Sviluppo di Software Sicuro - S^3 Informed design

Corso di Laurea Magistrale in
Sicurezza Informatica: Infrastrutture e Applicazioni
Università di Pisa – Polo di La Spezia
C. Montangero
Anno accademico 2009/10

Obiettivi di qualità di un progetto

- EALCH: encapsulation, abstraction, loose coupling, cohesion, hierarchy
- Encapsulation
  - the clear separation of specification from implementation
  - "contract model" of programming

Obiettivi di qualità di un progetto (2)

- Abstraction
  - a necessary component of encapsulation
  - it allows stripping away (ignore) certain levels of detail which is encapsulated in the object
  - "information hiding" is frequently used in this context
    - This term is somewhat inappropriate:
    - information, by definition, *informs. We cannot reason in the absence of information;*
    - *however, we can ensure that our* reasoning takes place at an appropriate level of abstraction
    - *Hiding* unnecessary detail to focus on the *essential properties of an object.*
  - *S*PARK annotations are essential in this respect
    - particularly the own variable clause, which reveals the presence of "state" in a package without revealing unnecessary details

Obiettivi di qualità di un progetto (3)

- Loose coupling
  - a measure of the connections between objects
    - Highly coupled objects interact in ways that make their separate modification difficult
    - high levels of coupling arise when abstractions are poor and encapsulation inadequate
  - SPARK provides a simple and clear distinction:
    - the appearance of a package name only as a prefix in an *inherits* annotation represents **weak** coupling (use of a service)
    - its appearance in a *global* or *derives* annotation indicates **strong** coupling (sharing of data)
  - Excessive coupling is revealed by the size and complexity SPARK's *derives* and *global* annotations
  - The goal of optimising information flow is equivalent to achieving loose coupling

Obiettivi di qualità di un progetto (4)

- Cohesion
  - Whereas coupling is measured *between* objects, *cohesion* is a property of an object.
  - it is measure of focus or singleness of purpose:
    - For example, a car has both door handles and pistons but we would not expect to find both represented by a single software object; If they were, we would not have high cohesion:
    - modifying the software to support a 2-door rather than 4-door model (or to replace a straight 4 with a V8 engine) would involve changing rather unexpected parts of the design

Obiettivi di qualità di un progetto (5)

- Hierarchy
  - Here we recognise that in the real-world objects have exhibit hierarchy.
  - Certain objects are contained inside others and cannot be reached directly.
    - When we approach a car we can grasp a door handle but not a piston: the latter is inside the engine object which is itself inside the car.
  - a clear object hierarchy
    - encouraged and checked by the rules of SPARK
    - contributes to the goal of keeping the flow of information under control — loose coupling

Elementi base di progetto

- INFORMED designs use
  - three key building blocks
    - main program
      - the top-level, entry point controlling the behaviour of a system
    - variable packages
      - they contain "state" as revealed by an *own* variable annotation
    - type packages
      - they provide template from which objects can later be formed
  - plus two auxiliary ones
    - utility packages
      - they provide shared services to the main components
    - boundary variables
      - they provide interfaces for all the communication across the system boundary

Main program

```
with A, B, C;
--# inherit A, B, C, D;
--# main_program;
procedure Main
--# global in out A.State, B.State, ...
--# derives ...
is
procedure Initialize;
....
begin -- Main
 Initialize;
 loop
  ControlProcedure;
 end loop;
end Main;
```

- The control procedure is likely to be decomposed into several smaller procedures
- Aim: make each such decomposed procedure responsible for a single "mode" of the system's behaviour
- For example, in a cycle computer there are separate modes for programming the unit
  - with the size of the bicycle's wheel
  - for normal operation
- cohesion and clear and useful annotations

Variable Packages

```
package Stack
--# own State;
is
procedure Clear;
--# global out State;
--# derives State from ;
procedure Push(X : in Integer);
--# global in out State;
--# derives State from State, X;
procedure Pop(X : out Integer);
--# global in out State;
--# derives X, State from State;
end Stack;
```

- names a container which can contain values of a certain shape upon which certain operations are possible
- like a variable
- cannot be passed as argument

