Interface Based Development

Kevlin Henney
kevlin@curbralan.com

Presented at the *JaCC*, Oxford, 25th March 2000.

Kevlin Henney

`kevlin@curbralan.com`

Curbralan Ltd

Phone	0117 942 2990 (UK)	+44 117 942 2990 (International)
Mobile	07801 073 508 (UK)	+44 7801 073 508 (International)
Fax	0870 052 2289 (UK)	+44 870 052 2289 (International)

Overview

- Interfaces
  - Technical view of interfaces
- Development
  - Development with interfaces
- Practice
  - Worked examples

2

Interfaces between components in software, whether at the function API level or executable component level, define and constrain the possible relationships within the software and the behaviour of its parts. This talk takes a broad view of development based on defining and understanding software interfaces, focusing on concepts and practices appropriate for C, C++, Java, CORBA and COM development.

It is often the case that developers slap an interface onto an implementation as an afterthought. Such an approach stems from the view that an interface does not actually "do anything", and therefore procedural code is more significant both in terms of its effect and bulk. Hence less effort is invested by developers in writing the interface.

Although an interface indeed requires much less code to express it than an associated implementation, this is perhaps inversely proportional to its relative significance. The interface is the point of agreement between a component supplier and consumer, and should therefore be well considered, complete, comprehensible and stable. Such a state of affairs cannot be reached through casual coding. Bugs in an implementation may be irritating, but they are failures of a component to satisfy an interface, and they can be fixed without adversely impacting clients. Modifications to poorly designed interfaces, however, will break clients that have been written against (and worked around) them; such changes will be seen as causing problems rather than fixing them.

Interfaces

> **Interfaces are about psychological chunking.**
>
> ***David Ungar***

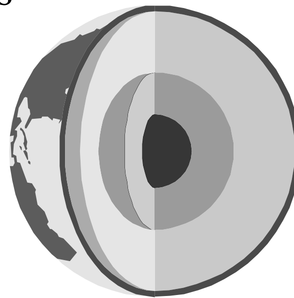
Object orientation offers us a convenient, if somewhat oversimplified, view of what an interface is, in the sense that it is the publicly accessible set of features on an object. Component-based development – in the sense of COM, CORBA, JavaBeans, etc – goes further, explicitly formalising the concept as a class-like construct that bridges to a more rigorously black box definition of implementation than OO languages offer.

However, the term interface is heavily overloaded, and the metaphor it represents is in many places throughout software: graphical user interface, device interface, function interface, class interface, etc. In some ways the current trend towards component-based development (CBD) has hijacked the term, restricting it to mean only a platform independent construct associated with a deployable unit of code.

More generally an interface represents a boundary between two elements, and describes the interaction across it. The focus here is on interfaces between parts of a program as opposed to many of the other uses of the term *interface*.

Boundaries

- Architecture captures the arrangement of structural elements in a system
  - From its gross structure to its detail
- Interfaces represent partitions and introduce separations
  - Intent from realisation
  - Conceptual from concrete
  - User from author



4

It is possible to home in on a definition of software architecture by framing the questions that must be asked of it [Dyson1998]:

> *An architecture is* something *that answers the following three questions:*
> 1. *What are the structural elements of the system, what are their roles, and how do we share responsibility between them?*
> 2. *What is the nature of communication between these elements?*
> 3. *What is the overriding style or philosophy that guides the answers to these two questions?*

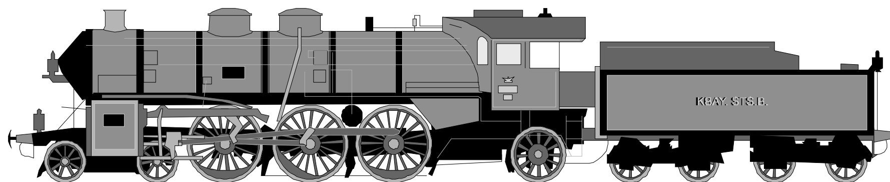
Architecture is recursive and in-depth, so it is not just about the big blocks: takes in the structure and style from imposing load bearing pillars and elegant sweeping arches to the architraves and door handles. In this sense it mirrors development (or vice-versa), which must be concerned with production and process at all levels of detail.

Architecture may be considered the result of a conscious design process, or simply a post hoc description of a system configuration. Thus, in spite of its etymology (*architect* is derived from the ancient Greek for *chief builder*), architecture can be accidental rather than intentional.

In establishing a structure, boundaries are defined and there is a separation of inside from outside. This boundary is an interface, and the metaphor extends to software with a separation of executable constructs on either side of a software interface.

Partitioning

- Quality and qualities of separation
  - Coupling describes interconnectedness
  - Cohesion describes intraconnectedness
- Separation introduces connections
  - A structure of components and connectors



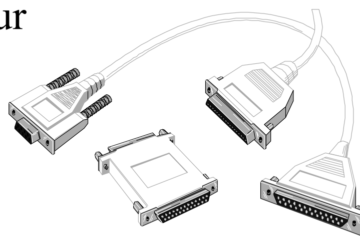
In managing complexity, partitioning a system allows work to be understood, managed, and executed, and offers a scalability and security greater than a single individual's mind at a point in time (i.e. gives a project a higher *truck number*, as observed by Don Olson and Neil Harrison [Rising1998]).

Such a partitioning should be controlled rather than arbitrary, and guided by well understood principles. This is where interfaces as elements of design become significant; they form durable technical and political boundaries. They describe the nature of interaction, i.e. of the connection, between different components in a system. Coupling and cohesion are properties that can be observed of any partitioning.

Note that the definition of *component* used in the context of Component-Based Development (CBD) [Szyperski1998] is a specialisation of the more general term used here. CBD uses the term more strictly to mean identifiable, executable unit of deployment, i.e. COM, CORBA and the Java technologies.

Protocols

- Connection implies communication
- Interfaces name and express usage models
  - Play a role in communicating meaning within a system as well as encapsulating its parts
  - Conformance to a published interface implies particular behaviour



6

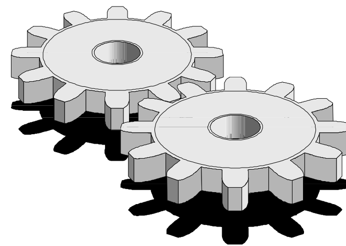
However, an interface does not simply set up static barriers in a system. Interfaces are permeable, allowing flow of concepts – information and behaviour – across the boundary. They establish and name models of usage. They represent the protocol used to connect components together.

With any flow there is possibility of leakage, and this is true of interfaces. Interfaces offer a separation of checkable usage from implementation, often with the intent of encapsulating the implementation. However, unless care is taken, details of the implementation can seep (or gush) through the interface, whether as pointers to private state or usage that is clearly coupled to the otherwise private representation.

Interface communicate meaning in a system and therefore they should be clear and expressive. A counterexample of expressiveness is the constructor for Java's `FileWriter` class which requires a `boolean` to indicate whether or not a file should be opened for appending. The use of such flags is inexpressive. A more appropriate interface would be to offer named class Factory Methods [Gamma+1995] `openForAppending` and `openForWriting`.

Mechanisms

- Common approach to defining interfaces is to focus on a class's public operations
- Focusing on the interface alone leads to further decoupling
  - Using *interface* in Java or IDL
  - Using interface classes in C++, with only *public* pure *virtual* functions are defined



The conventional notion is that an interface is the public section of a class. This is what clients of its instances depend on, and in effect all they can call. Inheritance introduces the idea of accumulation of interface, typically in step with accumulation of implementation.

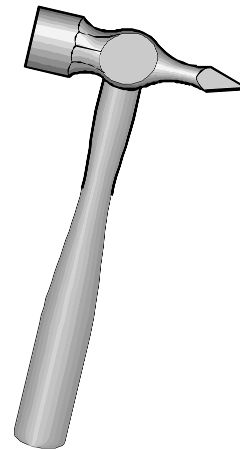
If the client wishes to depend on interface alone rather than implementation detail, some form of interface decoupling is needed. If they specifically require runtime polymorphism, reduced compile time dependencies, and dynamically allocated objects, this is provided directly in COM and CORBA; alternative mechanisms are not available. In Java an `interface` can be defined explicitly. In C++ an interface class [Carroll+1995] can express the common capability of derived classes, i.e. a class to represent the contract and many derived classes to fulfil it and express the implementation detail. This class has no data and the only ordinary member functions are declared `public` and pure `virtual`. If it is used as a mix-in class – albeit a mix-in that provides protocol only – `virtual` inheritance should be considered to avoid repeated inheritance issues.

Where an interface class represents the *usage type*, or some aspect of the usage, of an object, the concrete class instantiated for the interface user represents a *creation type* [Barton+1994]. This distinction can be made clearer, and the dependencies reduced in a system, by enforcing such a model of use: the interface class is used only for manipulation and the only time the name of the concrete class appears is at the point of creation, i.e. in a `new` expression, which may itself be encapsulated within a factory object.

Beyond Class

Interfaces are more than a concept based only on public operations of a class...

