

## Chapter Thirteen

# Object-Oriented Software Development Design and Implementation

---

### Summing Up

The CRH cards of the previous chapter represent an analysis deliverable. Through role playing, the team discovered major classes while developing a deep understanding of the system. The team has established a common vocabulary amongst themselves. Even the client, Blaine, can continue to communicate with the programmers as long as they avoid implementation details concerning vectors and pointers.

---

### Coming Up

These CRH cards will now be used to help design the class definitions. This roughly bounds the design phase. After this, the member functions will be implemented, roughly bounding the implementation phase. After studying this chapter, you will be able to

- ★ use the analysis deliverable (CRH cards) to help design your class definitions
- ★ implement action responsibilities as public member functions
- ★ implement knowledge responsibilities as private data members
- ★ add constructors to initialize the data members
- ★ perform unit testing
- ★ perform system testing

*Exercises and programming projects*

## 13.1 Designing Class Interfaces

---

The analysis phase helped the team understand *what* the system will do. It is now time to concentrate on the *how*. There is some haziness between analysis and design. In fact, during role-playing scenarios, some design decisions were made.

Chelsea comments that some software developers claim a separation between analysis and design. She is of the opinion that this is a holdover from the waterfall model of software development—before objects. This rigid waterfall model focused on well-defined deliverables. Practitioners want to complete the analysis phase before design, for example. The one-way waterfall model attempts to avoid revisiting previous steps (it is more difficult for water to travel uphill than downhill). The waterfall model was suggested in Chapter 1 to introduce analysis, design, and implementation as separate phases of software development. It was an attempt to suggest that there are things you can do before sitting at the computer and whacking out code. However, the problems were relatively simple. There was little need to revisit analysis and design.

Most of the problems up to this point haven't required much analysis or design. A small set of algorithmic patterns (IPO, Multiple Selection, Determinate Loop, and so forth) has solved most problems. And of course, it has been easy to guess that programming problems in the chapter where a certain algorithmic pattern or C++ structure (`if . . . else`, `for` loop) was introduced would need that construct to be completed. Software development usually isn't so easy.

Object-oriented software developers follow the iterative model of software development. This is a more flexible strategy that allows one to modify existing results in order to fix earlier mistakes or poor designs. Now, during design and implementation, the team must determine how things will get done. Now is the time to think about the computer system(s) that will be running the cashless jukebox and the programming language that will be used to implement it. The programmers have decided it will be easier to use text-based input and output. That's all they know. They have verified this decision with a quick call to the customer (Blaine). They will also use the language they know from their study of computing fundamentals: C++.

At this point, only three of the team members will continue the development. The other team members have agreed to help if the need arises. Chelsea will work with Matt, the computer engineer who was on the team during analysis. Charlie, the computer science major who has finished the first year of the computer science curriculum, has also promised to help. This is Matt's first experience with object-

oriented software development. But Matt is also interested in how the computer communicates with the physical CD player.

The jukebox team derived 10 classes. Looking ahead to the design and implementation of these classes, Chelsea suggests that with Ed's hardware device, `cdPlayer` and `playlist` need not be implemented. The `student` and `track` classes follow the State Object pattern. Charlie, Matt, and certainly Chelsea are quite familiar with classes that mostly exist to store state and provide suitable access to it. The `cdCollection` and `studentCollection` classes fit another pattern similar to that of the `bag` class that Charlie and Matt have studied. Both classes

- ★ contain a collection of many objects
- ★ provide a way to iterate over the collection—from the first to the last
- ★ provide suitable access to any object in the collection

Both programmers see a relationship between the responsibilities on the CRH cards and the C++ class definitions. The things that each instance of a class must *do* might be implemented as public member functions or as statements in an algorithm. The things that each instance of a class must *know* might become private data members of that class. Chelsea warns Matt and Charlie that although CRH cards provide some input into class design, translation from CRH cards to C++ class definitions requires more design decisions. The translation is not direct. Some things may be added, others removed. For example, the CRH cards may not contain such things as return types or parameters.

CRH cards document analysis decisions. They fashion a solution. They also provide a glimpse of the class definitions that need to be designed. As mentioned earlier, each CRH card lists its set of action responsibilities, which might end up as class member functions or as statements in an algorithm. Each class's CRH card also lists its knowledge responsibilities, which might be implemented as data members, and its helper classes, which indicate relationships between objects.

Some CRH cards denote one or more helper classes that may be sent messages. The helpers listed on the CRH cards are potential receivers of these messages. Chelsea reminds the team of the differences between a sender and a receiver. For example, the `jukeBox` object may send a `getStudent` message to the `studentCollection` object. `jukeBox` is the sender, `studentCollection` is the receiver. `studentCollection` (the receiver, in this case) must implement the `getStudent` responsibility. Therefore, `getStudent` should be written as a class member function of `studentCollection`.

The next major activity in the development of the cashless jukebox begins with Chelsea, Matt, and Charlie designing the class definitions. The programming team will get the class definitions to compile. They will eventually choose the data mem-

bers that most appropriately help each class fulfill its knowledge responsibilities. Later on, the constructors and other member functions will get implemented. Frequent compilation and testing will improve chances for a successful system.

The aforementioned design and implementation are iterative processes that include activities such as these:

- ★ Design the class interfaces by specifying the member functions, including parameters and return types, for the class member function headings.
- ★ Determine the knowledge responsibilities (data members) required to implement the action responsibilities.
- ★ Add appropriate constructors (or other initialization routines).
- ★ Implement the constructors that initialize the data members.
- ★ Implement the member functions.

The iterative model of software development allows programmers to back up into analysis, delve into implementation, and return to class design. Not only will the programmers refine algorithms while implementing the member functions, they will also refine class definitions and data members. They may also change the CRH cards.

At Chelsea's suggestion, Ed and Matt first consider the `student` and `studentCollection` classes.

### 13.1.1 The `studentCollection` Class

The `studentCollection` CRH card lists two action responsibilities: `getStudent` and `addStudent`.

FIGURE 13.1. CRH card for `studentCollection`

<b>Class:</b> <code>studentCollection</code>	
<b>Responsibilities:</b>	<b>Helpers:</b>
know all students	<code>student</code>
<code>getStudent</code>	
<code>addStudent</code>	

These responsibilities could be written as member functions.

```

class studentCollection { // First draft
public:
    ??? getStudent(???);
    ??? addStudent(???);
private:
    // TBA: Know all students
};

```

However, some things are missing. As the programming team designs class definitions, it must also make design decisions concerning parameters and return types of those functions.

Chelsea asks Matt and Charlie if the `getStudent` message requires any arguments. Yes, it appears that `studentCollection` will need the student ID number from the sender (`jukeBox`). Matt asks, “What return type should `getStudent` have?” Chelsea says that the return type should be `student`. If we assume the student ID will be represented as a string, the member function heading could look like this:

```

student getStudent(string ID);
// Return the student with the given ID

```

The `addStudent` signature might reasonably be

```

void addStudent(student newStudent);
// Add the given student to the collection

```

Matt wants to know what private data members are needed so `studentCollection` “knows all students.” What data would the constructor(s) need to initialize a `studentCollection`? Chelsea answers, “Member function definitions first, data members later.” Since we don’t know how `studentCollection` is being stored yet, we will also postpone constructors. Charlie reminds Matt that accessing functions should be declared `const`. A `getStudent` message will return the `student` object associated with an ID number. It will not modify the `studentCollection` object. On the other hand, `addStudent` will. Here is a second refinement of the class definition that will allow an object to accept the preceding message. The class definition still omits data members and constructors.

```

class studentCollection { // Second draft--no constructors yet
public:
    student getStudent(string ID) const;
    // Return the student with the given ID

    void addStudent(student newStudent);

```

```
// Add the given student to the collection

private:
    // TBA: Know all students
};
```

Matt remembers from role playing that `jukeBox` also wanted to know if a student was valid. The current `getStudent` heading returns a student. However, what happens if the ID number from a student ID card does not match one of the students in `studentCollection`? Matt advises Charlie of an earlier design decision: “The team decided it was okay to simply create a new account.” The new student would begin with 1,500 minutes credit and no songs played on the current date. Chelsea confirms that if `studentCollection` constructs a new student account automatically, `jukeBox` will have no knowledge nor concern over this detail. This is a cleaner design.

