Design Experiences in C++

Designing With Objects

Mark Radford
twoNine
Computer Services Limited

mark.radford@twonine.co.uk
www.twonine.co.uk

1

Contents

2

Values & Quantities in C++

- *Whole Values*
  - The problem
  - The *Whole Value* solution and its costs
- Quantities
  - Selected issues and pitfalls

3

The Problem

- In languages without data abstraction support, built in types are used to represent values, consequently:
  - Compile-time type checking is weak
  - Communication is weak (because the domain vocabulary is absent from the code)
- Unfortunately, C++ programmers have a tendency to follow suit

4

Whole Values

```
class minutes
{
public:
  explicit minutes(int initial);
  ...
};
```

```
class transaction
{
public:
  void timeout(
    const minutes& duration);
  ...
};
```

```
void f(transaction* current)
{
  current->timeout(minutes(10));
  ...
}
```

- In C++ classes should be used to implement first class data abstractions
  - The compiler is empowered to do stronger type checking
  - The code communicates application domain vocabulary

5

The Cost

- There is effort in producing the classes
  - Automating their production (e.g. using templates) helps keep this cost down
- The proliferation of classes must be managed
  - This cost is something of a red herring with the right tools
  - Tools cost money, but the cost is tangible

6

Automation Using Templates – Pitfall

The problem

```
template <typename T> class value { ... };

typedef value<unsigned int> seconds;
typedef value<unsigned int> minutes;
```

Both types are the same in the eyes of the compiler

Solution by type tagging

```
template <
  typename type,
  typename tag> class value { ... };

struct secs_tag {};
typedef value<int, secs_tag> seconds;

struct mins_tag {};
typedef value<int, mins_tag> minutes;
```

The empty tag structs ensure that `seconds` and `minutes` are both unique types

7

Quantities

- E.g. minutes, voltages, kilograms
  - Quantities account for much of the value based bandwidth in programmes
- Quantities have their own additional specific characteristics, e.g…
  - They have *units*
  - They can have a value of zero

8

Quantity Abstraction Pitfall

```
class time_interval
{
public:
  struct from_seconds {};
  struct from_minutes {};

  time_interval(
    int value, const from_sec&);
  time_interval(
    int value, const from_min&);
  ...
};

std::ostream& operator<<(
  std::ostream& os,
  const time_interval& value);
```

Not only does initialisation require a very artificial looking interface design…

What exactly, is inserted into the stream?

9

Abstract Quantities as Units

```
class minutes
{
public:
  explicit minutes(int initial);

  minutes();
  ...
};


std::ostream& operator<<(
  std::ostream& os,
  const minutes& value);
```

Initialisation is now straightforward – either by default or from a single value

Operator overloads come naturally, and their meaning is automatically clear

10

Initialisation & Conversion

```
class minutes
{
public:
  typedef int value_type;

  minutes();

  explicit
    minutes(value_type initial_value);

  value_type value() const;
    ...
};
```

Default initialisation value of zero is a feature of quantity types

Conversion *must* be explicit to convey meaning

Conversion back to underlying type must be invoked *deliberately*

Note: the provision of `value()` *is necessary for the range of supported operations to be extensible*

11

The Costs vs. Benefits Summary

- Compile-time error detection is an asset
  - The cost of run-time debugging is open-ended
  - The benefit of compile time error detection is the greater *predictability* of costs
- Whole Value classes belong to the application domain
  - Subsequent projects can build on the existing application domain library

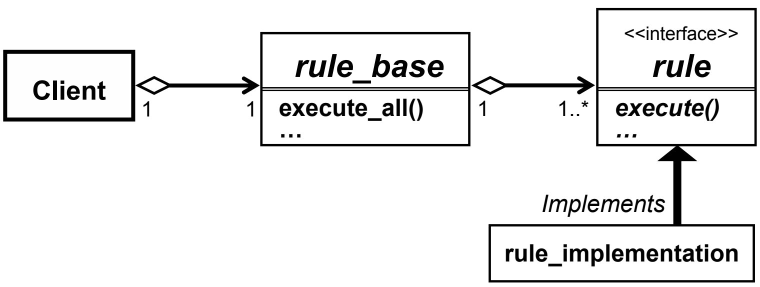
12

Choosing an Approach to Iteration

- Design of a system of business rules
- Approaches to iteration – two alternatives to the STL approach compared
- A design decision re-evaluated

13

Example – A System of Business Rules



The `rule_base` is the repository that manages the storage and execution of the rules.

