IDENTIFYING PROGRAMMING IDIOMS IN C++ GENERIC LIBRARIES

A thesis submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Masters of Science

by
Ryan Holeman
December, 2009
Thesis written by

Ryan Holeman

B.S., Kent State University, USA 2007
M.S., Kent State University, USA 2009

Approved by

Jonathan I. Maletic, Advisor

Robert A. Walker, Chair, Department of Computer Science

John Stalvey, Dean, College of Arts and Sciences
TABLE OF CONTENTS

LIST OF FIGURES ........................................................................................................... VI

LIST OF TABLES ............................................................................................................. X

DEDICATION ................................................................................................................ XI

ACKNOWLEDGEMENTS .............................................................................................. XII

CHAPTER 1 INTRODUCTION ...................................................................................... 1
  1.1 Motivation ........................................................................................................... 2
  1.2 Research Contributions .................................................................................... 2
  1.3 Organization ..................................................................................................... 3

CHAPTER 2 RELATED WORK .................................................................................. 5
  2.1 Design Pattern Identification .......................................................................... 5
  2.2 Generic Libraries ............................................................................................. 9

CHAPTER 3 IDIOMS FOR GENERIC PROGRAMMING ........................................ 11
  3.1 Functors ........................................................................................................ 11
  3.2 Metafunctions ............................................................................................... 12
    3.2.1 Example: Type Accessors ................................................................ 12
    3.2.2 Example: Type Traits ...................................................................... 13
    3.2.3 Example: Unified Metaprogramming .......................................... 14
  3.3 Tag Dispatch .................................................................................................. 16
    3.3.1 Example: STL Iterator Advance .................................................. 17
    3.3.2 Example: Boost GIL ...................................................................... 20
  3.4 Traits Class .................................................................................................. 21
6.2.1 Generic Libraries ................................................................. 54
6.2.2 Concept Libraries ............................................................ 56
6.2.3 Extension Libraries ........................................................... 57
6.2.4 Unclassified Libraries ....................................................... 59

CHAPTER 7 DISCUSSION ............................................................... 60
7.1 Measuring Library Quality ................................................... 60
7.2 Evolving Generic Libraries .................................................. 60
7.3 Motivating Language Evolution ............................................ 61

CHAPTER 8 CONCLUSIONS .......................................................... 62

REFERENCES .............................................................................. 63
LIST OF FIGURES

Figure 1. The greater class template encapsulates a strategy for the ordering of objects of type T. ........................................................................................................................................... 11

Figure 2. The iterator_reference metafunction is an alias for a more complex type expression. ........................................................................................................................................... 12

Figure 3. The is_same type trait returns true if the two template arguments are the same type. ........................................................................................................................................... 13

Figure 4. The remove_const metafunction will remove a const modifier from a typename if given. ........................................................................................................................................... 14

Figure 5. The MPL defines a Boolean type and constants as metafunctions with both type and integral components. ........................................................................................................................................... 15

Figure 6. The if_ metafunction returns either the type T or F depending on the value of the Boolean metafunction C. ........................................................................................................................................... 15

Figure 7. The STL tag class hierarchy classifies iterators based on their traversal operations and properties. ........................................................................................................................................... 17

Figure 8. The iterator_category function maps an iterator type to its corresponding tag class. ........................................................................................................................................... 18

Figure 9. Three implementations of the advance function specialize the semantics and performance of the operation for different kinds of iterators. ........................................................................................................................................... 19

Figure 10. The advance algorithm dispatches one of its variants by invoking an overload on the result type of iterator_category function. ........................................................................................................................................... 19
Figure 11. The fill_pixels algorithm dispatches the algorithm to one of the two auxiliary functions based on the planarity of the view, which is decided by the metafunction is_planar.

Figure 12. The iterator_traits class derives a number of different associated types for its Iter template parameter.

Figure 13. The partial specialization of iterator_traits for pointers (T*) adapts pointers to the STL iterator concept.

Figure 14. Two implementations of negate: one that negates an integral value and one that negates the result of a nullary function.

Figure 15. The enable_if template implements a SFINAE enabler that can be used to control the instantiation of templates based on the properties of types.

Figure 16. Two equivalent definitions of the infinity function are enabled (instantiable) if T satisfies the is_floating_point metafunction.

Figure 17. C++0x supports the application of enablers in the template parameter list if there are no other constrained overloads.

Figure 18. The sqrt function is overloaded for integral types and non-integral types.

Figure 19. The SameType concept class implements concept-checking logic and error message facilities for the Origin.Concepts library.

Figure 20. The concept_assert framework instantiates a Model’s nested assertion class and its destructor in order to execute a static assertion on a set of constraints.
Figure 21. The `find` algorithm uses the `concept_assert` asserter to ensure that its constraints are satisfied prior to instantiation. If the constraints are not satisfied, the compiler will generate a concept-specific error message. 31

Figure 22. The `less_than_comparable` template provides default implementations of complementary operations that can be derived from the less-than operator. 32

Figure 23. A hypothetical definition of an `integer` class implements ordering operators using the CRTP idiom by deriving from a base class template, parameterized over its own type. 32

Figure 24. An abbreviated implementation of Boost’s `iterator_facade` uses CRTP to adapt a user-defined type to an iterator concept. 33

Figure 25. The `node_iterator` class adapts a `list_node` to an iterator by using the CRTP-based `iterator_facade` and implementing the requisite protocol. 34

Figure 26. The architecture of the fingerprint framework system. 35

Figure 27. An example from STL `type_traits.h` using the preprocessor on a metafunction which abstracts the template instantiation. 37

Figure 28. The `class` keyword is highlighted to indicate that a fingerprint method uses this syntax to derive information about the program element. 38

Figure 29. Functors are recognized as classes with a call operator overload. 38

Figure 30. Type metafunctions are recognized as class or structure templates with a typedef named `type`. 39

Figure 31. Integral metafunctions are recognized as classes or structures with Boolean return types. 39
Figure 32. Tag classes are recognized as structures with empty bodies. ......................... 40

Figure 33. Tag dispatch is recognized from overloaded functions with tag classes or type checks as argument types. ................................................................. 40

Figure 34. Traits class is recognized from classes or structures with multiple typedefs where non of them set the type................................................................. 41

Figure 35. SFINAE is recognized from templated classes or structures setting a template parameter type to void. ................................................................. 41

Figure 36. CRTP is recognized from classes or structures using their own class names in an inherited template parameter. ................................................. 42

Figure 37. Definitions of precision and recall. ........................................................... 44

Figure 38. Counts of classes and functions were extracted from the studied libraries.... 51

Figure 39. The percentage of classes and functions that are defined as templates........ 51

Figure 40. Percentages of specialized class templates and overloaded functions. ....... 53
LIST OF TABLES

Table 1. Precision and recall statistics for the Boost::Graph library. ............................... 45
Table 2. Precision statistics for the Boost gil, and python libraries.................................. 47
Table 3. Libraries examined in the empirical study............................................................ 49
Table 4. Generic libraries provide sets of generic algorithms and data structures
described by concepts. The use of tag classes, tag dispatch, and type traits directly support this ........................................................................................................................................ 55
Table 5. Concept libraries provide metaprogramming support in the form of
metafunctions and traits classes.......................................................................................... 57
Table 6. Extension libraries extend an existing domain by providing new algorithms or
data structures.................................................................................................................. 58
DEDICATION

This work is dedicated to my mother Sophia Holeman.
ACKNOWLEDGEMENTS

I would like to thank Andrew Sutton for showing me the ways of template metaprogramming.

Ryan Holeman

November 2009, Kent, Ohio
CHAPTER 1

Introduction

Generic programming equips developers with a mechanism for designing adaptable and efficient generic libraries. The programming paradigm is rooted in the ability to parameterize algorithms and data structures with application-specific types at compile time in order to compose more complex software abstractions. While extensive use of the compiler may yield substantial benefits in terms of elegant software design and increased runtime performance, the resulting software is typically less easy to comprehend. Frequently littered with cryptic templates and macros, such libraries can perplex even the most seasoned programmer.

Complex source code is not, however, restricted to the domain of generic programming and the design of generic libraries. Regardless of the paradigm, as source code becomes more complex, systems become increasingly difficult to maintain and comprehend. In response, computer programmers have developed techniques to help manage the ever-increasing size and complexity of their software. In particular, programming idioms and design patterns provide mechanism for “chunking” recurring programming and design problems into smaller abstract elements that can be used to improve comprehension or facilitate reuse. These concepts have been effectively applied to improve the maintainability of procedural and object-oriented systems, and their real-world benefits are practically immeasurable.
1.1 Motivation

While idioms and design patterns are known to exist for the generic paradigm, they are less widely known which also implies that they are even less frequently recognized by (or recognizable to) to lay programmers. The lack of comprehension in the generic paradigm can be attributed to the relatively primitive nature of the host language and its idioms. Whereas languages that embrace other paradigms, esp. object-oriented programming, have evolved features that more readily express their idioms and abstractions, the language features underlying generic programming exist at the most base level – class and function templates. A direct result of the lack of linguistic features for expressing higher-level abstractions in the generic paradigm is the abuse of the C++ syntax to implement the core features of generic algorithms, data structures, and libraries. Analogically speaking, we claim that the state of the art in generic programming in C++ is nearly equivalent to the state of the art for object-oriented programming in C. In order for a programmer to become proficient in the paradigm, they must master the idioms and patterns that define its elements. As a result, most programmers are unable to easily comprehend the design and implementation of generic libraries, and this lack of understanding propagates a common misperception that generic programming cannot produce high-quality, maintainable software.

