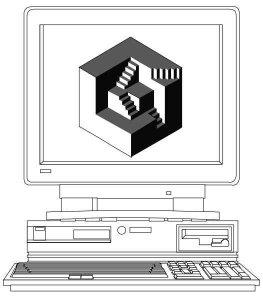
STL Patterns

A Design Language of Generic Programming

Kevlin Henney
kevlin@curbralan.com

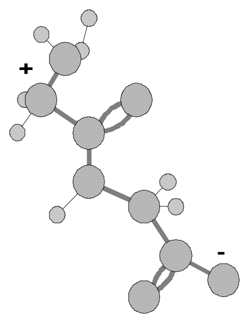
Agenda

- Pattern concepts
- The Standard Template Library
- Algorithmic patterns
- Iteration patterns
- Containment patterns
- Adaptation patterns
- Classification patterns
- Pattern misuses and non-uses



Pattern Concepts

- Intent
  - Briefly present pattern terminology and ideas
- Content
  - Patterns
  - Patterns of misunderstanding
  - Pattern communities
  - Pattern stories and sequences
  - Pattern compounds
  - Pattern languages



Patterns

- A pattern documents a recurring problem–solution pairing within a given context
  - A pattern is more than either the problem or the solution structure
  - A pattern contributes to design vocabulary
- A problem is considered with respect to forces and a solution that gives rise to consequences
  - The full form in which a pattern is presented should emphasise forces and consequences, also stating the essential problem and solution clearly

Patterns of Misunderstanding

- There are misconceptions concerning the pattern concept that are worth clearing up...
  - *Design Patterns* is a limited subset of design patterns and the pattern concept
  - Patterns are not frameworks, components, blueprints or parameter-based collaborations
  - Patterns are more than just a sample class diagram of the solution
  - Only language-independent patterns are language independent: patterns may be language specific

Pattern Communities

- Patterns can be used in isolation with some degree of success
  - Represent foci for discussion or point solutions
  - Offer localised design ideas
- However, patterns are in truth gregarious
  - They're rather fond of the company of patterns
  - To make practical sense as a design idea, patterns inevitably enlist other patterns for expression and variation

Pattern Stories and Sequences

- A pattern story brings out the sequence of patterns applied in a given design example
  - They capture the conceptual narrative behind a given piece of design, real or illustrative
  - Forces and consequences are played out in order
- More generally, pattern sequences describe specific ordered applications of patterns
  - A pattern story is to a pattern sequence as a pattern example is to an individual pattern

Pattern Compounds

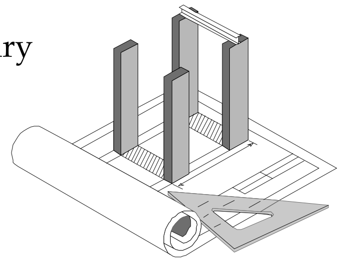
- Pattern compounds capture commonly recurring subcommunities of patterns
  - In truth, most patterns are compound, at one level or another or from one point of view or other
  - Also known as *compound patterns* or — originally and confusingly — *composite patterns*
- We can see many pattern compounds as named pattern subsequences
  - They are commonly recurring design fragments that can be further decomposed, if desired

Pattern Languages

- A pattern language connects many patterns together to capture a broader range of paths
  - The intent of a language is to generate a particular kind of system or subsystem
  - A pattern language can describe idiomatic design style, with general patterns incorporated into the language presented more specifically
- There may be many possible and practical sequences through a pattern language
  - In the limit, a sequence is a narrow language

STL Design Overview

- Intent
  - Offer a pattern-based overview of generic programming and the STL
- Content
  - Generic programming
  - The Standard Template Library
  - Design style
  - A pattern language



Generic Programming

- Generic programming is characterised by an open, orthogonal and expressive approach
  - It is an approach to program composition that emphasises algorithmic abstraction, loose coupling and a strong separation of concerns
  - More than just programming with templates
- Built on compile-time polymorphism and value-based programming
  - Templates, overloading and conversions
  - Copying and encapsulated memory management