Matt wonders if `studentCollection` should be designed with `addStudent` as a public member function. Charlie says no. The `getStudent` operation will be the only one that sends an `addStudent` message. Chelsea remarks that `addStudent` has enough detail to justify implementing it as a separate function. However, thinking ahead to implementation, making `addStudent` public would make it easier to add students to the collection.

Of course, `studentCollection` requires a `student` class, so perhaps `student` should have been the first class implemented. Charlie tells Matt not to worry, “We’ll work on `student` next.”

## 13.1.2 The student Class

Charlie and Matt review the `student` CRH card next. The only action responsibility is `canSelect`. The knowledge responsibilities are closely related—know remaining time credit and how many songs played today. Matt remarks that each student should know its ID number. After all, `studentCollection` will be asked to look up students based on ID numbers. Charlie suggests the CRH card be changed by adding “know ID number.”

FIGURE 13.2. *CRH card for student*

<b>Class:</b> student	
<b>Responsibilities:</b>	<b>Helpers:</b>
know remaining credit	date
know how many songs played today	
canSelect	
know ID number	

Charlie thinks it might also be necessary to search the Web for a date and time class to satisfy other knowledge responsibilities. However, the programmers decide to worry about that later—during implementation.

The team now considers the return type and parameters for the `canSelect` message. The student object needs to know if a certain track can be played. So a track object needs to be passed as an argument.

```
class student { // First draft--no constructors yet
public:
    bool canSelect(track currentSelection);
    // Returns true if the student has enough time credit and has
    // played zero or one track(s) on today's date
    // Update the state to reflect a played track
private:
    string my_ID;
    // TBA: Other data members might be date and time objects
};
```

Matt and Charlie decide to define `cdCollection` and `CD` next. (*Note:* The process is similar to what was done for the `studentCollection` and `student` classes.)

### Self-Check

- 13-1** This is a continuation of the college library system you began in Chapter 12. You should have found a `student` class in the college library. Design the class definition now as completely as possible.
- 13-2** This is a continuation of the college library system you began in Chapter 12. You should have found a `book` class in the college library. Design the class definition now as completely as possible.

### 13.1.3 The cdCollection Class

The `cdCollection` CRH card indicates that a `cdCollection` object must be able to add and remove CDs. It must also be able to store all CDs and let someone look at any or all CDs.

FIGURE 13.3. CRH card for `cdCollection`

Class: <code>cdCollection</code>	
Responsibilities:	Helpers:
know all CDs	vector of CDs
retrieve a CD	
<code>addNewCD(CD aCD, int trayNumber)</code>	
<code>removeCD(int trayNumber)</code>	
<code>CD getCD(int trayNumber)</code>	

Perhaps because Matt has just studied vectors, he wants to include one as a data member. Chelsea reluctantly agrees as she reminds Matt that this decision could wait. The underlying storage mechanism could change to a different container class.

Charlie says that since CD objects might be somewhat big, it makes sense to pass the CD object by const reference to `addCD` and `removeCD`. This leads to a first-draft definition for the `cdCollection` class:

```
class cdCollection { // First draft--no constructors yet
public:
    void addCD(const CD & aCD);
    // Add a CD to the collection if possible

    bool removeCD(const CD & aCD);
    // Remove CD if possible. Return false if it doesn't exist.

private:
    vector <CD> my_data;
    int my_size;
};
```

#### Self-Check

- 13-3** This is a continuation of the college library system you began in Chapter 12. If you found a container class for books or students, write the C++ class definition as completely as possible.

Matt now recalls Jessica’s complaint that as role player of `cdCollection`, she did not take part in the scenario of a user selecting a track. It reminds Matt that `cdCollection` must provide access to all CD objects. In fact, the CRH card states the `cdCollection` must “allow references to individual CDs.” Chelsea suggests that the process of sequentially iterating over a collection of objects from first to last is done so often that it has now been documented as a *design pattern*. “This is a classic instance of the Iterator design pattern,” claims Chelsea.

Charlie, who thinks he knows it all, admits that he is unaware of the Iterator design pattern. Chelsea reassures Charlie by informing him that the pattern is from a book written for an advanced audience, *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma/Helm/Johnson/Vlissides 95].<sup>1</sup> Charlie says he has studied other kinds of patterns—algorithmic and object patterns—but not any design patterns. Chelsea summarizes the Iterator pattern presented in the *Design Patterns* book:

---

DESIGN PATTERN 13.1

---

Pattern:	Iterator
Intent:	Provide a way to access the elements of a container object sequentially without exposing the underlying representation.
Motivation:	A container object, such as a bag of book objects, or a <code>cdCollection</code> of CD objects, should give access to its elements without exposing its internal structure. This allows the developer to specify an interface before the data members are chosen. It also allows the class designer to modify the internal structure of the container class without requiring changes in the code that uses the container. This pattern can be implemented with operations and a data member named <code>index</code> to <code>index currentItem</code> :  <pre>first() next() isDone() currentItem()</pre>

---

Matt and Charlie have both seen the Iterator pattern used with the bag class. Individual objects in bag containers were visited using `first`, `next`, `isDone`, and `currentItem` messages. From his CS2 class (weeks 16–30), Charlie knows about

---

1. These four authors, inspired by Christopher Alexander’s book *A Pattern Language* [Alexander 77], have been affectionately referred to as the Gang of Four (GOF).

changing underlying structure. Charlie's instructor asked him to modify underlying structures of container classes. The underlying data member used to store the items was first a `vector`, then a `list`. The data members and algorithms were dramatically different, yet the interface never changed! The programs using the containers did not have to be changed. Matt asks Charlie why his instructor asked him to change the underlying structure. Charlie tries to explain that there are different ways of doing the same thing, and that one way may be more appropriate than another when the collection gets really big: "Some containers are better when there is a small size, say in the hundreds. Other containers are more appropriate when there are millions of objects. Sometimes the objects need to be made persistent so they live beyond the program run—`vector` and `list` objects die off when the program ends." Matt is confused: "What has this got to do with iterators?"

Chelsea jumps in. She suggests that the design patterns were developed by professional software developers with a lot of combined experience and with contributions from many object-oriented software developers. Chelsea tells Matt to trust the pattern, even if it is not yet clear why the Iterator pattern leads to better design and a higher degree of usability.

