Design Experiences in C++

Generative Programming Goes Live!

Mark Radford
twoNine
Computer Services Limited

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

1

ACCU Spring Conference, April 2005

Mark Radford

twoNine Computer Services Ltd

mark.radford@twonine.co.uk

www.twonine.co.uk

Modification History

8th April 2005 – Slides submitted for inclusion in conference CD

12th April 2005 – Notes added. Correction to “Adding in the Pre-Processor” slide (obj corrected to obj.as_represented()).

3rd May 2005 – Notes updated after conference presentation.

Preface

- Topics covered
  - The Generative Programming paradigm
  - Discussion and examples of its implementation using C++
- Topic specifically *not* covered
  - Testing

2

In this presentation I will include material to do with Generative Programming, the paradigm and its implementation. Sorry, but I will not be including material on testing in this context – it would take far more time than is available. I may at some point include such material in a future presentation.

Having said that, I will offer some brief observations before moving on…

Generative Programming involves producing a number of related software products – i.e. a family of them – using an automated process. Therefore the emphasis shifts from testing the software produced, to testing the process producing it. If the production process is right, the end product will be right (in the same way as a working compiler produces correct object code for the source code fed into it).

A possibility that turns out to not quite work is generating test software as part of the process. This is actually somewhat pointless – if the process can be trusted to produce correct test software, it can be trusted to produce a correct product in the first place.

Contents

- Introduction to *Generative Programming*
- Case study 1
  - Software for a VCR range
- Tools and mechanisms for generation
- Case study 2
  - Value based domain types

3

This presentation is about my experiences of trying to understand this material, and my experiences of it in practice. To this end, the above four sections fall into two broader ones.

The *introduction to Generative Programming* and *case study 1* form the first part, while *tools and mechanisms for generation* and *case study 2* form the second. The former two are the learning experiences, while the latter two are the practical experiences.

Introduction to *Generative Programming*

- Purpose
  - Introduce Generative Programming
- Contents
  - Definition
  - Process anatomy
  - Requirements, components and configuration
  - Examples of software system families

4

In the C++ community, Generative Programming is not a mainstream software development paradigm. I can see why it is not mainstream, but I'm rather baffled as to why it seems to be absent from popular literature altogether. A certain amount of material focusing on the related topic of C++ template meta-programming has appeared ([Alexandrescu2001] and [Abrahams2004] for example), but not on Generative Programming itself.

This section aims to introduce Generative Programming, taking a close look at the seminal definition from [Czarnecki2000].

Definition

"A software development paradigm based on modelling software system *families* such that, given a particular *requirements* specification, a highly customised and optimised *intermediate* or *end*-product can be *automatically manufactured* on demand from elementary, reusable implementation *components* by means of *configuration* knowledge. The generated products may also contain non-software artefacts, such as test plans, manuals, tutorials, maintenance and troubleshooting guidelines, and so on."

(My *emphasis*)

[Krzysztof Czarnecki, Ulrich Eisenecker, "Generative Programming: Methods, Techniques and Applications", Addison Wesley]

5

This is the definition given in Czarnecki and Eisenecker's seminal book [Czarnecki2000] on Generative Programming. I have added my own emphasis to draw attention to the key participants.

Note that products may be end or intermediate. I have taken intermediate products to be libraries that will be used as part of an end product development.

Requirements, components and configuration

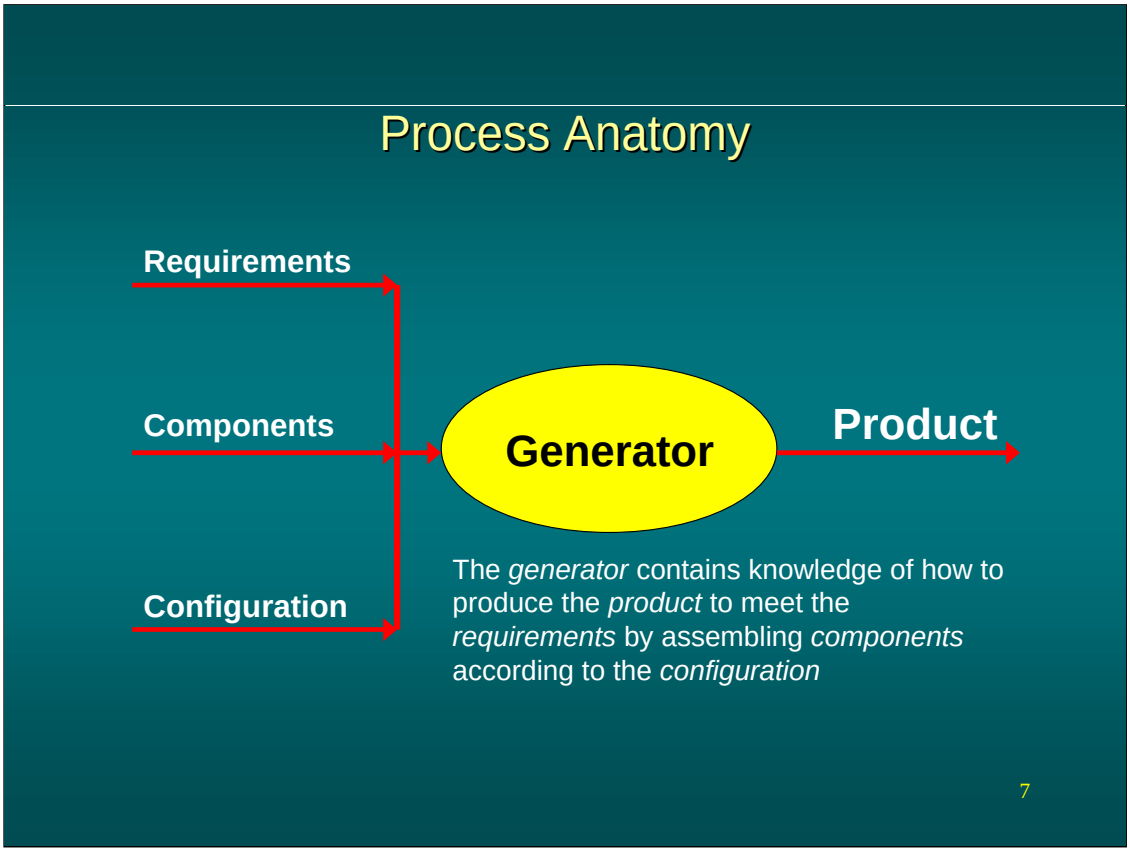
- Requirements
  - Describe the product to be generated
- Components
  - Encapsulate functionality from which a product can be assembled
- Configuration
  - Describes how to assemble components such that the resulting product meets the requirements