- **Function-based APIs**
- **Scoping and packaging**
- **Operator overloading**
- **Template generic requirements**
- **Non-public class operations**
- **Markers and external frameworks**



Software interfaces go much further than class interfaces. In C an interface is considered to be the type and functions deployed in a header file. In IDLs the definition of interface seems self explanatory, but must also take into account file partitioning. Java has explicit interfaces as well as classes, which have their own public, package and familial interfaces. For C++, the Interface Principle [Sutter2000] defines a class as a slightly more extended family :

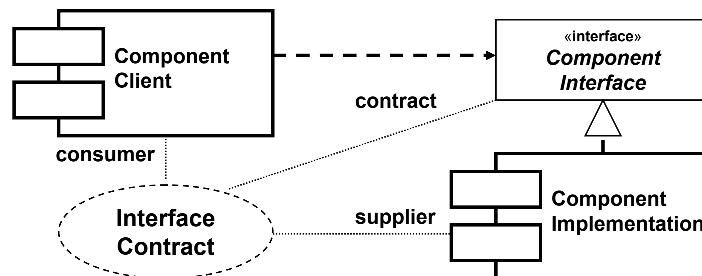
> *For a class X, all function, including free functions, that both (a) "mention" X, and (b) are "supplied with" X are logically part of X, because they form part of the interface of X.*

This takes into account Koenig Lookup which give namespaces stronger semantic connotations than simply a name collision avoidance mechanism. The fact that class interfaces extend outside the interface is further reinforced by styles statements made in [Stroustrup1997] and [Meyers2000].

In Java the existence of reflection provides again a different view of interfaces and conformance. The common use of marker interfaces, such as `Remote` and `Serializable`, demonstrates again how class behaviour can exist outside the class definition. The same is true in any framework, but it is the distinction is made in sharp relief with marker interfaces because they are empty, tagging a type for a capability. It is the helper classes in the framework that provide the full capability based on inspection of the type.

Design by Contract

- An interface defines an agreement between a component supplier and its users
  - Goes further than simply defining signatures to include guarantees on behaviour



Type systems provide part of the story when it comes to establishing interface usage, but at best they can provide no more than compile time confidence in the structure; in many systems there is plenty left to hit the fan at runtime if interface usage is well formed but otherwise incorrect. A contractual view establishes further limits on an interface by defining the legal requirements on the behaviour of operations.

Most methods concern themselves with only the functional requirements of a system, i.e. what the system must do. Such requirements offer a high degree of traceability through the lifecycle. However, it is often the case that there are a number of non-functional requirements that are as important to a system, i.e. how a system does what it does. Such requirements include quality of service, failure strategies, use of specific technologies, etc. These can be harder to quantify and test for in a design, but are nonetheless often of great importance. For instance, the non-functional behaviour of a distributed system cannot merely be dismissed as an implementation detail. In C++'s standard library many generic functions have complexity constraints placed upon them.

Where quality of service requirements are significant, these must clearly be a part of the contract. However, they are difficult to explicitly capture in code, as non-functional concerns tend to cut across the component nature of a system. As with other forms of contractual requirement, quality of service provided can exceed, but not fall short of, the quality of service required.

Substitutability

- A measure of goodness of fit between interface and implementation
  - Interfaces offer uniform treatment of similarities between implementations
  - Conformance to interfaces by implementations

Express coordinate ideas in similar form
This principle, that of parallel construction, requires that expressions similar in context and function be outwardly similar. The likeness of form enables the reader to recognise more readily the likeness of content and function.
Strunk and White [Strunk+1979]

10

The Liskov Substitution Principle (LSP) [Liskov1987, Coplien1992] is often cited as giving more detailed guidance on the use of inheritance. It make a clear separation between type – an interface described in terms of the behaviour of operations – and class – the realisation of the behaviour in programmatic detail – before establishing the criteria for a subtyping relationships in terms of conformant behaviour:

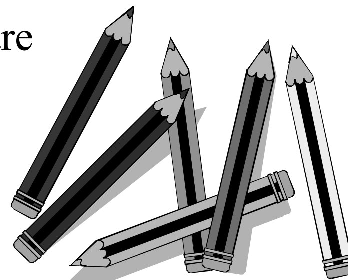
> *A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a* subtype *is one whose objects provide all the behavior of objects of another type (the* supertype*) plus something extra. What is wanted here is something like the following substitution property: If for each object 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.*

LSP is normally presented as an inheritance guideline, but taking a step back we can see that it need not be so narrow: It is a relationship about types and not implementations, i.e. subtyping and not subclassing. LSP relies on polymorphism; the concept of structural relationship, such as conventional inheritance, need not enter into it. Deserving of its name, polymorphism manifests itself in many forms [Cardelli+1985].

In C++, substitutability can be found through conversions, overloading, derivation, mutability and generics; Java does not have quite the same reach with conversions, overloading or generics, but supports reflection, offering a different substitutability mechanism in turn.

Choice

- An interface should represent reasonable goals and present reasonable choices
- Additional options and features can lead to confusion rather than clarity
  - Overachieving interfaces are weaker and more complex not stronger and simpler



11

The belief that "less is more" seems to be heeded more in the breach than in the observance. It seems a common enough piece of advice from which we can learn and shape our software [Strunk+1979]:

> *[Rule] 17. Omit needless words*
>
> *Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all his sentences short, or that he avoid all detail and treat his subjects only in outline, but that every word tell.*
>
> *Many expressions in common use violate this principle.*

Consistency in interfaces is important in meeting expectation and presenting reasonable choices; C++'s `basic_string` and `vector` functions `at` and `operator[]` break this principle. They differ in their bounds checking: `at` does, and throws an exception if necessary, and `operator[]` does not. To use anything other than the intuitive and idiomatic subscript operator takes a conscious effort, one that must in this case be accompanied by a willingness to not write correct code – "I'm using this because I have chosen to write code that will go out of bounds"!

A misguided quest for completeness often leads to unmanageable kitchen sink interfaces that lack focus, e.g. C++'s `basic_string` template class. Design for "flexibility" without goals leads to complex interfaces that perpetuate the goal-less design decisions, e.g. C++'s `allocator` model.

Development

> **To achieve simplicity paradoxically requires an enormous amount of effort.**
>
> ***John Pawson [Pawson1996]***

In establishing the partitions that create a structure we can take a more useful view than simply stating that the design must satisfy the user requirements ([Petroski1992] quoting *The Structural Engineer*, the official journal of the British Institution of Structural Engineers):

> *Structural engineering is the science and art of designing and making, with economy and elegance, buildings, bridges, frameworks, and other similar structures so that they can safely resist the forces to which they may be subjected.*

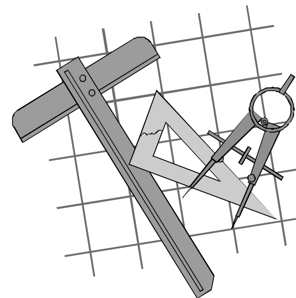
This provides a useful starting point from which to view development of software systems. In software, the principles and practices of development sometimes seem to be as flexible as the medium of software itself... and sometimes just as rigid.

Whilst design should not be an end in itself, it is something that will assist in any construction, large or small [Strunk+1979].

> *[Reminder] 3. Work from a suitable design*
>
> *Before beginning to compose something, gauge the nature and extent of the enterprise and work from a suitable design. Design informs even the simplest structure, whether of brick and steel or of prose. You raise a pup tent from one sort of vision, a cathedral from another. This does not mean that you must sit with a blueprint always in front of you, merely that you had best anticipate what you are getting into.*

Design

- Design is a creational and intentional act
  - Conception and construction of a structure on purpose for a purpose
- Interfaces provide a connection between the conceptual and the concrete



13

Design is synthesis as opposed to analysis. In truth, much of what is considered to be analysis in software development is design, and a separation of implementation from design (or vice-versa) is also a false division [Lea1998]:

> *Sometimes, describing software is the same as constructing it.*

Design embraces a more profound endeavour than simply elaborating a simple model of the problem into a complex one, and then coding (Winograd and Flores quoted in [Lea1998]):

> *The most successful designs are not those that try to fully model the domain in which they operate, but those that are "in alignment" with the fundamental structure of that domain, and that allow for modification and evolution to generate new structural coupling.*

Such a structural view is intertwined with a process [Coplien1999]:

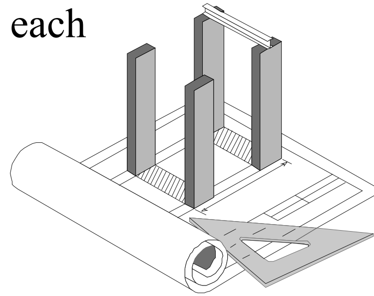
> *Design is the activity of aligning the structure of the application analysis with the available structures of the solution domain.*

Thus, where architecture is a description of system structure, regardless of intention (i.e. all systems have architecture, whether deliberate or not), design describes an intentional activity. 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.

Design has many foci, including dealing with conceptual interfaces in a problem, describing the relationships between conceptual parts, and actual interfaces in a solution, delimiting the constructed parts.

