Abstract Data Types

Abstract Data Types (ADTs)

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Conceptualisation

- Abstract Data Types (ADTs) are based on
  - Data abstraction
    - Programming techniques that relies on the separation of interface and implementation
      - The **interface** specifies the operations that can be executed
      - The **implementation** defines how these operations are realized
  - Encapsulation
    - Enforces the separation between interface and implementation
      - Users can only access the interface
      - Only developers can access the implementation

Conceptualisation

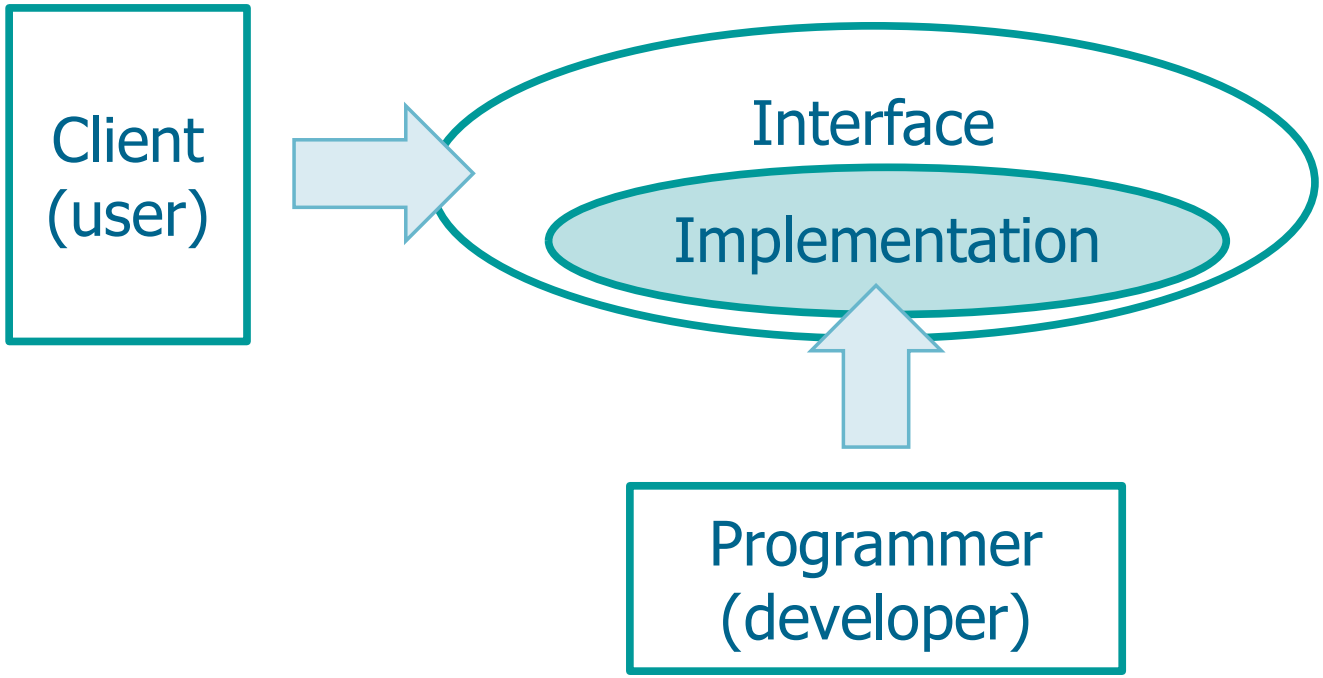
- The C language does not have proper support to data abstraction and encapsulation
- The target of this section is to learn how to
  - Increase the abstraction level of our applications
  - Separate the interface and the implementation of our applications
  - Hiding, as long as possible, implementation details

ADT Version 1

- An Abstract Data Type (ADT) is
  - A set of objects
  - A collection of operations on those objects

ADT Version 1

- We refer to the code that
  - Uses an ADT, as a **client**
  - Specifies the operation, as the **interface**
  - Defines how the operations are realized, as an **implementation**



