Information Hiding and ADTs

Steven Zeil

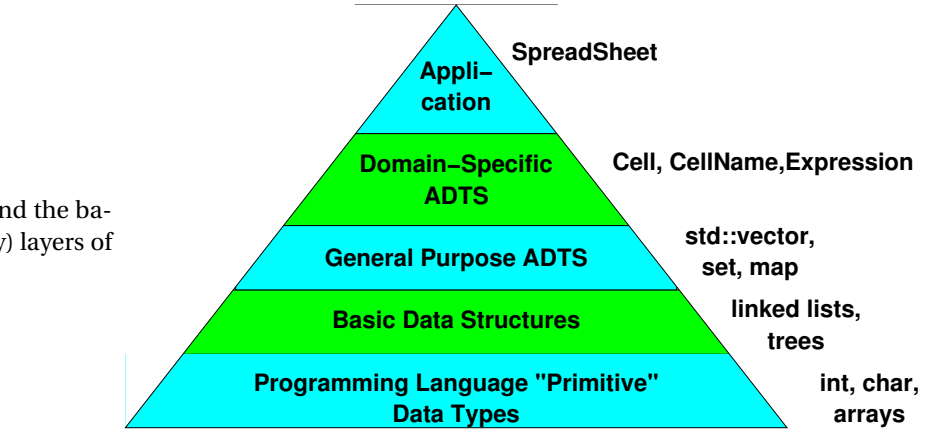
June 17, 2013

Contents

Programs are Layered

Large programs are built in layers.

- Between the main applicaiton and the basic data structures are (hopefully) layers of Abstract Data Types (ADTs).



1 Abstraction

Abstraction

Abstraction is a creative process of focusing attention on the main problems by ignoring lower-level details.

1. *procedural abstraction* and
2. *data abstraction*.

Procedural Abstraction

A *procedural abstraction* is a mental model of *what* we want a subprogram to do (but not *how* to do it).
Example:

```
double hypotenuse
    = sqrt(side1*side1 + side2*side2);
```

- We can use this even if we have no idea how that square root actually gets computed.

. . . . . .

Data Abstraction

A *data abstraction* is a mental model of `what` can be done to a collection of data.

- Deliberately excludes details of `how` to do it.
- We *implement* a data abstraction by
  - choosing an appropriate *data structure*, and
  - providing appropriate *operations* (algorithms) to manipulate that data.

. . . . . .

Example: times

A time denotes a given moment within a day, to the nearest second.

. . . . . .

Example: ordered collection

An *ordered collection* is a sequence of data in which each element is smaller than the ones that follow it

. . . . . .

2 Abstract Data Types

ADTs

An *abstract data type* (ADT) is a type name and a set of members belonging to that type

- The list of members includes
  - the member names,
  - member types
  - expected behavior
- a.k.a. an ADT specification

. . . . . .

2.1 ADT Members

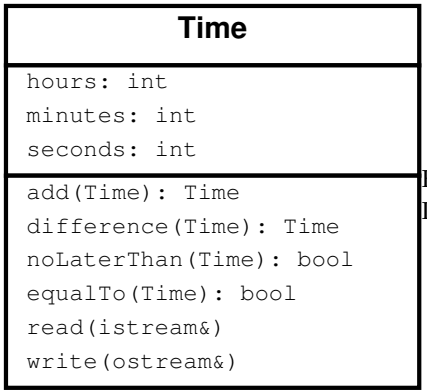
ADT Members

The members of an ADT can be divided into two kinds:

- attributes
  - data contained "within" the ADT
- functions
  - Since we have previously claimed that the ADT must list the types of its members, this may seem a bit odd. What would the type of a function be? But programming language designers and theorists have long considered functions to be typed:
    - The "type" of a function consists of its return type and an ordered list of of its parameters' types

. . . . . .