6

Requirements describe the product to be generated.

The product is assembled from *components* that encapsulate the functionality form which the product can be assembled. There are two levels of component: the elementary implementation components, and the components these are used to generate. The elementary components are build on to produce more specific components, and so on, until the end product (rather than a component) of it emerges.

The Generative Programming process can be likened to an assembly line such as those used for manufacturing cars. Components can be changed, leading to a different product coming out at the end. Similarly in Generative Programming, the *configuration* is a specification of the knowledge of what components to include, and how to assemble them. The configuration can be viewed as an implementation of the requirements, in the context of the Generative Programming process – it determines which product in the family is actually produced.



The slide is a pictorial representation of the Generative Programming process. The *requirements*, *components* and *configuration* are all inputs to the *generator*, while the *product* is its output – i.e. inputs are fed into the generator, and a product comes out the other side.

Note that here the components are the elementary components that form the basis for all software to be generated. In this representation, I have taken *product* to refer to all output from the *generator* – i.e. product includes components that have been generated as part of the process.

Examples of Software System Families

- Complete banking systems
  - Customised to for each hardware and operating system combination
- Container libraries
  - Versions supporting alternative tradeoffs can be generated, e.g.
    - One supporting maximum run speed
    - One supporting minimum memory consumption

8

Banking systems are examples of end products.

The slide mentions customisation of hardware and operating system combinations, but this is only one possible area of customisation. Another such area of variability the is the repositories in which reports are to be stored. For example, if reports are to be stored in a database, the system will need the correct code to interface with database used by the bank.

Container libraries are an example of intermediate products – i.e. products to be used as part of a larger software system.

Note the vagueness in the tradeoff mentioned on the slide – i.e. that of run speed versus memory consumption. The point is that optimising for speed is itself a whole area of potential customisation! For example, for sequenced containers, there are indexing schemes that (assuming lookup starts at the beginning of the container) would improve search lookup speed for elements further away from the starting point. Other schemes are possible, e.g. those involving hashes, and these have their own tradeoffs.

Case Study 1: Software for a VCR Range

- Purpose
  - Illustrate Generative Programming process by describing the production of onboard software for a range of Video Cassette Recorder (VCR) machines
- Contents
  - Driving the process using templates
  - Using the compiler as a generation tool

9

This example focuses on using the C++ compiler as a generator, and using C++ templates to specify the requirements and configuration.

Before going any further, let me make something clear: this is a *contrived* example, not one from a real world project. Therefore you may ask, what is it doing in a design *experiences* talk? The answer is that it has played a role in the development of my understanding of this material.

I first read Czarnecki and Eisenecker's seminal book on Generative Programming [Czarnecki2000] nearly five years ago at the time of writing. After reading the book, I formulated this example in an attempt to help myself understand the material. I have periodically revisited it in the intervening years, and in particular, have made several updates to it while preparing this presentation.

Note that the example is intended to show only how the software can be produced using Generative Programming. To this end to keep the illustrations simple, certain details such as some of the parameters that would need passing have been ignored.

(The example probably needs updating to use something other than video recorders, as it seems these are becoming increasingly rare these days).

VCR Machine Features

- Some of the features a VCR machine may have are:
  - Record, play, fast forward, rewind etc (basic features)
  - Extra fast winding (forward and rewind)
  - Digital stereo
  - PDC (Programme Delivery Centre) signal handling
  - Automatic channel tuning

10

I am assuming that the selection of possible VCR features listed on the slide will not need any explanation. One would expect any VCR machine, even the most basic model, to have record, play, rewind and fast forward. Other features may or may not be present depending on the model.

Different VCR models have different sets of features. Obviously, each onboard software product will need to contain code to handle the features of the particular model for which it is intended. For the purposes of this example, I am assuming that the software should contain only the code to handle the features of the model for which it is intended.

Therefore, for a range of VCR models, a family of onboard software products is required.

In these days of cheap memory, the reality is that one software product would be produced with all features catered for. However, this does not detract from this case study's illustrative value.

