CPSC 5011: Object-Oriented Concepts

Lecture 3: Programming by Contract,
Object-Oriented Design (RDD and CRC cards)

WHEN YOU WANT TO TALK ABOUT OBJECT-ORIENTED DESIGN
BUT YOU HAVE TO TAKE ATTENDANCE FIRST
imgflip.com

Defensive Programming

- Few assumptions made about data, state or environment
    - Little assumed to be correct (form, content, value, …)
    - Implicit assumption is that errors abound
- Extensive testing needed to ensure correct execution
- Significant overhead (and clutter code)
- Effectively contains error

Encapsulation => Reliable Code

Class Design dependent on Class Designer -- IDEALS

- No function can modify (class, object) data
  - WITHOUT permission
- All constructors create objects in an initial valid state
- All mutator operations preserve validity of state
  - Insure legal state transitions
- Data Integrity
- Application Programmer Need not check for valid state

Proper Use Dependent on Client (application programmer)

Contractual Design

Published invariants, pre and postconditions

- Software viewed as a contract
  - Between the class designer and the client
- Responsibility
  - Client assumes that the software has been implemented correctly
  - Client notes invariants
  - Client must satisfy preconditions
  - Client tracks postconditions for potential state change(s)
- Efficiency regained
  - No need to check data for form or value

May not be appropriate for safety critical software

Programming by Contract

- Alternative to Defensive Programming
  - avoid overhead of defensive programming
  - retain safety of defensive programming
  - responsibility for secure code shared between class designer and application programmer
  - alleviates demand for extensive testing
- Documentation Convention => Communicates
  - ASSUMPTIONS about environment & use
    - Must be met by application programmer
    - Increased need to indicate state change
    - Decreased responsibility for error
  - RESTRICTIONS
    - Must be followed, as in contract, when using class

Efficiency vs Safety

Example: STACK `pop()` operation

Client pops off EMPTY stack : ILLEGAL

STACK class response choices:

1) Undefined Behavior – no internal check

    precondition for `pop()`: STACK object may NOT be empty

    `pop()` does not check for empty state – no overhead

  Client deals with consequences of violation of precondition

    data corruption

    delayed failure (hard to trace)

2) Return default value (zero) – internal check

  no precondition – defensive programming check

  for empty layered on top of all pop() calls

    All users penalized by overhead

3) EXCEPTION -- a viable option? How is state known without check?

Programming by Contract

- Bertrand Meyer (architect of Eiffel)
- Operations viewed as agents FULFILLING a contract
- Defensive programming possible but NOT default
- Formal agreement between class designer and application programmer

If client (application programmer) meets preconditions, class designer guarantees postconditions

Tradeoffs EXPLICIT: efficiency versus security

Programming by Contract

- Preconditions
  - State required expectations for correct function call
  - Must be met before call made
  - Can be “none”
- Postconditions
  - Guarantee of state after call processed
  - Used to record potential and actual state changes after function execution
- Interface invariants
  - Document stable conditions for use of the class
  - Only PUBLIC interface
  - Informs client of constraints
- Implementation invariants
  - Internal, for software maintenance
  - Identifies internal design decisions and constraints

Meyer: Pre and Postconditions

- A precondition-postcondition pair describes the contract that the routine (supplier of a certain service) defines for its callers (clients of that service).
- Perhaps the most distinctive feature of contracts is that any good contract entails <u>obligations</u> as well as <u>benefits</u> for both parties — with an obligation for one usually turning into a benefit for the other. This is true of contracts between classes, too:
    - The **precondition** binds the ***client***: it defines the conditions under which a call to the routine is legitimate. It is an obligation for the client and a benefit for the supplier.
    - The **postcondition** binds the ***class***: it defines the conditions that must be ensured by the routine on return. It is a benefit for the client and an obligation for the supplier.
- The <u>benefits</u> are, for the client, the ***guarantee*** that certain properties will hold after the call; for the supplier, the guarantee that certain assumptions will be satisfied whenever the routine is called. The <u>obligations</u> are, for the client, to satisfy the requirements as stated by the precondition; for the supplier, to do the job as stated by the postcondition.