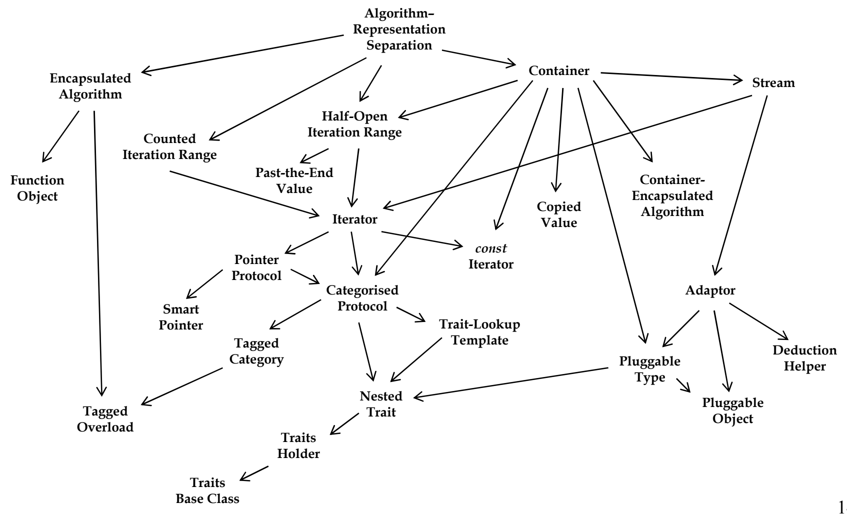
The Standard Template Library

- The STL originated in the work of Alex Stepanov and others
  - Revolutionary in design and native in style to C++
  - Because of its extensibility, the STL is more of a framework than just a library
- Incorporated into the draft C++ standard at a late stage in the standardisation process
  - Displaced previous, more simplistic utilities
  - Changed to fit it into the standard, and vice versa

Design Style

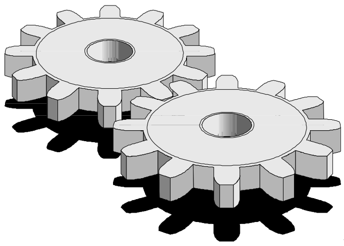
- Templates allow logical concepts to be more loosely coupled
  - Physical coupling may either increase or decrease
- Independent concepts can be expressed independently
  - For instance, algorithms are fully decoupled from containers via iterators
- Many libraries are now written in the generic style of the STL, e.g. numerous Boost libraries

A Pattern Language



Algorithmic Patterns

- Intent
  - Define algorithms independently of representation
- Content
  - *Algorithm–Representation Separation*
  - *Encapsulated Algorithm*
  - *Function Object*
  - *Half-Open Iteration Range*
  - *Counted Iteration Range*



Algorithm–Representation Separation

- A task involving traversal over a sequence of elements is independent of the representation
  - Encapsulating the traversal within the target that holds the element sequence couples them together
- Separate the logic of traversal from the target that represents the sequence of elements
  - A sequence is passed to an algorithm as a Half-Open Iteration Range or a Counted Iteration Range
  - The algorithm may be a custom loop or an Encapsulated Algorithm

Encapsulated Algorithm

- An iteration-based task is repeated with common control structure
  - It may be a complex algorithm or a simple loop
- Encapsulate the common control structure in a named function that is passed a range
  - A function template is typically used, so range traversal requires standard syntax and semantics constrained with respect to Categorised Protocol
  - Loop actions and predicates may be defined in terms of passed functions or Function Objects

Function Object

- An algorithm may need to be open with respect to its actions and conditions
  - Hardwiring reduces the basic genericity of an Encapsulated Algorithm
  - A function pointer may not carry sufficient execution context
- Define a type supporting the function-call operator and pass instances to algorithms
  - Transparent use with respect to real functions
  - Execution context captured in member data

Half-Open Iteration Range

- A whole or partial sequence of elements from a target needs to be traversed
  - The sequence is terminated intrinsically
- Denote the range by including the initial and excluding the post-ultimate elements
  - A Half-Open Iteration Range may refer to elements from a Container, array or an adapted type
  - Implement the range limits as Iterators
  - A Past-the-End Value is used to mark the end of the range, but is not otherwise used for access

Counted Iteration Range

- A known number of sequence of elements from a target needs to be traversed
  - The sequence may not be terminated intrinsically, i.e. no useful Past-the-End Value exists
- Denote the range by including the initial element with an explicit iteration count
  - The initial element is pointed to by an Iterator
  - For an Encapsulated Algorithm, it is conventional to use an _*n* suffix to differentiate it from an equivalent Half-Open Iteration Range algorithm

