Design

Concepts and Practices

Kevlin Henney

QA Training

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

Keynote presentation at *JaCC*, Oxford, 18th September 1999.

Kevlin Henney

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

QA Training

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

Overview

- Concepts
  - What concepts underpin design?
- Practices
  - Systems reflect principles and practices used in their construction

"The time has come," the guru said,
"To talk of many things:
Of use – and cases – and object models –
Of processes – and strings –
And why notations unify –
And if we need such things."

Design is often talked about these days, but there are few adequate definitions of what it really is and how it relates to other parts of development. Is it simply a stage that adds detail to a model, sitting between analysis and implementation? Is it the production of UML diagrams with programming detail? Is it fully dependent or independent of implementation technology?

This talk takes the view that design is a generative process that takes in many levels of detail, and balances forces in the problem, solution and lifecycle to create a practical and working system from an understanding of requirements. By asking and answering key questions, based on direct experience and patterns, the constraints found in requirements can be expressed and dependencies between parts of a system managed. Design takes in the range of detail from analysis and broad architecture to the fine granularity of programming languages, and desirable properties include modularity, modelarity and minimalism.

Concepts

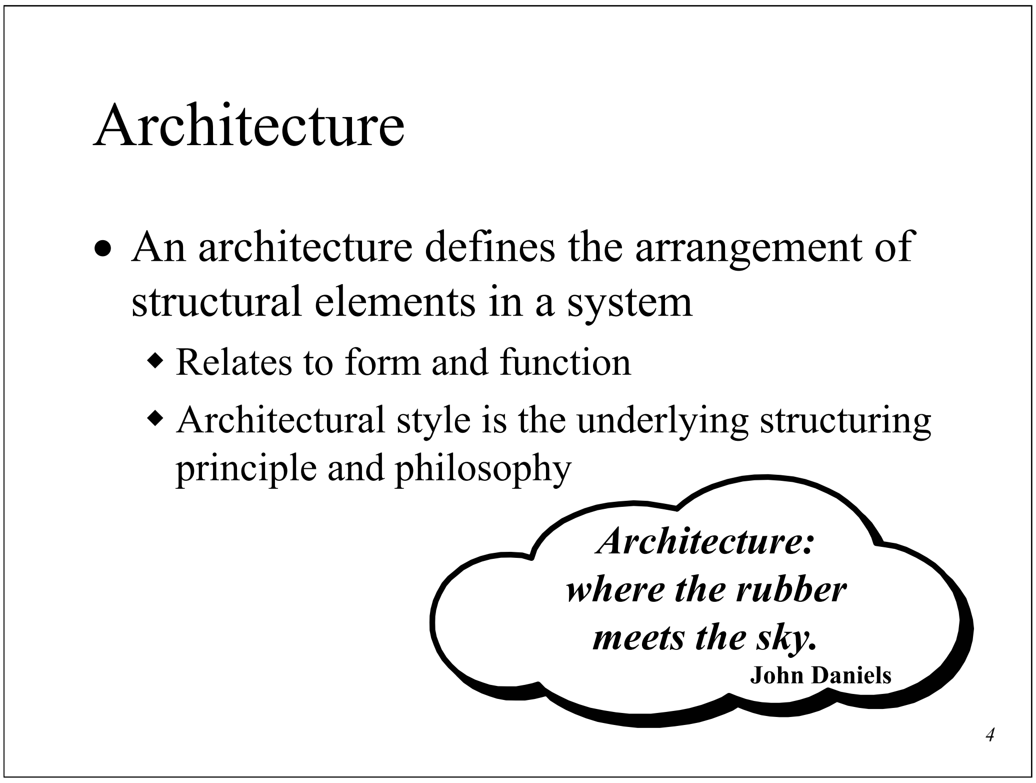
We think in generalities, but we live in detail.
Alfred North Whitehead

Software development been likened to many things – architecture, art, craft, engineering, manufacturing, maths, music, writing, etc – and rather than exclusively considering it as being one or other of these, it is perhaps more useful to explore it as a discipline in its own right. This implies neither isolation from other disciplines nor simple mimicry. Software has unique characteristics, which will differentiate its development from other forms of creation or manufacture, but it also shares attributes with other disciplines, which suggests that we can learn *from* them rather than attempting to *be* them.

Software developers seeking inspiration from other domains often forget that other domains overlap in their interests and concerns. A focus on software as engineering often neglects mention of any influence or attribute outside what is considered "core syllabus" engineering, and to its detriment. In practice we can see that something as robust and traditional as structural engineering does not confine itself to such a definition ([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 design of software systems. In software, the principles and practices of design sometimes seem to be as flexible as the medium of software itself... and sometimes just as rigid.