Models

- A model is an abstraction from a point of view for a purpose
  - But don't confuse the map with the territory
- Interfaces embody purpose and usage
  - The designer and the user each have conceptual models
  - Modelarity is degree of alignment between different views



Just as we can view a model as a bridge between the problem and solution domains [Jackson1995], we can have separate models of the problem domain from models of the solution. [Norman1989] further differentiates the relationship between the solution domain and the user's interaction with it:

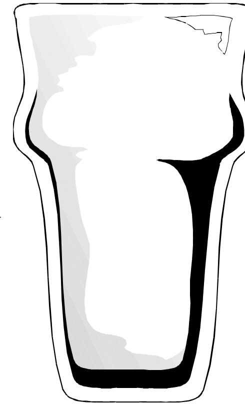
> *The* design model *is the designer's conceptual model. The* user's model *is the mental model developed through interaction with the system. The* system image *results from the physical structure that has been built (including documentation, instructions, and labels). The designer expects the user's model to be identical to the design model. But the designer doesn't talk directly with the user – all communication takes place through the system image. If the system image does not make the design model clear and consistent, then the user will end up with the wrong mental model.*

If a model gives you an understanding of the problem, try to put as much of that understanding into the solution as possible. Don't go all programmatic! For example, property style programming (typified by *getters* and *setters*) devalues the meaning of a system; it becomes weakly defined rather than generally defined. There can be a tendency to design interfaces that have *just enough encapsulation* [Box+1999], leading to *structification* of objects [Taligent1994]. As an unquestioned habit it leads to such ridiculous methods as *setSpouse* on a *Person* object and *setBalance* on an *Account* object, which ignore the vocabulary, behaviour and constraints of the original domain to no good effect.

Modelarity can be considered a measure of the correspondence between different views, e.g. problem and solution, designer and user. At the same time, one must not mistake the map for the territory.

Constraints

- Constraints bound the meaningful behaviour of a system
  - Intended degrees of freedom
- Constraints can be liberating
  - Ensuring what's true is true and what's not is not frees rather than binds a developer



Design should observe and preserve constraints. A system that weakens them has the illusion of being more flexible, but in truth is simply vaguer and less committed, opening up more gaps in which bugs can breed.

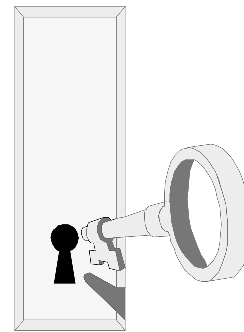
A simple example of constraint preservation is the use of `const` in C++ to clearly indicate to compiler and human alike something plays the role of a query function or query only data. The preservation of such a constraint makes the system richer. In Java a more inductive approach based on naming conventions, e.g. the JavaBeans `get*` convention, immutable value objects, and `final` are used to achieve a similar effect.

For another example, let us say that it has been established that a relationship between two objects is mandatory, i.e. one-to-one, then implementation in Java using an object reference or in C++ using a pointer allows the possibility of a null, i.e. one-to-zero-or-one. It is the responsibility of the developer to ensure that nulls are recognised as meaningless and handled appropriately, rather than to assume correct usage of a cluster of classes. Also, where does such a relationship come from? If it can never be null, this means a valid object cannot be created without being given a non-null relationship at creation. This effect manifests itself in Java and C++ in the constructors provided. If this cannot be achieved, is the constraint in the problem domain correct? Or must it be loosened and enforced another way in the software?

One temptation in C++ is to attempt to use references to enforce non-nullness. However, these convey a very different meaning to C++ programmers – and indeed C++ compilers – and so more is lost than is gained by such an approach.

Affordances

- An interface affords particular usage
  - An interface represents intended use, but it does not necessarily mean it will be used that way
- Constraints and affordances should match up
  - The intended and actual degrees of freedom should be similar
  - Minimise the possibility of incorrect use



Affordances describe possible as opposed to intended usage [Norman1989]:

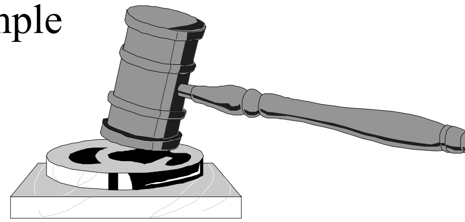
> *The term* affordance *refers to the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used. A chair affords ("is for") support and, therefore, affords sitting. A chair can also be carried. Glass is for seeing through, and for breaking.... Affordances provide strong clues to the operations of things. Plates are for pushing. Knobs are for turning. Slots are for inserting things into. Balls are for throwing or bouncing. When affordances are taken advantage of, the user knows what to do just by looking: no picture, label, or instruction is required. Complex things may require explanation, but simple things should not. When simple things need pictures, labels, or instructions, the design has failed.*

In essence, where possible, constraints should be communicated through affordances and affordances should align with constraints. Any mismatch creates an opportunity for misunderstanding and misuse.

For instance, a sign of weakness in a class interface design is that its users are required to remember lots of particular and subtle conditions of use: "function *a* must be called before function *b*, unless condition *c* is true, in which case function *d* followed by *e*, etc". Such interfaces suggest that the design is incomplete because the user of the class is doing most of the work that should be captured by a good design. Their code is repetitive; writing it is error prone. The problem of intermediate states and subtle sequential dependencies is a common problem in interface design.

Legislation and Litigation

- Contracts can be expressed formally
  - Using preconditions and postconditions to define boundaries of correct behaviour
- Contracts can be expressed by example
  - A more empirical approach, often based on specification by example and unit tests



The signature level of an interface can be captured in many languages, but the full semantics of the contract are less clear. Pre and postconditions offer one way of reasoning about abstract behaviour [Meyer1997]:

- A precondition defines the conditions that must be met by the caller requesting an operation. For instance, the permitted ranges of arguments, the required state of the object, etc. If these conditions are not met, the operation cannot be expected to perform its task correctly.
- A postcondition defines the conditions that must be met by an operation assuming that the precondition has been met; it is the supplier's half of obligations in the contract.

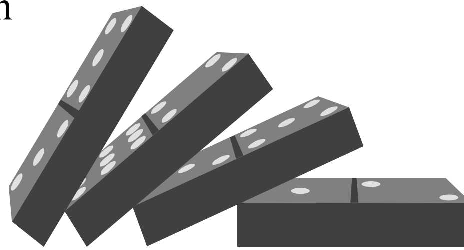
They establish a theory of correct behaviour of a system. The enforcement can range from a fully robust, supplier checked approach based on exceptions to nothing. `assert` is sometimes used in such cases, but suffers from such frequent misuse that its use should often be questioned. Documentation can be used to express contracts before the fact; after the fact, if no other enforcement is provided, violations lead to undefined behaviour (i.e. debugging).

An alternative perspective is to specify by correct usage, so that the interface's behaviour is expressed and checked through unit tests and sample code. This more empirical and inductive approach underpins iterative development strategies [Beck+1998, Beck2000, Fowler1999, Gamma+1999].

In effect these represent the two ends of the spectrum, with many developments – that make a choice – lying somewhere between the hardline theoretical and the empirical specification-by-example approaches.

Dependencies

- Partition to minimise dependencies
  - Low coupling and high cohesion
- Coupling and cohesion are relative rather than absolute measures
  - Decoupling is with respect to *what*?



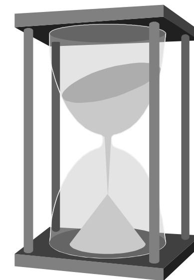
One of the features that typifies any architecture driven approach is the management of dependencies in a design [Lakos1996]. Dependencies should be managed throughout the runtime, design time and construction time of a system. Coupling and cohesion define, respectively, inter-connectedness and intra-connectedness of components and their interfaces. It is these quantities that must be managed if an architecture is to be stable and resilient in the face of change, supporting natural growth and evolution, as well as out of the box fitness for purpose and buildability. Interfaces may be established with respect to levels of abstraction, rate of change, development skills, or organisational structure.

On the whole a designer should strive to minimise dependencies between elements of a system. This should not be at the cost of making elements uncohesive. They should be as loosely coupled as is meaningful, and this will lead to a more supple component structure. In turn this should lead to a more maintainable and stable system. Where something is stable it can be depended upon without concern.

A subtle, but nonetheless problematic, form of coupling comes in the form of cyclic dependencies, where one component depends, directly or indirectly, on the contents of another which in turn depends, directly or indirectly, on the first component.

Stability

- Dependencies should be on more stable elements with the same rate of change
  - Put things together that change together
- Interfaces should be more stable than their implementations
  - Either because of good design or because of fear of change



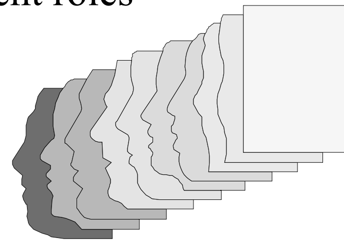
One mark of success is how an architecture endures, how it responds to change, how it suggests change, how it is accepted by developers, and so on. Thus an architecture may also measured against change and the passage of time [Brand1994 , Dai+1999]. It is therefore tempting to program only in the future tense, adding complexity by building for possibilities that may never happen, but it is also tempting to program only in the here and now, ignoring the possibility of and processes for change, and therefore being surprised and unprepared when change occurs. Thus we can see an additional quality in an architecture [Coplien1999]:

A good architecture encapsulates change.

In addition to managing physical dependencies to minimise the effect of change, architects must also be aware of what can and cannot change easily: interfaces that are private to a component can be more volatile than those that are public, and therefore part of more durable (and accountable) contracts. This can be considered a distinction between public and published interfaces [Fowler1999]. [Martin1995] outlines a metric that can be used to gauge the relationship between abstractness and dependency, based on the principle that the more abstract something the more stable it should or must be, i.e. program to interface not an implementation [Gamma+1995].

