Idioms

Breaking the Language Barrier

Kevlin Henney

QA Training

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

Presented at the ACCU's *C and C++ European Developers Forum*, the Oxford Union, Oxford, UK, 12th September 1998.

Kevlin Henney

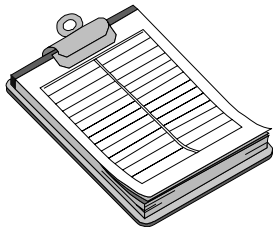
`khenney@qatraining.com`
`kevlin@acm.org`

QA Training

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

Agenda

- Objectives
  - Understand the role that idioms play when programming in a particular language
- Contents
  - Linguistics
  - Information hiding
  - Value based programming
  - Callbacks
  - Iteration



2

When comparing programming languages, and particularly when moving to a different language, we often look at the similarities and differences. It is tempting to concentrate on the similarities and program around the differences.

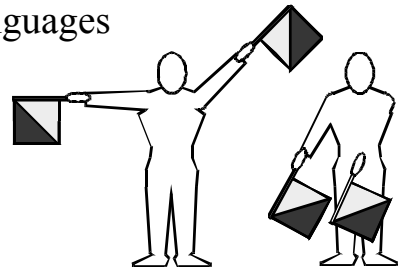
Superficial similarities can lead to a false sense of familiarity that accelerates the early part of the learning curve, but can put the brakes on further skills development. Programmers bringing the mindset of their previous language with them often fail to take advantage of the solutions available to them and, in many cases, consequently create code that is expressed inappropriately.

Patterns, as popularised by the Gang of Four's work on design patterns [Gamma, Helm, Johnson and Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley], are seen as a way of capturing and communicating successful solutions to problems. There are many kinds of pattern, including analysis and architectural, and at the level of the programming language such patterns are often known as idioms. Although languages in the C family – such as C, C++, Java, and IDL – have superficial similarities they are nonetheless quite separate languages supporting their own styles and idioms.

Understanding that a programming language provides a context – an essential feature of any pattern – determines the suitability of a particular idiom and mindset. This talk looks at how a language affects programming style, highlighting issues through some examples and counterexamples.

Linguistics

- Intent
  - Use of language is affected by experience and expectation as well as engineering
- Contents
  - Borrowing between languages
  - The effect of syntax
  - Language and technology feature sets



3

What defines a language? The market view is that syntax and tools define a language, and proficiency in a language is simply a matter of acquiring the right tool and learning the right features. Sometimes the view is taken that programming language is superfluous and is all that stands between the software development industry and timely successful projects.

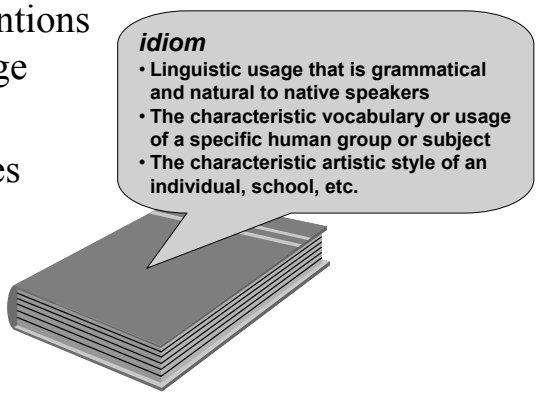
The demise of the programming language has been fashionably predicted every decade since programming rose out of the primeval first generation machine code swamp into the mnemonic second generation of assembly language. However, programming languages are as popular now as they ever have been, and just as rich a source of disagreement amongst developers – "language of choice", like editor of choice, still ranks high as something of a "religious" issue.

The deliverance from coding promised by 4GL, visual programming environments and code generators has failed to materialise: all the effort that is saved in developing another part of a system is either pushed to another part or raises the expectation of functionality, which in turn raises expectations.

In truth, use of language is a complex skill – programming – rather than a plug and play asset on a CV: mastering a language requires more than knowledge of which buttons to press. As a means of expression, language therefore engenders a mindset and a culture.

Idioms

- Idioms are language, language model or technology specific patterns
  - Common conventions of style and usage
  - Dependency on language features



4

A pattern may be defined as a generalised solution to a problem in context. More usefully – in terms of a definition – patterns capture and document common design concepts and vocabulary as well as, in recent years, the imagination of many developers.

Patterns originated in architecture and the design of human centred environments, in the work of Christopher Alexander [*The Timeless Way of Building*, Oxford University Press], where emphasis is firmly on practice and prior art rather than invention. Christopher Alexander defines the essential content of any pattern:

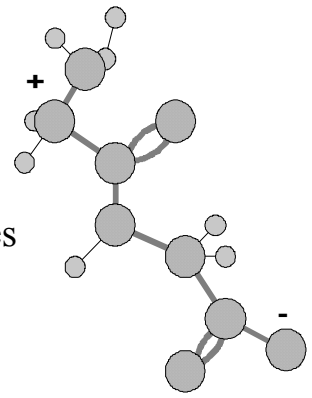
> *We know that every pattern is an instruction of the general form:*
> *context ⇒ conflicting forces ⇒ configuration*

Within software development they are most often discussed in the context of design, e.g. the Gang of Four [*Design Patterns*], but have also been applied to many stages of the software development lifecycle (e.g. analysis patterns [Martin Fowler, *Analysis Patterns*, Addison-Wesley]) and many scales (e.g. architecture and language [Buschmann et al, *A System of Patterns: Pattern-Oriented Software Architecture*, Wiley]).

Retrospectively idioms can be recast as low level patterns, being indigenous to a particular language, language paradigm (e.g. procedural as opposed to functional) or technology (e.g. CORBA). Idioms and data/algorithm concerns are at the micro-architecture level of a system, whereas design patterns bridge micro and macro-architecture. Thus idioms are more specific than design patterns but are nonetheless part of the same continuum.

Transgenics

- In another language a given idiom may be...
  - In place and relevant
  - Innovative and relevant
  - Irrelevant
  - Inexpressible
  - Inverted with respect to roles and structure



5

James Coplien catalogues a number of idioms for C++ in *Advanced C++ Programming Styles and Idioms* [Addison-Wesley]. Kent Beck documents a great many Smalltalk patterns [*Smalltalk Best Practice Patterns*, Prentice Hall]. 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 [James Coplien, *Software Patterns*, SIGS].

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 [*Software Patterns*] 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.

Syntax

- Familiarity breeds comfort
  - An accelerated initial learning curve may often followed by an early closing of mind
- Similarity between languages can be superficial and misleading
  - C++ and C
  - Java and C++
  - IDL and C, C++ & Java

6

Language syntax is perhaps one of the first things that developers latch onto when presented with a language. Views differ as to the significance of syntax and low level semantics: from not being an issue to being one of the most important features. As ever, the truth lies somewhere between these two.