Variable Packages (2)

- SPARK's own variable annotation gives a name to the encapsulated state
- to be use in annotations to clarify the intended use
- abstraction can be maintained because no details of the internal structure of the state need be revealed
- Hierarchy is facilitated because SPARK refinement rules allow the abstract state, to represent a number of more detailed state items which are conceptually inside the object
- E.g., a car object could contain an engine object which might contain a number of piston objects
- At the most concrete level will be a number of Ada primitive objects
- SPARK 95 private child packages allow the logical nesting or embedding of variable packages without the need to embed physically the packages that represent them

Type Packages

```
package Stack
is
 type T is private;
 procedure Clear(S : out T);
 --# derives S from ;
 procedure Push(S : in out T;
                X : in Integer);
 --# derives S from S, X;
 procedure Pop(S : in out T;
               X : out Integer);
 --# derives X, S from S;
 private
  -- full SPARK declaration of
  -- type T would go here
end Stack;
```

- type packages do not have state nor *own* annotation
- roughly equivalent to a *class* or an ADT
- increasing level of abstraction and encapsulation:
  - Ada type
  - private type
  - limited private type
- Variables of the types declared in type packages can be declared at the point of use and passed as parameters
  - reduction of information flow and coupling

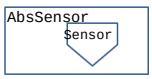
Utility Packages

- Never contain
  - state variables (they are not variable packages)
  - exported type declaration (they are not type packages)
- Define operations which
  - affect or use more than one variable/type package
  - would not be appropriate to consider part of one or another
- Given type packages representing "man" and "woman"
  - the operation "marry" should probably operate on one of each and would therefore need to be an utility package
  - Placing the marry operation in either the man or woman type package would suggest a false hierarchy and certainly be politically incorrect!

Boundary Variables

- Variable packages that
  - provide interfaces between the software functionality described by the INFORMED design and elements outside it with which it must communicate
  - with some kind of hardware sensor or actuator; or an "API" of some library or co-operating software system

Boundary Variable Abstraction Layers



- An abstraction layer between
  - the boundary variables of a system and
  - their users (other variable packages or the main)
- is appropriate where direct use of the boundary variables would allow too much detail to become visible in higher level SPARK annotations
- The abstraction may hide
  - that more than one boundary variable is involved in providing the abstract inputs
  - some other processing such as calibration that is taking place

Esempio: pulsanti di controllo

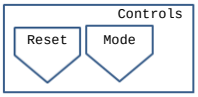
- cycle computer
- abstraction of two control buttons
  - Reset and Mode
  - obtained from a single external source
    - Controls.State

```
package Controls
--# own in State;
is
type Buttons is (Pressed, NotPressed);
procedure ReadReset(Setting : out Buttons);
--# global in State;
--# derives Setting from State;
procedure ReadMode(Setting : out Buttons);
--# global in State;
--# derives Setting from State;
end Controls;
```

Realizzazione dell'astrazione

```
--# inherit Controls;
private package Controls.Reset
--# own in State;
is
procedure Read(Setting :
          out Controls.Buttons);
--# global in State;
--# derives Setting from State;
end Controls.Reset;
```

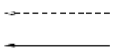
- Ciascun bottone ha la sua variabile volatile




```
with Controls.Reset, Controls.Mode;
package body Controls
--# own State is in Controls.Reset.State,
--#              in Controls.Mode.State;
is
procedure ReadReset(Setting : out Buttons)
--# global in Reset.State;
--# derives Setting from Reset.State;
is
begin
 Reset.Read(Setting);
end ReadReset;
```

Coupling (visually)

- package name *only* as a prefix in an *inherits* annotation represents **weak** coupling (use of a service)



- in a *global* or *derives* annotation indicates **strong** coupling (sharing of data)

Processo

1. Identification of
   1. the system boundary, inputs and outputs
2. Identification of the SPARK boundary
   1. Select a SPARK boundary within the overall system boundary; this defines the parts of the system that will be modelled and coded in SPARK and which will be subject to SPARK Examination
   2. Define boundary variables to give controlled interfaces across the SPARK boundary annotated in problem domain terms.
   3. Consider adding boundary abstraction layers