The bag iterators were member functions. Although the *Design Patterns* book describes a more powerful and flexible design, its Iterator design pattern requires inheritance (see Chapter 16) and implementation of a separate iterator class. Charlie says he has an easier way to iterate over CDs in the `cdCollection`. The algorithm could look something like the following, where messages are sent to the appropriate public member functions of `cdCollection`.

```
// Iterate over cdCollection
for(theCdCollection.first() ! theCdCollection.isDone()
    theCdCollection.next());
{ // assert: theCdCollection has at least one more CD to visit
  aCD = theCdCollection.currentItem();
  // . . .
  // Do whatever you want with aCD
  // . . .
}
```

Charlie refines the class definition to include the iterator member functions. *Design Patterns* also recommends a private data member to refer to the items during traversals. It is named `my_index` here to match the conventions used in this textbook (precede all data members with `my_`).

```

class cdCollection { // Third draft
public:
    void addCD(const CD & aCD);
    // Add a CD to the collection

    bool removeCD(const CD & aCD);
    // Remove CD if possible. Return false if it doesn't exist.

    //--iterator functions
    void first(); // Let index point to the first item
    void next(); // Let index point to the next item (or end)
    bool isDone() const; // Return true when traversal is done
    CD currentItem() const; // Return the item referred to by next

private:
    vector <CD> my_data; // Store the CDs
    int my_size; // The number of CDs in the collection
    int my_index; // The subscript of the current item
};

```

Now there are four additional members—the iterator functions—and a private data member named `my_index` to help. Chelsea states that CRH cards cannot, and do not pretend to, provide all operations during analysis. Some things become obvious only during design and implementation. Object-oriented software development is an iterative process. Chelsea reminds Matt and Charlie, “Things change during refinement.”

## 13.1.4 The CD Class

The CRH card for CD indicates no action responsibilities.

FIGURE 13.4. *CRH card for CD*

Class: CD	
Responsibilities:	Helpers:
know tracks	
know CD title and artist name	
allow references to individual tracks	
know play time	

At first, it looks like CD will follow the State Object pattern. Each CD must know its artist, title, and tracks.

```
class CD { // First draft--no constructors yet
public:
    string artist() const;
    string title() const;
private:
    string my_artist;
    string my_title;
    vector <track> my_data; // Stores many tracks, typically 10
    int my_size;           // The total number of tracks
};
```

However, as with cdCollection, CD must allow someone to look over the individual tracks. The following algorithm uses an inner loop to iterate over all the tracks on one CD. The loop in int main() iterates over all CDs in the collection. Each of those CDs is passed to showOneCd that in turn iterates over all the tracks on the CD.

```
#include "cdcollec" // For the CD, track, and cdCollection classes

void showOneCD(CD & aCD)
{ // post: Show titles of all the tracks on aCD
    track aTrack;
    for( aCD.first(); ! aCD.isDone(); aCD.next() )
    {
        aTrack = aCD.currentItem();
        cout << aTrack.title() << endl;
    }
}

int main()
{
    cdCollection theCDs;
    CD aCD;

    for( theCDs.first(); ! theCDs.isDone(); theCDs.next() );
    {
        aCD = theCDs.currentItem();
        cout << aCD.title() << " " << aCD.artist() << endl;
        showOneCD(aCD);
    }

    return 0;
}
```

CD is a collection. “That Iterator pattern again,” observes Charlie. The following class CD refinement provides clients access to individual tracks:

```
class CD { // Second draft
public:
    string artist() const;
    string title() const;

    //-iterator functions
    void first();
    void next();
    bool isDone() const;
    track currentItem() const;

private:
    string my_artist;
    string my_title;
    vector <track> my_data;
    int my_size;
    int my_index;
};
```

Charlie wonders out loud how a CD should be initialized. If it didn’t have a vector of tracks, it would be easy to simply have a constructor set the artist and title. It is not clear where and how CD data is or will be stored. Can the data be read from the CD? Charlie says he can easily look at the number of tracks and play time using Microsoft’s “CD Player” program. But this doesn’t list the track titles or the artist’s name. As of this writing, many CDs do not yet contain that information.

Matt suggests the data could be obtained from the liner notes of each CD. The data could be maintained in a file. Chelsea suggests that the format of the CD data (and the student data) must be specified in a precise manner. Chelsea suggests the following format but warns that it might change as the CD and track classes are implemented and tested. This file provides at least some idea of how information will be stored:

```
#ARTIST Hashitani, Samantha
1 Samantha Hashitani
  1 3 4 Easy
  2 3 13 Step Aside
  3 0 26 Rokujo Interlude
  4 3 9 Rokujo
  5 4 46 Sweet
  6 3 11 Ain't that Lovin' You Baby
```

```

7 4 30 Faith Never Leaves
8 4 17 Falling in Love
9 4 0 Boy
10 2 58 Taming
11 4 51 Until Then
12 6 10 Parts
13 2 44 Carousel Song
#ARTIST Browne, Jackson
2 Looking East
  1 4 56 Looking East
  2 5 42 Barricades of Heaven
  3 4 51 Some Bridges
  4 5 14 Information Wars
  5 3 55 I'm the Cat
  6 5 45 Culver Moon
  7 5 5 Baby How Long
  8 5 14 Nino
  9 4 51 Alive in the World
10 4 57 It Is One
#ARTIST Raitt, Bonnie
3 Luck of the Draw
  1 3 48 Something to Talk About
  2 3 33 Good Man, Good Woman
  // . . .
  // Much data deleted. Collection ends with R.E.M., Out of Time . . .
  // . . .
10 4 9 Country Feedback
11 5 6 Me in Honey
#END

```

This file represents the data necessary to initialize CD objects. It could be created as part of a maintenance procedure. For now, the data will simply be typed into a plain text file.

Initializing one CD would involve reading the artist name, the tray where it is located in the physical CD player, and the title name. These are the first two lines beginning with #ARTIST. For example, the following two lines represent the artist's name (Raitt, Bonnie), the tray location in the physical jukebox (3), and the title of the CD (Luck of the Draw).

```

#ARTIST Raitt, Bonnie
3 Luck of the Draw

```

The #ARTIST label begins each and every new CD. #END is the end-of-file indicator—a sentinel. This data is followed by the tracks, listed on separate lines in a

format easily generated from CD liner notes or perhaps a CD reader if the liner notes neglect to list the play time. For example, the following line means the track number 1 takes 3 minutes and 48 seconds to play, and it has the title “Something to Talk About”:

```
1 3 48 Something to Talk About
```

Chelsea affirms that much progress has been made. At this point, the programmers could almost begin implementing the `student`, `studentCollection`, `CD`, and `cdCollection` classes. However, Matt remembers that `cdCollection` needs `CD` and `CD` needs `track`. So the programming team needs to define `track` before `CD` and `CD` before `cdCollection`.

### 13.1.5 The `track` Class

Each track must know its play time and where it is located in the physical CD player. Remember, the team decided that `cdPlayer` must be able to play a track. This means `track` has knowledge responsibilities for the CD number and the track number. The `track` class looks like another instance of the State Object pattern. It stores state and provides suitable access to it. Here is the CRH card.

FIGURE 13.5. CRH card for `track`

Class: <code>track</code>	
Responsibilities:	Helpers:
know play time	
know physical location in the CD player (CD number, track number)	

Charlie wonders about the return type of a `playTime` message. He has heard of a `time` class for manipulating time. Matt suggests that Charlie is going overboard. Play time could simply be stored as seconds. Chelsea argues that this may cause confusion since we think of play time in terms of minutes and seconds. Matt counters by stating that this is an implementation detail that the user will never see. Charlie

argues that using an integer to store seconds will require conversions such as seconds into minutes and seconds. Client code would have to deal with the conversions.

