Appendix A

Solutions to the Exercises

This section presents example solutions to the exercises of the previous lectures.

A.1 A Survey of Programming Techniques

1. Discussion of module *Singly-Linked-List-2*.

   (a) Interface definition of module *Integer-List*

   ```
   MODULE Integer-List

   DECLARE TYPE int_list_handle_t;

   int_list_handle_t int_list_create();
   BOOL    int_list_append(int_list_handle_t this,
                           int data);
   INTEGER int_list_getFirst(int_list_handle_t this);
   INTEGER int_list_getNext(int_list_handle_t this);
   BOOL    int_list_isEmpty(int_list_handle_t this);

   END Integer-List;
   ```

   This representation introduces additional problems which are caused by not separating traversal from data structure. As you may recall, to iterate over the elements of the list, we have used a loop statement with the following condition:

   ```
   WHILE data IS VALID DO
   ```

   Data was initialized by a call to *list_getFirst()*. The integer list procedure *int_list_getFirst()* returns an integer, consequently, there is no such thing like an "invalid integer" which we could use for loop termination checking.

2. Differences between object-oriented programming and other techniques. In object-oriented programming *objects* exchange *messages* with each other. In the other programming techniques, *data* is exchanged between procedures under control of a main program. Objects of the same kind but each with its own state can coexist. This contrasts the modular approach where each module only has one global state.

A.2 Abstract Data Types

1. ADT *Integer*.

   (a) Both operations *add* and *sub* can be applied for whatever value is hold by *N*. Thus, these operations can be applied at any time: There is no restriction to their use. However, you can describe this with a precondition which equals *true*.

   (b) We define three new operations as requested: *mul*, *div* and *abs*. The latter should return the absolute value of the integer. The operations are defined as follows:

   ```
   mul(k)
   div(k)
   abs()
   ```

   The operation *mul* does not require any precondition. That's similar to *add* and *sub*. The postcondition is of course $res = N*k$. The next operation *div* requires k to be not 0 (zero). Consequently, we define the following precondition: $k \neq 0$. The last operation *abs* returns the value of N if N is positive or 0 or $-N$ if N is negative. Again it does not matter what value N has when this operation is applied. Here is its postcondition:

   $$abs = \begin{cases} N & : \quad N \geq 0 \\ -N & : \quad N < 0 \end{cases}$$

2. ADT *Fraction*.

   (a) A simple fraction consists of numerator and denominator. Both are integer numbers. This is similar to the complex number example presented in the section. We could choose at least two data structures to hold the values: an *array* or a *record*.

   (b) Interface layout. Remember that the interface is just the set of operations viewable to the outside world. We could describe an interface of a fraction in a verbal manner. Consequently, we need operations:

   - to *get* the value of nominator/denominator,
   - to *set* the value of nominator/denominator,
   - to *add* a fraction returning the sum,
   - to *sub*tract a fraction returning the difference,
   - ...

   (c) Here are some axioms and preconditions for each fraction which also hold for the ADT:

   - The denominator must not equal 0 (zero), otherwise the value of the fraction is not defined.
   - If the nominator is 0 (zero) the value of the fraction is 0 for any value of the denominator.
   - Each whole number can be represented by a fraction of which the nominator is the number and the denominator is 1.

3. ADTs define properties of a set of instances. They provide an abstract view to these properties by providing a set of operations which can be applied on the instances. It is this set of operations, the *interface*, which defines properties of the instances. The use of an ADT is restricted by axioms and preconditions.

Both define conditions and properties of an environment in which instances of the ADT can be used.

4. We need to state axioms and to define preconditions to ensure the correct use of instances of ADTs. For example, if we do not declare 0 to be the neutral element of the addition of integers, there could be an ADT *Integer* which do something weird when adding 0 to N. This is not what is expected from an integer. Thus, axioms and preconditions provide a means to ensure that ADTs "function" as we wish them to.

5. Description of relationships.

   (a) An instance is an actual representative of an ADT. It is thus an "example" of it. Where the ADT declare to use a "signed whole number" as its data structure, an instance actually holds a value, say, "-5".

   (b) Generic ADTs define the same properties of their corresponding ADT. However, they are dedicated to another particular type. For example, the ADT *List* defines properties of lists. Thus, we might have an operation *append(elem)* which appends a new element *elem* to the list. We do not say of what type *elem* actually is, just that it will be the last element of the list after this operation. If we now use a generic ADT *List* the type of this element is known: it's provided by the generic parameter.