1.2 Research Contributions

We address the gap in comprehension by identifying the common idioms and patterns used in the design of generic libraries and formulating a reverse engineering system which has been integrated into srcTools for the automatic identification and
recovery of those elements. We present a catalog of common idioms and patterns used in the construction of generic libraries. These descriptions are based on a long-term study and use of such libraries. We describe a lightweight design recovery system for automatically identifying instances of these idioms within generic libraries. The reverse engineering system defines a layered approach for automatically identifying design elements by first identifying the underlying idioms in which higher level abstractions are defined. The system is validated by applying the tool to recover design instances from a number of real world generic libraries including Boost’s Graph, GIL and Python Libraries.

We present our results by describing the composition of these libraries and others such as the Computation Geometry Algorithms Library (CGAL), Loki, Soci, the Standard Template Library (STL) and the rest of the boost libraries with respect to their use of idioms and design patterns. In this way, the approach and design recovery tool help reduce the cognitive burden on developers by helping to raise the level of discourse for describing the design of generic libraries written in C++.

1.3  Organization

This thesis is organized as follows: In section 2 related works in respects to design pattern analysis, generic libraries and reverse engendering are presented. In section 3 selected idioms present in generic libraries are explained and examples of each are given. Section 4 describes the fingerprinting framework tool implemented for this research along with detailed descriptions of each fingerprints unique signature. Section 5 then goes on to validate the accuracy of this system though the calculation of precision and
recall statistics. Research results from running this framework against multiple widely used generic libraries is presented in section 6. Section 7 discusses the findings of this paper along with how they are relevant in the scope of the language.
CHAPTER 2
Related Work

Design recovery is the process of generating high-level abstract views or models of low-level source code [Chikofsky, Cross 1990]. The primary goal of design recovery is to promote comprehension, which is concerned with reducing the effort of programmers to understand large or complex programs. Design recovery is most often associated with the recovery of a program’s design, but it can also be applied to recover the program’s architecture or even the elements of the problem domain. In the design recovery process, source code models are produced through source code analysis techniques and are used to produce documentary artifacts or other (analyzable) models.

One specific goal in the domain of design recovery is the automated identification of design patterns [Gamma, Helm, Johnson, Vlissides 1995] within a body of source code. In this work, we focus on the recovery of idioms, which are similar to design patterns in that there are distinguishable components and relationships.

2.1 Design Pattern Identification

The identification of design patterns is rooted in the parsing and abstraction of information from source code. Many of the current automated design detection programs use the Columbus reverse engineering framework as a fact extractor [Ferenc et al. 2001; Ferenc et al. 2002; Ferenc, Siket, Gyimóthy 2004]. The framework parses C++ source
code and can extract program elements and relations. Columbus’s UML output feature is what most design pattern identification programs use.

DPML [Balanyi, Ferenc 2003] is a plugin written for the Columbus framework to detect design patterns in source code. It uses an XML syntax to represent design patterns (basically a graph description) and then compares the XML representation against Columbus’s abstract semantic graph output. DPML project attempted to find a total of 28 different pattern types in four large open source software projects. Of the 28, only about six patterns were accurately found (Factory, Prototype, Adapter, Bridge, Strategy and Template Method). Their accuracy was validated though the use of precision statistics.

Pat [Kramer, Prechelt 1996] is a program that converts C++ code into PROLOG for design pattern analysis. As the research was carried out in 1996, the paper has become dated but its content is still very relevant to related works. It examines class, attribute, and method names, semantics such as constructors, destructors, aggregation, association, and inheritance relationships to determine design pattern classifications. Once the desired source code attributes are converted to PROLOG, they are then compared against predefined PROLOG representations of the design patterns. The Pat system finds four patterns with subpar accuracy and is validated though hit statistics as in DPML. The four patterns found in this research are Adapters, Bridges, Composites and Decorators.

Unlike the above related work, where the source code is examined in a lower level, other related applications convert source code to UML to determine design
patterns. Many of these, use tools such as Columbus as a backend to convert the source code to UML. Once the source code has been converted to UML they then analyze the UML representation for design patterns. These UML based systems include Maisa, IDEA and SPOOL [Bergenti, Poggi 2000; Ferenc, Gustafsson, Muller, Paakki 2002; Keller, Schauer, Robitaille, Page 1999].

Maisa [Ferenc, Gustafsson, Muller, Paakki 2002] is likely to be the most significant research project done utilizing UML analysis techniques to extract design patterns. This research merges the two frameworks: Columbus and Maisa. Maisa itself is a tool that analyzes software’s architecture from its UML representation. Marisa is capable of identifying an array of software metrics and also capable of identifying design patterns present in UML diagrams. By combining this tool with Columbus UML output capabilities, design patterns can be identified from source code. There are seven patterns identified by this system. They include singleton, visitor, builder, factory, prototype, proxy and memento. Unfortunately, there are no statistics in their research that states its accuracy.

Spool [Keller, Schauer, Robitaille, Page 1999] is yet another UML based pattern detection system that find the bridge, factory method, and template method. In this work, once a UML based system is composed its attributes are then stored in a design repository. It is from this repository pattern queries are run in order to determine patterns. As with Maisa, no accuracy results on the systems precision or recall is presented.
The design pattern recovery tool OSPREY [Asencio, Cardman, Harris, Laderman 2002] implements a middleweight parsing technique that analyzes declaration and definition files to recover design patterns. OSPREY proposes to find six patterns which include factory, singleton, decorator, proxy, bridge and strategy. Their precision and recall for their system was based on a small set of test source code instead of actual OO software. Their results were accurate for the test code it was run against but is undetermined for real life software.

All of the applications above target the identification of design patterns in object-oriented programs. In our work, we focus on the identification of these elements in generic libraries in C++. The reverse engineering of these libraries is known to be a difficult problem due (in part) to the ubiquitous use of templates – a serious challenge for reverse engineering and design recovery programs. Although there have been some efforts to reverse engineer or analyze extensively templated source code [Gregor, Schupp 2005; Gshwind, Pinzger, Gall 2004; Porkoláb, Mihalicza, Sipos 2006; Schupp, Gregor, Musser, Liu 2002; Sutton, Holeman, Maletic 2009; Sutton, Maletic 2008; Zalewski, Schupp 2006], we know of no other applications that can identify the elements of a generic library’s design.

Unlike the approaches listed, this underlying technologies work is based on srcML, which defines a lexical XML markup for source code [Collard, Kagdi, Maletic 2003; Collard, Maletic, Marcus 2002]. The lightweight parsing and analysis approach espoused by srcML enables the rapid construction of lightweight analysis tools [Maletic, Collard, Kagdi 2004]. srcML has been used to support program transformation [Collard,
Maletic 2004], meta-differencing [Collard, Kagdi, Maletic 2006], traceability [Maletic, Collard, Simoes 2005], the automatic identification of function stereotypes [Dragan, Collard, Maletic 2006; 2009], design recovery for object-oriented C++ programs [Sutton, Maletic 2005; 2007b], C preprocessor analysis [Sutton, Maletic 2007a], and efforts to reverse engineer template instances [Sutton, Holeman, Maletic 2009] and C++0x concepts [Sutton, Maletic 2008]. In this work, we develop fingerprinting programs directly on the srcML output.

2.2 Generic Libraries

The first specification of concepts in their current form can be attributed to the Standard Template Library (STL) [Austern 1998; Musser, Stepanov 1994]. Concepts for these libraries are specified as supplemental documentation that classifies the elements in the library according to their syntax and semantics. The concepts are used to express requirements on template parameters to generic algorithms and data structures. The STL is the archetypal example of a C++ generic library. It helped lay the foundations for every C++ generic library that followed. Techniques for generic programming that were pioneered by the STL are at the core of many open source libraries including in a variety of domains: graph data structures and algorithms [Kühl 1998; Siek, Lee, Lumsdaine 1999; 2001], matrices and linear algebra [Gottschling, Wise, Adams 2007; Gregor, Lumsdaine 2005; Siek 1998], image processing and vision [Köthe, Weihe 1998], computational geometry and topology [Brönnimann, Kettner, Schirra, Veltkamp 1998; Fabri et al. 2000; Heinzl, Spevak, Schwaha 2006], data mining [Hasan, Chaoji, Salem,
Zaki 2005; Spevak, Heinzl, Schwaha 2006], and financial engineering [Leong et al. 2005].

