Chapter 1, Lecture notes

Data Abstraction: The Walls

1. Most of us starting out in programming just start coding and hope for the best. However, in the workday world, there has to be teamwork that includes an overall plan, organization, and communication. Being haphazard is not cost effective. Software engineering provides techniques to aid in developing software.

   1.1 Object-Oriented Concepts

   1.1.1 **Problem solving** involves taking the statement of a problem and developing a computer program that solves the problem.

   1.1.1.1 There are several ways to do that. Let's explore the "**object-oriented analysis and design (OOA/D)**" for problem solving.

   1.1.1.2 For object-oriented, a "**solution**" is a computer program with interacting classes of objects.

   1.1.1.3 An "**object**" has the characteristics and behaviors related to the solution. Each object would deal with an aspect of the solution.

   1.1.1.4 A set of objects of the same type is a "**class**".

   1.1.1.5 A particular object of a class is called an **instance** of a class

   1.1.1.6 When you create an object, you **instantiate** the object

   1.1.1.7 **Objected-Oriented Analysis and Design (OOA)** – process of understanding what the problem is and the requirements of a solution

   1.1.1.7.1 In analysis, need to get what the **end users** want

   1.1.1.7.2 **Requirements** give *what* solution must be and must do – but can leave out the *how*

   1.1.1.7.3 So OOD, you describe a solution to a problem

   1.1.1.7.4 Express the solution in terms of software objects and how they will **collaborate** (objects collaborate by sending **messages** to each other)

   1.1.1.7.5 The interactions between objects are important and must be planned carefully

   1.1.1.7.6 OOD typically involves creating one or more **models**

   1.1.2 Aspects of an Object-Oriented Solution

   1.1.2.1 Typically a solution to a problem in the textbook in a program

   1.1.2.1.1 A program is made up of **modules** working together

1.1.2.1.2 A module is self-contained unit of code – a stand-alone **function**, a class **method**, class itself, a group of several functions or classes working together, or other blocks of code

1.1.2.1.3 Functions and methods implement **algorithms** (step-by-step recipes for doing a task within a finite amount of time)

1.1.2.2 Object-oriented programming languages allow building classes of objects

1.1.2.2.1 Class combines **attributes** (characteristics) of objects of a single type together with objects' operations (**behaviors**) into a single unit

1.1.2.2.2 Individual data items in a class are called **data members**

1.1.2.2.3 The operations specified in the class are methods or **member functions**

1.1.2.3 **Encapsulation** hides inner details

1.1.2.3.1 **Functions** encapsulate behavior and objects encapsulate data as well as behavior

1.1.2.3.2 Like a clock that encapsulates time (an attribute) and certain operations (like setting time). You don't see how it does it unless you have a see-through clock

1.1.2.3.3 Classes can inherit properties and operations from other classes (like an alarm clock inherits from clock)

1.1.2.3.4 **Inheritance** is another object-oriented concept that allows you to reuse classes

1.1.2.3.5 Inheritance may make it impossible for the compiler to figure out which operation you need at a particular situation

1.1.2.3.6 **Polymorphism** (means *many forms*) allows determination to be made at execution time

"Object-oriented programming" or "OOP" has three principles:

1. Encapsulation: Objects that combine data and operations
2. Inheritance: Classes can inherit properties from other classes
3. Polymorphism: objects can determine appropriate operations at execution time

1.2 Achieving a Better Solution – What if you came up with three different solutions? Which one do you determine is best?

1.2.1 Cohesion

1.2.1.1 Highly **cohesive** module should do one well-defined task

1.2.1.1.1 First, the module should have a good name that describes what it is doing (i.e. *sort* should just sort)

1.2.1.1.2 Second, should be easy to reuse in other software projects (*sort* should be used without change)

1.2.1.1.3 Third, highly cohesive module is much easier to maintain (*sort* is easier to fix since it all it does is sort)