Matt insists on using an `int` as the return type from `playTime`. Charlie argues that 500 minutes is 30,000 seconds. On some older compilers, the maximum `int` is only 32,767. What will happen if Blaine asks us to increase the maximum time credit to 777 minutes? Chelsea says overflow could occur as time credit goes negative. Charlie recommends `long`, which can store more than 2 billion seconds, more than enough. Chelsea recommends the following type definition. `seconds` provides a more meaningful name than `long`.

```
typedef long seconds; // Seconds and long are interchangeable

class track { // First draft--no constructors yet
public:
    seconds playTime() const;
    // Returns the number of seconds required to play this track

    int trackNumber() const;
    // This track's location in the CD

    int cdNumber() const;
    // The tray location of the CD in the physical CD player

private:
    seconds my_playTime;
    int my_trackNumber;
    int my_cdNumber;
};
```

Charlie makes a mental note to also use `seconds` for the `student` class. Instead of starting with 500 minutes, students can start with  $500 * 60 = 30,000$  seconds of play time. It will be an easy matter to deduct play time from a student's credit using integer subtraction.

## 13.1.6 The cardReader Class

The cardReader CRH card currently has only one action responsibility: `getStudentID`

FIGURE 13.6. CRH card for cardReader

Class: cardReader	
Responsibilities:	Helpers:
<code>getStudentID</code>	physical card reader, which
	in turn collaborates with
	the magnetic student ID card

Assuming the ID will be stored as a string, the class definition is easy.

```
class cardReader { // First draft
public:
    string getStudentID() const;
    // Returns the user's ID number as a string
private:
    // TBA: Knows physical card reader
};
```

Until the money is there to buy the magnetic card reader, the `cdPlayer` class can simply read an ID number typed at the keyboard. The card reader is read as if it were the keyboard. Swiping a card through the card reader is equivalent to typing in the magnetically stored data. It will be easy to change the `cardReader` class later on after testing, as long as the class definition does not change.

## 13.1.7 The trackSelector Class

The `trackSelector` class has one major responsibility—`getTrack`. To accomplish this, `cdCollection` could be passed as an argument—by const reference because `cdCollection` will be fairly big and it should not be modified. The `trackSelector` object will show CDs and individual tracks on a selected CD in order to return the track desired by the user.

```
class trackSelector { // First draft
public:
```

```

    track getTrack(const cdCollection & theCDCollection) const;
private:
    // TBA: May not have any private data
};

```

### 13.1.8 The cdPlayer Class

The innocent-looking `cdPlayer` CRH card only has one major responsibility: `playTrack`

FIGURE 13.7. CRH card for `cdPlayer`

Class: <code>cdPlayer</code>	
Responsibilities:	Helpers:
<code>playTrack</code>	the physical CD player
	<code>playList</code>
	CD

The `cdPlayer` object will “talk” to the physical CD player that Ed Slatt implemented from Chris Dodge’s circuit design. The `cdPlayer` class also requires signal-capturing software to record the IR output to the CD player, use of the parallel port on the printer, some assembly language programming, and some complicated C++ code. However, `cdPlayer` proves to be the easiest class to deal with. Chris Dodge already completed most of it. It is already done. Here is the class definition for the `cdPlayer` class that acts as a wrapper around Chris’s code:

```

#ifndef CDPLAYER_H
#define CDPLAYER_H
#include "track" // For the track class
#include "irdev" // For the IRDEV class and the assembler routine
                // PlaySignal. The IRDEV class was implemented by
                // Chris Dodge.
class cdPlayer {
public:
    void initKenwoodDP_M7740();
    // Because of the idiosyncrasies of my old CD player, for now I
    // want to have an initialization function named after my CD
    // player. This initialization may not work on other brands! This
    // cdPlayer class could eventually allow the same program to
    // initialize different brands of CD players.

```

```

void playTrack (const track & currentSelection);
// Plays a track using the CD number and track number on that CD

private:
    bool SIMULATING; // If true allow testing; if false, music plays
    IRDEV my_cdPlayer; // The IRDEV class was written by Chris Dodge
    string TrayCode(int cdNumber); // Send codes to the CD player
    void delay(long n); // Need this delay to send signals
};

#endif

```

## 13.1.9 The jukeBox Class

The jukeBox class coordinates most activities.

FIGURE 13.8. CRH card for jukeBox

Class: jukeBox	Helpers:
<b>Responsibilities:</b>	cdPlayer
know current track	trackSelector
know current student	student
waitForUser	cardReader
processOneUser	studentCollection
	cdCollection

Charlie suggests that `waitForUser` is only a small part of coordinating activities. `waitForUser` should not be part of the public interface. The `waitForUser` responsibility could be part of `jukeBox::processOneUser`. Matt wants to know, “Who will send the `processOneUser` message?” Chelsea advises the team to consider that “the answer might be found if you try to write the main function.” Matt says, “That makes sense since `jukeBox` was always the object that got things going and because `int main()` is the first function executed.” Charlie goes on, “And if `jukeBox` could be made to recognize when the system should be running and when it should be shut down, the main function might look like this”:

```

#include "jukebox" // For the jukeBox class

int main() // First draft
{ // One possibility--still undecided

```

```

    jukeBox theJukeBox;

    while(theJukeBox.isRunning())
    {
        theJukeBox.processOneUser();
    }

    return 0;
}

```

This particular main function suggests two new public messages in the jukeBox interface: processOneUser and isRunning. The waitForUser member function moves to the private section. With the class definition suggested in int main, the single jukeBox object would contain most objects.

```

class jukeBox { // Second draft
public:
    void processOneUser();
    // Do whatever is necessary to process one user request

    bool isRunning();
    // True until jukeBox somehow detects that it's time to shut down

private:
    //--private operation
    void waitForUser();

    //--data members
    cardReader my_cardReader;
    cdCollection my_cdCollection;
    studentCollection my_studentCollection;
    trackSelector my_trackSelector;
};

```

At this point, the class definitions have been started. Some have been refined. A different team might have established different classes, operations, names, data members, and algorithms.

Chelsea feels the team is now ready to implement individual classes. Because cdCollection needs CD, and CD needs track, Chelsea decides to complete track, CD, and cdCollection. Once these classes have been individually tested, they can be integrated and tested as part of the bigger system.

Chelsea reports the IR device built by Ed is no longer working. Apparently the power supply is inadequate. So Chelsea asks Matt, a computer engineering major, to either fix the existing piece of hardware or build a new one. Matt will also imple-

ment `cardReader` and `trackSelector`. Chelsea reminds Matt to have `cardReader` read from the keyboard until the physical card reader comes. Matt is also charged with the task of procuring the physical card reader and determining how to get the student ID number from the data during a card swipe.

This leaves Charlie to implement `student` and `studentCollection`. Chelsea also asks Charlie, who has built circuits before, to help Matt with the hardware. With that, the implementation has been divided somewhat evenly amongst the three-member team.

## 13.2 Implementing Member Functions and Refining Class Definitions

Charlie and Matt understand the process of reading class definitions and implementing the member functions based on those definitions. However, in their previous experiences they have always been given class definitions that already

- ★ defined the constructors
- ★ documented member functions with pre- and postconditions
- ★ listed the data members

The class definitions had already been designed. They had to make few, if any, design decisions.