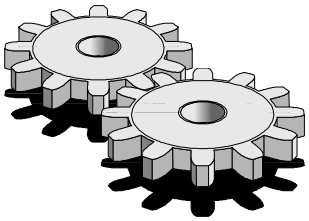
Idealistically one might expect syntax to be irrelevant; practically it is the very foundation of what a developer must know to be proficient. The ability to generalise between syntax models varies between programmers and, if not accounted for, can prove to be an occasional and surprising obstacle. For instance, long time COBOL or VB programmers moving to C, C++ or Java (the "curly bracket family") sometimes find the syntax obscure, and so rather than learning new high level concepts they find themselves struggling at the lower end of the language feature chain. There is almost a sense that they are not "really programming"; their experience defines what looks like programming to them, and there is an initial mismatch. This is more commonly felt by developers with a narrow experience of languages.

On the other hand, the similarity between languages can be deceptive. It can encourage programmers to think in terms of a common subset of mindset, and they find themselves programming "into" a language rather than in it. Their initial learning of a language may seem more rapid, but they may reach a plateau of complacency where they have seen how all the familiar features map across, as well as a couple of new ones. What is often the challenge is the "unlearning" of habits.

Mechanism

- Object lifetime management
- Exceptions
- Type checking
- Polymorphism
- Genericity
- Execution architecture

What features influence a design?



7

The degree to which a language or technology specifically supports certain mechanisms will have an impact not only on the way that programmers will think about a problem, but also on the way that an application should be designed. The realisation that there is a two way flow between architecture and implementation is in many ways not surprising, but is at odds with the purist school of thought which maintains that a system may be fully designed in the abstract independently of its deployment technology and engineering model.

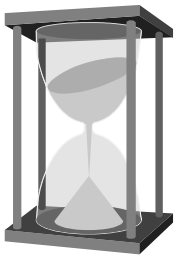
The mechanisms through which a programmer will express a system have an impact on both learning a language and the way in which problems will be solved. Syntax may present newcomers to C and C++ with an initial challenge, but it is more likely to be the underlying memory model of these languages – explicit management of object lifetime and the use of pointers – that present the challenges of substance.

Here are a few features that are likely to have an impact:

- Object lifetime management: programmer versus system managed.
- Exceptions: exception handling mechanisms can have a significant effect on ordinary control flow.
- Type checking: weak or strong, static or dynamic.
- Polymorphism: static versus dynamic binding, substitutability.
- Genericity: type safety, efficiency, expressiveness.
- Execution architecture: concurrent, distributed, event driven, call/return, asynchronous.

Time

- Developers acquire experience
  - Affects use of language and technology
- Languages and technologies evolve
  - Standards provide islands of stability
    - So many to choose from!
  - Growing languages and progressive standards create generation gaps
    - Internet years to wall clock years



8

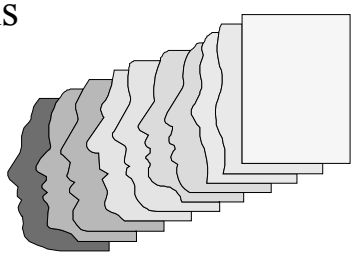
It is only natural that developer's acquire more experience as time goes by. However, it is how that wisdom is applied that is the important factor. Developers may stay with a single language and technology for a long time, or they may find themselves moving across different platforms with time, or even using many simultaneously. Understanding that differences can make the difference is important; too often it is the case that old lessons learnt are abandoned simply because this is not appreciated – consider the evolution of modern commercial operating systems.

A developer's ability to model a system is often biased by experience. This is typically a good thing, and covers the languages, APIs and problem domains that developers are involved in. Libraries and frameworks have a particular architecture that will influence those that "live" within them as an additional force.

Standardisation is always seen as a mixed blessing. Such standards may be *de jure* (e.g. ISO standard C++) or *de facto* (ARM C++), but they are often the solution to the Tower of Babel of system portability. The forces working on the establishment and uptake of any standard are timeliness (neither before its time, nor too late) and relevance (subsets of proprietary systems are rarely of any use).

Pluralism

- There are no final solutions, so...
  - Don't look for one
  - Don't believe anyone who tries to sell you one
- Developers work with and are influenced by mixed technology systems
  - Component architectures
  - Libraries and frameworks
  - Languages



9

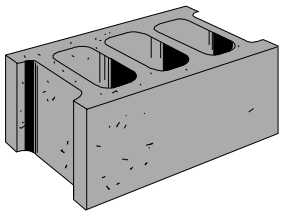
Occasionally the idea that a single language can satisfy all programming needs and all programmers has emerged, resulting in languages such as PL/1, Algol 68 and Ada. This final solution approach has not prevailed. In the 1970s there was something of a Cambrian explosion in languages that only a few languages survived. It was expected that this trend would continue and the market would cluster around only a few mainstream languages.

In the late 1990s we can see that this static equilibrium is unlikely to be achieved, and that the market is a more steady state dynamic equilibrium: the requirement older languages such as Fortran and COBOL sit cheek by jowl with the more modern C++ and Java in the skills market; there is an explosion of "little languages" in the form of script languages, inspired by the rise of the web and, in some part, growing out of the Unix scripting culture; proprietary and domain specific languages and environments are often on an equal, if not a more equal footing, than general purpose, open programming languages.

The emphasis now is not on the use of single languages for monolithic projects, but on the development integration of components. Developers working in many languages on a single project face a challenge in balancing both the need for design consistency and the need for appropriate local use of language, as well as having diverse knowledge of different APIs. This is especially true when bridging through declarative middleware languages such as IDL (Interface Definition Language). Although derived from C++, and therefore similar also to C and Java, OMG's IDL has its own idioms and has a style and intent different to the implementation and client languages that may be used on either side of it. Indeed, a C++ style can be detrimental to IDL design.