Architecture

- An architecture defines the arrangement of structural elements in a system
  - Relates to form and function
  - Architectural style is the underlying structuring principle and philosophy

Architecture: where the rubber meets the sky.
John Daniels

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?*

Booch [Booch+1999] describes architecture broadly:

The set of significant decisions about the organization of a software system, the selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization—these elements and their interfaces, collaborations, and their composition. Software architecture is not only concerned with structure and behaviour, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.

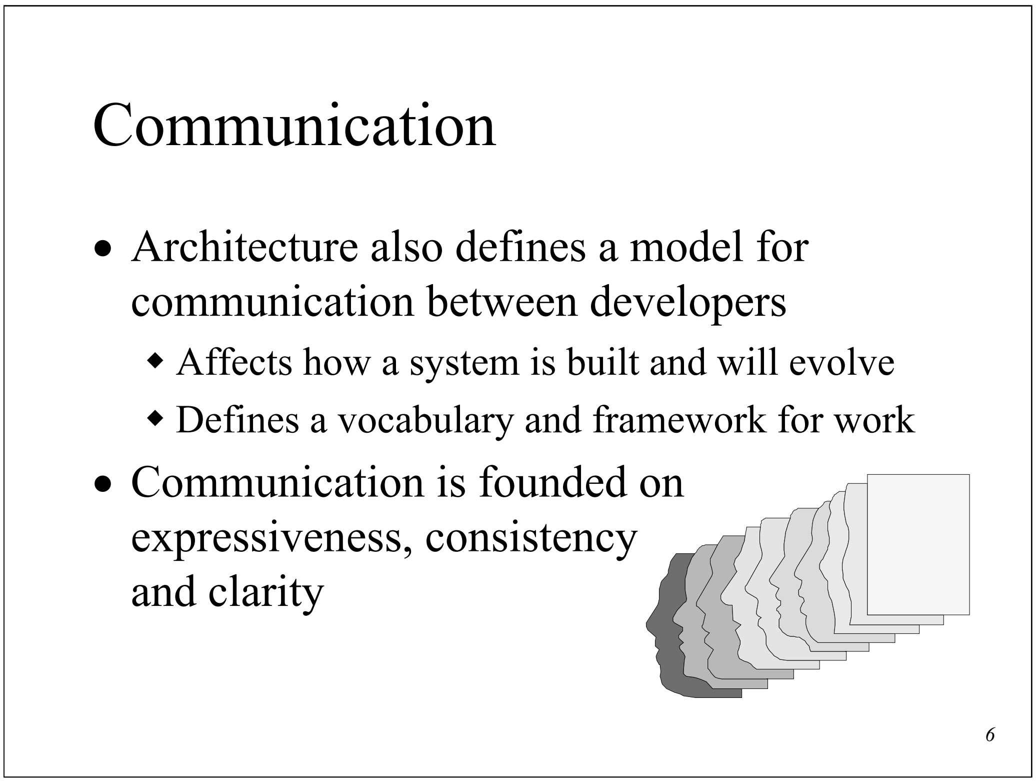
Architecture may be considered the result of a conscious design process, or simply a posthoc description of a system configuration. Thus, in spite of its etymology (*architect* is derived from the ancient Greek for *chief builder*), architecture can be accidental rather than intentional.

Structure

- Architecture defines the form and characteristics of a system
  - Properties include organisational structure and development process...
  - As well as the more visible features normally associated with structure

The architecture of a system will determine to a great degree how it is built in terms of both its components and connections, 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.

The role of people in the development process as related to the architecture has also been recognised to some extent in the definitions of modern multi-tier architectures, e.g. Enterprise JavaBeans and the CORBA Component Architecture, where there is correspondence between development roles and the architecture. These roles are readily identifiable in any CBD system [Szyperski1998]. More generally they can be seen in the organisational patterns that surround development [Coplien1995]. The architectural model and process is reflected in developer knowledge and experience; mismatches between these two models cause fault lines in a system and its development.



