Structuur Van Computerprogramma's II

D. Vermeir

2008-2009

Outline

a good programs is:

- **correct**, i.e. it implements its **specification**,
- **robust**, i.e. it behaves **gracefully** when confronted with unexpected events,
- easy to **maintain** (maintenance costs 5$\times$ development). This implies:
  - it is easy to **understand**.
  - it is easy to **modify** and **extend**.
  - it consists of **parts** that can be **reused** elsewhere.

These criteria are not completely independent. E.g. decomposing into reusable parts may make the program easier to understand.

an example program

```
#include <iostream>
#include <stdlib.h> // for strtod(char*,char**)
// quotient: write quotient of arg1 and arg2 on stdout.
int
main(int argc,char* argv[]) {
  // strtod(char* p,0) converts initial part of
  //   C-string starting at p to double
  std::cout << strtod(argv[1], 0)/strtod(argv[2], 0) << "\n";
}
```

Is this a good program?

a robust version

But more difficult to read & maintain.

```
#include <string>
#include <iostream>
#include <stdlib.h> // for strtod(char*,char**)
// quotient: write quotient of arg1 and arg2 on stdout.
static const std::string USAGE("quotient number number");
static const std::string FORMAT_ERR("not a number");
static const std::string DIVIDE_BY_ZERO("divide by 0");

int
main(int argc, char* argv[]) {
  char *end; // see the man page for strtod

  if (argc!=3) {
    std::cerr << "usage: " << USAGE << std::endl;
    return 1;
  }
```

```
  double  a1(strtod(argv[1], &end));
  if (end==argv[1]) {
    std::cerr << "\"" << argv[1] << "\": "
      << FORMAT_ERR << std::endl;
    return 1;
  }

  double  a2(strtod(argv[2], &end));
  if (end==argv[2]) {
    std::cerr << "\"" << argv[2] << "\": "
      << FORMAT_ERR << std::endl;
    return 1;
  }
  if (a2==0) {
    std::cerr << DIVIDE_BY_ZERO << std::endl;
    return 1;
  }

  std::cout << a1/a2 << std::endl;
  return 0;
}
```

good decomposition: better maintenance

```
#include <iostream>
#include <string>
#include <stdexcept> // for standard exception classes
#include <stdlib.h> // for strtod(char*, char**)
// quotient: write quotient of arg1 and arg2 on stdout.
static const std::string
  USAGE("usage: quotient number number");
static const std::string
  DIVIDE_BY_ZERO("cannot divide by 0");
```

a reusable part

```cpp
// A reusable part: this function returns the double
// represented by s; it throws a range_error exception
// if s does not represent a double.

double
cstr2double(const char* s) throw(std::range_error) {
  static const std::string
    FORMAT_ERR("cstr2double: not a number");
  char*  end;
  double  d(strtod(s, &end));

  if (s==end)
    throw std::range_error(std::string(s)+": "+FORMAT_ERR);
  return d;
}
```

new main program

```
int
main(int argc, char* argv[]) {
  try {
    if (argc!=3)
      throw std::runtime_error(USAGE);
    double  a1(cstr2double(argv[1]));
    double  a2(cstr2double(argv[2]));
    if (a2==0)
      throw std::runtime_error(DIVIDE_BY_ZERO);
    std::cout << a1/a2 << std::endl;
    return 0;
  }
  catch (std::exception& e) {
    // reference preserves e's "real" type
    std::cerr << e.what() << std::endl;
    return 1; // error return
  }
}
```

Easier to understand **and** more reusable parts.

a bad decomposition

```
#include <iostream>
#include <string>
#include <stdexcept> // for standard exception classes
#include <stdlib.h> // for strtod(char*, char**)

static const std::string
  USAGE("quotient number number");
static const std::string
  FORMAT_ERR("not a number");
static const std::string
  DIVIDE_BY_ZERO("cannot divide by 0");
```