14

The Iteration Problem

- `rule_base` must provide some mechanism to allow iteration through the `rule`s it stores
- The STL style is not always an appropriate choice
  - It is fragile in cases where updates and iteration might happen concurrently
  - STL style components provide a toolkit, but sometimes more of an application *framework* is needed

15

Alternatives to STL style iteration

- Batched copying
  - `rule_base` (internally) makes a complete copy of its collection of `rule`s and returns it in a single operation
- Enumeration by callback
  - `rule_base` drives the flow of control during iteration
  - A callback function supplied by the client is called for each `rule`

16

Batched Copying

```
class rule_base
{
public:
  template <
    typename out_it,
    typename copy_fn>
  void copy(
    out_it dest,
    copy_fn make_copy) const
  {
    mutex guard;
    std::transform(
      rules.begin(),
      rules.end(),
      dest,
      make_copy);
  }
  ...
private:
  std::list<rule*> rules;
};
```

- The `copy()` member copies the state of each rule to a sequence starting at `dest`
- Knowledge of the value based data structure used by the sequence is encapsulated in `make_copy()`
- Allowing `rule_base` to drive the control flow allows synchronisation to be made integral to the copying process

17

Enumeration by Callback

```
class rule_base
{
  typedef std::list<rule*> container;
  typedef container::const_iterator
    const_iterator;
public:
  void for_each(
    void(*callback)(const rule&))
  {
    mutex guard;
    for (const_iterator it=rules.begin();
         it != rules.end();
         ++it)
      callback(**it);
  }
  ...
private:
  container rules;
};
```

- The `for_each()` member calls `callback()` for each stored `rule`
- Putting `rule_base` in charge of the control flow allows synchronisation to be made integral to the iteration

18

Interface Class as Callback

- Use of an *interface class* allows `callback` to be treated as *mixin* functionality
  - note that `callback_client`'s destructor is *protected* and *non-virtual*, clearly stating that it is not intended as primary design type

```
class callback_client
{
public:
  virtual void
    callback(const rule& obj) = 0;
Protected:
  ~callback_client();
};
```

```
class rule_base
{
public:
  void for_each(
    callback_client& cl) const;
  ...
};
```

```
class rules_view :
  public view,
  private callback_client
{
public:
  void update(const rule_base& rb)
  { rb.for_each(*this); }
  ...
private:
  virtual void callback(
    const rule& r)
  { Code to add rule to view ... }
  ...
};
```

19

The Problem of Selective Traversal

- In many cases the callback function should only be called for `rules` that execute in response to a particular `trigger` event
  - Making filtering options available via overloading means the choice of filtering must made at compile time

```
class rule_base
{
public:
  void for_each(callback_client& c) const;
  void for_each(callback_client& c, const trigger& e) const;
  ...
};
```

20

Adding a Run Time Filter

```
class callback_filter
{public:
  virtual bool required(const rule& obj)
    const=0;
protected:
  ~callback_filter();
};
```

```
class rule_base
{public:
  void for_each(
    callback_client& client,
    const callback_filter& filter) const
  {
    for (const_it it = rules.begin();
         it != rules.end();
         ++it)
      if (filter.required(**it))
        client.callback(**it);
  }
  ...
};
```

- The most versatile approach involves `for_each()` taking an interface class that defines a (virtual) filter function as a second parameter
  - The filter function returns `true` if the `rule` is to be included

21

Tradeoffs of Run Time Filter Approach

- Introducing run-time polymorphism affords more versatility than the compile-time polymorphism of overloading
- But, `rule_base`'s implementation can't be taken advantage of
  - E.g. maintaining indices to improve traversal performance is much more complicated

22

Design Decision Re-evaluated

- With the benefit of hindsight, using a Batched Copying approach was probably simpler
  - It isn't clear that the framework afforded by Enumeration by Callback (in conjunction with Callback Using Interface Class) lead to cleaner code

23

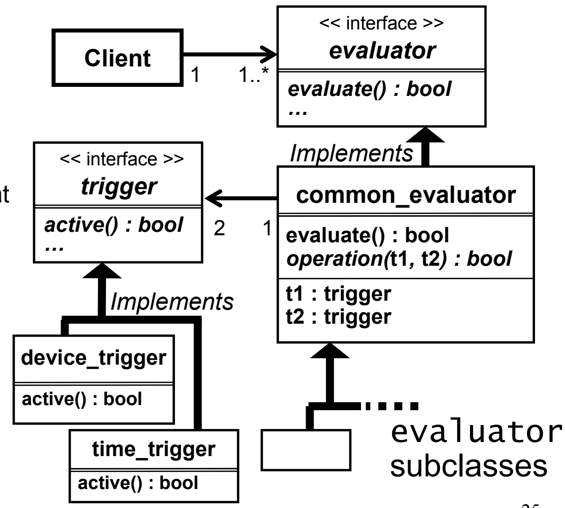
Mixing Generic with Object Oriented Programming

- Design of (part of) a logical expression evaluation engine
- The tradeoffs in mixing dynamic polymorphism with generic programming in design