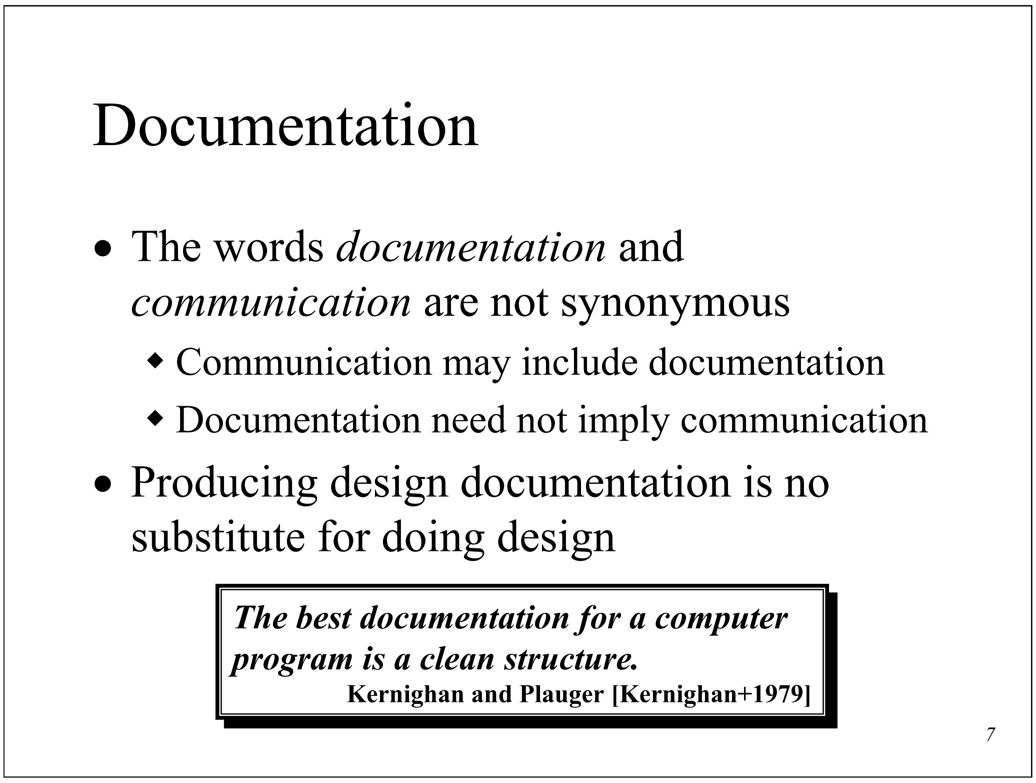
Communication

- Architecture also defines a model for communication between developers
  - Affects how a system is built and will evolve
  - Defines a vocabulary and framework for work
- Communication is founded on expressiveness, consistency and clarity

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.

Good communication is underpinned by requirements of expressiveness, consistency and clarity. These apply at all levels of a system, for instance: names are important, whether of operations or whole subsystems; a practice should communicate the same thing all the time, e.g. C++ pure `virtual` functions communicate "no implementation" and therefore should not be provided with a default implementation, knowledge of which breaks with encapsulation; clarity is its own virtue [Strunk+1979]:

> *[Reminder] 16. Be clear*
>
> *Clarity is not the prize in writing, nor is it always the principal mark of a good style. There are occasions when obscurity serves a literary yearning, if not a literary purpose, and there are writers whose mien is more overcast than clear. But since writing is communication, clarity can only be a virtue. And although there is no substitute for merit in writing, clarity comes closest to being one. Even to a writer who is being intentionally obscure or wild of tongue we can say, "Be obscure clearly! Be wild of tongue in a way we can understand!"*



Documentation

- The words *documentation* and *communication* are not synonymous
  - Communication may include documentation
  - Documentation need not imply communication
- Producing design documentation is no substitute for doing design