Information hiding

- Intent
  - Isolation of client code from representation decisions
- Contents
  - Opaque type
  - Fragile base class problem
  - Interfaces
  - Support for interfaces



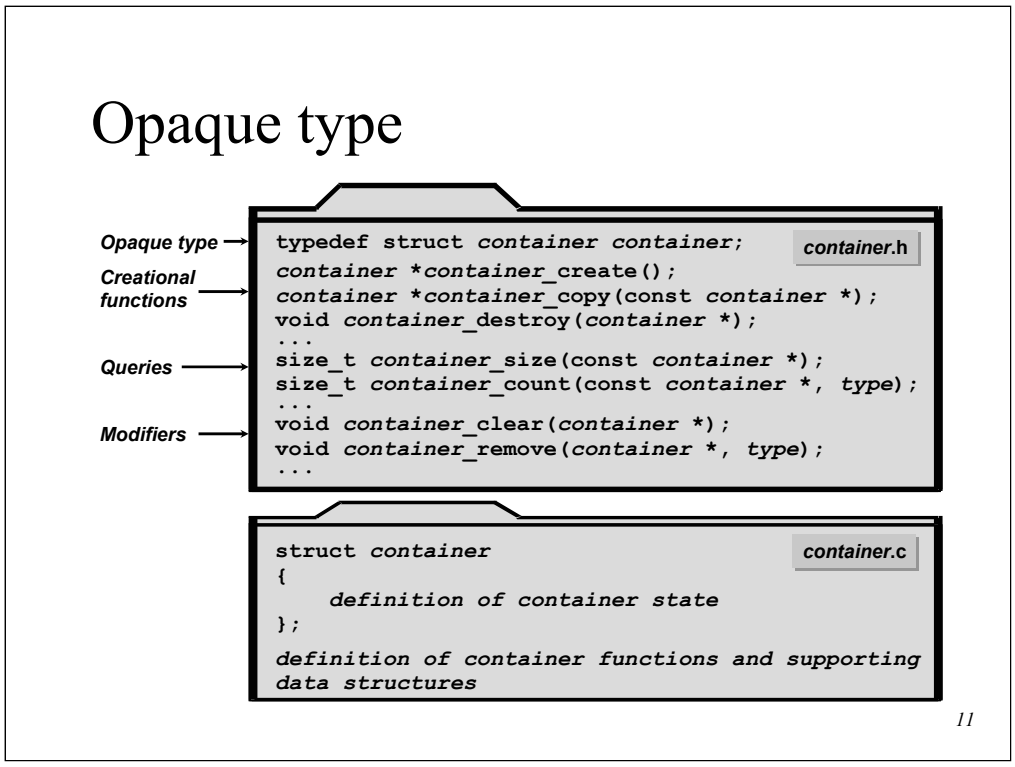
10

Information hiding is an architectural principle that seeks to separate the implementation of programming concept from its interface. Itis one of the oldest and most respected software engineering principles, stemming principally from the work of Parnas in the early 70s. Information hiding is based on the partitioning of software into components, each of which has two parts: an interface that other components may use; and an implementation that provides the representation and functionality specified by the interface.

In the physical structure of a program information hiding can be applied through the use of header and source files, or more formally in languages that support the necessary constructs: Java's package, Ada's package, and the Modula family's module.

Information hiding can also be applied at the level of the data type, where design decisions about representation can be hidden. An abstract data type (ADT) is a data type defined by its behaviour and operations on it rather than by its representation: hence abstract rather than concrete. This form of information hiding is one of the cornerstones of object-orientation, where encapsulation defines both the property of associating function and data in the same unit (literally "within a capsule"), and the protection of representation from clients.

Information hiding is a security principle rather than some arbitrary constraint of a programmer's freedom. It offers a firewall around an abstraction that protects it from accidental and malicious misuse. It also reduces dependencies on potentially volatile design decisions. Importantly it is also a simplifier: to use a well designed abstraction all that is required is knowledge of the interface and its associated behaviour, and not the details of implementation.



It is often said – incorrectly – that C does not support encapsulation of data types. C certainly supports many forms of abstraction, one of which is the ability to implement ADTs. Whilst C does not have the same rigorous support for this style offered by OO languages, the clean separation of actual data type representation from the declaration of the data type and the operations on it is very much within its reach. On the other hand it would be fair to say that a clean implementation of inheritance and polymorphism is not!

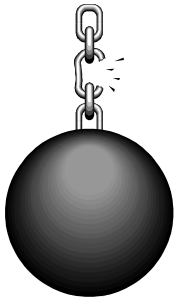
Opaque types, as found in languages like Modula 2, can be supported in C by forward declaring a `struct` with only its name. This declaration is placed in a header file along with prototypes for the functions that implement the operations of the ADT. A client then simply includes the header and uses pointers to the opaque type without depending on the actual implementation.

The implementation of the type is exposed in a source file that corresponds to the header file along with the function definitions. The translation unit acts as the basic unit of support for information hiding, so that any support data and functions are declared `static` within to prevent access from other units.

Part of every pattern is a discussion of the consequences of using it. The most obvious one is that an extra level of indirection has been introduced: only pointers to an incomplete type may be defined, probably leading to an increased use of dynamic memory. Also there can be no inline optimisations.

Opaque Type also has application in languages supporting OO concepts: in C++ it manifests itself as the Cheshire Cat idiom [Robert Murray, *C++ Strategies and Tactics*, Addison-Wesley].

Fragile base class problem



- Derived classes have a strong dependency on their base classes
  - Not as fully encapsulated or decoupled as delegation based relationships
- Base class evolution is a problem...
  - Release to release binary compatibility
  - Private changes lead to public builds
  - Derived class semantics when base is modified

There is a strong structural dependency upward from derived to base classes: a base class cannot be defined without full reference to its base(s). Modifications to a base class can therefore lead to something of a ripple effect especially if implementation is being inherited. This is known as the *fragile base class problem*.

Szyperski [*Component Software: Beyond Object-Oriented Programming*, Addison-Wesley] classifies two different aspects of the problem: the syntactic fragile base class problem and the semantic fragile base class problem.

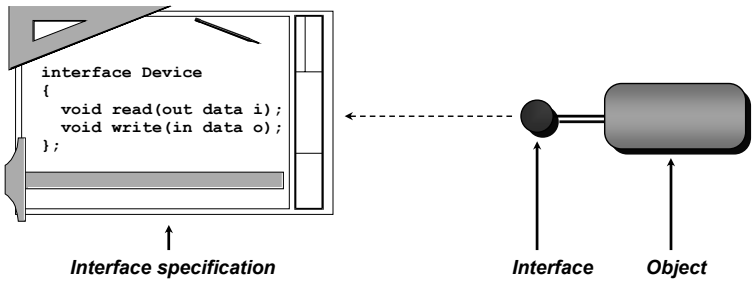
The syntactic problem is concerned with both source code and binary compatibility effects [Carroll and Ellis, *Design and Coding Reusable C++*, Addison-Wesley] of modifying a base class.

For instance, the use of `protected` is often questioned, especially for data. This creates even stronger dependencies between derived and base classes. In systems that have wide distribution, such as frameworks, this is almost equivalent to making the data `public`! The use of `protected` should be considered carefully, with ordinary instance data considered the least acceptable feature for this visibility. Java programmers should also be aware that `protected` also offers package visibility and not just subclass visibility of a feature. C++ programmers should note that `private virtual` functions can be overridden and may therefore create a dependency.

