Valued Idioms

Patterns for Value-Based Programming

Kevlin Henney
kevlin@curbralan.com

Presented to the *Denmark Design Patterns Study Group*, Copenhagen, 1st May 2001.

Kevlin Henney

`kevlin@curbralan.com`
`kevlin@acm.org`

Curbralan Ltd

`http://www.curbralan.com`
Voice: +44 117 942 2990
Fax: +44 870 052 2289

Agenda

- Intent
  - Express object types for which information content is significant but instance identity is not
- Content
  - Classifying object style
  - Patterns, idioms and pattern languages
  - Value-based programming in C++
  - Value-based programming in Java

Value objects represent small pieces of information in a system, such as dates, money and measurements. The practices for defining value object classes vary from system to system, depending on programming language, use of concurrency and distribution, as well as awareness of the need to express such objects differently from entity, service or task objects.

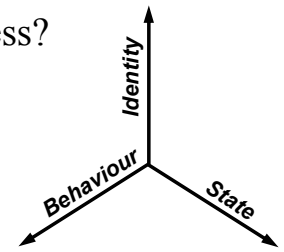
This talk considers the role of value objects, how they compare to other kinds of objects, and how context affects the use and expression of values. Patterns are presented for working with values in both C++ and Java.

This talk also demonstrates the difference between a simple collection of patterns and the expressiveness of a pattern language. For C++ the patterns are simply presented as a collection, but for Java they are presented as a connected pattern language.

The bibliography at the end details all of the references cited.

Identity, State and Behaviour

- Objects can be characterised in terms of their identity, state and behaviour
- These aspects are rarely equal in importance
  - Is identity significant or transparent?
  - Is an object stateful or stateless?
  - Does an object have significant behaviour independent of its state?



3

Objects are said to have identity, state and behaviour [Booch1994]. The relative significance of each of these parameters varies for different styles of object and development, and we can take the basic classification further, using it as a means to distinguish different kinds of object. The broad allocation of responsibility over system components is affected by, and will affect, the style of the objects.

We can view applications as centres resting on or being focused on a sea of information and behaviour defining the domain, where the significance of information is given by its identity as well as its state.

In certain styles of development and architecture one particular dimension may dominate. For instance, behaviour is either not present or is a secondary concern in conventional database development, which is focused very much on the concepts of identity and state. On the other hand, delivery of content and features to clients in modern layered systems favours service-based approaches layered on top of more entity-based layers.

Classifying Object Style

Entity based

Represent information in a system. Identity and state are significant, whereas behaviour is secondary and directly related to these.

Service based

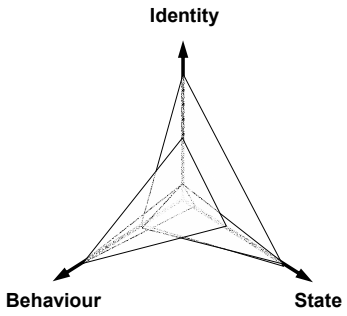
Represent activities in a system. Behaviour is significant. Typically stateless and identity transparent.

Value based

Represent transient or trivial information in a system. State is significant, behaviour directly related to it, and identity transparent.

Task based

Represent control based activities in a system. Behaviour is most significant, state and identity less so.

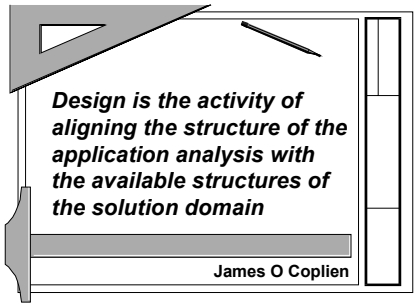


Identity, state and behaviour parameters identify significant characteristics for objects, but for many one or other of these features is either not important or will be dominated by others. Certain profiles can be characterised:

- *Entity*: Entities express system information, typically of a persistent nature. They are distinct from each, hence entities are distinguished from values by their identity. Their behaviour is secondary to their state concerns, and can be seen to correspond closely to the state model, e.g. operations on entities are often focused on querying and modifying attributes, and enforcing consistency rules through more transactional operations.
- *Service*: Service-based objects represent system activities. Services are distinguished by their behaviour rather than their state content or identity. The absence (or reduced significance) of identity therefore means that one service instance can be substituted transparently for another so long as its manifest behaviour is equivalent. Such transparency also means that services and their interactions tend to be stateless, operating on entities or passing values around.
- *Value*: For value-based objects, interpreted content is the dominant characteristic, followed by behaviour in terms of this state. Contrasting with entities, values are transient and do not have significant identity, e.g. dates and strings.
- *Task*: Like service-based objects, task-based objects represent system activities. However, they have an element of identity and state that means their role is as controllers. They most naturally balance with values in a system, i.e. values are used as the currency of information.