1.2.1.1.4 Fourth, highly cohesive module is more **robust** – i.e. less likely to be affected by change

1.2.1.1.5 Low cohesion is like a person with "too many irons in the fire"

1.2.2 Coupling

1.2.2.1 **Coupling** is a measure of dependence among modules

1.2.2.2 Dependence could be sharing data structures or calling each other's methods

1.2.2.3 Ideally the modules in a design should be independent of one another

1.2.2.4 Modules should be **loosely coupled** – **highly coupled** modules should be avoided

1.2.2.5 Loose coupling has several benefits

1.2.2.5.1 First, a loosely coupled module is more adaptable to change

1.2.2.5.2 If class A depends (highly coupled to) class B and class B changed, then that would probably affect class A

1.2.2.5.3 Second, loosely coupled modules are easier to understand

1.2.2.5.4 If class A depends on class B then you must understand class B as well (can get a bit murky with many modules depending on many other modules)

1.2.2.5.5 Third, loosely coupled modules have more reusability.

1.2.2.5.6 If class A does not depend on class B, it is easier to use class A

1.2.2.5.7 Fourth, loosely coupled modules have increased cohesion

1.2.2.5.8 Highly cohesive modules tend to be loosely coupled

1.2.2.5.9 Remember that some coupling is required

1.3 Specifications – in a modular solution design, each module states what it does but not how it does it (allows modules to be isolated from each other). A sort will state *what* it does (sort) but not *how* it does it

1.3.1 Operation Contract

1.3.1.1 **Operation contract** describes how a module is to be used and its limitations

1.3.1.2 Should be started during analysis, finished in specification, and at least placed as comments in the header file (i.e. document)

1.3.1.2.1 Needs the **interface** to the method which states how to call the method, the arguments it expects, and what must be true *before* it is called

1.3.1.2.2 In design, clearly specify the purpose of each module and the **data flow** between the modules – i.e. what data is available to the module before execution, what is assumed, what actions are done, and what does the data look like after execution of the module (assumptions, input, and output)

1.3.1.2.3 Example with a sort of integers:

1.3.1.2.3.1 Function will take an array of *num* integers, num > 0

1.3.1.2.3.2 Function will return array with integers sorted

1.3.1.3 The contract helps programmers understand what the modules does but the contract does not specify how the module does its task

1.3.1.4 You need to write the **precondition** and **postcondition** of the module, like so:

```
// Sorts and array
// Precondition: anArray is an array of num integers, num > 0
// Postcondition: The integers in an anArray are sorted
sort(anArry, num)
```

1.3.1.5 That was just a draft, a refined specification would be:

```
// Sorts an array into ascending order
// Precondition: anArray is an array of num integers and 1 <= num <= MAX_ARRAY
// Postconditon: anArray[0] <= anArray[1] <= … <= anArray[num – 1]
sort(anArray, num)
```

1.3.1.6 Note that you need to specify any global constants like MAX_ARRAY

1.3.1.7 This will be required in programming assignments

1.3.2 Unusual Conditions – there are several ways to deal with unusual conditions:

1.3.2.1 **Assume that invalid situations don't occur**

1.3.2.1.1 That actually can be stated as an assumption in the contract

1.3.2.1.2 Then it is up to the client to insure that the invalid situations don't happen

1.3.2.2 **Ignore the invalid situation**

1.3.2.2.1 Method just ignores the invalid situation

1.3.2.2.2 However, that doesn't inform the client of what happened

1.3.2.3 **Guess at the client's intention**

1.3.2.3.1 That still can cause problems in that the client is not informed

1.3.2.4 **Return a value that signals a problem**

1.3.2.4.1 Could return a Boolean where true is success and false is not

1.3.2.5 **Throw an exception**

1.3.2.5.1 C++ allows throwing an exception

1.3.2.5.2 That allows the method to report a problem without doing anything about it

1.3.2.5.3 Then the client can take action

1.3.2.5.4 But it is a bit radical so we'll reserve it to truly unusual circumstances

1.3.3 Abstraction – separates the purpose of a module from its implementation

1.3.3.1 Modularity breaks a solution into modules; abstraction specifies each module clearly *before* you implement it in a program

1.3.3.2 You can focus on the design of your solution without worrying about the implementation details

1.3.3.3 Separating the purpose of a module from its implementation is known as **functional** (or **procedural**) **abstraction**

1.3.3.3.1 Once the module is written, you can use it without knowing the particulars of that module

1.3.3.3.2 All you need is the specifications of that module

1.3.3.3.3 Don't need to know the implementation

1.3.3.4 Function abstraction is needed in team projects

1.3.3.4.1 You have to use modules written by others – no programmer in the real world writes a complex program by themselves