What arises from this analysis is an understanding that inheritance and polymorphism should be used principally for defining and developing against interfaces, with an emphasis on establishing stability in the interface's features and its meaning. Reuse of implementation is better left to delegation techniques.

Interfaces

- Fully decouple usage from creation type
  - Cleaner expression of design
  - Specification of contract



13

The type of an object describes the object's behaviour in terms of operations that can be performed on it. In effect it specifies the contract that the object will fulfil, where a contract is a promise of functionality offered by the supplier or servant when the object is used by the client. They describe the services and capabilities of an object and can be used generally to describe any object, not just coarse grain distributed objects, i.e. they can be used to describe the behaviour of ordinary data objects.

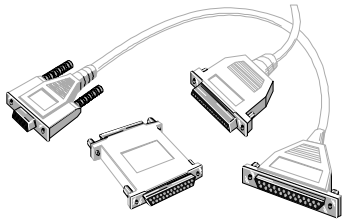
Interfaces only publish the public view of an object; they do not contain private implementation details. This means that they can be – given the appropriate middleware – language independent, which is an essential requirement in a heterogeneous distributed system.

The essential manifesto of information hiding might be stated in terms of: "Is it right that a user should depend on your representation decisions?". Given a "no" answer, we can see that interfaces present a significant inheritance based separation in a system: they fully separate *usage type* from *creation type* [Barton and Nackman, *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, Addison-Wesley].

The signature level of an interface can be captured in many languages, but the full semantics are less clear. Pre and postconditions offer one way of reasoning about abstract behaviour [Meyer, *Object-Oriented Software Construction*, Prentice Hall] but there are limitations when describing callback architectures [*Component Software*].

Support for interfaces

- Directly supported in some languages
  - Feature of Java
  - Principle structuring mechanism of IDLs
- May be supported idiomatically elsewhere
  - Pure abstract base classes in C++
  - Aggregates of function pointers in C



Support for interfaces varies across languages and systems:

- They are a standard part of Java. They should be seen as a tool in their own right rather than as a pauper's abstract class, as is often the case. This contrasts a proactive versus reactive use of interfaces.
- Interface Definition Languages (IDLs), such as the OMG's IDL and Microsoft's extended DCE IDL (MIDL), are purely declarative languages based on interfaces as their principal structuring mechanism. The implementation/interface separation is more complete in these systems as platform details are also encapsulated.
- Pure abstract base classes – classes with no implementation details (except perhaps a destructor) and only pure `virtual` functions – can express interfaces in C++. These can be used to reduce build dependencies. An implementation class may support multiple interfaces by inheriting from multiple interface classes. Querying interfaces is achieved by cross casting using the `dyanmic_cast` operator.
- In C the options for interface style programming are more limited, but they certainly exist. A coarse grain approach, at the level of deployed library, is to partition a system into dynamically linked libraries and program to DLL entry points. A more fine grained approach is to use aggregates (`structs` or arrays) of function pointers to emulate polymorphic lookup, i.e. like a C++ *vtable*.

Value based programming

- Intent
  - Model referentially transparent attribute types
- Contents
  - Value versus reference based
  - Whole value
  - Immutable value
  - The effect of distribution

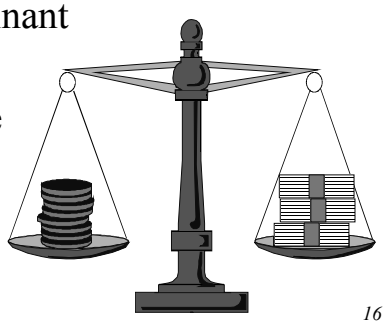
Language support for value versus reference based programming can make a difference to programming style as well as perception. Many programmers are subliminally aware of these differences, but making them explicit tends to lead to a cleaner and crisper approach to class design.

The requirements on the classes for such objects are often quite different to those used in associations. For instance, in C++ it is likely that a value based object will support copying operations (copy constructor and assignment operator), whereas this tends to be meaningless for many associated objects, hence copying operations would be disabled.

Features that mark out values are often their granularity and content: they are typically fine grained rather than coarse, and their behaviour is closely structured around their state.

Value versus reference based

- Objects are often characterised by...
  - Identity, behaviour, and state
- For values based objects...
  - Interpreted content is dominant characteristic
  - Behaviour in terms of state
  - Identity is least important



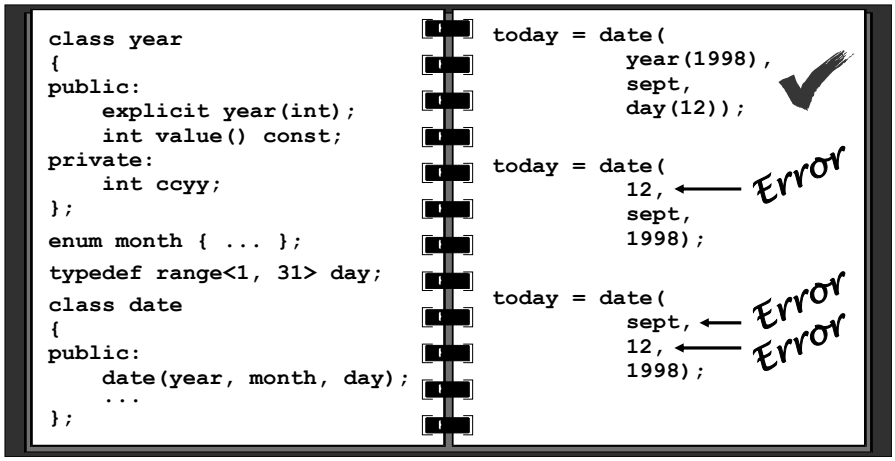
16

Grady Booch characterises objects by identity, state and behaviour [*Object-Oriented Analysis and Design with Applications*, Benjamin Cummins]. This is the most general model, but for many objects one or other of these features is either not important or will dominate the others. For instance, with stateless objects that are service oriented identity is not an important distinguishing characteristic, and neither is state, but behaviour is the key to their design. In such cases optimisations and simplifications are possible and desirable.

Value based programming focuses on those objects for which simple behaviour based directly on state is more important than identity, i.e. the ability to distinguish one object from another by its identity is not as important as the role played by its contents. Behaviour for values simplifies use of content, but is not an end in itself, e.g. one would not expect a value to be the subject of complex event notification mechanisms.

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*.

Whole value



17

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 Ward Cunningham's Whole Value pattern, from the CHECKS pattern language [Coplien and Schmidt (editors), *Pattern Languages of Program Design*, Addison-Wesley].