The adoption of the paradigm inspired several methods for implementing concepts as constraints [Siek, Lumsdaine 2000; Zólyomi, Porkoláb 2004]. This work pioneered the basic infrastructure required for providing language-based concept support for C++. Comparative studies of languages supporting generic programming helped elicit requirements [Garcia et al. 2003; Siek, Lumsdaine 2005]. In 2005 two competing proposals for integrating concepts into the C++ language were introduced [Siek et al. 2005; Stroustrup, Dos Reis 2005]. Ultimately, the introduction of concepts into the C++ language and Standard Template Library introduced more issues than could be solved in a timely fashion, and they were removed from the language for the time being.

The introduction of concepts has already encouraged researchers to begin exploring practical aspects of their usage within generic libraries [Bourdev, Järvi 2008; Järvi, Marcus, Smith 2007; 2009]. Approaches for axiom-based testing have also been investigated [Bagge, Valentin, Haveraaen 2008]. A number of source code analysis methods have been proposed to help support the maintenance and evolution of these generic libraries. Applications include the static checking of library usage [Gregor, Schupp 2005], the impact analysis of changing concepts [Zalewski, Schupp 2005; 2006], the automated inference of type constraints, [Sutton, Maletic 2008] and the abstract interpretation of template structures [Sutton, Holeman, Maletic 2009].
CHAPTER 3

Idioms for Generic Programming

In this section, we present a list of idioms and recurring patterns that are specific to the domain of generic programming. These programming elements represent the basic units of construction of generic libraries.

3.1 Functors

Functors are a type of micro-pattern which is closely related to the strategy design pattern. They encapsulate functional variations that parameterize components of an algorithm. The functor itself is an encapsulation of a function that overloads the call operator, \texttt{operator()}. A commonly used example of a functor is the \texttt{greater} class that is shown in Figure 1. In this example, the \texttt{greater} functor encapsulates the greater-than comparison of two objects via \texttt{operator>}
.

\begin{figure}[h]
\begin{lstlisting}[language=C++]
Template <typename T>
struct greater : binary_function<T, T, bool> {
    bool operator(T const& x, T const& y) const{
        return x > y;
    }
};
\end{lstlisting}
\caption{The greater class template encapsulates a strategy for the ordering of objects of type \texttt{T}.
}
\end{figure}

The reason why they are closely related to the strategy pattern is that the strategy pattern is what is calling the function or class template.
3.2 Metafunctions

A template metafunction is a class template that performs compile-time computations on integral constant expressions (ICEs) or type expressions, returning the computed result as a `typedef` or `static` constant. Operating on compiler metadata, metafunctions come in two flavors: type metafunctions operate on type expressions and integral metafunctions operate on integral constant expressions (ICEs). A Boolean metafunction is a special kind of integral metafunction that returns a Boolean constant. Although metafunctions can be used to create integral metaprograms, they are most frequently used as a method of effecting adaptation in generic libraries by allowing library authors to develop compile-time programs that work within the C++ type system to select or compose new data types.

3.2.1 Example: Type Accessors

One common use of metafunctions is to create so-called type accessor metafunctions. A type accessor is a metafunction that returns a type associated with or derived from another. Figure 2 shows the `vertex_property_type` metafunction from the Boost.Graph library (BGL), which returns the corresponding type defined within the graph.

```c++
template <typename Graph>
struct vertex_property_type {
    typedef typename Graph::vertex_property_type type;
};
```

Figure 2. The `iterator_reference` metafunction is an alias for a more complex type expression.

Here, the `vertex_property_type` is a type accessor that returns the vertex property type of a graph. Note that since the metafunction is implemented as a class template, the
accessor can be implemented for a 3rd party graph definition by simply specializing the template over the externally defined graph type. Decoupling the access of an associated type from the source of its definition allows greater interoperability between components in a generic context.

3.2.2 Example: Type Traits

A type trait is a kind of metafunction that implements a query or transformation on a type expression. A type trait query is a Boolean metafunction, and is typically written as a set of two or more class templates: a general case, and a set of specializations. For example, the `is_same` type query in Figure 3 determines whether two types are the same. Versions of this type trait can be found in C++0x Standard Library and the Boost.TypeTraits library.

```cpp
template <typename T, typename U>
struct is_same {
    static bool const value = false;
};
template <typename T>
struct is_same<T, T> {
    static bool const value = true;
};
```

**Figure 3.** The *is_same* type trait returns true if the two template arguments are the same type.

A type transformation is a type metafunction that returns a modified definition of a type. For example, the `remove_const` type transformation will safely remove a const specifier from a typename. Its implementation is shown in Figure 4.
Figure 4. The `remove_const` metafunction will remove a `const` modifier from a typename if given.

Here, the general case is instantiated for any type that is not prefixed with a `const` specifier, and simply applies an identity transformation. The specialization matches any type that does include a `const` specifier, and implements the transformation by leaving the `const` out of the result type.

3.2.3 Example: Unified Metaprogramming

We use the term unified metaprogramming to describe the integration of integral and type metafunctions. This is done by projecting non-type elements of the programming language (i.e., integral constants and templates) into the type system by wrapping them as class templates. The Boost.MPL is a collection of metaprogramming algorithms and data structures that helps systematize the use and development of metafunctions in generic libraries.

The projection of integral and Boolean constants in the MPL is done by defining them as type metafunctions with associated integral constants. The definition of the canonical Boolean types and values of the MPL are shown in Figure 5.
template <bool B>
struct bool_ {
    typedef bool value_type;
    static bool const value = B;
    typedef bool_<B> type;
};
typedef bool_<true> true_;
typedef bool_<false> false_;
and type metafunctions into a single set of concepts has had a substantial impact on the way that programmers develop generic libraries.

3.3 Tag Dispatch
Tag dispatch is a technique that is used in generic libraries to enable function overloading of categories or kinds of types (as opposed to specific types). This technique allows library developers to implement different versions of a generic algorithm or operation that vary with properties or kinds of a type, but retain the same name. This idiom enables library designers to minimize programming effort while preserving the semantics of an abstraction.

There are four components to the tag dispatch idiom: a set of tag classes, a map function, the dispatcher, and the algorithm implementations. A *tag class* is a (typically) empty class that describes a kind of type or a property of a type. Tag classes are often organized inheritance hierarchies (called tag hierarchies) in order to further classify types or their properties. The *map function* is a function or metafunction that maps a type to its most specialized (derived) tag class. The dispatch framework is comprised of a set of implementations and a dispatcher. Each implementation corresponds to a tag class in the tag hierarchy, and takes that tag class as a function parameter. The dispatcher invokes an implementation by constructing an object of a tag class returned by the map function. The C++ overloading rules require the compiler to select the implementation corresponding to the tag defined for the given types.

Another popular approach to tag dispatch is to use constant metafunctions (e.g. true_ and false_) as tag classes when there are only two variants of the algorithm. In this
approach, the map function is simply a Boolean metafunction that evaluates some property of the given types.

### 3.3.1 Example: STL Iterator Advance

One of the best known examples of tag dispatch is the STL `advance` operation for iterators. This example describes how the algorithm is implemented using the tag dispatch idiom. The STL defines a tag hierarchy for iterator, which is shown in Figure 7.

```cpp
struct input_iterator_tag { }
struct forward_iterator_tag : input_iterator_tag { }
struct bidirectional_iterator_tag : bidirectional_iterator_tag { }
struct random_access_iterator_tag : random_access_iterator_tag { }
```

**Figure 7.** The STL tag class hierarchy classifies iterators based on their traversal operations and properties.

Here, the STL tag class hierarchy defines four distinct classifications of iterators based on their traversal properties—how an iterator moves along or consumes a sequence of objects. Each kind of iterator is represented by a tag class in this hierarchy. An input iterator allows forward traversal through a sequence. A forward iterator is a multi-pass input iterator (the sequence is not consumed during traversal). A bidirectional iterator can also traverse backwards in the sequence, and a random access iterator can move multiple steps in either direction in constant time (moving \( n \) steps forward or backwards is a single operation, not \( n \)).

These tag classes are associated with an iterator type through the `iterator_category` function, whose implementation is shown in Figure 8.
template <typename Iter>
typename Iter::category iterator_category(Iter) {
    return typename Iter::category();
}

template <typename T>
random_access_iterator_tag iterator_category(T*) {
    return random_access_category_tag();
}

Figure 8. The iterator_category function maps an iterator type to its corresponding tag class.

There are two overloads of the iterator_category function. The first is a generic operation and relies on the Iter class to provide an associated type: category. This associated type explicitly denotes the iterator’s category. When the template is instantiated, the return type is replaced by the iterator’s associated tag class. The second overload is partially specialized for pointer types, which do not (and cannot) define a nested type. This explicitly defines all pointer types to be in the random access iterator category.

