Substitutability

Principles, Idioms and Techniques for C++

Kevlin Henney

khenney@qatraining.com
http://www.qatraining.com

Presented at *JaCC*, Oxford, 16th September 1999.

Kevlin Henney

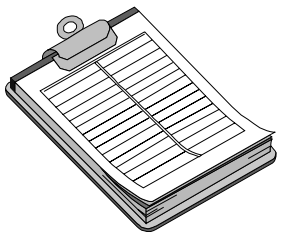
`khenney@qatraining.com`
`kevlin@acm.org`

QA Training

Cecily Hill Castle, Cirencester, Gloucestershire, GL7 2EF, UK
`http://www.qatraining.com`
`http://techland.qatraining.com`
Tel. +44 (0) 1285 655 888
Fax. +44 (0) 1285 650 537

Agenda

- Principles
- Patterns
- Particulars
  - Conversions, Overloading, Derivation, Mutability, Genericity
- Practice



C++ offers a great range of expression that is sometimes obscured by approaching it from a single viewpoint, such as lowest common denomination OO or as a better C. Included in its features is direct support for classic object-oriented inheritance, however in common with many other languages the meaning of such inheritance is beyond the scope of the language. In conventional best practice, an inheritance relationship is taken to mean that there is a substitutable relationship between derived and base classes. Such extension of a core working language through patterns of usage, in effect defining a broader level of dialogue in a system, is common in any discipline.

The principles and features which govern the idiomatic practices of substitutability in C++ go beyond public inheritance, and include overloading, conversions, generic programming, and mutability. Understanding the basic principles and how C++ can support them can make systems more expressive and development more effective.

Principles

Express coordinate ideas in similar form

This principle, that of parallel construction, requires that expressions similar in context and function be outwardly similar. The likeness of form enables the reader to recognise more readily the likeness of content and function.

Strunk and White [Strunk+1979]

C++ provides a context for design and expressing design. It offers a number of mechanisms that may be used to express substitutability concepts. Principles provide a mental framework for developers to work within and take advantage of, where Carolyn Morris defines a framework as "a skeleton on which a model of work is built".

Principles

- Intent
  - Motivation for and concepts describing substitutability
- Content
  - Transparency
  - Hierarchy and heterarchy
  - Polymorphism
  - Value-based programming

4

Using the principle of substitutability as one of the criteria for judging and assembling one's architecture (macro and micro) seems a little removed from the common concerns of architecture, but as a structuring principle it has much to commend it: if one element can be treated transparently as another, then it removes the need for consideration of differences. Or, to be precise, it removes the need for consideration of unnecessary differences. However, the ability to treat things uniformly it is not the same as saying that everything is the same: "when everything is a triangulation point, nothing is a triangulation point" [Jackson1995].

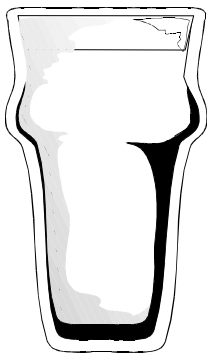
Architectures can organise their elements in many ways, including hierarchically and heterarchically. Such arrangements refer not only to the structural and execution arrangements, but also to classification. It is classification that forms the basis of substitutability.

For OO developers, the most familiar form of substitutability is through inheritance and polymorphism. There is more to polymorphism than simply inheritance, overridden functions, and dynamic binding associated with classes; it proves to be a useful starting point for appreciating the diversity of type systems.

Values, which are objects with significant state but not identity, represent information in a system in different ways to objects with significant identity. They tend to rely more on other forms of polymorphism than the inclusion polymorphism of OO.

Transparency

- Transparency is a property of separation
  - The visibility or intrusion of the connection between usage and implementation
- Uniform treatment of similarities
  - Use of a common interface without concern for detail of implementation
  - Also implies greater clarity and ease of comprehension



5

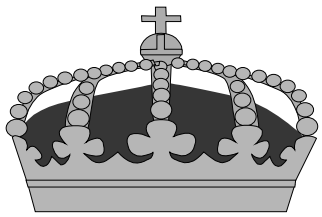
Substitutability is the property that different elements have if they are grouped – and then treated – in terms of their commonality; variations that are not relevant to their use are not a part of this abstraction, or rather they do not dominate. In the context of OO, such transparency normally equates to the use of inheritance and polymorphism, and bound by the Liskov Substitution Principle [Liskov1987, Coplien1992].

C++ is a rich language, supporting many other mechanisms for communicating common usage to both the human and machine reader, in addition to its many forms and uses of inheritance. A fuller exploration of the commonality and variability mechanisms in C++ supports the view that the mechanisms of the language can be used to greater effect than just in support of a subset-OO style [Coplien1999]. The behavioural concepts behind LSP can be generalised to cover all forms of substitutability.

Classification is the basis of substitutability, and this notion is enforced to differing degrees by the compiler and developer. Where one element can be classified as another, or in terms of another, then that element may be used in place of the other element transparently. The degree of substitutability varies across specific mechanisms and problems. Transparency is often not completely crystal clear, and transparent should not be taken to mean "not there". If something is not immediately visible, it may reveal itself under a change of circumstance, e.g. the effect of failure in distributed systems cannot be made totally transparent, and thus distributed programming models cannot seamlessly and completely be made to look like local, sequential programming models.

Hierarchy

- Substitutability hierarchies...
  - Type Hierarchy and Class Hierarchy
- Structural hierarchies...
  - Class Hierarchy, Object Hierarchy and Function Hierarchy
- Execution hierarchies...
  - Object Hierarchy and Function Hierarchy



6

Hierarchies can often be the simplest ways of organising what we know, as [Jackson1995] observes under the heading of *Arboricide*. The word *hierarchy* is used a little more loosely in the following discussion, to include directed acyclic graphs (DAG) which are hierarchical, but not strict hierarchies.

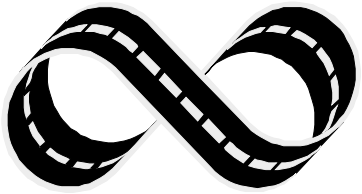
Where type defines the accessible operations and semantics of an object, a Type Hierarchy defines a classification relationship between types based on extension and substitutability. A Type Hierarchy may be realised through a number of mechanisms, including a Class Hierarchy or concatenation of generic requirements. Vertical delegation moves execution responsibility up and down a Class Hierarchy; commitment to implementation can be deferred or delayed down a hierarchy, or the knowledge of detail can be held toward the root. Exceptions are normally partitioned within a Class Hierarchy with respect to classification.

An Object Hierarchy expresses a structure over which horizontal delegation acts, with significant objects cascading responsibility – and being defined in terms of – other objects that are identifiably more primitive. A common example of Object Hierarchy is Handle/Body [Coplien1992]. This can be mixed with Class Hierarchy to give Handle/Body Hierarchy (also known as Bridge [Gamma+1995]).

A Function Hierarchy for distributing execution across elements may exist inside, outside or across class or source file boundaries. A Function Heterarchy supports a mix rather than flow of levels, possibly involving recursion, in call chains.

Heterarchy

- A heterarchical configuration mixes relationships between levels
  - Type Heterarchy accommodates cyclic schemes of classification and substitutability
  - Function Heterarchy for recursive execution
  - Object Heterarchy includes cycles between objects for sharing responsibility



7

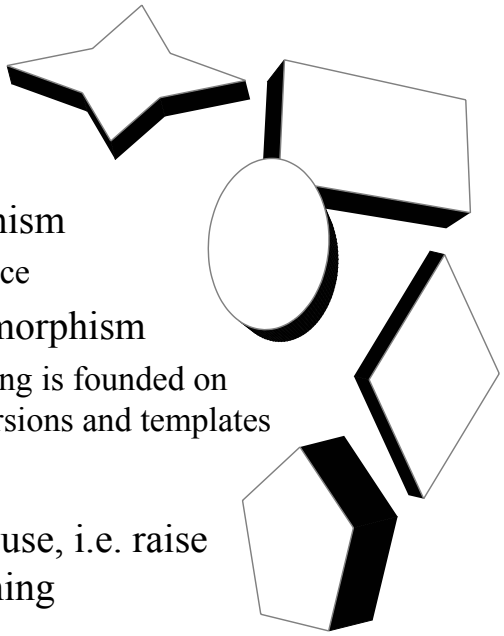
However, hierarchies are not suitable for every description, as [Jackson1995] notes under *Hierarchical Structure*. But the fact that hierarchies do not cover all eventualities does not mean that the alternative structures are chaotic; before a descent into arbitrary categorisation and structure, the notion of *heterarchy* [Hofstadter1979] provides a useful way of describing certain relationships that have natural cycles between elements, such as recursion:

> *A program which has such a structure in which there is no single "highest level"... is called a* heterarchy *(as distinguished from a hierarchy).*

In a heterarchy there may be localised regions of hierarchy and layering, but these can, like inverted geological strata, be folded back over themselves in a self referential way. In a heterarchy, one element defines the context for another, which in turn (directly or indirectly) defines a context for the first element.

A Type Heterarchy is an example, where types can be defined as readily convertible between each other. This is perhaps not always wise, as in the two way implicit conversion relationship between `int` and `double` in C++, but a heterarchy does provide a means of describing such relationships, and there are cases where it can be useful. Recursion, either single or mutual recursion, provides a common example of a Function Hierarchy. An Object Hierarchy defines a set of object relationships where there is no definite root. The Mutual Registration pattern [Henney1999a] illustrates a collaboration that is heterarchical in both its object relationships and its execution.

Polymorphism



- Mechanisms
  - Runtime polymorphism
    - Related to inheritance
  - Compile time polymorphism
    - Generic programming is founded on overloading, conversions and templates
- Substitutability
  - Offers guidance on use, i.e. raise mechanism to meaning

One traditional pillar of OO is its foundation on the idea of polymorphism as a runtime binding based on inheritance. C++ goes further than just its `virtual` function and derivation mechanisms, also offering compile time polymorphism [Koenig+1995] in the form of conversions, overloading and templates. Polymorphism has been broadly classified along other lines [Cardelli+1985]:

- *Inclusion polymorphism* is the standard OO model of polymorphism.
- *Parametric polymorphism* is based on the provision or deduction of extra type information that determines the types used in a function.
- *Overloading polymorphism* is based on the use of the same name denoting different functions, with selection based on the context of use, i.e. argument number and type.
- *Coercion polymorphism* is based on the conversion of an object of one type to an object of another based on its context of use.

From one perspective, polymorphism describes a set of mechanisms. To use these mechanisms to be best effect, they must be associated with intent, i.e. they come to represent and express certain concepts. This view can be seen in the concept of types as meaning rather than mechanism [Cardelli+1985]:

> *As soon as we start working in an untyped universe, we begin to organize it in different ways for different purposes. Types arise informally in any domain to categorize objects according to their usage and behaviour. The classification of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system. Types arise naturally, even starting from untyped universes.*

Value-Based Programming

- Values are referentially transparent objects
  - Typically not in a Class Hierarchy
  - Strongly informational but no separate identity
- In C++ values are associated with a set of capabilities and conventions
  - Uniform usage dictates how to pass values around, i.e. by copy or *const* reference, but not pointer

Values are objects for which identity is not significant, i.e. the focus is principally on their state and then their behaviour. It can be argued that for a value its state *is* its identity, as that is the only property of interest that distinguishes it from other objects. Other distinguishing features of values include their granularity and content: they are typically fine grained rather than coarse, and their behaviour is closely structured around their state.

An example of this is a string where the focus is on its content and its manipulation, but not on its address in memory: comparison of the content of two strings is of interest, but comparison of their identity is less useful. One value is substitutable for another with the same state. Thus in C++, value-based programming relies heavily on `const`, aliasing through references, the ability to copy by construction and assignment, and typically little or no involvement with Class Hierarchies, although they may be in Type Hierarchies or Type Heterarchies. However, simply because a value can be copied, it does not mean that it must be copied – this is part of the transparency. This is in contrast for objects whose identity is significant, where copying is often not meaningful – and is therefore disabled – and which often play a role in a Class Hierarchy.

A common question asked by many C++ developers is when to pass objects around by pointer and when to pass by reference. This is perhaps the wrong question, but it can be resolved by an appeal to common usage: values are typically stack objects or data members, and they will often support operator overloading. This suggests that they should always appear either as references or directly declared objects. Objects for which identity is significant should be passed around with their identity significant, i.e. by pointer.

Patterns

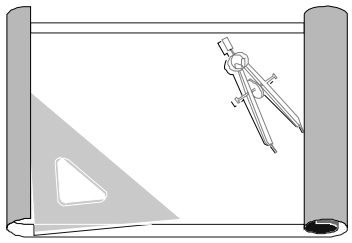
> ***The repetition of patterns is quite a different thing from the repetition of parts.***
>
> **Christopher Alexander [Alexander1979]**

Many problems have solutions that are structurally similar to each other, with common issues and rationale. This is something that more experienced developers learn to recognise and apply. A pattern captures and communicates this experience in a reusable form.

The focus of this section is on introducing and exploring patterns, with an eye to supporting substitutability and expressing concepts idiomatically in C++.