VCR Machine Range

- Consider three models and their associated features
  - *Basic* – all the usual features, such as record, play, fast forward and rewind
  - *Deluxe* – in addition, has extra-fast forward & rewind
  - *Super Deluxe* – in addition, has facilities to use the Programme Delivery Centre (PDC) signal

11

This example continues with a range consisting of three VCR models, as shown in the slide. Other models may be added, and these will have their own combinations of features (taken from the list on the previous slide).

For the purpose of this example, the three models listed on the slide are the three in the range at the time of its launch. The range may have models added and removed during its commercial lifetime.

VCR Onboard Software

- For any particular model, the onboard software should contain only the code needed to handle the features the model has
  - It must be possible to generate onboard software to accommodate new combinations of features when new models are brought out

12

It may be necessary to put together the onboard software for a new model in a very short time (perhaps in order to compete in the market, in response to a new model being brought out by a rival manufacturer). In this circumstance, there is an obvious advantage to being able to produce the software simply by running the build process, having set up the desired (parameterised) configuration.

In the following slides, I will show the implementation of only PDC signal and extra fast winding features – that's all there is space for on a slide. However, all features are implemented in a similar way.

Features as Policy Classes

```
enum pdc_states { pdc_on, pdc_off };

template <pdc_states state>
    struct pdc_feature;

template <> struct pdc_feature<pdc_on>
{
  static void handle_pdc()
  { ... code ... }
};

template <> struct pdc_feature<pdc_off>
{
};
```

`pdc_on` and `pdc_off` flag the presence or absence (respectively) of PDC signal handling

Handler function is declared and implemented for `pdc_on`

Handler function is not needed for `pdc_off`

Note: the extra-fast winding policy classes are implemented similarly

13

Policy classes [Alexandrescu2001] use templates to capture specific pieces of functionality, and allow one implementation to be exchanged for another.

The slide shows the implementation of the PDC signal handling feature using a policy class. The code implementing the feature is encapsulated in a class (or rather, a struct, to be precise) that contains code only for the implementation of the PDC feature. The presence or absence of the feature is flagged by the `enums` `pdc_on` and `pdc_off`, respectively.

There is no implementation of the generalised class template – just a declaration to meet the needs of the compiler. There are two specialisations: one containing code, and one empty, for when the PDC feature is present and absent, respectively.

Having an empty struct for the `pdc_feature<pdc_off>` specialisation is one approach and perhaps too minimalist. The `handle_pdc()` member function could be declared but not implemented. More about this in a later slide.

Configuration Templates

```
enum vcr_models { basic, deluxe, super_deluxe };

template <pdc_states pdc_flag,
          extra_fast_winding_states extra_fast_winding_flag,
          ... >
struct vcr_feature_configuration
{ static const pdc_states pdc_state = pdc_flag;

  static const extra_fast_winding_states extra_fast_winding_state =
    extra_fast_winding_flag;
  ...
};

template <vcr_models model> struct vcr_model_configuration;

template <> struct vcr_model_configuration<deluxe> :
  public vcr_feature_configuration <pdc_off, extra_fast_winding_on, ...>
{};
...
```

Configuration Templates specify whether a feature is enabled or not for a particular model

14

The slide shows how the configuration is implemented as C++ class templates. The configuration templates are class templates containing the configuration knowledge enabling the desired product to be generated.

There are two stages of configuration: *feature* configuration *model* configuration. The former is represented as a general class template, while the latter is represented as an explicit specialisation for the particular model. When a new model is added to the range, a model configuration template specialisation must be written for it.

The mechanisms are as follows…

First `enum`s are declared to denote the models to be supported (these must be updated when models are added to or withdrawn from the range).

Next a class template is defined to contain the feature configuration knowledge. It has a template parameter for each feature, to denote whether the feature is present or absent. In specialisations of this class, a member constant is initialised with the value of each template *argument* (there is a member constant for each feature, although only one is shown in the fragment on the slide).

Finally, a class template is declared containing knowledge of the feature configuration for the model. This is simply an empty definition derived from a specialisation of the feature configuration class template – the latter being specialised using arguments for the desired set of features.

Configuration Templates in Action

```
template <vcr_models model> struct vcr_onboard_software
{
  typedef vcr_model_configuration<model> config;

  static void handle_pdc()
  {
    pdc_feature<config::pdc_state>::handle_pdc();
  }

  static void handle_extra_fast_rewind()
  {
    extra_fast_winding_feature<
      config::extra_fast_winding_state>::handle_rewind();
  }
  ...
};
```

```
typedef vcr_onboard_software<deluxe> sw;
```

Here, configuration templates determine the code selected for inclusion when the onboard software is generated

15

This slide shows the apparatus from which the (onboard software) product is actually generated – the product is implemented as a single class template containing static member functions for the various features. The product is generated by explicitly specialising this class template for a particular VCR model, using the `enum`s denoting the models (see previous slide and notes).

Note, given that this is the *deluxe* model, which does not have the PDC signal handling feature. Therefore, in the `handle_pdc()` function the `pdc_feature` policy class resolves to the empty `pdc_feature<pdc_off>` specialisation. Given that the model has no PDC signal handling, this the onboard software's `handle_pdc()` member function is never called, and as the class of which it is a member is a class template, it is therefore never instantiated. This means that when the onboard software class template is instantiated, the code containing the call to pdc_feature::handle_pdc() is not instantiated, so no error will result – assuming all is according to plan, and the possibility this error could occur (in the event of a configuration error) has diagnostic benefit.