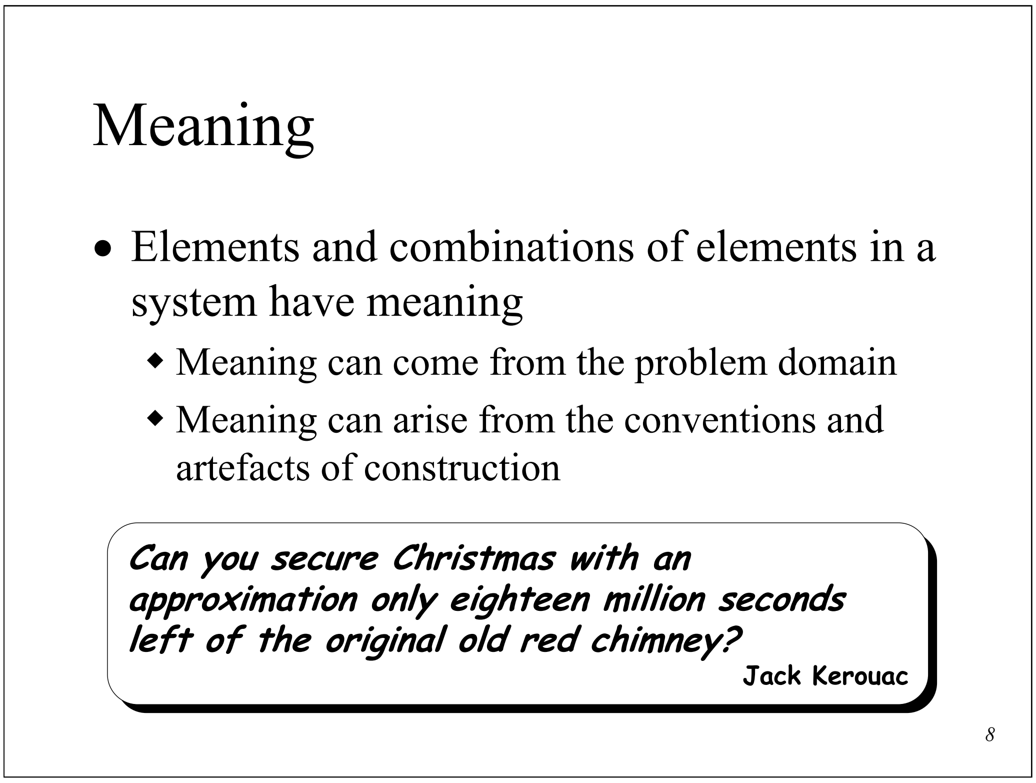
The best documentation for a computer program is a clean structure.
Kernighan and Plauger [Kernighan+1979]

Documentation at all levels can take on a cargo cult status in many organisations, replacing the main activities of development. However, it is worth keeping the following in mind [Kernighan+1979]:

1. *If a program is incorrect, it matters little what the documentation says.*
2. *If documentation does not agree with the code, it is not worth much.*
3. *Consequently, code must largely document itself. If it cannot, rewrite the code rather than increase the supplementary documentation. Good code needs fewer comments than bad code does.*
4. *Comments should provide additional information that is not readily obtainable from the code itself. They should never parrot the code.*
5. *Mnemonic variable names and labels, and a layout that emphasizes logical structure, help make a program self-documenting.*

Is this to say that documentation, in the form of comments, linked web pages, treeware, etc, is necessarily a bad thing? No, quite the opposite in fact: the right documentation is worth its weight in gold; pursuit of documentation by dogma generates too much of a "good thing", and it becomes fools gold.

If documentation is considered to have a linear quantity/quality relationship (where "some documentation is good" is extrapolated to "lots of documentation is very good"), the systems clouds over rather than becoming clearer. Those who work with the system become surrounded by noise – "when everything is a triangulation point, nothing is a triangulation point" [Jackson1995]. Note that the converse is also true, if such advice is misinterpreted: absence of documentation does not equate to presence of quality – "when nothing is a triangulation point, nothing is a triangulation point".



Meaning

- Elements and combinations of elements in a system have meaning
  - Meaning can come from the problem domain
  - Meaning can arise from the conventions and artefacts of construction

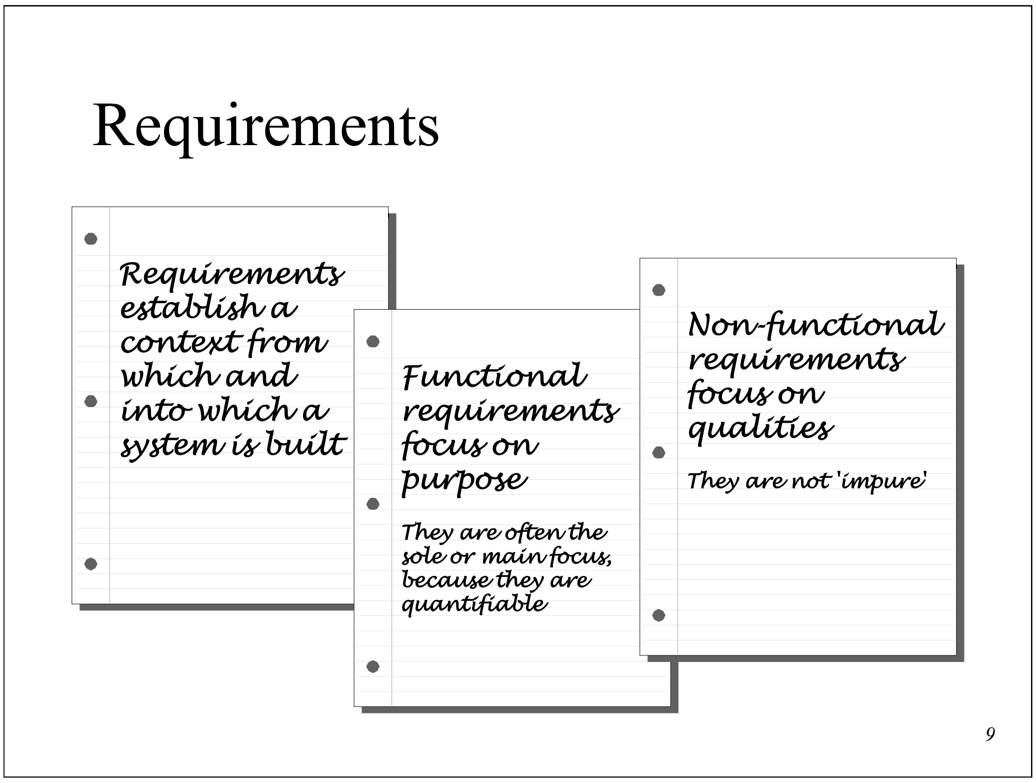
Can you secure Christmas with an approximation only eighteen million seconds left of the original old red chimney?
Jack Kerouac