Expressing Object Style

- How do we represent these different kinds of object in our system?
- It is a matter of design, which is a creational and intentional act
  - The conception and construction of a structure on purpose for a purpose



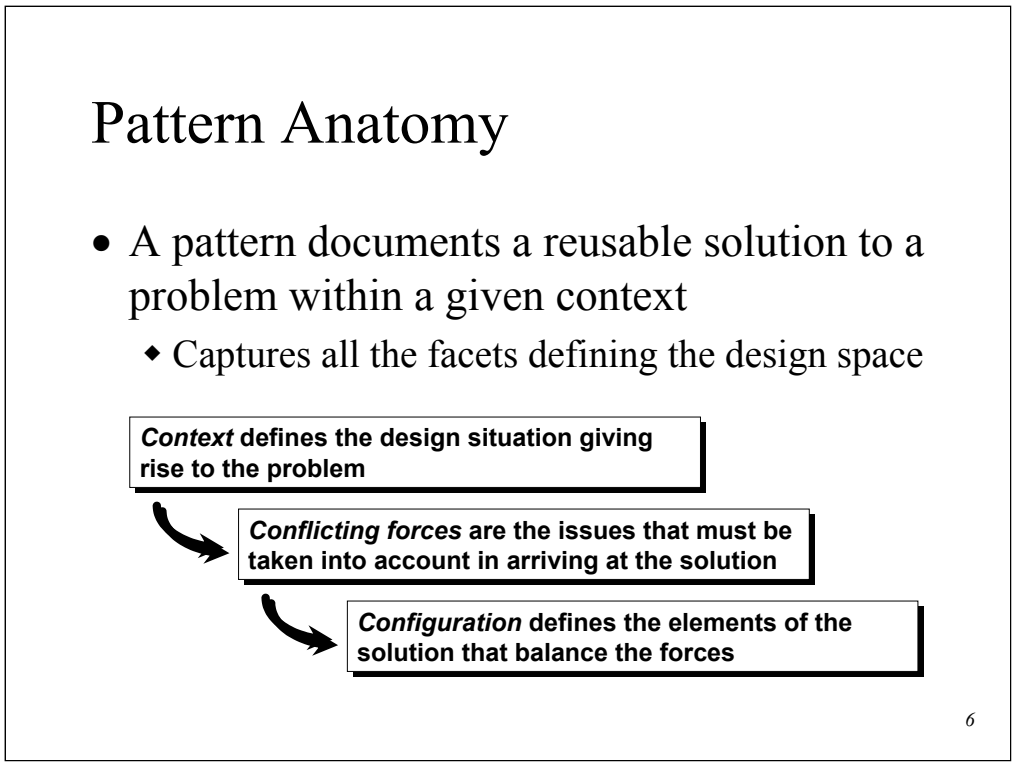
5

Simply identifying the need for values in a system is not enough. What do they look like? What are their properties? How will they interact with other features of the system? These are not accidental, throwaway code-as-you-go features, but are consequences of clear, intentional design.

The meaning of what is meant by design is not always clear. There are a number of definitions that complement or conflict with various definitions of architecture and each other:

- Architecture is the broad shape of the system, i.e. is equivalent to the concept of macro-architecture, and design is concerned with a finer level of system detail, i.e. micro-architecture.
- Architecture is the product of design, covering all scales in a system, and design is the activity that produces it. Therefore *design* is treated strictly as a verb and *architecture* as a noun [Coplien1999].
- Architecture is a description of system structure, regardless of intention (i.e. all systems have architecture, whether deliberate or not), whereas design describes an intentional approach. The word *intentional* has two meanings, both of which relate to the view of design presented here: performed by or expressing intention, i.e. deliberate; of or relating to intention or purpose.

The process of design must identify and balance various forces within a system. These forces, which often conflict, may arise from interplay between the requirements, or from other architectural decisions.



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:

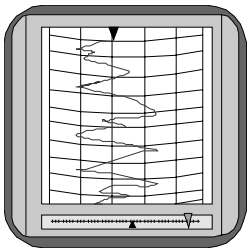
> *We know that every pattern is an instruction of the general form:*
> *context ⇒ conflicting forces ⇒ configuration*

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.

Design embraces all levels of detail in the system, not simply the intermediate level of classes and their relationships. Although the terms *design* and *design patterns* have commonly come to mean general, intermediate-level design, it is perhaps better to distinguish this category by referring to them as *general design patterns*. The Gang of Four's work [Gamma+1995] is an example of a catalogue of general-purpose design patterns.