If it was necessary to explicitly instantiate the class template or the `handle_pdc()` member function, then it would be necessary to declare (but not necessarily implement) all member functions of component policy classes.

Case Study – End Note

- Templates arguments were used to specify the feature set for a particular model
- Software for a model supporting any combination of features could be generated
  - Assuming the availability of suitable components (policy classes)

16

The illustrations in the previous few slides have shown how the (onboard software) product can be generated, but these illustrations have only shown the implementation of a couple of features. The illustration of the software product for the *deluxe* model used the extra fast winding features and PDC signal handling feature, showing how the former was included, while the latter was not.

The apparatus and techniques shown could be applied to any combination of features – assuming the availability of policy classes for each feature, the necessary configuration templates (supporting the desired feature set) just need to be written.

Therefore, once the necessary configuration templates have been written, generating the onboard software for a new VCR model is just a matter of running the build process.

Tools and Mechanisms for Generation

- Purpose
  - Present an overview of tools and mechanisms for generating C++ source code
- Contents
  - Templates
  - The pre-processor
  - External programs

17

This series of slides presents a tools and mechanisms overview. On a pedantic note: the distinction between *tools* and *mechanisms* is rather burred in this context, and I have not paid much attention to making any distinction.

Generating C++ source code is a natural approach to implementing Generative Programming in C++. To this end, there is no shortage of mechanisms – two of them (Templates and the pre-processor) provided by the C++ implementation. Note that regarding templates, the generation of C++ source code takes a certain *perspective*, which hopefully will become clear over the next few slides.

I have used the term “external program” to refer to programs outside the translation phases of C++. External programs include utilities such as Make, as well as custom written generators. Having said that, when using Make (or similar) a custom program will be needed to do the generation at some point. The generator can be either one program, or more than one working together.

Templates

- The C++ compiler is a C++ code generator
  - Templates are the input language
  - The C++ generated during instantiation is the "object code"
- Templates are popular mechanism for *meta-programming* in C++
  - Meta-programs manipulate other programs or themselves

18

Since the addition of templates to C++, the compiler has itself been a C++ code generator. Note that Czarnecki and Eisenecker (in [Czarnecki2000], section 10.5) point out that C++ at the static level is Turing complete – that is, it supports conditional and looping constructs (with looping constructs emulated by recursion).

Templates are an input language, and the "object code" from the template instantiation process is C++. Obviously this is not what happens in practice (the compiler instantiates templates to its own internal format). However this perspective is valid, because when a template is instantiated, the programmer effectively has use of a class in the same way they would if a non-template had been written.

Meta programs are programs that manipulate themselves or other programs. In C++ Generative Programming, template meta-programming is a powerful device for driving the configuration of components.

Issues with Templates

- Experience with templates has shown they can achieve far more than they were designed for
  - A high degree of syntactic complexity is a price that must be paid for pushing the boundaries
  - Programmers who are comfortable with advanced template techniques are rare
  - In practice, error messages are often cryptic

19

C++ template programming techniques have become ever more sophisticated since the ratification of the language standard [ISO1998] in 1998. The examples in [Alexandrescu2001] and [Czarnecki2000] demonstrate just how powerful and versatile C++ templates really are.

However, sadly the news isn't all good. I would like to consider three issues with templates…

1. Sophisticated use leads to much syntactic complexity, largely as a consequence of templates being pushed far beyond the limits their designers envisaged. The result is that typically, template meta-programming produces C++ code that many consider to be cryptic (but your experience may be different).

2. Templates are seen as an advanced language feature, and there are not many programmer around who are comfortable writing them (although more appear to be happy to *use* template libraries such as STL). Programmers who are comfortable with "advanced" template techniques are quite rare.

3. Error message resulting from templates (and their use) are often cryptic. Several years after the ratification of the C++ standard, there are popular commercial compilers that still have a long way to go in this area.

The C++ Pre-Processor

- Compared with templates, the pre-processor has advantages
  - For example, in certain cases where its token pasting capabilities are useful
  - Its output is available for viewing, whereas template instantiations are private to the compiler
- However, cryptic error messages are still a problem

20

The pre-processor is much maligned in C++. This is largely because of its traditional use – e.g. in C, for defining static constants – is error prone, while C++ supports compile time constants.

In my view the C++ pre-processor is still a perfectly valid tool – it just has a different role to play. The level of indirection it introduces (between the code that is processed by the pre-processor and the code that is compiled) opens up possibilities that can be exploited when generating code.

Compared with templates…

•The pre-processor has facilities for converting arguments to strings, and for token pasting. This makes possible, techniques that are either difficult or impossible to achieve using templates.

•The output from templates is private to the compiler. In many (most?) C++ implementations, a switch is available that causes the pre-processor output to be written to a file.

Having said that there is a downside. The pre-processor does produce cryptic error messages, as a result of the compiled code having been modified by the pre-processor from the original source written by the programmer. However, while quality of implementation varies, there are template implementations in current popular compilers that also produce highly cryptic error messages.

External Programs

- From the configuration point of view, these afford the most flexibility
- The C++ generated is available for viewing
  - Template instantiations are private to the compiler
- An extra stage must be managed in the build process

21

These can be custom written for the purpose, or can be existing utilities such as *Make*. However, utilities such as Make are not useful on their own – they must be used in conjunction with either the template/pre-processor facilities of the C++ implementation, or in conjunction with a custom written program. The first two observations on the slide assume a custom program is involved somewhere in the generation mechanism.