Patterns

- Intent
  - Understand essential pattern principles
- Content
  - Pattern anatomy
  - Proxy
  - Idioms
  - Smart Pointer
  - Pattern relationships



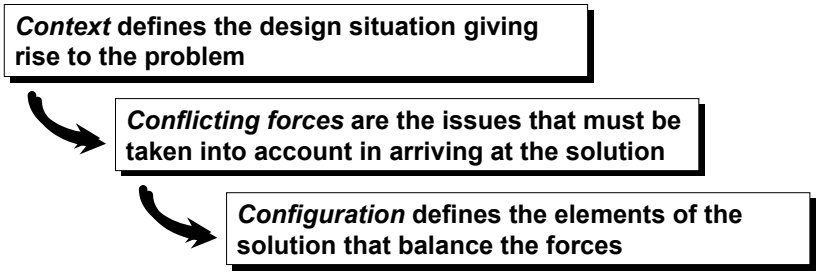
The seminal design patterns book [Gamma+1995] has perhaps done more than any other to popularise the use of patterns in OO design. The award winning book has raised the level of awareness of the importance of patterns, and the key role of design, to the point that its patterns now form the base vocabulary of many new development projects.

Patterns can be identified and applied in a broad range of domains, highlighted especially by their origins in the architecture of the built environment [Alexander+1977, Alexander1979] before being adopted in software architecture. The particular patterns documented by the "Gang of Four" are general purpose, common, and focused on OO design. In spite of this, there is a lingering perception by some developers that the patterns in their book are all that there are in this rich field, and that this level of design is their only context of relevance.

That this is not the case can be quickly dispelled by picking up any of the PLoPD books [PLoP1995, PLoP1996, PLoP1998], surfing [Hillside], or browsing any of a number of the object magazines and journals (*JOOP*, *C++ Report*, *Java Report*, etc).

Pattern Anatomy

- A pattern documents a reusable solution to a problem within a given context
  - Captures all the facets defining the design space



12

Good patterns are recurring solutions to similar problems that are known to work, i.e. they are empirically based and drawn from experience rather than invented for their own sake. That they are documented pieces of successful design experience and not single instances of design is a significant distinction from things that are simply pieces of "neat design" [Coplien1996]:

> *A pattern is a piece of literature that describes a design problem and a general solution for the problem in a particular context.*

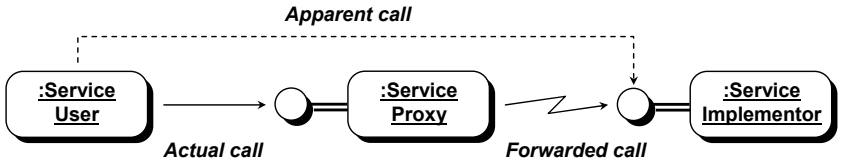
Patterns originated in architecture and the design of human centred environments, in the work of Christopher Alexander [Alexander+1977, Alexander1979], with an emphasis more on practice and prior art than invention. Alexander defines the essential content of any pattern:

> *We know that every pattern is an instruction of the general form:*
> *context ⇒ conflicting forces ⇒ configuration*

At heart, a pattern is a descriptive text that describes a problem within a context, the forces that must be balanced in evaluating the solution, and a configuration that defines a general solution. It names and documents the problem, the solution and the rationale. As such patterns are reusable concepts that can shape a design. In describing a pattern it is often the case that a motivating example will illustrate it better than a model that speaks only in the most abstract terms.

A Motivating Example

- Problem...
  - How to communicate transparently with an object in a different address space?
- Solution...
  - Provide a placeholder to forward the request



13

In message-based distributed and component-based systems address spaces are disjoint, presenting the programmer with a loss of transparency when communicating with objects in other processes: for local objects direct communication through references and pointers is possible; for remote objects this is not possible, and seems to encourage the use of more low level programming techniques, such as transport layer programming. An obvious disadvantage with this approach it creates an intrusive mismatch between local and remote object communication, as well as burdening the application programmer with excessive system programming details.

The essence of the solution is that an interface describing the desired behaviour of an object is implemented in two parts: one part is the concrete implementation; the other is the part responsible for any forwarding and translation. Each operation in the proxy forwards to the actual, associated target. With the exception of any system failure modes involved with connection, the use of the proxy as opposed to the actual target is transparent because they implement the same interface.

We see this design solution directly in, and as a core part of, many distributed system and component-based implementations, including CORBA and COM. The mechanism of delegation is used to forward and translate messages across address spaces and representations.

As we shall see, the salient features of this common problem and solution pair may be understood and generalised as the Proxy pattern [Gamma+1995].

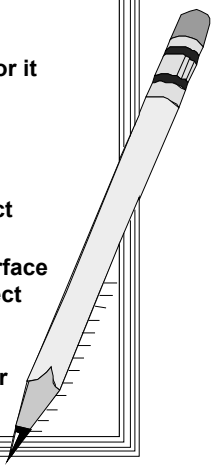
Proxy

Context
- **A client communicating with a target object**

Forces
- **Direct access to the target object is either not possible or it must be controlled and managed**
- **The control and management of access should affect neither the client nor the target**

Solution
- **Provide a proxy that stands in for the actual target object**

Consequences
- **The proxy and target objects implement a common interface**
- **The proxy holds a link to the target implementation object**
- **The proxy controls and manages access to the target implementation object, forwarding requests to it**
- **Adding a level of indirection provides an additional layer**
- **This extra layer is effectively transparent to the client**



14

The use of a proxy object resolved the problem of transparent distributed procedural communication in the motivating example. A similar solution can also be found in other object systems, suggesting that there is a pattern that can be generalised. The intent of the Proxy pattern [Gamma+1995] is given as

Provide a surrogate or placeholder for another object to control access to it.

The alternative names Surrogate [Taligent1994] and Ambassador [Coplien1992] are also suggestive of the basic concept behind this pattern. The form above illustrates the key features of this pattern for transparent forwarding.

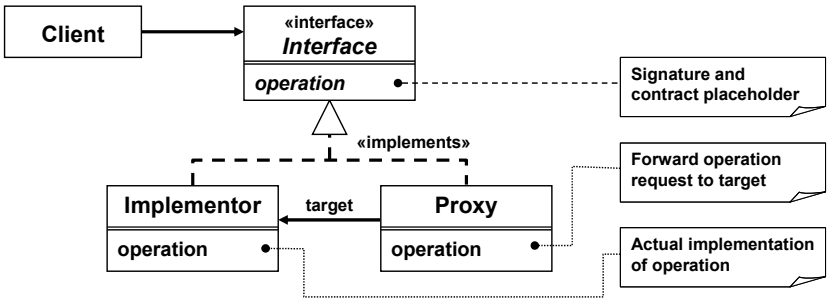
There are many variants of Proxy [Gamma+1995, Buschmann+1996] which can be considered specialisations of the basic pattern:

- Remote Proxy corresponds to the case in the motivating example.
- Virtual Proxy virtualises the presence of an object, for instance by applying the Lazy Evaluation to support load or creation on demand.
- Protection Proxy controls access to an object.
- Synchronization Proxy synchronises multiple access of the target.
- Counting Proxy keeps a usage count of users of the target.
- Smart Reference and Smart Pointer provide language specific proxy solutions where it is the mechanism to access the object that is uniformly overloaded for all invocations.

[Taligent1994] provides a more detailed taxonomy of surrogates based on their structure and intent.

Proxy Configuration

- Common configurations for the pattern solution may be illustrated with diagrams
  - A generalised model of the solution



15

Pattern text is often supported by diagrams in a semi-formal notation, such as UML (Unified Modeling Language) [Rumbaugh+1999]. The most common diagrams tend to be *class diagrams*, showing the static and 'spatial' relationships between classes and their instances, or *object diagrams*, showing a snapshot of an instance configuration.

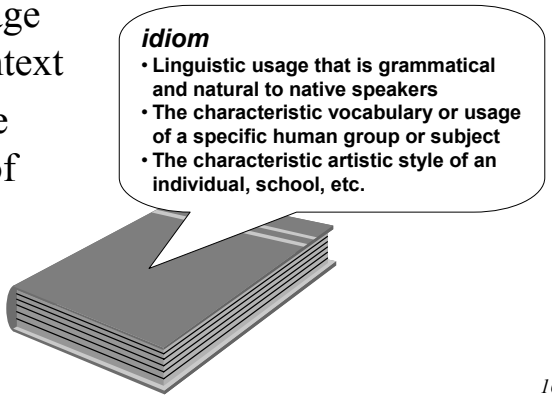
Although such static configuration diagrams constitute the majority, there are many patterns whose structure is temporal or control oriented, and hence additional or alternative diagrams (such as statechart or interaction diagrams) or notation are more appropriate. Such notation includes code, pseudocode or more rigorously in a formal notation, such as UML's Object Constraint Language (OCL).

CRC cards can also be used to document part of the configuration, providing a convenient format for outlining the responsibilities and collaborations of the participants.

It is important to note that while diagram and supporting code may be used to illustrate or support a pattern, they do not themselves constitute the whole pattern.

Idioms

- An idiom is a pattern specific to a language, language model or technology
  - It has the language as part of its context
  - Some idioms are specialisations of general design patterns



16

A number of idioms have been catalogued for C++ [Coplien1992]and Smalltalk [Beck1997]. Some of these patterns include those more generally concerned with design issues, whereas others are at the opposite end of the scale, being concerned instead with naming of variables and methods, such as Beck's Intention Revealing Message, and layout, such as Richard Gabriel's Simply Understood Code [Coplien1996].

Some idioms, such as those for naming, can be transferred easily across languages. Others depend on features of a language model and are simply inapplicable when translated, such as a strong and statically checked type system (e.g. C++ and Java) versus a looser, dynamically checked one (e.g. Smalltalk and Lisp). For instance, the Detached Counted Handle/Body idiom [Coplien1996] describes a reference counting mechanism for C++, the need for which is obviated in Smalltalk and Java by the presence of automatic garbage collection.

Sometimes idioms need to be imported from one language to another, breaking a language culture out of a local minima. Idiom imports can offer greater expressive power by offering solutions in one language that exploit similar features as in the language of origin, but which have not otherwise been considered part of the received style of the target language.

However, it is important to understand that this is anything but a generalisation and the forces must be considered carefully. For example, a great many C++ libraries have suffered from inappropriate application of Smalltalk idioms, and the same can also be said of many Java systems with respect to C++.

Smart Pointer

- A C++ specific adaptation of Proxy
  - Classes that support pointer-like protocol

Detached Counted Handle/Body:

- *Reference counting automates object lifetime management*
- *Keeping the usage count separate from the body means that the type of the counted object is unaffected*
- *Smart Pointer idiom makes use of this idiom familiar*

```
template<typename type>
class counted_ptr
{
public:
    explicit counted_ptr(type *initial_ptr = 0);
    counted_ptr(const counted_ptr &);
    ~counted_ptr();
    counted_ptr &operator=(const counted_ptr &);
    type &operator*() const;
    type *operator->() const;
    ...
private:
    type   *target;
    size_t *count;
};
```

17

Pointers conventionally represent a level of indirection to a target; structurally this is a feature shared with Proxy classes. In C++ the Smart Pointer idiom builds on the understood protocol for pointers, specialising the Proxy pattern for certain examples where access control takes the same form for all operations.

The essential features of a Smart Pointer are its dereferencing operations: through overloading, `operator*` and `operator->` are supported. Both of these operators provide access to the indirectly referenced object, but do not themselves modify the proxy hence they are `const`. The `operator*` typically returns a reference rather than a value, and the `operator->` must return something that in turn supports an `operator->`, such as a raw pointer.

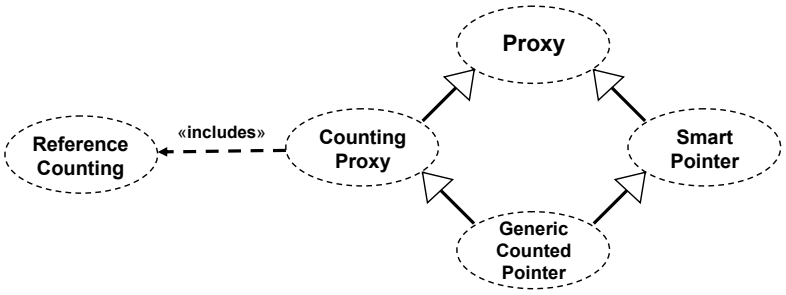
It is normally the responsibility of an object user to deal explicitly with the ownership and lifetime of objects. Reference counting puts the responsibility for management of object ownership into the association. The principle behind reference counting is to arrange to keep a shared usage count of the object pointed to, updating its value when they are shared or forgotten. On falling to zero the target object is effectively unused, and is deleted.

Reference counting can be used as a simplified form of garbage collection, as well as the basis for optimisation through representation sharing. For a language such as Java or Smalltalk that already supports garbage collection in the runtime environment, the Smart Pointer idiom is superfluous.

There are many different reference counting idioms that may be used in C++ (see [Boost] for some discussion and implementations of these). Detached Counted Handle/Body has the benefit that it can be made generic without intrusion on the type of the object being pointed to.

Pattern Relationships

- Proxy defines a family of patterns
  - Specialised with respect to context or purpose
  - Inclusion of other patterns for completion



18

One pattern may be considered a specialisation of another pattern, i.e. more specific in some way. The specialisation occurs in the lead up to the solution, i.e. the context is more specific or the intent and purpose of a pattern is made more specific. Sometimes such specialised patterns are termed *pattern variations*. Some idioms are examples of design patterns specialised within a context: they occur and are adapted within the context of a specific language, technology or architecture. For instance, the expression of many behavioural patterns may be affected by the presence of distribution. In other patterns, they are specialised by intent, i.e. a narrowing of the problem scope.