People and Technology

- Partitioning is not simply about abstraction
  - Additional forces create partitions with respect to organisational and technical boundaries
- Interfaces have social and economic effects
  - Interfaces define development roles
  - Complexity management is dependent on stakeholders and investment

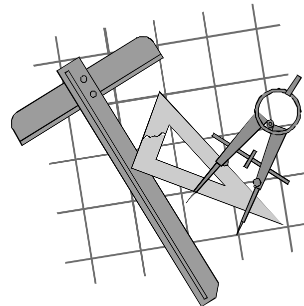


Carolyn Morris defines a framework, in the most general sense, as "a skeleton on which a model of work is built", and this is no more true than it is in software. In addition to the conventional idea of a code framework as a half finished application, we can have conceptual frameworks. For the developer an architecture is such a framework. It partitions the system both with respect to its code structure but also with respect to responsibilities for developers. This has implications for organisations when it is realised that an organisation also defines a model for communication. This leads to patterns such as Conway's Law [Coplien1995] where organisation follows architecture and vice-versa.

Therefore interfaces in a system will determine to a great degree how it is built, and in terms of how developers organise around the tasks of development and each other. This dynamic aspect extends beyond the initial idealism of green field development to influence how a system responds and adapts over its lifetime. Regardless of whether or not the architecture is explicitly articulated, it will affect how effectively developers can modify a system, how easily such change can be managed, and how long a system will live before outgrowing either its utility or its worth.

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



Design is not simply a feed forward process where 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 it to be 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. The developer participates in a complex set of decisions and is not merely a cog in the works or a hands-off analyst.

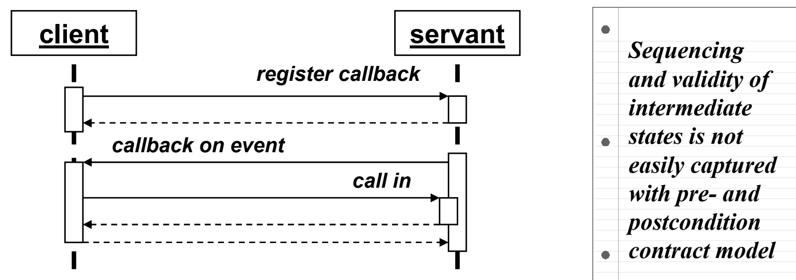
The degree to which a language or technology specifically supports certain mechanisms will have an impact not only on the way that programmers will think about a problem, but also on the way that a system should be designed. The realisation that there is a two way flow between architecture and implementation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

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.*

Re-entrancy

- Event notification from a component object to its clients handled through callbacks
  - Client may then call in to component object



How are events propagated out through an interface to interested parties? The callback model allows clients to register an interest in specific – or all – events. Typically inward calling interfaces in event driven environments are associated with one or more outward interfaces. The Observer pattern [Gamma+1995] details the basic callback form for one to many dependencies; Model/View/Controller [Buschmann+1996] represents a more strategic pattern built on these principles.

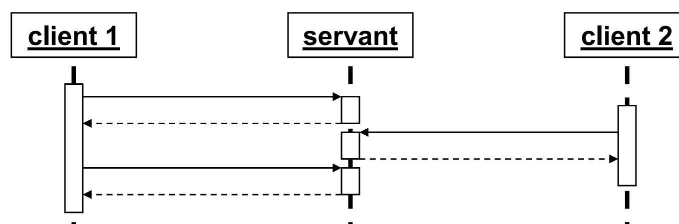
The common contract model, based on pre and postconditions, works well for conventional call-return procedural control flow. Limitations are, however, revealed when describing asynchronous callback architectures [Szyperski1998].

Where the flow of control is downward and explicit, pre and postconditions can be asserted about discrete and coherent states of an object. Where callbacks occur from a component to a client in response to an event within the component, it is typically the case that the client will then query or manipulate the component during the callback (the pull model of notification). It is possible that intermediate states in the component may be revealed (the converse is also true of the caller if a callback occurs during registration).

Sometimes a simpler and more visible approach is to capture the constraints through dynamic rather than static models. Sequence and state models are better used to bound correct behaviour in the presence of callbacks by defining legal sequences of actions. This can be seen, to some degree, to represent a conflict between static and dynamic models of a system. An alternative view accommodates re-entrancy, and also validity in concurrent execution contexts, by extension of the basic pre and postcondition contract model. Operation invariants, in the form of *guarantee* and *rely* conditions [D'Souza+1999], make assertions about the intermediate states of an operation, thereby guaranteeing the conditions under which a callback may occur and on what it may depend.

Concurrency as a Context

- Synchronisation is required to ensure consistent and coherent state
- Property-style programming is inappropriate
  - E.g. MIDL *properties,* OMG IDL *attributes*, *set* and *get* operation pairs, etc.



Property style programming, whether through the use of attributes (e.g. OMG IDL's `attribute`) or simple operations relating to attribute-like values (e.g. paired *get* and *set* operations) often leads to sequences of operations which assume that an object remains in the state the caller last left it. Without explicit locking this cannot be guaranteed, and the absence of some kind of synchronisation might lead to surprising behaviour.

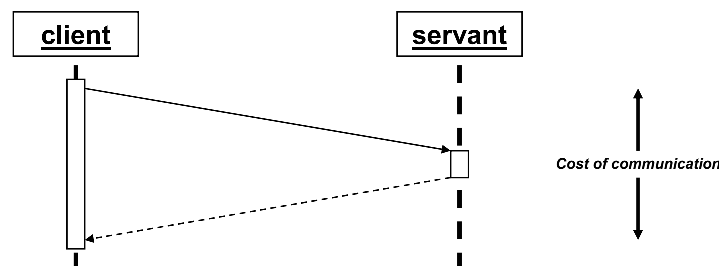
Note that attempting to generalise concurrent programming practices from sequential programming is the wrong way: sequential programming is a limited case of concurrent programming, and not vice-versa. This means that techniques such command/query separation and programming by contract [Meyer1997] do not automatically translate as is into a new context.

[Mannion1999] attempts to argue that in Java class clients should use `synchronized` explicitly in their code, i.e. it is not the class supplier's responsibility to resolve concurrency issues, it is the class user's. Whilst such devolution certainly leads to more complex and more error prone client code, with notable loss of transparency and some efficiency, the crunch comes when it is realised that it is not simply a matter of style preference to reject universal use of this approach: there are common cases when it simply does not work.

Enforcing the separation works in simple cases where object communication is direct (i.e. the reference used to communicate with an object is actually a reference to that object) and reliable (e.g. not distributed); in the presence of proxies [Gamma+1995], such as used in RMI, the use of `synchronized` blocks fails: the synchronisation is on the proxy object and not the target object.

Distribution as a Context

- Concurrency is implicit
- Operation invocations are no longer trivial
  - Communication can dominate computation
  - Partial failure is almost inevitable



Concurrency and distribution introduce design contexts unfamiliar to many developers, and ones fraught with subtleties. If the consequences of decoupled execution are not fully appreciated (i.e. the developer must genuinely *grok* them rather than pay lip service to them), the subtle design context becomes a subtle debugging context.

Operation invocations are assumed, in most designs, to be instantaneous and reliable. In a distributed system the process of delivering invocations across a network requires extensive middleware support, meaning that the connection domain [Jackson1995] can dominate the behaviour of the system.

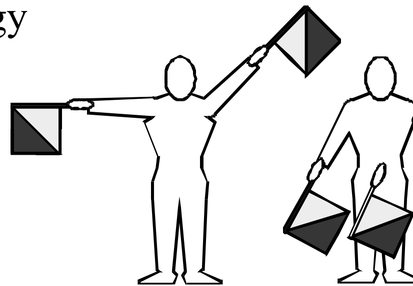
In addition to all of the issues raised with concurrency, there is additional cost involved in remote operation calls. When sketched out on a sequence diagram, sequences of property *get*s and *set*s suffer the 'sawtooth' effect, spending more time in communication than they do in performing useful work. The ratio of communication to computation is a key consideration in distributed computing.

The scope for failure is even greater in a distributed system than in a local concurrent system. Consider a failure during a sequence of queries of a server object by a client: the client is left with an incoherent and partial view of a server object if an invocation fails during a sequence of queries; likewise, and perhaps more damaging, is the event of failure during a sequence of modifications which may leave a server object in an incoherent state.

All of these issues extend the issues raised by concurrency. For large and complex operation sequences involving the use of many objects, a transaction processor (e.g. OMG's OTS, Java's JTS, Microsoft's MTS) is appropriate; for small common operations on a single object, a TP is overkill.

Idioms

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



Idioms are what the locals speak. In this case techniques applied by the users in a programming language culture and common execution contexts. They help to stabilise language usage and create a common vocabulary of techniques, constraining the potentially infinite possibilities of a language grammar and semantics. One of the classic works on idioms is James Coplien's book on advanced C++ [Coplien1992]. More recently Kent Beck documented many Smalltalk patterns [Beck1997]. Idioms can be considered low level patterns, being indigenous to a particular language, language paradigm (e.g. procedural as opposed to functional), or technology (e.g. distribution).

Some idioms, such as those for procedural control flow, can be transferred easily across languages. Others depend on features of a language model and are simply inapplicable when translated: strong and statically checked type system (e.g. C++ and Java) versus a looser, dynamically checked one (e.g. Smalltalk and Lisp); reference counting mechanisms for C++ are made inappropriate in Java and Smalltalk 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 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. Many implementation and interface design practices are often bound to an implied context rather than being truly general. For example, concurrency invalidates practices appropriate for sequential code.

Practice

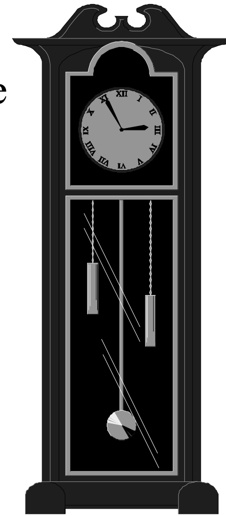
The history of all hitherto existing society is the history of class struggle.