The advance operation moves an iterator by n steps. If n is negative and the iterator supports backwards traversal, then advance will cause the iterator to retreat along its sequence. For non-random access iterators, this operation is O(n). Because random access iterators support constant-time “jumps”, the operation is O(1) for that category. The three implementations of advance are shown in Figure 9.
template <typename Iter>
void advance(Iter& i, size_t n, input_iterator_tag) {
    assert(n >= 0);
    while(n-- ) ++i;
}

template <typename Iter>
void advance(Iter& i, size_t n, bidirectional_iterator_tag) {
    if(n > 0) while(n-- ) ++i;
    else while(n++) --i;
}

template <typename Iter>
void advance(Iter& i, size_t n, random_access_iterator_tag) {
    i += n;
}

Figure 9. Three implementations of the advance function specialize the semantics and performance of the operation for different kinds of iterators.

Each variant takes a tag class as a parameter. The first variant of advance is implemented for the “weakest” iterator—input iterators. Since input iterators only allow traversal in a forward method, this operation has a precondition \( n \geq 0 \). The second variant is defined over bidirectional iterators which do allow backwards traversal and so \( n \) may be less than 0. Finally, the third variant for random access iterators simply moves \( i \) by \( n \) steps in a single operation. These functions are dispatched by another implementation of advance, shown in Figure 10.

Figure 10. The advance algorithm dispatches one of its variants by invoking an overload on the result type of iterator_category function.

This implementation of advance is the dispatcher. It simply delegates the call to one of its variants. Which variant is determined at compile time based on overload
resolution on the result type of the `iterator_category` function. Using the dispatch mechanism, a library use can call `advance` on any iterator and have it perform correctly and optimally.

### 3.3.2 Example: Boost GIL

Tag dispatch is also frequently used with constant metafunctions (esp., `true_` and `false_`) when the number of dispatches is relatively small, and the cases are unrelated, as with the tag hierarchies above. The Boost Generic Image Library (Boost.GIL) uses this technique extensively for dispatching between two cases of a generic operation. For example, consider an abbreviated implementation of the `fill_pixels` algorithm, shown in Figure 11, which fills a pixel view (image) with a given value.

```cpp
template <typename Iter, typename Val>
void fill_aux(Iter f, Iter l, Val const& x, true_);

template <typename Iter, typename Val>
void fill_aux(Iter f, Iter l, Val const& x, false_);

template <typename View, typename Value>
void fill_pixels(View const& img, Value const& val) {
    if(img.is_id_traversable()) {
        fill_aux(img.begin().x(), img.end().x(), val,
        is_planar<View>());
    } else {
        ...
    }
}
```

**Figure 11.** The `fill_pixels` algorithm dispatches the algorithm to one of the two auxiliary functions based on the planarity of the view, which is decided by the metafunction `is_planar`.

The `fill_pixels` algorithm dispatches the fill process to one of the two auxiliary fill algorithms based on the results of the `is_planar` metafunction. The two variants each take an object of type `true_` or `false_`, which despite being instances of
the same template are unrelated types. Because the is_planar metafunction is ultimately derived from either true_ or false_, constructing an object of that type will cause the compiler to select the appropriate overload.

3.4 Traits Class

A traits class is a class template that adapts a type to a generic abstraction (or concept) by providing associated types or even static functions. A traits class decouples the access of these elements from implementation types, allowing better interoperability in a generic context. A traits class typically has two components: a generic specification that implements the adaptation for a set of conforming data types and a set of specializations that adapt built-in or externally defined types to the same concept.

3.4.1 Example: STL Iterator Traits

The canonical example of traits classes is the STL’s iterator_traits. The iterator_traits class makes it possible to write algorithms in terms of a generic iterator abstraction by providing a mechanism for adapting arbitrary types to the iterator concept. The sole purpose of this adaptive technique is to decouple the access of associated types from the iterator. This is required for pointer types, which do not define nested types (e.g., value_type, reference, pointer, etc.).

The generic iterator_traits class implements an adaptor for conformant types – any iterator implementation can be made to conform to the iterator concept by defining the correct associated types. Its definition is shown in Figure 12.
Figure 12. The `iterator_traits` class derives a number of different associated types for its `Iter` template parameter.

The `iterator_traits` class acts as a kind of façade for a number of different type accessors on its `Iter` template parameter. It derives the names of these types by accessing them directly. In some cases, however, it is not possible for an iterator implementation to define these associated types. If, for example, the iterator is a pointer type (e.g., `int*`), instantiating this template over that type will lead to compiler errors. In this case, the traits class is specialized to accommodate the adaptation. An `iterator_traits` specialization for pointers is shown in Figure 13.

Figure 13. The partial specialization of `iterator_traits` for pointers (`T*`) adapts pointers to the STL iterator concept.

This specialization defines types associated with pointers that cannot be nested within the type. This enables generic algorithms to be written abstractly in terms traits rather than specific types. For example, a reference to
iterator_traits<Iter>::difference_type will always refer to valid, meaningful type.

3.5 Constrained Polymorphism

There are times, in the design of a generic library, where it becomes useful to enable or disable a subset of generic algorithms or data structures based on the properties of their template parameters [Järvi, Willcock, Lumsdaine 2003]. A library developer could, for example, implement logic that fails to instantiate a template based on some metafunction computation of an algorithm. The same techniques can be used to alter the overload resolution rules by quietly failing the instantiation of one or more overloads in a set. Both techniques are rooted in the use of SFINAE (Substitution Failure Is Not An Error) to reject the instantiation of function overloads and class template specializations without resulting in a compiler error.

Substitution failure is an artifact of the C++ template instantiation mechanism. It states that if an invalid argument or return type is formed during the instantiation of a function template, then the instantiation will be removed as a candidate overload, but an error will not be emitted. Consider, for example, the two following overloads shown in Figure 14.

```cpp
int negate(int n) {
    return -n;
}

template <typename F>
typename F::result_type negate(F f) {
    return -f();
}
```

Figure 14. Two implementations of negate: one that negates an integral value and one that negates the result of a nullary function.
If we make call `negate(1)`, the compiler must consider (or instantiate) all of the overloads in order to determine a best candidate. Instantiating the second overload for function types will result in the formation of the identifier `int::result_type`, which is clearly an error—the type `int` has no nested type names. The SFINAE rule states that in this context, the overload is not an error but the function will not be considered as a candidate.

This technique can be exploited and applied to implement constrained (parametric) polymorphism by using metaprogramming techniques to trigger substitution failures. Constrained polymorphism is predicated upon the use of SFINAE-based enablers and disablers. SFINAE enablers are designed to trigger substitution failures based on the evaluation of Boolean constant expressions or metafunctions and are canonically implemented as a kind of partial metafunction. The definition of `enable_if`, which is somewhat similar to the `if_` metafunction from Figure 6, is shown in Figure 15.

```cpp
template <bool, typename T = void>
struct enable_if_c {
    typedef T type;
};
template <typename T>
struct enable_if_c<false, T> { };

template <typename C, typename T>
struct enable_if : enable_if_c<C::value, T> { };  
```

**Figure 15.** The `enable_if` template implements a SFINAE enabler that can be used to control the instantiation of templates based on the properties of types.

As with the `if_` metafunction, there are two components of this implementation: `enable_if_c` and `enable_if`. The `enable_if_c` facility is a kind of “partial” metafunction; only one of its variants defines a result `type`. Here, if the Boolean
argument is true, type is defined to be the type T (defaulting to void if not given). If the Boolean argument to the template is false, then type is not defined. This missing definition of type is sufficient to trigger a substitution failure, causing the overload or specialization to be excluded as a candidate. In both cases, the Boolean expression or metafunction being evaluated denotes a constraint on its argument. By generalizing the technique to work with metafunctions, we can construct arbitrarily complex constraints on the template parameters of a generic algorithm.

SFINAE enablers are applied to function declarations as modifiers of the return type or as optional function parameters. Two equivalent function definitions are shown in Figure 16.

```cpp
template <typename T>
typename enable_if<is_floating_point<T>, T>::type infinity() { 
    return numeric_limits<T>::inf(); 
}

template <typename T>
T infinity(typename enable_if<is_floating_point<T>>::type* = 0) { 
    return numeric_limits<T>::inf(); 
}
```

**Figure 16. Two equivalent definitions of the infinity function are enabled (instantiable) if T satisfies the is_floating_point metafunction.**

The two alternative function definitions shown in Figure 16 demonstrate how SFINAE enablers are often applied to the design of generic algorithms. The first case implements the enabler as a modification to the result type. Here, if is_floating_point<T> evaluates to true, the type is instantiated as T. If the type trait is evaluated as false, then no nested type is defined and the instantiation generates a substitution failure. The second version operates in exactly the same way,
although the enabler is applied as a function parameter—a pointer to `type` with that
defaults to a null value. As before, if the type trait is `true_`, then `type` is instantiated as
a valid type for a function parameter (void*), otherwise `type` is a substitution failure and
the function is excluded from lookup. In either case, if a substitution failure occurs, the
compiler excludes the function from the candidate set, and the compiler will generate an
error—typically “no matching overload found”.