Although not formally part of UML, it is possible to illustrate pattern relationships by generalising the notation slightly. Pattern specialisation can be considered a refinement of detail from one pattern to another. Refinement may be either by elaborating the context or narrowing the problem scope.

Some patterns appear, on closer inspection, to be composed of other patterns in the way that their problem and solution statements are constructed. It is often the case that more than one pattern is used to solve a particular recurring problem, suggesting a larger chunk of recognisable and nameable design. These have been termed *compound patterns* [Vlissides1998a, Vlissides1998b].

Compound patterns express patterns that are themselves particular communities of patterns. They can, in many ways, be viewed as small pattern languages. In some cases a base pattern is specialised by the use of another pattern, either for purpose or for context. The relationship of a design pattern affected consistently by a practice, such as the application of concurrent programming idioms, to that practice can be considered one of inclusion.

Particulars

> ***In this connection it might be worthwhile to point out that the purpose of abstraction is* not *to be vague, but to create a new semantic level in which one can be absolutely precise.***
>
> **Edsger Dijkstra [Dijkstra1972]**

This section is broadly divided along the lines of the different substitutability mechanisms available in C++, as perceived by the author. Because there is a certain overlap between the categories – they are not intended to be mutually exclusive – they are ordered such that the more primitive substitutability types may be used as building blocks for later ones:

- *Conversions* are enabled by specific forms of overloading, namely constructors and user defined conversion operators.
- *Overloading* defines a substitutability by visual appearance, and the expectation that follows from that. It can be said to be at the heart of conversions and genericity, but in a practical sense it interacts (and interferes) most with conversions.
- *Derivation* substitutability is the classic substitutability based on inheritance, overriding, and conversions between derived and base class pointers and references.
- *Mutability* defines a form of substitutability based on presence or absence of state change, and is founded on conversions (trivial non-`const` to `const`), overloading and derivation.
- *Genericity* builds mainly on support for conversions and overloading, but also to some extent on derivation and mutability.

Consideration of a category includes consideration of its opposite, i.e. non-substitutability – negative variability [Coplien1999]. Monomorphism, for example, is where a type does not co-operate with other types and functions operate only on that type, and is a degenerate case of polymorphism.

Conversions

- Intent
  - Understand conversion-based substitutability
- Patterns
  - Inward Conversion
  - Outward Conversion
  - Explicit Inward Conversion
  - Custom Keyword Cast

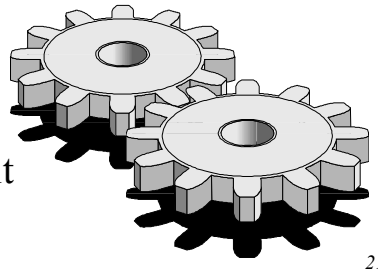
C++ supports implicit and explicit conversions between values of built-in types and user defined types. Here the focus is on the available conversion mechanisms with respect to the practices appropriate for their use.

Inward Conversion describes when and how to provide a converting constructor for a class. Outward Conversion describes when and how to provide a user defined conversion operator for a class. Note that too many conversions can lead to ambiguity problems, which is particularly the case with Outward Conversion. Explicit Inward Conversion describes appropriate motivation and use of explicit constructors. Because of limitations in the C++ language, there is no natural opposite to Explicit Inward Conversion. However, the use of explicit template function qualification supports the creation and use of Custom Keyword Cast functions that emulate the appearance of the built-in keyword casts.

In action these patterns can be seen to emulate (or fake, depending on your sensibilities) the substitutability offered by inheritance. For instance, to create a family of interoperable numeric types in C++, use of inheritance presents a number of challenges and often proves to be overkill. Inheritance also does not follow the idiom of C++'s own built-in numeric types, which form a Type Heterarchy. The most appropriate solution is to define simple abstracted value types and provide conversions between them.

Principles and Mechanisms

- A conversion may be implicit or explicit
  - *Implicit* places control with the compiler
  - *Explicit* places control with the developer
- Conversions may be widening or narrowing
  - A widening conversion is always safe
  - A narrowing conversion is not and should be explicit



Conversions may be implicit or explicit, which places them under the control of the compiler and the developer, respectively. Whether a conversion should be implicit or explicit is a mixed matter of taste, safety and requirement. Widening conversions, where a conversion is from a specific to a general type, are always safe and can be implicit without offending sensibilities or the compiler, e.g. `int` to `double` or derived to base. Narrowing conversions, where a conversion is from a general to a specific type, are not guaranteed to be safe and should be explicit. One would hope that narrowing conversions would be required to be explicit, but this is not always the case, e.g. `double` to `int`. Even though the compiler does not require it, one might argue that taste does. Where possible, narrowing conversions should be checked, e.g. the use of `dynamic_cast` to go from base to derived.

User defined conversions, through converting constructor and user defined conversion operators, can allow a developer control over class miscibility. This is useful for value-based classes and where Type Heterarchies need defining.

Copying is a degenerate form of conversion, and therefore includes copy constructors and assignment operators. Other assignment operators are also considered to express a form of conversion, i.e. the ability to use an object on the right hand side that is of a different type to the left hand side. Conceptually the conversion is expressed in the implementation of the assignment operator.

Conversions can also be constrained through Preventative Overloading, where member functions are defined as private to intercept unwanted implicit conversions. This is common in preventing copying for non-value-based classes, but can also be used to good effect in other classes for other conversions.

Inward Conversion

- Meaningful implicit conversion between types by converting constructor
  - Can made be explicit by use of traditional cast form, constructor-like form, and *static_cast*

A* string *class and* char * *are different realisations of the same basic concept, i.e. character string. Implicit conversion from* char * *to* string *is meaningful and desirable.

```
class string
{
public:
    string(const char *);
    ...
};
```

22

An Inward Conversion can be supported by the introduction of one or more converting constructors on a class. These allow conversions into a type from other types. This should be used in support of conceptually similar types, e.g. `string` and `const char *` are both representations of strings. The more normal case is that two types are not so related, e.g. a file and its name, and an Explicit Inward Conversion should be used instead.

An Inward Conversion supports an implied equivalence meaning that should be respected by the developer. Both this principle, and pragmatics related to overloading and ambiguity, mean that multiple Inward Conversions – copying excluded – should be viewed with some suspicion, and need strong justification.

Providing an Inward Conversion also provides the developer with a cast form for a type. It is not possible to define a literal form for a new type, but the constructor expression syntax comes close, e.g. `string("theory")`. This is stylistically preferable to using `static_cast` in this context as it is a well defined conversion (as opposed to a potentially dangerous conversion that must be highlighted in the source code) and corresponds well to the idea of constructing a new value. Thus programming guidelines that recommend `static_cast` instead of function (or traditional) cast notation for all such conversions are misguided, and give the code the wrong meaning, i.e. "look here, there's a suspicious narrowing conversion going on". The preferred Constructor Literal style also means that code appears consistent when used with other multiple argument constructed forms, e.g. `string(5, '*')`.

Outward Conversion

```
class file
{
public:
    char read(size_t index) const;
    void write(size_t index, char value);
    reference operator[](size_t);
    class reference
    {
    public:
        operator char() const;
        reference &operator=(char);
        ...
    };
    ...
};
```

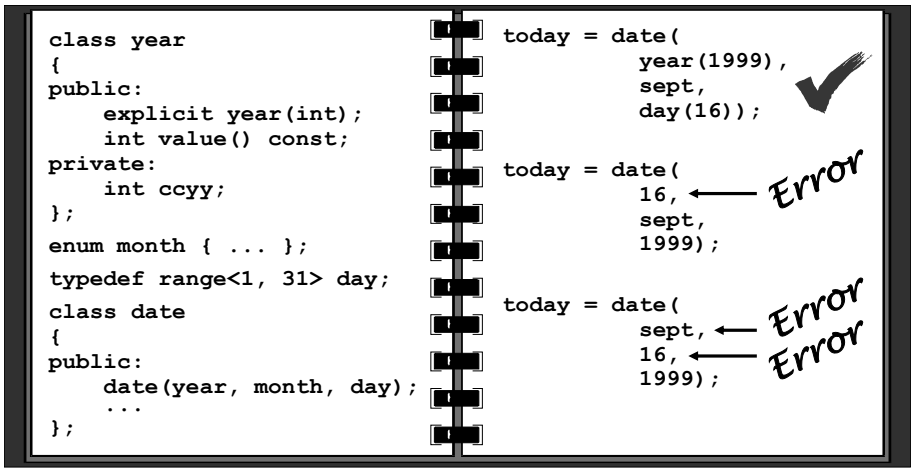
- *For files modelled as indexable containers, a Smart Reference (Proxy) is returned in place of a reference for indexing to allow writing back.*

In the example above a file is modelled as an indexable container, supporting `operator[]` for subscripting. It must return something that may be the target of assignment. A conventional *lvalue* will not do, and so a Smart Reference is returned to act on the operator's behalf:

```
file::reference file::operator[](size_t offset)
{
    return reference(*this, offset);
}
file::reference::operator char() const
{
    return target.read(where);
}
file::reference &file::reference::operator=(char rhs)
{
    target.write(where, rhs);
    return *this;
}
```

An Outward Conversion ensures that this behaves as – and yields – a value in the required context. There are some limitations to Smart References, and they cannot be used as generally and freely as smart pointers. For instance, they must be type specific and support all the relevant operations on the target directly; there is no way to overload `operator.` and so a smart reference cannot be made completely generic, as is the case for smart pointers. [Meyers1996] documents a number of other issues.

Explicit Inward Conversion



It is tempting, and indeed common, to represent value quantities in a program in the most fundamental units possible, e.g. durations as integers. However, this fails to communicate the understanding of the problem and its quantities into the solution and shows a poor use of the type system.

The loss of meaning and checking can be recovered by applying the Whole Value pattern [Cunningham1995] (also known as Quantity [Fowler1997]). In this, distinct types are used to correspond to domain value types. This affords greater annotation in the code and improved checking by the compiler, as illustrated in the example above. In C++ this distinction is supported – and indeed, the pattern underpinned – by Explicit Inward Conversion. For many whole value types it is intended that they should be distinct from their fundamental unit type and should not cause any ambiguous conversions. For this, use `explicit` to inhibit converting constructors.

A raw Whole Value can be expressed as a Constructor Literal, which reduces the number of named temporaries cluttering code and increases the code's expressiveness and richness of meaning.

Whole Value is similar to dimensional analysis in the physical sciences, and a variation of Whole Value can be generalised as a Generic Value Type for this purpose using templates [Barton+1994].

Custom Keyword Cast

- How can explicit conversions be added?
  - Emulate the standard keyword casts

```
template<typename result_type, typename arg_type>
result_type interpret_cast(const arg_type &arg)
{
    std::stringstream interpreter;
    interpreter << arg;
    result_type result = result_type();
    interpreter >> result;
    return result;
}
```

General conversions between types based on stringified forms can be supported conveniently using the syntax **interpret_cast<target_type>(source)**

How can an explicit outward conversion be provided for a type, or for a conversion between two existing types using a particular conversion method not already implemented by either type?

C++ does not unfortunately support `explicit` on user defined conversion operators. Thus where outward conversions are possible and meaningful, but need to be controlled, the developer cannot use the same conventions as for other conversions. Named conversion functions are functionally equivalent, but are certainly not consistent with expectation. Another case is that where conversion is required between two existing types or type families already exist, but neither can be extended conveniently to accommodate the new conversion.

The issue can be resolved with a Custom Keyword Cast, where the keyword casts (e.g. `static_cast`) are emulated in appearance using explicit template function qualification. This approach is used in [Cantrip, Stroustrup1997, Henney1998b, Boost]. For instance, a range checked conversion between numeric types can be implemented (in simplified form) as follows:

```
template<typename result_type, typename arg_type>
result_type numeric_cast(arg_type arg)
{
    typedef std::numeric_limits<arg_type> arg_traits;
    typedef std::numeric_limits<result_type> result_traits;
    if((arg < 0 && !result_traits::is_signed) ||
       (arg_traits::is_signed && arg < result_traits::min()) ||
       (arg > result_traits::max()))
        throw std::bad_cast();
    return static_cast<result_type>(arg);
}
```

Conversions for Smart Pointers

An implicit conversion from a raw pointer to a counted_ptr *is undesirable, but as an explicit conversion this is OK. Implicit conversions from related pointers, e.g. to types in a Class Hierarchy, are OK. Implicit conversions from a* counted_ptr *to a raw pointer are undesirable, so named queries and keyword cast functions are appropriate.*

```
template<typename type>
class counted_ptr
{
public:
    explicit counted_ptr(type *initial_ptr = 0);
    counted_ptr(const counted_ptr &);
    template<typename related_type>
      counted_ptr(
        const counted_ptr<related_type> &);
    type *get() const;
    ...
};
```

```
template<typename related_type, typename type>
  counted_ptr<related_type>
    dynamic_counted_cast(
      const counted_ptr<type> &);
```

26

Different conversions make sense for different Smart Pointers. For any Smart Pointer that takes ownership of a pointer, such as `counted_ptr`, an Inward Conversion from a raw pointer is undesirable. Such implicit behaviour can lead to surprising silent conversions. Instead a controlled Explicit Inward Conversion ensures that developers get what they ask for and ask for what they get.