Example: Time



Here is a possible ADT for our earlier abstraction of “Time”:
It is expressed in three parts:

1. ADT name
2. Attributes
   - Attributes are *not* necessarily data members
     - despite what the textbook says.
     - This is more general - we consider something to be an attribute if we *think* of it as data stored in the ADT (as opposed to something "computed" from the data stored in the ADT.
       If that seems vague, remember that we are seeking to capture a *mental model* of a collection of data. So what we *think* is actually quite important.
     - Later we’ll see that the decision of what is "really" stored and what is "really" computed is a design decision that we are deliberately ignoring (abstracting) for now.
3. operations (function members that do things other than just access attributes)

. . . . . .

Example: Ordered Collection

OrderedCollection
`seq of Elements`

`size(): int`
`add(Element)`
`get(index:int): Element`
`remove(index:int)`
`find(Element): int`

And here is a similar description, this time of an ADT for our abstraction of an ordered collection.

. . . . . .

2.2 The ADT as Contract

The Contract

When an ADT specification/implementation is provided to users (application programmers):

- Users are expected to alter/examine values of this type only via the members specified.
- The creator of the ADT promises to leave the member specifications unchanged.

. . . . . .

Why the Contract?

What do we gain by holding ourselves to this contract?

- Users can be designing and even implementing the application before the details of the ADT implemenation have been worked out.
- The ADT implementors knows exactly what they must provide and what they are allowed to change.
- ADT's designed in this manner are often re-usable.

- We gain the flexibility to try/substitute different data structures to actually implement the ADT, *without needing to alter the application code.*
- By encouraging modularity, application code becomes more readable.

. . . . . .

3 ADT Implementations

ADT Implementations

We go from ADT to *ADT implementation* by adding specific data structures and algorithms.
In C++, this is generally done using a C++ `class` or `struct`.

. . . . . .

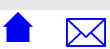
Example: Time implementation

As ADT designer, I might consider two possible data structures:

- store the hours, minutes, & seconds as integers

  time.h

```
#ifndef TIMES_H
#define TIMES_H

#include <iostream>

/**
 * Times in this program are represented by three integers: H, M, & S, representing
 * the hours, minutes, and seconds, respecitvely.
 */
```

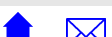
```
struct Time {
  // Create time objects
  Time(); // midnight
  Time (int h, int m, int s);

  // Access to attributes
  int getHours();
  int getMinutes();
  int getSeconds();

  // Calculations with time
  void add (Time delta);
  Time difference (Time fromTime);

  /**
   * Read a time (in the format HH:MM:SS) after skipping any
   * prior whitepace.
   */
  void read (std::istream& in);


  /**
   * Print a time in the format HH:MM:SS (two digits each)
   */
  void print (std::ostream& out);

  /**
   *  Compare two times. Return true iff time1 is earlier than or equal to
   *  time2
   */
  bool noLaterThan(const Time& time2);
```


```
  /**
   *  Compare two times. Return true iff time1 is equal to
   *  time2
   *
   */
  bool equalTo(const Time& time2);

  // From here on is hidden
  int hours;
  int minutes;
  int seconds;

};

#endif // TIMES_H
```