Chelsea encourages Matt and Charlie to proceed and make decisions on their own. “Select whatever data members you feel are appropriate. We have already made one design decision—use `vector` in the container classes.” Chelsea reviews the class definitions and writes a list of classes that, hopefully, will satisfy many knowledge responsibilities:

- ★ `int`
- ★ `string`
- ★ `seconds` (`typedef seconds` as equivalent to `long`)
- ★ `vector <track>`
- ★ `vector <CD>`
- ★ `vector <student>`

Charlie wonders if the standard C++ `list` class (see Chapter 14) would be better than `vector`. He can apply standard algorithms for finding students, adding students, removing students, and sorting the list of students. “Go for it, Charlie,” says Chelsea. “The system will be insulated from actual implementation details—that’s why we have public member functions and private data members.”

The next step in divvying up the work involves getting the classes into the proper header files and making sure they compile. Here are the three header files Chelsea created from the class definitions under refinement:

```
// -----
// File name: track.h
// This file defines an early version of the track class
// -----
#ifndef TRACK_H
#define TRACK_H

typedef long seconds;

class track { // Early draft
public:
    seconds playTime() const;
    // Returns the number of seconds required to play this track

    int trackNumber() const;
    // This track's location on the CD

    int cdNumber() const;
    // The tray location of the CD in the physical CD player

private:
    seconds my_playTime;
    int my_trackNumber;
    int my_cdNumber;
};

#endif

// -----
// File name: cd.h
// This file defines an early version of the CD class
// -----
#ifndef CD_H
#define CD_H

class CD { // Early draft
public:
    string artist() const;
    string title() const;
    //--iterator functions
    void first();
    void next();
    bool isDone() const;
    track currentItem() const;
```

```

private:
    string my_artist;
    string my_title;
    vector <track> my_data;
    int my_size;
    int my_index;
};

#endif

// -----
// File name: cdcollec.h
// This file defines an early version of the cdCollection class
// -----
#ifndef CDCOLLEC_H
#define CDCOLLEC_H

class cdCollection { // Early draft
public:
    void addCD(const CD & aCD);
    // Add a CD to the collection if possible

    bool removeCD(const CD & aCD);
    // Remove CD if possible. Return false if it doesn't exist.

//--iterator functions
    void first(); // Let index point to the first item
    void next(); // Let index point to the next item (false at end)
    bool isDone() const; // Return true when traversal is done
    CD currentItem() const; // Return the item referred to by next

private:
    vector <CD> my_data; // Store the CDs
    int my_size; // The number of CDs in the collection
    int my_index; // The subscript of the current item
};

#endif

```

These three class definitions can now be included from `int main()` with the file named `track`, which includes both files necessary to have `track` objects. This makes compilation and linking easy.

```

// File name: track
#ifndef _TRACK_
#define _TRACK_
#include "track.h"
#include "track.cpp"
#endif

```

Three other .cpp files (track.cpp, cd.cpp, and cdcollec.cpp) permit implementation and testing of these three classes. Even though the .cpp files are empty new files, they allow for the following start of a test driver, which compiles and creates an executable program that doesn't do much:

```
#include "track" // Includes the .h file and an empty .cpp file
#include "cd"    // For class CD
#include "cdcollec" // For the cdCollection class
int main()
{ // Invoke the compiler-supplied constructors
  CD aCD;
  track aTrack;
  cdCollection theCDs;
  return 0;
}
```

This program is executable due to a few facts about the C++ compiler:

- ★ Member functions are not needed until link time. If a member of a class definition is never called, no error occurs.
- ★ The C++ compiler automatically provides a default constructor for any class that has no constructor in the class definition.

So in actuality, these three constructors were automatically created by C++:

---

#### COMPILER-GENERATED DEFAULT CONSTRUCTORS

---

```
track::track()
{ // The C++ compiler creates this, unless the programmer does
}

CD::CD()
{ // The C++ compiler creates this, unless the programmer does
}

cdCollection::cdCollection()
{ // The C++ compiler creates this, unless the programmer does
}
```

---

Constructors have not been chosen yet for any of the jukebox classes. The team members will decide what to do about constructors and data member initialization while the data members are being selected. This happens next.

## 13.2.1 Testing Individual Classes

The framework has now been laid for Chelsea to get down to the task of implementing her three classes. The main function above can be used to test all three classes. Chelsea decides to take this approach:

1. Completely implement `track` and test it.
2. Completely implement `CD` and test it.
3. Completely implement `cdCollection` and test it.

## 13.2.2 Implementing `track`

The `track` class follows the State Object pattern. As usual, there is a default constructor necessary for storage in a vector. Chelsea also adds a constructor with four arguments. Chelsea notices that the team did not specify the track's title, so she adds `my_title` as a data member and an accessor member function named `title`. Here is the more refined class definition:

```
#ifndef TRACK_H
#define TRACK_H
#include <string>
using namespace std;

typedef long seconds;

class track {
public:
    //--constructors
    track();

    track( string initTitle, seconds initPlayTime,
           int initCdNumber, int initTrackNumber );
    // Initialize the private data of a track

    //--accessors
    string title() const;
    // Return this track's title

    seconds playTime() const;
    // Return the number of seconds required to play this track

    int cdNumber() const;
    // Where the CD is located in physical CD player (tray number)

    int trackNumber() const;
```

```

    // This track's location on the CD

private:
    string my_title;
    seconds my_playTime;
    int my_cdNumber;
    int my_trackNumber;
};

#endif

```

The track class took Chelsea about one hour to implement and test. Here is one test driver and the output it generated.

```

// This program test drives the simple track class. It should be
// run twice; once as is, and then with only the default object t1.
#include <iostream>
using namespace std;
#include "track" // For the track class

int main()
{ // Test drive track
    track t1;
    track t2("I'm the Cat", 3*60+55, 2, 5);

    cout << "Title: " << t2.title() << endl;
    cout << t2.playTime() << " seconds, which is also "
        << t2.playTime() / 60 << " minutes and "
        << t2.playTime() % 60 << " seconds" << endl;
    cout << "CD# " << t2.cdNumber() << endl;
    cout << "Track# " << t2.trackNumber() << endl;

    return 0;
}

```

---

## OUTPUT

---

```

Title: I'm the Cat
235 seconds, which is also 3 minutes and 55 seconds
CD# 2
Track# 5

```

---

Running the same test driver to test a default track object generated the following output:

```

Title: ?title?
0 seconds, which is also 0 minutes and 0 seconds

```

```
CD# 0
Track# 0
```

`track.cpp` and most of the other `.cpp` files are not shown in this chapter. The files with all their details can be found on the accompanying disk and at this textbook's Web site.

### 13.2.3 Implementing CD

During implementation, the programmers must decide on the data members that take care of the knowledge responsibilities. The programmers must also determine what, if any, constructors must be added to the class definition. Not until Chelsea considers adding constructors does it become obvious that an `addTrack` operation could prove useful. This will make it easier to get tracks added to a CD object. The algorithm might go like this:

```
while(there is another track to add to the CD)
{
    aCD.addTrack(aTrack)
}
```

Since this was an initialization issue that was purposely ignored during design, the `CD::addTrack` operation can be added during implementation. Chelsea logs her implementation efforts. One note states the following: “The test driver worked after some easy debugging. When I tried adding more than the default capacity tracks (10), it bombed. I forgot to resize the vector to be one bigger for each track over the default size of 10.”