It is possible to use mechanisms of a programming or modelling language in a particular way so that a system has the appearance of being well formed but, like the Jack Kerouac quote above and Noam Chomsky's famous grammatically well formed nonsense sentence "colorless green ideas sleep furiously", this is not sufficient to guarantee a useful and meaningful system.

The meaning of elements in a system in part comes from the problem domain, where the stuff of the system can be described by behaviour and relationships. This constitutes one source of vocabulary in a system's development. A customer whose focus is the problem domain would expect to be able to talk at to some extent about the construction and behaviour of the system in those terms. Constraints and features in the problem domain should be preserved as the conception of the system goes from the abstract, e.g. use of UML diagrams at varying levels of detail, to code.

Another source of meaning in a system is the use of mechanisms in the solution domain, i.e. implementation technology. Meaning can be deduced, i.e. the meaning of an abstraction is made explicit as in the use of `interface` in Java, or inductive, e.g. using pure abstract classes in C++ to denote interfaces.

The meaning of the system can be seen in the code and in the people that work with the code [O'Callaghan+1996]. Where does documentation fit in? By trying to put as much meaning as possible into the system structure itself, every additional piece of documentation will count. This avoids both meaningless documentation mountains and barren documentation deserts. The latter often occur because of document aversion felt by those who have had to climb document mountains where the summit is lost and shrouded in cloud.



Requirements

Requirements establish a context from which and into which a system is built

Functional requirements focus on purpose

They are often the sole or main focus, because they are quantifiable

Non-functional requirements focus on qualities

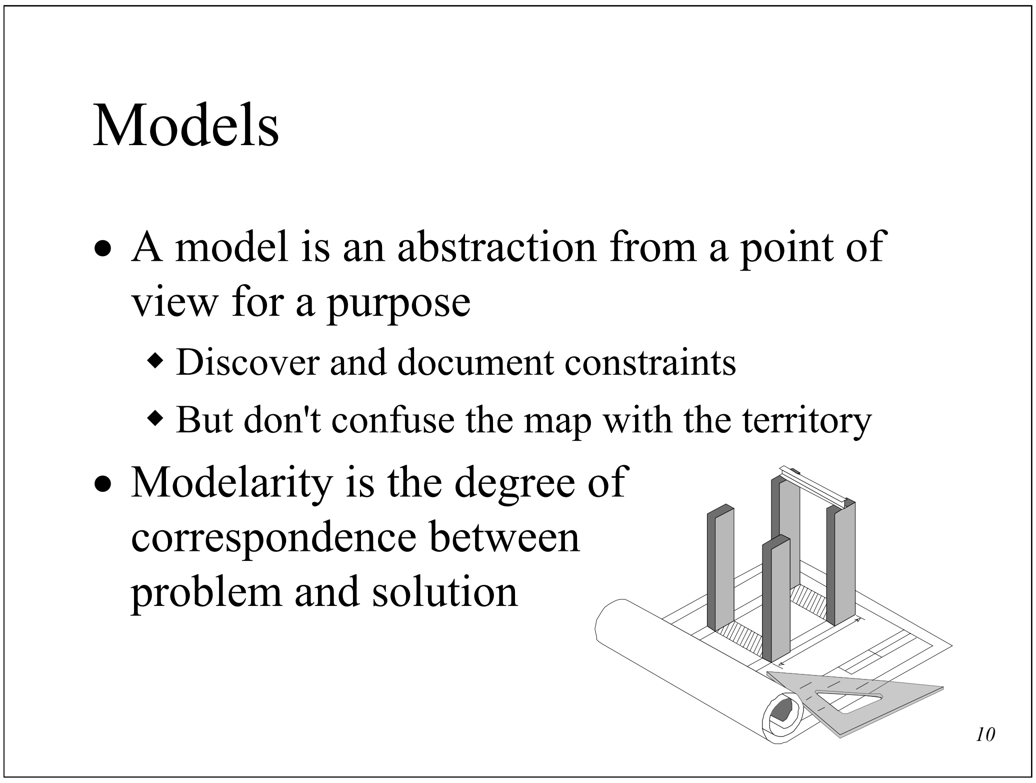
They are not 'impure'