time.cpp

```
#include "time1.h"

/**
 * Times in this program are represented by three integers: H, M, & S, representing
 * the hours, minutes, and seconds, respecitvely.
 */



  // Create time objects
Time::Time() // midnight
```


```
{
  hours = minutes = seconds = 0;
}


Time::Time (int h, int m, int s)
{
  hours = h;
  minutes = m;
  seconds = s;
}

  // Access to attributes
int Time::getHours()
{
  return hours;
}

int Time::getMinutes()
{
  return minutes;
}

int Time::getSeconds()
{
  return seconds;
}

// Calculations with time
void Time::add (Time delta)
{
```

```
  hours += delta.hours;
  minutes += delta.minutes;
  seconds += delta.seconds;

  minutes += seconds / 60;
  seconds = seconds % 60;

  hours += minutes / 60;
  minutes = minutes % 60;
}


Time Time::difference (Time fromTime)
{
  Time diff (hours - fromTime.hours,
         minutes - fromTime.minutes,
         seconds - fromTime.seconds);
  while (diff.seconds < 0)
    {
      diff.seconds += 60;
      --diff.minutes;
    }
  while (diff.minutes < 0)
    {
      diff.minutes += 60;
      --diff.hours;
    }
}


/**
```


```
 * Read a time (in the format HH:MM:SS) after skipping any
 * prior whitepace.
 */
void Time::read (std::istream& in)
{
  char c;
  in >> hours >> c >> minutes >> c >> seconds;
}


/**
 * Print a time in the format HH:MM:SS (two digits each)
 */
void Time::print (std::ostream& out)
{
  out << hours << ':' << minutes << ':' << seconds;
}

/**
 *  Compare two times. Return true iff time1 is earlier than or equal to
 *  time2
 */
bool Time::noLaterThan(const Time& time2)
{
  // First check the hours
  if (hours > time2.hours)
    return false;
  if (hours < time2.hours)
    return true;
  // If hours are the same, compare the minutes
  if (minutes > time2.minutes)
```


```
    return false;
  if (minutes < time2.minutes)
    return true;
  // If hours and minutes are the same, compare the seconds
  if (seconds > time2.seconds)
    return false;
  return true;
}



/**
 *  Compare two times. Return true iff time1 is equal to
 *  time2
 *
 */
bool Time::equalTo(const Time& time2)
{
  return hours == time2.hours
    && minutes == time2.minutes
    && seconds == time2.seconds;
}
```

- store a single integer, indicating # of seconds since midnight

  time.h

```
#ifndef TIMES_H
#define TIMES_H

#include <iostream>
```

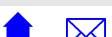

```
/**
 * Times in this program are represented by three integers: H, M, & S, representing
 * the hours, minutes, and seconds, respecitvely.
 */


struct Time {
  // Create time objects
  Time(); // midnight
  Time (int h, int m, int s);

  // Access to attributes
  int getHours();
  int getMinutes();
  int getSeconds();

  // Calculations with time
  void add (Time delta);
  Time difference (Time fromTime);

  /**
   * Read a time (in the format HH:MM:SS) after skipping any
   * prior whitepace.
   */
  void read (std::istream& in);


  /**
   * Print a time in the format HH:MM:SS (two digits each)
   */
  void print (std::ostream& out);
```


```
  /**
   *  Compare two times. Return true iff time1 is earlier than or equal to
   *  time2
   */
  bool noLaterThan(const Time& time2);


  /**
   *  Compare two times. Return true iff time1 is equal to
   *  time2
   *
   */
  bool equalTo(const Time& time2);

  // From here on is hidden
  int secondsSinceMidnight;
};

#endif // TIMES_H
```

time.cpp

```
#include "time2.h"

/**
 * Times in this program are represented by three integers: H, M, & S, representing
 * the hours, minutes, and seconds, respecitvely.
 */
```


```
  // Create time objects
Time::Time() // midnight
{
  secondsSinceMidnight = 0;
}


Time::Time (int h, int m, int s)
{
  secondsSinceMidnight = s + 60 * m + 3600*h;
}

  // Access to attributes
int Time::getHours()
{
  return secondsSinceMidnight / 3600;
}

int Time::getMinutes()
{
  return (secondsSinceMidnight % 3600) / 60;
}

int Time::getSeconds()
{
  return secondsSinceMidnight % 60;
}

// Calculations with time
void Time::add (Time delta)
```

```
{
  secondsSinceMidnight += delta.secondsSinceMidnight;
}


Time Time::difference (Time fromTime)
{
  Time diff;
  diff.secondsSinceMidnight =
    secondsSinceMidnight - fromTime.secondsSinceMidnight;
}


/**
 * Read a time (in the format HH:MM:SS) after skipping any
 * prior whitepace.
 */
void Time::read (std::istream& in)
{
  char c;
  int hours, minutes, seconds;
  in >> hours >> c >> minutes >> c >> seconds;
  Time t (hours, minutes, seconds);
  secondsSinceMidnight = t.secondsSinceMidnight;
}


/**
 * Print a time in the format HH:MM:SS (two digits each)
 */
void Time::print (std::ostream& out)
```


```
{
  out << getHours() << ':' << getMinutes() << ':' << getSeconds();
}

/**
 *  Compare two times. Return true iff time1 is earlier than or equal to
 *  time2
 */
bool Time::noLaterThan(const Time& time2)
{
  return secondsSinceMidnight <= time2.secondsSinceMidnight;
}



/**
 *  Compare two times. Return true iff time1 is equal to
 *  time2
 *
 */
bool Time::equalTo(const Time& time2)
{
  return secondsSinceMidnight == time2.secondsSinceMidnight;
}
```

Each approach has pro's and cons. The first will be faster if we are mainly reading and writing times. The second will be faster if we are doing lots of calculations involving times.