Context Sensitivity

- Solution structure is sensitive to details of purpose and context
  - Problem and solution feed forward and back
- Context-free design is meaningless
  - No universal or independent model of design
  - Context can challenge and invalidate assumptions



7

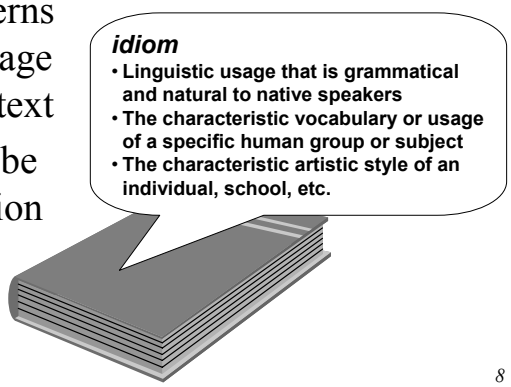
Design is not simply a feed forward process into which an analysis is fed, a handle turned, and a suitable implementation spat out. There are those who maintain such a view, but close inspection of what they define as *analysis* reveals *synthesis*: construction detail and compromises relevant only to the solution and not an understanding of the problem. The belief that *a* problem has *a* solution is also at the root of this misconception; this is not school, and there are typically many solutions to any given problem. A developer participates in a complex set of decisions, rather than being merely a cog in the works or a hands-off analyst.

On the compromise of design, David Pye is quoted in [Petroski1992]:

> *It follows that all designs for use are arbitrary. The designer or his client has to choose in what degree and where there shall be failure. Thus the shape of all design things is the product of arbitrary choice. If you vary the terms of your compromise—say, more speed, more heat, less safety, more discomfort, lower first cost—then you vary the shape of the thing designed. It is quite impossible for any design to be 'the logical outcome of the requirements' simply because, the requirements being in conflict, their logical outcome is an impossibility.*

Idioms

- An idiom is a practice specific to a language, language model or technology
  - Idioms can be patterns that have the language as part of their context
  - Idioms can simply be matters of convention



8

A number of classic 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 among languages. Others depend on features of a language model and are simply inapplicable when translated, such as between a strong and statically checked type system (e.g. C++ and Java) versus a looser, dynamically checked one (e.g. Smalltalk and Lisp).

Sometimes idioms need to be imported from one language to another, breaking language cultures out of 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 remember that this is anything but a generalisation and the forces must be considered carefully. For example, a great many C++ libraries suffered from inappropriate application of Smalltalk idioms, and the same can also be said of many Java systems with respect to C++.

Idioms relate strongly to style, and can be said to form a vernacular for a given context, e.g. [Vermeulen+2000] and [Warren+1999].

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

9

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.

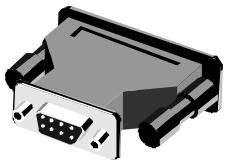
In constructing a system, catalogues of patterns such as [Gamma+1995] and [Buschmann+1996] can play a role, as can individual patterns. However, a pattern language [Alexander+1977] defines a more complete approach to connecting patterns together in a generative fashion for a particular task. They provide 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.

> *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.*
>
> [Coplien1996]

Some patterns, on closer inspection, appear to be composed of other patterns. 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]. They can be viewed as small pattern languages.

C++ Value-Based Programming

- Value semantics are in the core of C++
  - Deterministic destruction, conversions, operator overloading, templates, etc.
- A collection of principles and patterns can support writing user-defined value types
  - C++ has many mechanisms, not all of which are appropriate for values
  - *Substitutability* informs sensible use of these mechanisms



10

In the context of OO, transparency of use normally equates to the use of inheritance and polymorphism, and bound by the Liskov Substitution Principle [Liskov1987]:

> *A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a* subtype *is one whose objects provide all the behavior of objects of another type (the* supertype*) plus something extra. What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.*

A more approachable form of substitutability is the following [Strunk+1979]:

> *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.*

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, Czarnecki+2000]. The behavioural concepts behind LSP can be generalised to cover all forms of substitutability [Henney2000c]. In C++ substitutability goes beyond public inheritance, and includes overloading, conversions, generic programming, and mutability. These correspond closely to the forms of polymorphism that are available in the language [Cardelli+1985].

Value Object

- Values are strongly informational objects without a role for separate identity
  - Support copying (construction and assignment)
  - Uniform use suggests pass by copy or *const* reference, but not by pointer
- Substitutability typically based on conversions and overloading
  - Typically not in a class hierarchy

Values are objects for which identity is not significant, i.e. the focus is principally on their state and then their behaviour. Other distinguishing features of values include their granularity and content: They are typically fine grained rather than course, and their behaviour is closely structured around their state.

In C++, values are associated with a set of capabilities and conventions that are captured by the Value Object pattern. A string is an example of a Value Object: 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 substitutable in other ways. However, simply because a value can be copied, it does not mean that it must be copied – this is part of their 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.

