Pattern Experiences in C++

Mark Radford

twoNine *Computer Services Ltd*

mark.radford@twonine.co.uk

www.twonine.co.uk

1

ACCU Spring Conference, April 2003.

Mark Radford

twoNine Computer Services Ltd

mark.radford@twonine.co.uk

www.twonine.co.uk

Agenda

- About patterns
- Design & lessons learned
  - Real world C++ design episodes

2

This talk is about my experiences with patterns - taken from the book *Design Patterns* and from other sources - over the last seven years, and it falls into two parts.

About patterns gives an introduction to patterns and why they are of interest. Various points made in this section coincide with advancements made in my understanding over these years.

Design & lessons learned recounts a couple of episodes from my professional work that I look back on as particularly instructive. In both these episodes, I want to discuss a piece of design work (or recurring feature in design work), that I would do differently now.

About Patterns

- Motivation
- Types of Pattern
- Fundamentals & Exposition

3

Patterns as they are now known, came to the attention of the software development community in the 1990s and have accumulated a healthy body of literature: the “Gang of Four” book is the best known and the one responsible for getting the mainstream of the community interested. Unfortunately, other works have not achieved such a high profile, and this has left too many people unaware that the Designs Patterns book is just one resource in the body of patterns literature. Other examples include: four books of selected papers from four Pattern Languages of Programming conferences (see *PLoP*, *PLoP2*, *PLoP3*, *PLoP4*), two (at the time of writing) volumes of the series now known as “POSA” (*POSA*, *POSA2*), and various resources on the internet such as the Portland Pattern Repository (*Portland Pattern Repository*).

Motivation

- Solving design problems uses resources
  - Deploying the wrong solution is wasteful
- Problem solving is an area of *risk*
  - To minimise the risk, *all facets* of the problem and its solution must be understood

4

Patterns are about capturing solutions to problems. Therefore, before going on to try to understand patterns, we need to step back and look at the problem solving process. To make things go right we first need to understand what can go wrong.

One of the activities in project management is managing the *risk*s associated with the project, and doing whatever possible to minimise the overall risk. Obviously deploying a solution is wasteful, if ultimately and for whatever reason, it fails to solve the problem or solves the wrong problem. It follows that problem solving is an area of *risk* for a project; while this may seem like an obvious thing to say, problem solving – and in particular the approach developers take to it – is all too frequently a neglected area of risk.

Solutions & Tradeoffs

- There is hardly ever any such thing as *the* solution to a problem
  - Choosing a solution from the available options involves accepting *tradeoffs*
- A classic example…
  - Execution speed improvement versus amount of memory used

5

Some simple problems might have simple and absolute solutions, but a vast majority of the time life just doesn't work like that! Unfortunately my experience has been that if there is one thing in software design that is too often missed, this is it. Too often developers think they have solved a problem: well maybe it looks that way, but what they have actually done is trade one thing for another. Yes, the problem has gone away and can therefore be considered solved, but in order to gain that benefit, a price will have been paid somewhere.

Execution speed versus memory usage is a classic example of a tradeoff. In his recent Overload article (see *Indexing STL*) Silas Brown describes a method of improving the speed of `std::list` element lookup by using a `std::map` to maintain an index of the elements in the list. This is an prime example of the speed versus memory trade – the index (`std::map`) requires memory in which to store its contents (as the author points out in the concluding paragraphs).

Patterns

- Patterns capture known problems and solutions
  - No need to reinvent knowledge
    - Solutions with solid track records are captured and placed on record
  - Pattern use raises confidence
    - Approaches used are known to be tried and tested

6

Patterns have existed “in spirit” in the software development community for as long as the community itself, even though they didn’t have a name. The point is this: skilled software developers have always known that some ways of doing things just felt like the right way. This phenomenon is not unusual among craftsmen of various disciplines. For example, the work of the architect (that is, architect in the building sense) Christopher Alexander [*The Timeless Way of Building* and *A Pattern Language*] provided prior art behind patterns in software!

Software developers often experience a sense of deja-vu when examining a problem they are trying to solve. Often, the inability to pin down exactly where and when they have seen this problem before is a source of some frustration – the obstacle being the fact that often problems appear over and over again in different guises. Strangely enough, when a solution is found, it also has a look and feel of familiarity.

Essentially patterns seek to record problems and their solutions. However, doing so is not as simple as it may seem, because if doing so is to be of benefit, the tradeoffs accepted must also be recorded.

Types of Pattern

- Each stage of the development process has patterns applicable to it, for example…
  - *Architectural patterns* deal with issues of overall system structure
  - *Design patterns* deal with system components and the interactions between them
  - *Idioms* are applicable at the level of the programming language

7

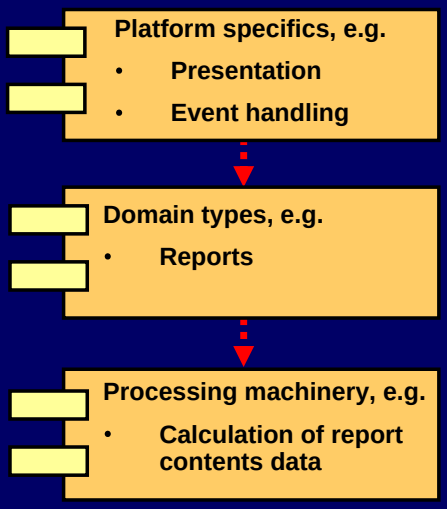
Patterns occur at, and are of benefit to, all stages of the development cycle. It is unfortunate that only patterns aimed at object oriented design have really caught on (and object oriented design patterns have only caught on because of the popularity of *Design Patterns*).

The book *Pattern Oriented Software Architecture* (*POSA*) came out soon after *Design Patterns*, and covers architectural patterns, design patterns and idioms. Also, the design patterns in it talk in terms of components rather than objects, and in doing so take a more generalised view of design than the object oriented design patterns presented in *Design Patterns*.

Another noteworthy book is *Analysis Patterns*, presenting patterns occurring in logical models from various application areas. Some are fairly specific, but some – notably those relating to observations and measurements – are more generally relevant.

Layers – Architectural Pattern

Intent: organise and separate levels of abstraction



Example

Many systems decompose clearly into three layers, each using the services of the one immediately below.

8

The use of abstraction is fundamental to software development, and is an essential part of the designer's mental toolkit. The system is decomposed into functionality at different levels of interest. Components are grouped together to form broader abstractions in a hierarchical structure, where any particular layer uses only components from the layer below it. Alternatively, in some layered designs, use of components from any lower layer is permitted – such a design is known as a *relaxed layered system* (see *POSA*).

The architectural pattern *Layers* has all the hallmarks of a good pattern. It can be found in software going back through the years – not in all software, but its presence very often coincides with structure that gives the impression people who were thinking clearly when they designed it. Patterns are discovered, not invented. The seminal documentation of *Layers* as a pattern is in *POSA*.