1.3.3.4.2 Can you actually use code written by others – you already have since you use the C++ Standard library

1.3.3.4.3 For instance, you can use the sqrt function that is in the C++ math library – don't need to know how it actually does a square root

1.3.3.5 You have a collection of data and a set of operations on that data

1.3.3.5.1 Operations might include adding new data, removing data, or searching for some data

1.3.3.5.2 **Data abstraction** focuses on what the operations do with the collection of data and not how to implement the data

1.3.3.5.3 Other modules know *what* operations they can do but not *how* the operations are implemented

1.3.4 Information Hiding – abstraction deals with what should be visible to the outside or public view and what should be hidden

1.3.4.1 Information hiding not only hide details but ensures that no other module can tamper with those hidden details

1.3.4.1.1 In the module's specifications, you must identify details you can hide in the module

1.3.4.1.2 You hide *and* make those details inaccessible

1.3.4.1.3 Imagine placing walls around those various tasks a program does – the walls prevent the tasks from becoming entangled

1.3.4.1.4 Isolation of modules cannot be total – must pretend there is a slit in the wall

1.3.4.1.5 The slit would be the **prototype, declaration, or header** of the function

1.3.4.1.6 The slit would have the function or method's name, parameter list, and return type

1.3.4.1.7 Example using MyProgram calling the sort function in figure 1-2

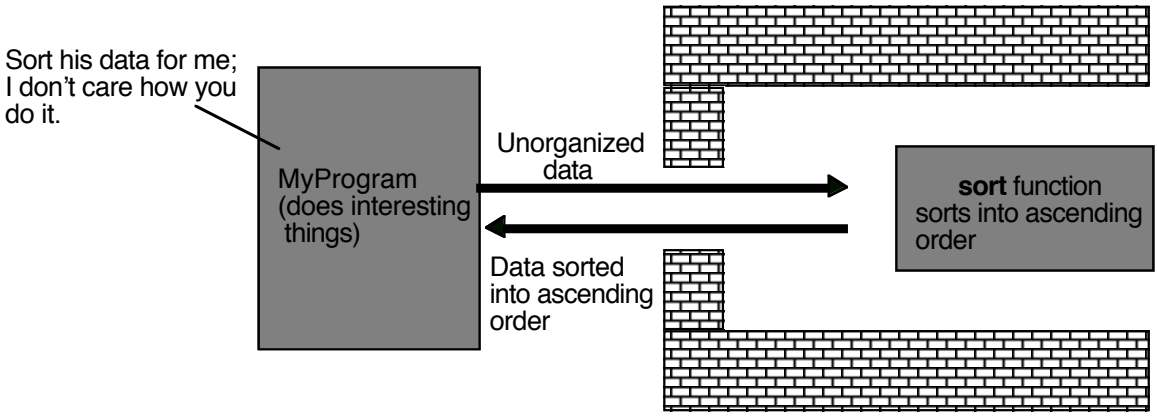

Figure 1-2, Tasks communicate through a slit in the wall

1.3.4.2 Works pretty good. However, suppose the sort function is improved, what would happen then? Like in figure 1-3?

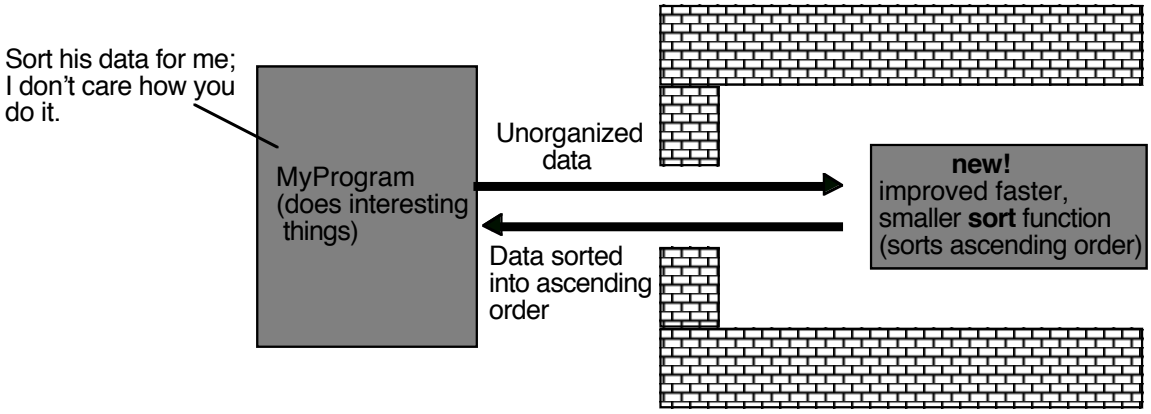

Figure 1-3, A revised implementation communicates through same slit in the wall

1.3.4.2.1 Notice that MyProgram does not have to change

1.3.4.2.2 This is a major strength of information hiding

1.3.5 Minimal and Complete Interfaces – interface for a class is made up of publicly accessible methods and data

1.3.5.1 Typically only the methods are publicly accessible to avoid problems with accessing public data methods

1.3.5.2 Interface for a class describes the only way for programmers to interact with that class

1.3.5.3 Each class should be easy to understand – keep the number of methods small yet give programmers the ability to do their tasks

1.3.5.4 A **complete interface** gives the programmer the ability to accomplish any reasonable task