Example

- Let us consider an application manipulating points in the Cartesian 3D space
  - The "developer" of the library
    - Defines the operations that the library should support and make available
    - Specifies how these operations are implemented in C code
  - The "user" of the library
    - Write an application using the functionality made available by the library
    - All available functionalities may be gathered looking at the interface; the implementation must **not** be analyzed

Solution 0

point.h

```
#ifndef _POINT
#define _POINT
#include <stdio.h>
#include <math.h>
typedef struct point_s {
  float x, y, z;
} point_t;
float dist (point_t, point_t);
#endif
```

The interface of the **point** library

The implementation of the **point** library

point.c

```
#include "point.h"
float dist (point_t p1, point_t p2) {
  float d;
  d = 0;
  d = d + (p1.x - p2.x) * (p1.x - p2.x);
  d = d + (p1.y - p2.y) * (p1.y - p2.y);
  d = d + (p1.z - p2.z) * (p1.z - p2.z);
  d = sqrt (d);
  return d;
}
```

Solution 0

The client
A program using the
point library

client.c

```
#include "point.h"
int main (void) {
  point_t p1, p2;
  float d;
  p1.x = p1.y = p1.z = 0.0;
  p2.x = p2.y = p2.z = 10.0;
  d = dist (p1, p2);
  fprintf (stdout, "D = %f\n", d);
  return 1;
}
```

Considerations

- Following this scheme (or more refined ones)
  - When you write a new application
    - You should separate the interface and the implementation, hiding, as long as possible, implementation details
    - The application is thus made of a client (using a library) and the library (delivering the required functionality)
- For example
  - You can write a **sorting** or a **list** library such that they can be used in different applications without rewriting them from scratch every time you need their functionality

Considerations

- In the previous example the C type is defined in the interface
  - As the client includes the interface, the C structure is fully visible from the client as the structure definition is visible
  - This ADT reaches only **partial** data hiding, even if function prototypes are defined and included correctly

ex,
to increse the level use a function that sets the coords to 000

Data hiding is very partial

client.c

```
#include "point.h"
int main (void) {
  point_t p1, p2;
  float d;
  p1.x = p1.y = p1.z = 0.0;
  p2.x = p2.y = p2.z = 10.0;
  d = dist (p1, p2);
  fprintf (stdout, "D = %f\n", d);
  return 1;
}
```

ADT Version 2

- For an ADT we should potentially have many different instances
  - We should be able to assign the ADT to variables to hold an instance of the ADT
  - The client programs should manipulate the ADT without direct access, but rather with indirect access, through operations defined in the ADT
  - Each operation should possibly have different implementations within the implementation program, even if the interface defines it in the same way

Solution 1

Sedgewich style

- The implementation
  - Includes the data definition which is not visible from the client
- The interface
  - Provides all functions to get, set (getters and setters), destroy, etc., all data type fields
- The client
  - Includes the interface and it cannot access the ADT directly
  - It manages the ADT only through pointers (handles or wrappers), and it needs dynamic memory allocation

Solution 1

The client
A program using the library **point**

client.c

```
#include "point.h"
int main (void) {
  point_t p1, p2;
  float d;
  p1 = new ();
  p2 = new ();
  set (p1, 0.0, 0.0, 0.0);
  set (p2, 10.0, 10.0, 10.0);
  d = dist (p1, p2);
  fprintf (stdout, "D = %f\n", d);
  disp (p1);
  disp (p2);
  return 1;
}
```

Solution 1

The interface
A header file specifying the functionalities of the library **point**

point.h

```
#ifndef _POINT
#define _POINT

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct point_s *point_t;
point_t new ();
void set (point_t, float, float,  float);
float dist (point_t, point_t);
void disp (point_t);

#endif
```

Solution 1

The implementation of the library **point**

point.c

```
#include "point.h"
struct point_s {
  float x, y, z;
};
point_t new () {
  point_t p;
  p = malloc (1 * sizeof (struct point_s));
  if (p == NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (1);
   }
  return p;
}
void set (point_t p, float x, float y, float z) {
  p->x = x; p->y = y; p->z = z;
}
```

