Programming Languages

Modules

G22.2110

Summer 2010

Modules

Programs are built out of components.

Each component:

- has a public interface that defines entities exported by the component
- may depend on the entities defined in the interface of another component (weak external coupling)
- may include other (private) entities that are not exported
- should define a set of logically related entities (strong internal coupling)

We call these components modules.

What is a module?

- different languages use different terms
- different languages have different semantics for this construct (sometimes very different)
- a module is somewhat like a record, but with an important distinction:
  - **record** $\Longrightarrow$ consists of a set of names called *fields*, which refer to values in the record
  - **module** $\Longrightarrow$ consists of a set of names, which can refer to values, types, routines, other language-specific entities, and possibly other modules

Note that the similarity is between a *record* and a *module*, not a *record type* and a *module*.

Language constructs for modularity

Issues:

- public interface
- private implementation
- dependencies between modules
- naming conventions of imported entities
- relationship between modules and files

Language choices

- **Ada** : package declaration and body, `with` and `use` clauses, renamings
- **C** : header files, `#include` directives
- **C++** : header files, `#include` directives, namespaces, `using` declarations/directives, namespace alias definitions
- **Java** : packages, `import` statements
- **ML** : `signature`, `structure` and `functor` definitions

Ada: Packages

```
package Queues is
  Size: constant Integer := 1000;

  type Queue is private; -- information hiding

  procedure Enqueue (Q: in out Queue, Elem: Integer);
  procedure Dequeue (Q: in out Queue; Elem: out Integer);
  function Empty (Q: Queue) return Boolean;
  function Full (Q: Queue) return Boolean;
  function Slack (Q: Queue) return Integer;
  -- overloaded operator "=":
  function "=" (Q1, Q2: Queue) return Boolean;

private
  ... -- concern of implementation, not of package client
end Queues;
```

Private parts and information hiding

```
package Queues is
  ... -- visible declarations
private
  type Storage is
    array (Integer range <>) of Integer;
  type Queue is record
    Front: Integer := 0; -- next elem to remove
    Back: Integer := 0;  -- next available slot
    Contents: Storage (0 .. Size-1); -- actual contents
    Num: Integer := 0;
  end record;
end Queues;
```

Implementation of Queues

```
package body Queues is
  procedure Enqueue (Q: in out Queue;
                     Elem: Integer) is
  begin
    if Full(Q) then
      -- need to signal error: raise exception
    else
      Q.Contents(Q.Back) := Elem;
    end if;
    Q.Num := Q.Num + 1;
    Q.Back := (Q.Back + 1) mod Size;
  end Enqueue;
```

Predicates on queues

```
function Empty (Q: Queue) return Boolean is
begin
  return Q.Num = 0;    -- client cannot access
                       --   Num directly
end Empty;

function Full (Q: Queue) return Boolean is
begin
  return Q.Num = Size;
end Full;

function Slack (Q: Queue) return Integer is
begin
  return Size - Q.Num;
end Slack;
```

Operator Overloading

```
function "=" (Q1, Q2 : Queue) return Boolean is
begin
  if Q1.Num /= Q2.Num then
    return False;
  else
    for J in 1 .. Q1.Num loop
      -- check corresponding elements
      if Q1.Contents((Q1.Front + J - 1) mod Size) /=
         Q2.Contents((Q2.Front + J - 1) mod Size)
      then
        return False;
      end if;
    end loop;
    return True; -- all elements are equal
  end if;
end "=";     -- operator "/=" implicitly defined
             --   as negation of "="
```

Client can only use visible interface

```
with Queues;  use Queues;  with Text_IO;

procedure Test is
  Q1, Q2: Queue; -- local objects of a private type
  Val : Integer;
begin
  Enqueue(Q1, 200); -- visible operation
  for J in 1 .. 25 loop
    Enqueue(Q1, J);
    Enqueue(Q2, J);
  end loop;
  Deqeue(Q1, Val); -- visible operation
  if Q1 /= Q2 then
    Text_IO.Put_Line("lousy␣implementation");
  end if;
end Test;
```

Implementation

- package body holds bodies of subprograms that implement interface
- package may not require a body:

```
package Days is
  type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

  subtype Weekday is Day range Mon .. Fri;

  Tomorrow: constant array (Day) of Day
     := (Tue, Wed, Thu, Fri, Sat, Sun, Mon);

  Next_Work_Day: constant array (Weekday) of Weekday
     := (Tue, Wed, Thu, Fri, Mon);
end Days;
```

Syntactic sugar: `use` and `renames`

Visible entities can be denoted with an expanded name:

```
with Text_IO;
...
Text_IO.Put_Line("hello");
```

`use` clause makes name of entity directly usable:

```
with Text_IO;  use Text_IO;
...
Put_Line("hello");
```

`renames` clause makes name of entity more manageable:

```
with Text_IO;
package T renames Text_IO;
...
T.Put_Line("hello");
```

Sugar can be indispensable

```
with Queues;

procedure Test is
  Q1, Q2: Queues.Queue;
begin
  if Q1 = Q2 then ...
    -- error: "=" is not directly visible
    -- must write instead: Queues."="(Q1, Q2)
```

Two solutions:

- import all entities:

```
use Queues;
```

- import operators only:

```
use type Queues.Queue;
```

C++ namespaces

- late addition to the language
- an entity requires one or more declarations and a single definition
- a namespace declaration can contain both, but definitions may also be given separately

```
// in .h file
namespace util {
  int f (int); /* declaration of f */
}

// in .cpp file
namespace util {
  int f (int i) {
    // definition provides body of function
    ...
  }
}
```