Proxy – Design Pattern

- Intent
  - Provide a surrogate allowing transparent but controlled access to the target object
- For example…
  - Defer loading an object's state until first use
  - Facilitating access to a target object on a different computer via a network

9

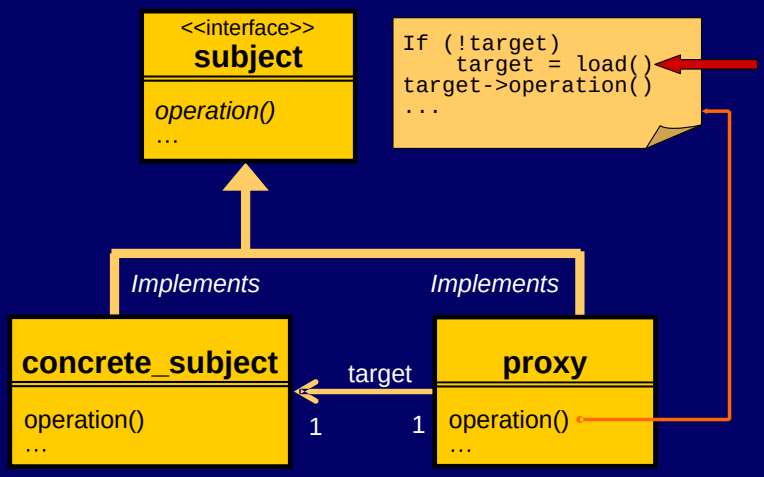
A client needs to access the services of a target object (or, more generally, target component), where direct access to the target is technically possible but is inappropriate. *Therefore*, provide a surrogate object for the client to access that absorbs whatever "machinery" is needed between the client and target, avoiding intrusion of such machinery into the client code.

For example (in addition to the examples on the slide), a proxy could be used to reference count the target object. Also, it may be necessary, for security reasons, to first negotiate with the process in which the target object runs before access to the object is allowed.

The slide cites the use of a *Proxy* to transparently access an object via a network where the target object is running on another computer: in such cases it is possible for reasons of efficiency, for the Proxy to cache results from the target object. Where such a "Caching Proxy" is used, there is obviously a requirement for a strategy by which the Proxy's cache can be notified when the state of the target object is updated; this may involve the cache being fully refreshed, or simply invalidated so that when its state is queried the query really is forwarded all the way to the target object.

The seminal documentation of *Proxy* is in *Design Patterns*. Additional material, in particular classification of different kinds of Proxy, can be found in *POSA*.

Proxy – Configuration



Example

The target object is loaded only when it is first accessed.

10

The slide shows a general configuration in UML with a pseudo-code implementation showing as an example, the case where the loading of the object's state is delayed until first use.

The interface class `subject` defines the interface for *Proxy* objects as well as for the target object. This is the interface that affords transparency between a `concrete_subject` and a `proxy`.

In C++, one mechanism for implementing this transparency is using run time polymorphism (via inheritance). However, in C++ there are other mechanisms and the transparency is not necessarily a run time issue. For example, consider the smart pointers used for memory management and reference counting (e.g. `shared_ptr` found in the Boost library, see *Boost*); such smart pointers implement the interface `subject` available via the indirection operator (`operator->()`).

Design Patterns as Idioms

- Many idioms are other types of pattern in a form specific particular to a language
- For example, in C++
  - The *Whole Value* idiom is a C++ form of a pattern from problem domain modelling
  - Many types of *smart pointer* (those used to ensure safe resource acquisition/release) are language level forms of *Proxy*

11

Some programming language idioms are specialisations of (or are similar to language implementations of) design patterns or patterns applicable in other areas of the development process for that matter.

For example, in C++ the acquisition and release of resources (e.g. memory new and delete, file open and close) is complicated by the presence of exceptions in the language; the possibility of being interrupted by the propagation of an exception makes the flow of control less predictable than it might at first look. The normal idiom for ensuring resources are released on all control flows, is to access the resource via a *handle* – often in the form of a smart pointer. The handle takes the form of a class, objects of which are used "by value" (i.e. the object itself is used, not a pointer or reference to it), and which releases the resource in its destructor. The Boost memory management smart pointers (e.g. `scoped_ptr`, see *Boost*) are examples of handles supporting this idiom. Such handles are actually examples of the *Proxy* design pattern in a form particular to C++.

Whole Value – Idiom

- When built in types are used to represent domain types…
  - Compile time type checking is weakened
  - The domain vocabulary is absent from the code
- *Therefore*… create classes for domain types, so that…
  - Their use can be checked by the compiler
  - The code communicates using the vocabulary of to the domain

12

The Whole Value pattern originates in "The CHECKS Pattern Language of Information Integrity" by Ward Cunningham *PLoPD*. It is applicable as an idiom in C++ and other languages with direct support for user defined value based types.

There is a tendency when developing C++ software, for programmers to represent value based domain types using only the built in types. This is a traditional approach, used for many years in languages lacking support for user defined value based types (i.e. types, objects of which have state and identity indistinguishable from one another). This is not the case in C++, the language being designed to provide strong support for a variety of programming paradigms – including value based programming.

In C++, creating classes to represent domain types (e.g. currency, velocity) offers a better set of tradeoffs. The most obvious advantage is the type checking the compiler can do. Another compelling advantage is strengthened communication, because much of domain vocabulary is visible in the code itself, without recourse to separate documentation.

Naturally as always, the advantages must be traded against the disadvantages. The most obvious being management of the proliferation of small classes (the cost of producing them is another). However my experience has been that any disadvantages fade into insignificance compared to *just* the benefits of strengthened compile time type checking. The cost of errors that only manifest themselves at run time is notoriously unpredictable!

Whole Value – Example

```
enum tag_type { hour_tag, minute_tag, second_tag };

template <
  typename numeric_type, numeric_type first, numeric_type last, tag_type tag>
class numeric_range
{
public:
    explicit numeric_range(numeric_type n);
    // ...
};

typedef numeric_range<unsigned int, 0, 23, hour_tag>   hour;
typedef numeric_range<unsigned int, 0, 59, minute_tag> minute;
typedef numeric_range<unsigned int, 0, 59, second_tag> second;

class time_of_day
{
public:
    time_of_day(hour in_hour, minute in_minute, second in_second);
// ...
};

void f()
{
    time_of_day now(hour(14), minute(12), second(45));
//...
}
```