Meyer: Pre and Postconditions

- **Example:** `Stack`
  - preconditions:
    - `push` may not be called if the `Stack` representation is full.
    - `pop` and `peek` may not be applied to an empty `Stack`.
  - postconditions:
    - After a `push`, the `Stack` may not be empty, its top is the element just pushed, and its number of elements has been increased by one.
    - After a `pop`, the `Stack` may not be full, and its number of elements has been decreased by one.
    - No change to `Stack` after `peek`

Meyer: Contract for Push

	OBLIGATIONS	BENEFITS
SUPPLIER	(satisfy postcondition:) update stack to have x on `top` (`peek` yields x), count increased by 1, not empty	(from precondition:) simpler processing thanks to the assumption that stack is not full

Meyer: Non-redundancy Principle

Non-Redundancy principle : Under no circumstances shall the body of a routine ever test for the routine's precondition.

- Defensive programming
  - redundant checking
  - add unnecessary complexity and overhead; compromises reliability
  - adds checks instead of addressing issue
- Design by Contract
  - identify the consistency conditions that are necessary to the proper functioning of each client-supplier cooperation (each contract)
  - specify, for each one of these conditions, whose responsibility it is to enforce it: the client's, or the supplier's

Preconditions

- Removes need for class operation to verify precondition
- Published so understood by those requesting service
  - Potentially SEVERE consequences (i.e. `pop()` off empty STACK)
- Describe required state necessary for correct behavior
- If not met, no guarantee about resulting behavior
- Must be verifiable
  - Client (AP) must be able to verify precondition
  - i.e. can check if STACK empty, AP can choose or avoid overhead
- Define compatibility between object state & operation
  - Not always easy to verify long-term
  - i.e. file open before reading (file existence is one time check)
- Define validity of argument
  - Acceptable values (not type: compiler checks type)

Postconditions

- Identifies potential and actual state changes
- Published so application programmer can
  - track state changes
  - verify subsequent preconditions
- Describe state object is left in after function exited
  - NOT a description of operation
- EXAMPLE:
  - `push()` STACK not empty
  - `pop()` STACK may be empty

Meyer: Invariants

Preconditions and postconditions describe the properties of individual routines. There is also a need for expressing <u>global properties of the instances of a class, which must be preserved by all routines</u>. Such properties will make up the class invariant, capturing the deeper semantic properties and integrity constraints characterizing a class.

Meyer: Invariants

- **Example:** `Stack`
  - Assume we have a class `Stack` with features `count`, `capacity`, `top`, `size`. Then the class invariants could include:
    - acount_non_negative: `0 <= count`
    - count_bounded: `count <= capacity`
    - consistent_with_array_size: `capacity = object.capacity`
    - empty_if_no_elements: `isEmpty = (count = 0)`
- **Example:** `BankAccount`
  - Assume that we have a class `BankAccount` with features `deposits_list`, `withdrawals_list` and `balance`. Then the class invariants could include:
    - consistent_balance: `balance = deposits_list.total – withdrawals_list.total`

    where the function `total` gives the cumulated value of a list of operations (deposits or withdrawals). This states the basic consistency condition between the values accessible through `deposits_list`, `withdrawals_list` and `balance`.

Interface Invariants

- Published at top of class: higher level than preconditions
  - Interface is public face of class
  - Provision/guarantee of behavior
- Describe restrictions on use of objects
  - Assignment operator private => copying suppressed
  - Copy constructor private => call by value not supported
- Describe preconditions that apply to all public functions
  - i.e. threshold affects magnitude of functionality
  - True upon entry to all operations
    - *If object has been manipulated correctly (preconditions met)*
  - Reduce need for internal testing
- Define relationship between two (or more) mutators
- Restrict state and state transitions

Implementation Invariants

- Design details for class designer(s) and software maintenance
- Published at top of implementation file
- Class integrity maintained when subsequent designers
  - implement changes/upgrades
  - add new operations
- Describe design choices
  - Hash table collisions handled via chaining
  - Priority queue items aged so as to minimize starvation
  - Interface of subObject echoed (i.e. wrapper)
- Describe bookkeeping details – ownership, etc.
- Define legal values of data fields
- Define relationships between fields
  - i.e. inventory value drives commission percentage

Object-Oriented Design (OOD)

- Dual Perspective
  - Client (application programmer) programs to interface
  - Class designer controls private implementation
- Encapsulation
  - Preserves internal control of state
- Programming by Contract
  - Capitalizes on encapsulation and dual perspective
  - Published pre and post conditions allow client to track state
  - Removes need for extensive testing
- Documentation is (not?) a contract!
- Documentation is specification of design

Single Responsibility Principle (SRP)

Every object has a single responsibility

=> *only ever one reason to modify a class*

- Emphasizes cohesion and promotes software maintenance
- Class functionality focuses on primary goal
  - Class designer precisely targets use, and potential reuse
  - Class integrity easier to preserve
- When preservation of state (implementation invariant) is consistent with expectations of use (interface invariant), the *single responsibility principle holds*

ICL: Programming by Contract

Create preconditions, postconditions, interface and implementation invariants for the `Stack` class.