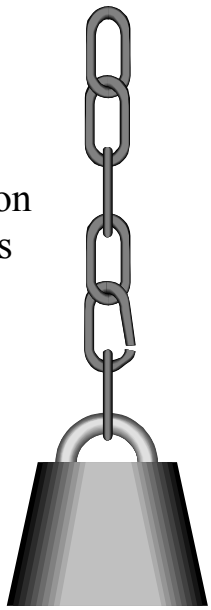
However, for mixing Smart Pointer objects to classes in a Class Hierarchy, it makes sense to support implicit conversions, e.g. from `counted_ptr<`*`derived`*`>` to `counted_ptr<`*`base`*`>` but not vice-versa. This can be generalised generically with the use of member templates. Friendship may need to be granted so that all instantiations of a templated Smart Pointer class can see inside all others, otherwise mechanism cannot be accessed and shared for copy construction and assignment.

For similar reasons that Inward Conversion is inappropriate, Outward Conversion is even more undesirable. So how is a user to access the raw pointer content? Often they will not need to, using `operator*` and `operator->` instead, but in those cases that they do it is best to make it explicit with a named query function, e.g. `get`.

Use of objects within a Class Hierarchy implies that some form of checked downward conversion may be required, to balance the implicit upward conversion. Performing a `dynamic_cast` on the result of get does not have the desired consequence: the result is a raw pointer and not a `counted_ptr`. This can be resolved by providing a Custom Keyword Cast, `dynamic_counted_cast`.

Overloading

- Intent
  - Understand substitutability based on overloading, in particular operators
- Patterns
  - Operator Follows Built-ins
  - Balance Overloaded Operators



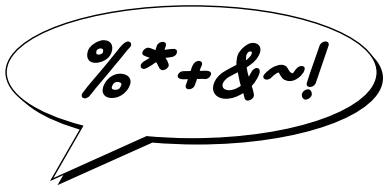
Overloading is a feature common to many modern languages. The principle guiding its best use is that common name implies common purpose, and hence it can be used to achieve a form of substitutability. Default arguments can be considered to be a degenerate form of overloading.

Because of its power – and potential to complicate as well to simplify – operator overloading is of particular interest where guidelines are concerned. At the finer levels of design, operator overloading often builds on, or establishes, a conceptual framework, e.g. Smart Pointer. Such an expression of interface and commonality is important for many value-based classes and, when taken together with the conversion-based substitutability, provides a cornerstone of generic programming.

In many respects operator overloading is akin to small language design. Thus, operator overloading is at least as much about taste as it is about mechanism and rote practice. However, while all good practices are elegant, there is no single pattern that adequately and generally captures taste in design – this is definitely a quality of the individual!

Principles and Mechanisms

- Common name implies common purpose
  - Operators provide a predefined set of names and definitions, and therefore expectations
- Operators should be considered part of the class interface
  - Regardless of whether or not they are actually class members



Overloading is based on the idea that common name implies common purpose, and this frees programmers from indulging in their own name mangling to differentiate similar functions (this is the job of the compiler). Overloading works closely with – and sometimes against – conversions. Developers are cautioned to keep any eye open for any such interference.

Extension through overloading forms an important part of operator overloading. For instance, a class that is closed to change can apparently be extended to work within a framework, e.g. 'extending' I/O streams to understand new types. Some extension is less transparent, but importantly follows as much of the base form as possible. An obvious example is the use of placement new operators which, in spite of taking additional arguments, have the same purpose and much of the same form as the vanilla `new`. Tagged overloads, such as `new(std::nothrow)`, provide a means of compile time selection that is a compromise between operators and named functions.

A combination of language constraints and common practice idioms suggest which operators should be members and which should be global. Regardless of this, operators should be considered a part of the class interface. This used to be simply a matter convention, but since the introduction of Koenig Lookup for namespaces it has become a compiler supported idiom. [Sutter2000] refers to the resulting guideline as the Interface Principle:

> *For a class X, all functions, including free functions, that both (a) "mention" X, and (b) are "supplied with" X are logically part of X, because they form part of the interface of X.*

Operator Follows Built-ins

- What guidelines should the behaviour of overloaded operators follow?
  - Compilers neither care nor check
- Built-in operators set expectations and offer a familiar set of behaviours
  - Follow their lead where possible

When in doubt, do as the `ints` do.

Scott Meyers [Meyers1996]

The founding good practice for operator overloading can be considered Operator Follows Built-ins. Such set of recommendations is common and can be found in many places, including [Meyers1996] and [Meyers1998] and is further formalised in generic programming requirements [Stepanov+1995, Austern1999, ISO1998]. For any user of the code this idiom supports the principle of least astonishment.

Where the implied semantics of built-ins cannot be met, other accepted idioms, such as streaming, provide a second port of call. Thus we can temper a hard line view of operator overloading with a little – but not too much – pragmatism:

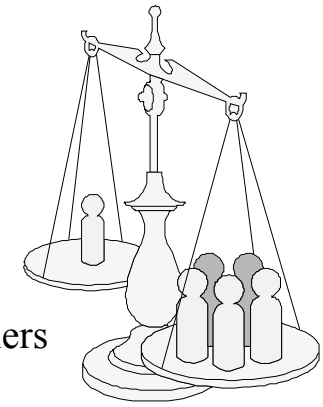
> *"In view of this philosophy of only overloading operators intuitively, how do you explain the first example in your book, which shows an overloaded Left Shift << operator acting as a stream operator?"*
> "Good question. We have a saying in Denmark: 'Don't do as the priest does, do what the priest says.'"
>
> [EXE1990]

For some cases, overloading can be considered unreasonable: because the short circuiting evaluation cannot be emulated for `&&`, `||` and `,`, they should not be overloaded. For others, a little laxness in interpretation is often considered reasonable, e.g. the use of `operator+` for string concatenation, which is clearly not commutative.

Where operators redefine something fundamental, such as memory management, writing to common form becomes more than just a courtesy: failure to follow form can corrupt a system.

Balance Overloaded Operators

- What does *completeness* mean for operators provided for a type?
  - Completeness is related to expectation and ease of use
- Relationships exist between built-in operators
  - Overloading one operator often leads to overloading others



Interfaces should be as complete as is meaningful. But what does this mean for overloaded operators? One can extend the basic advice of Operator Follows Built-ins to cover relationships between operators. This results in a need to Balance Overloaded Operators [Taligent1994]. There is a set of expectations that should be met by the class developer: operator overloading comes with a greater set of obligations than the compiler will check.

Class users have the right to full functionality. For instance:

- Equality as `operator==` implies `operator!=`.
- Relational comparison in the form of `operator>` implies, in addition, `operator>=`, `operator<` and `operator<=`.
- Prefix `operator++` implies postfix `operator++`.
- Binary `operator+` implies `operator+=`.
- Dereference `operator*` implies `operator->`.

The can also reasonably expect such behaviour to be consistent, in following Operator Follows Built-ins. For instance:

- Equality and inequality are clearly related, such that `a != b` should be equivalent to `!(a == b)`.
- Symmetric operators overloaded to ensure symmetry.

Note that the use and style of the templated operators in `std::rel_ops` is a questionable shortcut in gaining such balance.

Overloading for Smart Pointers

operator* *and* operator-> *are a balanced pair of operators.* operator! *can be balanced by* operator const void *, but its consequences must be held in check by further overloading* operator== *and* operator!=*, which are themselves balanced.

```
template<typename type>
class counted_ptr
{
public:
    type &operator*() const;
    type *operator->() const;
    bool operator!() const;
    operator const void *() const;
    ...
};
```

```
template<typename lhs_type, typename rhs_type>
  bool operator==(
    const counted_ptr<lhs_type> &,
    const counted_ptr<rhs_type> &);
template<typename lhs_type, typename rhs_type>
  bool operator==(
    const counted_ptr<lhs_type> &,
    const rhs_type *);
```

31

A Smart Pointer is already an example of Operator Follows Built-ins. The most obvious aspect of this is to ensure that operators dealing with indirection are included. Principally this means both `operator->` and `operator*`, which form a pair that Balance Overloaded Operators. For a Smart Pointer to a function or a Function Object [Stepanov+1995, Austern1999], Operator Follows Built-ins suggests that `operator()` can be provided as an additional operator.

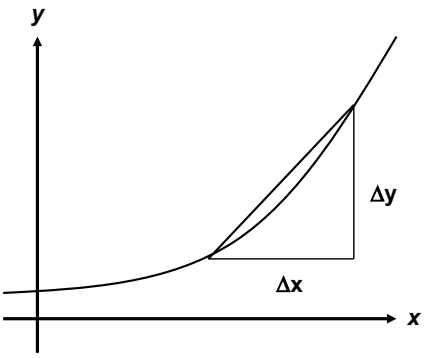
Equality comparison is a common Smart Pointer requirement, and both `operator==` and `operator!=` can be provided to Balance Overloaded Operators. There is no need to fully parrot built-ins: some normal pointer operations, such as pointer arithmetic and therefore full relational comparison, introduce liabilities rather than something useful. A sliding scale of feature inclusion can create an expressive Type Hierarchy, e.g. iterators [ISO1998].

It is a standard idiom to see pointer values used on their own as an existence check, i.e. `ptr` or `!ptr` representing a Boolean value. The provision of `operator!` is trivial, but to Balance Overloaded Operators is a little trickier: what is the natural complement of `operator!`? For many classes `operator bool` makes the most sense, but it introduces a number of undesirable conversion problems for a pointer-like class [Meyer1996]. Outward Conversion is not appropriate for all Smart Pointers. `operator const void *` is a compromise as no user actually gains useful access to a type object through it. To prevent permissive conversions between pointer types, `operator==` and `operator!=` need to be further overloaded for raw pointer types.

Some smart pointers break such recommendations (e.g. the C++ mapping in [OMG]) and this reduces their usability or makes them more of a liability.

Derivation

- Intent
  - Understand substitutability based on inheritance
- Patterns
  - Mix-in
  - Interface Class
  - Implementation-only Class
  - Cloning



32

Substitutability with respect to derivation is perhaps the most familiar category for programmers coming from a straight OO background. This is most effectively expressed by the Liskov Substitution Principle (LSP).

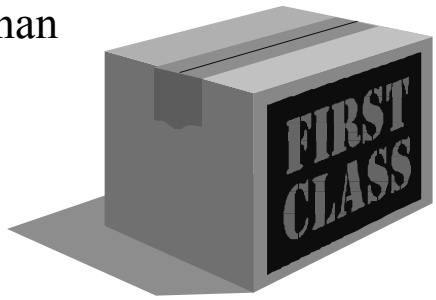
Mix-ins express a common and useful approach to the expression of optional capabilities and the use of multiple inheritance, which is typically contentious when used outside a framework of good practice.

Interface Class describes how an abstract class can be used to represent only an object's interface, and the benefits of doing so. The alternative is often to succumb to the temptation of mixing a little default or convenience implementation in, which is a more classic use of abstract classes. An Implementation-only Class describes how to provide a class that implements Interface Classes but, apart from object creation, cannot be used to manipulate an object. This basic separation is sometimes seen in terms of *usage type* and *creation type* [Barton+1994].

Although the common case is not to copy objects in a Class Hierarchy, there are cases when it is needed. Cloning overcomes the consequences of separation of interface from implementation.

Principles and Mechanisms

- Public inheritance should conform to the Liskov Substitution Principle
  - A Class Hierarchy should also be a Type Hierarchy, i.e. subclass should also be subtype
- LSP is more precise than simply *"is a kind of"*
  - Bounds behaviour of overridden functions



33

Inheritance is a class relationship, and is one of the strongest relationships one can have in an object system. As such inheritance introduces a strong coupling. However, if used with care, inheritance can have the opposite effect, i.e. that of reducing dependencies in a system. This relies on grouping stable classes together, and separate from more volatile classes. This distinction basically boils down to abstractness [Martin1995], and the acknowledgement that in a system interfaces leave a more enduring mark than implementations.

The Liskov Substitution Principle [Liskov1987, Coplien1992] focuses on the role of class as behavioural classifier, i.e. Class Hierarchy should follow classification. This implies that public inheritance should be used to model, first and foremost, a subtyping relationship rather than a subclassing relationship, which can be considered as secondary. In this context we consider a type to be a description of interface and behaviour, whereas a class is a realisation of this.

LSP defines a higher degree of conformance between derived and base classes than the oft quoted piece of advice – that public inheritance should only be used to model an *is a kind of* relationship – perhaps suggests. The substitutability of a derived class is defined at the level of its behaviour, which means that overridden functions must respect the expectations implied (or explicitly stated) for their predecessors in the base class.

Following LSP should not simply be considered a nicety, but a requirement where possible. The compiler implies that LSP is the way to go because of the way it implicitly deals with pointer conversions from derived to base. However, as with all other forms of substitutability, the compiler is powerless to pass judgement on the merits or otherwise of a particular class relationship.

Mix-in

- Optional functionality for derived classes
  - Provides interface and/or implementation

```
class lockable
{
public:
    void lock() const;
    bool try_lock() const;
    void unlock() const;
    ...
};
```

```
class example : public lockable
{
    ...
};
```

34

A Mix-in offers a way of adding properties to objects at compile time through their class. Thus a Mix-in class often provides partial implementation. Where interface is not as significant, we might differentiate this role by referring to it as a *mix-imp*.