1.3.5.5 A **minimal interface** only contains a module if it is essential to that class's responsibility

1.3.5.5.1 A minimal interface is easier to understand and less problematic that an interface that has many methods

1.3.5.5.2 However, sometimes a nonessential method can be too useful to omit

1.3.5.6 The interface to a function or method is more accurately called its **signature**

1.3.5.6.1 That includes the function's name, number, order, and types of its arguments; and any qualifiers like *const* that might apply

1.3.5.6.2 A signature looks like a function's prototype but does not include its return type

1.4 Abstract Data Types

1.4.1 Typically a solution to a problem requires operation on the data – like so (broadly):

- **Add** data to a data collection
- **Remove** data from a data collection
- **Ask questions** about the data in a data collection

1.4.1.1 The details vary from application to application but the overall theme is there

1.4.1.2 Most of the book is about data abstraction

1.4.1.2.1 Need to be able to define an **abstract data type** or **ADT**

1.4.1.2.2 An ADT is a collection of data AND a set of operations

1.4.1.2.3 You can specify the operations without knowing implementation details

1.4.1.3 Ultimately you need to implement an ADT using a **data structure** – a construct you can define in a programming language to store a collection of data

1.4.1.3.1 For example, you need to store a collection of names that can be accessed rapidly

1.4.1.3.2 *A collection of name items providing for rapid searches* is a value description of a simple ADT

1.4.1.3.3 The description must be rigorous enough to specify completely their effect on data yet not specify neither

how to store the data nor how to carry out the operations

1.4.1.3.4 You must choose a particular data structure when you **implement** the ADT

1.4.1.4 An example of the difference of an ADT and a data structure is in figure 1-4 showing a refrigerator's ice dispenser

1.4.1.4.1 The ADT would be the "user view from specifications"

1.4.1.4.2 You see what the operations are available including an indicator that glows if there is no ice

1.4.1.4.3 But the user does not care how the ice dispenser does its tasks

1.4.1.4.4 The Technician view would be like the data structure – the implementation of the ice dispenser

1.4.1.4.5 The only breaks in the steel wall around the ice dispenser would be the input (water) and output (chilled water, crushed ice, or ice cubes – or the indicator stating no ice)

