18

OO Languages

Chris Kelsey not only authored Chapter 18 in the text, but also prepared most of these answers. Brian Blaha also prepared some of the answers.

18.1 Figure A18.1 and Figure A18.2 add C++ and Java data types for each attribute. Measurements are real number types to allow fractions of inches/centimeters. The C++ *string* requires use of the Standard Template Library. The Java *String* is the class java.lang.String.

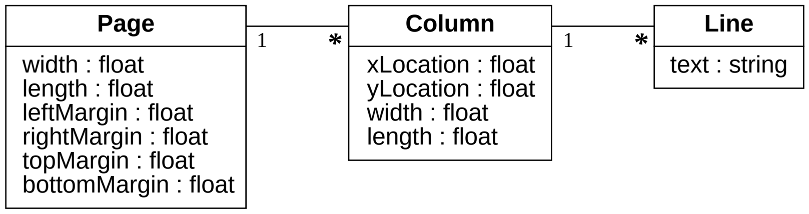

Figure A18.1 Portion of a class diagram of a newspaper with C++ data types

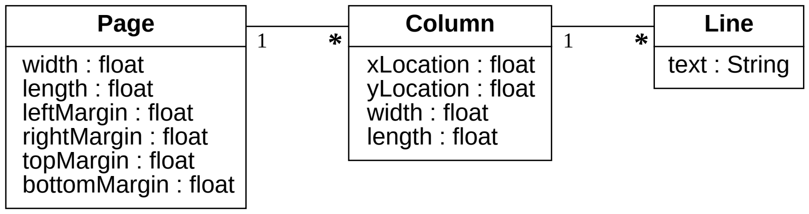

Figure A18.2 Portion of a class diagram of a newspaper with Java data types

18.2 Here are C++ declarations.

```
class Card {
};
class CardCollection {
};
class Hand : public CardCollection {
};
class Pile : public CardCollection {
};
class Deck : public Pile {
};
class DiscardPile : public Pile {
};
class DrawPile : public Pile {
};
```

18.3 Here are C++ classes. We omit *CardCollection.location* because a better model would handle that aspect in a separate graphics rendering layer. Note that the enumerations are public and the attributes are private. We implement the *CardCollection—Card* association by adding an attribute for each class.

```
class Card {
public:
   enum Suit { CLUB, DIAMOND, HEART, SPADE };
   enum Rank { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
      NINE, TEN, JACK, QUEEN, KING, ACE };
   enum CompareResult { LESS_THAN, EQUAL_TO, GREATER_THAN };
      //FOR 18.5
   CompareResult compare(Card& otherCard) const; // 18.5
   void discard(DrawPile& drawpile); // 18.5
private:
   Suit suit;
   Rank rank;
   CardCollection* cc; // current card collection
};

#include<list> // for collection, 18.4

class CardCollection {
public:
   enum Visibility { FRONT, BACK };
   virtual void initialize();
      // virtual to override in Deck 18.5
   virtual void insert(Card& card) = 0;
   virtual void discard(Card& card) = 0;
protected:
   Visibility visibility;
   // Point location;
   list<Card*> cardsInCollection; // exercise 18.4
};
```

```
class Hand : public CardCollection {
   int initialSize;
public:
   void insert(Card& card); // 18.5
   void discard(Card& card); // must override pure virtual
   bool deleteCard(Card& card); // "delete" is c++ keyword
   void sort();
};

class Pile : public CardCollection { // still abstract class
public:
   Card* topOfPile();
      // return pointer to card, so can be null, 18.5
   Card* bottomOfPile();
   void draw(Hand& aHand);
   void insert(Card& card);
   bool deleteCard(Card& card);
};

class Deck : public Pile {
public:
   void initialize(); // override to "no-op" 18.5
   void shuffle();
   Hand* deal( int nhands, int hsize );
   void discard(Card& card); // must override pure virtual
};

class DiscardPile : public Pile {
public:
   void discard(Card& card); // must override pure virtual
};

class DrawPile : public Pile {
public:
   void discard(Card& card); // must override pure virtual
};
```

18.4 See answer to Exercise 18.3.

18.5 See answer to Exercise 18.3.

18.6a. **One to one association traversed in both directions**. The solutions presented in the book (see p. 465) are extended here.

Several issues arise with bidirectional one-to-one associations. It is essential to build in safeguards to ensure that both sides of a link are linked or unlinked symmetrically, especially at construction and destruction of objects. Implementations differ according to whether links are mandatory or optional, whether they are changeable or of object-lifetime duration, and whether linking itself is a public operation or whether constraints dictate when or which objects may be linked.

The Java example in the book places emphasis on type independence. Classes *A* and *B* reside in different packages, and each uses a flag to terminate mutually recursive calls for linking and unlinking. Linking and unlinking operations are publicly available. The C++ example in the book allows *A* and *B* to accept requests to link (e.g. `A::setB(B& newB)`), but unlinking is performed only in the context of setting a new link, and is conducted through friendship. The linking object accesses the private data member of its linkee and directly manipulates it (e.g. `b->a = this;`).

In the extended versions we have assumed the link is 1) not mandatory (and so objects may be constructed without linking) and is 2) changeable, such that an object *A* may be linked to different objects of type *B* at different times, and 3) linking and unlinking can be publicly requested. Note the linking and unlinking method names have been changed to promote matching syntax across the classes.

In the extended C++ version, both *A* and *B* can accept public requests to link or unlink to their respective associates. The *A* side is responsible for actually executing the link; if a *B* is asked to do so, it delegates the duty to its prospective *A*. *A* has a small, private method that wraps the actual link exchange; the wrapper encapsulates the specifics of pointer exchange and can serve as an lockable block in a threaded environment. Note that copying and assignment are prohibited: To allow either (without redefining the default semantics of the operations) would allow the possibility of multiple links to a single target object. The destructor ensures unlinking to prevent dangling pointers in objects that outlive their linked partners. For unlinking, we've demonstrated a mutually recursive technique that ensures both sides detach, but without a need for private access.

The extended Java example parallels the C++. The book example's fine-grained packaging is relaxed—associated classes typically share a package, which allows friendship-like access by default. It is important that the class author tighten default access with explicit private specifiers to ensure collaborative classes do not remain unguarded against the ability to excessively and randomly reach into one another. Java does not allow object copying by default nor assignment by value, and so does not require the construction and assignment safeguards needed in C++.