Examples of Mix-ins that a derived class might acquire include reference counting mechanisms, thread lockability, persistence, etc. In the simplest cases these concepts can be represented as regular classes, but it is possible to express with a greater degree of control through the use of templates, e.g. one way `lockable` can be refined is as follows:

```
template<typename lock_type>
class lockable
{
    ...
    mutable lock_type lock;
};
```

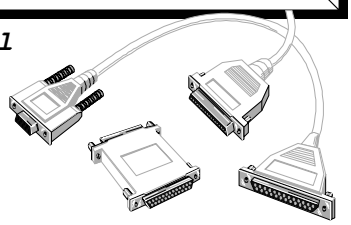
[Taligent1994] offers a number of guidelines that may be associated with derivation, including the use of Mix-ins:

> *Classes [may be] artificially partitioned into two categories:* base classes *that represent fundamental functional objects (like a car), and* mixin classes *that represent optional functionality (like power steering).... A class may inherit from* zero or one *base classes, plus* zero or more *mixin classes.... A class that inherits from a base class is itself a base class; it can't be a mixin class. Mixin classes can only inherit from other mixin classes.*

Interface Class

```
class file
{
public:
    virtual ~file();
    virtual bool read(string &to, size_t max) = 0;
    virtual bool write(const string &from) = 0;
    virtual bool flush() = 0;
    virtual bool eof() const = 0;
    ...
};
```

All ordinary functions are pure `virtual`



35

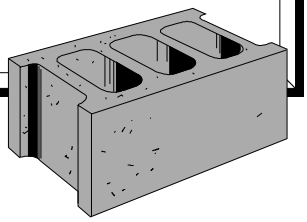
How can we represent the protocol for class usage without also expressing any implementation? The client wishes to depend on interface rather than implementation detail, therefore define a separate Interface Class [Carroll+1995] to express the common capability of derived classes, i.e. a class to represent the contract and many derived classes to fulfil it and express the implementation detail. This class has no data and the only ordinary member functions are declared `public` and pure `virtual`. If it is used as a Mix-in class – albeit a Mix-in that provides protocol only – `virtual` inheritance should be considered to avoid repeated inheritance issues.

If it is intended that objects can be deleted via a pointer to the Interface Class the destructor should be declared `public` and `virtual`. It is a matter of personal taste as to whether or not the destructor is also declared pure `virtual` as it must be defined either way. Personal preference is not to declare it as pure `virtual` as the occurrence of this usage – i.e. pure `virtual` declaration but with definition – is sufficiently rare that the majority of C++ developers are not familiar with it, and automatically take pure `virtual` to be equivalent to "has no definition". For `virtual` destructors such implied meaning clearly contradicts the reality.

On the other hand, if objects should not be deleted via the Interface Class it should not offer this ability, i.e. the destructor should be made `protected` and non-`virtual` thus deferring full responsibility for destruction policy to derived classes. In either case, the body of the destructor should be empty.

Implementation-only Class

```
class file_as_stream : public file
{
public:
    file_as_stream();
    ...
private:
    virtual ~file_as_stream();
    virtual bool read(string &to, size_t max);
    virtual bool write(const string &from);
    virtual bool flush();
    virtual bool eof() const;
    ...
    fstream stream;
};
```



Only constructors are `public` – all overridden functions are `private`

36

Where an Interface Class represents the *usage type*, or some aspect of the usage, of an object, the concrete class instantiated for the interface user represents a *creation type*. This distinction can be made clearer, and the dependencies reduced in a system, by enforcing such a model of use: the Interface Class is used only for manipulation and the only time the name of the concrete class appears is at the point of creation, i.e. in a new expression, which may itself be encapsulated within a factory object.

However, programmer convention alone does not explicitly guarantee such a constraint is observed; the emphasis is made stronger by recruiting the compiler. An Implementation-only Class is a concrete class that implements one or more Interface Classes. With the exception of its constructors all of its member functions, including its destructor, are private [Barton+1994]. This form of contravariance (interface variation against the direction of inheritance) is a feature of the orthogonality of access and polymorphism in C++. Thus creation type may only be used for creation, and the user must use the usage type to manipulate created objects. This reduces the dependency a system can – and therefore will – have on non-interface types. Such a decoupling allows us to view and discuss interfaces as the natural boundaries in our system.

The combination of these two idioms – Interface Class and Implementation-only Class – is only appropriate for objects managed using `new` and `delete`. The Interface Class should also sport a `public virtual` destructor, or the object should be used in conjunction with another memory management scheme. Where multiple Interface Classes are used for Role Decoupling [D'Souza+1999], `dynamic_cast` can be used to query supported interfaces.

Cloning

```
class graphic
{
public:
    virtual graphic *clone() const = 0;
    ...
};
```

Polymorphic copying is introduced at the base class level...

```
class ellipse : public graphic
{
public:
    virtual ellipse *clone() const
    {
        return new ellipse(*this);
    }
    ...
};
```

And realised in terms of conventional copying in the derived class.

Separation through Class Hierarchy means copying cannot be value based. Copying in a Class Hierarchy is not that common, but there are often cases where it is desirable. The nature of C++ means that the following is either not possible (because of abstract classes) or not what is meant:

```
class graphic {...};                        // abstract class
class ellipse : public graphic {...};       // concrete class
...
void copy(const graphic *original)
{
    graphic *copy = new graphic(*original); // won't compile
    ...
}
```

How can objects in a class hierarchy be copied if their runtime class is not known? The solution is not to resort to brittle, explicit type selection in the form of `dynamic_cast` and RTTI. The location of the knowledge of the exact type of the source object for copying is within that object. Therefore we should turn the problem around and ask it for a copy of itself. The Cloning solution provides the entry point for this capability in the base class and realises its implementation in the concrete derived classes, each of which knows how to make a copy of itself.

Another name for this technique is Virtual Copy Constructor, as it is a special case of Virtual Constructor [Coplien1992, Gamma+1995]. The specialisation is that it is a Factory Method [Gamma+1995] where the product and producer are instances of the same class.

Mutability

- Intent
  - Understand substitutability with respect to mutability of an object
- Patterns
  - Result Follows Qualification
  - Qualified Interface
  - Qualifying Proxy

> ***When it is not necessary to change, it is necessary not to change.***
>
> **Lucius Cary**

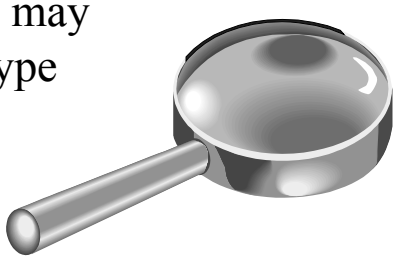
Just as it is difficult to consider silence until one ceases making noise, an oft neglected substitutability category is with respect to mutability; system state, and changes to it, are the very essence of modern systems, so that non-state changing aspects are often overlooked in design.

Procedural programming is focused on changes to system state; modular programming and object-oriented programming after it offer more disciplined models of state. However, ill considered changes to state (or uncertainty about changes to state) can lead to design problems (and, by implication, bugs); nowhere is this more apparent than in concurrent programming, which is becoming an increasingly common experience for developers in today's multithreaded environments. By contrast, functional and logic programming are based on declarative models of computation based solely on side effect free queries.

Result Follows Qualification ensures that access to a part of an object, via a query function, is the same as that for the whole object, i.e. a `const` query does not grant non-`const` access. The use of a Qualified Interface explicitly separates query and modifier functions into separate classes, normally related by inheritance. A Qualifying Proxy combines a Proxy, such as a Smart Pointer, with Result Follows Qualification to reflect the qualification of the handle in the access to the body.

Principles and Mechanisms

- From the perspective of qualification every class has multiple interfaces
  - For instance, with respect to *const* a class has a *const* interface and a non-*const* interface
- The qualified interface may be considered a supertype of the unqualified one



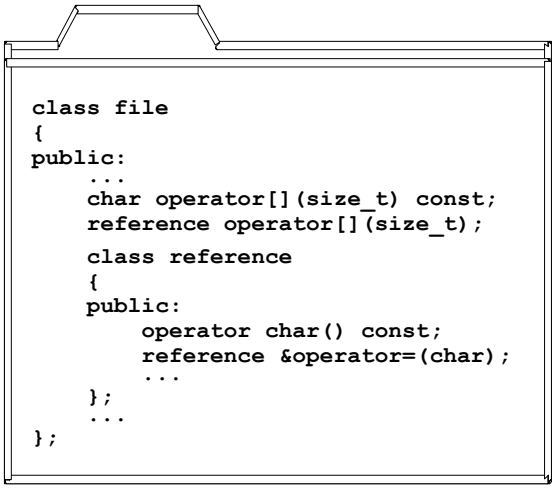
39

Explicit separation of modifier from query functions can benefit a system, and this is a concept that can be expressed in C++ using type qualifiers. Thus `volatile` and `const` – as well as `mutable` – are unified under the heading of change, even if the names are not as well chosen as they might be (that access to an object may be `const volatile` is a curio that leaves many developers bemused).

We can see immediately how substitutability plays a part with respect to `const` when we consider that any C++ class typically has two public interfaces: the interface for `const` qualified objects and the interface for non-`const` qualified objects. The `const` interface is a subset of the non-`const` interface, and therefore a non-`const` object may be used wherever a `const` one is expected. Note that the subtyping relationship implied need not be strict, as overloading can be used to block functions from default access, i.e. overloaded functions differing only in respect of `const` or non-`const` will be selected according to the calling `const`-ness.

Interfaces should present a logical view of an object's behaviour, and hence `const` should reflect logical queries. `mutable` can be considered a feature of negative variability that allows expression of physical change behind a logically immutable façade. It allows separation of logical and physical `const`-ness, for instance in the case of caching.

Result Follows Qualification



```
class file
{
public:
    ...
    char operator[](size_t) const;
    reference operator[](size_t);
    class reference
    {
    public:
        operator char() const;
        reference &operator=(char);
        ...
    };
    ...
};
```

Reflect the access to the whole in the access to the parts.

A non-`const` function may 'specialise' a function by overloading a `const` function with a non-`const` variant. This is most commonly of use where Result Follows Qualification, such as overloading the subscript operator for a string or sequence, or providing iterator accessors for a container. The effect of this idiom is to ensure that a query that returns a part of an object does not grant more privilege to the part than is granted to the whole. For instance, a string class is expected to support subscripting, but it is also expected to be `const` correct. Thus subscripting a `const` object should not allow updating:

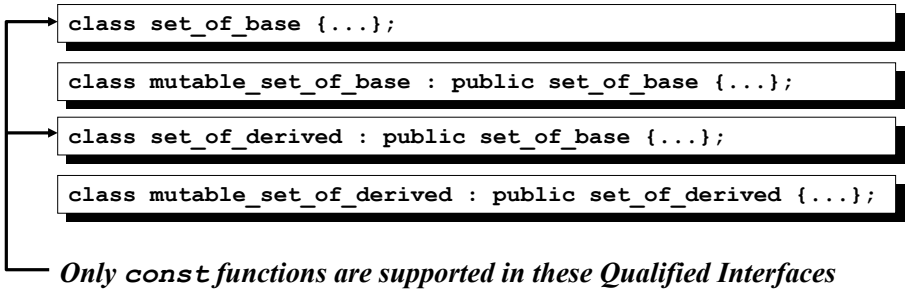
```
class string
{
public:
    char operator[](size_t) const;
    char &operator[](size_t);
    ...
};
```

The variation in return types is not restricted to values, references and `const`-ness, e.g.

```
class container
{
public:
    class iterator;
    class const_iterator;
    iterator begin();
    const_iterator begin() const;
    ...
};
```

Qualified Interface

- Interfaces can be separated by qualification
  - Substitutability of objects sometimes only makes sense when mutability is considered



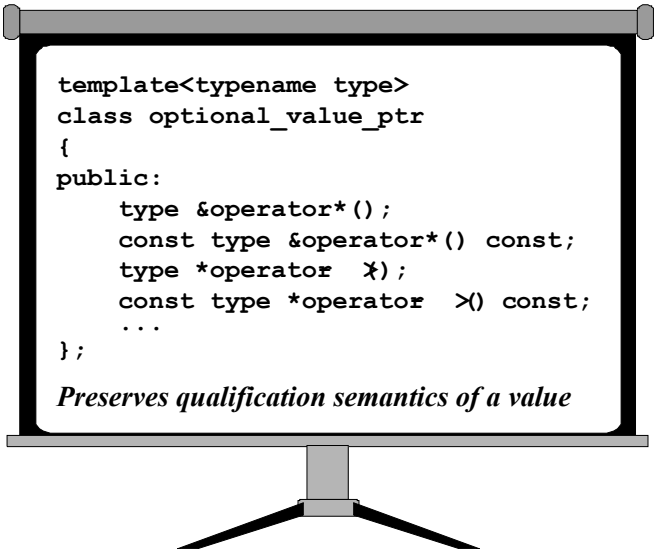
41

It is possible to explicitly separate out the implicit interfaces into two explicit Interface Classes differentiated by their qualification, i.e. one for `const` and one for non-`const` usage. A Qualified Interface resolves a number of substitutability problems, such the issues relating to substitutability of containers of polymorphic objects [Koenig+1997], i.e. `list<derived *>` cannot be used where `list<base *>` is expected.