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 *);
    ...
};
```

12

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

- A conversion can be defined to go out from a user-defined type to another type
  - Should be used with caution and good taste

```
class value_type
{
public:
    operator bool() const;
    ...
};
```

***If you want to include an implicit Boolean check for object state, consider* const void * *over* bool.**

```
class value_type
{
public:
    operator const void *() const;
    ...
};
```

13

An implicit conversion from another type into a type we are defining can be provided through a user-defined conversion operator (UDC). However, UDCs should be treated with some caution. For instance, although a `const char *` can be reasonably passed where a `string` is expected, the converse is not true:

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

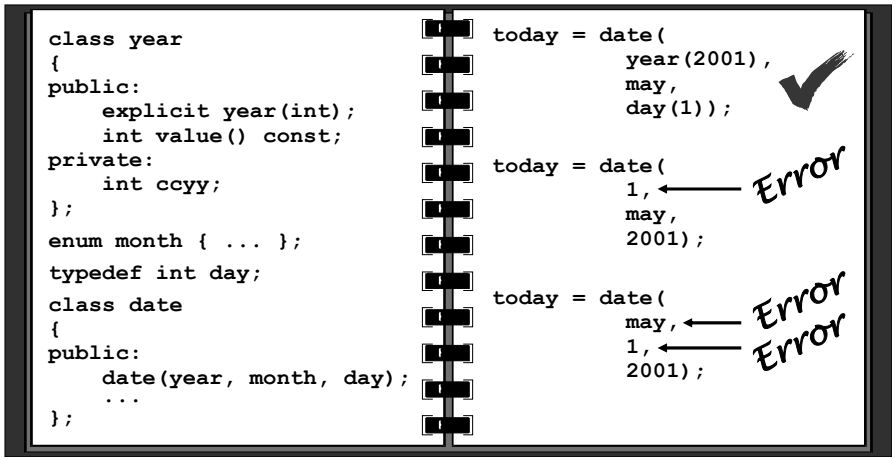
Because of the lifetime of temporary objects, the following would result in undefined behaviour:

```
string prefix, suffix;
...
const char *whole = prefix + suffix;
cout << whole << endl;
```

This is the reason that `std::string` does not support such a conversion.

There is one query, however, that may sometimes be conveniently expressed through a UDC: Is an object set? For many classes this immediately translates to `operator bool`. However, in many cases it turns out that `bool` is not the safest realisation of a Boolean type. It introduces a number of subtle conversion problems for many classes. These problems stem typically from `bool`'s underlying integer nature: Its eagerness to participate in all kinds of (surprising) arithmetic and comparison. In contrast to `bool`, a `const void *` is positively hermit-like in its interactions with other types and operators.

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 Object type for this purpose using templates [Barton+1994].

Custom Keyword Cast

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

```
const int c = lexical_cast<int>("299792458");
std::string s = lexical_cast<std::string>(c);
```

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

15

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, Henney1998, Henney2000d, 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);
}
```

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]**

16

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. It makes sense for user-defined value types to consider operator overloading because they share common semantics and ideas, which suggests common notation. This is the basis of generic programming.

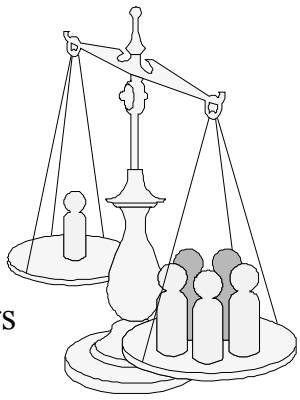
Where the implied semantics of built-ins cannot be met, other accepted idioms, such as streaming, provide a second port of call. Thus we can temper a hard line view of operator overloading with a little – but not too much – pragmatism:

> *"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



17

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. In essence, when you are defining value types, you are extending the core of the language: you are doing language design, and with that comes a certain set of responsibilities.

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.

Java Value-Based Programming

- Java's object model does not directly support value concepts
  - Referentially-transparent objects in a reference-based language objects
- However, this does not diminish the need to express values!
  - Fine-grained information still needs representation

Instance identity is normally seen as a fundamental of any object. However, there are many cases where the identity is incidental and it is the behaviour against the state of an object that is of interest, e.g. in any comparison it would be `equals` rather than == that would be used between comparands.

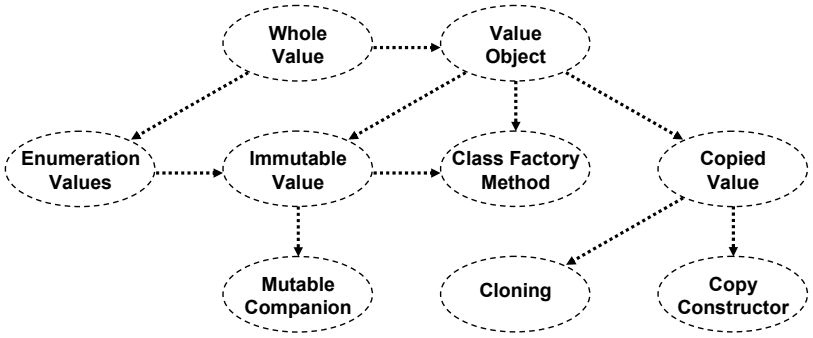
Where behaviour and state are significant, such objects are conventionally known as value objects. Language support for value versus reference based programming can make a difference to programming style as well as perception. Many programmers are subliminally aware of these differences, but making them explicit tends to lead to a cleaner and crisper approach to class design.

Examples of value types are strings, numbers, low level collections, dates, etc. In other words, types that are used to represent attributes in a class rather than associations, which are implicitly reference based. In UML, a composite object also represents a by-value concept; composition is sometimes also known as aggregation by value.