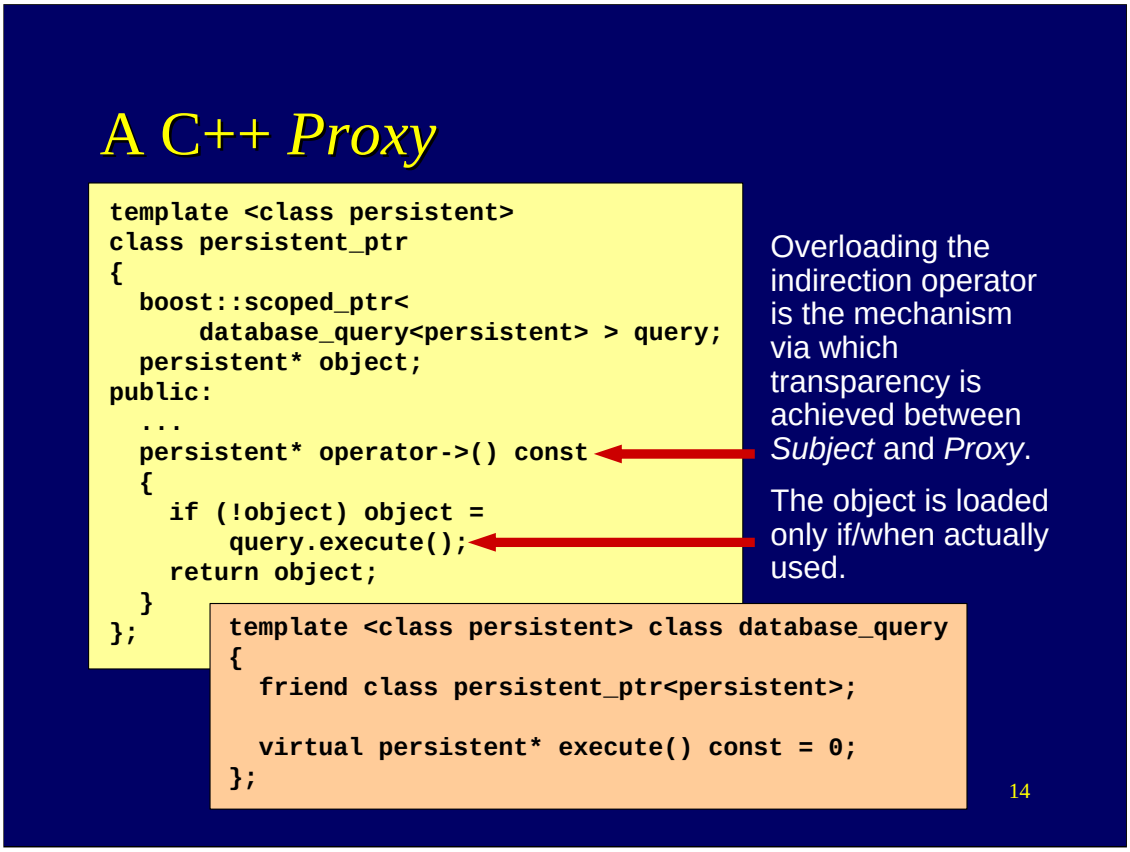
Compiler checks correct type use

13

Time of day is a typical example of a value. Also, the component parts – the hour, minute and second – are also examples of values.

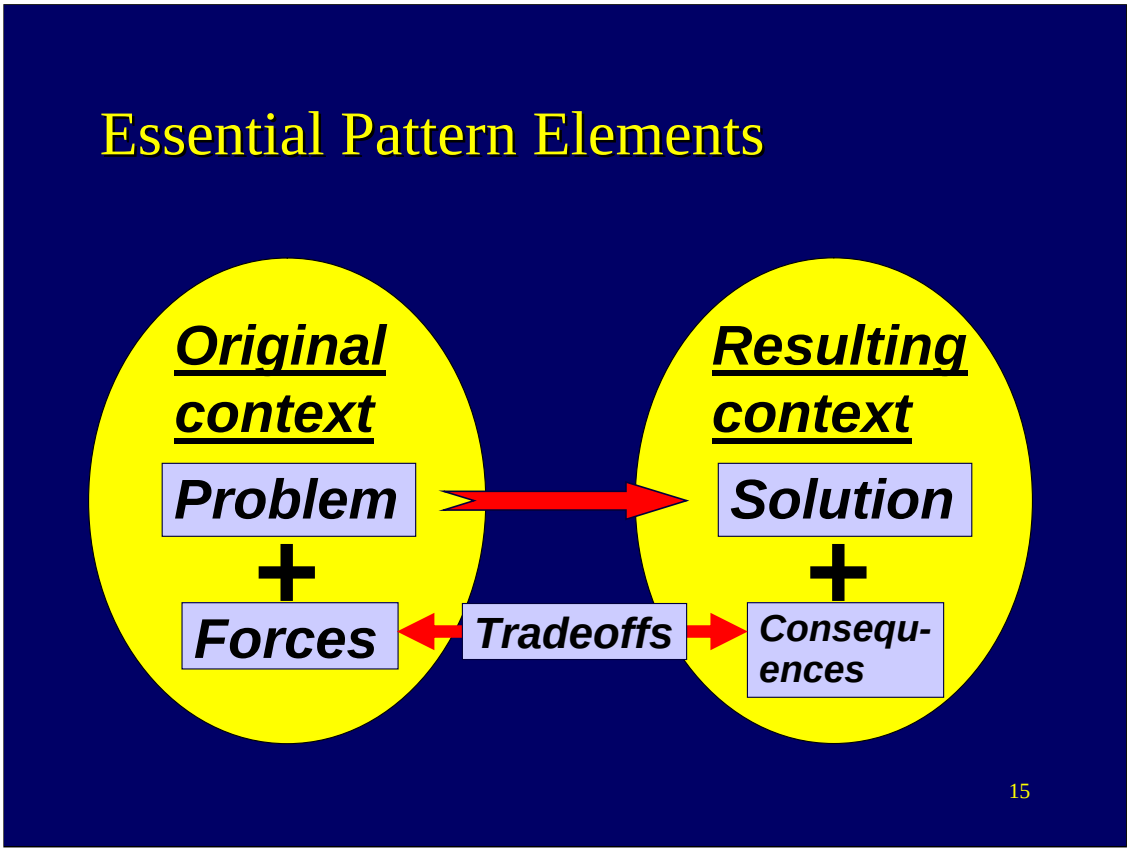
By creating the class template `numeric_range`, classes `hour`, `minute` and `second` can be generated fairly easily (note the tag template parameter, necessary because otherwise `minute` and `second` would not be distinct types).

Making these domain types into first class types has two benefits: first, the compiler is recruited to help check the correct construction of `time_of_day` instances, and second, the improved *communication* afforded also plays its part in promoting correctness.



When querying a database for the (saved) state of an object, it may be desirable to defer execution of the query until the object is actually used. Then, it is possible to follow control paths that do not actually use the object, without incurring the overhead of a database read. To achieve this, some housekeeping to keep track of whether the information has been retrieved or not is needed. In the design shown on the slide, this housekeeping is done by accessing the object via a “handle”, represented here in the form of a smart pointer called `persistent_ptr`.

This is an example of Proxy being used to defer the loading of information into an object until it is actually used, therefore avoiding an expensive operation unless it is actually required.



The *context* is the scenario or situation in which the problem arises, together with any factors that contribute to the problem's occurrence. Examples of contributors to the context are: the presence or absence of concurrent execution, whether all components are local or distributed, the need for an object oriented system to work with a legacy procedural system.

Forces are the influences that must be balanced in choosing a solution from the available options. A problem specified in a context may have more than one solution, and balancing the forces is part of determining which is the right one. The term *forces* is a metaphor: by analogy with forces in physics, forces in patterns resolve to steer the solution in a certain direction.

The *resulting context* is the new context brought about by the application of the pattern. The problem has been solved, but that's not the end of the story because there are *consequences* as a result. The consequences consist of both good news and bad. The bad news: new problems may have arisen as a result, and these will require solutions themselves. The good news: the problem is solved, the forces resolved (and there may also be beneficial side effects).

The resolution of particular forces versus the acceptance of particular consequences, is where the pattern captures a specific set of *tradeoffs*.