24

An Evaluation Engine for a Security System

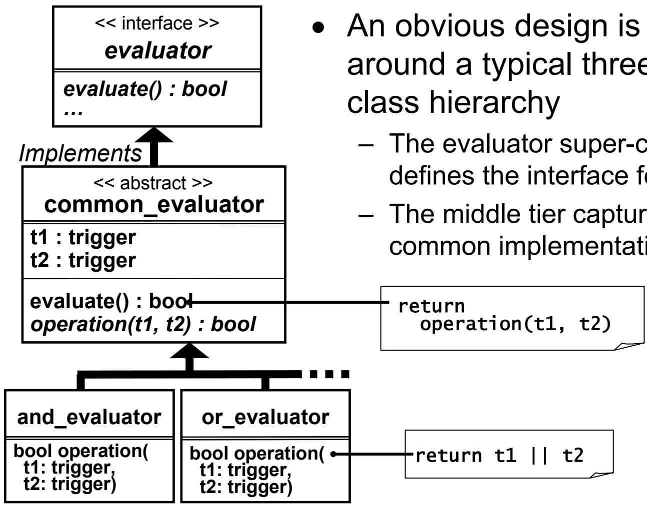
- `trigger` objects represent:
  - Physical devices such as motion detectors
  - Time intervals that are entered daily, weekly etc.
- `evaluator` evaluates whether or not a combination of `triggers` requires action to be taken



The C++ Three Level Hierarchy

- The three level inheritance hierarchy is a common design idiom in C++
  - The base class implements only the common interface (pure virtual functions only)
  - The next level of derivation captures common implementation detail in an abstract class
  - The most derived level contains the concrete classes

The `evaluator` Hierarchy



- An obvious design is based around a typical three level class hierarchy
  - The evaluator super-class defines the interface for clients
  - The middle tier captures common implementation

27

The Principle C++ *Interface* Classes

```
class trigger
{
public:
  virtual ~trigger();
  virtual bool active() const = 0;
};
```

```
class evaluator
{
public:
  evaluator() {}
  virtual ~evaluator();
  virtual bool evaluate() const = 0;
private:
  evaluator(const evaluator&);
  evaluator& operator=(const evaluator&);
};
```

- `trigger` & `evaluator` are *interface* classes in C++
  - `trigger`'s `active()` member function returns true if the device the `trigger` object represents has been activated
  - `evaluator`'s `evaluate()` evaluates an expression made up of two `trigger` objects

28

The Problem of Repetitive Code

```
class common_evaluator : public evaluator
{protected:
  common_evaluator(
    shared_ptr<trigger> trigger_1, shared_ptr<trigger> trigger_2);
private:
  virtual bool evaluate() const { return operation(*t1, *t2); }
  virtual bool operation(const trigger& t2, const trigger& t1) const = 0;
  trigger::shared_ptr t2, t1;
  ...
};
```

```
class or_evaluator : public common_evaluator
{public:
  or_evaluator(
    boost::shared_ptr<trigger> trigger_1,
    boost::shared_ptr<trigger> trigger_2)
  : common_evaluator(trigger_1, trigger_1)
  {}
private:
  virtual bool operation(
    const trigger& trigger_1,
    const trigger& trigger_2) const
  { return trigger_1.active() ||
           trigger_2.active(); }
  ...
};
```

- The repetition of code in the concrete classes introduces a risk of error
  - Note the erroneous appearance of `trigger_1` which may or may not have been duplicated in `and_evaluator`

29

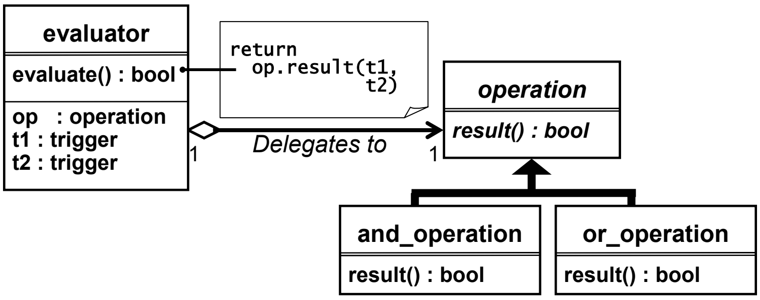
Alternative Factorings of Variability

- An alternative factoring taking better account of the commonality/variability in the `evaluator` hierarchy is needed
  - The variability is purely behavioural (i.e. the relational operation)
- Options are available for configuring the variability at both statically and dynamically
  - Naturally each has different tradeoffs associated with it

30

Solution Using Run-Time Delegation

- The variability can be factored into a separate *interface* class
  - Evaluator is then configured at when it is instantiated



31

Delegation Solution – Implementation

```
class operation
{public:
  virtual bool result(
    const trigger& t1, const trigger& t2)
      const = 0;
...
};
```