Java's only true values are the built-in types such as `int`. It is not, however, possible for developers to create their own types to have the same behaviour, i.e. pass by copy and operator overloading are not supported outside the core language. Passing by copy is supported only in the context of remoting, specifically `java.io.Serializable` types under RMI. Value-based programming in Java is a matter of adopting practices, and perhaps integration with a framework (e.g. [JValue]).

Patterns of Value

- A simple set of connected patterns can support value-based programming in Java



19

Although all objects in Java are reference based, this does not remove the need for developers to create types that act as values (as opposed to objects with significant identity and behaviour). Value-based programming in Java is supported by a variety of patterns [Henney2000a, Henney2000b]:

Class Factory Method: Provide a convenient way to create an object that encapsulates the details of creation.

Cloning: Provide a way to copy an object polymorphically through an interface without knowing its concrete type.

Copied Value: Describe how modifiable value objects should be passed around so as to avoid aliasing problems.

Copy Constructor: Provide a way to construct an object based on another instance with exactly the same type.

Enumeration Values: Represent a distinct and bounded set of values as objects.

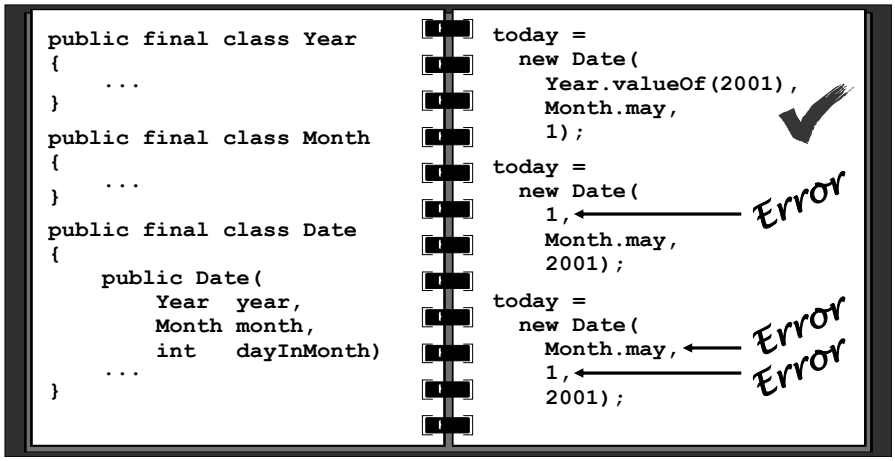
Immutable Value: Define a class for which instances have fixed state to avoid aliasing problems.

Mutable Companion: Define a companion class to simplify complex manipulation of Immutable Value objects.

Value Object: Define features for a class, whose instances represent values, so that it conforms to expectations.

Whole Value: Provide distinct classes to express distinct domain-specific value quantities clearly and safely.

Whole Value



How can you represent a primitive domain quantity in your system without loss of meaning? It is tempting, and indeed common, to represent value quantities in a program in the most fundamental units possible, e.g. integers and floating point numbers. However, this fails to communicate the understanding of the problem and its quantities into the solution and shows poor use of the checked type system. Something like an `int` is a pretty meaningless unit of currency: It does not define its units and it cannot be checked for sensible use. Compile-time errors should always be preferred to runtime errors – it can mean the difference between debugging before a release or after it.

The solution is to express the type of the quantity as a class: a Whole Value [Cunningham1995] (also known as Quantity [Fowler1997]). A Whole Value class can range from a simple wrapper class – making the intent of the code clearer to both the human reader and the compiler – to one with more complete behaviour. A Whole Value makes a method interface clearer and safer; whether it is used in the encapsulated implementation – as opposed to reverting to raw values – depends on other design choices and preferences.

Thus a simple `Year` class elevates years to a checkable and named quantity.

```
public final class Year
{
    public static Year valueOf(int year)
        { return new Year(year); }
    public Year(int year) { value = year; }
    public int getValue() { return value; }
    private final int value;
}
```

Value Object

- Need to emphasise content over identity
  - Need to consider creation and sharing issues
  - Override and provide methods based on comparison of content
    - E.g. *equals*, *hashCode*, *compareTo*
- Typically not in a class hierarchy but may implement interfaces

21

Having established that a value type needs implementing as a class, often because it is a Whole Value representing a domain specific quantity, what steps must be taken to define the class so that it meets user expectations?

The use of value objects revolves around their represented content rather than their state: Comparisons and ordering between value objects should depend on their content and not on their reference identity. This applies to their use as keys in associative collections as well as the more obvious comparisons for equality.

Override the methods in `Object` whose action should be related to content and not identity (e.g. `equals`) and, if appropriate, implement `Serializable`.