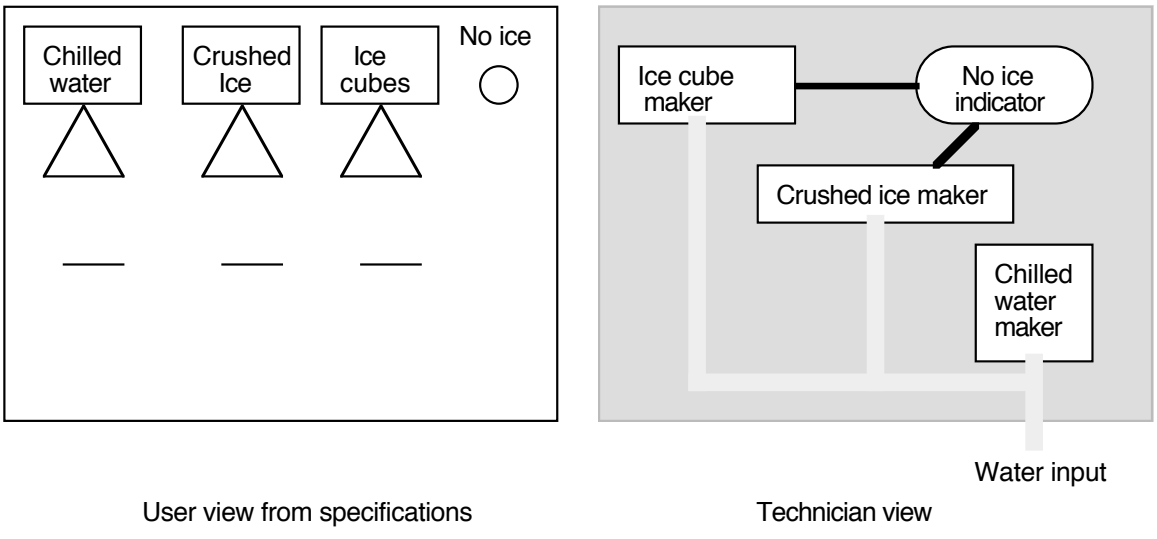

Figure 1-4, A dispenser of chilled water, crushed ice, and ice cubes

1.4.2 Designing an ADT

1.4.2.1 Design of the ADT should evolve naturally during the problem-solving

1.4.2.1.1 For instance, what if you wanted to determine dates of all holidays of a given year

1.4.2.1.2 The following pseudocode is a possible solution:

```
// Displays the dates of all holidays in a given year
listHolidays(year)

  date = date of first day of year
  while (date is before the first day of year + 1)
  {
    if (date is a holiday)
      write(date, " is a holiday")

    date = date of next day
  }
```

1.4.2.1.3 What data would be involved? It would be dates of month, day, and year

1.4.2.1.4 From the previous pseudocode you determine that you must:

- Determine date of first day of given year
- Decide whether a date is before another date
- Decide whether a data is a holiday
- Determine date of the day that follows a given date

1.4.2.1.5 **Unified Modeling Language (UML)** as shown in Appendix C would describe the functions as:

```
// Returns the data of the first day of a given year
+getFirstDay(year: integer) : Date

// Returns true if this date is before the given date;
//   otherwise return false
+isBefore(otherDate: Date): boolean

// Returns true if this date is a holiday; otherwise returns false
+isHoliday(): boolean

// Returns the date of the day after this date
+getNextDay(): Date
```

1.4.2.1.6 Now the listHolidays pseudocode looks like:

```
// Displays the dates of all holidays in a given year
listHolidays(year)

  date = getFirstDay(year)
  while (date.isBefore(getFirstDay(year+1))
  {
```

```
    if (date.isHoliday())
      write(date, " is a holiday")

    date = date.getNextDay()
  }
```

1.4.2.2 Another example would be an appointment book – one that spans a one-year period and makes appointments only on the hour and the half hour between 8 a.m. and 5 p.m (and store a brief notation about each appointment along with the date and time)

1.4.2.2.1 To solve this, you can define an ADT appointment book – the data items in this ADT are appointments (date, time, and purpose)

1.4.2.2.2 The two obvious operations are:

- Make an appointment for a certain date, time, and purpose (and don't make an appointment where an appointment is already there)
- Cancel appointment for a certain date and time

1.4.2.2.3 Some more operations you would probably want:

- Ask if you have an appointment at a given time
- Get purpose of your appointment at a given time

1.4.2.2.4 ADTs typically have initialization and destruction operations that are assumed but not specified

1.4.2.2.5 So the ADT appointment book operations are:

```
// Returns true is an appointment exists for date and time specified
// false otherwise
+isAppointment(apptDate: Date, apptTime: Time): boolean

// Inserts appointment for data, time, and purpose specified
// as long as there is no conflict with existing appointment
// Returns true if successful, false otherwise
+makeAppointment(apptDate: Date, apptTime: Time,
                 apptPurpose: string): boolean

// Deletes appointment for date and time specified
// Returns true if successful, false otherwise
+cancelAppointment(apptDate: Date, apptTime: Time): boolean

// Gets purpose of appointment at given date and time, if one exists
// Otherwise returns an empty string
+getAppointmentPurpose(apptDate: Date, apptTime: Time): string
```