Solution 1

The implementation of the library **point**

```
float dist (point_t p1, point_t p2) {
  float d;
  d = 0;
  d = d + (p1->x - p2->x) * (p1->x - p2->x);
  d = d + (p1->y - p2->y) * (p1->y - p2->y);
  d = d + (p1->z - p2->z) * (p1->z - p2->z);
  d = sqrt (d);
  return d;
}
void disp (point_t p) {
  free (p);
}
```

Solution 2

- A first evolution of the previous scheme
  - There is a public and a private interface
  - The client only includes the public interface
    - Through it, the client will have access to public functions and type pointers as well
  - The implementation does include the private interface and through it the public interface too
    - Through it, the implementation sees everything

Solution 2

The client
A program using the library **point**

client.c

```
#include "pointPublic.h"
int main (void) {
  point_t p1, p2;
  float d;
  p1 = new ();
  p2 = new ();
  set (p1, 0.0, 0.0, 0.0);
  set (p2, 10.0, 10.0, 10.0);
  d = dist (p1, p2);
  fprintf (stdout, "D = %f\n", d);
  disp (p1);
  disp (p2);
  return 1;
}
```

Solution 2

pointPublic.h

```
#ifndef _POINT_PUBLIC
#define _POINT_PUBLIC
#include <stdio.h>
typedef struct point_s *point_t;
Point_t new ();
void set (point_t, float, float,  float);
float dist (point_t, point_t);
void disp ();
#endif
```

pointPrivate.h

```
#ifndef _POINT_PRIVATE
#define _POINT_PRIVATE
#include <stdlib.h>
#include <math.h>
#include "pointPublic.h"
struct point_s {
  float x, y, z;
};
#endif
```

Solution 2

The implementation of the library **point**

point.c

```
#include "pointPrivate.h"


point_t new () {
   ... as previous version ...
}
void set (point_t p, float x, float y, float z) {
   ... as previous version ...
}
float dist (point_t p1, point_t p2) {
   ... as previous version ...
}
void disp (point_t p) {
   ... as previous version ...
}
```

Solution 3

- The previous solution hide the pointer definition
  - Pointers are hidden as **point_t** is actually a **struct point_s ***
  - Many programming styles (Java-based, C in Windows API, etc.) hide pointer definitions to avoid using the * operator explicitly
- The following solution is more suited for a native C programming style
  - Pointers are made explicit
  - It all boils down to the programmer's preferences

Solution 3

The client
A program using the **point** library

client.c

```
#include "pointPublic.h"
int main (void) {
  point_t *p1, *p2;
  float d;
  p1 = new ();
  p2 = new ();
  set (p1, 0.0, 0.0, 0.0);
  set (p2, 10.0, 10.0, 10.0);
  d = dist (p1, p2);
  fprintf (stdout, "D = %f\n", d);
  disp (p1);
  disp (p2);
  return 1;
}
```

Solution 3

pointPublic.h

```
#ifndef _POINT_PUBLIC
#define _POINT_PUBLIC
#include <stdio.h>
typedef struct point_s point_t;
point_t *new ();
void set (point_t *, float, float,  float);
float dist (point_t *, point_t *);
void disp (point_t *);
#endif
```

pointPrivate.h

```
#ifndef _POINT_PRIVATE
#define _POINT_PRIVATE
#include <stdlib.h>
#include <math.h>
#include "pointPublic.h"
struct point_s {
  float x, y, z;
};
#endif
```

Solution 3

The implementation of the library **point**

point.c

```
#include "pointPrivate.h"


point_t *new () {
  point_t *p;
  ... as previous version ...
}
void set (point_t *p, float x, float y, float z) {
 ... as previous version ...
}
float dist (point_t *p1, point_t *p2) {
 ... as previous version ...
}
void disp (point_t *p) {
 ... as previous version ...
```