The correspondence between structure and purpose is an important one for any architecture [Shaw+1996]:

The architecture of a software system defines that system in terms of computational components and interactions among those components.... In addition to specifying the structure and topology of a system, the architecture shows the correspondence between the system requirements and elements of the constructed system.

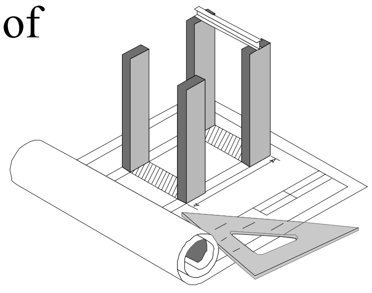
Most development methods concern themselves with the functional requirements of a system, i.e. what the system must do. Such requirements offer a high degree of traceability through the lifecycle. However, it is often the case that there are a number of non-functional requirements that are as important to a system, which include quality of service, failure strategies, use of specific technologies, etc. These can be harder to quantify and test for, but are nonetheless of great importance. For instance, the non-functional behaviour of a distributed system cannot merely be dismissed as an implementation detail.

Perhaps it is because non-functional requirements cannot be quantified as easily as functional requirements that they are at once both important and ignored.



Models

- A model is an abstraction from a point of view for a purpose
  - Discover and document constraints
  - But don't confuse the map with the territory
- Modelarity is the degree of correspondence between problem and solution



So what exactly is a model? There are many definitions, of which the following captures a lot of what is commonly understood in software [Rumbaugh+1999]:

> *A model is a more or less complete abstraction of a system from a particular viewpoint. It is complete in the sense that it fully describes the system or entity, at the chosen level of precision and viewpoint.*

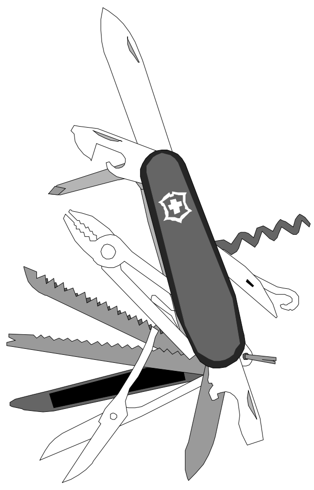
[Jackson1995] goes further in distinguishing between different definitions of model, and what the purpose of a model is. Importantly he draws a distinction between the real world, the model that is built of it, and the machine (i.e. the software) that is built from that. An important observation being that there are properties of the real world that are not true of the model, and vice-versa, and the same between model and machine. We can go a stage further and separate models of the problem domain from models of the solution.

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 the components of the problem being modelled and those in its solution. At the same time, one must not mistake the map for the territory.