Pattern Exposition

- Essential elements are always present, explicitly or implicitly
- Problem and solution, plus some or all of the other elements, can be précised in a statement of *intent*
  - *Intent* conveys only a flavour of what the pattern is about

16

There are several pattern exposition forms in common use. Forms vary in how explicitly visible the essential elements are: for example, *forces* and *consequences* can, rather than be stated explicitly, be expressed in the more relaxed form of “pros and cons”.

The statement of *Intent* is a précise of the problem, solution, and any other of the elements that will help communicate, in a nutshell, what the pattern achieves. The statement of intent heads up, several exposition forms such as those used in *Design Patterns* and *POSA*, and is a useful starting point when deciding whether or not a particular pattern is applicable to a particular scenario.

The statement of *intent* cannot replace a more detailed exposition of any pattern; it just communicates enough about the pattern to serve as a point for getting started with that pattern (or for the selection of a pattern to use, from two or more possible candidates).

The "Gang of Four" Form

- Pattern is introduced in terms of intent and a motivating example
- *Problem*, *context* and *forces* are summarised in a statement of *applicability*
  - The view communicated is one of a solution and its applicability

17

This is the best known exposition form and was first introduced in *Design Patterns*.

The pattern is introduced with a short statement of intent, and this is followed by a motivating example. Configuration is expressed in a detailed manner: *Participants*, *Structure* and *Collaborations* sections describe the roles played in the pattern, how they fit together (statically) and how they work together (dynamically), respectively. *Consequences* is the only *essential element* to be made explicit. *Implementation* described hints, traps and pitfalls to be considered when implementing the pattern. *Sample Code*, *Known Uses* and *Related Patterns* are self-explanatory.

The strength of this form is that it makes pattern expositions very accessible to a large number of software developers. The high profile of aspects of the configuration together with sample code provide something they can immediately relate to. This is also a weakness, because it distracts attention from the the problem and tradeoffs that are part and parcel of any particular solution.

Other Popular Exposition Forms

- The *Coplien form* makes the elements explicit, adding *rationale*
- The *Alexandrian form* consists simply of two sections separated by the word "therefore"
  - The first combines *problem*, *context* and *forces*
  - The second combines *solution* and *resulting context*

18

The Coplien (see *Software Patterns*) form is perhaps the most rigorous because its focus is on the essential elements, these being made explicit as the section headings. A further section, *rationale*, provides supporting information: for example, history and other sources of information.

The Alexandrian form is the original form used in *A Pattern Language* and *The Timeless Way of Building*, and is much more relaxed than the Coplien form. It was originally intended to document patterns occurring in building architecture.

There is something that stands out that Coplied and Alexandrian forms both have in common: they both start by communicating understanding of the problem, and lead up to the solution.

Different exposition forms are useful in different contexts. For example, the limited space available on a slide points to the use of an abridged description in Alexandrian form, or just an elaborated statement of intent; in both these forms, the exposition benefits from being supported by a diagram or concrete example (whichever best serves the clarity and effectiveness of exposition).

Design & Lessons Learned

- *Singleton* misunderstood
- Emulating *multi-methods* in C++

19

Presented here, are stories of design episodes. Now unfortunately contractual obligations prohibit too much background about origins being given, and while permission could probably in some cases be obtained, the process would be lengthy and I can't see how being able to name the projects would add much value anyway. Suffice to say, the problems describes are real.

I've been working in software development since 1987, and have come across a variety of different design problems in that time. There are certainly many I can't remember, at least not very well. Here though, are a couple of the cases that stand out very clearly in my memory – perhaps they do so because they are all particularly instructive!

One of the reasons for picking these specific cases is this: in each one there is, with the benefit of hindsight and increased knowledge and experience, something more to add. In each case I have either done things differently, or would do things differently if repeating the exercise.

Singleton Misunderstood

- Contents…
  - The *Singleton* design pattern
  - An example of the misuse of *Singleton*
  - An argument that some of the common *Singleton* uses are not good ideas

20

Actually it is not just that the *Singleton* design pattern (see *Design Patterns*) has been misunderstood, but that it has proved something of a design red herring.

Many times I have implemented *Singleton* over the past several years, and now, I can't think of one that was actually a good solution to the problem it attempted to solve.

Here I will present just one very recent example, with an explanation of why it was the wrong approach. Then I want to look at some other commonly cited scenarios where employing *Singleton* is deemed to be a good solution; in these cases I will attempt to present and argument to the contrary, and also present alternative approaches I believe to be better.

The *Singleton* Design Pattern

- A class must only ever have a single instance
  - There must be a global point of access to that instance
- *Therefore*… define a static member function that returns the single instance
  - Make copy construction and default construction private/protected, etc…

21

The slide shows a brief exposition of *Singleton* (see *Design Patterns* for the seminal documentation), with a slight C++ slant to it.

The intent of *Singleton* is to facilitate the implementation of and provide a global point of access to, any problem domain abstraction for which it makes sense for only one instance to ever exist. One approach is to use a global variable, but that has two drawbacks:

(1) The instance must be created regardless of whether or not it is actually needed on the control flow being followed.

(2) No mechanism is put in place to ensure only one instance is ever created.

Singleton makes the class responsible for managing its sole instance. The use of a class operation – static member function in C++ terms – solves both the above problems: the class maintains its sole instance ensuring there is only ever one instance, and if the operation returning the instance is never accessed there is no need to create the instance this avoiding an unnecessary overhead.

Singleton Illustration

```
class singleton
{
public:
  static singleton& instance()
  {
    static singleton inst;
    return inst;
  }

protected:
  singleton();

private:
  singleton(const singleton&);
  singleton& operator=(
      const singleton&);
};
```

Static **`instance()`** member function allows clients to access the one and only instance.

Making default construction protected allows derivation.

Allowing any form of copying would undermine the uniqueness of the single instance.

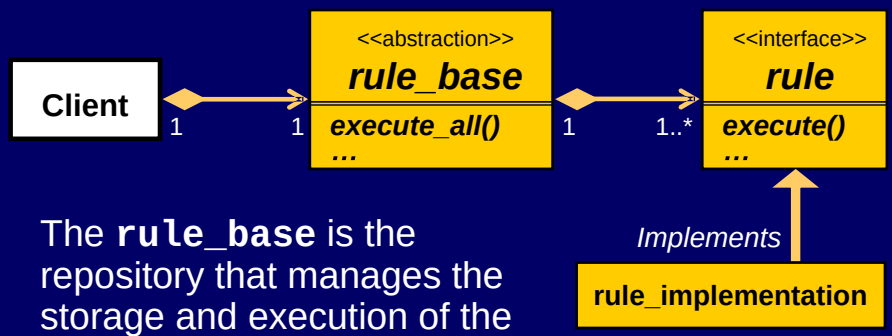
22

Illustrated on the slide is just one way to implement *Singleton* in C++.

To ensure only one instance can be created it is necessary for no constructors to be publicly accessible. Therefore the copy constructor is declared private. The copy assignment operator is also private – not strictly necessary (it is not possible to create an instance to assign to) but it brings a look of symmetry to the class design.