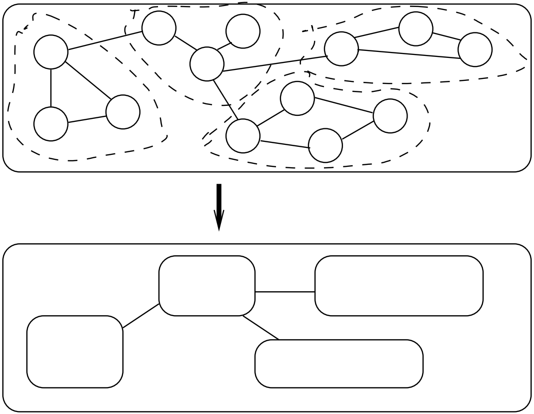
a part that is less reuseable

```
// Get two doubles from two C strings in an array.
bool
get_arguments(char* args[], double& arg1, double& arg2) {
  char *end; // see the man page for strtod

  arg1 = strtod(args[0], &end);
  if (end==args[0]) {
    std::cerr << "\"" << args[0] << "\": "
      << FORMAT_ERR << std::endl;
    return false;
  }

  arg2 = strtod(args[1], &end);
  if (end==args[1]) {
    std::cerr << "\"" << args[1] << "\": "
      << FORMAT_ERR << std::endl;
    return false;
  }
  return true;
}
```

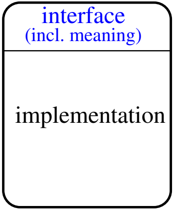
```
int
main(int argc, char* argv[]) {
  if (argc!=3) {
    std::cerr << "usage: " << USAGE << std::endl;
    return 1; // program failed
  }
  double a1;
  double a2;
  if (!get_arguments(argv+1, a1, a2))
    return 1; // program failed
  if (a2==0) {
    std::cerr << DIVIDE_BY_ZERO << std::endl;
    return 1; // program failed
  }
  std::cout << a1/a2 << std::endl;
  return 0;
}
```

design = decomposition

Decompose such that overall structure (**architecture**) becomes **simpler**, using **abstractions.**



an example abstraction



(a)

```
double cstr2double(const char* s) throw(range_error)
// convert C string to double or throw range_error

static const FORMAT_ERROR("not a number");
char* end;
double d(strtod(s,&end));
if (s==end)
  throw range_error(string(s)+": "+FORMAT_ERR;
return d;
```

(b)

An abstraction has

- An **interface** which is as simple as possible and which hides
- a possibly complex **implementation**.

C++ abstraction mechanisms

- **Functions** abstract **behavior.**
- **Classes** abstract **data + behavior.**
- **Templates** abstract structurally similar skeleton data and/or behaviors.
- **Overloading** abstracts different behavior with same “meaning”.
- **Inheritance** abstracts common interface for related concepts.

components and modules

Several functions and/or classes may be needed to represent a single abstraction. A **component** is such a collection. A **module** is the physical representation of a component: typically a header file with the interface(s) and a collection of source files containing the implementation.

Example

class AccountDatabase,
class AccountDatabase::iterator.

Example

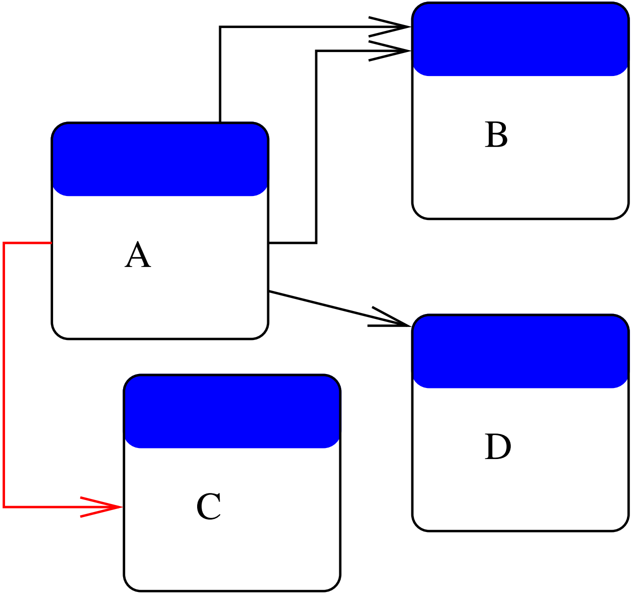
class Rational,
Rational operator+(const Rational&, const Rational&)

dependencies between abstractions

- **Interface** dependency: when the interface of an abstraction depends on another abstraction, e.g. a function depends on the (class) type of its parameters.
- **Implementation** dependency: when the implementation of an abstraction depends on another abstraction, e.g. a function's body may call other functions.