Code: Week2/Code/stack

Object-Oriented Analysis

- **Analysis requires**
    - Detailed textual description, commonly called a functional specification, of the tasks that the software system needs to be carried out
        - Focuses on what needs to be done, not how it should be done
        - Use case: description of the sequence of actions that yield a benefit for a user of a system
- *Design requires*
    - *Identification of the classes, the responsibilities of these classes, and the relationships among these classes*
        - *Responsibility-driven design, CRC cards*
- *Implementation requires*
    - *Coding, testing, and deployment of the classes and methods*
        - *Unit testing, prototyping, agile development*

Responsibility Driven Design (RDD)

Identify responsibilities (functionality) and required information

- RDD is 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
- Works well with Programming by Contract
  - Specify design and all contractual expectations as to use
- Implementation invariant
  - Specifies the design of the object
  - Focus on functionality and internal responsibility for state
- Interface invariant
  - Specifies public functionality
  - Identifies client responsibility for consistent use
- Clear and cohesive interfaces reinforce class

Behavior & Use cases

- Just as an Abstract Data Type is characterized more by behavior than by representation, the goal in using Responsibility Driven Design will be to first characterize the application by behavior.
    - First capture the behavior of the entire application
    - Refine this into behavioral descriptions of subsystems
    - Refine behavior descriptions into code
- Because of the ambiguity in the specification, the major tool we will use to uncover the desired behavior is to walk through application scenarios to create use cases.
    - 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

Use Case

Use cases are an analysis technique to describe in a **formal** way how a computer system should work.

- Each use case focuses on a specific scenario, and describes the steps that are necessary to bring it to successful completion
- Each step in a use case represents an interaction with people or entities outside the computer system (the actors) and the system itself
- A use case must describe a scenario (or sequence of actions) that yield a result that is of some value to one of the actors
- Most scenarios that potentially deliver a valuable outcome can also fail; a use case should include variations that describe these situations
- Minimally, use cases should have a name that describes it concisely, a main sequence of actions, and, if appropriate, variants to the main sequence

Example: IIKH

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 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.

- Here are some of the things a user can do with the IIKH (use cases):
  - 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

Object-Oriented Design

- *Analysis requires*
  - *Detailed textual description, commonly called a functional specification, of the tasks that the software system needs to be carried out*
    - *Focuses on what needs to be done, not how it should be done*
    - *Use case: description of the sequence of actions that yield a benefit for a user of a system*
- **Design requires**
  - Identification of the classes, the responsibilities of these classes, and the relationships among these classes
    - Responsibility-driven design, CRC cards
- *Implementation requires*
  - *Coding, testing, and deployment of the classes and methods*
    - *Unit testing, prototyping, agile development*

Classes, Responsibilities, Relationships

- Identify classes
  - **Nouns**; should be in the **singular form** and remove any instances of classes (i.e. objects)
  - After finding the obvious classes, look for additional classes required to carry out the necessary work
- Identify responsibilities
  - **Verbs** (or active verb phrases)
  - A responsibility must belong to exactly one class (*decompose if not*)
- Identify relationships
  - Three common relationships between classes include dependency ("uses"), aggregation ("has"), and inheritance ("is)
    - A class depends on another if it manipulates objects of the other class
    - A class aggregates another if its objects contain objects of the other class
    - A class inherits from another if it incorporates the behavior of the other class