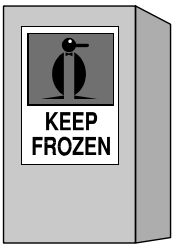
In this distinct types are used to correspond to domain value types. This affords greater annotation in the code and improved checking by the compiler, as illustrated in the example above. This is similar to dimensional analysis in the physical sciences, and a variation of Whole Value can be generalised for this purpose using templates [*Scientific and Engineering C++*].

Quantity types are value types that support copying and are unlikely to be polymorphic. In C++ this means that they will support a copy constructor and assignment operator (explicitly or generated by the compiler), and will contain no `virtual` functions.

It is not possible to define a literal form for a new type, but the constructor expression syntax comes close, e.g. `year(1998)`. 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. 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.

Immutable value

- In reference based languages value objects with modifiable state...
  - Must be cloned 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!



Context

- Small objects with simple construction for which distinct object identity is not important but value based content is.
- Reference based object semantics, possibly in a concurrent system.

Forces

- Attributes are instance variables that represent simple values rather than associations for an object, e.g. date of birth is an attribute of a person whereas their employer is an association. To avoid changes through other references when passing attributes around they must be copied, which may incur an inappropriate overhead.
- 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 required use of `clone` to copy objects that should not be aliased is potentially error prone, i.e. it is easy to forget, and it cannot be enforced automatically.
- Synchronisation of state change incurs an overhead, but is essential in guarding against race conditions and ensuring thread safety.
- It must be simple to use different values.

Solution

- Make an object's state immutable, i.e. freeze it at construction.
- Provide an intuitive and complete set of constructors whose construction is lightweight.

Rationale

- By definition the state of an immutable object cannot be changed, and hence will not suffer from undesirable or unpredictable change.

Resulting Context

- No need for synchronisation and no race condition problems.
- Sharing without aliasing side effects and therefore no need for copying.
- Change of value is effected by replacement by another object with a different value.
- The class must be `final` or any given subclass must also be an Immutable Value.
- Much more dynamic memory allocation where values are changed often. If this incurs an undesirable overhead, Flyweight [*Design Patterns*] can be applied as an optimisation.

The effect of distribution

- Values must be copied and not shared
- Passing by reference...
  - Implement java.rmi.Remote in Java
  - Implicit for ordinary OMG IDL interfaces
- Passing by value...
  - Implement java.io.Serializable in Java
  - Deconstruct to primitive attribute values
  - CORBA *objects by value* proposal

Values are not intended to be shared across address spaces: this would incur communication and context switch costs, as well as reliability liability. Given that objects contain behaviour as well as state, what does it mean to pass by value, and how can it be achieved?

The simplest solution is to define a homogeneous distributed system, which means that internal representation will have the same meaning everywhere. If a value is copied to an environment that does not have the code corresponding to the actual type of the transmitted value, then it can be downloaded from the same host. This is the basis of pass by value in Java's RMI. The JVM (Java Virtual Machine) defines the homogeneous architecture, serialisation provides the state, and class loading supplies the behaviour.

For traditional CORBA interfaces define remote objects interfaces in a heterogeneous environment. Either the Externalization Service should be used to send objects by value, or their logical attributes should be decomposed and sent as a `struct`. Alternatively, stepping out of the confines of CORBA for a moment, object state can be sent as a self describing XML stream – note that this requires a basic framework on both sides that is not implied by the declared interfaces. More recently, the OMG has been establishing mechanism and syntax for passing objects by value.

Callbacks

- Intent
  - Decouple the selection of functionality from its execution
- Contents
  - Function pointers
  - Command
  - Function objects
  - Block

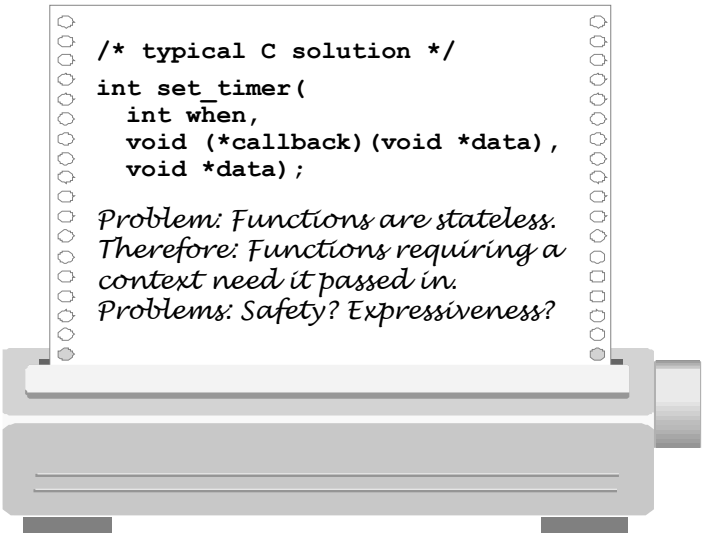
How can the selection of functionality be decoupled from its execution, either for parameterising behaviour or for handling an event from a framework?

An event is something that happens at a point in time, and is of interest to one or more other objects. Notification of an event also includes data associated with the event, either generated by the event or registered for use by an event handler. On the occurrence of an event an action somehow associated with the event must be executed.

Static function definitions and compile time determined calls are certainly easy to understand and analyse. However, sometimes the downside of "simplicity" is "simplistic". Such a model cannot adequately express the flexible behaviour required of many systems.

The handling of functions as data for later callback is the key to event driven systems and for decoupling layers of a component design. Typically languages and specific technologies provide their own set of features to support appropriate callback idioms; some languages do not support this concept at all or support a very limited model, e.g. 4GLs and scripting languages.

Function pointers



21

Event driven programming, whether on a GUI messaged based system or down at the level of interrupts, is characterised in C by the use of function pointers. Function pointers can be seen as a form of handcrafted polymorphism.

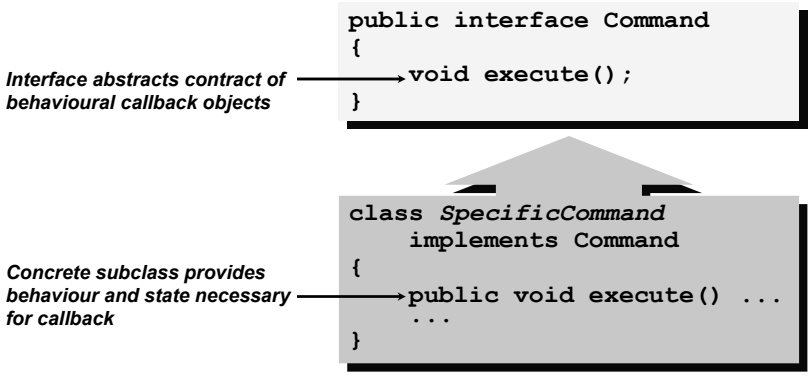
A review of the techniques used in C reveals that the representation of callback concepts is done through clumsy interfaces: for data to be associated with an event it is often passed in as a parameter of generic form, i.e. `void *` or worse (e.g. an integer type); as functions are stateless, any context data must be associated with the callback at registration only to be passed back to the function with the event; the definition of functions is fixed at compile time — although selection may be performed dynamically, definition cannot.