In both, we've included dummy data and data accessor methods, should the reader wish to readily implement clients.

```
C++:

#include <string>
using namespace std;

class B;

class A {

    // prohibit copying/assigning A's to prohibit
    // multiple A's linked to same B...

    A(const A&);
```

```
   A& operator=(const A&);

   inline void exchangeLinks(B& aB);
      // A side has responsibility
      // of executing the link on both sides
   string data;
   B* b;

public:

   A(const char* dataStr) : data(dataStr), b(0) {}
   ~A() { UnLink(); } // ensure unlinking at destruction

   bool hasB() { return b != 0; }

   string Data() { return data; }
   B* myB() { return b; }

   inline string BData();
      // define after B implementation known

   inline bool Link(B& aB); // ensure bidirectional link...
   inline bool UnLink(); // and bidirectional unlinking too!
};

class B {

   // friendship declared by B grants to A::exchangeLinks the
   // privilege of access to B's non-public members, i.e. it
   // "disencapsulates" B for A only in the context of A's
   // link-exchanging operation.

   friend void A::exchangeLinks(B& aB);

   B(const B&);
   B& operator=(const B&);

   string data;
   A* a;

public:

   B(const char* dataStr) : data(dataStr), a(0) {}

   ~B(){ UnLink(); }

   string Data() { return data; }
   bool hasA() { return a != 0; }

   bool Link(A& anA) {
      if (a || anA.hasB() ) return false; // already linked
      return anA.Link(*this);
   }

   bool UnLink() {
      if (!a) return false; // nothing to unlink!
      A* aptr = a; a = 0;
      return aptr->UnLink();
   }

      // navigate to A data through B, or
      // allow B to expose it's linked A publicly...

   string AData(){ return a ? a->Data() : ""; } // someB.AData()
   inline A* MyA() { return a; } // someB.MyA->Data();
};


// deferred A implementations:

string A::BData() { return b->Data(); }

bool A::Link(B& aB) {
   if (b || aB.hasA() ) return false; // already linked
   exchangeLinks(aB); return true;
}

bool A::UnLink() {
   if (!b) return false; // nothing to unlink!
   B* bptr = b; b = 0;
   return bptr->UnLink();
}

void A::exchangeLinks(B& aB) { b = &aB; aB.a = this; }

int main()
{ ... }
```

Java:

```
// A.java

package oneone;

public class A {

   public A(String dataStr) { data = dataStr; }

   public boolean IsLinked(){ return b != null; }

   public String Data() { return data; }
   public B myB() { return b; }
```

```
      public String BData(){ return b.Data(); }

      public boolean Link(B aB){
         if (b != null || aB.IsLinked() ) return false;
            // already linked
         exchangeLinks(aB); return true;
            // ensure bidirectional link...
      }

      public boolean UnLink(){
         if ( b == null ) return false; // nothing to unlink!
         B bptr = b; b = null;
         return bptr.UnLink(); // and bidirectional unlinking too!
      }

      private void exchangeLinks(B aB){
         // A does the linking on both sides
         b = aB; aB.a = this;
            // B's x is package-access, not private
      }

      private String data;
      private B b = null;
   };



   // B.java

   package oneone;

   public class B {

      public B(String dataStr) { data = dataStr; }

      public String Data() { return data; }
      public boolean IsLinked() { return a != null; }

      public boolean Link(A anA) {
         if (a != null || anA.IsLinked() ) return false;
            // already linked
         return anA.Link(this); // let A do it
      }

      public boolean UnLink() {
         if (a == null) return false; // nothing to unlink!
         A aptr = a; a = null;
         return aptr.UnLink();
      }

         // navigate to A data through B, or
```

```
        // allow B to expose it's linked A publicly...

    public String AData(){ return a != null ? a.Data() : ""; }
    public A myA() { return a; }

    private String data;

    A a = null; // note package-level access instead of private!
    // or, could have private data + package-access
    // "mutator" method
}
```

b. Unordered one-to-many association traversed from the one to the many.

In navigating from one to many, note that the *Collection* is aware of its *Items*, but the *Items* have no awareness that they may be in a *Collection*. This presents a problem in the destruction of objects: If an *Item* is destroyed while linked into a *Collection*, the *Collection* will retain a dangling pointer to an *Item* that no longer exists.

To avoid these dangers, a robust implementation would exert some control over the creation and destruction of *Items* (i.e. the *Collection* might manage the lifecycles of *Items*, or a managing class would oversee the links), or *Items* would have links to *Collections* in which they appear. The dangers of dumb *Items* must be weighed against the additional dependencies created in having *Collections* manage *Items* or using bidirectional links, or the complexity of adding manager classes.

To more clearly demonstrate the logistics of links, we've elected to forgo the additional code that management would entail.

C++'s Standard Template Library includes the <*set*>. The set precludes inclusion of duplicates. A multiset is provided to implement bags.

Here, the one *Collection* may contain links to many *Items*:

```
#include <iostream> // for ListEm() debugging method

#include<string>
#include<set>
using namespace std;

class Item {

public:
    Item(const char* str) : data(str) {}
    string Data() { return data; }

private:
    string data;
};

class Collection {

public:
    bool Add(Item& item) { return items.insert(&item).second; }

    bool Remove(Item& item) {
       return(items.erase(&item) != 0);
    }

    bool InCollection(Item& item) {
       set<Item*>::iterator it = items.find(&item);
       return it != items.end();
    }

    Item* Retrieve(const char* dataStr) { // find by value
       /* c++'s STL employs value semantics -- containers retain
          copies of inserted objects. Here, we create a set that
          retains the pointer values of the inserted items
          rather than a set<Item> that would create copies of
          entire Items. Typically, a class encapsulating a
          collection of pointers provides a method to retrieve
          a pointer based on a data key. We leave this as a
          exercise for the reader. See also exercise 18.6d.
       */
    }

    // for debugging -- print item data values to console
    void ListEm() {
       set<Item*>::iterator it = items.begin();
       while (it != items.end())
          cout << (*it++)->Data() << endl;
    }

private:
    set<Item*> items;
};
```

Java: The issue of dangling pointers does not apply in Java. When an object is added to a Java *Collection*, the system creates a reference to it. The object will not be destroyed and garbage-collected until all references to it are destroyed. However, programmers may unwittingly keep objects alive if they are stored and forgotten in a Java *Collection*. (Note: *Collection* is a Java interface; the *Coll* in this example corresponds to *Collection* in the C++ version of the exercise.)

```
// Coll.java
package onemany;

import java.util.*;

public class Coll {

   private Set<Item> items = new HashSet<Item>();
```

```
    public boolean Add(Item item) { return items.add(item); }

    public boolean Remove(Item item) {
       return(items.remove(item));
    }

    public boolean InCollection(Item item) {
       return items.contains(item);
    }

    public void ListEm() {
       Iterator<Item> it = items.iterator();
       while (it.hasNext())
          System.out.println(it.next().Data());
    }
}

// Item.java
package onemany;

public class Item {

    public Item(String str) { data = str; }
    public String Data() { return data; }

    private String data;
}
```

c. Ordered one-to-many association traversed from the one to the many.

The inheritance structure of the C++ STL dictates that all containers are manipulated in similar fashion. The implementation of the ordered association would be essentially the same as the above, but users can impose an order on a *<set>* by specifying that order as an argument to the template instance. Alternatively, one might use the *<list>*, with similar syntax, but with the additional responsibilities of coding to locate the appropriate insertion point for an object/pointer.

See Exercise 18.6d, where the *ShareHolder* side demonstrates ordering of its many *Horses*. Supporting code demonstrates template support for determining the order of objects through their pointers.

Java's *Collections* framework distinguishes between the *Set* and *SortedSet* interfaces. To be eligible for inclusion in a *SortedSet*, a class *E* must implement the *Comparable* interface to describe the natural order of a user-defined type, or a *Comparator<E>* which describes the comparison that may be specified as an argument to the constructor of the implementing *TreeSet<E>* class, much as the C++ set can be constructed with a predicate object. See 18.6d.

d. Many-to-many, ordered one way and unordered the other.

Consider racehorse syndication. A horse can have many owners who hold shares in the horse, and an owner may invest in many horses. The following code demonstrates M:M where objects directly maintain links. No association class is used.

Sets are used for both Java and C++. Although the mathematical definition of set implies an unordered collection, both languages supply standardized library routines to impose and maintain order on sets in efficient time.