1.4.2.2.6 Using the above ADT operations you can create pseudocode – for instance, how to change the date or time of an appointment

```
// Change date or time of an appointment

Get following data from user: oldDate, oldTime, newDate,
newTime

// Get purpose of appointment
oldPurpose = apptBook.getAppointmentPurpose(oldDate,
oldTime)
if (oldPurpose is not an empty string)
{
  // See whether a new date/time is available
  if (apptBook.isAppointment(newDate, newTime)
    // New date/time is booked
    write("You already have an appointment at ", newTime
          " on ", newDate)
  else
  {
    // New date/time is available; cancel old appointment;
    //   make new one
    apptBook.cancelAppointment(oldDate, oldTime)
    if (apptBook.makeAppointment(newDate, newTime,
       oldPurpose)
      write("Your appointment has been rescheduled to ",
            newTime, " on ", newDate)
  }
}
else
  write("You do not have an appointment at ", oldTime,
        " on ", oldDate)
```

1.4.2.2.7 Can design applications using ADT operations without knowing how the ADT is implemented

1.4.3 ADTs that suggest other ADTs

1.4.3.1 Both the holiday and appointment book had to use a date – the appointment book had to use time

1.4.3.1.1 You can design ADTs for the date and time – one set of ADTs suggesting other ADTs

1.4.3.1.2 That is not unusual

1.4.3.2 Let's have another example of implementing a database of recipes

1.4.3.2.1 Typical operations on the recipes would be:

// *Inserts a recipe into database*
+insertRecipe(aRecipe: Recipe): boolean

// *Deletes a recipe from database*
+deleteRecipe(aRecipe: Recipe): boolean

// *Gets named recipe from database*
+getRecipe(name: string): Recipe

1.4.3.2.2 Now what if you want to have an operation that would scale a recipe (if recipe is for *n* people, you want to revise it so that it will serve *m* people)

1.4.3.2.3 The recipe has measurements like 2 ½ cpus, 1 tablespoon, and ¼ teaspoon

1.4.3.2.4 This suggests another ADT – measurement … with the following measurements

// *Returns this measure*
+getMeasure(): Measurement

// *Sets this measure to another one*
+setMeasure(m: Measurement)

// *Returns this measure multiplied by a fractional scale factor,*
// *which has no units*
+scaleMeasure(scaleFactor: float): Measurement

// *Returns this measure converted from its old units to new units*
+convertMeasure(oldUnits: MeasureUnit,
newUnits: MeasureUnit): Measurement

1.4.3.2.5 What about adding fractions? C++ doesn't have a fraction data type so need to make an ADT for fraction and use it like:

// *Returns sum, reduced to lowest terms, of this fraction and*
// *given fraction*
+add(other: Fraction): Fraction

1.5 The ADT bag

1.5.1 A bag contains things – it is a **container**

1.5.1.1 You can specify and use an ADT **bag**

1.5.1.2 A bag doesn't do much more than contain the items

1.5.1.3 A bag is a container of a finite number of objects having same data type and in no particular order (a bag can contain duplicate items)

1.5.2 Identifying Behaviors

1.5.2.1 Some bag's behaviors would be:

- Get number of items currently in bag
- See whether bag is empty
- Add a given object to bag
- Remove an occurrence of specific object from bag, if possible
- Remove all objects from bag

1.5.2.2 Note that add operation doesn't indicate where in bag an object should go

1.5.2.3 The remove operation looks for particular item in bag (if it finds it, it takes it out)

1.5.2.4 If there are several objects equal to the removed one, they remain in bag

1.5.2.5 What is in a particular bag? The following operations answer that:

- Count number of times a certain object occurs in bag
- Test whether bag has a particular object
- Look at all objects that are in bag