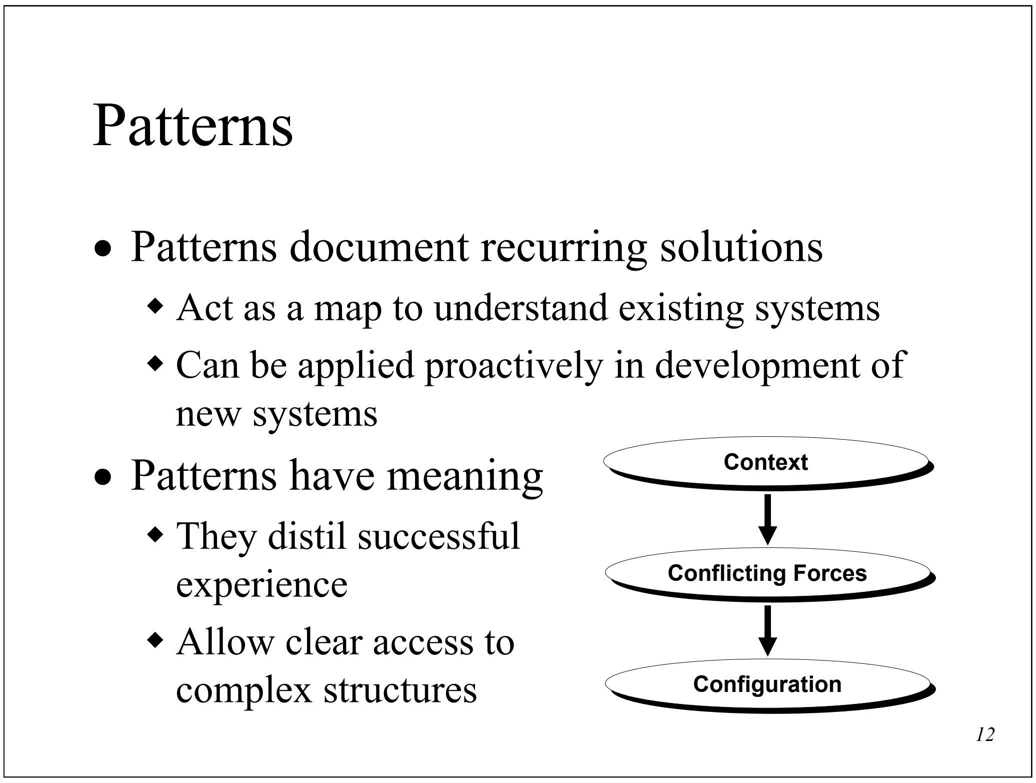
Simplicity and Complexity

- Essential complexity versus actual complexity
- *Simple* is not the same as *simplistic*
  - *Simple* implies a close fit between essential and actual complexity
  - *Simplistic* implies ignoring constraints and requirements



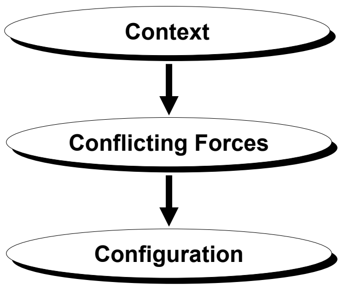
The sources of complexity in a system come from the problem domain (i.e. the essential or inherent complexity) and the solution domain (i.e. the structure built) [Henney1999]:

> *Software development is about many things, one of which is the management of complexity. It is a lesson that has been oft repeated, but apparently not as often heeded. The inherent complexity of a software system is something that we can do little about; we cannot eliminate it, but we can hide and abstract it. We can also create complexity.*
>
> *The fact that software systems are getting bigger and more pervasive is on the one hand testimony to the fact that we can at times muster the wherewithal to manage the inherent complexity of constructing large systems. At other times, it supports the view that creating complexity is easier than hiding it: the creeping featurism, physical size, and bugginess of certain operating systems and word processing packages is tantamount to a public admission of software engineering failure – "the software to solve your problem is larger and more complex than necessary, because we did not have the ability or resources to make it smaller and simpler", to liberally misquote Blaise Pascal.*
>
> *The inherent complexity of a software system is related to the problem it is trying to solve; the actual complexity of a software system is related to the size and structure of the software system as built; the difference is a measure of our inability to match the solution to the problem.*



Patterns

- Patterns document recurring solutions
  - Act as a map to understand existing systems
  - Can be applied proactively in development of new systems
- Patterns have meaning
  - They distil successful experience
  - Allow clear access to complex structures



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

The seminal design patterns book, *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma+1995], has done more perhaps than any other to popularise the use of patterns in OO design, bridging the gap from the origin of patterns in the built environment [Alexander+1977, Alexander1979] to software architecture. There is, however, more to software patterns than just this, as the PLoPD books [PLoP1995, PLoP1996, PLoP1998], [Coplien1996], [Buschmann+1996], surfing [Hillside], or browsing any of a number of the object journals (*JOOP*, *C++ Report*, *Java Report*, etc) will reveal.