The provision of the static member function to return the sole instance is the most common way of enforcing the single instance feature of *Singleton* – the main area of variation in implementation is in the scheme used to create the instance. In many languages this is not an issue because only one memory allocation scheme is available, but in C++ there is the choice of static storage or allocating on the heap. The simplest method is to use static storage via the static variable in function scope approach shown by the slide: this approach takes advantage of the fact that such static objects are initialised the first time the control flow passes through the definition.

A Business Rules System



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

Because all the rules in the system are in the **`rule_base`**, the client only ever deals with one instance.

23

Business rules are applicable to and occur in a variety of domains and designs. Consider, for example, a security system for intruder detection: there are a variety of actions (e.g. sounds an alarm) that could be taken in response to a variety of potentially suspicious events (e.g. motion detected in an office at 3am). Suppose staff are working through the night to meet a deadline: the rules need to be changed so that motion in the office at 3am does not sound the alarm.

In the design on the slide, `rule` is the interface for objects encapsulating event/response *rules*. These are managed by `rule_base` – which not only maintains a repository of `rule` objects but also provides an interface for invoking their execution.

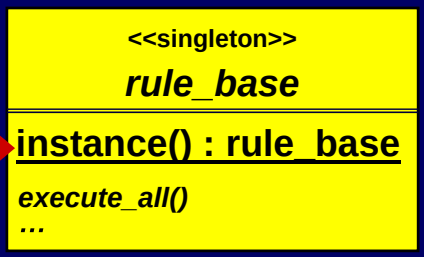
The the security system needs to keep all its rules in one place, and therefore there is only ever one `rule_base` instance.

Confusion Over Cardinality

It makes no sense for the client to use more than one instance of **`rule_base`**;

therefore,

make **`rule_base`** single instance only.



- The cardinality in the **`client`**/**`rule_base`** association should be controlled by the client!
  - No facet of **`rule_base`** requires it to be single instance only
  - Singleton has been used to put the solution to a cardinality problem in the wrong place

24

The reasoning behind making `rule_base` a Singleton was this: all the rule are to be kept in one place, and therefore only one instance will be needed by the client. Unfortunately this logic exemplifies a common misunderstanding – i.e. because only one instance is needed, it is a good idea to enforce this by making the class a *Singleton*.

The idea of *Singleton* is to limit instances to one, in cases where there can be only one instance – that is where, by virtue of the constraints of the domain, the type can physically only have one instance. In the case of the `rule_base`, the fact is that the client only requires one instance – a very different thing from only one instance being possible. There was nothing characteristic of `rule_base` requiring instances to be limited to one.

This misuse of *Singleton* exemplifies a common misunderstanding and misuse of patterns: the configuration (or just the sample code) seem to serve the purpose. However this is not the same thing as applying the pattern, at least not correctly, because it takes no account of what the problem to be solved really is.

Problems with *Singleton*

- Non-trivial initialisation is awkward to implement
  - How can arguments be passed on first use?
- Memory acquisition/release is inflexible
  - A Policy template parameter can not adequately address changing between heap allocation and function scope static
- Difficult to refactor if the design changes to require more than one instance

25

If there is only one instance of a *Singleton*, then it must be initialised only once, on or just before the first call to `instance()` (the static member function returning the instance). However, this is difficult to implement because `instance()` should not be burdened with parameters, as these would be redundant after the first call that actually instantiates the *Singleton*.

The memory acquisition and release problem is a problem is a particularly C++ problem (some languages have, Java for example, have only one means of allocating memory for user defined types). One attempt at solving this problem is to make the Singleton a class template with a memory acquisition/release policy (see *Modern C++ Design*) as a parameter. However although this can work, it is messy because the *Singleton* itself needs, in the case of heap allocation, to keep a member pointer to its instance that it can pass to the policy function that releases the memory. In the case of static storage, this member pointer will still be there but not used.

One advantage of *Singleton* cited in *Design Patterns* is that it is easy to to change the code if in the future the class needs to have more than one instance. This is unfortunately not the case because it relies on changing client code at every point where the instance is acquired (see *AF*).

Common Practice Questioned

- Purely behavioural (i.e. stateless) classes are often viewed as making good *Singletons*
  - For example, this approach is often used for the implementation of factory classes
- *But*, the single instance logic is just unnecessary extra baggage
  - Creating an instance as and when one is needed is both easy and efficient

26

A few years ago I worked on a project involving the design and implementation of a C++ framework supporting persistent objects. One feature of the design was a large number of factory classes, and these were implemented as singletons. At the time I didn't question this approach, but looking back I should have done. Each factory class had to carry extra machinery for the management of its single instance.

Contrast this with the overhead of creating and destroying instances at block scope. Constructing an instance of a stateless class is a fairly simple matter. If the class has no virtual functions, then it will be quite trivial. If there are virtual functions, then (in a typical implementation that implements virtual functions using a pointer table) it involves initialising a single pointer – the instance member (invisible to all but the compiler) pointing to the class' virtual function table; not a big overhead.

Instances of stateless classes do not have individual identity; therefore all instances are functionally the same – hence *Singleton* implementation. However, the point really, is that where an instance of such a class is needed, *any* instance can be used (there is no need to use the same instance everywhere one is needed).

Emulating *Multi-Methods* in C++

- Contents…
  - The *Extension Object* design pattern
  - An overview of the design of a mechanism for calculating intersections of geometric shapes, in 2D technical drawing software

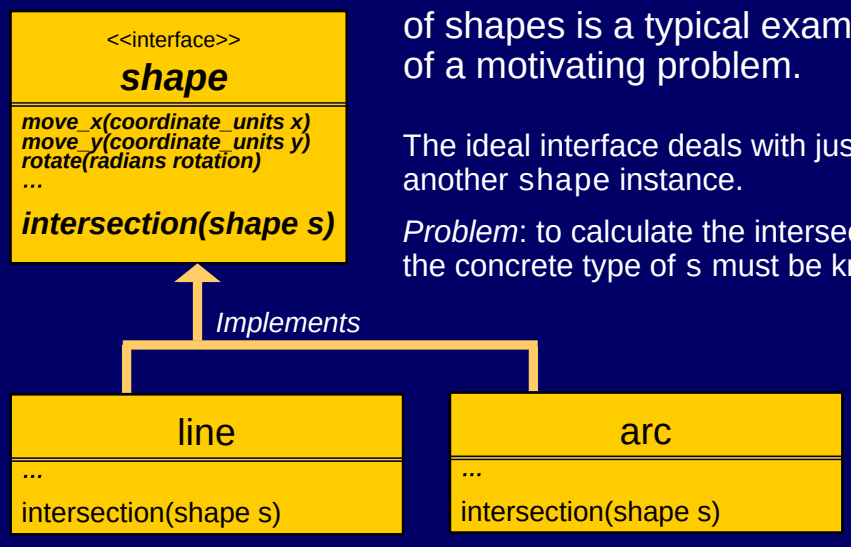
27

It was to my delight that I found Bjarne Stroustrup sites an example in [D&E] involving the intersections of shapes in a drawing program. I was pleased he used this example because a few years ago I was involved in the development of a package for producing two-dimensional technical drawings, and in the process faced exactly this problem. In a nutshell, the crux problem is this: when working out if/where shapes intersect, a `shape` abstraction is no good – it is necessary to know the `shape`'s concrete type.