Multiple roots of control, association of data with function, grouping of related functions, generic interfaces with alternative implementations, etc. are mapped more easily onto an object model than a procedural one. Objects allow functionality to be customised and composed dynamically at runtime. As well as supporting greater extensibility it remains type safe. The dispatch mechanism is implicit in the object's type, as is any context data for the operation.

In C there is no adequate alternative to the basic callback model, and we are left with casts, clumsy usage, and fingers crossed hoping for the best. At the other extreme, Java has no concept of function pointers and so all callbacks concepts are handled within an object framework. C++ perhaps offers the best and worst of all possible worlds: it has C's function pointers; it also has member function pointers which generalise the function pointer model; and it supports conventional object-oriented solutions.

Command

- Extensible interface based approach
  - Callback details in implementation class



22

The intent of the Command pattern [*Design Patterns*] explicitly objectifies the concept of a function as object:

> *Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.*

The Command pattern is a general purpose design pattern found in object systems, independent of language. However, Java's direct support for interfaces makes it an easy language to express this concept in. The genericity and type safety issues are directly addressed by separation of concepts through inheritance.

A command undo system, as found in many GUI applications such as editors, can be implemented using this pattern:

```
public interface UndoableCommand extends Command
{
    void undo();
}
```

Given a `Stack` object referred to by history and an `UndoableCommand` referred to by `cmd`, the following would perform actions:

```
cmd.execute();
history.push(cmd);
```

And the following would undo them:

```
((UndoableCommand) history.pop()).undo();
```

Function objects

- C++ idiom takes callbacks further...
  - Behaves like a function?
  - Looks like a function!

```
class c_str_less :
  public binary_function<const char *, const char *, bool>
{
public:
    bool operator()(const char *lhs, const char *rhs) const
    {
        return strcmp(lhs, rhs) < 0;
    }
};
```

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

23

In C++ we can provide a further level of syntactic convenience and uniformity for Command based classes by overloading `operator()` for a class. A functor (or functional, functionoid, or simply function object) is an object masquerading as a function. The use of `operator()` in code relies on expectation and one of two forms of polymorphism:

- The operator may be provided as a pure `virtual` in an abstract base class, supporting conventional runtime polymorphism [*Advanced C++ Programming Styles and Idioms*].
- Alternatively, a generic programming approach may be taken to exploit compile time polymorphism [Koenig and Stroustrup, "Foundations for Native C++ Styles", *Software Practice and Experience* 25(S4)], i.e. overloading and templates, so that generalisation is at compile time rather than runtime.

The latter approach is used extensively in the part of standard C++ library commonly known as the STL (Standard Template Library). The library also provides classification template classes, such as `binary_function`, for use in this scheme.

Programming with higher order functions can be expressed by further extending the function object idiom [*Scientific and Engineering C++*].

Block

- Java's anonymous inner classes can be used to simplify use of Command pattern
  - Can define code that is dynamically bound to its context

```
Command callback =
    new Command()
    {
        public void execute()
        {
            block of code
        }
    };
```

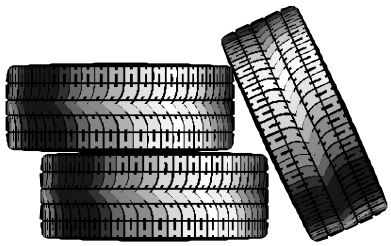

One of the key differences to the Java language between version 1.1 and 1.0 of the JDK was the addition of inner classes. Superficially inner classes look like the nested classes of C++. However, the resemblance ends there. Objects of inner classes can bind to the variables within the scope of their definition. For local inner classes, i.e. those defined in block scope rather than within a class, these variables must be declared `final`. Top level `static` classes may also be nested within classes; they are more like C++'s nested classes as they do not bind to instance data.

A further refinement is that an inner class may be anonymous. In this case it is defined as part of a `new` expression using simply the name of the superclass or interface as the constructor type and the class body is attached to that. This can be used to emulate Smalltalk's block, which is a proper object that can be composed and passed to another object to be evaluated using the value method. Refining the Command pattern and using anonymous inner classes we can achieve a similar, low ceremony closure effect in Java.

Inner class go further than simply single method classes, and are the basis for easy use of JDK 1.1's event handling model in its windowing library, the AWT.

Iteration

- Intent
  - Traverse objects in a collection
- Contents
  - Asymmetric bounds
  - Iterators
  - Concurrent and distributed iterators
  - Enumeration method



The area of iteration is rich in that it is both a common programming task and at the same time – in implementation – subject to a variety of different expressions depending on the context. Iteration patterns are a subclassification of behavioural patterns.

Although iteration is an expression of control flow, it is often associated with and reflected in a data structure. Such collections may be real, e.g. linked lists, or more abstract, such as input streams.

Asymmetric bounds

- C idiom for loop writing applicable to index based loops
  - Either direct index, i.e. integer, or abstracted, i.e. iterators

```
for(index = 0; index < size; ++index)
{
    use or update array[index]
}
```

26

At the most fundamental level, the anatomy of a loop is the same and corresponds closely to the structure of the C family's `for` loop:

```
for(loop initialisation;
    check for continuation;
    advance to the next)
{
    perform task
}
```

Another feature common to the C family, and indeed any language for which array indexing is zero based, is the use of asymmetric bounds for iteration [Andrew Koenig, *C Traps and Pitfalls*, Addison-Wesley]. Iteration is over a half open interval, i.e. `[0, size)` rather than `[0, size - 1]`:

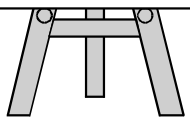
```
/* non-idiomatic looping over closed interval */
for(index = 0; index <= size - 1; ++index)
    ...
/* idiomatic looping over half open interval */
for(index = 0; index < size; ++index)
    ...
```

In this, asymmetric bounds in the loop structure reflect the data structure.