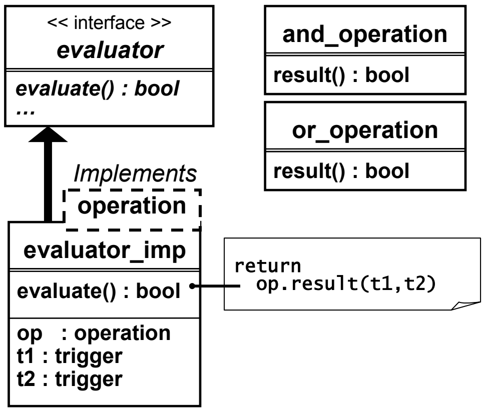
```
class evaluator
{public:
  evaluator(
    boost::shared_ptr<trigger> trig_2,
    boost::shared_ptr<trigger> trig_1,
    boost::shared_ptr<operation> eval_op);
private:
  virtual bool evaluate() const
  { return op->result(*t1, *t2); }

  boost::shared_ptr<trigger> t2,
  boost::shared_ptr<trigger> t1,
  boost::shared_ptr<operation> op;
};
```

- Less repetition of code leads to fewer opportunities for error
- But there are still many objects whose lifecycles must be managed
  - The client code has to deal with `operations` as well as `triggers`

32

Solution Using *Static* Parameterisation

- The variability can be captured using a parameterised class
  - A C++ *class template* being the natural implementation



Implementation by C++ Class Template

```
template <typename function>
class evaluator_implementor : public evaluator
{public:
  evaluator_implementor(
    shared_ptr<trigger> trig1, shared_ptr<trigger> trig2)
    : t1(trig1), t2(trig2) {}
private:
  virtual bool evaluate() const
  { return function()(t1->active(), t2->active()); }
  shared_ptr<trigger> t1, t2;
};
```

```
typedef evaluator_implementor<std::logical_and<bool> >
    and_evaluator;
...
```

- Specialisations for *and*/*or* (etc.) can be created using function objects available in the standard library

Observations

- An approach using delegation leads to simpler and less error prone code
  - The approach using inheritance would produce lots of similar code
- Combining generics with dynamic polymorphism lead to the simplest implementation
  - No duplication of almost the same code
  - Easy to extend (e.g. for an XOR operation)

35

Complexity & Design

- Ignoring complexity is costly
  - How to deal with the complexity is an issue to be addressed during design
- C++ has features designed to help absorb complexity
  - E.g. templates and exceptions
  - But understanding these (advanced) features places an added burden on the skills of the programmer

36

Observations on Design

- Design/programming paradigms
- Multi-Paradigm design
- An alternative perspective on combining design/programming paradigms

37

Inheritance & Run Time Polymorphism

- It is a popular myth that inheritance is a defining feature of an object oriented technology
  - Several programming languages have helped propagate this myth by using inheritance as their run-time polymorphism mechanism
- Inheritance and run-time polymorphism are individual and separate concepts

38

Object Design Paradigms

- Object based
  - Supports abstraction and encapsulation
  - E.g. value based programming using the *Whole Value* idiom
- Object oriented
  - Defined by the presence of run-time polymorphism as a first class feature

39

Other Design Paradigms

- Procedural
  - Concerned with flow of control
- Generic
  - Separates types, data structures, operations and flow of control
  - Not useful in its own right, but used in combination with other paradigms

40

Multi-Paradigm Design

```
class shape
{public:
  virtual void rotate(const radians& angle) = 0;
  ...
};
class line : public shape { ... };
...
```

```
void rotate_all(
  std::vector<shape*>& shapes,
  const radians& angle)
{
  typedef std::vector<shape*>::iterator
    iterator;
  for (iterator pos = shapes.begin();
       pos != shapes.end();
       ++pos) {
    shape* current = *pos;
    current->rotate(angle);
  }
}
```

Generic programming

Procedural programming

Object oriented programming

41

Tools in the Mental Toolbox

```
class shape
{public:
  virtual void rotate(const radians& angle) = 0;
  ...
};
class line : public shape { ... };
...
```

```
void rotate_all(
  std::vector<shape*>& shapes,
  const radians& angle)
{
  typedef std::vector<shape*>::iterator
    iterator;
  for (iterator pos = shapes.begin();
       pos != shapes.end();
       ++pos) {
    shape* current = *pos;
    current->rotate(angle);
  }
}
```

Static parameterisation

Flow of control

Run-time polymorphism

42

Designing With Objects

- One of the great benefits of C++ is its support for a variety of design paradigms
  - But in practice, distinguishing between paradigms constrains the thought process
  - It is better to view elements of paradigms as tools in the intellectual toolbox of the design process as a whole

43

<u>The End</u>

- I hope you found this talk interesting
- Thank you for your attention!

44