Here is the test driver for CD. It hints at what `trackSelector` might do to show a selected CD.

```
#include "cd" // For the CD class

void show(const track & t2)
{
    cout.width(2);
    cout << t2.cdNumber();
    cout.width(3);
    cout << t2.trackNumber() << " ";
    cout << t2.title() << " " << (t2.playTime() / 60)
        << ":" << (t2.playTime() % 60) << endl;
}
```

```

int main()
{ // Test drive CD
  CD aCD("Browne, Jackson", "Looking East", 5);

  aCD.addTrack( track( "Looking East",          4*60+56, 5, 1 ) );
  aCD.addTrack( track( "Barricades of Heaven", 5*60+42, 5, 2 ) );
  aCD.addTrack( track( "Some Bridges",         4*60+51, 5, 3 ) );
  aCD.addTrack( track( "Information Wars",     5*60+14, 5, 4 ) );
  aCD.addTrack( track( "I'm the Cat",         3*60+55, 5, 5 ) );
  aCD.addTrack( track( "Culver Moon",         5*60+45, 5, 6 ) );
  aCD.addTrack( track( "Baby How Long",       5*60+05, 5, 7 ) );
  aCD.addTrack( track( "Nino",                5*60+14, 5, 8 ) );
  aCD.addTrack( track( "Alive in the World",  4*60+51, 5, 9 ) );
  aCD.addTrack( track( "It Is One",           4*60+57, 5, 10) );
  // Add a couple more than the default CD capacity size of 10
  aCD.addTrack( track("Fake Eleventh", 0, 5, 11) );
  aCD.addTrack( track("Fake Twelfth", 0, 5, 12) );

  cout << aCD.artist() << " " << aCD.title() << endl;

  // Iterate over all tracks in the CD
  track aTrack;

  aCD.first();
  while( ! aCD.isDone() )
  {
    aTrack = aCD.currentItem();
    show(aTrack);

    aCD.next();
  }
  return 0;
}

```

---

### OUTPUT

---

```

Browne, Jackson Looking East
5 1 Looking East 4:56
5 2 Barricades of Heaven 5:42
5 3 Some Bridges 4:51
5 4 Information Wars 5:14
5 5 I'm the Cat 3:55
5 6 Culver Moon 5:45
5 7 Baby How Long 5:5
5 8 Nino 5:14
5 9 Alive in the World 4:51
5 10 It Is One 4:57
5 11 Fake Eleventh 0:0
5 12 Fake Twelfth 0:0

```

---

The last two tracks cause the CD to expand beyond its capacity. Chelsea notes, “Later on, it may be appropriate to increase the default initial capacity (currently 10 tracks per CD) and the increase capacity by more than 1. That is because

`vector::resize` is an expensive operation—it takes some time. On the other hand, if the program runs out of memory, the size of the increase capacity may need to be decreased.” The member function includes some comments to this effect. However, Chelsea isn’t worried as long as there are no more than 200, 400, or even 600 CDs.

```
void CD::addTrack(const track & nextTrack)
{ // This function tries to balance time/space tradeoffs. You could
  // increase DEFAULT_NUMBER_OF_TRACKS or resize by two.
  if(my_size + 1 >= my_data.capacity())
    my_data.resize( my_data.capacity() + 1 );
  my_data[my_size] = nextTrack;
  my_size++;
}
```

The CD class definition is refined to the following:

```
// -----
// File name: cd.h
// This file defines an early version of the CD class
// -----
#ifndef CD_H
#define CD_H
#include <vector>
#include <string>
using namespace std;
#include "track" // For the track class

class CD {
public:
    CD();
    // Default constructor allows vector of tracks

    CD(string initArtist, string initTitle, int initCdNumber);
    // Initialize a CD with zero tracks

    void addTrack(const track & nextTrack);
    // Add a new track--missed this during design

//--accessors
    string artist() const;
    // Return this CD's artist

    string title() const;
    // Return the title of this CD

    int cdNumber() const;
    // Return physical location of this CD in CD player (tray number)
```

```

        track getTrack(int trackNum) const;

    int size() const;
    // Return the number of tracks in this CD

    //--iterator functions
    void first();
    void next();
    bool isDone() const;
    track currentItem() const;

private:
    string my_artist;
    string my_title;
    vector <track> my_data;
    int my_size;
    int my_index;
    int my_cdNumber;
};
#endif

```

The `getTrack` and `size` member functions were added after Chelsea realized they might be useful and important for the algorithm in `trackSelector`. For example, the `CD::size` accessor could be used to get a valid track number with code like this:

```

do {
    cout << "Enter track number [1..10]: ";
    cin >> trackNum;
} while (trackNum < 1 || trackNum > aCD.size());
cout << "You entered track " << trackNum << endl;

```

---

#### DIALOGUE

---

```

Enter track number [1..10]: 0
Enter track number [1..10]: -1
Enter track number [1..10]: 11
Enter track number [1..10]: 999
Enter track number [1..10]: 10
You entered track 10

```

---

This prevents users from entering track numbers that don't exist. `trackSelector` might also have trouble returning a track to `jukeBox` during the `getTrack` implementation. It is probably easy to get the CD number and a track number from the user. `trackSelector` can then use `CD::getTrack`.

```

track CD::getTrack(int trackNum) const
{
    return my_data[trackNum - 1];
}

```

Once `trackSelector` has a `CD` handy, it is easy to get the `track` object from the `CD` and return it to the client (`jukeBox`).

```

track aTrack( aCD.getTrack(trackNum) );
return aTrack;

```

The `cd.cpp` file containing the member function implementations is on the accompanying disk and at this textbook's Web site.

## 13.2.4 Implementing `cdCollection`

`cdCollection` is the most difficult of the first three classes to implement. Chelsea decides to have a default constructor do all the work. Here is the beginning of the default constructor that initializes the collection by reading from the file containing the `CD` data.

```

string CD_FILE_NAME = "cd.dat";

const int DEFAULT_NUMBER_OF_CDS = 200;

cdCollection::cdCollection()
{
    my_size = 0;
    my_index = -1;
    string fileName(CD_FILE_NAME) ;
    ifstream inFile(fileName.c_str());

    // Terminate right away if the file is not found
    if(! inFile)
    {
        cout << "***Error initializing CD collection from "
             << cdCollection::cdCollection() << endl;
        cout << "***Could not open '" << fileName
             << "'. Terminating program" << endl;
        exit(0);
    }
    // . . . Lots of initialization detail removed. See the file
    // cdcollec.cpp.

```

Here is the refined `cdCollection` class definition inside the file:

```

// -----
// File name: cdcollec.h
// This file defines an early version of the cdCollection class
// -----
#ifndef CDCOLLEC_H
#define CDCOLLEC_H
#include <vector>
#include <string>
using namespace std;
#include "cd" // For the cd class

class cdCollection {
public:
    cdCollection();
    // pre: The file with the CD data is in the right place
    // post: cdCollection represents all CDs in physical CD player

    void addCD(const CD & nextCD);
    // post: Add a CD to the collection if possible

    bool removeCD(const CD & aCD);
    // post: Remove CD if possible. Return false if it doesn't exist.

    int size() const;
    // post: Return the number of CDs in the collection

    CD getCD(int cdNumber) const;
    // pre: cdNumber >= 1 and cdNumber <= cdCollection::size()
    // post: Return an entire CD based on the CD# argument

    //--iterator functions
    void first(); // Let index point to the first item
    void next(); // Let index point to the next item (false at end)
    bool isDone() const; // Return true when traversal is done
    CD currentItem() const; // Return the item referred to by next

private:
    vector <CD> my_data; // Store the CDs
    int my_size; // The number of CDs in the collection
    int my_index; // The subscript of the current item
};

#endif // #ifndef CDCOLLEC_H

```

The `cdCollection` member function implementations are stored in the file `cdcollec.cpp` on the accompanying disk and at this textbook's Web site. The file

testcd.cpp contains the cdCollection test driver, which you could run in simulation mode—just follow the prompts. testcd.cpp is omitted here for space reasons. Even the output from this test driver is extensive. Only two of many screens are shown here.