The solution I came up with at the time was not very good. The irritating thing was that at the time I knew my solution was not very good – I just didn't know what else to do. I could think of other approaches, but they all seemed worse than the one I used. For example, some sources (e.g. *More Effective C++*) use the brute force approach of down casting in conjunction with RTTI; in hindsight though, the RTTI approach probably offered a better set of tradeoffs.

This problem has been in my mind (on and off) ever since. Years later, I have come up with what I think is a satisfactory approach.

A Motivating Problem



Calculation of the intersections of shapes is a typical example of a motivating problem.

The ideal interface deals with just another `shape` instance.

Problem: to calculate the intersection, the concrete type of `s` must be known.

28

The drawing program supported two basic shapes: straight lines, and semi-circular arcs. It is obvious that these shapes would need an interface capable of supporting the operations expected by the user, such as being able to move the shapes around and rotate them. Also, because the program was for producing drawings of a technical nature – essentially 2D CAD – an operation to calculate the intersection with another shape was also necessary.

Therefore the `intersection()` methods need to implement the mathematical formula for calculating the intersections. Unfortunately having available a `shape` abstraction is no good. The concrete type of both shapes is needed at the point where the calculation is implemented.

RTTI Solution

```
void intersection(
  const line& l,
  const shape& s,
  intersection_points& where)
{
  if (const line* lp =
        dynamic_cast<const line*>(&s))
  {
    lines_intersection(.., where);
  }
  else if (const arc* ap =
        dynamic_cast<const arc*>(&s))
  {
    line_arc_intersection(.., where);
  }
  else
    //..
}
```

```
void intersection(
  const arc& a,
  const shape& s,
  intersection_points& where)
{ /* .. */ }
```

This is a “brute force” approach of using **`dynamic_cast`** to test for each possible type.

Adding a new shape means adding a new intersection() function, and modifying all the existing ones.

29

This is the brute force solution, using down-casting to recover the concrete type of the object.

Consider the consequences of adding a new type of shape (e.g. an elliptical arc). This would mean two things:

(1) Adding a new `intersection()` function overload.

(2) Adding more code to the existing intersection functions. Further, it is necessary to replicate the type recovery control flow code in each `intersection()` function overload.

The above applies to using this approach with a current C++ compiler that implements `dynamic_cast<>` – a language feature not implemented in the compiler used on the 2D CAD project! Therefore, this approach would have required the manual implementation of some kind of RTTI substitute (e.g. each class having an integer constant to identify it).

A Flawed Object Oriented Solution

```
class shape
{public:

  virtual ~shape();

  virtual void intersection(
      const shape& s, intersection_points& where)
  const = 0;

  virtual void intersection(
      const line& s, intersection_points& where)
  const = 0;
//...
};
```

The shape class implements a *multiple dispatch* mechanism to resolve the concrete type, **but** …

The base (interface) class needs to know about its derived classes

```
class arc : public shape
{private:
 virtual void intersection(
     const shape& s, intersection_points& where);

 virtual void intersection(
    const line& s, intersection_points& where) const;
// ...
};
```

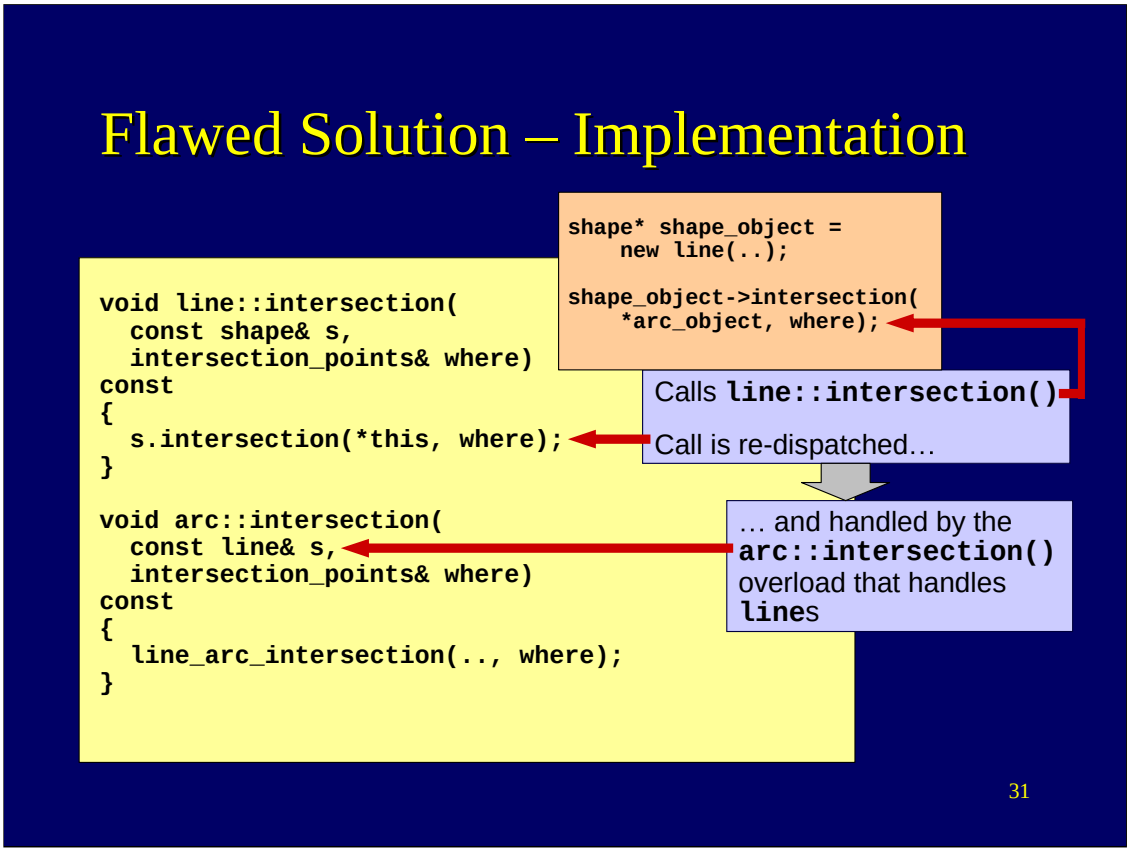
Also, derived classes need to know about each other

30

This is the solution I implemented at the time. It employs an object-oriented mechanism of type recovery using virtual functions. The mechanism takes advantage of the fact that the place where the concrete type of an object is known, is within the member functions.

The `shape` class is the interface class heading up the hierarchy. Note that it has a virtual function taking `shape` as a parameter, as well as one for each of `line` and `arc`; if another type of shape (e.g. an elliptical arc) were ever to be added to the hierarchy, `shape` would need a further virtual function taking the new type as a parameter, and derived classes would need to implement it. Therefore, this design is awkward to extend because it would require a change to code in many files implementing the `shape` hierarchy.

This solution is flawed. In a nutshell this is because of the intrusiveness of derived classes on each other, and on the base class. It must be remembered that calculating intersection points is only one aspect of `shape` functionality, yet providing it needs three virtual functions in the interface of each class in the hierarchy.



The slide shows what happens during an attempt to find the intersection of objects of type `line` and `arc` (if they intersect at all).