These simplified implementation assume insert/removal of a *Horse* in a *ShareHolder's* collection of *Horses* is successful; if not, we would need to undo insert/erase of *ShareHolder* within *Horse*. Because changes in *Horse* ownership can occur only through the (publicly available) *buy* and *sell* methods of the *ShareHolder*, updates occur only from the *ShareHolder* side (after all, a *Horse* does not determine its own ownership!). Therefore, we'll assume if the *Horse* accepts the *ShareHolder* change at the request of the *ShareHolder*, the *ShareHolder* can and will accept the update of its *Horses*. In circumstances where updates can be triggered from either side of an association, methods on both sides ensure the entire transaction is acceptable and complete on both sides.

```
#include <iostream>
#include<string>
#include<set>

using namespace std;

class ShareHolder;
class Horse {

   friend class ShareHolder;

   // to enforce buyer/seller control, these are private...
   bool AddShareHolder(ShareHolder& sh)
      {return shs.insert(&sh).second; }
   bool RemoveShareHolder(ShareHolder& sh)
      {return shs.erase(&sh) != 0;}

public:

   Horse(const char* nm): name(nm) {}
   string Name() const { return name; }
   void ListShareHolders();

   // specifies < for ordering of set
   bool operator<(const Horse& rhs) const
      { return name < rhs.name; }

private:

   string name;
   set<ShareHolder*> shs;
};
```

```
// Define a function-object to compare T's by pointer...
// Works with any class that implements operator <
// See class Horse, where < is implemented as alphabetical order

template<class T>
struct lessPtr : public less<T> {
   bool operator()(const T* const lhs, const T* const rhs) const
   { return *lhs < *rhs ;}
};

class ShareHolder {

public:

   ShareHolder(const char* nm) : name(nm) {}
   string Name() { return name; }

   bool Buy(Horse& horse)
      { return horse.AddShareHolder(*this) ?
      horses.insert( &horse).second : false ;
   }
   bool Sell(Horse& horse)
      { return horse.RemoveShareHolder(*this) ?
      horses.erase( &horse) != 0 : false;
   }

   // Typical C++ implementation of a derived attribute...
   int HorseCount() { return horses.size(); }

   void ListHorses();

private:

   string name;
   set<Horse*,lessPtr<Horse> > horses; // set order specified!!
};

void Horse::ListShareHolders()
{
   cout << "\n" << Name() << "'s owners:\n";
   set<ShareHolder*>::iterator it = shs.begin();
   while ( it != shs.end() ) cout << "\n " << (*it++)->Name();
   cout << endl;
}

void ShareHolder::ListHorses()
{
   cout << "\n" << Name() << "'s horses:\n";
   set<Horse*, lessPtr<Horse> >::iterator it = horses.begin();
   while ( it != horses.end() )
      cout << "\n " << (*it++)->Name();
   cout << endl;
}
```

Java:

```
//Horse.java

package syndicate;
import java.util.*;

/* We do not use the Comparator here, but instead rely on the
Horse's implementation of the Comparable interface, which is
expressed by the compareTo method. see notes in 18.6c
*/

public class Horse implements Comparable<Horse> {

   // to enforce buyer/seller control, these are package
   // access.
   boolean AddShareHolder(ShareHolder sh)
      { return shs.add(sh); }
   boolean RemoveShareHolder(ShareHolder sh)
      { return shs.remove(sh); }

   public Horse(String nm){ name = nm; }
   public String Name() { return name; }

   // implementing Comparable interface here:
   public int compareTo( Horse rhs ) {
      return Name().compareTo( rhs.Name()) ;
   }

   public void ListShareHolders()
   {
      System.out.println( Name() + "'s owners:");
      Iterator<ShareHolder> it = shs.iterator();
      while (it.hasNext())
         System.out.println(it.next().Name());
      System.out.println();
   }

   private String name;
   private Set<ShareHolder> shs = new HashSet<ShareHolder>();
}

// ShareHolder.java

package syndicate;
import java.util.*;

public class ShareHolder {
    public ShareHolder(String nm) { name = nm; }
    public String Name() { return name; }

    // Simplified implementation -- see C++ note

    public boolean Buy(Horse horse) {
        return horse.AddShareHolder(this) ?
        horses.add(horse) : false ;
    }
    public boolean Sell(Horse horse) {
        return horse.RemoveShareHolder(this) ?
        horses.remove(horse) : false;
    }

    public int HorseCount() { return horses.size(); }

    public void ListHorses() {
        System.out.println( Name() + "'s horses:");
        Iterator<Horse> it = horses.iterator();
        while (it.hasNext())
            System.out.println(it.next().Name());
        System.out.println();
    }

    private String name;
    private SortedSet<Horse> horses = new TreeSet<Horse>();
}
```

18.7a. The system should deallocate strings as they are no longer needed. The methods that modify strings could perform the deallocation. One way to combine strings is as follows.

```
    Determine total size of the strings that are to be combined.
    Allocate enough memory for the result.
    Copy the original strings into the allocated memory.
    Deallocate memory assigned to the original strings.
```

We assume that the old strings are no longer needed.

b. (1) **Forego garbage collection and rely on virtual memory**. The programmer does nothing and lets the operating system manage memory. Without garbage collection, virtual memory performance is bound to degrade over time. If, however, objects tend to be accessed and created sequentially, this approach may be acceptable, because references to contiguous objects will cluster and limit memory swapping.
(2) **Recover memory for each compiler pass**. You can allocate a large block of memory for all the objects in a pass and deallocate it all at once. Application software must manage the block of memory for each compiler pass.

c. You cannot let the operating system allocate a large amount of virtual memory and forget about garbage collection in systems that run indefinitely. Eventually all of the memory will be consumed. Long-running software must deallocate memory for objects that are no longer referenced. Consider writing your own memory allocation/deallocation function to replace the library function, taking the characteristics of your own objects into account.

d. The first approach works only in special circumstances in which the lifetime of the object is short. It is a dangerous approach that builds many assumptions into the code. A safer approach is for the caller to explicitly destroy the object.

18.8 Although Java packages affect access control, the primary logical purpose of packages is to group together classes that are used together, particularly those that have interdependencies among themselves. Applications can then readily import all related classes with a reasonable number of statements, rather than having to know and declare each and every class that will be required. To prevent unintentional dependencies among classes, it is important to specify access control as appropriate for each package member's individual data and methods rather than simply allow Java's default-when-unspecified package-level access control. Failure to privatize allows accidental package-wide disencapsulation and unwitting dependencies, while failure to publicize can produce a useful class that is virtually unusable by clients outside its package.

Package Competitors: Competitor, Team, League
Package Competition: Season, Meet, Event, Figure, Station
Package Scoring: Trial, Judge, Scorekeeper

18.9 The exercise specifies that all associations are bidirectional, requiring updates in both directions as links change. In this application, there are situations where certain updates should be prohibited or controlled. For example, the deletion of a *Figure* implies the deletion of an *Event* for that *Figure*, which implies the deletion of all *Trials* in that *Event*. This is not acceptable (at least under normal circumstances) for *Events* where the *Trials* have already occurred. Access control can help constrain unacceptable operations by encapsulating possibly dangerous methods from some or all callers. We demonstrate such techniques here; see comments in the code. We omit declarations for routine maintenance and queries to more clearly show the relevant operations. Skeleton implementation is shown for certain methods.