---

OUTPUT (FIRST SCREEN FROM TEST DRIVING cdCollection WITH testcd.cpp)

---

```

1 Hashitani, Samantha      Samantha Hashitani
2 Browne, Jackson         Looking East
3 Raitt, Bonnie           Luck of the Draw
4 Sting                   Ten Summoners' Tales
5 Gabriel, Peter          Us
6 R.E.M                   Out of Time
Enter CD number [1..6]: 2

```

---



---

OUTPUT (NEXT SCREEN)

---

```

1 Looking East
2 Barricades of Heaven
3 Some Bridges
4 Information Wars
5 I'm the Cat
6 Culver Moon
7 Baby How Long
8 Nino
9 Alive in the World
10 It Is One
Enter track number [1..10]: 9
You picked:
=====
Looking East by Browne, Jackson
  title: Alive in the World
    time: 291
    cd#: 2
  track#: 9

Confirm <y/n>: y

```

---

At this point, a new team member, Jim Neumeyer, begins to work on the student and studentCollection classes. His development should be similar to CD and cdCollection. Therefore, to save space, these classes are not revealed in this chapter.

## 13.3 System Testing

At this point, most classes have been independently implemented and tested. The team is still waiting for `student` and `studentCollection`. Additionally, Matt admits he has not completed the `cardReader` class as suggested. “The money hasn’t been made available to buy the hardware. However, I do have it working so it asks the student to type in their ID.” Chelsea remarks, “Well at least that’s okay. We could simulate the running jukebox system by having a fake ID typed in. And even though `studentCollection` is not complete, we can test the other major classes together.” After a pause, Chelsea suggests the team could continue: “We’ll allow anyone to select a song. We won’t update the students in `studentCollection`. Let’s just see if we can integrate what we have.”

The team gets all their `.h` and `.cpp` files into the same directory on one computer. Chelsea makes a few changes with constant definitions and typedefs and stores the complete list in a file named `cddefs.h`, which is short for CD jukebox definitions.

```
// -----
// File name: cddefs.h
// Contains all typedefs and constant objects for jukebox classes
// -----
#ifndef CDDEFS_H
#define CDDEFS_H
typedef long seconds;
const int DEFAULT_NUMBER_OF_TRACKS = 10;
const string CD_FILE_NAME = "cd.dat";
const int DEFAULT_NUMBER_OF_CDS = 200;
#endif
```

Although some of the files had to be changed to include `cddefs.h`, the typedef and all global constants are now included in one file to make it easier to maintain the system. The final task now is to implement the `jukeBox` class.

From the analysis, it appears that `jukeBox` will be sending messages to most of the other major objects in the system. `jukeBox` will

- ★ ask `cardReader` for the student ID
- ★ ask `studentCollection` for a student
- ★ ask `trackSelector` for the user’s choice
- ★ send a message to `cdPlayer` to play a track

Somehow, `jukeBox` must have access to these objects. Consider that `trackSelector` has a *uses relationship* with `cdCollection`. `trackSelector` needs `cdCollection` to

fulfill its `getTrack` responsibility. To realize this relationship, the class was designed to have `cdCollection` passed as an argument to `getTrack`. But this is not the same as a `jukeBox` class that must send interesting messages to many objects. The jukebox could be designed to have these five objects passed as arguments to `jukeBox`: `cardReader`, `studentCollection`, `cdCollection`, `trackSelector`, and `cdPlayer`. Or referential attributes with pointers could be used. However, Matt doesn't know what pointers are and Charlie doesn't know what referential attributes are. Instead, Chelsea decides the jukebox should be designed with a *containment relationship*.

The containment relationship can be implemented by adding objects as data members. With a containment approach, `jukeBox` *contains* the objects as data members. "It does model a real-world jukebox," claims Matt.

```
#ifndef JUKEBOX_H
#define JUKEBOX_H
#include "cdcollec" // For the cdCollection class
#include "selector" // For the trackSelector class
#include "cdplayer" // For the cdPlayer class
#include "cardread" // For the cardReader class

class jukeBox {
public:
    jukeBox();
    // Initialize the cdPlayer and possibly do some other things

    void processOneUser();
    // Do whatever is necessary to process one user request

    bool isRunning();
    // True until jukeBox detects that it is time to shut down

private:
    //--data members
    student my_currentStudent;
    track my_currentSelection;
    cardReader my_cardReader;
    cdCollection my_cdCollection;
    studentCollection my_studentCollection;
    trackSelector my_trackSelector;
    cdPlayer my_cdPlayer;
};
#endif // #ifndef JUKEBOX_H
```

## 13.3.1 Assessing the Design

“Why is this a good design?” asks Matt. Chelsea defers to Riel’s object-oriented design heuristics book to illustrate the design heuristics followed by the jukebox design.

---

### OBJECT-ORIENTED DESIGN HEURISTIC 13.1 (RIEL’S 4.5)

---

If a class contains objects of another class, then the containing class should be sending messages to the contained objects. The containment relationship should always imply a uses relationship.

---

“This is true for `jukeBox`,” Charlie perceives. “`jukeBox` sends messages to all of its contained objects. The jukebox is *using* those objects to fulfill its responsibilities. This too is an example of the uses relationship. It is brought about through containment.” Chelsea introduces a few other object-oriented design heuristics that apply in this situation:

---

### OBJECT-ORIENTED DESIGN HEURISTIC 13.2 (RIEL’S 4.13)

---

A class must know what it contains, but it should not know who contains it.

---

This allows the contained classes to be reused in other applications. The `cdCollection` class could be used in a system that records your personal CDs, for example. `cdCollection` could be used elsewhere. In fact, `cdCollection` was individually tested. This implies it can stand alone. It does not need `jukeBox`.

Chelsea highlights another guideline that indicates the design is good.

---

### OBJECT-ORIENTED DESIGN HEURISTIC 13.3 (RIEL’S 4.14)

---

Objects that share lexical scope—those contained in the same containing class—should not have a uses relationship between them.

---

Sharing lexical scope means the objects could potentially send messages to each other. The five major objects in the `jukeBox` class are:

```
cardReader my_cardReader;  
cdCollection my_cdCollection;  
studentCollection my_studentCollection;  
trackSelector my_trackSelector;  
cdPlayer my_cdPlayer;
```

During role playing, the team could have sent a message to anyone around the table. Now, with the current design of `jukeBox`, any of these five objects could send a message to any of the others also. During role playing, no message was sent from `cardReader` to `cdCollection`. No message went from `cdPlayer` to `studentCollection`. Additionally, the current design implements what the team felt was reasonable during role playing. The `jukeBox` class—containing the major abstractions—was designed well.