1.5.2.6 Appendix C has the **class-responsibility-collaboration (CRC)**

Bag
Responsibilities
Get the number of items in the bag
See whether bag is empty
Add a given object to bag
Remove an occurrence of a specific item from
bag, if possible
Remove all objects from bag
Count the number of times a certain object occurs in bag
Test whether bag contains a particular object
Look at all objects that are in bag
Collaborations
The class of objects that bag can contain
Figure 1-6, A CRC card for a class bag

1.5.2.7 Since a bag is an ADT, we can only describe its data and specify its operations – can specify how to store the data nor how to implement its operations

1.5.3 Specifying Data and Operations

1.5.3.1 Before the bag can be implemented in C++, we need to describe its data and specify in detail the methods that correspond to bag's

behaviors (name methods, choose parameters, decide their return types, and write comments to fully describe their effect on the bag's contents)

1.5.3.2 Need to use pseudocode and UML notation, like so (UML):

// *Returns current number of entries in bag*
+getCurrentSize(): integer

// *Returns true if bag is empty*
+isEmpty(): boolean

1.5.3.2.1 Here is pseudocode to add an item and give it a parameter

// *Adds a new entry to bag*
add(newEntry)

1.5.3.2.2 But what to do if the add does not successfully complete? There are two options:

- Do nothing. We cannot add another item so we ignore it and leave bag unchanged
- Leave the bag unchanged, but signal the client that addition is impossible

1.5.3.2.3 Second option is the better one and we can specify it using UML notation like so:

+add(newEntry: ItemType): boolean

1.5.3.3 Two behaviors of removing entries: remove a particular entry or remove all entries – the pseudocode is:

// *Removes one occurrence of a particular entry from bag, if possible*
remove(anEntry)

// *Removes all entries from bag*
clear()

1.5.3.4 Now what would be the return types for the above functions? The remove needs to tell the client if the operation doesn't work, but the clear removes all items and doesn't need to return anything – as specified in UML notation:

+remove(anEntry: ItemType): boolean

+clear(): void

1.5.3.5 Remaining behaviors do not change contents of bag, like so:

// *Counts the number of times a given entry appears in bag*
+getFrequencyOf(anEntry: ItemType): integer

// *Test whether the bag contains a given entry*
+contains(anEntry: ItemType): boolean

// *Gets all entries in bag*
+toVector(): vector

1.5.3.6 The last one, toVector, gets all the entries into a vector to allow examining all the entries in the bag

1.5.3.7 The UML for the Bag

Bag
+getCurrentSize(): integer +isEmpty(): boolean +add(newEntr: ItemType): boolean +remove(anEntry: ItemType): boolean +clear(): void +getFrequencyOf(anEntry: ItemType): integer +contains(anEntry: ItemType): boolean +toVector(): vector
Figure 1-7, UML notation for class Bag

1.5.3.8 Note that the CRC and UML don't reflect all the details – like assumptions and unusual circumstances

1.5.3.9 The following shows how you can write them out:

Abstract Data Type: Bag	
DATA	
• A finite number of objects, not necessarily distinct, in no particular order and having same data type • The number of objects in this collection	
OPERATIONS	
PSEUDOCODE	DESCRIPTION
getCurrentSize()	Task: Reports the current number of objects in this bag Input: None Output: Number of objects currently in bag
isEmpty()	Task: Sees whether this bag is empty Input: None Output: True or false according to whether bag is empty
add(newEntry)	Task: Adds a given object to this bag Input: newEntry is an object Output: True or false according to whether addition succeeds
remove(anEntry)	Task: Removes an occurrence of a particular object from bag, if possible Input: anEntry is an object Output: True or false according to whether removal succeeds
clear()	Task: Removes all objects from this bag Input: None Output: None
getFrequencyOf(anEntry)	Task: Counts number of times an object occurs in bag Input: anEntry is an object Output: The integer number of times anEntry occurs in bag
contains(anEntry)	Task: Tests whether this bag has a particular object Input: anEntry is an object Output: True or false according to whether anEntry occurs in bag
toVector()	Task: Gets all object in this bag Input: None Output: A vector containing all entries currently in bag

1.5.4 An Interface Template for the ADT

1.5.4.1 As specification become more detailed, they should increasingly reflect your choice of programming language – should be able to write C++ headers for the bag's methods and organize them into a header file

1.5.4.2 For now, items in the bag will be objects of same class – can have the bag methods use a **generic type** of ItemType for each entry (can use a template<class ItemType>)

1.5.4.2.1 The class BagInterface is an **abstract base class** or just **abstract class** in C++ contains at least one method that is declared as virtual and has no implementation.