First, the call is made on an object of concrete type `line`, so the first virtual function implementation entered is `line::intersection(const shape&, ..)`. The important thing to note here is the type of the pointer returned by `this`: it is of type `line*` (rather than of type `shape*`).

Next, a call `s.intersection(*this, ..)` is made, and results in a call to the implementation of intersection taking a `line` as a parameter. Given that the pointer passed in pointed to an object of concrete type `arc`, the result is a call to `arc::intersection(const line&, ..)`. A point has been reached at which the concrete type of both objects is known.

Multi-Methods

- The **`intersection()`** functions emulate the behaviour of *multi-methods*
  - Multi-methods are functions that are virtual w.r.t. more than one object
  - They are supported directly in some languages, but not in C++
    - When required in C++, multi-methods must be emulated using design & programming techniques

32

Described previously is a mechanism effectively emulating functions that are virtual w.r.t. two objects rather than just one. Some languages (e.g. CLOS) allow such functions as a language feature. In general object-oriented parlance, C++ class member functions are called *methods*, and methods whose invocation is resolved on the concrete type of more than one object are commonly known as *multi-methods*. Where multi-methods are required in C++ they must be emulated using design and programming techniques.

Bjarne Stroustrup discusses multi-methods in *D&E*, noting that he considered multi-methods for inclusion in C++ although the feature never made it into the language (see *D&E* for the full discussion of why this is so).

The *Extension Object* Design Pattern

- Clients of an object (the *Subject*) may need interfaces that can not be anticipated at the time of designing the *Subject*
  - Interface bloat must be avoided
  - In C++, freestanding functions can not be polymorphic at run time
- *Therefore*…support these interfaces using separate objects – *Extension Objects*
  - Give the *Subject* an interface for returning *Extension Objects*

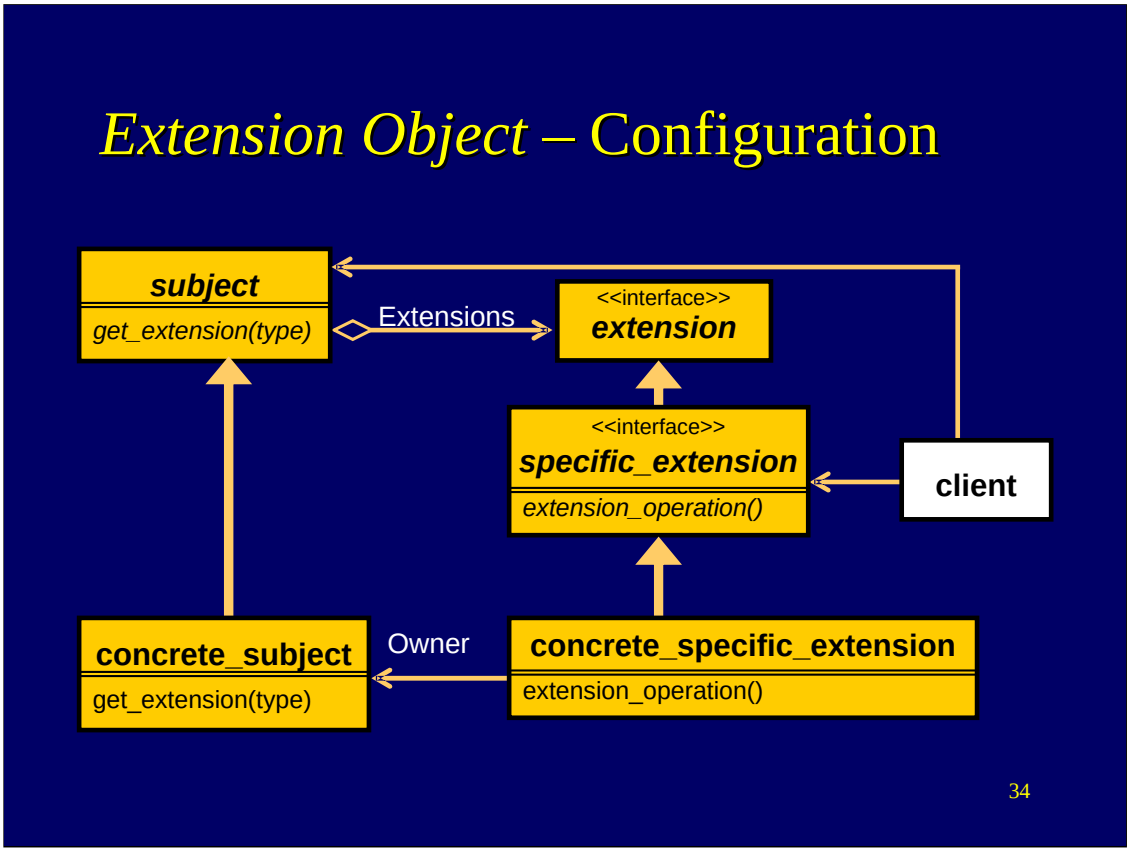
33

Extension Object is a design pattern originally documented by Erich Gamma (see *PLoPD3* for the full write-up).

Different clients will have different requirements of an object's interface. The precise interface that will be required by each client cannot always be anticipated at design time. In cases where it is possible to anticipate clients' requirements, it is often unacceptable to trade provision for them against the interface bloat that would result. The problem therefore, is how to allow clients the interfaces they require, but in a non-intrusive manner.

In C++ this problem can be addressed to some extent by an approach using freestanding functions. However this does not solve all the problems (for example, freestanding functions cannot be virtual).

Another approach is the *Extension Object* design pattern: interfaces required by clients are provided as separate classes, and are used at run time by creating instances of these classes.



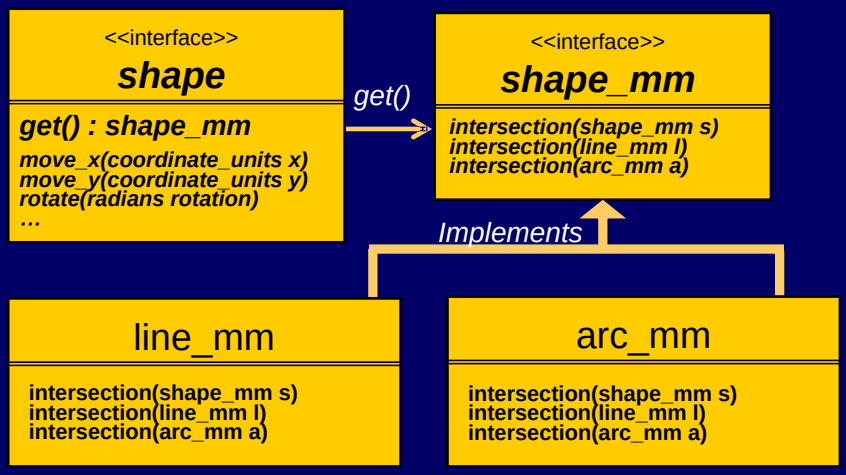
The extensions hierarchy is headed up by the `extension` interface, while the facilities the extension offers to clients are made available through the interface `specific_extension`. The `extension` type interface does not support the operations required by the client, because different extensions will offer different operations. `client` obtains access to extensions via `get_extension()`, to which it passes *type*, where *type* being simply some kind of indication of the extension type being requested.