Java declarations would be similar to the C++ below, but packaged as described in Exercise 18.8. Although Java does provide garbage collection, the programmer must detach all links to ensure the object can be destroyed. Operations found in C++ destructors would be executed in a “regular” Java method to ensure full and timely unlinking of objects. In Java, actual destruction takes place under the control of the garbage collector (see a detailed Java reference for an explanation of Java object lifecycles), and so logical end-of-object-life operations may not coincide with the physical destruction of the object.

C++: In most cases, we have used a <*set*> to maintain the many side of an association. When queries are expected, the <*set*> offers efficient order/search facilities. See Exercise 18.6d for a demonstration of imposing order on a <*set*> (omitted here).

```
#include<string>
#include<list>
#include<set>

using namespace std;

// forward declarations and assumed utility class decl's omitted

// Figure has many events; events involve a single figure
class Figure {
   string figureTitle;
   float difficultyFactor;
   string description;
   set<Event*> events; // events in which figure appears

// The model dictates that Figures can stand alone, but each
// Event must specify a Figure. The destruction of any Figure
// that appears in an Event should propagate to the Event, which
// in turn dictates the destruction of any Trials hat have
// already performed in that Event...

// To ensure scores persist, then, Figures should only be
// destroyed if they have no links to Events, or links only to
// destructible Events (events that have no scored Trials). To
// enforce this, we privatize destruction and control it
// through a static member function that implements the
// appropriate constraints.

// note that a private destructor forces dynamic creation of
// Figures, i.e. Figure* f = new Figure(...); not Figure f(...)

~Figure() { /* delete events dependent on this figure */ }

public:
   Figure(const char* title, float factor, const char* descrip) :
      figureTitle(title), difficultyFactor(factor),
      description(descrip) {}

   static bool DeleteFigure(Figure& fig) {
      // control delete-figure constraints here... e.g.
      if (fig.events.size() == 0) { delete &fig; return true; }
      // ...
      return false;
   }

   // add event to figure's events
   bool AddEvent(Event& ev);
   // other maintenance and query methods
};

// Competitor has many trials
class Competitor {
   string name;
   int age;
   Address addr; // presumes Address utility class
   Phone telephoneNumber; // presumes Phone utility class
   list<Trial*> trials; // this competitor's trials
public:
   Competitor(const char* nm) : name(nm) { }

   bool AddTrial(Trial& t); // add trial to competitors list...

   // other maintenance methods...
   // query methods...
};

// utility class; although a struct's members are public by
// default, instances will remain encapsulated within a Trial
struct RawScore {
   float score;
   Judge* judge;
   RawScore(Judge& j) : score(0), judge(&j) {}

   // required for set<RawScore> as any set must be "orderable".
   bool operator<(const RawScore& rhs) const;
};

// a Trial is the participation of a single competitor in a
// single Event and has * judges associated with it...
class Trial {
   friend class Event; // allow Event to manage Trials

   float netScore;
   Competitor* who;
   Event* event;
   set<RawScore> scoring; // judge-score pairs

   // Trials exist only in the context of Events, so we delegate
   // the responsibility of creating and removing them to their
   // Event. Friendship ensures that only Events can call
   // Trial's ctor/dtor. See Event::AddTrial
   Trial(Competitor& comp, Event& ev) : who(&comp) {}
   ~Trial() { /* delete trial from competitor's and
                each judge's trial lists... */ }

public:
   void computeNetScore();
   bool AddJudge(Judge& j);
   // maintenance and query methods
};

// A judge participates in * trials, and assigns a single
// raw score per trial, i.e. judge and rawscore is 1:1, while
// trial and judge-score-pair is 1:Many
```

```
class Judge {
   friend class Trial; // Trial::AddJudge inserts trial
                       // in Judge's trial roster
   string name;
   set<Trial*> trialsJudged;

public:
   Judge(const char* nm) : name(nm) {}
   string Name() const { return name; }
   // query methods
};

inline bool Trial::AddJudge(Judge& j) {
   scoring.insert(*new RawScore(j));
   j.trialsJudged.insert(this);
   return true;
}

// an event is a group of trials (competitor preformances)
// over a particular figure
class Event {
   Time startingTime; // presumes a Time utility class
   Figure& figure; // reference member is a permanent link --
      // cannot change figures for event
      // trials in this event:
      // because Trials can exist only in the context of an
      // Event, we would typically use a list of instances, not
      // pointers, but the privatized Trial destructor forces
      // dynamic allocation to obtain pointers for explicit
      // deletion. Events will manage the construction and
      // destruction of their Trials.

   list<Trial*> trials; // trials in this event

// see Figure note on private destructor. Events should not
// be deleted if linked to scored trials.
~Event() {
   // delete this event from figure's event registry
   // cascade-delete trials dependent on this event
}

public:
   // an event is created for a specific figure...
   Event(Figure& fig, Time& start) : figure(fig),
      startingTime(start) { fig.AddEvent(*this); }

   static bool DeleteEvent(Event& ev) {
      /* ensure any linked trials are unscored; if ok, delete
      trials, then event */
   }

   bool AddTrial(Competitor& who) {
```

```
            trials.push_back(new Trial(who), this);
         }

         bool DeleteTrial(Competitor& who);
            // interface to Trial, e.g.
            // GetTrial(someone)->AddJudge(judgename);
            Trial* GetTrial(Competitor& who);
      };
```