The inheritance relationship between the qualified and unqualified version is optional. If not present, it is an example of Role Decoupling [D'Souza+1999]. If present, this approach can be used to implement iterators in terms of `const` iterators [Murray1993, ISO1998]. In terms of substitutability, an `iterator` would inherit from a `const_iterator`.

A further idiom that is particularly resistant to state change is Immutable Value [Henney1997] which, for many problems and programmers, offers a lateral approach to addressing many problems associated with state changes. In many ways this is a specialisation of Qualified Interface, but where Qualified Interface addresses partitioning a single class interface into separate class interfaces based on method side effects, Immutable Value eliminate side effects altogether. An Immutable Value is `const` from its moment of creation, and as a result is safe for sharing across different contexts. In particular, an Immutable Value – assuming that its state is genuinely `const` and does not take advantage of, for instance, on `mutable` – is thread safe.

Qualifying Proxy



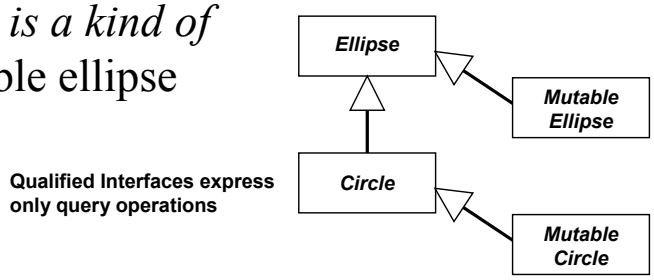
Qualifying Proxy is a compound pattern specialising Proxy with Result Follows Qualification. The `const`-ness of the Proxy – the handle – is reflected in the `const`-ness of the target object – the body. For instance, consider the case of a Smart Pointer that is supposed to support optional values, i.e. the value may or may not be present. Although in use it appears to be like a pointer externally, it behaves more like a value and should have 'deep `const`-ness':

```
template<typename type>
class optional_value_ptr
{
public:
    type &operator*()              { return *body; }
    const type &operator*() const  { return *body; }
    type *operator->()             { return body; }
    const type *operator->() const { return body; }
    optional_value_ptr &operator=(const optional_value_ptr &rhs)
    {
        type *old_body = body;
        body = rhs.body ? new type(*rhs.body) : 0;
        delete old_body;
        return *this;
    }
    ...
private:
    type *body;
};
```

There are many ways to implement assignment for such a class. In this case it is through copy replacement, using Copy Before Release [Henney1998a], which is brief and exception safe, although not particularly efficient for value copying.

Total Ellipse

- If a circle *is a kind of* ellipse, why are there substitutability problems?
  - Whenever state changes occur, such as resizing, substitutability of a circle for an ellipse fails
- A circle *is a kind of* immutable ellipse



43

Modelling the relationship and similarity between circles and ellipses (or squares and rectangles) with inheritance is a recurring question in books, articles and news groups. A close inspection reveals both subtlety in the problem and a solution [Henney1995].

The reason a solution is so hard to come by is because the problem is poorly stated: mathematics tells us that a circle *is* an ellipse, therefore one can substitute a circle wherever an ellipse is required, suggesting that a circle is a proper and full subtype of an ellipse under all circumstances. So far so good, but the troubles start when we introduce any state modifying functions, such as assignment or the ability to change the major and minor axes independently. The invariant for a circle states that its axes are the same, i.e. it has a radius.

This is a superficial analysis of the problem, but the root of the problem is more subtle: *what are the requirements and where is the analysis*? We are so confident that we understand the mathematical concepts behind circles and ellipses that we have not bothered to ask any more questions of that domain. We have also not said what we wish to use our circles and ellipses for. Not only is the previous design flawed by internal contradiction, it is not fit for a purpose for the simple reason that we have failed to identify one.

Factoring out Qualified Interfaces resolves this problem by introducing an appropriate and explicit separation between mutable and immutable operation sets. If one wishes to be strictly mathematical, this can be taken a step further, dispensing with the mutable classes completely, and working with the objects as true Immutable Values: maths has no side effects, conic sections do not undergo state changes, and there are no variables in the programming sense of the word.

Genericity

- Intent
  - Understand substitutability for generic programming
- Patterns
  - Tabled Requirements
  - Reflexive Mix-in
  - Generic Value Type

Generic programming is a form of compile time polymorphism built on C++ templates, as well as the concepts and mechanisms of the other substitutability mechanisms. The C++ standard library, notably the part known as the STL [Stepanov+1995, ISO1998, Austern1999], makes extensive use of it.

The style of substitutability is best captured through the use of tables of operational forms that place requirements on the syntax and types of usage, relying heavily on conversions and overloading. These define Type Hierarchies. A Reflexive Mix-in intertwines the relationship between base and derived classes, in effect forming not only a Type Heterarchy but also a Class Heterarchy. Generic Value Types parameterise the common relations between a family of types.

Principles and Mechanisms

- Templates extend overloading principles
  - Common name implies common structure as well as common purpose...
  - Specialisation expresses any structural exceptions to this
- Genericity is principally expressed with templates

45

Templates offer a route to substitutability that grows out of the basic concept of overloading. A templated function can be considered to have a potentially infinite number of overloads, all sharing a common name, purpose and structure, but differing in their parameter types. A similar view can be extended to template classes, and from that member templates (both functions and classes). This highlights the commonality and positive variability aspects of templates [Coplien1999]. There is also an element of negative variability associated with templates in the form of specialisation, either through overloading or explicit template specialisation. A specialisation can introduce different structure (control or data), different interface, and indeed different semantics, to a name.

These template features, together with the other substitutability categories, go to make up generic programming. The STL (Standard Template Library) represents the classic collection of generic components. The STL's philosophy is to make algorithm use and design significantly easier. The non-inheritance approach comes as a surprise to many, but this should not be taken to mean that the library components are inflexible. On the contrary, components are heavily parameterised, the difference being that most of the parameterisation is at compile time. As such, the STL constitutes the more fundamental approach to library components. The emphasis on algorithms endows the STL with a particular flavour, once and for all giving C++ containers and iterators a style of their own – a more indigenous style [Koenig+1995] rather than the appearance of a Smalltalk hand-me-down.

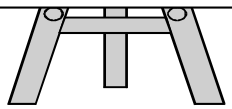
Tabled Requirements

- Document generic parameters in terms of...
  - Legal expression form
  - Expression type
  - Non-functional requirements

CopyConstructible

Expression	*Return Type*	*Requirement*
`T(t)`		`t` is equivalent to `T(t)`
`T(u)`		`u` is equivalent to `T(u)`
`t.~T()`		
`&t`	`T *`	denotes address of `t`
`&u`	`const T *`	denotes address of `u`

Where `T` is a type supplied as a template parameter, `t` is a value of type `T`, and `u` is a value of type `const T`.



Template classes are somewhat lacking in explicit constraints on their parameters. This can be seen as both a benefit, allowing the expansive expressive range of generic programming, as well as a liability, such as obscure and unhelpful compiler messages when implied constraints are not observed. It is possible to make some constraints more explicit, as well as adopt naming and syntax usage practices – such as when to use `class` and when to use `typename` for a template parameter – that assist the human audience (in truth, the one that really counts) of program source [Henney1996].

Tabled Requirements offer a comprehensive external source of documentation for expected behaviour of template parameters. These were introduced with the STL [Stepanov+1995] and were incorporated into the C++ standard library specification [ISO1998]. Parameters are defined in terms of the legal expressions they must support – which are ultimately compiler checkable – and any non-functional requirements placed on them – which are not.

Choosing the data structure to simplify the algorithm is hardly new advice, but it is this approach more than any other that typifies the library. An important feature of algorithms is their complexity, i.e. their relative performance in terms of the number of elements they operate on. This, as well as the notion of interface, is used to fully define what a type is; although the STL is not traditionally considered object-oriented, it is firmly based on abstract data types.

Type requirements can be formed into a Type Hierarchy. For example, the requirements placed on iterator categories form a substitutability group.

Reflexive Mix-in

- How can a Mix-in be factored out if it depends on the derived class?
  - Decouple the dependency through a template parameter

```
template<class derived_type>
class allocated
{
public:
    void *operator new(size_t);
    void operator delete(void *, size_t);
    ...
};
```

```
class example : public allocated<example>
{
    ...
};
```


How can common behaviour and similar code structure be factored into a base class if there are details of the implementation that depend on the type name of the derived class?

Consider defining a base class that offers customised allocation, through `new` and `delete`, to and of derived classes. Factoring out such an implementation immediately suggests a Mix-in. However, a conventional Mix-in would not meet the expectation that the allocation was for each derived class, i.e. it was differentiated on the derived class. Because `new` and `delete` are `static`, and therefore also any mechanism internal to the allocation class, all derived classes will end up sharing the same memory pool (or whatever strategy has been chosen) rather than having one dedicated, and optimised, for the derived class. In short, the base depends on the derived class.

This cycle can be broken conventionally using delegation: each class that wants customised allocation must defines a `static` allocation object, and then its own `new` and `delete` operators are written to forward to the allocation object. This is tedious, and is also prone to cut and paste errors. It also means that the empty base class optimisation cannot be used [Cantrip]. Also from both engineering and aesthetic perspectives (which are not necessarily mutually exlcusive) it is not pleasing: repetition implies that there is an abstraction missing.

The solution is to define a template class that is intended for use as a base. It defines the common code in terms of the varying part, i.e. the derived class, which is supplied as a template parameter [Barton+1994, Coplien1999].

Generic Value Type

Providing a fixed point value type that is compile time checked for precision leads to the use of templates to factor out the precision parameters. Templates support compile time calculation and lookup which allows generalised operators.

```
template<int digits, int scale> class fixed;

template<
    int lhs_digits, int lhs_scale,
    int rhs_digits, int rhs_scale>
  fixed<
      lhs_digits + rhs_digits,
      lhs_scale + rhs_scale>
    operator*(
      const fixed<lhs_digits, lhs_scale> &lhs,
      const fixed<rhs_digits, rhs_scale> &rhs);
```

48

Distinct types, as modelled by Whole Value and other value-based type idioms, allow greater expressiveness and checking in code. There are cases when distinct types need more miscibility, but to enumerate all the possibilities would lead to class explosion. For instance, physical quantities such as length, mass, time, velocity, etc can be combined in a regular fashion, as can fixed point numbers that are compile time checked.

The parameter of change, whether type or number, can be templated for the value type and – assuming that the rules for combination are sufficiently simple – operators and functions provided for mixing types. This is illustrated above in the fixed point number example [OMG]. Using dimensional analysis, Generic Value Type can also be used to abstract physical quantities [Barton+1994].

For some operators the rules for combination can become a little more involved, and additional templates may need to be called upon to simplify declarations. For instance, providing `operator+` for `fixed`:

```
template<
    int lhs_digits, int lhs_scale,
    int rhs_digits, int rhs_scale>
  fixed<
      sum_digits<
        lhs_digits, lhs_scale, rhs_digits, rhs_scale>::value,
      max<lhs_scale, rhs_scale>::value>
    operator+(
      const fixed<lhs_digits, lhs_scale> &lhs,
      const fixed<rhs_digits, rhs_scale> &rhs);
```

The detail of `max` and `sum_digits` is left as an exercise for the reader....

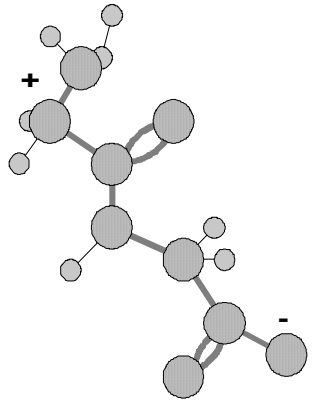
Practice

> **The history of all hitherto existing society is the history of class struggle.**
>
> ***Karl Marx***

It has been said that "in theory there is no difference between theory and practice, but in practice there is". Applying a set of principles and practices is the natural complement to presenting them. This section looks briefly at ways we might combine patterns, before moving on to two worked examples that demonstrate the ideas presented so far.

Combining Patterns

- Intent
  - Apply patterns together to form complete solutions
- Content
  - Pattern languages
  - Patterns stories



Often the focus of developers is on individual patterns. But in just the same way that design is not a sequence of isolated activities, and the resulting architecture is not a group of isolated fragments (low coupling is good, but no coupling yields a system that does nothing!), patterns should not be treated in isolation.

Thus, in terms of understanding particular patterns, an insular approach does not bring out their full power or, indeed, honesty with respect to how they play a part in design. Pattern languages and worked examples showing patterns in action (*pattern stories* or *design stories*) are often the best way to illustrate how patterns interlock and flow, both narratively and diagrammatically.

Pattern Languages

- Some patterns may be applied in a sequence from one to another
  - Context and resulting context describe a predecessor/successor relationship
- A pattern language connects many patterns together
  - Intent of a language is to generate a particular kind of system

51

In constructing a system, catalogues of patterns such as [Gamma+1995] and [Buschmann+1996] can play a role, as can individual patterns. However, a pattern language [Alexander+1977] defines a more complete approach to connecting patterns together in a generative fashion for a particular task. They provide a graph of patterns that are connected in a logical fashion by their contexts and resulting contexts, helping to inspire and drive the developer towards solutions. Thus pattern languages include decisions and sequences of pattern application: which patterns might follow from application of a pattern, and which might precede it.