Karl Marx

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

Clock

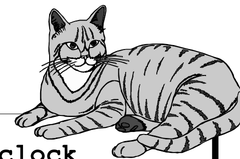
- Example...
  - A simple clock class that can handle queries and modal setting of time
- Forces...
  - Representation hiding
  - Thread safety
  - Expressiveness
  - Encapsulation of behaviour



A simple clock example in C++ can illustrate detailed design with a focus on interface and the issues affecting interface, such as concurrency and dependency management, can affect the exterior and interior design of a class. It is by no means either a rocket science or realistic application class, but it serves to illustrate valid interface-based decisions.

Cheshire Cat

- How can the representation of the class be fully decoupled from clients?
- Remove everything until there is nothing left but the smile...
  - Fully hidden representation
  - Same definition for different implementations
  - Fewer build dependencies



```
class clock
{
    ...
private:
    struct body;
    body *self;
};
```


Cheshire Cat [Murray1993] is one of the oldest recognised C++ idioms (also known as the *Pimpl* idiom [Sutter2000]). It was found originally in the Glockenspiel class libraries. C++'s model for encapsulation ensures that, from a written source perspective, clients of a class are not – and cannot become – dependent on its internal representation. However, although they cannot access the private section it is still visible in the source code. This can create compilation and binary dependencies: the types of its data members may differ and the `sizeof` an object may change.

In addition, it exposes some of the implementation to the user who may be able to second guess the workings of the class, or gain access by more malicious means (e.g. insert their own `friend` declaration or `#define private public`). As well as being open to such terrorist action, it may be undesirable to distribute the exposed internal workings of a product, e.g. where the same header file is distributed for different releases or platform versions of a product.

The solution is to wrap up an opaque type within a class, using only a pointer to the forward declared representation type. This removes the body from the class definition, leaving only the 'smile'. The full definition of this private type is provided in the relevant implementation source file where the member function definitions can access it.

This idiom for representation hiding works well for concrete classes and supports binary compatibility. If there is further abstraction required, such as the clock representation could be one of many different forms, the encapsulation hides any further generalisation to Bridge [Gamma+1995].

Combined Function

```
class clock
{
public:
    short hours() const;
    short minutes() const;
    clock &hours(short);
    clock &minutes(short);
    ...
};
```

Sequential...

```
class clock
{
public:
    struct time
    {
        short hours;
        short minutes;
    };
    time now() const;
    clock &set(
        short hours,
        short minutes);
    ...
};
```

Concurrent...

Property style programming can make the use of a class awkward and unsafe...

Grouping functions and data together gives a more coherent and efficient view

29

The overriding principle in concurrent operation design is to aim for complete, transactional and stateless operations, i.e. operations that do not rely on sequence or implicit state held between calls that is not actually part of an object's logical state. This means that the emphasis should be in capturing common usage sequences as atomic rather than individual attribute access.

Not only does this make interfaces implicitly safer with respect to concurrency, but it also makes them more self descriptive: rather than being presented with a bucket of attributes, the user is presented with a meaningful vocabulary for using objects through that interface. The result is that it is easier to program to such interfaces than larger and less cohesive kitchen sink interfaces.

The first fragment shows an interface with what might be considered a good primitive interface. The next fragment shows alternative applications of the Combined Function idiom to make it an appropriate design for concurrency; it is also appropriate for distribution and exception safety.

Although the examples do not mix command and query semantics in a single function, such combinations are an inevitable consequence of simplifying and shoring up the safety of concurrent programming. This does not mean that command/query separation is incorrect, just that it is a practice bound to a context, just as Combined Function is. However, unquestioning adherence to a single viewpoint [Meyer1997, Mannion1999] can lead to interfaces that are awkward and unsafe, defeating the original objective.

Whole Value

```
class hour
{
public:
    explicit hour(int);
    int value() const;
private:
    int hour_of_day;
};
class minute {...};
class time
{
public:
    time(hour, minute);
    ...
};
```

```
class clock
{
public:
    time now() const;
    clock &set(
        hour, minute);
    ...
};
...
clock system;
...
time now = system.now();
system.set(
    hour(12),
    minute(30));
```

30

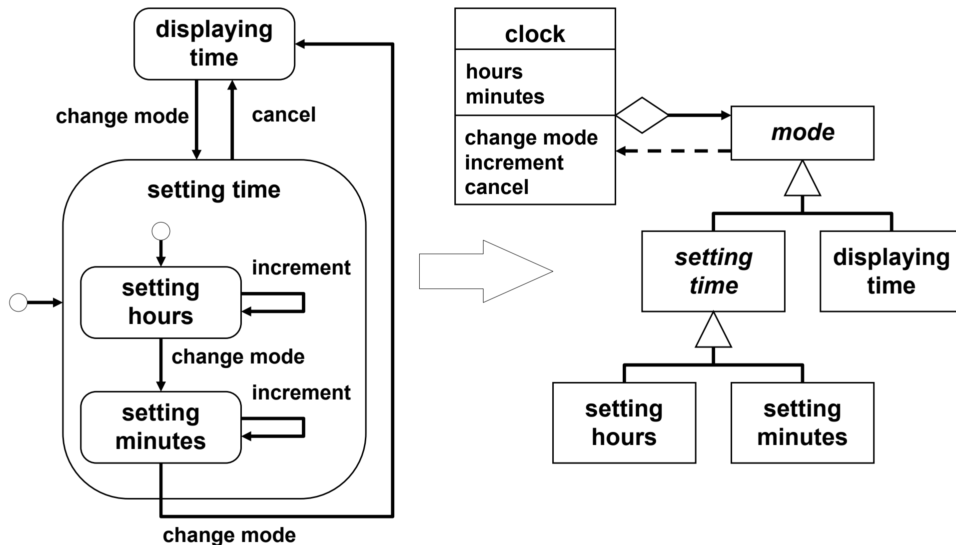
It is tempting, and indeed common, to represent value quantities in a program in the most fundamental units possible, e.g. durations as integers. However, this fails to communicate the understanding of the problem and its quantities into the solution and shows a poor use of the type system.

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. It also provides a location for appropriate range checking to enforce constraints.

In C++ the distinction from fundamental types is further supported by ensuring that there are no implicit conversions. 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. Note that the temptation should be resisted to offer overloads that allow alternative argument orderings: these are not reasonable choices, the interface trying to please all the people all of the time instead of sticking to one clear and cohesive model of use.

A raw Whole Value can be expressed as a using function/constructor style notation, 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 for this purpose using templates [Barton+1994].



Many objects have state on which their behaviour is based. The values of such state can sometimes be grouped into significant modes, each of which corresponds to a different set of behaviours, i.e. methods may behave significantly different in each one. These modes (or states) can be modelled using a variety of notations, including the Harel statechart notation used in UML. The modes and transitions between them constitute an object's lifecycle. Clearly, this approach does not apply to stateless objects.

How should such a lifecycle be implemented for an object? In some cases it is appropriate to take the approach of using a flag to represent the state. However, in all but the simplest cases this leads to a lot of conditional code: functions become dominated by large `switch` or `if` statements. This is error prone and obscure – it becomes hard to add new states, or to comprehend the behaviour in a particular state.

The Objects for States pattern [Gamma+1995], perhaps more commonly – but misleadingly – known as the State pattern, offers a solution based on a direct correspondence between the state model and a class model.

The context object, with which the client communicates, aggregates an object that is used to represent the behaviour in one of its states. Calls on the context are forwarded to the state object; responsibility for implementing behaviour in a particular state is therefore delegated. Transitions between states can be managed either by the behavioural objects themselves, or by a centralised mechanism, such as a table lookup.

Cat Anatomy

```
class clock
{
public:
    void change_mode();
    void increment();
    void cancel();
    ...
private:
    struct body;
    body *self;
};
```

Cheshire Cat has been used to fully factor the representation out of the main class definition...

```
struct clock::body
{
    class mode;
      class displaying_time;
      class setting_time;
        class setting_hours;
        class setting_minutes;

    int  hours, minutes;
    mode *current;
};
```

The definition of the body includes not only the data held for each instance, but also the definition of any types required for the implementation

32

There are many ways of implementing the `mode` hierarchy for an Objects for States configuration. It is just this motivation that suggests whichever route is taken should affect the class user as little as possible. Cheshire Cat has already provided the most appropriate way of hiding the representation of the class, and the Cheshire Cat body also provides the best location to house the declarations of the classes to be used in the `mode` hierarchy: away from the user. Now any changes to the state model will not affect the user.

The fully encapsulated route also means that all the types used for object can access each other, so there is no need for `friend` relationships, as is often the case with C++ implementations of this pattern, e.g. [Gamma+1995].

Another consequence of this approach is the clear separation of an object's aspects into corresponding types:

- *Identity* is represented by the main `clock` class, which sports the main interface through which the user interacts.
- *State* (or, less ambiguously, *data*) is held in `clock::body`.
- *Behaviour* is represented polymorphically within the class hierarchy rooted in `clock::body::mode`.

Thus scope and access are used to reflect the structure of the problem, establishing interfaces on the inside as well as the outside of the main class.