The algorithm for `processOneUser` is now actually quite simple. This simplicity is possible because the system intelligence has been distributed evenly amongst the key abstractions. Each class was implemented and thoroughly tested by different members of the programming team. Here is the algorithm before `studentCollection` was added. One of the programming projects asks you to add the `studentCollection` class and change this `jukeBox::processOneUser`.

```
void jukeBox::processOneUser()
{ // Do whatever is necessary to process one user request

    while ( ! my_cardReader.isCardSwiped() )
        ; // Do nothing as the cardReader is continuously polled

    my_currentStudent = my_cardReader.getStudentID();

    // Need to get studentCollection before this is completed. For
    // now, just proceed as if every student were valid.

    my_currentSelection = my_trackSelector.getTrack(my_cdCollection);

    my_cdPlayer.playTrack(my_currentSelection);
}
```

All this makes it easy to create a very short system main function:

```
#include "jukebox" // For the jukebox class

int main()
{
    jukeBox theJukeBox;

    while(theJukeBox.isRunning())
    {
        theJukeBox.processOneUser();
    }

    return 0;
}
```

All the necessary files were stored into one directory so the system can be tested. Along the way, modifications were made to get the running system. The current state of the project is a running program that allows anyone to enter a bogus ID and select as many songs as desired (see the programming projects to change this). Currently, the tester has a choice whether to actually play the music or to simulate selections so the software can be tested and enhanced without the IR device and an attached CD player. However, the disk files allow you to run and test in simulation mode only, since it is highly unlikely you have the same CDs and CD player and the IR device attached to your printer port.

---

## Chapter Summary

- ★ This chapter showed how to proceed from analysis to design of an object-oriented system. Along the way, several object-oriented design heuristics indicated that team decisions resulted in good design.
- ★ Good design can be obtained and measured by such things as:
  - ★ high cohesion—member functions and data members belong together
  - ★ low coupling—allows easier partitioning of work amongst the team and easier unit testing
  - ★ proper return types and arguments
  - ★ proper messages
  - ★ understandable set of major classes with descriptive names to model a solution that is understandable to one and all
  - ★ actual application of object-oriented design heuristics—the jukebox contains objects that don't talk to each other, for example
- ★ Designing class definitions roughly bounds the design phase of object-oriented software development. Programmers choose
  - ★ member functions with appropriate return types, names, and arguments
  - ★ the data members
- ★ Implementing and testing the algorithms of the member functions roughly bound the implementation phase.
- ★ Action responsibilities may be implemented as member functions.
- ★ Knowledge responsibilities may be implemented as
  - ★ data members
  - ★ values returned from a message
  - ★ data gotten as an argument in a message

## Design/Implementation Tips

### 1. Refined CRH cards provide excellent input to the design of class definitions.

The better the CRH cards, the easier the class definitions are to design. Consider role playing your scenarios with class definitions in mind. The return types and arguments could be written directly on the CRH card. For example:

<b>Class:</b> trackSelector	
<b>Responsibilities:</b>	<b>Helpers:</b>
track getTrack(int)	the user
	track
	CD
	cdCollection

### 2. Look for divisions of labor.

Team members can divide some of the work up to completely implement and test individual classes. For example, one or two members of your team could be implementing track, CD, and cdCollection while other team members implement student and studentCollection. This can speed up a project.

### 3. If you say you're going to do it, do it!

A team is only as strong as its weakest member. If you don't show up for a meeting, you can hurt your team. If you commit to doing a specific task, don't let your team down. The whole project can fail if you don't do your part.

### 4. Learn to work as a team.

Industry wants students who know how to work with a team. The best way to learn this is to do it. Few people work by themselves. Analysis, design, and implementation benefit when people work together. Experience it. Be open to others' ideas. Learn to deal with diverse opinions. But don't be afraid to fight for what you believe in.

## 5. Take the time to write test drivers.

Each class you implement should be tested independently if possible. This may require writing a little program that does little more than call every member function of a class. However, this testing pays off in the long run.

## 6. If you need someone else's class, get it.

Pay attention to which class needs another class before it can be designed and implemented. For example, `jukeBox` needs just about every other class. So `jukeBox` might be defined last. In this case, your team members must make their classes available to you.

## 7. Sometimes a class declaration (not definition) will get you farther on down the line.

For example, consider that `trackSelector` needs `cdCollection` as a parameter. `jukeBox::getTrack` also needs `track` as a return type. The `trackSelector` class will compile with the following two class declarations standing in for the complete class definitions:

```
class cdCollection; // C++ allows these declarations to stand in for
class track;      // the class definitions

class trackSelector {
public:
    track getTrack(cdCollection & theCDCollection);
private:
    // No data members when cout and cin are used
};
```

Now someone could go ahead and implement `trackSelector::message`.

## 8. Sometimes a class definition (not the implementation) will get you farther on down the line.

Now, if the `cdCollection` interface were available before the member functions were implemented, you could `#include` it and then go ahead and try to compile `getTrack`—even though it will not link until the `cdCollection` member functions are done.

```
// File name: ui.cpp
#include "cdcollec" // Make cdCollection class definition available
class CD; // Still need CD class definition to compile
class track; // Still need track definition to compile
```

```

track trackSelector::getTrack(const cdCollection & theCdCollection)
{
    track aTrack; // Will compile at least
    CD aCD;
    theCdCollection.first();
    int n = 0;

    while( ! theCdCollection.isDone() )
    {
        aCD = theCdCollection.currentItem();
        // . . .
    }

    return aTrack;
}

```

## 9. Don't be surprised to see yourself adding new stuff.

You might find new data members and member functions as you design the class definitions and implement the member functions. Don't be surprised if arguments and return types change.

---

## Programming Projects—Complete the Jukebox

By completing the following programming projects, you will get the jukebox running in simulation mode. The major activity involves getting the student to become part of the system.

1. Run the program that simulates the running jukeBox and doesn't care what student ID you type in. You can select as many songs as you want. All files are in the jukebox directory on this textbook's disk. To run the program, you'll need many files. The file named `testjuke.cpp` looks like this:

```

#include "jukebox" // For the jukebox class
int main()
{
    jukeBox theJukeBox;

    while(theJukeBox.isRunning())
    {
        theJukeBox.processOneUser();
    }

    return 0;
}

```

2. Implement the `student` class and test it independently. Consider using the `Date` class stored in `date.h` and `date.cpp` to deal with `student::canSelect`. Test the function.
3. Implement the `studentCollection` class and test it independently. Enter several IDs to ensure that no one ID can select more than two tracks on a given date.
4. Integrate `student` into the jukebox system.

---

## Design/Implementation Projects

Each of the following design and implementation projects requires completion of the associated analysis and design project in the previous chapter.

### 13A Design and Implement 12A, Bank Teller Application

Make sure you complete 12A first. If you haven't already done so, read the "Analysis/Design Tips" section in Chapter 12 for help on how to do this. Then implement the bank teller system as a team. Also read the "Design/Implementation Tips" section of this chapter.

### 13B Design and Implement 12B, Voice Mail System

Make sure you complete 12B first. If you haven't already done so, read the "Analysis/Design Tips" section in Chapter 12 for help on how to do this. Then implement the voice mail system as a team. Also read the "Design/Implementation Tips" section of this chapter.

### 13C Design and Implement 12C, Video Rental System

Make sure you complete 12C first. If you haven't already done so, read the "Analysis/Design Tips" section in Chapter 12 for help on how to do this. Then implement the video rental system as a team. Also read the "Design/Implementation Tips" section of this chapter.

### 13D Design and Implement 12D, Checkbook Application

Make sure you complete 12D first. If you haven't already done so, read the "Analysis/Design Tips" section in Chapter 12 for help on how to do this. Then implement the checkbook system as a team. Also read the "Design/Implementation Tips" section of this chapter.