Iteration Patterns

- Intent
  - Define the mechanism for representing a position in a traversal through a sequence of elements
- Content
  - *Iterator*
  - *Pointer Protocol*
  - *Smart Pointer*
  - *Past-the-End Value*

Iterator

- How can a sequence of elements be traversed without compromising its encapsulation?
  - An iteration range and its use should be decoupled
- Introduce a separate object that represents and encapsulates a position in an iteration range
  - It is a handle that represents a level of indirection to and supports traversal of elements in its target
  - It may be coupled to the target representation
  - Ensuring the Iterator interface follows a Pointer Protocol capitalises on this notion of indirection

Pointer Protocol

- For a handle representing a level of indirection, what is an appropriate interface?
  - An Iterator is a handle to an element of its target
- Ensure its usage interface is that of a pointer
  - Makes use of existing programmer knowledge and offers a high degree of genericity
  - Raw pointers can be used for Half-Open Iterator Ranges on arrays, including a natural Past-the-End Value, and Smart Pointers for other target types
  - A Categorised Protocol narrows the full protocol

Smart Pointer

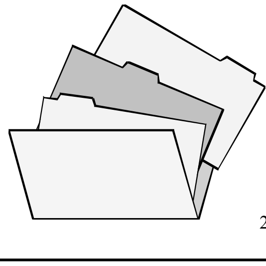
- A handle is to follow a Pointer Protocol, but it is not itself a pointer
  - A raw pointer would expose underlying representation of the target element sequence, and would offer an inappropriate iteration interface
- Through overloading, define the handle to support the Pointer Protocol directly
  - Define the operators appropriate for the Categorised Protocol modelled by the Iterator
  - A Past-the-End Value may need specific handling

Past-the-End Value

- A Half-Open Iteration Range requires a marker for its upper limit
  - For a full sequence of elements, such a post-ultimate marker cannot refer to a real element
- Ensure a special Iterator value exists that can be compared to, but is not for element access
  - A sentinel value may need specific implementation
  - Not all Categorised Protocols may support a Past-the-End Value, in which case a Counted Iteration Range must be used

Containment Patterns

- Intent
  - Define mechanisms for representing targets that hold traversable sequences of elements
- Content
  - *Container*
  - *Copied Value*
  - const *Iterator*
  - *Container-Encapsulated Algorithm*
  - *Stream*



Container

- Sequence elements are to be held within a program and modified or accessed repeatedly
  - As opposed to sources and sinks used once only
- A type that encapsulates a suitable data structure is used to hold Copied Values
  - Its interface follows a Categorised Protocol plus appropriate Container-Encapsulated Algorithms
  - Iterator accessors offer a Half-Open Iteration Range
  - If appropriate, Pluggable Types and Pluggable Objects can be used for structural policy

Copied Value

- Use of a Container should be as non-intrusive as possible for arbitrary element objects
  - A Container encapsulates its data structure's management
- Elements are copied into the Container and memory management is encapsulated
  - Non-intrusive and minimal requirement that copyability must be supported for element types
  - For heap-based objects, memory management must be handled by the Container owner

const *Iterator*

- A Container should not offer writeable access to its elements if its use is *const* qualified
  - A Container offers Iterator types as Nested Traits according to its Categorised Protocol, and these may allow writeable access indirectly
- Provide an Iterator and Iterator accessors for *const*-qualified Containers
  - Dereferencing a *const* Iterator does not allow non-*const* operations on the elements of the Container

Container-Encapsulated Algorithm

- External Encapsulated Algorithms allow use of existing or new operations on Containers
  - However, although this works in the general case, an external algorithm may not be able to take advantage of the data structure for its efficiency
- Define member functions for operations that are implemented more optimal internally
  - Where it corresponds, follow the name and semantics of the externally Encapsulated Algorithm

Stream

- Sequence elements are to be used in a single-pass sequence and may be based externally
  - Revisiting a readable sequence would involve recalculation or revisiting the external source
- Offer access to elements as a stream that consumes its elements as it progresses
  - Read, write or read-and-write access to elements is through extraction and insertion operators
  - Offer Iterators through an Adaptor interface that takes the Stream as a Pluggable Object