One of the distinctions between collections of individual, separately motivated patterns and a pattern language is that of *generativity* [Coplien1996]:

> *A pattern language is a collection of patterns that build on each other to generate a system. A pattern in isolation solves an isolated design problem; a pattern language builds a system. It is through pattern languages that patterns achieve their fullest power.*

An example of generativity is to note that whenever a separation is introduced in a structure, typically through adding a level of indirection, there is the need for a creational pattern to construct the collaborative structure. For instance, the use of the Bridge pattern often follows on to the Abstract Factory pattern. Such a statement can be considered generative as it informs the developer of a possible route that their design might take once they have applied a pattern. Pattern languages capture the decisions involved in following a sequence through a collection of patterns, but are not themselves rigid decision trees.

Pattern Stories

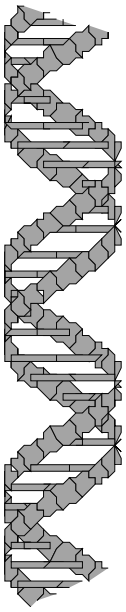
- Any given design can be seen as an interlocking and flowing set of decisions
  - Context leads to problem, is resolved by solution, establishes new context, which in turn...
- A story captures a given flow
  - Study of design accompanied by study of designs

Where pattern languages are narrative frameworks, the development of a system can be considered a single narrative example, where design questions are asked and answered, structures assembled for specific reasons, and so on. We can view the progress of some designs as the progressive application of particular patterns. Examples of this approach can be found in [Gamma+1995], [Beck1997], [Beck+1998], [Gamma+1999], and [Fowler1999].

What we can term *pattern stories* are narrative structures, i.e. they exist as complete and in the real world. As with many stories, they capture the spirit and not necessarily the truth of the detail in what happens. It is rare that our design thinking will so readily submit itself to named classification and sequential arrangement, so there is a certain amount of retrospective and revisionism. However, as an educational tool, such an approach can reveal more than conventional case studies, as each example literally tells the story of a design. This is a valuable aid in studying design through the study of designs.

Cloning Framework

- Intent
  - Development of a copying framework for value objects in a Class Hierarchy
- Content
  - Generic cloning requirements
  - Inheritance-based requirements
  - Simplifying cloning realisation
  - Simplifying cloning use



For objects in a Class Hierarchy we find that dynamic allocation and usage by pointer is the norm. We find that copying is less meaningful: What does it mean to copy something for which identity is significant? What is the relationship between the copy and the original in terms of identity? What would be a meaningful copy constructor or assignment operator for a person, for instance? In such cases, we normally unask the question and restrict copying behaviour, i.e. by making the copy constructor and assignment operator private.

This distinction serves us well for most cases, but in the territory between these crisp definitions lie some shades of grey. What mechanisms are open to us if we have referential objects in a class hierarchy for which copying is meaningful? Alternatively, what if we have a classification hierarchy for value-based objects that require copying?

The Cloning idiom describes how to introduce polymorphic copying capability. It is possible to use it as the basis of a framework [Henney1999b].

Generic Cloning Requirements

Cloneable

Expression	*Return Type*	*Requirement*
`ptr->clone()`	convertible to `T *`	*pre:* `ptr != 0` *post:* copy of `*ptr`, made with respect to `typeid(*ptr)`, returned
`delete ptr`	`void`	`*ptr` no longer usable

Where `T` is the *Cloneable* type supplied as a template parameter, and `ptr` is a value of type `T *` or `const T *`.

In standard STL/standard style, we can capture *Cloneable* capability as Tabled Requirements. This means that any type satisfying *Cloneable* may be used consistently in a generic framework written against those requirements. For instance, here is a cloning variation of the standard `copy` algorithm:

```
template<typename input, typename output>
output clone(input begin, input end, output result)
{
    while(begin != end)
        *result++ = (*begin++)->clone();
    return result;
}
```

Here the requirement is that `**begin` and `**result` satisfy *Cloneable*. Compile time discovery is used in checking the expression and return type constraints. The more dynamic requirements are, as ever, related to quality of implementation with any violations manifesting themselves at runtime.

Inheritance-Based Requirements

```
class cloneable
{
public:
    virtual ~cloneable();
    virtual cloneable *clone() const = 0;
private:
    cloneable &operator=(const cloneable &);
};
```

An Interface Class may be used to represent cloning capability, deferring detail to derived classes.

```
class example : public virtual cloneable
{
public:
    example(const example &);
    virtual example *clone() const
    {
        return new example(*this);
    }
    ...
};
```

55

Cloning capability can be represented as an Interface Class, `cloneable`. This class also satisfies the previous generic *Cloneable* requirements. There is also support for runtime discovery in the form of `dynamic_cast` [Henney1998b].

Interfaces express capabilities of objects, and where the object's class multiply inherits from other classes that ultimately inherit from the same interface, the object still only expresses these capabilities once. The default inheritance mechanism leads to the idea that any repeated inheritance represents multiple instances of the repeated class, with multiple instances of the data; this is not meaningful in the case of `cloneable`. Nonetheless, tied to this is the idea that functions are still associated with separate subobjects and that the derived class is substitutable for one of two occurrences of the base class – an ambiguity that must be resolved by the programmer – rather than being directly substitutable for the interface, which is after all the point of defining one! This ambiguity can manifest itself at compile time if the clashing base classes are visible, or at runtime with a failed `dynamic_cast`. Therefore, `cloneable` should be inherited as a `virtual` base class to resolve this tension.

For `cloneable` support for public deletion is a requirement, and so it supports a `public virtual` destructor. Assignment, however, is neither meaningful nor desirable at the Interface Class level, and so Preventative Overloading is used to inhibit it.

Simplifying Cloning Realisation

```
template<class derived>
class cloner : public virtual cloneable
{
public:
    virtual derived *clone() const
    {
        return new derived(
            static_cast<const derived &>(*this));
    }
    ...
};
```

A Reflexive Mix in factors common implementation structure.

```
class example : public cloner<example>
{
    ...
};
```

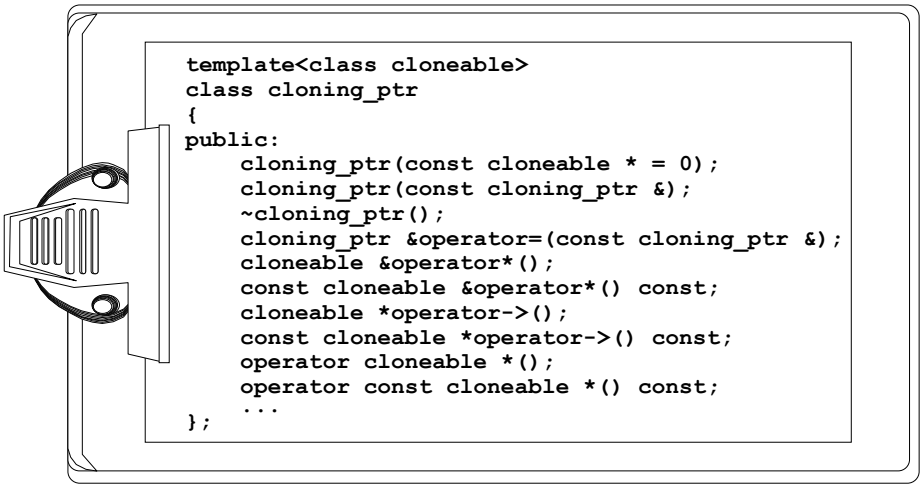

There is no sensible default implementation for `cloneable::clone`, which is one of the reasons that it is a pure `virtual`. However, for `cloneable` it is likely that almost without fail all implementations in the Class Hierarchy will be structurally similar, simply returning a newly allocated copy of the current instance made using the copy constructor of the derived class. Implementations will typically repeat the same schema. A Reflexive Mix-in, `cloner`, can resolve the apparently cyclic class relationship.

Because `cloner` classes derive from `cloneable`, `clone` must return a type that is ultimately derived from `cloneable`. Because `derived` must be derived from `cloner<derived>` it satisfies this requirement. Knowing the exact type of the derived class also allows `cloner<derived>` to use this as an instance of `derived` to create a copy of the correct type.

To take advantage of the `cloner` facility requires that derived classes derived from the template using themselves as the template parameter. With the exception that each derived class must provide a meaningful copy constructor, no further implementation work is needed: all of the implementation comes from the Reflexive Mix-in. The constraints on the relationship between the derived class and template parameter are sufficiently well enforced by the type system, and constraints on the use of `static_cast` as opposed to traditional casts, that misuse is caught at compile time.

The `cloner` template is effectively abstract even though it apparently provides a complete implementation of `cloneable`. It is incomplete because it cannot be used on any freestanding type: it requires a derived class to provide meaning, e.g. a declaration of `cloner<std::string>` will not compile.

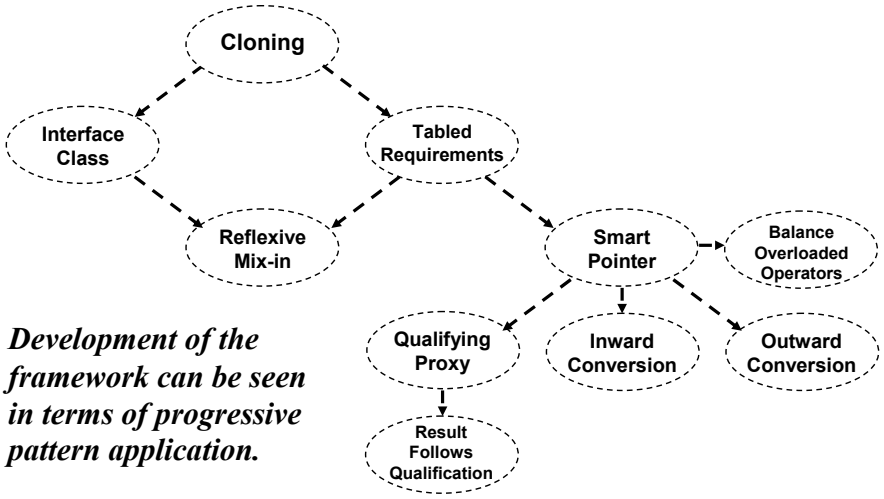
Simplifying Cloning Use



The introduction of a Smart Pointer can simplify the use of Cloneable objects as values. A number of other patterns lead from this to assist in creating a whole class, part of which is shown here:

```
template<class cloneable>
class cloning_ptr
{
public:
    cloning_ptr(const cloneable *other = 0)
      : body(other ? other->clone() : 0) {}
    cloning_ptr(const cloning_ptr &other)
      : body(other.body ? other.body->clone() : 0) {}
    ~cloning_ptr() { delete body; }
    cloning_ptr &operator=(const cloning_ptr &rhs)
    {
        cloneable *old_body = body;
        body = rhs.body ? rhs.body->clone() : 0;
        delete old_body;
        return *this;
    }
    cloneable &operator*() { return *body; }
    const cloneable &operator*() const { return *body; }
    cloneable *operator->() { return body; }
    const cloneable *operator->() const { return body; }
    operator cloneable *() { return body; }
    operator const cloneable *() const { return body; }
    ...
private:
    cloneable *body;
};
```

Framework Story



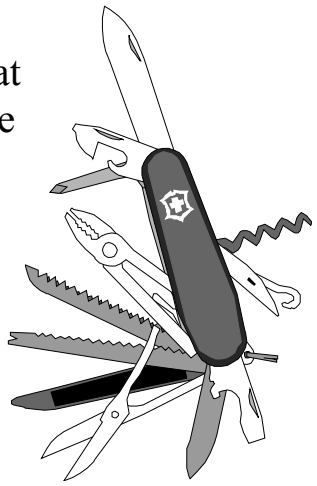
Development of the framework can be seen in terms of progressive pattern application.

The resulting design shows a progressive addition and interaction of detail:

- Cloneability expressed in terms of Tabled Requirements for generic programming.
- Cloneability also expressed through an Interface Class, which satisfies Tabled Requirements.
- A Reflexive Mix-in combines, and builds on, the Interface Class and Tabled Requirements to factor out common implementation.
- A Smart Pointer, written against the Tabled Requirements, allows automatic management of cloning objects in a value-based fashion. To ensure uniform and expected use, Balance Overloaded Operators on the Smart Pointer.
- The Smart Pointer can be implemented as a Qualifying Proxy to more closely model value constraints, such that for each operator Result Follows Qualification.
- An Inward Conversion on the Smart Pointer is safe because object ownership is not transferred in any way. Thus a raw pointer is substitutable for the Smart Pointer. The converse can also be accommodated with an Outward Conversion, which is relatively safe. However, some designs might chose to omit this.
- Further improvements, not discussed, include adding named queries and modifiers, such as `get`, `acquire` and `release`, and generalising Inward Conversions with member templates, making the Smart Pointer more of a Generic Value Type.

