Casually browse the database of recipes.
  - Add a new recipe.
  - Edit or annotate a recipe.
  - Review a plan for several meals.
  - Create a plan of meals.

The Greeter CRC Card

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	

The Recipe Database

- Must Maintain the Database of Recipes.
- Must Allow the user to browse the database.
- Must permit the user to edit or annotate an existing recipe.
- Must permit the user to add a new recipe.

The Recipe

- Maintains the list of ingredients and transformation algorithm.
- Must know how to edit these data values.
- Must know how to interactively display itself on the output device.
- Must know how to print itself.
- We will add other actions later (ability to scale itself, produce integrate ingredients into a grocery list, and so on).

The Planner

- Permits the user to select a sequence of dates for planning.
- Permits the user to edit an existing plan.
- Associates with Date object.

The Date

- User can edit specific meals.
- User can annotate information about dates (''Bob's Birthday'', ``Christmas Dinner'', and so on).
- Can print out grocery list for entire set of meals.

The Meal

- Allows user to interact with the recipe database to select individual recipes for meals.
- User sets number of people to be present at meal, recipes are automatically scaled.
- Can produce grocery list for entire meal, by combining grocery lists from individual scaled recipes.

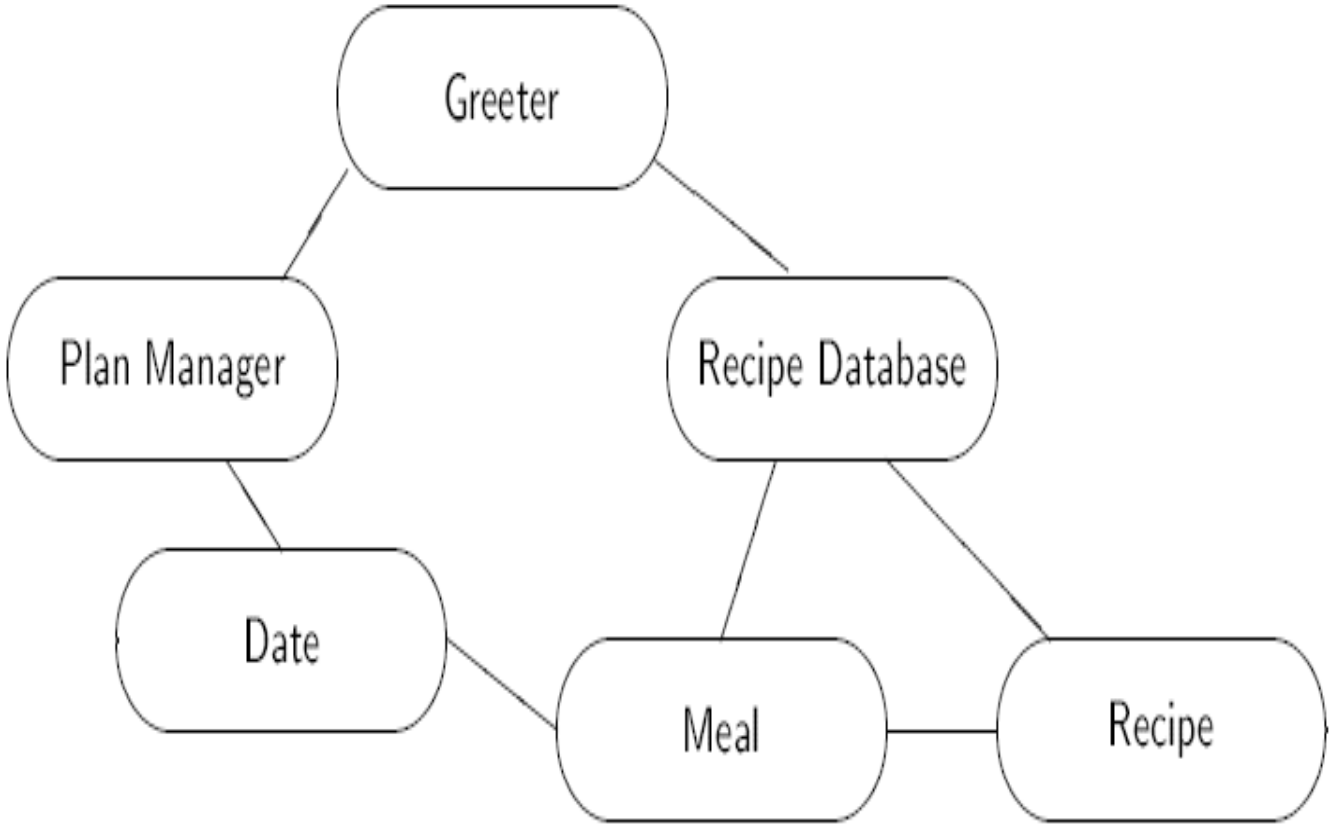


Fig: Communication between the six components in the IIKH.

The Six Components

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

Interaction/ Sequence Diagram

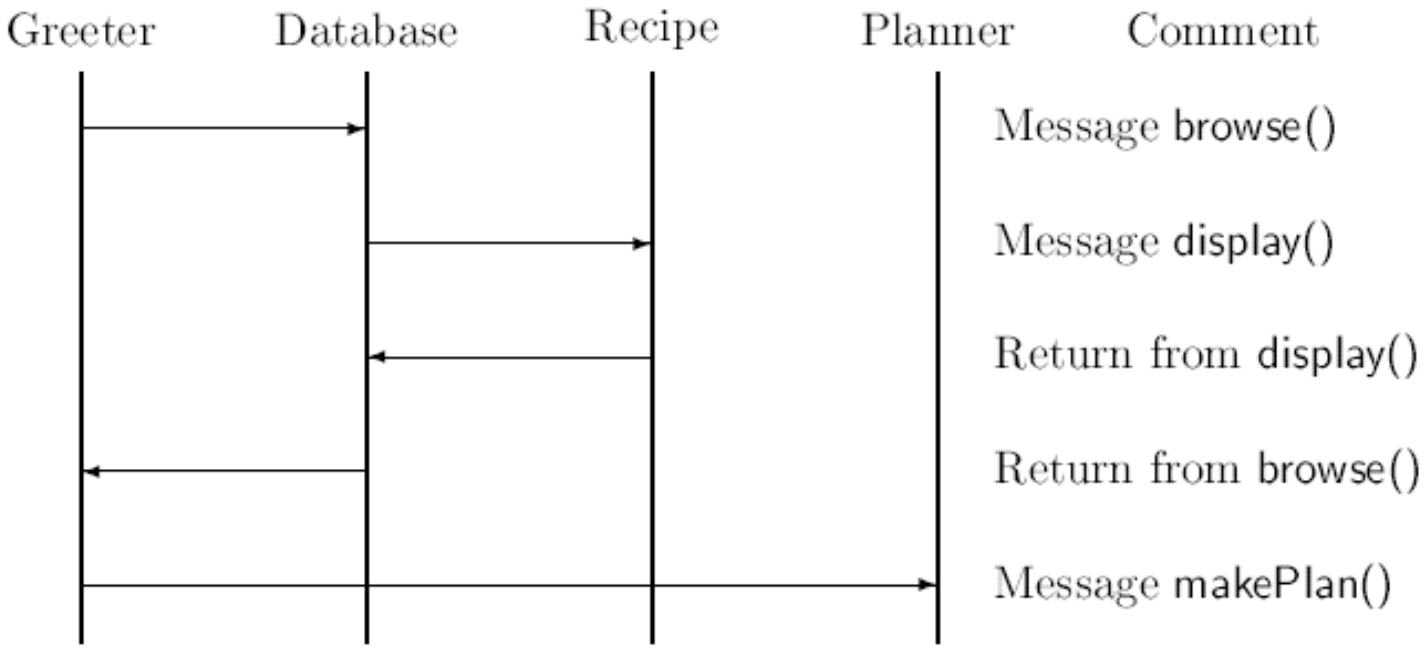


Figure: An Example interaction diagram.

While a description shown in the earlier Figure (Communication between the six components in the IIKH) may describe the static relationships between components, it is not very good for describing their dynamic interactions during the execution of a scenario. A better tool for this purpose is an *interaction diagram*.