18.10a. Both the C++ and Java examples below include simple implementations of the pseudocode from Exercise 15.11. Additional methods are implemented as needed for support or testing. C++ Note: Many of the methods here would be defined out of line; they are defined within the class declarations (unless declaration order prohibits it) for ease of reading.

```
#include<iostream>
#include<list>
#include<set>
#include<algorithm>
#include<iterator>
#include<string>

using namespace std;

#include <cstdlib>
#include <ctime>

class Random { // supports shuffle


public:
   Random() { srand( (unsigned)time( NULL )); }
      // seed generator
      // 0 <= i < max + 1
   int getRandom(int maxplus1) { return rand() % maxplus1; }
};

template<class T>
struct lessPtr : public less<T> { // supports sorting hands

   bool operator()(const T* const lhs, const T* const rhs)
      const { return *lhs < *rhs ;}
};

class Card;

class CardCollection;

class Pile;
class Card {
   friend class CardCollection; // collection sets card's cc
   static const char * SuitStr[]; // support console display
   static const char * RankStr[];

public:
   enum Suit { CLUB, DIAMOND, HEART, SPADE };
   enum Rank { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
      NINE, TEN, JACK, QUEEN, KING, ACE };
   enum CompareResult { LESS_THAN, EQUAL_TO, GREATER_THAN };
   Card(Rank r, Suit s) : suit(s), rank(r) { }

   // exercise specifies enum order as criterion for <. == , >
   CompareResult compare(const Card& otherCard) const {
      if (index() == otherCard.index()) return EQUAL_TO;
      return ( index() < otherCard.index() ) ? LESS_THAN :
         GREATER_THAN ;
   }

   // for ordering. < operator qualifies class for use with STL
   bool operator<(const Card& otherCard) const {
      return compare(otherCard) == LESS_THAN;
   }
   bool operator>(const Card& otherCard) const {
      return compare(otherCard) == GREATER_THAN;
   }

   // support dummy console display
   string Name() const {
      return string(RankStr[rank]) + string(SuitStr[suit]);
   }

   // dummy display for console output...
   void display() const { cout << Name() << " "; }

   // remove self from current collection;
   // insert in destination pile
   // delegates work to card's current collection;
   // defined after Pile
   void discard(Pile& destination);

   // supports testing..., not required
   // bool IsIn(const CardCollection& coll) const
      { return cc == &coll; }

private:
   int index() const { return suit * 13 + rank; }
   void setCollection(CardCollection* coll) { cc = coll; }

   Suit suit;
   Rank rank;
   CardCollection* cc;
};

// define statics
const char * Card::SuitStr[] = {"C","D","H","S" };
const char * Card::RankStr[] = {
      "2","3","4","5","6","7","8","9","T","J","Q","K","A" };

class CardCollection {
public:
   enum Visibility { FRONT, BACK }; // NOT USED HERE...
   typedef list<Card*> CC;

   // default "initialize" clears out collection
   // beware MEMORY LEAK!! don't clear out collections
   // without having stored the Card*'s elsewhere...
   // client application should have TrashPile to collect cards
   // for destruction...
   virtual void initialize() {
      cards.erase( cards.begin(), cards.end() );
   }

   // insert and discard/delete renamed to insertCard and
   // deleteCard to avoid confusion with c++ insert/delete and
   // Card::discard by default, add at top
   virtual void insertCard( Card& card ) {
      cards.insert(cards.begin(), &card);
         setCardCollection(card);
   }

   // error/exception not implemented.
   // could return "false" if card not found...
   virtual void deleteCard(Card& card) {
      CC::iterator it =
         find(cards.begin(), cards.end(), &card) ;
      if (it != cards.end() ) {
         cards.erase(it); clearCardCollection(card);
      }
   }

   // encapsulate implementations; expose utilities
   bool empty() { return cards.size() == 0; }
   int size() { return cards.size(); }

   // simple output of collection onto console
   void Show() {
      cout << endl;
      CC::iterator it = cards.begin();
      while (it != cards.end()) (*it++)->display();
   }

protected:
   Visibility visibility;
   // Point location;
   CC cards;

   // subs invoke base-level, befriended method to set card's
   // collection at insert. friendship is not inherited, so
   // overrides cannot access Card::setCollection

   void setCardCollection(Card& card)
      {card.setCollection( this ); }

   void clearCardCollection(Card& card)
      {card.setCollection( 0 ); }
};

class Hand : public CardCollection {
   int initialSize;

public:
   void insertCard(Card& card) {
      CC::iterator it = cards.begin();
         // find insert point
      while (it != cards.end() && **it < card ) it++;
      cards.insert(it,&card);
      setCardCollection(card); /*display...*/;
   }

   // no need to override except to add display.
   void deleteCard(Card& card){
      CardCollection::deleteCard(card) ; /*display()...*/;
   }

   // sort() renamed to avoid STL name confusion...
   // inserts are done in sort order, so this won't make any
   // changes...

   void sortCards() {
      // we can't use the STL's sort or list::sort because types
      // being compared must match the list item type, and we
      // can't define a comparison for built-in types -- which
      // all pointer types are! so... we'll make a <set> out of
      // the hand and copy it back to the list:

      set<Card*,lessPtr<Card> > tempset;
      CC::iterator it = cards.begin();
      while (it != cards.end()) tempset.insert(*it++);
      cards.erase( cards.begin(), cards.end() );
         // clear the list
      set<Card*,lessPtr<Card> >::iterator sit =
         tempset.begin();
      while (sit != tempset.end()) cards.push_back(*sit++);
   }
```

```
};

class Pile : public CardCollection {
public:
   // note: top of pile is head of list, not last in list...
   Card* topOfPile() { return empty() ? 0 : *cards.begin();}

   Card* bottomOfPile() {
      if (empty()) return 0;
      CC::iterator it = cards.end(); return *--it;
   }

   // draw from Pile into Hand
   void draw(Hand& aHand) {
      Card* card = topOfPile();
      if (!card) return; // return false...
      deleteCard(*card);
      aHand.insertCard(*card); // return true...
   }

   // no need to override except display;
   // CardCollection inserts are at top

   // void insertCard(Card& card) {
      CardCollection::insertCard(card) ; /*display()*/; }


   // remove only top card...
   void deleteCard(Card& card){
      if (&card != topOfPile()) return;
      CardCollection::deleteCard(card) ; /*display()*/;
   }
};

void Card::discard(Pile& destination){
   cc->deleteCard(*this);
   destination.insertCard(*this);
}

class Deck : public Pile {
public:
   Deck() { initialize(); } // constructor builds a deck...

   Hand* deal( int nhands, int hsize ) {
      if (nhands * hsize > cards.size()) return 0;
         // not enough cards!
      Hand* hands = new Hand[nhands]; // create the hands
         // conventional "around the table" deal
      for (int j = 0; j < hsize; j++ )
         for (int i = 0; i < nhands; i++)
            draw(hands[i]);
               // draw from deck to appropriate hand
```

```
        return hands;
    }

    // swap cards in existing deck N times
    void shuffle(int nTimes = 5) {
        CC::iterator it,it2;

        for (int i = 0; i < nTimes; i++ ) {
            it = cards.begin();
            while ( it != cards.end()) {
                it2 = cards.begin();
                advance(it2,r.getRandom(52));
                swap(*it++,*it2);
            }
        }
    }

    void initialize() {
        CardCollection::initialize();
            // clear what's left; see note at base
        Card* card;
        for (int i = 0; i < 52 ; i++) { create and insert cards
            cards.insert( cards.end(),
                card = new Card( static_cast<Card::Rank>(i % 13) ,
                    static_cast<Card::Suit>(i / 13 )));
            setCardCollection( *card );
        }
    }

private:
    static Random r; // supports shuffling
};

Random Deck::r;
// int main { ... }
```