Dependencies between modules in C++

- files have semantic significance: `#include` directives means textual substitution of one file in another
- convention is to use header files for shared interfaces

```
#include <iostream> // import declarations

int main () {
  std::cout << "C++␣is␣really␣different"
            << std::endl;
  return 0;
}
```

Header files are visible interfaces

```
namespace stack {  // in file stack.h
  void push (char);
  char pop ();
}
```

```
#include "stack.h"  // import into client file

void f () {
  stack::push('c');
  if (stack::pop() != 'c') error("impossible");
}
```

Namespace Definitions

```
#include "stack.h" // import declarations

namespace stack {  // the definition
  const unsigned int MaxSize = 200;
  char v[MaxSize];
  unsigned int numElems = 0;

  void push (char c) {
    if (numElems >= MaxSize)
      throw std::out_of_range("stack␣overflow");
    v[numElems++] = c;
  }

  char pop () {
    if (numElems == 0)
      throw std::out_of_range("stack␣underflow");
    return v[--numElems];
  }
}
```

Syntactic sugar: `using` declarations

```
namespace queue { // works on single queue
  void enqueue (int);
  int dequeue ();
}
```

```
#include "queue.h"  // in client file

using queue::dequeue;  // selective: a single entity

void f () {
  queue::enqueue(10);  // prefix needed for enqueue
  queue::enqueue(-999);
  if (dequeue() != 10)  // but not for dequeue
    error("buggy␣implementation");
}
```

Wholesale import: the using directive

```
#include "queue.h"  // in client file

using namespace queue;  // import everything

void f () {
  enqueue(10);  // prefix not needed
  enqueue(-999);
  if (dequeue() != 10)  // for anything
    error("buggy␣implementation");
}
```

Shortening names

Sometimes, we want to qualify names, but with a shorter name.

In Ada:

```
package PN renames A.Very_Long.Package_Name;
```

In C++:

```
namespace pn = a::very_long::package_name;
```

We can now use `PN` as the qualifier instead of the long name.

Visibility: Koenig lookup

When an unqualified name is used as the postfix-expression in a function call (**expr.call**), other namespaces not considered during the usual unqualified look up (**basic.lookup.unqual**) may be searched; this search depends on the types of the arguments.

For each argument type T in the function call, there is a set of zero or more associated namespaces to be considered. The set of namespaces is determined entirely by the types of the function arguments. `typedef` names used to specify the types do not contribute to this set.

The set of namespaces are determined in the following way:

Koenig lookup: details

- If T is a fundamental type, its associated set of namespaces is empty.
- If T is a class type, its associated namespaces are the namespaces in which the class and its direct and indirect base classes are defined.
- If T is a union or enumeration type, its associated namespace is the namespace in which it is defined.
- If T is a pointer to U, a reference to U, or an array of U, its associated namespaces are the namespaces associated with U.
- If T is a pointer to function type, its associated namespaces are the namespaces associated with the function parameter types and the namespaces associated with the return type. [recursive]

Linking

- an external declaration for a variable indicates that the entity is defined elsewhere

```
extern int x; // will be found later
```

- a function declaration indicates that the body is defined elsewhere
- multiple declarations may denote the same entity

```
extern int x; // in some other file
```

- an entity can only be *defined* once
- missing/multiple definitions cannot be detected by the compiler: link-time errors

Include directives = multiple declarations

```
#include "queue.h" // as if declaration were
                   //   textually present
void f () { ... }
```

```
#include "queue.h" // second declaration in
                   //   different client
void g () { ... }
```

- definitions are legal if textually identical (but compiler can't check!)
- headers are safer than cut-and-paste, but not as good as a proper module system

Modules in Java

- package structure parallels file system
- a package corresponds to a directory
- a class is compiled into a separate object file
- each class declares the package in which it appears (open structure)

```
package polynomials;
class poly {
  ... // in file .../alg/polynomials/poly.java
}
```

```
package polynomials;
class iterator {
  ... // in file .../alg/polynomials/iterator.java
}
```

Default: anonymous package in current directory.

Dependencies between classes

- dependencies indicated with `import` statements:

```
import java.awt.Rectangle; // declared in java.awt

import java.awt.*;     // import all classes
                       //   in package
```

- no syntactic sugar across packages: use expanded names
- none needed in same package: all classes in package are directly visible to each other

Modules in ML

There are three entities:

- `signature` : an interface
- `structure` : an implementation
- `functor` : a parameterized `structure`

A `structure` implements a `signature` if it defines everything mentioned in the `signature` (in the correct way).

ML signature

An ML *signature* specifies an interface for a module.

```
signature STACKS =
sig
    type stack
    exception Underflow
    val empty : stack
    val push : char * stack -> stack
    val pop  : stack -> char * stack
    val isEmpty : stack -> bool
end
```

ML structure

A *structure* provides an implementation.

```
structure Stacks : STACKS =
struct
    type stack = char list
    exception Underflow
    val empty = [ ]
    val push = op::
    fun pop (c::cs) = (c, cs)
      | pop []      = raise Underflow
    fun isEmpty [] = true
      | isEmpty _  = false
end
```

Comparisons

	Ada	C++	Java	ML
used to avoid name clashes	✔	✔	✔	✔
access control	✔	weak	✔	✔
is closed	✔	✘	✘	✔

Relation between interface and implementation:

- Ada :

  one package (interface) ⇔ one package body

- ML :

  one signature *can be implemented by* many structures
  one structure *can implement* many signatures