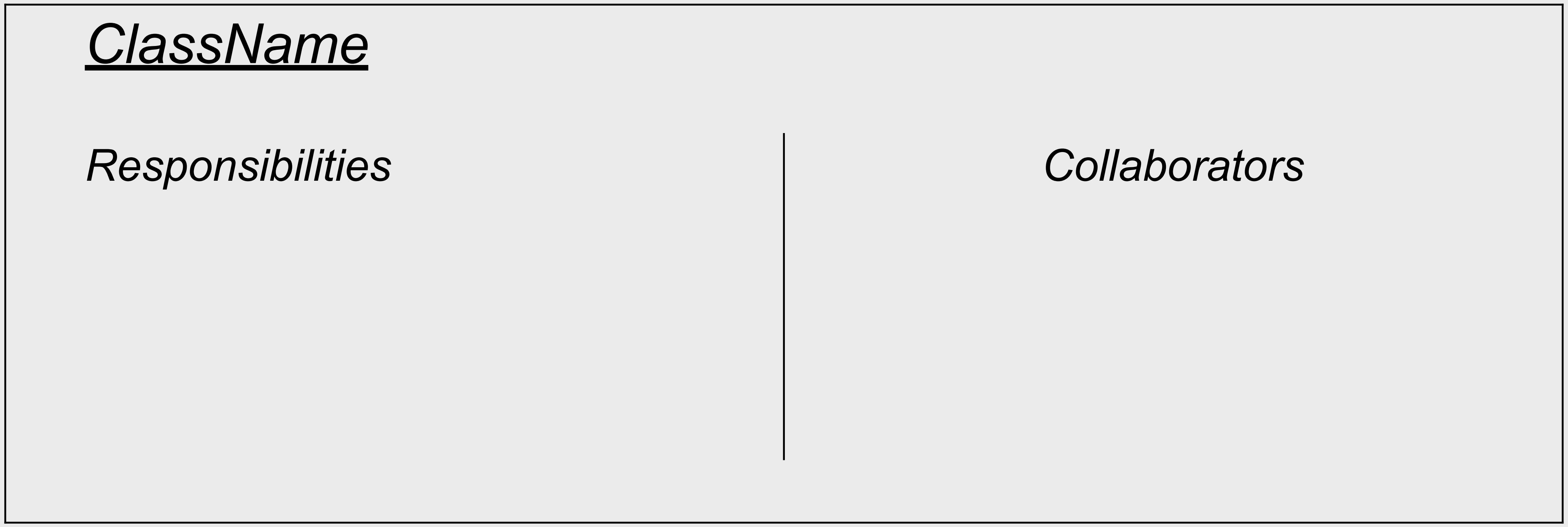
Software Components

Before defining classes, operations, and relationship, we will work with software components, their responsibilities and potential collaborators.

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.

- 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



- <u>ClassName</u> identify abstract design entities with which we can associate responsibilities for different tasks. Nouns are used to describe classes.
- <u>Responsibilities</u> identify problems to be solved. The responsibilities of a class are expressed by a handful of short (active) verb phrases
- <u>Collaborators</u> send or are sent messages in the course of satisfying responsibilities. Collaboration is not necessarily a symmetric relation. Helps identify **coupling** in the design.

Example: IIKH – 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 (the greeting) 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.

Many of the details concerning exactly how this is to be done ~~can~~ *should* be ignored for the moment.

Example: IIKH – Recipe Database

Ignoring the planning of meals for the moment, your team elects to next explore the **Recipe Database** component.

- 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 Who/What Cycle

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 charged with performing the action

Every *what* must have a *who*, otherwise it simply will not happen. Sometimes the who might not be obvious at first, i.e., who should be in charge of editing a recipe?

Example: IIKH – Recipe

We make the **Recipe** itself into an active data structure. It maintains information, but also performs tasks.

- 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).

Example: IIKH – Planner, Date, Meal

Returning to the greeter, we start a different scenario. This leads to the description of 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** component holds a sequence of meals for an individual date.