In cases where enablers are used to conditionally include or exclude a single
function, the enabler can also be written as part of the template parameter list in C++0x
by virtue of function templates accepting default template arguments. For example, we
might also choose to write `infinity` as shown in Figure 17.

```c++
template <
    typename T,
    typename = enable_if<is_floating_point<T>>::type>
T infinity() {
    return std::numeric_limits<T>::inf();
}
```

**Figure 17.** C++0x supports the application of enablers in the template parameter
list if there are no other constrained overloads.

Note that this style of constraint is only effective for a single overload with the
given signature. You cannot use this technique to differentiate between multiple
overloads with the same names and parameters. In those cases, you must apply the
enablers as return type modifiers or optional function parameters.

We note that SFINAE also applies to the specialization of class templates,
although we have found it rarely applied in practice. Also C++0x has extended the rules
for the kinds of expressions that may generate a substitution failure. We discuss applications of these techniques in the following examples.

### 3.5.1 Example: Arbitrary Overloading

SFINAE enablement can be used to affect arbitrary overloading based on the properties of a template parameter. For example, consider two versions of a `sqrt` function shown in Figure 18, one that is implemented for floating point types, the other for integral types.

```cpp
template <typename Int>
Int sqrt(Int n, enable_if<is_integral<Int>>::type* = 0) {
    return int_sqrt(n);
}

template <typename Float>
Float sqrt(Float x, disable_if<is_integral<Int>>::type* = 0) {
    return std::sqrt(x);
}
```

**Figure 18.** The `sqrt` function is overloaded for integral types and non-integral types.

The first overload of `sqrt` is enabled if the template parameter satisfies the `is_integral` type trait, effectively constraining that implementation to a subset of numeric representations. The second overload is `disabled` if the type is integral. This ensures that exactly one overload is selected by an enabler. If multiple instantiations succeed, the compiler will instantiate ambiguous function definitions and generate a compiler error. When used for this style of overloading, it is important to realize that the enablers must define system of mutually exclusive constraints. If one constraint is satisfied, then it must be the case that all others are not satisfied. Note that we could
probably have chosen to constrain the second overload using \texttt{is\_floating\_point} since the two traits theoretically define disjoint subsets of types.

Note that this technique is akin to tag dispatch. With tag dispatch, we rely on the partial ordering of overloads to select an appropriate dispatch based on tag classes or constant metafunctions. In concept-based overloading we use SFINAE to disable functions based on their properties (or concepts).

3.5.2 Example: Origin Concepts

We can also adapt this technique, based on enablement to force the compiler to generate compiler errors when a constraint is not met. This technique was pioneered to implement concept checking for generic libraries [Siek, Lumsdaine 2000; Zólyomi, Porkoláb 2004]. The Origin.Concepts library adapts this technique in its implementation of an emulation of concepts for C++0x. Instead of using an SFINAE enabler, the Origin.Concepts implementation is based upon an \textit{asserter}. The asserter is a metafunction that triggers a static assertion within a concept class if its constraints are not met.

To begin, a concept class is a class template that defines a) a metafunction that evaluates constraints and b) an assertion class that emits an error specific to the constraints being violated (as much as possible). For example, the SameType concept is implemented in Figure 19:
Figure 19. The `SameType` concept class implements concept-checking logic and error message facilities for the Origin.Concepts library.

Here, the `SameType` constraint wraps the metaprogramming and compile-time logic that allows it to be used as a concept class. Instances of this template are called `models`, and represent a constraint on a type. The nested check metafunction determines the validity of those models (i.e., the satisfaction of constraints). The assertion of constraints is accomplished by the `concept_assert` pseudo-metafunction.
template <void (*)(*)()> struct instantiate { };  

template <typename> struct check;  
template <typename Assert>  
struct check<void (*)(Assert)> {  
    static void failed() { ((Assert*)0)->~Assert(); }  
};  

template <typename Model>  
struct require {  
    typedef typename Model::assertion Assert;  
    typedef instantiate<&check<void(*)(Assertion)>::failed> type;  
};  

template <typename... Models> struct concept_assert;  

template <typename Model>  
struct concept_assert<Model>  
: require<Model>  
{ typedef void type; };  

template <typename Model, typename... Rest>  
struct concept_assert<Model, Rest...>  
: require<Model>, concept_assert<Rest...>  
{ typedef void type; };  

Figure 20. The concept Assert framework instantiates a Model’s nested assertion class and its destructor in order to execute a static assertion on a set of constraints.

The framework for implementing such a feature is fairly complex and not remotely idiomatic. The metaprogram works by forcing the compiler to instantiate (by calling) the destructor of a model’s assertion class (referring to Figure 19). Instantiating the destructor causes the static Assert function to evaluate the model’s check metaprogram. If the check results in false, the compiler emits an error that includes the given string. The actual result type is essentially ignored. If a static assertion is triggered, a compiler error is generated, and the metaprogram does not return. If the metaprogram does return, then the constraints have been satisfied.
We can apply assertions instead of enablers to enforce compile-time constraints on algorithms and data structures. For example, the standard find algorithm might be rewritten as shown in Figure 21.

```cpp
template <
  typename Iter, typename T,
  typename = concept_assert<
    InputIterator<Iter>,
    HasEqual<InputIterator<Iter>::value_type, T>
  >::type
Iter find(Iter f, Iter l, T const& x) {
  for( ; f != l; ++f) {
    if(*f == x) return f;
  }
  return l;
}
```

Figure 21. The `find` algorithm uses the `concept_assert` asserter to ensure that its constraints are satisfied prior to instantiation. If the constraints are not satisfied, the compiler will generate a concept-specific error message.

Here, the asserter is applied in the template parameter list of the find algorithm. Because the `concept_assert` template is variadic, we can simply list a sequence of models representing the constraints on the template. If any one model is invalid—its requirements unmet—the compiler will emit a meaningful error message. Like SFINAE enablers, asserters can be applied as modifiers of the return type or as optional functional parameters.

### 3.6 Curiously Recurring Templates

The Curiously Recurring Template Pattern (CRTP) is an inheritance pattern that most frequently used to provide default implementations of common operations for a user-defined type. In this idiom, the base class provides services that frequently depend on the type or operations found in the deriving class. The parameterization of the
inheritance pushes information about the deriving type into the base class, allowing the required operators to be instantiated. CRTP is largely a utility mechanism and is frequently used to simplify the process of adapting types to a known concept.

3.6.1 Example: Boost Operators

The Boost.Operators library makes extensive use of CRTP to provide default implementation of common arithmetic and relational operators. Consider the `less_than_comparable` class template shown in Figure 22.

```cpp
template <typename T>
struct less_than_comparable {
    friend bool operator>(T const& a, T const& b) { return b < a; }
    friend bool operator<=(T const& a, T const& b) { return !(b < a); }
    friend bool operator>=(T const& a, T const& b) { return !(a < b); }
};
```

**Figure 22.** The `less_than_comparable` template provides default implementations of complementary operations that can be derived from the less-than operator.

Here, `less_than_comparable` is the base class template and provides several operator implementations that can be written in terms of `operator<` for the type `T`. Using this base class to define those operations is trivial. For example, consider the `integer` class definition shown in Figure 23.

```cpp
class integer : less_than_comparable<integer> {
    bool operator<(integer const& x) const { ... }
};
```

**Figure 23.** A hypothetical definition of an `integer` class implements ordering operators using the CRTP idiom by deriving from a base class template, parameterized over its own type.
The integer class derives from an instance of `less_than_comparable` that is parameterized over its own type. This mechanism pushes sufficient type information into the base class so that the derived operators can be successfully instantiated.

### 3.6.2 Example: Boost Iterator Facade

CRTP is also used to simplify the implementation of iterator adaptors and façades (e.g., Boost’s `iterator_facade`). This use of CRTP is intended to simplify the creation of new iterator types. Consider an abbreviated implementation of the `iterator_facade` class shown in Figure 24.

```cpp
template <typename Derived, typename Value>
class iterator_facade {
    typedef Value value_type;
    typedef Value& reference;
    typedef Value* pointer;
    Derived* self() { return static_cast<Derived*>(this); }
    Derived& operator++() { self()->increment(); return *self(); }
    Derived& operator--() { self()->decrement(); return *this; }
    reference operator*() { return self()->dereference(); }
    pointer operator->() { return &(self()->dereference()); }
};
```

**Figure 24.** An abbreviated implementation of Boost’s `iterator_facade` uses CRTP to adapt a user-defined type to an iterator concept.

There, the `iterator_facade` is parameterized over a number of template arguments, the first of which is a user-defined class that implements some form of
iterator functions (this is the CRTP parameter). The second parameter defines the value type of the iterator, which is used to determine the reference type. The most important feature of this class is the self function, which statically downcasts this object to the type of its derived class—it is easy to think of self as a variant of the this pointer that refers to the complete object type. The required iterator interface, the operators ++, --, and *, and -> are provided by the façade. The Derived class, however, must implement the functions increment, decrement, and dereference. These functions define the protocol over which the adaptation occurs.