It can be seen that this pattern offers benefits in terms of flexible extensibility, but there are some drawbacks, for example:

(1) Some of the behaviour of `subject` is moved out of it, so `subject` no longer expresses all the behaviour that clients can perceive it as having.

(2) The client code will need to recover the `specific_extension` type. A typical method of doing so in C++ is by using `dynamic_cast<>`. Therefore, clients become more complex in the face of the machinery needed to use the extensions.

Multi-Methods Using *Extension Objects*



This design does not follow the canonical *Extension Object* configuration to the letter – it promotes the `specific_extension` interface to the base of the hierarchy

35

The solution presented as a flawed object-oriented solution was in some ways an attractive one, exhibiting the benefits of object-oriented design, keeping code performing a function together and separate from code performing other functions. It was only flawed as a consequence of making classes within the `shape` hierarchy intrusive on each other, and the interface clutter caused (three virtual functions were needed in each class' interface). Introducing the *Extension Object* design pattern allows the same mechanisms to be deployed while keeping the intrusiveness and interface clutter out of the `shape` hierarchy.

The design shown does not follow the canonical configuration exactly. The `extension` interface is removed and the `specific_extension` interface elevated to the top of the hierarchy. The `shape_mm` interface corresponds to `specific_extension`. This simplification trades flexibility for simplification – it is no longer necessary to recover the `specific_extension` type.

A Better Object Oriented Solution (?)

```
class shape_mm
{
public:
  virtual ~shape_mm();

  typedef boost::shared_ptr<shape_mm> shared_ptr;

  virtual void intersection(
    const shape_mm& obj, intersection_points& where)
  const = 0;

  virtual void intersection(
    const line_mm& obj, intersection_points& where)
  const = 0;

// ...
};
```

```
class shape
{
public:
  virtual shape_mm::shared_ptr create() const = 0;
// ...
};
```

36

The mechanics of recovering the types and working out the intersection points are the same as in the flawed solution – the only difference is that this time the participants are `shape_mm`, `arc_mm` and `line_mm`.

The `shape` interface class provides a factory member function `create()` that returns an instance of the `shape_mm` instance. The canonical configuration designates `concrete_subject` as the owner of the *Extension Object*, and here this is implemented using the C++ idiom of using a smart pointer to manage memory acquisition and release.

Using *Extension Object* – In Favour

- Multi-method emulation and intersection logic are non-intrusive w.r.t. `shape` hierarchy
  - For example: if another shape is added, only the classes in the multi-methods hierarchy are affected, whereas…
    - In the "Flawed OO Solution", `shape` hierarchy definitions must be changed
    - In the "RTTI Solution" a new **`intersection()`** function is needed and all **`intersection()`** functions need extending

37

In the case of the flawed object oriented solution, the problem was that derived classes were intrusive on the base class, and on each other. In the case of the solution that uses *Extension Objects*, classes derived from `shape_mm` are also intrusive on each other, but there is a very important difference: there is no intrusiveness on the `shape` hierarchy.

Note that, in the case of the example of adding another type of shape (an elliptical arc for example), the bodies of existing `shape_mm` member functions will not need their implementations changing. This is a consequence of virtual functions being used to automate the control flow by placing it in the hands of the C++ language. By contrast, in the case of the RTTI solution, the control flow is implemented directly in the code, and as a consequence adding the code for a new type of shape means modifying existing code. In the former case, the absence of a need to change existing code means that the chance of introducing an error into it is reduced.

Using *Extension Object* – Against

- There is added complication in logic being distributed across two class hierarchies
  - There are more types in the design
  - The `shape` no longer communicates any explicit reference to intersections in its interface
    - A direct consequence of the *Extension Object* pattern

38

The `shape` and `shape_mm` hierarchies have parallel corresponding classes. Working with and maintaining such parallel hierarchies always creates a balancing act of design.

The most obvious burden is the extra types that now inhabit the design, and these must be managed – not just in physical terms but in addressing the communication issues that arise (more documentation will be needed).

More subtle is the lack of any direct mention of intersections in the `shape` interface, and in the interfaces of classes derived from it. Here, a consequence associated with applying the *Extension Object* design pattern haunts the design.

Lessons Learned

- Using the object oriented paradigm does not automatically make a design superior
  - The “Flawed OO Solution” was object oriented but demonstrably poor
- Good OO design has benefits but may also have costs
  - The solution using *Extension Object* has demonstrable benefit (e.g. in extensibility) but also at demonstrable cost

39

In the past object orientation has been adopted in the hope that it would be the silver bullet that would solve all software development problems. Of course, history now records that nothing was further from the truth. There were many factors involved, one being the lack of understanding of object orientation itself. Another critical factor however, was the assumption that being object oriented automatically made a design a good one. The flawed object oriented solution presented earlier is an excellent counter example.

An important lesson is that even good OOD has its costs. It comes back to the fact that when solving problems with any level of complexity, there is no such thing as a solution per-se – there are options and tradeoffs.

Final Remarks

- The concept of a *pattern* in software is hard to define
  - A *slippery nail*, that's difficult to hit on the head!
- Patterns are many and varied
  - The "Gang of Four" book is just a small sample
- Revisiting past design work advances understanding

The End

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

References

AF, Thanks to Adrian Fagg for pointing this out (public house communication, 2002).

Analysis Patterns, Martin Fowler, *Analysis Patterns: Reusable Object Models*, Addison Wesley

A Pattern Language, Christopher Alexander, Sara Ishikawa and Murray Silverstein with Max Jacobson, Ingrid Fiksdahl-King and Shlomo Angel, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

The Timeless Way of Building, Christopher Alexander, *The Timeless Way of Building*, Oxford University Press, 1979.

Boost, See www.boost.org

D&E, Bjarne Stroustrup, The Design and Evolution of C++, Addison-Wesley, 1994

Design Patterns, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

Indexing STL, Silas S. Brown, *Indexing STL Lists with Maps*, Overload 53.

Modern C++ Design, Andrei Alexandrescu, Modern C++ Design: Applied Generic Programming and Design Patterns (C++ In-depth Series), Addison-Wesley, 2001

More Effective C++, Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.

References II

PLoPD, Edited by James O Coplien and Douglas C Schmidt, Pattern *Languages of Program Design*, Addison-Wesley, 1995.

PLoPD2, Edited by John Vlissides, James O Coplien and Norman L Kerth, *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.

PLoPD3, Edited by Robert Martin, Dirk Riehle and Frank Buschmann, *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

PLoPD4, Edited by Brian Foote, Neil Harrison, Hans Rohnert, *Pattern Languages of Program Design 4*, Addison-Wesley, 1999

POSA, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Pattern-oriented Software Architecture John Wiley 1996

POSA2, Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, *Pattern-oriented Software Architecture Vol 2: Patterns for Concurrent and Networked Objects*, John Wiley and Sons Ltd

Portland Pattern Repository, The Portland Pattern Repository (http://c2.com/ppr).

Software Patterns, James O Coplien, *Software Patterns*, SIGS, 1996. (Available from http://www.bell-labs.com/user/cope/Patterns/WhitePaper).