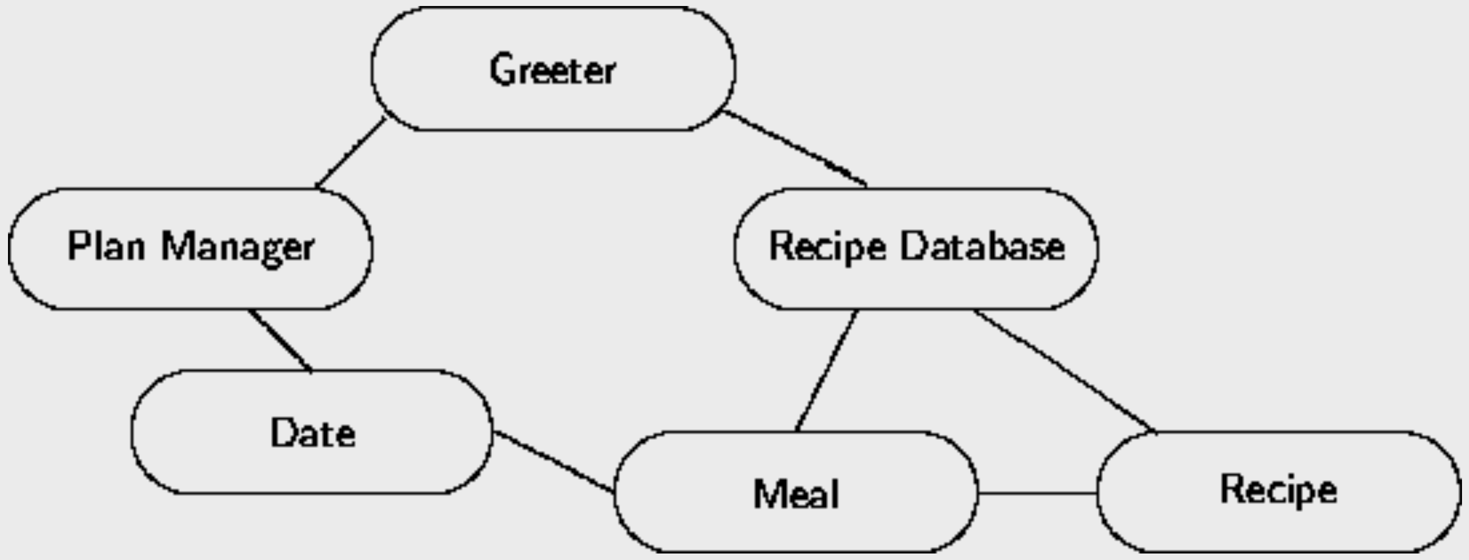
- User can edit specific meals.
- User can annotate information about dates ("Bob's Birthday", "Christmas Dinner", and so on).
- Can print out grocery list for entire set of meals.

The **Meal** component holds information about a single 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.

Example: IIKH – The Six Components

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



You can at this point assign the different components to different programmers for development.

Characteristics of Components

- Behavior and State
  - 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.
  - The state of a component represents all the information (data values) held within a component.
  - Notice that it is common for behavior to change state. For example, the edit behavior of a recipe may change the preparation instructions, which is part of the state.
- 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 Recipe.
- Coupling and Cohesion
  - Cohesion is the degree to which the tasks assigned to a component seem to form a meaningful unit. Want to maximize cohesion.
  - Coupling is the degree to which the ability to fulfill a certain responsibility depends upon the actions of another component. Want to minimize coupling.
- Interface and Implementation
  - We have characterized software components by what they can do. The user of a software component need only know what it does, not how it does it.

Parnas’ Principles

Separation of interface and implementation leads to two views of a software system. The term *information hiding* is used to describe the purposeful hiding of implementation details.

These ideas were captured by computer scientist David Parnas in a pair of rules, which are known as Parnas' 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 implementor 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.

Object-Oriented Implementation

- *Analysis requires*
  - *Detailed textual description, commonly called a functional specification, of the tasks that the software system needs to be carried out*
    - *Focuses on what needs to be done, not how it should be done*
    - *Use case: description of the sequence of actions that yield a benefit for a user of a system*
- *Design requires*
  - *Identification of the classes, the responsibilities of these classes, and the relationships among these classes*
    - *Responsibility-driven design, CRC cards*
- **Implementation requires**
  - Coding, testing, and deployment of the classes and methods
    - Unit testing, prototyping, agile development

Formalize the Interface

The next step is to 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.
- Information is assigned to each component and accounted for.
- Scenarios are replayed in order to ensure all data is available.

Objects and Classes

- Objects are entities in computer programs that include
    - State: the collection of all info held by an object is the object’s state; may change over time, but only when an operation has been carried out on the object that causes the state to change
    - Behavior: defined by the operations (or methods) than an object supports; not all operations are suitable for all objects, there must be an mechanism for rejecting improper requests
    - Identity: each object has its own identity i.e. 2 or more objects can support the same operations and have the same state, yet be different from each other
- A class definition must include
    - the operations that are allowed on the objects of the class
    - the possible states for objects the class

ICL: CRC Cards

Based on the use cases provided in ICL: Week 3, create CRC cards using the online CRC Card Maker.