Field by field comparison is more appropriate for values than the default, identity based `Object.equals`. The following shows `equals` defined with minimum casting and clutter:

```
public final class Date ...
{
    ...
    public boolean equals(Object rhs)
    {
        return rhs instanceof Date && equals((Date) rhs);
    }
    public boolean equals(Date rhs)
    {
        return rhs != null && rhs.year == year &&
               rhs.month.equals(month) && rhs.day == day;
    }
    private int year, day;
    private Month month;
}
```

Enumeration Values

- Representing distinct fixed sets of values
  - Emulates enumeration constant feature found in other languages

```
public final class Month ...
{
    public static final Month january = new Month(1);
    ...
    public static final Month december = new Month(12);
    public int getValue() { return value; }
    ...
    private Month(int month) { value = month; }
    private final int value;
}
```

22

How can you represent a fixed set of constant values and preserve type safety? It is common that a fixed set of options or indicators need representation. Even if they have no specific values they must still be distinct from one another.

For example, assuming you count months from January, do you number from 0 or from 1? A good case can be made for either one, which is exactly the problem with raw, meaningless `int`s. That said, there is a question of consistency with any API that thinks it is reasonable to start months from `0` and days from `1`. Given that Java defaults `int` fields to `0` – potentially hiding initialisation problems – it seems best to opt for the `1` convention.

Now that the decision has been made, how can it be enforced? The simplest way to communicate this intent with the user is to use named `int` constants. Although this is preferable to inviting users to plug in magic numbers, it is still open to intentional or accidental abuse: `int` is checked only as an `int` and not as a month. Modelling month as a Whole Value offers a more expressive approach, but does not necessarily constrain the set of values correctly.

Treat each constant as a Whole Value instance, declaring it as `public static final` in the Whole Value class. To ensure that the set of constant values remains fixed, prevent public construction by defining the constructor as `private`. Implementing the class as an Immutable Value prevents the possibility of the constant values being unexpectedly non-constant.

Users of the `Month` class can now both name the month of their choice and receive the full support of the compiler's type checking. And, because the type implicitly restricts the Enumeration Values, range checking comes for free.

Class Factory Method

- A *static* method can be used in place of or as well as a constructor for creating values
  - Simplifies complex expressions
  - Offers scope to optimise value creation

```
public final class Month ...
{
    ...
    public static Month valueOf(int month)
        { return months[month - 1]; }
    private static final Month[] months =
        { january, ..., december };
    ...
}
```

23

How can you simplify, and potentially optimise, construction of Value Objects in expressions without resorting to intrusive new expressions? How do you provide for the opportunity to reuse existing objects that have the same value, i.e. Flyweight objects [Gamma+1995]?

Assuming that year and month are represented by Whole Value objects, the following code to initialise a Date looks syntactically clumsy and does not support the use of Enumeration Values for months:

```
Date clumsy =
    new Date(new Year(year), new Month(month), day);
```

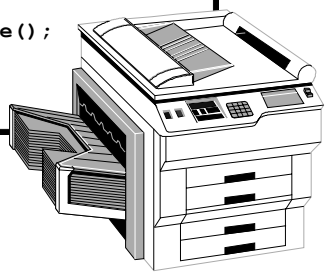
Provide `static` methods to be used instead of (or as well as) ordinary constructors. The methods return either newly created Value Objects or existing objects from a lookup table.

For value types with a clear and commonly used string representation, it is common to provide a Class Factory Method that takes a `String` and returns a value object based on the parsing of that string. For example, `Month` may provide a conversion from the month name to the relevant one of the Enumeration Values.

Even when it does not offer a lookup optimisation, a Class Factory Method has benefits in promoting uniform usage and syntactic simplification. It is a common variation of the vanilla Factory Method pattern [Gamma+1995].

Copied Value

```
class Order
{
    ...
    public void setDeliveryDate(Date newDeliveryDate)
    {
        deliveryDate = (Date) newDeliveryDate.clone();
    }
    public Date getDeliveryDate()
    {
        return (Date) deliveryDate.clone();
    }
    private Date deliveryDate;
}
```



For modifiable values, copying by cloning or copy construction avoids aliasing problems and preserves encapsulation.

24

How can you pass a modifiable Value Object into and out of methods without permitting callers or called methods to affect the original object? Value objects are often used as attributes inside other objects. Subtle aliasing problems can arise if the attribute is assigned directly from a method argument when it is set or if it is returned directly when queried:

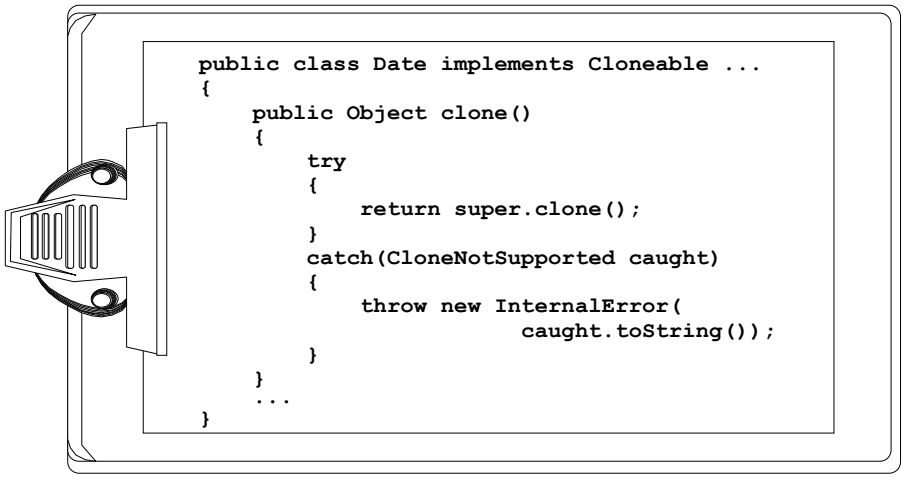
```
Order firstOrder, secondOrder;
...
Date date = new Date(year, month, day);
firstOrder.setDeliveryDate(date);
date.nextDay();
secondOrder.setDeliveryDate(date);
```