⇒ Ideal for ease of understanding and reuse:
  - Minimize dependencies.
  - **Only depend on interface** of other abstractions (**encapsulation**).
  - The interface should have fewer dependencies than the implementation.

dependencies between abstractions



interface dependencies in C++

Mechanism	Interface dependencies
function	types of parameters and of thrown exceptions
class	types of public members (functions and data)
publicly derived class	as above, plus the interface, implementation and dependencies of the base class
base class	as above, plus the implementation of any pure virtual functions by its derived classes
function template	types of non-template parameters and thrown exceptions, abstract types of template parameters
class template	non-template types of public members, abstract types of template parameters

inheritance dangers: BookCollection

```
class BookCollection {
  public: // stuff omitted
    typedef std::set<Book>::const_iterator iterator;

    iterator  begin() const { return books_.begin(); }
    iterator  end() const { return books_.end(); }

    virtual bool add(const Book& book) {
      return books_.insert(book).second;
    }

    virtual void merge(const BookCollection& collection) {
      for (iterator i=collection.begin();
           i!=collection.end(); ++i)
        add(*i);
    }
  private:
    std::set<Book>  books_;
};
```

TrackedBookCollection

A BookCollection that keeps statistics on the number of additions.

```
class TrackedBookCollection: public BookCollection {
  public: // stuff omitted
    int statistics() const { return n_additions_; }

    virtual bool add(const Book& book) {
      bool ok(BookCollection::add(book));
      if (ok) // keep count in n_additions_
         ++n_additions_;
      return ok;
    }
  private:
    int  n_additions_; // number of books added to collection
};
```

more efficient BookCollection

```
class BookCollection {
  public: // stuff omitted
    virtual bool add(const Book& book) {
      return books_.insert(book).second;
    }
    virtual void merge(const BookCollection& collection) {
      // more efficient: set<T> bulk insert
      books_.insert(collection.begin(), collection.end());
    }
  private:
    std::set<Book> books_;
};
```

- Now TrackedBookCollection:: statistics () has different meaning!
- TrackedBookCollection depends on implementation of BookCollection::merge

commonalities and variabilities

- An abstraction can be regarded as representing the set of its **instances**. E.g. a function represents all its calls, a class all its instance objects, a template all its instantiations.
- Each abstraction supports the specification of certain **commonalities** over its instances as well as **variabilities** that vary with the instantiation.

what to use when (1/2)

Commonality	Variability	C++ feature
function name and behavior, parameter types	parameter values	function
function name and semantics	everything else	overloaded function definitions
function name and behavior	everything else, e.g. parameter types	function template
precise behavior of operations available for an object and data structure of an object	actual data member values ("state") representing an object	class

what to use when (2/2)

Commonality	**Variability**	**C++ feature**
name and semantics (including type) of the related operations available on an object and (possibly) some data structure	everything else	abstract class and inheritance
precise behavior of operations available for an object and “template” data structure of an object	actual types used in the data structure and the operations	class template

negative variability

C++ has mechanisms to support **negative variability**, i.e. certain instances of the abstraction differ w.r.t. some commonalities:

- overloading
- template specialization
- function overriding in derived classes

sources for abstractions

Abstractions may "come from":

- **problem space**, i.e. the specifications of the application. E.g. `Customer`, `Account`.
- **solution space**, i.e. the implementation techniques used to implement the system. E.g. `Thread` for a multi-threaded system, container classes etc.

a good abstraction:

- is **non-trivial**
- is **abstract**
- has **high cohesion**
- has **low coupling**

non-trivial

```
double add6percent(double x) { return x * 1.06; }
```

is too trivial but

```
double
add_vat(double x) {
  static const double VAT_RATE(6.0);
  return x * (100 + VAT_RATE) / 100;
}
```

may be ok.

abstract

More abstract entities have a better chance of being reusable.

```
class Person {  // lots of stuff omitted
  public:
    Date  birth_date() const;
};

class Student: public Person {
  // lots of stuff omitted
};

int age(const Student& p) {
  return Date::now() - p.birth_date()
}
```

increasing abstraction

- More abstract:

```
int age(const Person& p) {
  return Date::now() - p.birth_date();
}
```