b. Here is the answer in Java.

```
// Card.java:

package cards;

public class Card implements Comparable<Card> {

    static String [] SuitStr = {"C","D","H","S" };
    static String [] RankStr = {
        "2","3","4","5","6","7","8","9","T","J","Q","K","A" };

    public enum Suit { CLUB, DIAMOND, HEART, SPADE };
    public enum Rank { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
        NINE, TEN, JACK, QUEEN, KING, ACE };
    public Card(int r, int s) { suit = s; rank = r; }
    public Card(Rank r, Suit s) { suit = s.ordinal(); rank =
       r.ordinal(); }

    // Java requires an int be returned to meet contract of
    // Comparable interface
    public int compareTo(Card otherCard) {
       if (index() == otherCard.index()) return 0;
       return ( index() < otherCard.index() ) ? -1 : 1 ;
    }

    // support dummy console display
    public String Name() {
       return new String(RankStr[rank]) +
          new String(SuitStr[suit]);
    }

    // dummy display for console output...
    public void display(){ System.out.print( Name()+ " "); }

    // supports testing..., not required
    boolean IsIn(CardCollection coll) { return cc == coll; }

    // remove self from current collection; insert in
    // destination pile delegates work to card's current
    // collection;

    public void discard(Pile destination){
       cc.deleteCard(this);
       destination.insertCard(this);
    }

    private int index() { return suit * 13 + rank; }
    void setCollection(CardCollection coll) { cc = coll; }

    private int suit;
    private int rank;
    private CardCollection cc;
 };

 // CardCollection.java:

 package cards;
 import java.util.*;

 class CardCollection {

    enum Visibility { FRONT, BACK }; // NOT USED HERE...

    void initialize() { cards.clear(); }

    // by default, add at top
    // note package access for insert/delete;
    // transfers occur publicly through
    // Card.discard(destination) and
    // Pile.draw(destination hand)

    void insertCard(Card card ) {
       cards.add(0,card); setCardCollection(card);
    }

    void deleteCard(Card card) {
       cards.remove(card); clearCardCollection(card);
    }

    public boolean empty() { return cards.size() == 0; }
    public int size() { return cards.size(); }

    // simple output of collection onto console
    public void Show() {
       System.out.println();
       for(Card c : cards) c.display();
       System.out.println();
    }

    Visibility visibility;
    // Point location;

    ArrayList<Card> cards = new ArrayList<Card>();

    void setCardCollection(Card card)
       {card.setCollection( this ); }
    void clearCardCollection(Card card)
       {card.setCollection( null ); }
}

// Deck.java:

package cards;
import java.util.*;

public class Deck extends Pile {

    public Deck() { initialize(); }
       // constructor builds a deck...

    public Hand[] deal( int nhands, int hsize ) {
       if ( nhands * hsize > cards.size()) return null;
          // not enough!
       Hand hands[] = new Hand[nhands]; // create the hands
       for (int i = 0; i < nhands; i++) hands[i] = new Hand();
       for (int j = 0; j < hsize; j++ ) // deal "around the table"
       for (int i = 0; i < nhands; i++)
       draw(hands[i]); // draw from deck to appropriate hand
```

```
            return hands;
        }

        // swap cards in existing deck N times
        public void shuffle(int nTimes) {
            int ri;
            Card tmp;
            for (int i = 0; i < nTimes; i++ ) {
                for (int j = 0; j < 52; j++) {
                    ri = r.nextInt(52);
                    tmp = cards.get(j);
                    cards.set(j, cards.get(ri));
                    cards.set(ri, tmp );
                }
            }
        }

        public void shuffle() { shuffle(5); }

        void initialize() {
            super.initialize();
                // clear what's left; see note at base
            Card card;
            for (int i = 0; i < 52 ; i++) { //create and insert cards
                cards.add( card = new Card( i % 13, i / 13 ));
                setCardCollection( card );
            }
        }

        private static Random r = new Random(); // supports shuffling
    }

    // Pile.java:

    package cards;
    import java.util.*;

    class Pile extends CardCollection {

        // note: top of pile is head of list, not last in list...
        public Card topOfPile() {
            return cards.isEmpty() ? null : cards.get(0);
        }

        public Card bottomOfPile() {
            return cards.isEmpty() ? null :
                cards.get(cards.size() - 1);
        }

        // draw from Pile into Hand
        public void draw(Hand aHand) {
            Card card = topOfPile();
```

```
         if (card == null) return; // return false...
         deleteCard(card);
         aHand.insertCard(card); // return true...
      }

      // no need to override except display;
      // CardCollection inserts are at top

      void insertCard(Card card) {
         super.insertCard(card) ; /*display()*/; }

      // remove only top card...
      void deleteCard(Card card){
         if (card != topOfPile()) return;
         super.deleteCard(card) ; /*display()*/;
      }
   }

   // Hand.java:

   package cards;
   import java.util.*;

   public class Hand extends CardCollection {

      int initialSize;

      public void insertCard(Card card) {
         int ndx = 0;
         ListIterator<Card> it = cards.listIterator();
            // find insert point
         while (it.hasNext() && card.compareTo(it.next())> 0 )
            ndx++;
         cards.add(ndx, card);
         setCardCollection(card); /*display...*/;
      }

      // no need to override except to add display.
      void deleteCard(Card card){
         super.deleteCard(card) ; /*display()...*/;
      }

      // inserts are done in sort order,
      // so this won't make any changes...

      void sortCards() {
         TreeSet<Card> tempset = new TreeSet<Card>(cards);
         cards.clear();
         Iterator<Card>it = tempset.iterator();
         cards.addAll(tempset);
      }
   }
```

18.11 For a typical application, these kinds of queries would normally be handled using database code. So we do not provide a C++ or Java answer to this question.

18.12 Implement all associations involving *Box*, *Link*, *LineSegment* and *Point*.

Note that the composition relationships for *Box* and *Link* a in the exercise re incorrect. According to the definition of composition, a constituent part has a coincident lifetime with its assembly. This certainly is not the case for a diagram editor as a *Box* can be in a *Sheet* and then cut and placed in the *Buffer*. We should have used an aggregation relationship (a hollow diamond instead of a solid diamond). Our solution to the exercise assumes this correction.

- *Box* has * (0..n) *Links.* This is implemented as *set<Links*>* links within *Box*. The link destructor detaches the *Link* from its *Boxes* at destruction.
- *Box* has a mandatory aggregation to one *Collection.* This is implemented as a *Collection&* member of *Box*. Using a reference member ensures a *Box* cannot be created in the absence of its *Collection*. In addition, the *Box* constructor is private and *Box* creation is executed by the *Box* friend *Collection*, through the method *Collection::createBox*.
- *Link* must be associated with exactly two *Boxes. Link* contains *Box* boxes[2]*. The *Link* constructor takes two *Box&*'s as arguments, ensuring the *Link* is created between two existing boxes.
- *Link* has a mandatory aggregation to one *Collection.* This is implemented by privatizing the *Link* constructor and constraining *Link* creation to the method *Collection::createLink*. The *Collection&* attribute of *Link* is derived through the member function *Link::myCollection()*. Because *Links* cannot exist without their *Boxes*, and because a *Box* cannot exist without its *Collection*, a *Link* can readily determine its *Collection* by asking its *Box(es)*.
- *Link* has * (0..n) *LineSegments.* The points defining a link's line segments are embedded as a *list<Point>* member within the link itself.
- *LineSegment* has a mandatory composition to one *Link.* This is implemented by embedding the segment-defining points within the *Link* itself.
- *LineSegment* is associated with exactly two *Points*. *LineSegment* objects could be dynamically created as needed from two of a link's points. We have not implemented a physical *LineSegment* class, which would most likely be associated with the graphical elements of the editor and used as an interface to the link.
- *Point* can participate in one or two *LineSegments*. *Points* are implemented as “value objects”—they have no logical behavior or associations of their own. They serve purely as data values. We could enforce the model’s prescription that *Points* participate in at least, and no more than, one or two *LineSegments* by comparing their data values to existing *Points* in *Links*, or by promoting *Points* to “reference objects” with constrained construction and/or behavior. It is likely that any constraints on the loci of drawing objects would be imposed and translated not by the application’s internal

model, but by the display components of the application. The costs of promoting *Points* are not justified by their behaviorless use.