Interaction/ Sequence Diagram cont.

- Figure shows the beginning of an interaction diagram for the interactive kitchen helper.
- Time moves forward from the top to the bottom.
- Each component is represented by a labeled vertical line.
- A component sending a message to another component is represented by a horizontal arrow from one line to another. Similarly, a component returning control and perhaps a result value back to the caller is represented by an arrow. (Some authors use two different arrow forms, such as a solid line to represent message passing and a dashed line to represent returning control.)
- The commentary on the right side of the figure explains more fully the interaction taking place.
- With a time axis, the interaction diagram can describe better the sequencing of events during a scenario. For this reason, interaction diagrams can be a useful *documentation tool* for complex software systems.

Software Components

- Behavior and State
- Instances and Classes
- Coupling and Cohesion
- Interface and Implementation

Behavior and State

- All components can be characterized by two aspects:
  - ✓ The *behavior* of a component is the set of actions a component can perform. The complete set of behavior for a component is sometimes called the protocol. For the Recipe component this includes activities such as editing the preparation instructions, displaying the recipe on a terminal screen, or printing a copy of the recipe.
  - ✓ The *state* of a component represents all the information held within it at a given point of time. For our Recipe component the state includes the ingredients and preparation instructions. Notice that the state is not static and can change over time. For example, by editing a recipe (a behavior) the user can make changes to the preparation instructions (part of the state).
- It is not necessary that all components maintain state information. For example, it is possible that the Greeter component will not have any state since it does not need to remember any information during execution. However, most components will consist of a combination of behavior and state.

Instances and Classes

- The term class is used to describe a set of objects with similar behavior.
- An individual representative of a class is known as an instance.
- Note that behavior is associated with a class, not with an individual. That is, all instances of a class will respond to the same instructions and perform in a similar manner.
- On the other hand, state is a property of an individual.
- We see this in the various instances of the class Recipe. They can all perform the same actions (editing, displaying, printing) but use different data values.

Coupling and Cohesion

- Cohesion is the degree to which the responsibilities of a single component form a meaningful unit.
- High cohesion is achieved by associating in a single component tasks that are related in some manner.
- Probably the most frequent way in which tasks are related is through the necessity to access a common data value.
- Coupling describes the relationship between software components.
- In general, it is desirable to reduce the amount of coupling as much as possible, since connections between software components inhibit ease of development, modification, or reuse.
- coupling is increased when one software component must access data values--the state--held by another component.
- Such situations should almost always be avoided in favor of moving a task into the list of responsibilities of the component that holds the necessary data.

Interface and Implementation

- It is possible for one programmer to know how to use a component developed by another programmer, without needing to know how the component is implemented.
- The purposeful omission of implementation details behind a simple interface is known as information hiding.
- We say the component encapsulates the behavior, showing only how the component can be used, not the detailed actions it performs.
- This naturally leads to two different views of a software system. The interface view is the face seen by other programmers. It describes what a software component can perform.
- The implementation view is the face seen by the programmer working on a particular component. It describes how a component goes about completing a task.
- The computer scientist David Parnas in a pair of rules, known as ***Parnas's principles***:

✓ *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.*

✓ *The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component and should be provided with no other information.*

- A consequence of the separation of interface from implementation is that a programmer can experiment with several different implementations of the same structure without a effecting other software components.

5. Formalize the Interface

- Formalize the channels of communication between the components
  - The general structure of each component is identified.
  - Components with only one behavior may be made into functions.
  - Components with many behaviors are probably more easily implemented as classes.
  - Names are given to each of the responsibilities - these will eventually be mapped on to procedure names.
  - Function parameters should be identified.
  - Information is assigned to each component and accounted for.
  - Scenarios are replayed in order to ensure all data is available.