   (c) Instances of the same generic ADT could be viewed as "siblings". They would be "cousins" of instances of another generic ADT if both generic ADTs share the same ADT.

A.3 Object-Oriented Concepts

1. Class.

   (a) A *class* is the actual implementation of an ADT. For example, an ADT for integers might include an operation *set* to set the value of its instance. This operation is implemented differently in languages such as C or Pascal. In C the equal sign "=" defines the set operation for integers, whereas in Pascal the character string ":=" is used. Consequently, classes implement operations by providing *methods*. Similarly, the data structure of the ADT is implemented by *attributes* of the class.

   (b) Class *Complex*

```
class Complex {
attributes:
  Real real,
       imaginary

methods:
  :=(Complex c)    /* Set value to the one of c */
  Real realPart()
  Real imaginaryPart()
  Complex +(Complex c)
  Complex -(Complex c)
  Complex /(Complex c)
  Complex *(Complex c)
}
```

We choose the well-known operator symbols "+" for addition, "-" for subtraction, "/" for division and "*" for multiplication to implement the corresponding operations of the ADT *Complex*. Thus, objects of class *Complex* can be used like:

```
Complex c1, c2, c3
c3 := c1 + c2
```

You may notice, that we could write the addition statement as follows:

```
c3 := c1.+(c2)
```

You may want to replace the "+" with "add" to come to a representation which we have used so far. However, you should be able to understand that "+" is nothing more than a different name for "add".

2. Interacting objects.

3. Object view.

4. Messages.

   (a) Objects are autonomous entities which only provide a well-defined interface. We'd like to talk of objects as if they are active entities. For example, objects "are responsible" for themselves, "they" might deny invocation of a method, etc.. This distinguishes an object from a module, which is passive. Therefore, we don't speak of procedure calls. We speak of messages with which we "ask" an object to invoke one of its methods.

   (b) The Internet provides several objects. Two of the most well known ones are "client" and "server". For example, you use an FTP client (object) to access data stored on an FTP server (object). Thus, you could view this as if the client "sends a message" to the server asking for providing data stored there.

   (c) In the client/server environment we really have two remotely acting entities: the client and server process. Typically, these two entities exchange data in form of Internet messages.

A.4 More Object-Oriented Concepts

1. Inheritance.

   (a) Definition of class *Rectangle*:

```
class Rectangle inherits from Point {
attributes:
  int _width,     // Width of rectangle
      _height     // Height of rectangle

methods:
  setWidth(int newWidth)
  getWidth()
  setHeight(int newHeight)
  getHeight()
}
```

In this example, we define a rectangle by its upper left corner (coordinates as inherited from *Point*) and its dimension. Alternatively, we could have defined it by its upper left and lower right corner.

We add access methods for the rectangle's width and height.

(b) 3D objects. A sphere is defined by a center in 3D space and a radius. The center is a point in 3D space, thus, we can define class *Sphere* as:

```
class Sphere inherits from 3D-Point {
attributes:
  int _radius;

methods:
  setRadius(int newRadius)
  getRadius()
}
```

This is similar to the circle class for 2D space. Now, *3D-Point* is just a *Point* with an additional dimension:

```
class 3D-Point inherits from Point {
attributes:
  int _z;

methods:
  setZ(int newZ);
  getZ();
}
```

Consequently, *3D-Point* and *Point* are related with a is-a relationship.

(c) Functionality of *move()*. *move()* as defined in the section allows 3D objects to move on the X-axis, thus only in one dimension. It does this, by modifying only the 2D part of 3D objects. This 2D part is defined by the *Point* class inherited directly or indirectly by 3D objects.

(d) Inheritance graph (see Figure A.1).