The following code answers Exercise 18.12 through 18.15.

```
#include<list>
#include<set>

#include<string>
#include<iostream>

//---------------------------------------------

// POINT IS A SIMPLE, UTILITARIAN VALUE-BASED OBJECT...
// IT WILL BE USED AS AN ENCAPSULATED ATTRIBUTE BY ITS CLIENTS

struct Point {
   int x, y;

   Point(int xCoord = 0, int yCoord = 0) :
      x(xCoord), y(yCoord) {}

   bool operator==(const Point& other) const {
      return ( x == other.x && y == other.y );
   }
};

//-------------------------------------------------

class Box{
   friend class Collection;
      // COLLECTION MANIPULATES ITS BOXES...
   friend class Link;

   string text; // not used ...
   int left, top, width, height;
   bool selected;

   set<Link*> links; // A BOX CAN PARTICIPATE IN 0..n LINKS
   Collection& collection;
      // A BOX MUST PARTICIPATE IN ONE COLLECTION

   Box(); // PROHIBIT DEFAULT, COPY CONSTRUCTION
   Box( const Box& other );
   Box& operator=( const Box& other ); // PROHIBIT ASSIGNMENT

   set<Link*>& myLinks() { return links; }
      // ACCESSOR FOR FRIENDS

   // UNLINKING NOT PUBLICLY AVAILABLE;
   // ONLY FRIENDS CAN DO THIS...
```

```
      // REMOVES A SINGLE LINK FROM links SET
      bool unLink(Link& lnk) {
         set<Link*>::iterator it = links.find(&lnk);
         if (it == links.end()) return false; // invalid link

         links.erase(it); // ELIMINATE THE LINK FROM THIS BOX
         return true;
      }

      // REMOVE ALL LINKS FROM links SET
      void unLinkAllLinks() {
         links.erase(links.begin(), links.end());
      }

      // FOR USE BY PUBLIC AllConnectedBoxes()
      void AllConnectedBoxes(set<const Box*>& tmp) const;

      // BOX MUST EXIST WITHIN CONTEXT OF COLLECTION...
      // TO ENFORCE MODEL, CTOR IS PRIVATE AND COLLECTION IS FRIEND
      Box(Point& upperLeft, int wid, int ht, Collection& coll ) :
         top(upperLeft.y), left(upperLeft.x), width(wid),
         height(ht), selected(false),collection(coll){
      }

   public:

      // EXERCISE 18.13
      inline void cut(); // DELEGATES TO COLLECTION
         // BEWARE DANGLING PTRS/REFS TO CUT BOXES!

      ~Box();

      const Collection& myCollection() { return collection; }

      void select() { selected = true; }
      void deselect(){ selected = false; }

      bool toggleSelect() {
         return selected = selected ? false : true;
      }
      bool isSelected() { return selected; }

      // EXERCISE 18.15 a & b
      set<const Box*> DirectlyConnectedBoxes() const;
      set <const Box*> AllConnectedBoxes() const;

      // FOR DEBUG AND DEMO
      set<const Link*> Links();
   };

   //--------------------------------------------
```

```
class Link{
   friend class Collection;
      // COLLECTION MANIPULATES ITS LINKS...

   Box* boxes[2]; // LINK LINKS EXACTLY 2 BOXES

   list<Point> segmentPoints;
      // LINE SEGMENTS ARE DEFINED BY
      // A SEQUENCE OF POINTS
   bool selected;

   Link();
   Link( const Link& other );
   Link& operator=( const Link& other );

   // PRIVATE CONSTRUCTOR: PREVENT CONSTRUCTION IF PARAMS
   // ARE INVALID
   // LINK CTOR IS CALLED BY Collection::createLink (EX. 18.14)
   Link( Box& box0, Box& box1, list<Point>& points );

   // DETACH LINK FROM ITS BOXES...
   inline void detach();

public:

   ~Link() { detach(); } // UNATTACH BOXES

   void select() { selected = true; }
   void deselect(){ selected = false; }

   bool toggleSelect()
      { return selected = selected ? false : true; }
   bool isSelected() { return selected; }

   // 18. 11
   // DERIVED ATTRIBUTE: A LINK MUST HAVE BOXES TO LINK, AND
   // THOSE BOXES BELONG TO THE SAME COLLECTION AS THE LINK...
   const Collection& myCollection() const
      { return boxes[0]->myCollection(); }

   // EXERCISE 18.15.c
   // IS THIS LINK PART OF A DIRECT LINK INVOLVING ARGUMENT box?
   bool linksTo(const Box& box) const {
      return( boxes[0] == &box || boxes[1] == &box );
   }

   // EXERCISE 18.15.d
   const Box* OtherBox(const Box& box ) const {
      if (! linksTo(box)) return 0;
         // this DOES NOT LINK TO box IN THIS LINK
      return boxes[0] == &box ? boxes[1] : boxes[0] ;
   }
   // EXERCISE 18.15.g
   list<Point> SegmentPoints(const Box& start, const Box& end);
};

//----------------------------------------------

class Collection {
   set<Box*> boxes;
   set<Link*> links;

public:

   Box* createBox( Point& upperLeft, int wid, int ht );

   // EXERCISE 18.13
   bool cutBox( Box& box );

   // EXERCISE 18.14: METHOD TO LINK TWO BOXES.
   Link* createLink(Box& b1, Box& b2, list<Point>& pointlist);

   bool cutLink( Link* link );

   set<const Link*> Links() const;
   set<const Box*> Boxes() const;

   // EXERCISE 18.15.e
   list<const Link*> SharedLinks(const Box& b1, const Box& b2);

   // EXERCISE 18.15.f
   list<const Link*> LinksSelectedAndUnselected();
};
```