- Even more abstract:

```
template <class ThingWithBirthDate>
int age(const ThingWithBirthDate& t) {
  return Date::now() - t.birth_date();
}
```

more abstract is often more powerful

```
// product: write product of arg1, arg2, arg3
static const std::string  USAGE("usage: product num num num");

int
main(int argc, char* argv[]) {
  try {
    if (argc!=4) throw std::runtime_error(USAGE);
    double  a1(cstr2double(argv[1]));
    double  a2(cstr2double(argv[2]));
    double  a3(cstr2double(argv[3]));
    std::cout << a1*a2*a3 << std::endl;
    return 0;
  }
  catch (std::exception& e) {
    // reference preserves e's "real" type
    std::cerr << e.what() << std::endl;
    return 1; // error return
  }
}
```

```
#include <algorithm> // for transform
#include <numeric> // for accumulate
#include <vector>

static const std::string USAGE("usage: product [number]..");

int
main(int argc, char* argv[]) {
  try {
    std::vector<double> args(argc-1);
    std::transform(argv+1, argv+argc,
                   args.begin(), cstr2double);
    std::cout
      << std::accumulate(args.begin(), args.end(),
                         1.0, multiplies<double>())
      << std::endl;
    return 0;
  }
  catch (std::exception& e) {
    std::cerr << e.what() << std::endl;
    return 1; // error return
  }
}
```

high cohesion

- a **function** should do only 1 thing (and do it well)

⇒ **functional cohesion**

- a **class** should encapsulate data that are closely related and all necessary operations on these data (as member or friend functions)

⇒ **data cohesion**

low cohesion example

```
class Person { // stuff omitted
  public:
    Person(const std::string& name, int yr, int mo, int dy);
    std::string  name() const;
    std::string  birth_date(const std::string& format) const;
  private:
    std::string  name_;
    int  birth_year_;
    int  birth_month_;
    int  birth_day_;
};

Person lisa("Lisa", 1980, 12, 1);
std::cout << lisa.birth_date("%d %b, %Y");
// prints "1 december, 1980"
```

higher cohesion example

```
class Date { // stuff omitted
  public:
    Date(int year, int month, int day);
    std::string  str(const std::string& format) const;
    int  day_of_week() const;
  private:
    int  year_;
    int  month_;
    int  day_;
};

class Person { // stuff omitted
  public:
    Person(const std::string& name, const Date& d);
    std::string  name() const;
    const Date&  birth_date() const { return birth_date_; }
  private:
    std::string  name_;
    Date  birth_date_;
};
```

low coupling, mimimize dependencies

(bad) **representational** coupling: e.g. (member) function directly accessing a data member of another class.

$\Rightarrow$ always declare data members **private**

$\Rightarrow$ use accessor functions, if possible also within member functions

(bad) **global** coupling: e.g. dependence on global variable

$\Rightarrow$ never use global variables

(ok) **parameter** coupling

$\Rightarrow$ function uses only its parameter objects

(bad) **control** coupling

(bad) **derived class** coupling

control coupling

Caller explicitly determines flow of control in function, e.g. by passing a “flag”.

```
class Database {
  public: // lots of stuff omitted
    bool store(bool open_first, const Tuple& tuple) {
      if (open_first) {
        // open database
      }
      // store tuple
    }
};
```

control coupling, how to avoid

```
class Database {
  public:
    bool open(const std::string& name);

    // returns true iff database has been opened
    bool is_open() const;

    bool store(const Tuple& tuple) {
      if (!is_open())
        return false;
      // store tuple
    }
};
```

derived class coupling vs composition

- Derivation causes mutual dependencies between base and derived classes.

```
class Person: public Date {
  private:
    std::string name_;
};
```

⇒ use **composition** unless there is a clear **is-a** relationship between derived and base.

```
class Person {
  private:
    std::string  name_;
    Date  date_of_birth_;
};
```

- Private (or protected) derivation does not commit the public interface of a derived class.

design

- **iterative**: analyze → design → implement → evaluate → analyze → ...
- design steps:
  1. find abstractions
     1.1 distribute desired functionality over **domain classes** that provide **services**, possibly in collaboration with (objects of) other classes.
     1.2 add solution space classes to support the work of the domain classes (e.g. containers).
  2. **refactor**: improve by introducing more general and reusable abstractions; e.g. fuse into template, introduce common base class, ...