Naturally the involvement of custom written programs affords the most flexibility – not surprising given that they are written for the purpose. Note that such programs *can* have the elementary implementation components hard coded into them – leaving less artefacts to be managed.

The code produced by custom generators is available for human viewing (always useful) – compare with template instantiation which are private to the compiler. Further, note that the code produced will also be correct! Once the generator has been developed, there are no compilation errors resulting from the generated C++.

Having said all that, there are disadvantages. There is extra effort involved in producing a custom generator program, although this effort is a predictable overhead. Further, there is extra effort in managing the extra stage in the build process. In many projects, the advantages described above may not outweigh the disadvantages.

Case Study 2: Value Based Domain Types

- Purpose
  - Explore Generative Programming approaches to the production of *Whole Value* families in C++
- Contents
  - Introduction to the Whole Value idiom
  - Interface requirements
  - Approaches to specification and generation

22

Each application domain uses value based information, and each domain has a family of value based concepts associated with it. In C++ these value based concepts translate into value based *types*.

Some types are more general (measurements such as time in *seconds*, for example) while others are more specific (for example, a motor vehicle's *vehicle identification number*). Whole Value classes are a means of implementing such domain types in C++.

For small Whole Value libraries, typically template/pre-processor techniques will be appropriate – certainly these are the techniques I have found myself using most often. However, for larger projects and hence larger libraries, the case custom written programs as generators becomes much stronger.

The *Whole Value* pattern originates in "The CHECKS Pattern Language of Information Integrity" by Ward Cunningham (see [Coplien1995]), a pattern language that drew on its author's experience of implementing financial systems in Smalltalk. It is so called because it addresses the need to capture all the facets of a value – i.e. the need to retain its type and units, for example. *Whole Value* is applicable as an idiom in C++ and other languages with direct support for user defined value based types.

The Problem

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

23

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.

The price that must be paid for the consequent lack of strong type checking, is the expenditure of resource in dealing with bugs that result. From a project management perspective, the issue here, is that the amount of work involved is highly unpredictable.

The *Whole Value* Idiom -- A C++ Solution

- Representing domain value types as classes…
  - Empowers the compiler to detect type mismatch errors
  - Raises the code's level of self-documentation

```
void f()
{
  time_of_day now(
      hours(14),
      minutes(12),
      seconds(45));
  ...
}
```

- The compiler checks for correct type matching
- The code speaks clearly in the vocabulary of the domain
  - Comments are not needed

24

In C++, creating classes to represent domain types (e.g. GP Pounds, metres per second) 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 costs. The most obvious are the cost of producing, and the cost of managing the proliferation of, small classes. However my experience has been that any disadvantages fade into insignificance compared to *just* the benefits of strengthened compile time type checking. From the project management perspective, the extra work involved is *predictable*.

Note that conversion constructors should be `explicit`. The integer value 12 is not the same piece of information as 12 minutes. The code must represent all the facets of the value.

Automation for Cost Mitigation

- Implementing the *Whole Value* idiom requires the production of many small and similar classes
  - The effort required to produce and manage these adds cost to the project
- Automation helps to mitigate the project risk
  - It reduces variability in the cost of producing the classes

25

Whole Value classes normally don't have very much individual functionality. From a certain perspective, they are largely the same class repeated many times with a different name. Their generation using automated techniques must therefore be on the agenda.

Even if the benefit outweighs the cost, the production of Whole Value classes – especially large quantities of them – adds cost that must be included in the project management. Automation helps mitigate that cost. Again *predictability* is the project management watchword. If the production can be successfully automated, then the cost is the same regardless of whether ten or a thousand such classes are to be produced. Note that this assertion assumes the cost of computational resource is negligible compared to the cost of programmer resource.