Processo (2)

3. Identification and localization of system state
   1. Identify the essential state of the system
      - what must be stored?
   2. Decide how to express the annotations of the main program
      - Any state outside the main program will appear in its annotations, any inside will not
   3. Assess where state should be located and whether it should be implemented
      - as variable packages or
      - instances of type packages
   4. Identify any natural state hierarchies and use SPARK refinement to model them. Introduce utility layers where appropriate.
   5. Test to see if the resulting provisional design is a loop-free partial ordering of packages and produces a logical and minimal flow of information. Backtrack as necessary.

Processo (3)

4. Handling initialization of state.
   1. Consider how state will be initialized. Does this affect the location choices made?
   2. Examine and flow analyse the system framework.
5. Implementing the internal behaviour of components.
   1. Implement the chosen variable packages and type packages using top-down refinement with constant cross-checking against the design using the Examiner.
   2. Repeat these design steps for any identified subsystems.

Inizializzazione dello stato

- all'elaborazione
  - prima dell'inizio del main
  - (parliamo dello stato delle variabili package)
- package elaboration part (i.e. between the begin and end of the package's body).
- By providing an initial value at the point of the variable's declaration:
  - X : Integer := 0;
  - SPARK limits!
- Implicit initialization by the external environment.
  - this is the case with boundary variables

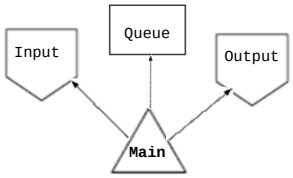
Inizializzazione dello stato (2)

- durante l'esecuzione del main
  - The variable does not have a valid value prior to execution of the statement concerned.
- By an assignment statement.
  - For concrete Ada variables any suitable expression may be assigned;
  - for variables of type packages a suitable deferred constant or function call will be required
- By a call to a procedure
  - which exports (and does not import) the variable concerned
  - this is the only way of initializing a variable of a type package implemented as an Ada limited private type

Inizializzazione: esempio

- Consider a tiny SPARK program that
  - transfers data from boundary variable “Input” to
  - boundary variable “Output”
  - via a global variable package “Queue”

```
--# derives Output.State,
--#          Queue.State
--#     from Queue.State,
--#          Input.State;
```



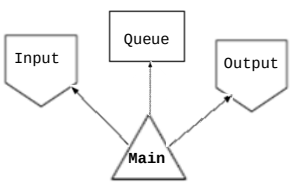
Già inizializzata in elaborazione

– devo chiamare subito enqueue che

--# derives State from State;

e quindi il main:

```
--# derives Output.State,
--#          Queue.State
--#     from Queue.State,
--#          Input.State;
```



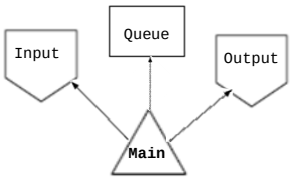
Da inizializzare

– devo chiamare prima Clear che

--# derives State from ;

e quindi il main non dipende più dallo stato iniziale della coda

```
--# derives Output.State,
--#              Queue.State
--#     from Input.State;
```

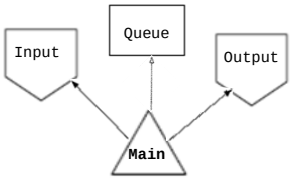


Se la coda diventa locale

– il main non dipende più dalla coda

– l'inizializzazione vien fatta dal 'costruttore'

```
--# derives Output.State
--#     from Input.State;
```



- Esempio di individuazione del flusso dell'informazione (vedi compilatore)

Esercizio

- Boiler Water Contents
- Requirements
- A device is needed to monitor the depth of water in a boiler vessel. Two sensors are provided “water low” and “water high”. When the water is low a fill valve is to be opened. When the water is high a drain valve is to be opened. When neither high nor low signal is present both valves are closed. To prevent the valves chattering some delay of operation is required with the valves only operating after 10 successive, consistent signals have been received from the associated sensor.