In essence, this is a violation of encapsulation: Although declared private, users can still interfere with the object's representation.

Implement the Value Object class to support either Cloning or a Copy Constructor, and make a copy of the original whenever it needs to be passed. This prevents sharing and preserves encapsulation, whilst still allowing value objects to be modified. Copied Value is also known as Cloneable Value given that this is the most common technique used.

For values that are queried far more often than they are modified, an Immutable Value with a Mutable Companion offers an alternative with less object creation.

Cloning



```
public class Date implements Cloneable ...
{
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch(CloneNotSupported caught)
        {
            throw new InternalError(
                        caught.toString());
        }
    }
    ...
}
```

25

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 `instanceof` and casts. 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 is standard within Java. It is based on the `Cloneable` marker interface.

The requirements for writing `Cloneable` can be seen as straightforward, but there is still scope for developers to be caught out by traps and pitfalls [Ball2000].

One limitation in Java's type system that makes the mechanism more awkward to use than is necessary is the absence of covariant return types: Although perfectly safe, method return types cannot be specialised when the method is overridden, e.g. so that `clone` returns `Date` rather than `Object`. This means that casting the result is always forced on to the `clone` user.

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. Note that Cloning should not be confused with the Prototype pattern: The Prototype pattern is built on Cloning but is not itself the same. The difference becomes clear from Prototype's intent:

> *Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.*

Copy Constructor

```
public final class Date ...
{
    public Date(Date that)
    {
        year  = that.year;
        month = that.month;
        day   = that.day;
    }
    ...
    private int year, month, day;
}
```

Copying with a Copy Constructor is based on compile time type rather than runtime type, and so makes sense for objects not in a class hierarchy

Where Cloning is based on underlying runtime type, allowing objects to be accurately copied within a class hierarchy, an idiom borrowed from C++ allows objects not in a class hierarchy to be copied more efficiently. Copy Constructor is based on compile time type, which makes sense for classes that are `final` and subclass only `Object`.

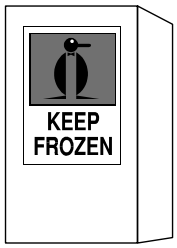
The class provides a constructor that takes another object of the same type as its argument. Because encapsulation is at the class rather than the object level, the fields are accessible for copying; there is no mechanism in Java that automatically copies the fields from one object to another, so copying appropriate for each field must be performed manually. On the other hand, there are no exception issues (such as quashing `CloneNotSupported`) or runtime type checks, and so the copy is efficient.

The suggested constraint on the class being `final` is to avoid the possibility of slicing: Where an instance of a subclass to be copied, any extra fields that it had would be ignored in the copy. In a few cases this may be what the developer wants, but more commonly it is not what they expect.

The standard `String` class is an example of a class that uses Copy Constructor but not Cloning. It is also a true and deep copy: The new `String` does not share its representation with the original.

Immutable Value

- In reference-based languages value objects with modifiable state...
  - Must be manually copied to emulate pass by value and avoid aliasing side effects
  - Must ensure they are synchronised if shared between threads
- Therefore...
  - Don't have modifiable state!



27

How can you share Value Objects and guarantee no side effect problems? Copied Value describes the conventions for using a modifiable value object to minimise aliasing issues. However, this can be error prone and may lead to excessive creation of small objects, especially where values are frequently queried or passed around.

If objects are shared between threads they must be thread safe: Synchronisation of state change incurs an overhead, but is essential in guarding against race conditions. Even where cloning or synchronisation are carefully attended to, there are still opportunities for undefined behaviour: The integrity of an associative collection may be compromised if the state of an object used as a key is modified via an aliasing reference.