Operations

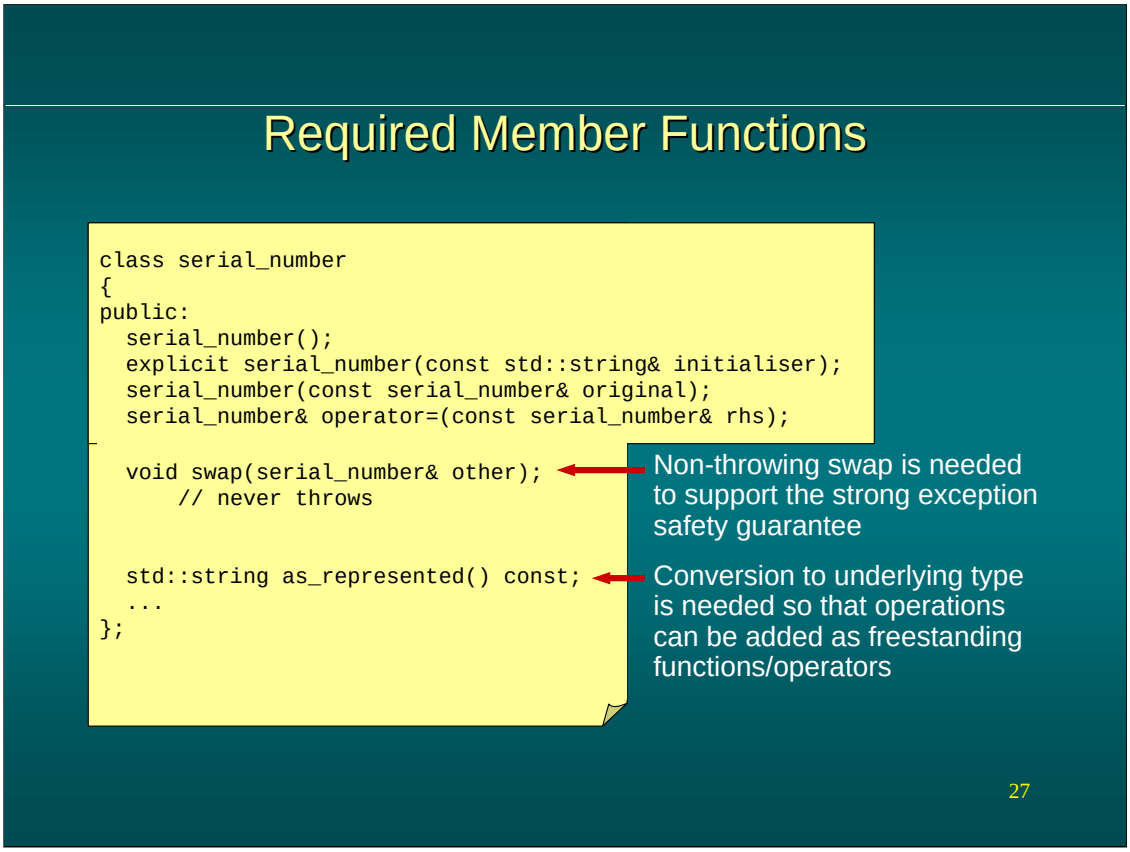
- There is a small minimum set of operations mandatory for objects to be usable
  - These must be implemented as members
- The additional operations needed depends on intended use
  - These can be implemented as freestanding functions/operators

26

Whole Value classes are very lightweight. Mostly they just have one data member – and underlying type (typically either a fundamental type or `std::string`), an instance of which holds the value.

There is a small set of operations that *must* be supported by each type. These operations must be implemented as member functions. This is the minimum set of member functions that allows further operations to be added as freestanding functions. Note that I'm not saying that generation schemes can not implement all operations as member functions – I am just setting a criteria for the minimum set of member functions.

Many types *will* need additional operations (what these are depends on the nature of the type). Therefore, generations schemes need, in their configuration, some means of specifying additional operations for the generator to add.



The slide shows an example illustrating the set of member functions that I consider to constitute the minimum set of operations. The presence of conversion construction, copy construction and copy assignment should come as no surprise. Therefore I'll just cover the remaining members.

Default constructor…

Not all values have natural default values. However, default construction must be included for the value to function with some parts of the standard library. For example some `std::map` operations require a default constructor.

`as_represented()`…

This returns the value as the underlying representation type. Its presence means that any additional operation (other than construction or destruction) can be added as a freestanding function.

`swap()`…

Given the possible need to write code honouring the strong exception safety guarantee, I regard this as an essential member function. True, it may be possible to get away without it, using only `as_represented()`. However, I think there will be situations in which writing generic code, where a swap operation guaranteed not to throw is required, becomes very tricky. Being able to fall back on a non-throwing `swap()` member function removes any need for client code to distinguish (for example) between whole values with `int` and `std::string` (or a user defined type, for that matter) as the underlying representation type.

Additional Operations

- Whole Values may require additional operations depending on how the type will be used
- For example:
  - A (textual) description may need `operator+` for concatenation purposes
  - *Quantities* require arithmetic operations

28

Which additional operations are required depends on how the type will be used. In some cases the types have characterisations associated with them that reflect the way in which they will be used, e.g. *quantities*.

In some cases, it will be necessary to add operations to types on an individual basis. In others, it will be necessary to add operations based on the types characterisation. For example, any type characterised as a *quantity* will require arithmetic operations [1] : increment, decrement, addition, subtraction, multiplication and division operations.

It is likely that two (textual) description objects will need concatenating. Therefore the type will need an `operator+` implementing this.

Note that conversions are an example of operations that need adding on an individual basis. Any attempt to generalise conversions (or implement them generically) is dangerous – unintended conversions may be implemented accidentally.

[1] Relational operations will also be required, but I've omitted these to keep things simple. Arithmetic operations will serve by way of example in later slides.

Type Production Using Templates

```
template <
    typename type,
    typename tag_type
    >
class whole_value
{
...
};
```

Whole Value types can be produced using a class templates and aliasing using `typedef`

Note the *tag* type used to disambiguate specialisations using the same underlying type

```
struct serial_number_tag {};

typedef whole_value<std::string, serial_number_tag>
        serial_number;
```

29

This is a technique I have used several times for producing small type libraries.

A class template is defined that supports the minimum set of member functions, and takes the underlying type as a template parameter. It also takes a second template parameter: a *tag* type to disambiguate specialisations of the template having the same underlying type. For example, consider the following two specialisations:

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

The above produced two aliases for the same specialisation.

This generation scheme is very limited in its usefulness because of the problem of adding operations (see next slide). Its usefulness is limited to producing classes supporting the only the minimum set of operations – i.e. the class template's member functions.

Problem with Adding Operations

- There is no straightforward way to specify additional operations
- Operations must not be declared as non-member function templates
  - It is too easy for types to acquire operations they are not intended to have

30

The only way to specify additional operations in this scheme is to code them specifically for each type requiring them.