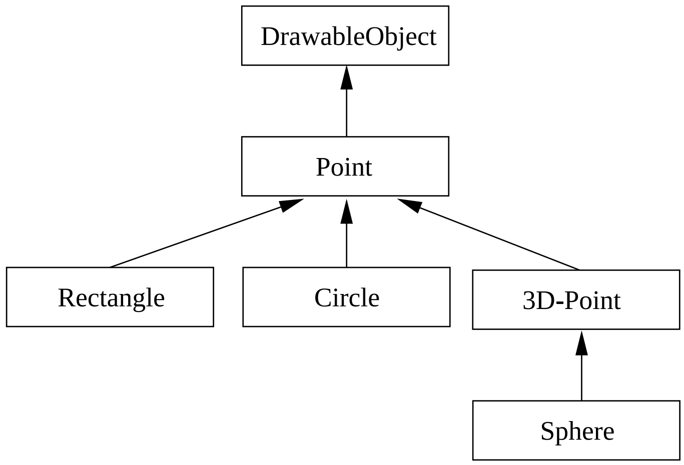

Figure A.1: Inheritance graph of some drawable objects.

(e) Alternative inheritance graph. In this example, class *Sphere* inherits from *Circle* and simply adds a third coordinate. This has the advantage that a sphere can be handled like a circle (for example, its radius can easily be modified by methods/functions which handle circles). It has the disadvantage, that it "distributes" the object's handle (the center point in 3D space) over the inheritance hierarchy: from *Point* over *Circle* to *Sphere*. Thus, this handle is not accessible as a whole.

2. Multiple inheritance. The inheritance graph in Figure 5.9 obviously introduces naming conflicts by properties of class *A*.

   However, these properties are uniquely identified by following the path from *D* up to *A*. Thus, *D* can change properties of *A* inherited by *B* by following the inheritance path through *B*. Similarly, *D* can change properties of *A* inheritied by *C* by following the inheritance path through *C*. Consequently, this naming conflict does not necessarily lead to an error, as long as the paths are designated.

A.5 More on C++

1. Polymorphism. When using the signature

   ```
   void display(const DrawableObject obj);
   ```

   First note, that in C++ function or method parameters are passed by value. Consequently, *obj* would be a *copy* of the actual provided function call argument. This means, that *DrawableObject* must be a class from which objects can be created. This is **not** the case, if *DrawableObject* is an abstract class (as it is when *print()* is defined as pure method.)

   If there exists a virtual method *print()* which is defined by class *DrawableObject*, then (as *obj* is only a copy of the actual argument) this method is invoked. It is **not** the method defined by the class of the actual argument (because it does no longer play any significant role!)

A.6 The List – A Case Study

1. Preincrement operator for iterators. The preincrement operator as defined in the exercise does not check for validity of *_current*. As *succ()* might set its value to *NULL* this may cause access to this NULL-pointer and, hence, might crash the program. A possible solution might be to define the operator as:

   ```
   T &operator ++() {
     succ();
     return(_current ? _current->data() : (T) 0);
   }
   ```

   However, this does not function as we now assume something about *T*. It must be possible to cast it to a kind of „NULL“ value.

2. Addition of remove method. We don't give the code solution. Instead we give the algorithm. The method *remove()* must iterate over the list until it reaches an element with the requested data item. It then deletes this element and returns 1. If the list is empty or if the data item could not be found, it return 0 (zero).

   During the iteration, *remove()* must compare the provided data item successively with those in the list. Consequently, there might exist a comparison like:

   ```
   if (data == current->data()) {
     // found the item
   }
   ```

Here we use the equation operator ,,==“ to compare both data items. As these items can be of any type, they especially can be objects of user defined classes. The question is: How is ,,equality“ defined for those new types? Consequently, to allow *remove()* to work properly, the list should only be used for types which define the comparison operator (namely, ,,==“ and ,,!=“) properly. Otherwise, default comparisons are used, which might lead to strange results.

3. Class *CountedList*. A counted list is a list, which keeps track of the number of elements in it. Thus, when a data item is added, the number is incremented by one, when an item is deleted it is decremented by one. Again, we do not give the complete implementation, we rather show one method (*append()*) and how it is altered:

```
class CountedList : public List {
  int _count;    // The number of elements
  ...
public:
  ...
  virtual void append(const T data) {
    _count++;    // Increment it and ...
    List::append(data); // ... use list append
  }
  ...
}
```

   Not every method can be implemented this way. In some methods, one must check whether *_count* needs to be altered or not. However, the main idea is, that each list method is just expanded (or *specialized*) for the counted list.

4. Iterator problem. To solve the iterator problem one could think of a solution, where the iterator stores a reference to its corresponding list. At iterator creation time, this reference is then initialized to reference the provided list. The iterator methods must then be modified to use this reference instead of the pointer *_start*.