A Directory Stream in Public

```
class dir_stream
{
public:
    explicit dir_stream(const std::string &dir_name)
      : handle(opendir(dir_name.c_str()))
    {
    }
    ~dir_stream()
    {
        close();
    }
    operator const void *() const // stricter bool substitute also appropriate
    {
        return handle;
    }
    dir_stream &operator>>(std::string &rhs) // conventional stream extractor
    {
        if(dirent *entry = readdir(handle))
            rhs = entry->d_name;
        else
            close();
        return *this;
    }
    ...
};
```

A Directory Stream in Private

```
class dir_stream
{
    ...
private:
    dir_stream(const dir_stream &);
    dir_stream &operator=(const dir_stream &);
    void close()
    {
        if(handle)
        {
            closedir(handle);
            handle = 0;
        }
    }
    DIR *handle; // based on POSIX <dirent.h> API
};
```


Using a Directory Stream

- The use of a directory stream is fairly intuitive
    - The streaming idiom is a familiar one

```
void list_dir(const char *dir_name)
{
    dir_stream dir(dir_name);
    for(std::string entry; dir >> entry;)
        std::cout << entry << std::endl;
}
int main(int argc, char *argv[])
{
    if(argc == 1)
        list_dir(".");
    else
        std::for_each(argv + 1, argv + argc, list_dir);
    return 0;
}
```

A Directory Stream Iterator

```
class dir_iterator
  : public std::iterator<std::input_iterator_tag, std::string>
{
public:
    dir_iterator(); // construct Past-the-End Value
    explicit dir_iterator(dir_stream &); // take Pluggable Object
    const std::string &operator*() const;
    const std::string *operator->() const;
    dir_iterator &operator++();
    dir_iterator operator++(int);
    bool operator==(const dir_iterator &) const;
    bool operator!=(const dir_iterator &) const;
private:
    bool at_end() const;
    std::string value;
    dir_stream *dir; // hold Pluggable Object
};
```


Inside a Directory Stream Iterator

```
class dir_iterator ...
{
    ...
    explicit dir_iterator(dir_stream &stream)
      : dir(&stream)
    {
        *dir >> value;
    }
    dir_iterator &operator++()
    {
        *dir >> value;
        return *this;
    }
    const std::string &operator*() const
    {
        return value;
    }
    bool operator==(const dir_iterator &rhs) const
    {
        return at_end() && rhs.at_end();
    }
    ...
    bool at_end() const
    {
        return !dir || !*dir;
    }
};
```

Using Directory Stream Iterators

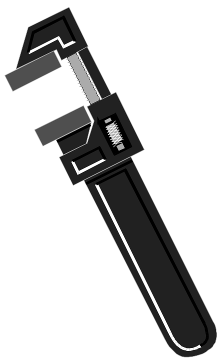
```
typedef std::ostream_iterator<std::string> output;
```

```
void list_dir(const char *dir_name)
{
    dir_stream begin(dir_name), end;
    std::copy(begin, end, output(std::cout, "\n"));
}
```

```
void sorted_list_dir(const char *dir_name)
{
    dir_stream begin(dir_name), end;
    std::set<std::string> sorted(begin, end);
    std::copy(
        sorted.begin(), sorted.end(),
        output(std::cout, "\n"));
}
```


Adaptation Patterns

- Intent
  - Patterns for adapting syntax and semantics
- Content
  - *Adaptor*
  - *Pluggable Type*
  - *Pluggable Object*
  - *Deduction Helper*



Adaptor

- Objects of a particular type need to be used in contexts that expect a different usage protocol
  - And changing either the code in calling context(s) or the called code is inappropriate or impossible
- Define a type that satisfies the expected interface and wraps the original object type
  - Heavy syntax can be lightened with the aid of a Deduction Helper
  - An Adaptor may be further generalised through a Pluggable Type or a Pluggable Object

Function and Container Adaptors

- For Function Objects there are many types that qualify as Adaptor types
  - Binders, negators and function-pointer adaptors, provided along with Deduction Helpers
  - These tend to rely on the presence of a full-set of Nested Traits that include argument types
- For Containers there are a number of 'dispenser' types that are Adaptors
  - *stack*, *queue* and *priority_queue* take a Container as a Pluggable Type, but are not themselves Containers