Note that supplying freestanding function templates is not a satisfactory solution – it is recipe for errors. For example, consider the following template:

```
template <typename T>
bool operator< (const T& left, const T& right)
{...}
```

If this function template is in scope, the user can write (in error) a<b for any type, and if the type does not already have an `operator<` that is an exact match, the compiler will produce one in the form of an implicit specialisation of the template.

Note that this problem is known from experience with the standard libraries attempt to provide generic relational operations in the `std::rel_ops` namespace.

Adding in the Pre-Processor

```
#define CREATE_TYPE(type_name, underlying_type)                    \
  struct type_name##_tag {};                                       \
  typedef whole_value<underlying_type, type_name##_tag> type_name;

#define ADD_STREAM_INSERTION_OPERATOR(type_name)                   \
  std::ostream& operator<<(std::ostream& os, const type_name& obj) \
  {                                                                \
      os << obj.as_represented();                                  \
      return os;                                                   \
  }
```

```
CREATE_TYPE(seconds, unsigned int)
ADD_STREAM_INSERTION_OPERATOR(seconds)
```

Operations can now be added by specifying a requirement

```
seconds sec(7);
std::cout << sec << std::endl;
```

31

Here the pre-processor approach adds the facility for the programmer to choose what operations they want their types to support, and have them automatically included in the product.

Looking at it from a different angle: the user – i.e. the programmer in this case – specifies the features the product is required to have.

Writing `ADD_STREAM_INSERTION_OPERATOR(seconds)` provides the configuration information the pre-processor needs in order to generate a stream insertion operator for `seconds`.

Templates and Pre-Processor – In Favour

- The techniques use mechanisms provided by the C++ implementation
  - Whole Value generation can be packaged (and distributed) as a library
  - There are no additional artefacts to manage
    - Specification and configuration information is combined with the C++ code

32

The main advantage of techniques involving templates and the pre-processor are, in my experience, their ease of availability – they are included as standard in every C++ implementation.

Further, there are no additions to the build process, which there are when custom generators are used. Managing the extra step custom generators add is quite easy with Make and similar utilities, but is more of a chore using some popular IDEs.

Templates and Pre-Processor – Against

- There is no straightforward way to define groups of types and add operations to all types in the group
- Specification and configuration information is combined with the C++ code
  - The surrounding code is “noise”

33

It is possible to define groups of types such as *quantities*, such that all conforming generated classes have the relevant operations added to them. However, using the template and pre-processor approaches, there is no way that I would call *straightforward*. *Quantity* is an obvious characterisation that works well as an example, and could easily be catered for by the generation scheme. However, including in the generation mechanism, a mechanism allowing *the user* to configure the generation of quantities (or types satisfying other characterisations), is difficult. There are probably ways involving pulling in sets of operations (implemented as base class member functions) using inheritance, but these are not what I class as *straightforward*.

Combining specification/configuration information with C++ code adds noise to this information. Using conventional C++ program organisation (with the type library spread across more than one header file) the information is not likely to be in one place.

External Programs as Generators

- This approach is suited to projects where more control over the generation process is needed
  - Typically this will be the case for projects where there are hundreds of domain types (or more)
- Specification information is not part of the code in which the types are deployed
  - Typically it will be in one or more configuration files that drive the generator

34

I am assuming here that a custom written program will be involved at some point, even if utilities such as *Make* are included in the generation process.

A few years ago, I worked on a large scale project [1], developing a warranty claims system for a leading car manufacturer. There were two development phases: the first phase required a large domain type library to be produced (I can't remember how many, over a two hundred through), and the second required another significant number (a few dozen) adding. This library also had to support many conversions between types.

One projects of this scale, the control over the generation process afforded by custom written programs is valuable.

Typically, specification information – unless it is hard coded into the generator – will be in configuration files. There is no surrounding noise (compare with template and pre-processor techniques where the specification is integral with the C++ code).

[1] The project involved three companies in two countries. Just afterwords, I estimated that over a hundred people passed through the project during its initial three year development.

Specification/Configuration Mechanisms

- Three possible mechanisms are:
  - Using one or more configuration files
  - Using a database
  - Hard-coding the information into the generator program
- Only the configuration file(s) approach will be discussed here