Coming up with Names

- Use pronounceable names. As a rule of thumb, if you cannot read a name out loud, it is not a good one.
- Use capitalization (or underscores) to mark the beginning of a new word within a name, such as “CardReader" or “Card reader," rather than the less readable “cardreader."
- Examine abbreviations carefully. An abbreviation that is clear to one person may be confusing to the next. Is a “TermProcess" a terminal process, something that terminates processes, or a process associated with a terminal?
- Avoid names with several interpretations. Does the empty function tell whether something is empty, or empty the values from the object?
- Avoid digits within a name. They are easy to misread as letters (0 as O, 1 as l, 2 as Z, 5 as S).
- Name functions and variables that yield Boolean values, so they describe clearly the interpretation of a true or false value. For example, “PrinterIsReady" clearly indicates that a true value means the printer is working, whereas “PrinterStatus" is much less precise.
- Take extra care in the selection of names for operations that are infrequently used. By doing so, errors caused by using the wrong function can be avoided.

6. Designing the Representation

- The task now is to transform the description of a component into a software system implementation.
- The major portion of this process is designing the data structures that will be used by each subsystem to maintain the state information required to fulfill the assigned responsibilities.
- The selection of data structures is an important task, central to the software design process.
- But data structures must be carefully matched to the task at hand. A wrong choice can result in complex and inefficient programs, while an intelligent choice can result in just the opposite.
- It is also at this point that descriptions of behavior must be transformed into algorithms.
- These descriptions should then be matched against the expectations of each component listed as a collaborator, to ensure that expectations are fulfilled, and necessary data items are available to carry out each process.

7. Implementing Components

- The task at this step is to implement the desired activities in a computer language.
- In the implementation of one component it will become clear that certain information or actions might be assigned to yet another component that will act “behind the scene," with little or no visibility to users of the software abstraction. Such components are sometimes known as *facilitators*.
- An important part of analysis and coding at this point is characterizing and documenting the necessary preconditions a software component requires to complete a task and verifying that the software component will perform correctly when presented with legal input values.

8. Integration of Components

- Once software subsystems have been individually designed and tested, they can be integrated into the final product. This is often not a single step, but part of a larger process.
- Starting from a simple base, elements are slowly added to the system and tested, using *stubs*--simple dummy routines with no behavior or with very limited behavior--for the yet unimplemented parts.
- Testing of an individual component is often referred to as *unit testing*.
- One or the other of the stubs can be replaced by more complete code.
- Further testing can be performed until it appears that the system is working as desired. This is sometimes referred to as *integration testing*.
- The application is finally complete when all stubs have been replaced with working components. The ability to test components in isolation is greatly facilitated by the conscious design goal of reducing connections between components, since this reduces the need for extensive stubbing.
- During integration it is not uncommon for an error to be manifested in one software system, and yet to be caused by a coding mistake in another system.
- Thus, testing during integration can involve the discovery of errors, which then results in changes to some of the components. Following these changes the components should be once again tested in isolation before an attempt to reintegrate the software, once more, into the larger system.
- Re-executing previously developed test cases following a change to a software component is sometimes referred to as *regression testing*.

9. Maintenance and Evolution

- The term software maintenance describes activities after the delivery of the initial working version of a software system.
- A wide variety of activities fall into this category.
  - ✓ Errors, or bugs, can be discovered in the delivered product. These must be corrected, either in updates or corrections to existing releases or in subsequent releases.
  - ✓ Requirements may change, perhaps as a result of government regulations or standardization among similar products.
  - ✓ Hardware may change. For example, the system may be moved to different platforms, or input devices, such as a pen-based system or a pressure sensitive touch screen, may become available. Output technology may change{for example, from a text-based system to a graphical window-based arrangement.
  - ✓ User expectations may change. Users may expect greater functionality, lower cost, and easier use. This can occur as a result of competition with similar products.
  - ✓ Better documentation may be requested by users.

Summary

- *Object-oriented design differs from conventional software design in that the driving force is the assignment of responsibilities to different software components.*
- *No action will take place without an agent to perform the action, and hence every action must be assigned to some member of the object community.*
- *Conversely, the behavior of the members of the community taken together must be sufficient to achieve the desired goal.*
- *The emphasis on behavior is a hall-mark of object-oriented programming.*
- *Behavior can be identified in even the most rudimentary descriptions of a system, long before any other aspect can be clearly discerned.*
- *By constantly being driven by behavior, responsibility driven design moves smoothly from problem description to software architecture to code development to finished application.*