Programming Languages

Modules

G22.2110
Summer 2008
Programs are built out of components.

Each component:

- defines a set of logically related entities (strong internal coupling)
- 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

We call these components modules.
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** $\Rightarrow$ consists of a set of names called *fields*, which refer to values in the record
  - **module** $\Rightarrow$ 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
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;
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;
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;
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 "+="
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;
  Dequeue(Q1, Val); -- visible operation
  if Q1 /= Q2 then
    Text_IO.Put_Line("lousy implementation");
  end if;
end Test;
- package body holds bodies of subprograms that implement interface
- package may not require a body:

```pascal
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;
```
Visible entities can be denoted with an expanded name:

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

**use** clause makes name of entity directly usable:

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

**renames** clause makes name of entity more manageable:

```plaintext
with Text_IO;
package T renames Text_IO;
...
T.Put_Line("hello");
```
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;
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
        ...
    }
}
files have semantic significance: `#include` directives mean textual substitution of one file in another

- convention is to use header files for shared interfaces

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

int main () {
    std::cout << "C++ is really different"
               << std::endl;
}
```
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");
}
```cpp
#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\n overflow");
        v[numElems++] = c;
    }

    char pop () {
        if (numElems == 0)
            throw std::out_of_range("stack\n underflow");
        return v[--numElems];
    }
}
```
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");
}
#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");
}
Sometimes, we want to qualify names, but with a shorter name.

In Ada:

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

In C++:

```cpp
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. Type-def names used to specify the types do not contribute to this set.

The set of namespaces are determined in the following way:
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]
- an external declaration for a variable indicates that the entity is defined elsewhere

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

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

```c
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

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

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

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

Default: anonymous package in current directory.
dependencies indicated with `import` statements:

```java
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
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).
An ML *signature* specifies an interface for a module.

```ml
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
```
A *structure* provides an implementation.

```ml
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
```
<table>
<thead>
<tr>
<th></th>
<th>Ada</th>
<th>C++</th>
<th>Java</th>
<th>ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>used to avoid name clashes</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>access control</td>
<td>✔</td>
<td>weak</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>is closed</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
</tr>
</tbody>
</table>

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