Using the façade trivializes the implementation of iterators. Consider the example in Figure 25, which adapts a list node into a bidirectional iterator using the iterator façade.

```cpp
template<typename T>
struct node_iterator
 : iterator_facade<node_iterator<T>, T>
{
    list_node<T>* node;
    void increment() { node = node->next; }
    void decrement() { node = node->prev; }
    T& dereference() { return node->data; }
};
```

Figure 25. The node_iterator class adapts a list_node to an iterator by using the CRTP-based iterator_facade and implementing the requisite protocol.

Here, the node_iterator class is an adaptor that adapts the list_node class to the iterator concept. This is done by deriving from the iterator_facade, parameterizing it over its own type, and the value type T.
CHAPTER 4
Implementation

To support this work, we developed tools to automatically analyze C++ generic libraries and extract programming language idioms from the source code. This fingerprinting technique is predicated on the notion that each idiom has unique layout characteristics within the source code.

4.1 Fingerprinting Framework

The fingerprinting framework is implemented on top of the srcML markup language for C++. SrcML is a lightweight parsing and analysis tool which transforms source code into a lexical XML markup. This source code representation facilitates the rapid development of reverse engineering and program analysis tools. The fingerprinting framework itself is implemented in Python, using the minidom module to load and parse (import) the translated srcML markup files. A simplified overview of the fingerprinting framework can be seen in Figure 26.

![Figure 26. The architecture of the fingerprint framework system.](image)

Once imported, the source code is examined to find simple yet important entities that would be needed later on to identify patterns and idioms. Examples of these entities
are language features such as classes, structures, functions, templates, arguments and line numbers. Because they make up the basis of idioms and patterns each class, structure and function was stored into respective container objects. Once stored, corresponding attributes such as template, template arguments, function arguments, public and private bodies, and line numbers were then associated to their respective possessors. After the initial storage and association of core components are complete, they are ready to be analyzed by carefully constructed idiom and pattern fingerprinting algorithms. These fingerprinting algorithms determine from the entity’s attributes, relations and unique source code layout what type of idiom or pattern they may be classified as. Once analyzed, each entity and its classification and attributes are stored in a relational database for later lookup.

The most prominent reason for choosing our lightweight approach was because of the syntactical architecture present in idioms. This syntactical architecture is their defining feature unlike the relational structure found in most design patterns. In order to find these strict syntax rules the underlying source code must be present for identification. Through the use of srcML we were still able to access every needed source code attribute required for idiom identification.

This lightweight, heuristic approach is both novel and effective, but not without weaknesses. As with any reverse engineering tools, there are fragments of the original system that get lost or altered during phase transitions. By its nature srcML defines a lightweight parsing approach, which is known to generate incorrect markup in cases of ambiguous or complex syntax (which is unfortunately a hallmark of C++ generic
libraries). In the instance where source code is incorrectly marked up, the framework simply loses the information contained within that section of source code. Fortunately, we have found that this is infrequent and losing a few classes or structures does not dramatically affect the overall results.

Another problem with our lightweight approach on generic libraries is the extensive use of preprocessing code in generic libraries. Unless the fingerprint framework is modified for a particular library’s preprocessors syntax, the probability of inaccurate results increases dramatically. For example, the code from the STL (shown in Figure 27) illustrates how the preprocessor is used to specialize the _Is_integer type trait.

```cpp
__STL TEMPLATE NULL struct _Is_integer<bool> {
  typedef __true_type _Integral;
};
```

**Figure 27. An example from STL type_traits.h using the preprocessor on a metafunction which abstracts the template instantiation.**

This style of preprocessor usage makes a syntactic classification virtually impossible for the idiom without first preprocessing the source code or expanding the macro during parsing—neither technique was considered for this implementation.

### 4.2 Idiom Identification

The implementation and heuristics of each of the fingerprinting modules is described in the following subsections. The description of each method is accompanied by a source code example that highlights the particular aspects of the program text that are used as beacons to identify the idiom. For example, the fingerprinting framework implements a number of linguistic classifiers, which capture information about the kinds
of elements found in a program. One such trivial classifier is used to recognize classes. The source code relevant to that task is shown in Figure 28.

```cpp
class foo { 
    void do_something(){ };
};
```

**Figure 28.** The `class` keyword is highlighted to indicate that a fingerprint method uses this syntax to derive information about the program element.

The classifier responsible for identifying class elements considers the use of the `class` keyword to be indicative of the declaration or definition of a class element within the language. Other linguistic classifiers are used to identify structures, functions, templates, overloads, and template specializations.

### 4.2.1 Functor

The functor pattern is a very simple idiom to fingerprint. First, structures and classes are chosen for analysis as candidates of the functor idiom. Both templated and non templated classes and structures can be used for this classification. If the class or structure contains a function overload of the call operator it is determined to be a functor.

```cpp
struct std_fill_t {
    template <typename It, typename P>
    void operator()(It first, It last, const P& p_in) {
        std::fill(first,last,p_in);
    }
};
```

**Figure 29.** Functors are recognized as classes with a call operator overload.

### 4.2.2 Metafunctions

The metafunction fingerprint occurs often in generic libraries and can require multiple signature confirmations. First, structures and classes are chosen for analysis as candidates of the metafunction idiom. Only templated classes and structures are accepted
for this classification. Next, the body of the entity is examined for typedefs or enumerations. During this check, it is valid if multiple typedefs and enumerations occur.

In order for a type metafunction to occur, at least one of the corresponding typedefs must set a type value. In order for an integral metafunction to occur an enumeration or Boolean value must be set. If either one of these signatures occur it can be determined that the class or structure in question is a metafunction.

```cpp
template <typename Graph>
struct vertex_property_type {
    typedef typename Graph::vertex_property_type type;
};
```

Figure 30. Type metafunctions are recognized as class or structure templates with a typedef named `type`.

```cpp
template <typename T, typename U>
struct is_same {
    static bool const value = false;
};
```

Figure 31. Integral metafunctions are recognized as classes or structures with Boolean return types.

4.2.3 Tag Dispatch

In order to be able to fingerprint the tag dispatch idiom, it is first required that all tag classes are identified. In order to classify tag classes, only non templated structures are chosen for candidates. Although inheritance is common among tag classes, it is not a requirement so this is not taken into account during detection. Next, the body of the candidate is examined to determine if it is completely empty. If this is the case then a tag class has been identified.
struct input_iterator_tag { };  
struct forward_iterator_tag : input_iterator_tag { };  
struct bidirectional_iterator_tag : bidirectional_iterator_tag { };  
struct random_access_iterator_tag : random_access_iterator_tag { };  

Figure 32. Tag classes are recognized as structures with empty bodies.

Once all tag classes have been identified, it is then possible to check for the tag dispatch fingerprint. For tag dispatch, all the templated and non templated functions are considered for candidates. After these candidates have been identified, it is then determined if these candidates utilize argument overloads. If a candidate does utilize argument overloading, its argument list is examined for tag classes or the special case where a type check is used as an overload. If one of these happens to be true, it can be determined that the function utilizes tag dispatch.

```c++
template <typename Iter>
void advance(Iter& i, size_t n, input_iterator_tag) {
    assert(n >= 0);
    while(n--) ++i;
}

template <typename Iter>
void advance(Iter& i, size_t n, bidirectional_iterator_tag) {
    if(n > 0) while(n--) ++i;
    else while(n++) --i;
}
```

Figure 33. Tag dispatch is recognized from overloaded functions with tag classes or type checks as argument types.

4.2.4 Traits Class

The traits class fingerprint is very similar to the type metafunction fingerprint but requires less signature confirmations. First, structures and classes are chosen for analysis as candidates of the traits class idiom. Only templated classes and structures are accepted
for this classification. Next, the body of the entity is examined for only typedefs. During this check, it is valid if multiple typedefs occur. The only stipulation for these typedefs is that not one sets a type value. If this is the case then a traits class idiom has been identified.

```cpp
template <typename Iter>
struct iterator_traits {
    typedef typename Iter::value_type value_type;
    typedef typename Iter::reference reference;
    typedef typename Iter::pointer pointer;
    typedef typename Iter::category category;
};
```

Figure 34. Traits class is recognized from classes or structures with multiple typedefs where none of them set the type.

4.2.5 Constrained Polymorphism

Constrained polymorphism refers to the SFINAE idiom. First, structures and classes are chosen for analysis as candidates of the SFINAE idiom. Only templated classes and structures are accepted for this classification. Next, the argument list of the template is examined. Within this list a template parameters value must be set to void. If this check is valid then the SFINAE idiom is identified.

```cpp
template <bool, typename T = void
struct enable_if_c {
    typedef T type;
};
```

Figure 35. SFINAE is recognized from templated classes or structures setting a template parameter type to void.