1.5.4.2.2 An abstract class can't be instantiated; can only be used as a base class.

1.5.4.2.3 Subclass must then implement methods specified but not defined in abstract class, the class is as follows:

LISTING 1-1, A file containing C++ interface for bags

```
/** @file BagInterface.h */
#ifndef  _BAG_INTERFACE
#define  _BAG_INTERFACE

#include <vector>
using namespace std;

template<class ItemType>
class BagInterface
{
public:
   /** Gets the current number of entries in this bad.
    @return  The integer number of entries currently in the bag */
   virtual int getCurrentSize() const = 0;

   /** Sees whether this is empty
    @return   True if the bag is empty, or false if not */

   /** Adds a new entry to this bag
    @post   If successful, newEntry is stored in the bag and
        the count of items in the bag has increased by 1
    @param newEntry  The object to be added as an new entry
    @return   True is addition was successful, or false if not */
   virtual bool add(const ItemType& newEntry) = 0;

   /** Removes one occurrence of a given entry from this bag,
       if possible
    @post   If successful, anEntry has been removed from the bag
        and the count of items in the bad has decreased by 1.
    @param anEntry   The entry to be removed
    @return   True if removal was successful, or false if not */
   virtual bool remove(const ItemType& anEntry) = 0;
```

```
    /** Removes all entries from this bag
     @post   Bag contains no items, and the count of items is 0 */
    virtual void clear() = 0;

    /** Counts the number of times a given entry appears in bag.
     @param anEntry    The entry to be counted
     @return    The numer of times anEntry appears in the bag */
    virtual int getFrequencyOf(const ItemType& anEntry) const = 0;

    /** Tests whether this bag contains an given entry
     @param anEntry   The entry to locate
     @return   True if bag contains anEntry, or false otherwise */
    virtual bool contains(const ItemType& anEntry) const = 0;

    /** Empties and then fills a given vector with all entries that
        are in this bag
     @return   A vector containing all the entries in the bag */
    virtual vector<ItemType> toVector() const = 0;
}; // end BagInterface
#endif
```

1.5.4.3 It only specifies what you can do, not how it is done

1.5.4.4 You don't need to have the class specified this way but its advantage is that you have documented the actions before doing the implementation

1.5.5 Using the ADT Bag

1.5.5.1 If everything is specified, you could hire a programmer to implement the methods

1.5.5.2 Let's use it to write a program that will take some cards, place them in the bag, then the user will guess what cards are in the bag

1.5.5.3 If the card guessed is in the bag, the card is removed, otherwise it will say card not in the bag

LISTING 1-2, A program for a card guessing game

```
#include <iostream> // For cout and cin
#include <string>     // For string objects
#include "Bag.h"      // For ADT bag
using namespace std;

int main()
{
   string clubs[] = { "Joker", "Ace", "Two", "Three",
                      "Four", "Five", "Six", "Seven",
                      "Eight", "Nine", "Ten", "Jack",
                      "Queen", "King" };
```

```
   // Create our bag to hold cards
   Bag<string> grabBag;

   // Place six cards in the bag
   grabBag.add(clubs[1]);
   grabBag.add(clubs[2]);
   grabBag.add(clubs[4]);
   grabBag.add(clubs[8]);
   grabBag.add(clubs[10]);
   grabBag.add(clubs[12]);

  // Get friend's guess and check it
   int guess = 0;
   while (!grabBag.isEmpty())
   {
      cout << "What is your guess?"
           << "(1 for Ace to 13 for King):";
      cin >> guess;

      // Is card in the bag?
      if  (grabBag.contains(clubs[guess]))
      {
         // Good guess – remove card from the bag
         cout << "You get the card!\n";
         grabBag.remove(clubs[guess]);
      }
      else
      {
         cout << "Sorry, card was not in the bag.\n";
      } // end if
   } // end while
   cout << "No more cards in the bag.\n";
   system("pause");
   return 0;
}; // end main
```