STL iterator

- An iterator is a pointer-like object
  - Either a raw pointer or a smart pointer
  - Abstracts idea of container position

```
template<typename type, ...>
class list
{
public:
  class iterator
  {
  public:
    type &operator*() const;
    type *operator->() const;
    iterator &operator++();
    iterator operator++(int);
    ...
  };
  friend class iterator;
  iterator begin();
  iterator end();
  ...
};
```



Given that a collection object is encapsulated, how can one conveniently access all of its elements in turn without knowledge of its internal structure? One solution is the Iterator pattern [*Design Patterns*]. It illustrates a clear separation of responsibilities and concerns with respect to objects: a collection object collects; an iterator object iterates. The allocation of objects to roles and responsibilities is a fundamental OO principle.

In the context of C++ the Iterator pattern may be expressed more idiomatically. Iterators are an abstraction of pointers in that a plain pointer is an indirect way of referring to an object and can be used to traverse built-in arrays. The STL takes this a step a further using the pointer protocol for iterator objects, i.e. `*` and `->` for dereferencing, and `++` for advancing. Different iterator categories support different capabilities, and a further refinement also defines `const` iterators for inspecting but not modifying elements – Barton and Nackman call such iterators *browsers* [*Scientific and Engineering C++*]. The design of the STL has been quite influential, and equivalent libraries now exist for Java (JGL) and Ada.

One benefit of using the same protocol is that generic algorithms may be used equally well regardless of whether iterators are either of some user defined class type or are plain pointers.

STL iterators follow the Asymmetric Bounds idioms, with a container defining a half open interval `[container.begin(), container.end())`. In other words, the end marker is one past the last element. It is a notional end marker, like the EOF at the end of a file.

Opaque iterator

Header file with container class definition and forward declaration of opaque iterator

```
class container
{
public:
    class position;
    position *begin() const;
    bool advance(position *) const;
    type &at(const position *);
    const type &at(const position *) const;
    ...
};
```

Source file with full definition of opaque iterator type and definition of container class members

```
struct container::position
{
    definition of iteration state
};
definition of container members
```

28

An iterator can be represented as an incomplete type, in which case it is passed to other functions for manipulation. This is a variant of the essential Iterator pattern using Opaque Type, such that the container type has visibility of the definition of the iterator type.

A correct implementation ensures type and memory safety. Note the way that `const` is handled: there is no need for a parallel type to handle this case. Once an iterator has reached the end of its traversal it can be invalidated and its memory reclaimed. For bidirectional opaque iterators an additional member function may be required, alternatively a proxy class may be introduced to manage the lifetime of the actual underlying opaque type.

The example above illustrates how Opaque Iterator may be used in C++ in addition to STL Iterator. The solution is also appropriate for C, especially where Opaque Type has been used for the container.

Note that the iteration model supported by MFC although superficially similar is a flawed and inappropriate design that is to type safety what politicians are to honesty. A poorly designed iterator is best described as an *irritator*. In short, an anti-pattern.

Polymorphic iterator

- Java library offers polymorphic iterators
  - java.util.Enumeration and java.util.Iterator

```
public interface Iterator
{
    public boolean hasNext();
    public Object next();
    public void remove();
}
```

Optional?!

```
public interface ListIterator extends Iterator
{
    public boolean hasPrevious();
    public Object previous();
    public void add(Object element);
}
```

29

The collections API introduced in Java 1.2 patches what has been a surprising and noticeable gap in the otherwise comprehensive Java standard library – a more surprising omission, perhaps, when it is considered that data structure and algorithm libraries are something of a traditional domain for object-oriented libraries.

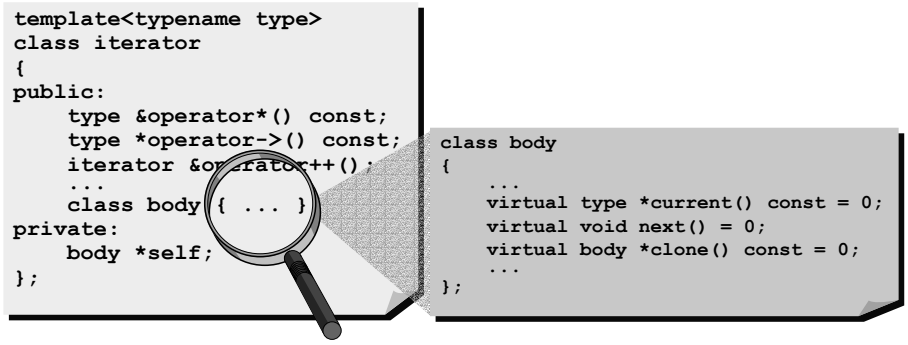
A more consistent model is introduced for defining and using collections, and part of this is iteration. There is in effect no significant difference between the new `Iterator` and older `Enumeration` interfaces except, as some have noted, the iterator interface now has the right name! There is, however, a questionable piece of design in the inclusion of a `remove` method which is considered optional, therefore bypassing clean subtyping and the use of Java's interface mechanism!

The library also introduces a collection interface hierarchy. There is an explicit separation between use and creation type: collections form an interface hierarchy, e.g. `Collection` and `List`, and beneath and to one side this there is a separate implementation hierarchy that includes different possible implementations for the same interface, e.g. `ArrayList` and `LinkedList`. The idiom for using this library is that only interfaces are declared in client code; classes exist only at the point of creation.

The `Iterator` interface may also be specialised, e.g. for `List`, to accommodate specific operations based on the target abstract data type. For instance, it makes sense to replace an element at an iterator's position for a list, but not for a set.

Abridged polymorphic iterator

- Polymorphic STL style iteration in C++?
  - Bridge is used for handle/body separation



To use Polymorphic Iterator in C++ there are some obvious drawbacks:

- The created iterator must be used indirectly via a pointer, and therefore the STL iterator idiom is inconvenient – to get to an object via an allocated iterator to a collection of pointers would result in three levels of dereferencing!
- There is the requirement that the programmer take explicit control of the disposal. This is error prone (it is too easy to forget) and prone in the face of error (it is not exception safe).

C++ is a value based rather than reference based language. This is reflected in the STL, which is entirely value based.

Two patterns can assist simplifying Polymorphic Iterator for use in C++:

- Handle/Body, which is the more appropriate synonym for Bridge [*Design Patterns*] in this context, separates an object into two halves: one half is the value based handle that the user manipulates; the other half is the body that implements the required behaviour, and is responsible for carrying the polymorphism.
- The Smart Pointer specialisation of Proxy [*Design Patterns*] is a delegation based pattern that adds some form of intelligence and simplification to a level of indirection. For instance, the use of Copy Before Release.

The Abridged Polymorphic Iterator idiom may be also applied to Opaque Iterator to wrap up them up as conventional self contained iterators.