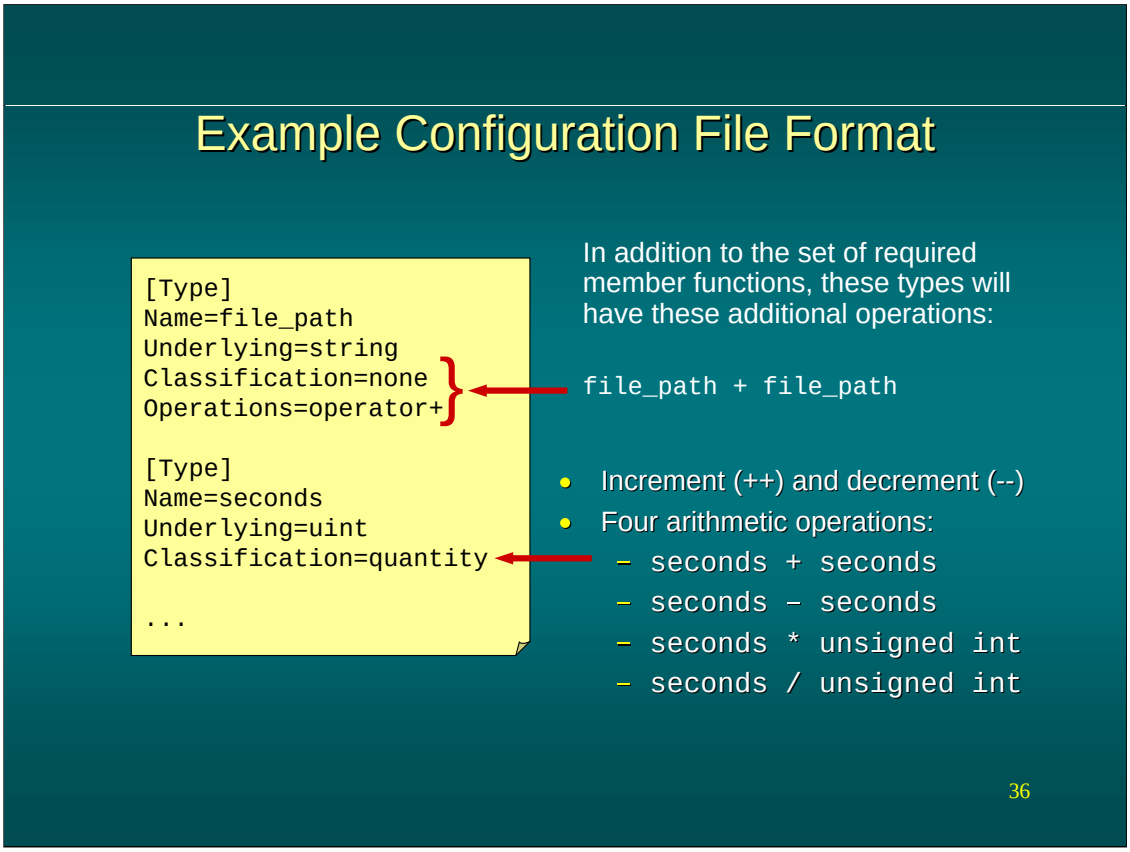
35

I will consider only the configuration file approach because it's the one I have experience of using – see the large project I alluded to on the previous slide. I've mentioned using a database and hard coding the information into the generator, because these are alternative approaches that occurred to me. I don't think they are of much use however. Using a database strikes me as overkill, while hard coding the information into the generator once again surrounds the configuration information with noise.

The approach we adopted in the project I have alluded to was to use a single configuration file. The generator was not very sophisticated in that, if the configuration file was modified, all types were regenerated, with the resulting several hour rebuild overhead. Happily this did not happen too often.

One option that has occurred to me since, is this…

When generating C++ classes/functions, the generator could keep a log of the specification to which they were generated. On subsequent runs, it could then compare the current input specification with the log, and regenerate only if the specification has changed. Note this assumes one header per class/function – otherwise a large rebuild results anyway, and the mechanism built into the generator looses its value.



On the project I've been talking about, the operations had to be specified individually in the configuration. I think, unfortunately, an opportunity was missed. The fragment on the slide shows how characterisations such as *quantity* can be catered for. Note that these characterisations could also be configured by providing a classification facility in the configuration file. For example:

```
[Classification]
Name=quantity
Operations=operator++, operator--                    \
           operator+, operator-,                     \
           operator*underlying, operator/underlying
```

Note that the default operands are the Whole Value type, whereas there is a simple method of specifying the second operand as being of the underlying type.

Now, any types specified as having the *quantity* classification will have these operations generated for them.

Case Study – End Note

- Whole Values exist in families determined by the (problem) domain they serve
- Families can be generated using a combination of templates and the pre-processor
  - It is symptomatic of C++'s richness, that this can be achieved entirely within the translation phases

37

Each domain has its own family of value based domain types. In the warranty claims project I have been alluding to, although some of the types were project specific, many (e.g. *vehicle identification number*) came directly from the motor vehicle industry.

Families of types can be generated using a combination of templates and the pre-processor. That this can be achieved without looking outside the C++ implementation, is a direct result of the richness of the features provided by modern C++.

Final Remarks

- C++ supports several approaches to programming – *Generative Programming* is among them
- The language functions well as both…
  - A means of implementation (templates and the pre-processor)
  - A target language (for generation by an external program)

38

Generative Programming is applicable to producing products both large and small in scale. This is in much the same way as Object Oriented Programming is applicable to projects both large and small.

Unfortunately, unlike Generic Programming and Object Oriented Programming, Generative Programming has not received the same amount of coverage in popular literature.

Generative Programming is for the generation of product families from components and requirements information captured as a configuration. However, I feel that the inclusion of the lifecycle – i.e. requirements through to product – in a single production process, gives this approach added value.

End

- I hope you found this talk interesting
- Thank you for your attention!
- I will post this presentation (with notes) at:

www.twonine.co.uk/documents.html

39

References…

[Abrahams2004] David Abrahams and Aleksey Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison Wesley, 2004

[Alexandrescu2001] Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley, 2001

[Coplien1995] Edited by James O Coplien and Douglas C Schmidt, Pattern *Languages of Program Design*, Addison-Wesley, 1995.

[Czarnecki2000] Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Techniques and Applications*, Addison Wesley, 2000

[Gamma1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[ISO1998] *International Standard: Programming Language - C++*, ISO/IEC 14882:1998(E), 1998.

[Radford2003] *Pattern Experiences in C++*, available from www.twonine.co.uk/documents.html