A Conversion-based Adaptor

```
class c_str
{
public:
    c_str(const char *ptr)
      : ptr(ptr)
    {
    }
    bool operator<(const c_str &rhs) const
    {
        return std::strcmp(ptr, rhs.ptr) < 0;
    }
    ...
private:
    const char *ptr;
};
```

```
std::map<c_str, symbol> symbols;
```


Pluggable Type

- An Adaptor relies on a particular adaptee type protocol but not on a specific implementation
  - The protocol may define an object type or a policy
- Define the Adaptor type as a template with a parameter for the adaptee type
  - For policies, Nested Traits are normally provided
  - A Pluggable Type may also correspond to Pluggable Objects
  - It is common to provide an out-of-the-box default

A Policy-based Adaptation

- In the STL a common policy approach is based on providing a Pluggable Type for instances
  - In contrast to using policies as a purely compile-time concept and solely for their Nested Traits

```
struct c_str_less
{
    bool operator()(const char *lhs, const char *rhs) const
    {
        return std::strcmp(lhs, rhs) < 0;
    }
};
```

```
std::map<const char *, symbol, c_str_less> symbols;
```


Pluggable Object

- An existing object needs to be adapted for use in a different context
  - It is common for Adaptors to create and fully encapsulate the adaptee on which they operate
- Define an Adaptor type whose instances are passed and wrap, by reference, an object
  - The object is plugged in at runtime and can be referred to outside the context of the Adaptor
  - The Adaptor may offer access to the adaptee

Iterator Adaptors

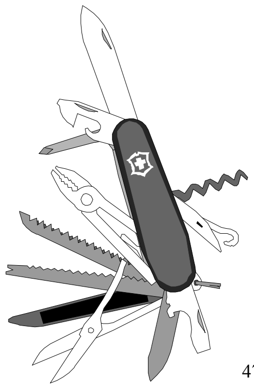
- For Iterators there are many different kinds of adaptation...
  - *reverse_iterator* adapts an Iterator directly
  - *raw_storage_iterator* adapts uninitialised memory
  - *back_insert_iterator* and *front_insert_iterator* adapt Containers to be Iterators
  - *insert_iterator* adapts a Container and Iterator combination to be an Iterator
  - The stream Iterator types can be considered to adapt Streams (and *streambuf*s) to be Iterators

Deduction Helper

- Generic code can lead to Pluggable Objects with complex and tedious declared types
  - Good generic code should be (very) light on its use of template parameter lists outside of declarations
- Where possible, wrap up and pass Pluggable Objects using a function template
  - The argument's type is deduced from usage and that type is in turn used in the declaration of the appropriate result type

Classification Patterns

- Intent
  - Patterns for compile-time type information
- Content
  - *Categorised Protocol*
  - *Tagged Category*
  - *Tagged Overload*
  - *Nested Trait*
  - *Traits Holder*
  - *Trait Base Class*
  - *Trait-Lookup Template*



Categorised Protocol

- There is no formal type model for template parameters, but types cannot be arbitrary
  - There is an implied type based on protocol that is used by the template
- Categorise a parameter's protocol according to usage syntax and expected semantics
  - Nested Traits are often a requirement
  - Categorised Protocols may subsume one another
  - For instances, constant-time operations are often the difference between subsuming protocols

Iterator Categories

- Iterators are categorised by the operations they offer and the traversal they support
  - *Output iterators* are for single-pass output (and do not require a Past-the-End Value)
  - *Input iterators* are for single-pass input (and do require a Past-the-End Value)
  - *Forward iterators* are for general single-pass access
  - *Bidirectional iterators* allow iteration to and fro
  - *Random access iterators* support constant-time indexing

Container Categories

- Containers are categorised by their operations and the organisation of their elements
  - *Containers* offer the *input iterator* category or better
  - *Reversible containers* offer the *bidirectional iterator* category or better
  - *Sequences* hold elements by position
    - Optional constant-time operations are also specified
  - *Associative containers* offer ordered key access
    - Unique and non-unique key variants