Implementations:

```
// BOX.CPP ----------------------------------------------

Box::~Box()
{
   set<Link*>::iterator it = links.begin();
   while (it != links.end()) collection.cutLink(*it);
}

// EXERCISE 18.13
void Box::cut()
{
   collection.cutBox(*this); // deleteS BOX; BEWARE DANGLERS!
}

// EXERCISE 18.15.a
// USE SET TO AVOID DUPLICATES WHEN THERE ARE DUPE LINKS!
set<const Box*> Box::DirectlyConnectedBoxes() const {
   set<const Box*> connects;
   set<Link*>::const_iterator it = links.begin();

   while( it != links.end())
      connects.insert( ((*it++)->OtherBox(*this)));

   return(connects);
}

// EXERCISE 18.15.b
set <const Box*> Box::AllConnectedBoxes() const {
   set<const Box*> bxs;
   bxs.insert(this);
   AllConnectedBoxes(bxs);
   bxs.erase(this);
   return bxs;
}

void Box::AllConnectedBoxes(set<const Box*>& bxs) const {
   set<const Box*> directconnects = DirectlyConnectedBoxes();
   set<const Box*>::iterator boxIt = directconnects.begin();

   while( boxIt != directconnects.end()) {
      // FOR EACH DIRECT CONNECT

      if ( bxs.find(*boxIt) != bxs.end() ) return;
         // BEEN THERE

      bxs.insert( *boxIt);
      (*boxIt++)->AllConnectedBoxes(bxs); // GET ITS CONNECTS
   }
}

set<const Link*> Box::Links()
{
   set<const Link*> myLinks;
   set<Link*>::iterator it = links.begin();
   while (it != links.end()) myLinks.insert(*it++);
   return myLinks;
}

// LINK.CPP ---------------------------------------------

// DERIVED ATTRIBUTE...
//const Collection& Link::myCollection() const
// { return boxes[0]->myCollection(); }

// LIST OF POINTS SHOULD BE VALIDATED BEFORE CONSTRUCTION IS
// ALLOWED. LIST GOES FROM BOX0 TO BOX1
Link::Link( Box& box0, Box& box1, list<Point>& points ) :
   selected(false)
{
   boxes[0] = &box0; boxes[1] = &box1;
   segmentPoints = points; // COPY PARAM LIST!
}

// LINK ASKS BOXES TO DETACH THEMSELVES, THEN NULLS
// OWN RECORDS OF BOX ATTACHMENTS
void Link::detach()
{
   if (boxes[0]) { boxes[0]->unLink(*this); boxes[0] = 0; }
   if (boxes[1]) { boxes[1]->unLink(*this); boxes[1] = 0; }
}

// EXERCISE 18.15.g
list<Point> Link::SegmentPoints(const Box& start,
   const Box& end)
{
   if (( &start == boxes[0] ) && (&end == boxes[1] ))
      return segmentPoints;

   list<Point> lst;

   if (( &start == boxes[1] ) && (&end == boxes[0] )) {
      segmentPoints.reverse();
      lst = segmentPoints; segmentPoints.reverse();
   }

   return lst;
      // LST WILL BE EMPTY IF A BOX IS INVALID FOR THIS LINK
}

// COLLECTION.CPP ------------------------------------------
// CONTROLLED CONSTRUCTION: BOX & LINK MUST BE "PART OF" A
// COLLECTION (ALTHOUGH A COLLECTION NEED NOT HAVE ANY OF
// EITHER). TO ENSURE THE ASSOCIATION, AND TO ALLOW SAFE
// DELETION OF BOXES AND LINKS, WE'LL DELEGATE CREATION OF BOXES
// AND LINKS TO THEIR COLLECTION, AND GUARANTEE ALL ARE
// DYNAMICALLY ALLOCATED AND THEREFORE SAFE TO DELETE...

Box* Collection::createBox( Point& upperLeft, int wid, int ht )
{
   Box* newBox = new Box(upperLeft, wid, ht, *this);
   boxes.insert(newBox);
   return newBox;
}

// EXERCISE 18.14
Link* Collection::createLink(Box& b1, Box& b2,
   list<Point>& pointlist)
{
   // NOT ENOUGH POINTS... AT LEAST TWO REQUIRED FOR DIRECT LINK
   if (pointlist.size() < 2 ) return 0;
```

```
    // VALIDATE POINTS -- ARE THEY A PROPER POINT SEQUENCE?
    // DO START AND END POINTS FALL IN BOUNDS FOR SPECIFIED
    // BOXES?? DO POINTS CONNECT TO LINE?

    // IF INVALID, return 0;

    // PUT POINTS IN ORDER IF NECESSARY...

    Link* newLink = new Link(b1,b2,pointlist);

    links.insert(newLink); // ADD TO COLLECTION'S LINK REGISTRY

    b1.links.insert(newLink); // ADD TO BOXS' LINK REGISTRIES
    b2.links.insert(newLink);

    return newLink;
}

// EXERCISE 18.13

// THE CUT OPERATIONS WILL BE THE RESPONSIBILITY OF
// THE CONTROLLING COLLECTION...

bool Collection::cutBox( Box& box )
{
    set<Box*>::iterator boxIt = boxes.find(&box);
    if (boxIt == boxes.end()) return false;
        // THIS BOX NOT IN THIS COLLECTION

    // FOR EACH LINK IN BOX, DESTROY THE LINK

    if (box.myLinks().size() > 0 ) {
        set<Link*>::iterator linkIt =
            (*boxIt)->myLinks().begin();
        while (linkIt != (*boxIt)->myLinks().end())
            cutLink(*linkIt++);
}

boxes.erase( boxIt ); // REMOVE FROM boxes REGISTRY
    delete &box; // DESTROY (cut) the box
    return true;
}

bool Collection::cutLink( Link* link )
{
    if (!link) return false;

    set<Link*>::iterator it = links.find(link);
    if (it == links.end()) return false;

    links.erase(it); // REMOVE FROM COLLECTION'S links REGISTRY
    delete link; // DESTROY
```

```
   return true;
}

// EXERCISE 18.15.e DETERMINE ALL LINKS BETWEEN TWO BOXES
list<const Link*> Collection::SharedLinks(const Box& b1,
   const Box& b2)
{
   list<const Link*> sharedlinks;
   set<Link*>::iterator it = links.begin();

   while (it != links.end()) {
      if ( (*it)->linksTo(b1) && (*it)->linksTo(b2) )
         sharedlinks.insert(sharedlinks.end(),*it);
      it++;
   }

   return sharedlinks;
}

// EXERCISE 18.15.f
list<const Link*> Collection::LinksSelectedAndUnselected()
{
   list<const Link*> linksEm;
   set<Link*>::iterator it = links.begin();

   while (it != links.end()) {
      if ( ((*it)->boxes[0]->isSelected() &&
         !(*it)->boxes[1]->isSelected()) ||
         ((*it)->boxes[1]->isSelected() &&
         !(*it)->boxes[0]->isSelected()) )

         linksEm.insert(linksEm.end(), *it);

      it++;
   }

   return linksEm;
}

// FOR DEMO/DEBUG
set<const Box*> Collection::Boxes() const
{
   set<const Box*> bxs;
   set<Box*>::const_iterator it = boxes.begin();
   while (it != boxes.end()) bxs.insert(*it++);
   return bxs;
}

set<const Link*> Collection::Links() const {
   set<const Link*> lnks;
   set<Link*>::const_iterator it = links.begin();
   while (it != links.end()) lnks.insert(*it++);
   return lnks;
}

//---------------------------------------------
```

18.13 Implement the *cut* operation on the class *Box*. See answer to Exercise 18.12.

- *Box::cut()* delegates its work to *Collection::cutBox(Box& box)*. The *Box* reports its set of *Links* to the *Collection*. For each *Link* in *Box's set<Link*>*, the *Collection* applies its *cutLink(Link&)* method. In *cutLink(Link&)*, the *Collection* asks the *Link* to detach itself from both of its *Boxes*, then removes the *Link** from its set of *Links*, and deletes the *Link* (recovers memory) itself. The *Link's Points* are not dynamically allocated, and so are automatically deallocated as part of *Link* destruction. The *Collection* then removes the *Box* from its set of *Boxes*, and deletes the (now *Link*-less) *Box*.

18.14 Write a method to create a link between two boxes. Inputs are two boxes and a list of points. Also write a method to destroy a link. See answer to Exercise 18.12.

- *Link* Collection::createLink(Box& b1, Box& b2, list<Point>& pointlist)* performs this service. The *Link* constructor itself is privatized, and *Links* are created only after the *Collection* validates the inputs. The new *Link* creates its own *LineSegments* based on the *Point* list. The *Collection* notifies the *Boxes* to record the *Link* in their *Link* registries, and records the *Link* in its own registry of *Links*.
- *Link* destruction is provided by *bool Collection::cutLink(Link* link)*. The *Collection* removes the *Link* from its *Link* registry, and deletes the *Link*. The *Link* destructor calls *Link::detach()*, which asks the attached boxes to remove the *Link* from their registries.

18.15 See answer to Exercise 18.12.

a. Other boxes directly linked to a box:
```
list<const Box*> Box::DirectlyConnectedBoxes();
```
b. Other boxes indirectly linked to a box:
```
set<const Box*> Box::AllConnectedBoxes() const;
```
c. Is a given box associated with a given link?
```
bool Link::linksTo(const Box& box) const;
```
d. Find the box at the other end of a link:
```
Box* Link::OtherBox(const Box& box ) const;
```
e. Find all links between two boxes:
```
list<const Link*> Collection:
   SharedLinks(const Box& b1, const Box& b2);
```

f. Determine which links connect a selected box and a deselected box: (Note: we implement this query in *Collection*, the superclass for both *Sheet* and *Selection*. The subclasses are not implemented here.)

```
list<const Link*> Collection::LinksSelectedAndUnselected()
```

g. Produce an ordered set of points for two boxes and their link.

```
list<Point> Link::SegmentPoints(const Box& start,
   const Box& end);
```