4.2.6 Curiously Recurring Templates

Fingerprinting the curiously recurring template pattern is fairly straightforward although automation but can be easily missed through manual classification. First, structures and classes are chosen for analysis as candidates of the metafunction idiom.
Both templated and non-templated classes and structures are accepted for this classification. Next, the candidate must inherit from at least one templated base class or structure. The final requirement is that the candidate must be included in the templated base class’s parameter list. If this is the case, then a CRTP idiom is classified.

```cpp
template <typename T>
struct node_iterator
    : iterator_facade<node_iterator<T>, T>
{
    list_node<T>* node;
    void increment() { node = node->next; }
    void decrement() { node = node->prev; }
    T& dereference() { return node->data; }
};
```

**Figure 36.** CRTP is recognized from classes or structures using their own class names in an inherited template parameter.
CHAPTER 5

Validation

In order to verify the correctness of the fingerprint framework two accuracy standards are applied to its output. The first measure precision is a measure of the fingerprint frameworks output to that of manual classification on the same source code. The manual work involved in this measure is extensive because of the time it takes to analyze and classify all of the idioms by hand. On top of this manual classification, the data from both sets must then be physically compared. Once both sets of comparison data are composed, precision is computed though the intersection of both sets and then bisected from the manual set. The equation for precision can be observed in Figure 37.

The other accuracy standard presented in this research is recall. Recall requires less manual work then the proceeding standard. For recall, false positives of the automated classification are counted by manually examining the automation results. The calculation for recall is derived from the intersection of the automated results and the manual check against them bisected by the automated results. The equation for this can also be observed in Figure 37.
The following subsection presents both precision and recall statistics for the fingerprint framework against the Boost.Graph library. Subsection 5.2 presents recall accuracy for a variety of generic libraries.

5.1 Precision and Recall

In order to calculate recall, an empirical study was conducted against the Boost.Graph library. In this empirical study, the majority of the source code from Boost.Graph was manually analyzed for design patterns and idioms and stored into a wiki for convenient manual lookup. The relevant classifications from this empirical study were functors, metafunctions, tag classes, tag dispatch, traits classes, SFINAE and CRTP. Along with recall, precision on the Boost.Graph library was also conducted to provide a very thorough accuracy analysis.
Table 1. Precision and recall statistics for the Boost::Graph library.

<table>
<thead>
<tr>
<th></th>
<th>Study</th>
<th>Auto</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functor</td>
<td>60</td>
<td>51</td>
<td>100%(51)</td>
<td>92%(55)</td>
</tr>
<tr>
<td>Metafunction</td>
<td>125</td>
<td>123</td>
<td>100%(123)</td>
<td>88%(110)</td>
</tr>
<tr>
<td>Tag Class</td>
<td>63</td>
<td>63</td>
<td>100%(63)</td>
<td>100%(63)</td>
</tr>
<tr>
<td>Tag Dispatch</td>
<td>14</td>
<td>12</td>
<td>100%(12)</td>
<td>85%(12)</td>
</tr>
<tr>
<td>Traits Class</td>
<td>23</td>
<td>20</td>
<td>100%(20)</td>
<td>86%(20)</td>
</tr>
<tr>
<td>SFINAE</td>
<td>1</td>
<td>1</td>
<td>100%(1)</td>
<td>100%(1)</td>
</tr>
<tr>
<td>CRTP</td>
<td>31</td>
<td>34</td>
<td>100%(34)</td>
<td>100%(31)</td>
</tr>
</tbody>
</table>

In the case of precision, the accuracy was very high, with the lowest being 100%. This implies that the classifications of the fingerprint program are rarely false positives. In other words, when the fingerprint framework classifies an idiom it is rarely a misclassification.

On the other hand, the recall statistics are not nearly as high. This implies that the fingerprint framework did not classify some valid idioms at all. There are many factors that may have lead to these results. The first is human error in the empirical study. It is completely possible that some of the classifications done by hand were wrong. After all,
labeling the over 30K lines of code from Boost.Graph can be a very exhausting and error prone task. Another explanation for this is preprocessor and srcML problems described in the implementation section of this research. For the preprocessed source code, it is still possible to identify idioms by hand. One last interesting note is how we were able to improve the empirical study’s results by finding unclassified idioms which were missed by hand and found in the automated approach.

5.2 Overall Precision

Having established a measure of precision and a measure of recall in the first experiment, we conducted a broader experiment to help further validate the accuracy of the technique. We extracted idiom and pattern counts from a larger number of libraries and manually verified though precision only, that each of the extracted idioms was truly represented in the source code. This precision only approach allowed a means of validating large data sets.

For the following precision table, both the Boost.Gil and Boost.Python libraries were examined. For each library, false positives were manually identified and tallied to calculate their corresponding precision scores.

The results of this precision calculation showed high results in all cases. These high scores collaborate with the above study seen in Table 1 that the fingerprint framework does not commonly generate false positives.
Table 2. Precision statistics for the Boost gil, and python libraries.

<table>
<thead>
<tr>
<th></th>
<th>Gil</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functor</td>
<td>100% (87)</td>
<td>100% (23)</td>
</tr>
<tr>
<td>Metafunction</td>
<td>100% (114)</td>
<td>100% (55)</td>
</tr>
<tr>
<td>Tag Classes</td>
<td>90% (9)</td>
<td>80% (12)</td>
</tr>
<tr>
<td>Tag Dispatch</td>
<td>94% (35)</td>
<td>100% (43)</td>
</tr>
<tr>
<td>Traits Class</td>
<td>96% (63)</td>
<td>100% (15)</td>
</tr>
<tr>
<td>SFINAE</td>
<td>100% (1)</td>
<td>100% (2)</td>
</tr>
<tr>
<td>CRTP</td>
<td>100% (90)</td>
<td>100% (41)</td>
</tr>
</tbody>
</table>
CHAPTER 6

Results

In the study of these libraries, we primarily investigated two aspects of their implementation: their composition with respect to language features supporting generic programming and their use of idioms for generic programming. The results presented in this section are derived from the analysis of the libraries given in Table 3.

Despite being part of the same basic umbrella project, each Boost library is essentially a standalone product with different authors, designs, and purposes. In this sense, we regard Boost as a collection of independent generic libraries rather than a single uniform library.

The Boost libraries listed above constitute a subset of the Boost C++ Libraries that were actually studied. We selected these libraries because they exhibited interesting properties with respect to the idioms studied. In some cases, problems in the translation of source code led to inaccurate results. In other cases, none or very few of the idioms were represented (e.g. Boost.Filesystem).

Unlike other large libraries such as Boost, CGAL’s source code structure is not subdivided into separate libraries. Because of CGAL massive girth in comparison to other generic libraries as a whole, only CGAL’s core components were considered in comparison scenarios. With most of its important functionality being contained within this core, it served as an adequate subset to effectively represent CGAL.
Table 3. Libraries examined in the empirical study.

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boot.Accumulators</td>
<td>A framework for incremental statistical computation</td>
</tr>
<tr>
<td>Boost.Algorithm</td>
<td>A collection of algorithms, especially on strings</td>
</tr>
<tr>
<td>Boost.Bimap</td>
<td>A data structure for bidirectional mappings</td>
</tr>
<tr>
<td>Boost.Function</td>
<td>Wrappers for function pointers</td>
</tr>
<tr>
<td>Boost.FunctionTypes</td>
<td>Type traits describing elements of functions</td>
</tr>
<tr>
<td>Boost.Fusion</td>
<td>Generic algorithms on heterogeneous data types</td>
</tr>
<tr>
<td>Boost.GIL</td>
<td>Generic algorithms and data structures for image processing</td>
</tr>
<tr>
<td>Boost.Graph</td>
<td>Generic graph algorithms and data structures</td>
</tr>
<tr>
<td>Boost.Interprocess</td>
<td>A library for interprocess communication and synchronization</td>
</tr>
<tr>
<td>Boost.Intrusive</td>
<td>A library to support building and adapting intrusive containers</td>
</tr>
<tr>
<td>Boost.Iterator</td>
<td>Tools and adaptors for building and operating on iterators</td>
</tr>
<tr>
<td>Boost.Lambda</td>
<td>A library for constructing lambda expressions</td>
</tr>
<tr>
<td>Boost.Math</td>
<td>A collection of mathematical algorithms and concepts</td>
</tr>
<tr>
<td>Boost.MPI</td>
<td>A wrapper around the MPI specifications</td>
</tr>
<tr>
<td>Boost.Numeric</td>
<td>Generic facilities for numerical analysis</td>
</tr>
<tr>
<td>Boost.Proto</td>
<td>A generic framework for DSEL construction</td>
</tr>
<tr>
<td>Boost.Python</td>
<td>A library for C++ and Python interoperability</td>
</tr>
<tr>
<td>Boost.Range</td>
<td>Adaptors and data types for iterator ranges</td>
</tr>
<tr>
<td>Boost.Spirit</td>
<td>A DSEL library for parser construction</td>
</tr>
<tr>
<td>Boost.TypeTraits</td>
<td>Metaprogramming facilities for basic types</td>
</tr>
<tr>
<td>Boost.eXpressive</td>
<td>A DSEL library for regular expression construction</td>
</tr>
<tr>
<td>STL</td>
<td>A library of container classes, algorithms and iterators</td>
</tr>
<tr>
<td>Loki</td>
<td>A library of designs containing common design patterns and idioms</td>
</tr>
<tr>
<td>CGAL</td>
<td>A computational geometry algorithm library</td>
</tr>
<tr>
<td>Soci</td>
<td>A database access library for SQL queries</td>
</tr>
</tbody>
</table>
6.1 Library Composition

The first aspect of the systems studied is their composition with regard to the use of language features that support the generic paradigm. This includes the use of templates, specialization, and overloading. A survey of these features is intended to provide some feel for the extent to which these language features are used in various domains and implementations. The counts of classes and functions, and the percentages of those that are templates are shown in Figure 38 and Figure 39 respectively.