Tagged Category

- How can a template take advantage of the Categorised Protocol of an actual parameter?
  - A Categorised Protocol places a requirement on an actual type, but a template does not know if the protocol is a more specialised subtype
- Present the category as an empty type make associated with the actual parameter type
  - A temporary instance of the Tagged Category can be used to select a Tagged Overload

Tagged Overload

- An operation can be run with a different algorithm, depending on Categorised Protocol
  - The balance is between genericity and specificity
- Use a constructed temporary Tagged Category object to differentiate between algorithms
  - The main operation simply dispatches across an overload set that is based on Tagged Category
  - Overload transparency with respect to support for different Categorised Protocols is offered

Nested Trait

- A type intended for use as a template parameter needs to make its traits available
  - Traits include types, constants and functions associated with the type rather than its instances
- Declare the traits as members of the type
  - Types that are classes can be defined as nested
  - Constants and functions are defined as *static*
  - If a type cannot be modified to have traits added, provide a Trait-Lookup Template specialisation

Traits Holder

- It can be tedious to declare Nested Traits when many are required and they are often related
  - And also similar across different implementations with the same or related Categorised Protocols
- Define a type in order to hold the traits of another type
  - A Traits Holder is often templated, so that it generates its Nested Traits from parameters
  - A Traits Holder can be used as a Pluggable Type, a Trait Base Class or a Trait-Lookup Template

Trait Base Class

- For a given type its traits may be defined or generated from a Traits Holder
  - However, re-exporting them as Nested Traits is tedious and error prone
- Use the Traits Holder type as a base class
  - The traits are accessible to the user of the derived type, but may need additional qualification when used within templated derived classes
  - The base class does not confer any runtime polymorphic capabilities

Trait-Lookup Template

- Traits are needed for working with a type that cannot have new traits added to it
  - The type may be a built-in or a closed class
  - The use of the type should be non-intrusive
- Define a class template for accessing the traits and specialise it according to the target type
  - The primary template can default to referring to its parameter's Nested Traits
  - Favour a narrow rather than a broad range of traits

Pattern Misuses and Non-uses

- Intent
  - Present examples of pattern misapplications and missed opportunities in the STL
- Content
  - The *allocator* model
  - Function objects
  - Containers
  - Strings
  - Streams

The *allocator* Model

- A Pluggable Type to define allocation policy is not unreasonable, but...
  - The *allocator* model is (very) limited and overly complex and intrusive for the little it achieves
  - It is constrained so as to make all the interesting opportunities unimplementable in a portable way
  - Effective memory management of a container whose representation and management is not fully open to you is like eating with a knife and fork... held with chopsticks... through mittens

Function Objects

- Nested Traits for Function Object arguments are unnecessary and incomplete
  - Argument types should be template deduced, and only the result type needs to be declared, which would offer greater flexibility and less syntax
  - Using a Traits Holder as a base class is therefore either unnecessary or a missed opportunity for a Traits-Lookup Template and a Tagged Category
- Adaptors are unnecessarily awkward to use
  - E.g. the binder model

Containers

- Container capabilities are not well published or consistent
  - Should have been explicit through a Tagged Category and a Traits-Lookup Template
  - E.g. *remove* versus *erase*
- Inappropriate template specialisation for *std::vector<bool>*
  - An optimisation that can also be a pessimisation
  - The specialisation is subtype incompatible with respect to template parameters

Strings

- Incomplete application of generic-programming techniques to strings
  - *std::basic_string* supports a limited Container interface, plus a larger index-based legacy interface
  - Strings should have been modelled generically with respect to Categorised Capabilities
- Misuse of Pluggable Type for character traits
  - This is not the way to handle localisation

Streams

- A common misuse of the STL model is to use a Container when a Stream is appropriate
  - Adopting the wrong metaphor leads to code of unnecessary complexity and no added benefit, e.g. directory streams
- Retrospectively, the notion of Streams could have been considered more generically
  - More general Stream Iterator Adaptors could then have been provided

In Conclusion

- The STL follows a largely consistent and clearly defined design model
  - Its regularity and orthogonality offer greater simplicity and more flexibility than might be assumed from the number of library components
- Generic programming and the design of the STL can be presented as a pattern language
  - 25 such patterns have been presented here
  - More can be identified that further cover interface and implementation design and usage