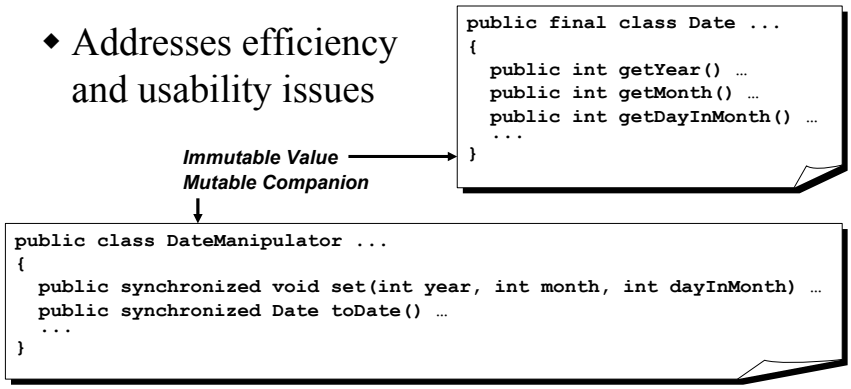
The issues can be resolved by setting the internal state of the Value Object at construction, and allowing no subsequent modifications. Declaring the fields `final` ensures this *no change* promise is honoured. This guarantee implies also that either the class itself must be `final` or its subclasses must also be Immutable Values.

The absence of any possible state changes means there no reason to synchronise. Not only does this make Immutable Value objects implicitly thread safe; the absence of locking means that their use in threaded environment is also efficient.

There are complementary techniques for creating Immutable Value objects: provide a complete and intuitive set of constructors; provide a number of Class Factory Methods; provide a Mutable Companion.

Mutable Companion

- A Mutable Companion is a factory object for Immutable Values
  - Addresses efficiency and usability issues



How can you simplify complex construction of an Immutable Value? Constructors provide a way of creating objects from a fixed set of arguments, but they cannot accumulate changes or handle complex expressions without becoming too complex, e.g. calculate the date 15 working days from today. Such requirements typically lead to expressions that create many temporary objects. Class Factory Methods may be able to optimise some of the creation and expressiveness issues, but they are still one-off creation requests.

Implement a companion class that supports modifier methods, and acts as a factory for Immutable Value objects. For convenience the factory can stand not only as a separate class, but can also take on some of the functional roles and capabilities of the Immutable Value, e.g. `StringBuffer` and `String`. The modifiers, which should be `synchronized`, allow for cumulative or complex state changes, and a query method allows users to get access to the resulting Immutable Value.

Although it may support many of the same methods as its associated Immutable Value, it is not related to it by inheritance: It cannot fulfil the same guarantees on immutability and therefore cannot be considered a subtype, hence its non-familial status as a companion.

Summary

- Objects do not live in a free society
  - They are created to serve a purpose
  - They are not all created equal
  - They shouldn't aspire to being equal
- However, respect them and their differences
  - Some objects offer valuable service, whereas others just have value

[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.

[Ball2000] Steve Ball, "Effective Java: Effective Cloning", *Java Report* 5(1), January 2000.

[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.

[Booch1994] Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin/Cummings, 1994.

[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.

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

[Coplien1996] James O Coplien, *Software Patterns*, SIGS, 1996, available from `http://www.bell-labs.com/user/cope/Patterns/WhitePaper/`.

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

[Cunningham1995] Ward Cunningham, "The CHECKS Pattern Language of Information Integrity", *Pattern Languages of Program Design*, edited by James O Coplien and Douglas C Schmidt, Addison-Wesley, 1995.

[Czarnecki+2000] Krzysztof Czarnecki and Ulrich W Eisenecker, *Generative Programming*, Addison-Wesley, 2000.

[EXE1990] "Around the Table with Bjarne Stroustrup", *.EXE*, 5(6), November 1990.
[Fowler1997] Martin Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.
[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[Henney1998] Kevlin Henney, "Keeping up with runtime type information", *EXE*, October 1998, also available at `http://www.curbralan.com`.
[Henney2000a] Kevlin Henney, "Patterns in Java: Patterns of Value", *Java Report* 5(2), February 2000, also available at `http://www.curbralan.com`.
[Henney2000b] Kevlin Henney, "Patterns in Java: Value Added", *Java Report* 5(4), April 2000, also available at `http://www.curbralan.com`.
[Henney2000c] Kevlin Henney, "From Mechanism to Method: Substitutability", *C++ Report* 12(5), May 2000, also available at `http://www.curbralan.com`.
[Henney2000d] Kevlin Henney, "From Mechanism to Method: Valued Conversions", *C++ Report* 12(7), July/August 2000, also available at `http://www.curbralan.com`.
[ISO1998] *International Standard: Programming Language - C++*, ISO/IEC 14882:1998(E), 1998.
[Liskov1987] Barbara Liskov, "Data Abstraction and Hierarchy", *OOPSLA '87 Addendum to the Proceedings*, October 1987.
[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.
[Petroski1992] Henry Petroski, *To Engineer is Human: The Role of Failure in Successful Design*, Vintage, 1992.
[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.
[Taligent1994] *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*, Addison-Wesley, 1994.
[Vermeulen+2000] Allan Vermeulen, Scott W Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jum Shur and Patrick Thompson, *The Elements of Java Style*, Cambridge University Press, 2000.
[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.
[Warren+1999] Nigel Warren and Philip Bishop, *Java in Practice: Design Styles and Idioms for Effective Java*, Addison-Wesley, 1999.