Hidden Interface Class

```
void clock::increment()
{
    self->current->increment(self);
}
```

```
class clock::body::mode
{
    ...
    virtual void increment(body *) = 0;
    ...
};
```

```
void clock::body::setting_hours::increment(body *self)
{
    ++self->hours;
}
```

33

Even though the `mode` hierarchy is already private from the `clock` class, many of the details can be further hidden behind an Interface Class [Carroll+1995] to simplify the view from the main object and expose no more detail than is strictly necessary.

How does an object in the `mode` class hierarchy know about the main object it is supposed to be operating on? There are two basic approaches to this:

- Each `mode` object holds a back pointer to the main object (or rather its body) with which it is associated. This means that each main object needs to either preallocate an object for each `mode`, or allocate each `mode` object when it is required, presumably deallocating it when it is done with it.
- When they are required to execute a function, `mode` objects are passed the body to operate on as an argument. Stateless objects can be shared, which eliminates the need for subtle allocation strategies. The simplest route to sharing is to define a `static` instance per concrete `mode` class. It is tempting to go to town on the design and apply Singleton [Gamma+1995], but in truth this is rarely the right solution; in this case it is definitely overkill.

One remaining issue is how to deal with state transitions. The responsibility can be taken by the main class itself, either explicitly or by using a state transition table. Alternatively, each state determines the next state – again either hardwired or looked up – and either makes the transition itself or returns the suggested next state as a result.

Queue

- Example...
  - A queue class to buffer work between producers and consumers in different threads
- Forces...
  - Thread safety
  - Exception safety
  - Ease of use
  - Evolving requirements

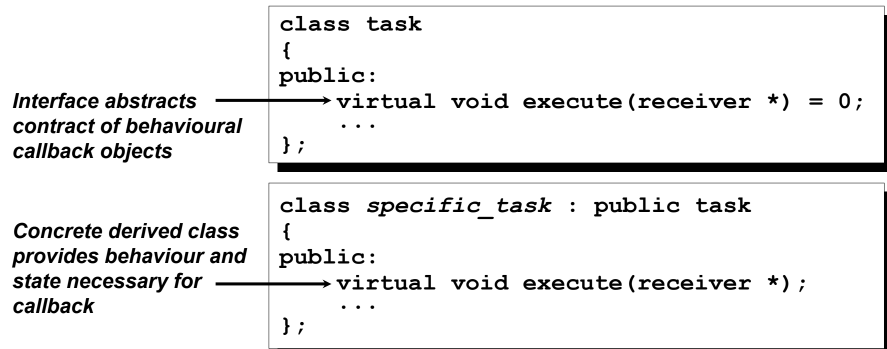
To further demonstrate interface-based principles, consider a queue class. The purpose of the queue is to buffer tasks supplied by one or more producers to consumers that then execute the tasks. All the producers and consumers are executing in different threads, requiring thread safe access of the queue. The control model is push–pull, i.e. the producer pushes tasks into the queue and the consumer pulls them from it.

Some of the design is a matter of refinement, but other changes are the result of shifting requirements intended to demonstrate how design decisions may be taken differently.

The queue is implemented as a concrete class in C++, `queue`. This should not be confused with `std::queue` in the standard, which can be used as part of the underlying implementation.

Command Interface Class

- A request can be encapsulated as an object
  - An interface class provided with the queue is implemented for specific tasks



How can the selection of functionality be decoupled from its execution, so a producer can select and deliver a task to be executed independently by a consumer?

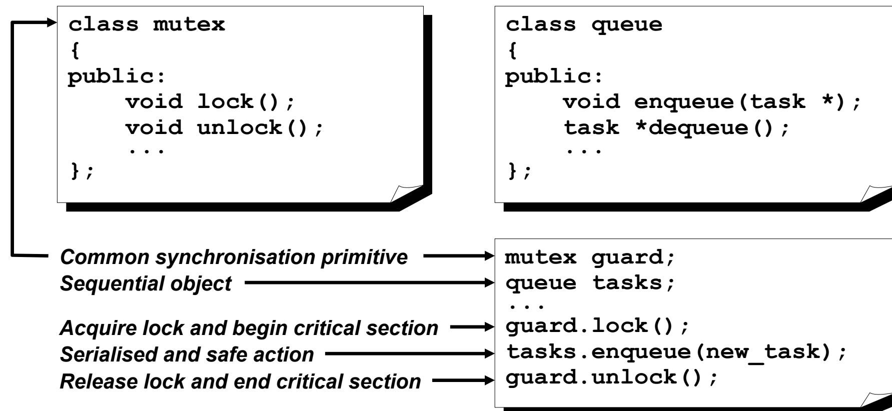
C++ supports a much richer set of features than simply C-style function pointer callbacks, which are rigid and not necessarily very type safe. In this case it is the Command pattern [Gamma+1995] that is appropriate. It explicitly objectifies the concept of method call as object:

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

The task is represented by an Interface Class, `task`, and derived classes realise the detail – data and function – required for particular tasks. This can be generalised further by using Function Objects and Adapters, but that is left as an exercise for the reader.

One issue that must be resolved is that of lifetime and ownership: the producer creates the command object, but who deletes it? Common best practice schemes, such as Creator as Sole Owner [Cargill1996], do not work: this would mean that the consumer had to send back the task once completed. For brevity and simplicity (of exposition, as opposed to final system) it will be assumed that the consumer deletes tasks once they have executed. A refinement to this would be the use of reference counted smart pointers [Boost, Coplien1992].

Basic Locking



In the presence of multiple threads, synchronisation of access to data is vital. A number of primitives are typically available on a threading system. These primitives are best wrapped up in classes, rather than used in their raw API form.

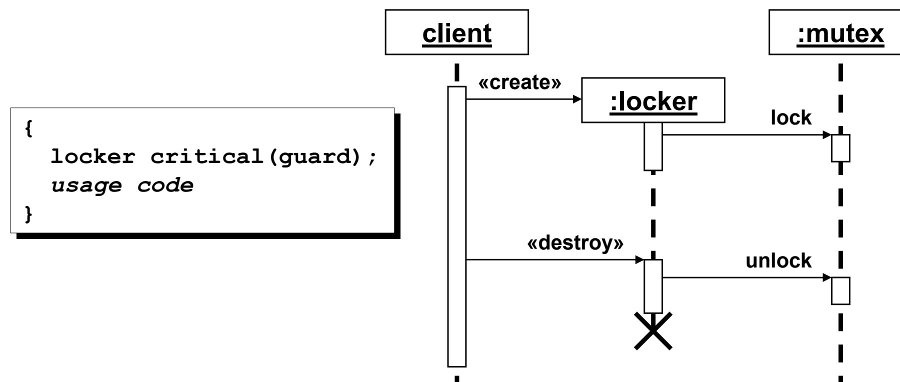
Mutexes and semaphores may be locked and unlocked, bracing a section of code termed the *critical section*. The execution of this code must appear to be executed atomically with respect to other threads, ie. although it may technically be pre-empted, it is not re-entered until completed by a given thread. This term is not to be confused with the Win32 `CRITICAL_SECTION`, which is simply a degenerate form of mutex.

Mutexes offer mutual exclusion in terms of a single thread, ie. the thread that locks it must be the thread that unlocks it. Binary semaphores are not quite so structured. It is system dependent as to whether or not a mutex is re-entrant, i.e. whether or not a locked mutex may be relocked by the locker. Some systems, such as Solaris, offer locks that allow multiple-reader/single-writer access. These are more structured than the other primitives. They may be faked up using condition variables or events and mutexes, or pairs of counted semaphores. Win32 events are effectively "smart blocking flags". Condition variables are like events, but they auto-lock a mutex.

The scope or reach of these primitives may be within a single process address space, or may be system wide for synchronisation across processes.