.....

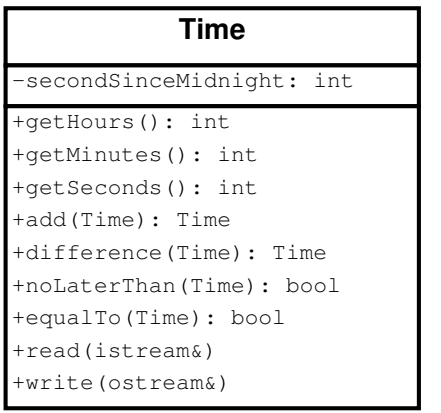
Switching Implementations

- Ideally, we should be able to switch between these two choices without breaking any application code.

For example, if we discover that our application program is doing an unexpectedly large number of time calculations, we might want to switch to the second implementation even if we had started with the first.

. . . . . .

Example of an ADT Implementation



(This really does show data members in the middle- the +/- markers are the giveaway.)
We get that flexibility by concentrating on operations that we want to keep public, and hiding the data structures used to provide those operations.
We'll explore this aspect of ADT design in more detail later.

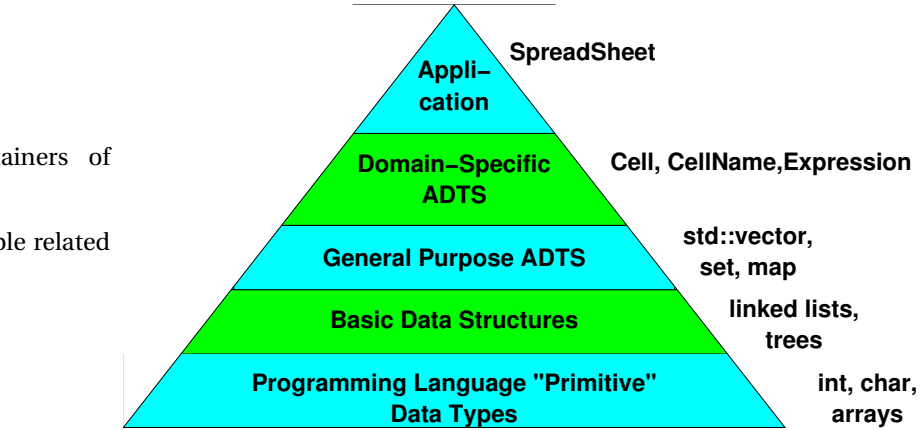
. . . . . .

4 Where Do ADTs Come From?

Where Do ADTs Come From?

ADT's may be

- General-Purpose: often containers of "smaller" ADTs
- Domain-specific, used in multiple related applications
- Application-specific



Real-World Objects make Good ADTs

Domain and application-specific ADTs generally reflect the real-world objects found in the application domain

- Example: Item, Bid, & Bidder in auction application
- The process of discovering good domain- and applicaiton-specific ADT's is covered in CS 330

The OO Philosophy

This is sometimes expressed as *The Object-Oriented Philosophy*

- Every program is really a simulation.
- The quality of a program's design is proportional to the faithfulness with which the structures and interactions in the program mirror those in the real world.

Example: Library Holdings

A question to think about:

What ADT's would you expect in a system to manage a library's inventory?

If you answered “Books”, “Shelves”, “Librarians”, etc., you have the right OO attitude.

If you answered DataBases, Inventory Files, etc., you are too tied up in the artificial world of programming. That’s a very 1970’s-style answer.

If you answered “check in”, “check out”, “process customer”, etc., you are thinking of functions instead of types of data.

. . . . . .