Concurrent iterator

Safe iteration for collections shared between threads requires a single combined iterator operation

```
class container
{
public:
    void lock();
    void unlock();
    class iterator
    {
    public:
        bool next(value_type &result);
        ...
    };
    ...
};
```

31

There are four basic properties that an iterator collaboration must support:

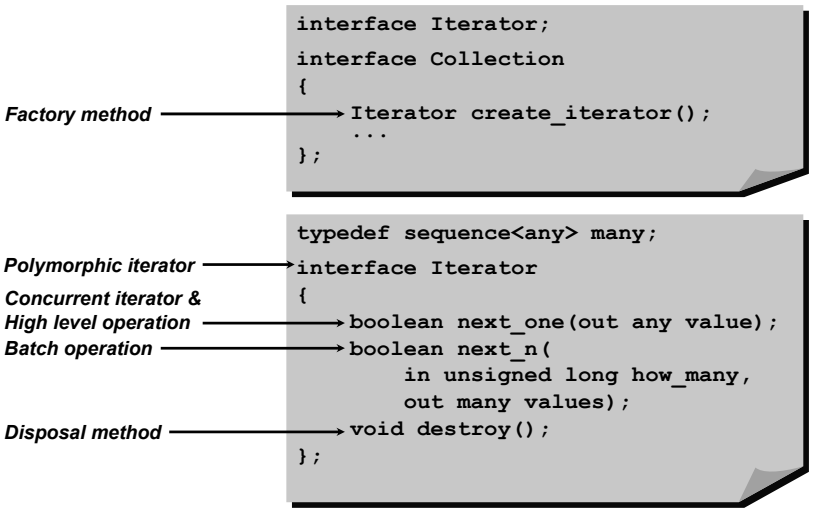
- Initialisation of the iteration;
- Access to the current iterated value;
- Advancing the iteration to the next value;
- The ability to determine if the iteration is complete.

Note also that these facilities may be mapped one for one with operations, or they may be folded into fewer operations that involve side effects.

In a concurrent system the uninterrupted sequential access of the basic features – except initialisation/creation – of an iterator is not guaranteed. For instance, having checked at the head of a loop that there are more values to fetch, there is no guarantee that by the time the query for the current value is executed that the collection will not have changed. Iterator validity is always an issue [*Design and Coding Reusable C++*], but it becomes more intractable in concurrent systems.

A solution is to combine the three main features of the iterator into a single operation, as shown above: `next` advances the iterator, returns `false` if no more elements are available, and sets the `result` argument to the element if `true`.

Distributed iterator



32

In addition to concurrency issues, iterators in a distributed context must address bandwidth concerns. The basic Concurrent Iterator addresses these: the essential operations of an iterator have been folded into a single one for integrity reasons, but there is also a performance payoff. A single remote call replaces three remote calls. The `next_one` operation is an example of High Level Operation: attributes are queried and set in related groups, reducing bandwidth usage and ensuring coherence. In other words, common usage of attributes is made atomic rather than the individual attributes' manipulation.

Efficiency is also the motivation for the `next_n` operation which slurps many elements ahead in a single go. This is an example of Batch Operation which addresses the problem of remote I/O bound loops repeatedly executing a simple operation. This incurs a great deal of communication overhead for relatively little computational benefit. The solution is to 'slurp' as much across as possible in a sequence. What this lacks in elegance it makes up for in efficiency.

The nature of distributed systems ensures very late binding of implementation to interface, and so iterators in a distributed system are implicitly polymorphic. The creation and initialisation is handled by the target collection, using Factory Method [*Design Patterns*], whereas its inverse (Disposal Method) is used to either destroy or return it to the collection to be cached for future use.

This iterator pattern is used in many of the CORBAservices. Interestingly it is not the model adopted for the CORBA Query and Collections Service! This uses something more along the lines of Java's iterator classes.

Enumeration method

- Inversion of iterator model
  - Iteration is encapsulated within the collection
  - Collection stateless with respect to iteration

```
public class Collection
{
    public static interface Operation
    {
        void executeOn(Object element);
    }
    public synchronized void forEachDo(Operation toDo)
    {
        for each element do toDo.executeOn(element)
    }
    ...
}
```

33

Enumeration Method [*Smalltalk Best Practice Patterns*] is a fundamental iteration pattern that encapsulates the actual control flow of the traversal loop. There are strong similarities with the Template Method design pattern [*Design Patterns*] that encapsulates an algorithm at the base of an inheritance hierarchy, as well as Visitor [Design *Patterns*] and Execute Around Method [*Smalltalk Best Practice Patterns*].

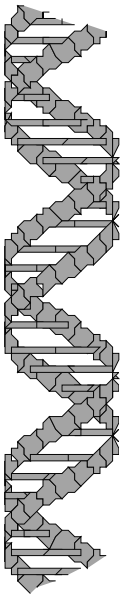
Implementations of Enumeration Method can be found in any language that supports some reification of the concept of a function, e.g. function pointers in C and C++ or behavioural objects in C++ and Java.

The question arises as to when to use Iterator and when to use Enumeration Method. Their intent and consequences are slightly different: Enumeration Method is a control flow pattern in that it encapsulates an algorithm, and a collection may offer the most common forms of collective usage in a method category with various implementations of Enumeration Method; Iterator is an access control pattern that abstracts a reference to a position within a collection and may be used to implement algorithms that are not genuinely part of the collection's remit, and which therefore should not clutter its interface.

In some senses, Enumeration Method is higher level: it captures and names common traversals for the programmer to use directly. It may also be combined transparently with acquisition and release patterns to ensure transactional and threadsafe behaviour, i.e. atomic, exception safe and consistent. However, although it works well for concurrency, it does not scale well for distribution unless a by value mechanism can be used.

Summary

- Language affects thinking
  - Idioms are norms in a language culture
- Suitability of patterns may vary with implementation language
  - Idioms change with context
- Idioms may need to be imported from one language to another
  - Break culture out of a local minima



By necessity, coverage of particular programming styles and techniques in this talk has not been complete. Hopefully, it has shone the spotlight on how a change in context – such as language or execution arhcitecture – affects even the most mundane tasks. Similarly, how idioms may be common to languages or technologies with the same features.

It is always important to keep in mind that programming is a social activity in the sense that very few programs are developed over time for one or more people by one or more and are not created in a void. Even the lone hacker, perhaps the archetype that would seem to fly in the face of the previous statement, participates in a culture. For instance, Eric Raymond's "The Cathedral and the Bazaar" and "Homesteading the Noosphere" [*http://earthspace.net/~esr/writings/*] document the culture and motivation of the open source movement.

Given a cultural context, programming in a language and with a particular set of technologies becomes more than a simple matter of engineering.