Execute Around Object

- Places control with a helper object
  - Lifetime of helper encloses usage

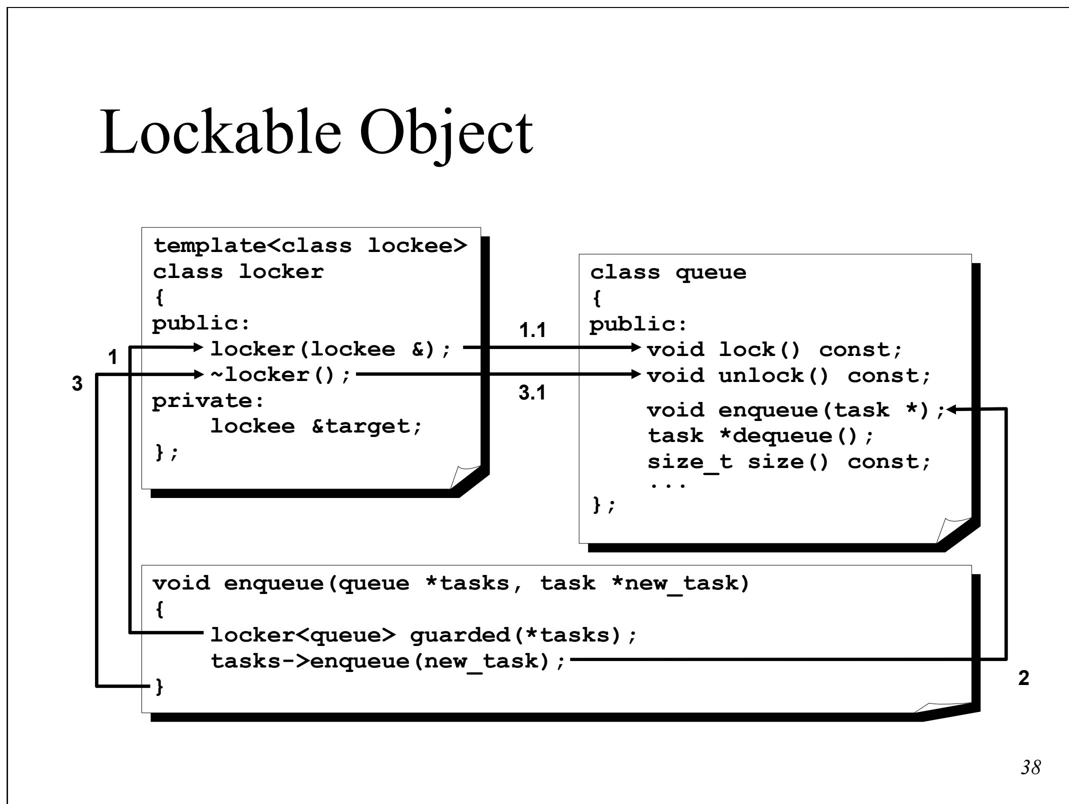


In C++ a constructor is called on creation of an object for the sole purpose of initialising it, i.e. it describes the "boot sequence" for an object. Conversely a destructor is automatically called at the end of an object's life to finalise or clean it up, i.e. to shut it down in an orderly fashion. Importantly, in C++ the calling of a destructor is deterministic: the life of a local stack variable is tied to its enclosing scope; the life of an object allocated dynamically using the `new` operator comes to an end by an explicit call to `delete`.

The former property, that of tying lifetime to scope, is what underpins the Execute Around Object pattern. A helper object is declared with a reference to the target object whose member functions must be paired around use. The helper object calls back in to the target object on creation to acquire or initialise the resource. In its destructor it automatically calls back to release or finalise the target's resource.

Such helper objects are good examples of the fine-grained helper objects that can simplify an overall implementation, as opposed to the coarse-grain abstractions apparent in the business model or user interface of a system.

The Execute Around Object pattern is found at the heart of the misnamed Resource Acquisition is Initialization idiom [Stroustrup1997]; misnamed because the essence of what makes this pattern work is the destructor. In many cases, resource acquisition occurs independently of the Execute Around Object, as in the case of memory acquisition. Perhaps *Resource Release is Finalization* would be a better name for the resource-based applications of Execute Around Object.



The control structure pairing can be generalised to apply to the target resource object itself. The `queue` class now supports its own `lock` and `unlock` functions, making it more cohesive.

The appropriate generalisation for this is to use templates rather than tying the classes together in a more committed inheritance hierarchy:

```
template<class lockee>
locker<lockee>::locker(lockee &to_lock)
  : target(to_lock)
{
    target.lock();
}

template<class lockee>
locker<lockee>::~locker()
{
    target.unlock();
}
```

Locking Function Adapter

```
locked(tasks)->enqueue(new_task);
```

Simplified expression- and statement-level locking

```
template<class lockee>
tmp_locker<lockee> locked(lockee *target)
{
    return tmp_locker<lockee>(target);
}
```

Locking is handled by returning proxy

```
template<class lockee>
class tmp_locker
{
public:
    explicit tmp_locker(lockee *);
    ~tmp_locker();
    lockee *operator->();
private:
    lockee *target;
    bool   is_locked;
};
```

Set up target but do not lock yet

Unlock if locked

Lock when target dereferenced and used through member access operator

Where single function calls are to be locked, the use of a temporary Execute Around Object is clumsy. It is possible to offer a function that performs the locking and unlocking for a single function call. In the example shown a function, named `locked`, is responsible for acquiring a lock from its argument and returning it so that it may now be dereferenced safely. However, the usage is outside the execution of `locked` so how is the resource unlocked? The solution returns a proxy from `locked` rather than a raw pointer:

```
template<class lockee>
class tmp_locker
{
public:
    explicit tmp_locker(lockee *to_lock)
      : target(to_lock), is_locked(false) {}
    ~tmp_locker()
    {
        if(is_locked)
            target->unlock();
    }
    lockee *operator->()
    {
        target->lock();
        is_locked = true;
        return target;
    }
private:
    lockee *target;
    bool   is_locked;
};
```

Execute Around Pointer

Smart pointers provide the simplest approach for users

```
locking_ptr<queue> ptr(tasks);
...
ptr->enqueue(new_task);
```

Member access operator returns locking proxy – calls to member access operator are chained

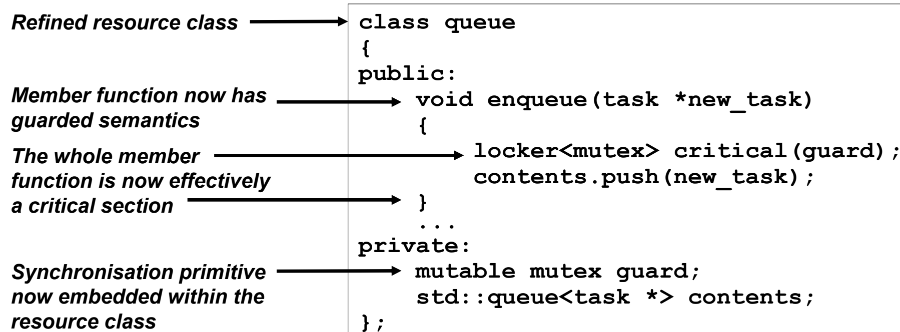
```
template<class lockee>
class locking_ptr
{
public:
    tmp_locker<lockee> operator->() const
    {
        return tmp_locker<lockee>(target);
    }
    ...
private:
    lockee *target;
};
```


However, although improved in convenience, the locking function is hardly transparent. A smart pointer can be used to build on the temporary acquisition concept. In the example shown, `locking_ptr` overloads `operator->` to return a `tmp_locker` that also overloads `operator->` and performs the actual locking. This idiom works because calls to `operator->` are automatically chained by the compiler until a raw pointer type is returned. Note that one consequence of the language and this design is that `operator*` is not meaningfully supported.

The locking behaviour can easily be disabled for single threaded applications, requiring only modifications to `tmp_locker` to make it a pass-through object, a recompile, and a relink.

Programmers should be careful about attempting to access the same object twice in a statement using `locking_ptr`s: this will cause deadlock if the synchronisation mechanism is not re-entrant.

Self-Locking Function



If single function calls – i.e. single calls to enqueue or dequeue – are the norm, an alternative approach simplifies the interface considerably from the perspective of the user. The affordances and constraints are brought more comfortably into line, making the interface more encapsulated (with respect to usage) and safer.

For objects that are known to be shared, and where operations are normally used in isolation – rather than typically being used for a sequence of operations – it often makes sense to provide the locking as part of the automatic behaviour, i.e. when a public function is called it locks an internal synchronisation object. Objects of such a class are said to be *monitors* or, in Ada 95 parlance, they are *protected*. In Java the same facility is implemented using `synchronized` methods.

Classes with internally locked `public` functions simplify programming from the class user's point of view. The class author must be careful not to call other public member functions from within the class if the synchronisation object is not re-entrant. Similarly, the class author must respect the fact that C++ supports class level rather than object level encapsulation; it is possible for an object to access the private members of another object of the same class and accidentally bypass the synchronised interface ordinary class users would use.

A typical locked function will have the object locked for the whole scope of the function. However, this need not always be the case and the lock scope should be as small as possible, i.e. any local variables should be declared and initialised before the section, and any further calculations and returns after.

Batch Function

- How can multiple tasks be enqueued or dequeued together without interruption?
  - Provide a self locking function that operates on sequences rather than just a single item

```
class queue
{
public:
    template<typename task_iterator>
      void enqueue(task_iterator begin, task_iterator end);
    template<typename task_iterator>
      void dequeue(task_iterator begin, task_iterator end);
    ...
};
```


Internally locked functions support thread safe single calls, but there is no guarantee that multiple calls from one thread will not be interleaved with calls from other threads. There is also the issue that repeated locking and unlocking of an object is costly.

If common usage of the queue requires that many tasks should be enqueued and dequeued atomically, the self locking model can be extended to cope with this. Additionally providing a Batch Function that handles multiple items effectively provides a flattening of a loop into a repeated data structure, and ensures that execution will be correctly locked.

This presents a safe and cohesive interface. It can be decoupled from the actual representation of the sequence by using the Iterator Range idiom found in STL. For `enqueue` the iterators are required to be at least Input Iterators. For `dequeue` they are required to be at least Output Iterators.

This means that not only is the user free to use their own chosen containers, but that no inclusion of container details is necessary in the `queue` header file, keeping the physical aspect of the interface clean.

Composite Command

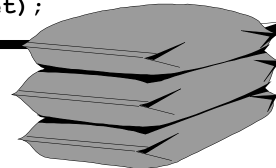
Composite command class allows many commands to be grouped together and treated as one

```
class composite_task : public task
{
public:
    virtual void execute(receiver *);
    ...
private:
    vector<task *> sequence;
};
```

Insert individual commands into sequence

Individual commands executed together in sequence

```
task *new_task =
    new composite_task(begin, end);
...
tasks.enqueue(new_task);
...
task *to_do = tasks.dequeue();
to_do->execute(target);
```



One issue that internally locked interfaces raise is how to group multiple actions together without interleaving. For instance, given the `queue` class shown previously, how can a sequence of `task` objects that must be executed together be placed in a queue? A Batch Function allows multiple enqueuing and dequeuing, but the number dequeued together need not be the number enqueued together.