Universal Variant Value Type

- Intent
  - Development of a value type that can handle any value of any type
- Content
  - State of the object
  - Player of types
  - Consider conversions



59

A statically checked type system is all very well when you know the types, or have some idea of the minimum requirements on types, that you are operating on. Derivation polymorphism and generic programming through templates offer a certain leeway in the degree of precision with which a type must be known, but still do not offer the same variation in type at either compile time or runtime that other dynamically checked languages offer. For applications that need a variant type that can accommodate typically any value, C++ developers pursue one of two routes:

- The use of a `union` to hold commonly used types. This must be accompanied by a type discriminator flag of some kind, typically an `enum`. Although there are some benefits to using an `enum`, this is quite C-like way of going about it. Holding a pointer to `std::type_info` often makes better use of the standard's features. The `union` is still potentially unsafe, and so it is often wisest to wrap it up within a class.
- The use of `void *` with a type discriminator field. This speaks for itself.

These approaches are limited with respect to to their boundedness and type safety. Bringing together a number of idioms, it is possible to define a type, `any` [OMG], that is type safe and unbounded with respect to the types it accommodates: in effect, a universal `union`. It combines a number of substitutability patterns to achieve the effect. The values it holds are treated as unrelated, and so there is no conversion between otherwise convertible types, e.g. between `int` and `double`, or `derived *` and `base *`. It is possible to introduce mechanisms that achieve something similar to this, but that introduces additional complexity and is beyond the scope of the current class.

State of the Object

```
class any
{
    ...
    class placeholder : public virtual cloneable
    {
    public:
       virtual const type_info &type() const = 0;
       ...
    };
    template<typename value_type>
    class holder : public placeholder, public cloner<holder>
    {
        ...
        virtual const type_info &type() const
        {
            return typeid(value_type);
        }
        value_type value;
    };
    cloning_ptr<placeholder> content;
};
```

60

Focusing on the representation of an `any`, we need to ask ourselves the question: how can we hold *any* value? A Generic Value Type can be defined to act as a holder for any value (cf. Adapter [Gamma+1995]), with Tabled Requirements constraining the type to be *CopyConstructible*:

```
template<typename value_type>
class holder ...
{
public:
    holder(const value_type &holdee) : value(holdee) {}
    value_type value;
};
```

Currently we cannot have a pointer to any `holder`, only specific `holder` types, e.g. `holder<string>`. For this we need a common base class:

```
class placeholder {...};
template<typename value_type>
class holder : public placeholder {...};
```

However, we are left with two problems:

- If we use a pointer to a placeholder as the representation, how can we copy it? E.g. when we copy construct or assign one `any` from another?
- How can we recover the type information of what we are holding in the `any`? A type laundering mechanism is required.

The first issue is addressed by Cloning, and for that we can fully take advantage of the previously explored cloning framework. Providing a `virtual` function that returns the `typeid` of the template parameter addresses the second issue.

Player of Types

```
class any
{
    ...
    template<typename value_type>
    void set(const value_type &new_value)
    {
        holder<value_type> new_content(new_value);
        content = &new_content;
    }
    template<typename value_type>
    bool get(value_type &result) const
    {
        const bool matching = type() == typeid(value_type);
        if(matching)
            result =
              static_cast<
                holder<value_type> *>(content.get())->value;
        return matching;
    }
    ...
};
```

61

The representation of the any resolved, we can focus on how it is used from the outside, i.e. its interface. Starting with simple queries, we need to address how a user finds out if the any is currently set to a value (its default state is unset) and what type of value it holds:

```
operator bool() const
{
    return content;
}
bool operator!() const
{
    return !content;
}
const type_info &type() const
{
    return content ? content->type() : typeid(void);
}
```

Following pointer semantics for nullness, we have given `any` basic `bool` capabilities. Thus each Operator Follows Built-ins, and we Balance Overloaded Operators.

To set a value is relatively straightforward: a named member template function is provided that creates the right holder type, which is then copied to the representation. To get a value is a little more involved, and involves some type laundering. The type query that is in place can be used to check for conformance between actual and expected type, with the result being set on a match. `true` or `false` is returned as an indication of success or otherwise.

Consider Conversions

```
class any
{
    ...
    template<typename value_type>
    any &operator=(const value_type &new_value)
    {
        set(new_value);
        return *this;
    }
    ...
};

template<typename result_type>
result_type any_cast(const any &operand)
{
    result_type result;
    if(!operand.get(result))
        throw bad_cast();
    return result;
}
```

Named accessor functions used for implementation of Inward Conversion and Custom Keyword Cast.

62

The named operators provide the foundation for a more familiar way of using values. It is safe to set an `any` from any value type, i.e. any value type may be substituted where an `any` is expected. This implies that `any` can support an Inward Conversion. The desirable Inward Conversion is supported by defining a generic assignment operator and a generic converting constructor.

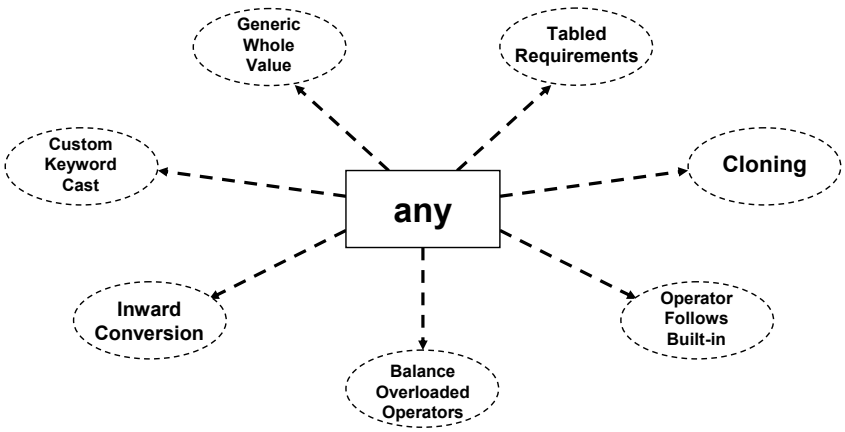
Whilst it is certainly possible to provide a generic Outward Conversion, it is seriously unwise to do so. An `any` is not substitutable for any value type: any value type to an `any` is a widening conversion, and therefore always safe, whereas an `any` to any value type is a narrowing conversion, and is therefore potentially unsafe and must be checked. Thus, from the user's perspective, introducing an Outward Conversion would be subtle and would smack of trickology.

An explicit equivalent of Outward Conversion can be provided in the form of a Custom Keyword Cast, `any_cast`:

```
any thing;
...
string along = any_cast<string>(thing);
```

As it is explicit it can be checked, and to be consistent with the standard library it can use the `std::bad_cast` exception to signal a failure. Note that the use of `any_cast` places an additional requirement, that of default constructability, on the value type accessed.

Use of Patterns



63

The resulting `any` class can be seen to have been built up around substitutability concepts:

- Generic Value Type, as an expression of Adapter, holds the actual value. It can be held as the body of an `any` thanks to a common base class (an expression of Bridge [Gamma+1995]) and a polymorphic type discovery mechanism.
- Tabled Requirements stipulate that any value type used must be *CopyConstructible*, as well as assignable if `get` is used. To use an `any_cast` further requires default constructibility.
- Copying the body representation is addressed with Cloning. In this case the requirements for cloning are actually met by the existing cloning framework, and thus the use of Cloning can be seen as an entry point into the pattern community previously described.
- Operator Follows Built-in informs the definition of the existential functions, as does the need to Balance Overloaded Operators.
- An Inward Conversion supports the widening conversion from any type to an `any`.
- A Custom Keyword Cast supports a safe and checked narrowing conversion from an `any` to a target type.

Summary

- Substitutability offers a useful way of structuring the meaning of a system
- C++ supports substitutability with respect to a number of mechanisms
  - Conversions, overloading, derivation, mutability and genericity
- Practices make sense of mechanism

Substitutability is a property of elements in a system based on transparency and abstraction. Consideration of substitutability can give developers a different perspective on how the constraints and dependencies in a system can be structured, and add meaning to the relationships in a system.

In support of substitutability, C++ goes beyond the conventional view of OO substitutability – which is based on inclusion polymorphism and inheritance, and articulated as LSP – to offer features that cover other recognised forms of polymorphism and substitutability with respect to state change.

Substitutability can be framed in terms of mechanism and principles, but it is in identifying and applying relevant practices that sense is made of the concepts, and meaning expressed in a system. Patterns represent motivated and generalised problem/solution pairs. Any system can be viewed in terms of its interwoven patterns and not simply its structure expressed as classes. Pattern stories can be considered as a way of capturing the flow of decisions in design, and can be used as a documentation form. They express the how and why of a design, and not just the resulting structure.

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

[Alexander1979] Christopher Alexander, *The Timeless Way of Building*, Oxford University Press, 1979.

[Austern1999] Matthew H Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley, 1999.

[Barton+1994] John J Barton and Lee R Nackman, *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, Addison-Wesley, 1994.

[Beck1997] Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.

[Beck+1998] Kent Beck and Erich Gamma, "Test Infected: Programmers Love Writing Tests", Java Report, SIGS, July 1998.

[Boost] Boost library website, `http://www.boost.org`.

[Buschmann+1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.

[Cantrip] Nathan Myers' website. `http://www.cantrip.org`.

[Cardelli+1985] Luca Cardelli and Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, 17(4):471-522, December 1985.

[Carroll+1995] Martin D Carroll and Margaret A Ellis, *Designing and Coding Reusable C++*, Addison-Wesley, 1995.

[Coplien1992] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

[Coplien1996] James O Coplien, *Software Patterns*, SIGS, 1996.

[Coplien1999] James O Coplien, *Multi-Paradigm Design for C++*, Addison-Wesley, 1999.

[Cunningham1995] Ward Cunningham, "The CHECKS Pattern Language of Information Integrity", [PLoPD1995].

[Dijkstra1972] Edsger W Dijkstra, "The Humble Programmer", *Communications of the ACM*, 15(10), October 1972.

[D'Souza+1999] Desmond D'Souza and Alan Cameron Wills, *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1999.

[EXE1990] "Around the Table with Bjarne Stroustrup", *.EXE*, 5(6), November 1990.

[Fowler1997] Martin Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.

[Fowler1999] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

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

[Gamma+1999] Erich Gamma and Kent Beck, "JUnit: A Cook's Tour", Java Report, SIGS, May 1999.

[Henney1995] Kevlin Henney, "Vicious Circles", *Overload* 8, June 1995.

[Henney1996] Kevlin Henney, "Constraining template parameter types", *Overload* 12, February 1996.

[Henney1997] Kevlin Henney, "Java Patterns and Implementations", presented at BCS OOPS *Patterns Day*, October 1997. Also available from "Articles and Conference Prensentations" on `http://techland.qatraining.com`.

[Henney1998a] Kevlin Henney, "Creating Stable Assignments", *C++ Report* 10(6), June 1998.

[Henney1998b] Kevlin Henney, "Keeping up with runtime type information", *EXE*, October 1998.

[Henney1999a] Kevlin Henney, "Mutual Registration: A Pattern for Ensuring Referential Integrity in Bidirectional Object Relationships", *EuroPLoP '99*, 1999.

[Henney1999b] Kevlin Henney, "Clone Alone", *Overload* 33, August 1999.

[Hillside] The Hillside Group, *Patterns Home Page*, `http://hillside.net/patterns`.

[Hofstadter1979] Douglas R Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*, Penguin, 1979.

[ISO1998] *International Standard: Programming Language - C++*, ISO/IEC 14882:1998(E), 1998.
[Jackson1995] Michael Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*, Addison-Wesley, 1995.
[Koenig+1995] Andrew Koenig and Bjarne Stroustrup, "Foundations for Native C++ Styles", *Software Practice and Experience*, 25(S4):45-86, December 1995.
[Koenig+1997] Andrew Koenig and Barbara Moo, *Ruminations on C++*, Addison-Wesley, 1997.
[Liskov1987] Barbara Liskov, "Data Abstraction and Hierarchy", *OOPSLA '87 Addendum to the Proceedings*, October 1987.
[Martin1995] Robert Martin, "Object-Oriented Design Quality Metrics: An Analysis of Dependencies", *ROAD*, SIGS, September-October 1995.
[Meyers1996] Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.
[Meyers1998] Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Design*, 2nd edition, Addison-Wesley, 1998.
[Murray1993] Robert B Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993.
[OMG] *The Common Object Request Broker: Architecture and Specification*, OMG, `http://www.omg.org`.
[PLoP1995] Edited by James O Coplien and Douglas C Schmidt, *Pattern Languages of Program Design*, Addison-Wesley, 1995.
[PLoP1996] Edited by John Vlissides, James O Coplien and Norman L Kerth, *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.
[PLoP1998] Edited by Robert Martin, Dirk Riehle and Frank Buschmann, *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.
[Rumbaugh+1999] James Rumbaugh, Ivar Jacobson and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.
[Stepanov+1995] Alexander Stepanov and Meng Lee, *The Standard Template Library*, `ftp://butler.hpl.hp.com/stl`, 1995.
[Stroustrup1997] Bjarne Stroustrup, *C++ Programming Language*, 3rd edition, Addison-Wesley, 1997.
[Strunk+1979] William Strunk Jr and E B White, *Elements of Style*, 3rd edition, Macmillan, 1979.
[Sutter2000] Herb Sutter, *Exceptional C++*, Addison-Wesley, 2000.
[Taligent1994] *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*, Addison-Wesley, 1994.
[Vlissides1998a] John Vlissides, "Composite Design Patterns", *C++ Report* 10(6), June 1998.
[Vlissides1998b] John Vlissides, "Pluggable Factory, Part 1", *C++ Report* 10(10), November/December 1998.