Figure 38 shows the raw counts of functions and classes extracted from each library. Libraries that provide more classes than functions are typically more engaged in providing support for generic programming or adaptation. For example, the Boost xPressive, Lambda, and Range libraries provide substantial metaprogramming support for compile-time regular expressions, lambda expressions, and a range abstraction for iterators. Conversely, libraries such as Numeric, Math, Interprocess, and Graph are primarily concerned with runtime functionality.

Figure 39 shows the percentage of classes and functions that are actually templates. Not surprisingly, the overwhelming number of elements are actually templates. However, it is interesting to note that libraries typically have a higher percentage of class templates than function templates. One reason for this is the use of classes to implement several idioms, especially metafunctions, tags, and traits.
Figure 38. Counts of classes and functions were extracted from the studied libraries.

Figure 39. The percentage of classes and functions that are defined as templates.
Figure 40 shows the usage of class template specialization and function overloading per library. These language features are studied because they allow mechanisms for selecting alternative implementations based on type.

The use of class template specialization and function overloading in a library indicates the degree to which the library’s elements are customized by user-provided types or properties of those types. The use of function overloading is more prevalent by far. On average 45% of all functions are overloads. Specialization is primarily used in metafunction implementations and traits class specializations. On average 36% of classes are involved in a specialization.
Figure 40. Percentages of specialized class templates and overloaded functions.
6.2 Use of Idioms

In the second aspect of the study, we collected counts of idioms from each of the libraries selected. Based on these counts, libraries are manually classified by the kinds of “signatures” they show with respect to those counts. This process identifies three distinct groups or kinds of libraries: generic, concept, and extension libraries. It is important to note that the classification process that identified these groups is not supported via statistical, cluster, or other automated analyses. Verification these classes is future work.

6.2.1 Generic Libraries

Generic libraries are identified by relatively high numbers of tag classes, traits classes, and the use of tag dispatch. These idioms are the basis of the techniques that define concepts for generic libraries. The idioms found in such libraries support the decoupling of algorithms from data types. Traits classes and metafunctions are often used to provide adaptation support for user-defined or application-specific data types.
Table 4. Generic libraries provide sets of generic algorithms and data structures described by concepts. The use of tag classes, tag dispatch, and type traits directly support this.
These libraries can be classified as generic libraries because of their extensive use of idioms in order to specialize its generic structure. This specialization requires the use of a variety of different idioms in order to resolve specific problems.

6.2.2 Concept Libraries

A concept library is a library that provides metaprogramming support for one or more concepts or a concept hierarchy. Metaprogramming in these libraries is primarily to provide adaptive support for the concepts they define. Concept libraries are primarily recognized by higher numbers of metafunctions and traits classes, and relatively low numbers of behavioral or structural elements (e.g., functions, CRTP instances, etc.).

6.2.3 Extension Libraries

Extension libraries typically contain a set of algorithms or data structures that are intended to extend the domain of an existing library. Typically, these libraries contain little more than additional data structures or algorithms that may be useful when used in conjunction with another library. Here, high numbers of tag dispatch denotes the presence of generic algorithms. Also, large numbers of functors also denote large numbers of behavioral features. Generic data structures also fall into this category, usually because their implementations are rooted in the definition of algorithms operating on a core structure.
Table 5. Concept libraries provide metaprogramming support in the form of metafunctions and traits classes.

<table>
<thead>
<tr>
<th>Boost.Acumulators</th>
<th>Boost.FunctionTypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFINAE 1</td>
<td>Traits Class 14</td>
</tr>
<tr>
<td>Tag Dispatch 2</td>
<td>Tag Class 36</td>
</tr>
<tr>
<td>Functor 0</td>
<td>Metafunc 33</td>
</tr>
<tr>
<td>CRTP 74</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Boost.Iterator</th>
<th>Boost.Lambda</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFINAE 0</td>
<td>Traits Class 2</td>
</tr>
<tr>
<td>Traits Class 2</td>
<td>Tag Dispatch 12</td>
</tr>
<tr>
<td>Tag Class 5</td>
<td>Tag Class 36</td>
</tr>
<tr>
<td>Functor 3</td>
<td>Metafunc 36</td>
</tr>
<tr>
<td>CRTP 7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Boost.Range</th>
<th>Boost.TypeTraits</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRTP 4</td>
<td>Traits Class 12</td>
</tr>
<tr>
<td>SFINAE 0</td>
<td>Tag Dispatch 0</td>
</tr>
<tr>
<td>Tag Class 6</td>
<td>Tag Class 1</td>
</tr>
<tr>
<td>Functor 2</td>
<td>Metafunc 79</td>
</tr>
<tr>
<td>CRTP 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Boost.TypeTraits</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SFINAE 0</td>
<td>Traits Class 12</td>
</tr>
<tr>
<td>Tag Dispatch 0</td>
<td>Tag Class 1</td>
</tr>
<tr>
<td>Tag Class 1</td>
<td>Metafunc 106</td>
</tr>
<tr>
<td>Functor 0</td>
<td></td>
</tr>
</tbody>
</table>

0 10 20 30 40 50 60 70 80
Table 6. Extension libraries extend an existing domain by providing new algorithms or data structures.
Note that Boost.MPI could legitimately be classified as its own generic library due to the large number of traits classes. This indicates that the MPI library defines a substantial domain with new concepts. However, the large number of algorithms suggests to us that it is an extension of another library.

6.2.4 Unclassified Libraries

It is important to note that a large number of libraries not reported in this study fall into this category. Specifically, libraries that did not exhibit any (or very few) of the idioms reported here could be classified as either simple generic libraries or extension libraries. For example, the Boost.Filesystem can be considered a generic library even though it exhibits very few of the idioms presented here. In this case, the interoperation between algorithms and data types is trivial and largely unconstrained. As a result, we could not classify the library.
7.1 Measuring Library Quality

It has been suggested that judging source code on its use and distribution of design patterns that it may actually constitute a higher quality of the code [Balanyi, Ferenc 2003]. However, the use of design patterns requires a higher degree of knowledge and expertise. This raises the question… are generic libraries that make extensive use of idioms in some way better then ones that use less?

7.2 Evolving Generic Libraries

The idioms examined in this research do provide a method for architecting generic libraries, and the utilization of these idioms requires an expert understanding to be correctly used. With this being said, it could be assumed that generic libraries with heavy utilization of idioms are designed by more experienced programmers who are more effective at proper and qualified design.

According to our research results, the fact that cannot be disputed is that these idioms are heavily used in generic libraries. Whether or not the use of these idioms proves a better design or ultimately hinders program comprehension and maintainability due to their complex syntax is irrelevant. The fact is that they are being adopted in substantial amounts of libraries. This prompts the question, at what point should the language evolve to accommodate and implement these idioms as language features?
7.3 **Motivating Language Evolution**

Our results show that generic programming in C++ is highly idiomatic and once decomposed; these libraries are constructed of idioms instead of language features. These idioms exist because they solve the commonly reoccurring problems that language features cannot.

Because generic libraries are hard to read and understand, tools for explaining them should not be the solution. Developing these automated tools is not a trivial task and only provides substandard results. An idioms mystical syntax and cryptic designs alone should constitute a need for replacement language features.

While these idioms do provide a solution for solving many problems that arise in generic programming, they still have shortcomings and are workarounds for problems that could be easily resolved by new language features. Take for example the proposed introduction of concepts and concept maps in the C++0x standard. The concept and concept map would have provided a means to resolve problems that the traits classes and tag dispatch idioms try to solve with a much cleaner and robust implementation. Many of these idioms are simply hack jobs to resolve problems much more suited for actual language features.
CHAPTER 8
Conclusions

In this paper we discussed the problem of the need for a better understanding of the makeup of generic libraries. We then assembled a catalog of the common idioms present in generic libraries. This catalog description analyzed the construction of, demonstrated, and explained reasons for the use of these idioms.

An automated lightweight reverse engineering approach was then presented which provides an automatic means of identifying the idioms. The tool created for this automated process was then validated against publicly available generic libraries though both though precision and recall accuracy.

Once validated, the tool was then used to analyze much larger generic libraries in order to gain an understanding of their underlying design makeup. It was observed in these results that once decomposed; these libraries were highly idiomatic in nature. It is though this analysis and classification of their idioms that a system for a better understanding these libraries developed.
REFERENCES


Symposium on Document Engineering (DocEng’02), McLean VA, November 8-9, pp. 34-41.


"Concept-Controlled Polymorphism", in Proceedings of 2nd International Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt, Germany, Sep 22-25, pp. 228-244.