Multiple actions that need to be grouped together, either as a result of scripting or for transactional reasons, can be implemented using the Composite pattern [Gamma+1995] with the Command pattern. Composite is a structural pattern that deals with the issue of recursive composition of Whole–Part hierarchies [Buschmann+1996] in which all components are treated uniformly. In this example it means that a composite command may contain any command objects, including other composite commands.

The most common composite command would be a sequence, although a concurrent construct is possible where a thread is launched for each command.

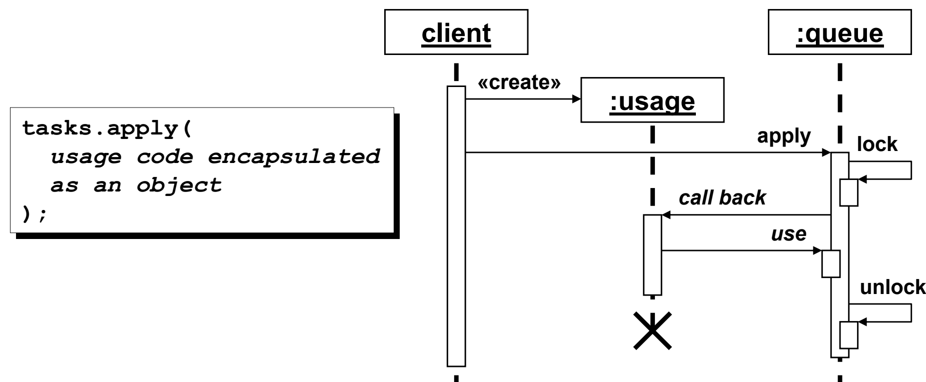
The detail of the execution for a sequence would be as follows:

```
void composite_task::execute(receiver *target)
{
    for(size_t index = 0; index < sequence.size(); ++index)
        sequence[index]->execute(target);
}
```

This can be made more idiomatic by applying standard algorithms and function objects.

Execute Around Function

- Places control within the resource
  - Usage is passed to the resource for execution



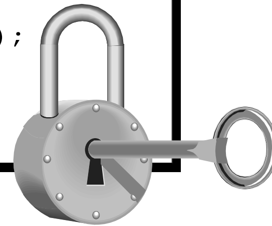
Execute Around Function (also known as Execute Around Method [Beck1997]) offers an alternative to the external management of a resource as well as a more sophisticated view of internal management. The code that must be enclosed by the two calls to the resource is itself passed to the resource itself for execution: The code may be encapsulated as a Command object [Gamma+1995]. The resource then executes the necessary actions before and after calling the code itself. This guarantees exception safety around resource-based tasks and atomicity of grouped operations.

There is a strong similarity with both Template Method and the double dispatch of Visitor [Gamma+1995] in this pattern. Interestingly, the control and object structure is effectively the inverse of that in Execute Around Object.

Function Object

- How can the usage code be passed in?
  - Represent usage as a function object

```
class queue
{
public:
    template<typename unary_function>
    void apply(unary_function callback)
    {
        locker<queue> critical(guard);
        callback(this);
    }
    ...
};
```



A specialisation of the Command pattern in C++ is the Function Object or Functor idiom [Coplien1992], where an object's type supports function call syntax, i.e. overloading `operator()`. The style of generic programming makes use of operator overloading and templates to make the distinction between use of a function object or function pointer transparent:

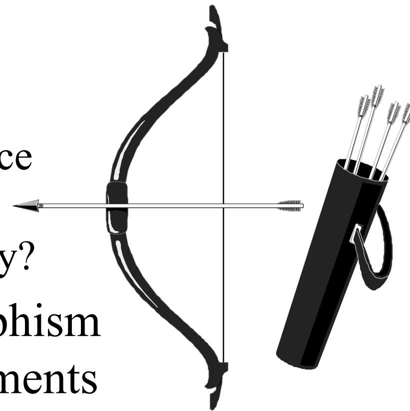
```
void nibble(cookie &);
pointer_to_unary_function<cookie &, void> nibbler(nibble);
...
container<cookie> jar;
jar.apply(nibble);
jar.apply(nibbler);
```

When implementing an Execute Around Function in C++ there is often a case for overloading the function with respect to `const`-ness:

```
class queue
{
public:
    template<typename unary_function>
      void apply(unary_function callback);
    template<typename function>
      void apply(unary_function callback) const;
    ...
};
```

Callback Target

- What should the callback object actually call back on?
  - Should it be on publicly available functions?
  - If not, should the interface be privately inherited or detached and on the body?
- Dependent on polymorphism and decoupling requirements



46

There is still a question as to what the callback target should be. In the previous example it was assumed that the current object would be the target. But does this mean that the interface the callback occurs on is public to other users? If this is the case, it means that in theory any public user can execute critical functions and avoid locking. If, on the other hand, all such functions are self locking, there are efficiency and deadlock considerations when the callback executes them, effectively relocking an object that is already locked.

An alternative is to publish the usable interface as a separate Interface Class, and privately inherit it. This means that the `this` pointer will be correctly converted to the private base class on the callback, but that no public user can use it directly. The constraints on this are that, because of runtime polymorphism, member templates cannot be used for any of the functions. It also means that there may be name clashes if the class also publishes self locking functions for common usage.

Yet another alternative is to introduce a Handle/Body split [Coplien1992, Gamma+1995] and have the callback interface on the body. This introduces a decoupling that was not there before, but also means that there is more object management involved than before.

Summary

- Interfaces represents connection vocabulary and units of obligation in a system
  - Interfaces establish terms of reference
  - Sometimes a truer expression of intent and understanding than classes
- Interfaces play a part in system structure
  - Identification and preservation of constraints
  - Management and reduction of dependencies

A focus on interfaces is implicit in many approaches to software development, whether function-based APIs, object orientation, component-based development, or distributed computing. However, understanding interface-based development (IBD) as a concept in its own right, making it an explicit approach demonstrates why many attempts at these other styles fall at the first hurdle: reusability, plug and play components, etc all remain mythical creatures until IBD is one of a number of key pillars in place in a development culture.

Interfaces are first class citizens in development, not an afterthought to be tacked onto an implementation. With interfaces comes the concept of aggressive dependency management, identification of constraints and their communication and preservation through affordances. Constraints and dependencies; semantics and connections; meaning and structure – these represent the distillation of essential design practice, and interfaces are part of the mix.

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

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

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

[Beck2000] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

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

[Box+1999] Don Box, Keith Brown, Tim Ewald and Chris Sells, *Effective COM: 50 Ways to Improve Your COM & MTS-based Applications*, Addison-Wesley, 1999.

[Brand1994] Stewart Brand, *How Buildings Learn: What Happens After They're Built*, Phoenix, 1994.

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

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

[Cargill1996] Tom Cargill, "Localized Ownership: Managing Dynamic Objects in C++", [PLoP1996].

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

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

[Coplien1995] James O Coplien, "A Generative Development-Process Pattern Language", [PLoPD1995].

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

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

[Dai+1999] Ping Dai, Ray Farmer and Alan O'Callaghan, "Patterns for Change", *EuroPLoP '99*, 1999.

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

[Dyson1998] Paul Dyson, "Patterns in Software Architecture", *Patterns '98*, February 1998.

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

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

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

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

[Jackson1995] Michael Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*, Addison-Wesley, 1995.

[Lakos1996] John Lakos, *Large-Scale C++ Software Design*, Addison-Wesley, 1996.

[Lea1998] Doug Lea, "Christopher Alexander: An Introduction for Object-Oriented Designers", [Rising1998].

[Liskov1987] Barbara Liskov, "Data Abstraction and Hierarchy", *OOPSLA '87 Addendum to the Proceedings*, October 1987.

[Mannion1999] Mike Mannion, "Concurrent Contracts: Design by Contract™ and Concurrency in Java", *Java Report*, SIGS, May 1999.

[Martin1995] Robert Martin, "Object-Oriented Design Quality Metrics: An Analysis of Dependencies", *ROAD*, SIGS, September-October 1995.

[Meyer1997] Bertrand Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.

[Meyers2000] Scott Meyers, "How Non-Member Functions Improve Encapsulation", *C/C++ Users Journal*, February 2000.

[Murray1993] Robert B Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993.

[Norman1989] Donald A Norman, *The Design of Everyday Things*, paperback edition, MIT Press, 1988.

[Pawson1996] John Pawson, *Minimum*, Phaidon, 1996.

[Petroski1992] Henry Petroski, *To Engineer is Human: The Role of Failure in Successful Design*, Vintage, 1992.

[PLoP1995] Edited by James O Coplien and Douglas C Schmidt, *Pattern Languages of Program Design*, Addison-Wesley, 1995.

[PLoP1996] Edited by John Vlissides, James O Coplien and Norman L Kerth, *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.

[Rising1998] Linda Rising, *The Patterns Handbook: Techniques, Strategies and Applications*, Cambridge University Press, 1998.

[Stroustrup1997] Bjarne Stroustrup, *C++ Programming Language*, 3rd edition, Addison-Wesley, 1997.

[Strunk+1979] William Strunk Jr and E B White, *Elements of Style*, 3rd edition, Macmillan, 1979.

[Sutter2000] Herb Sutter, *Exceptional C++*, Addison-Wesley, 2000.

[Szyperski1998] Clemens Szyperski, *Component Software*, Addison-Wesley, 1998.

[Taligent1994] *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*, Addison-Wesley, 1994.