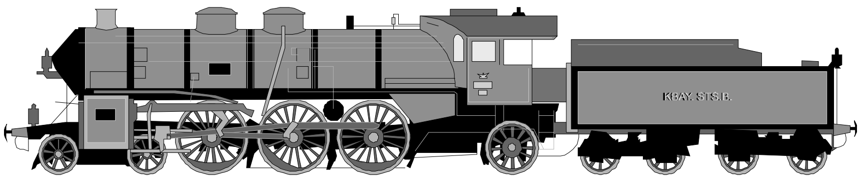
In constructing a system, catalogues of patterns can play a role, as can individual patterns. However, a pattern language [Alexander+1977] defines a more complete approach to connecting patterns together in a generative fashion for a particular purpose [Coplien1996]:

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

An idea related to patterns and generative solutions is that of problem frames and multiframe problems [Jackson1995].

Partitioning

- Quality and qualities of separation
  - Coupling describes interconnectedness
  - Cohesion describes intraconnectedness
- Separation introduces connections
  - Connections can be directional or cyclic



13

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]).

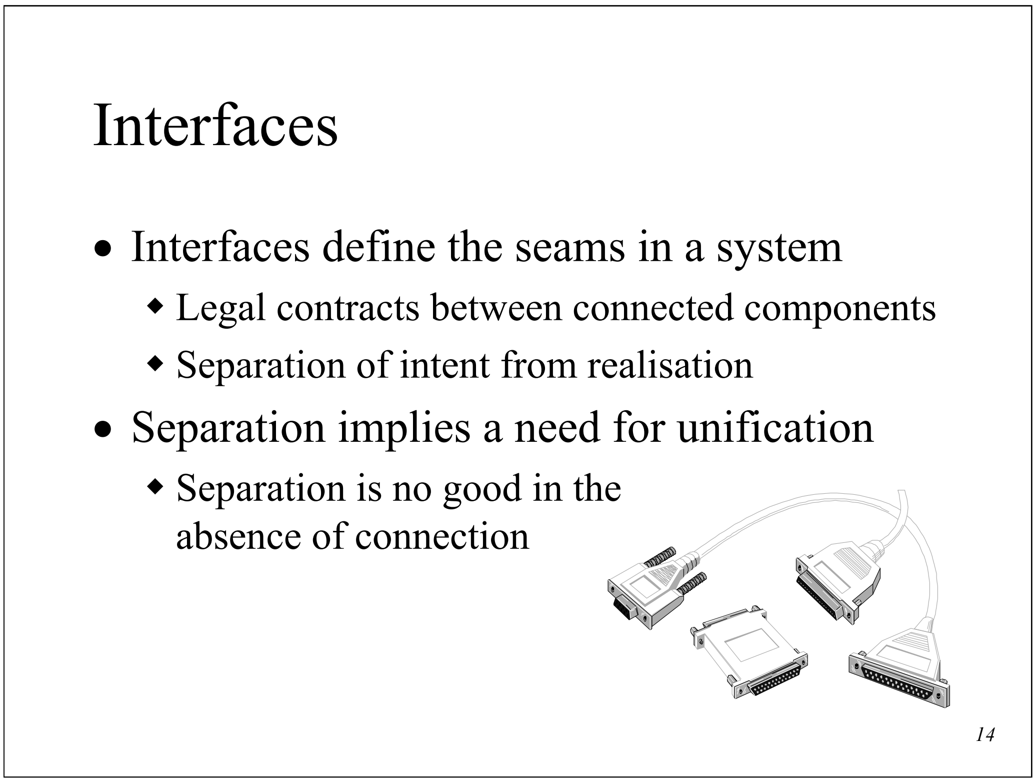
Such a partitioning should be controlled rather than arbitrary, and guided by well understood principles.

1. *Generally management of many is the same as management of few. It is a matter of organization.*
2. *And to control many is the same as to control few. This is a matter of formations and signals.*

Sun Tzu [SunTzu1963]

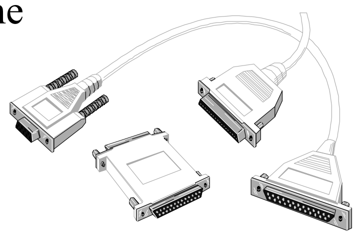
Partitioning leads to an architecture composed of components and connectors that define the linkage between components, including the nature of their interaction. Interfaces represent the protocol or model used to connect components together. Note that the definition of *component* used in the context of Component Based Development (CBD) [Szyperski1998] is a specialisation of the more general term used here. CBD uses the term more strictly to mean executable unit of deployment, i.e. COM, CORBA and the Java technologies.

Coupling and cohesion are the properties that can be observed of any partitioning. 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.



Interfaces

- Interfaces define the seams in a system
  - Legal contracts between connected components
  - Separation of intent from realisation
- Separation implies a need for unification
  - Separation is no good in the absence of connection

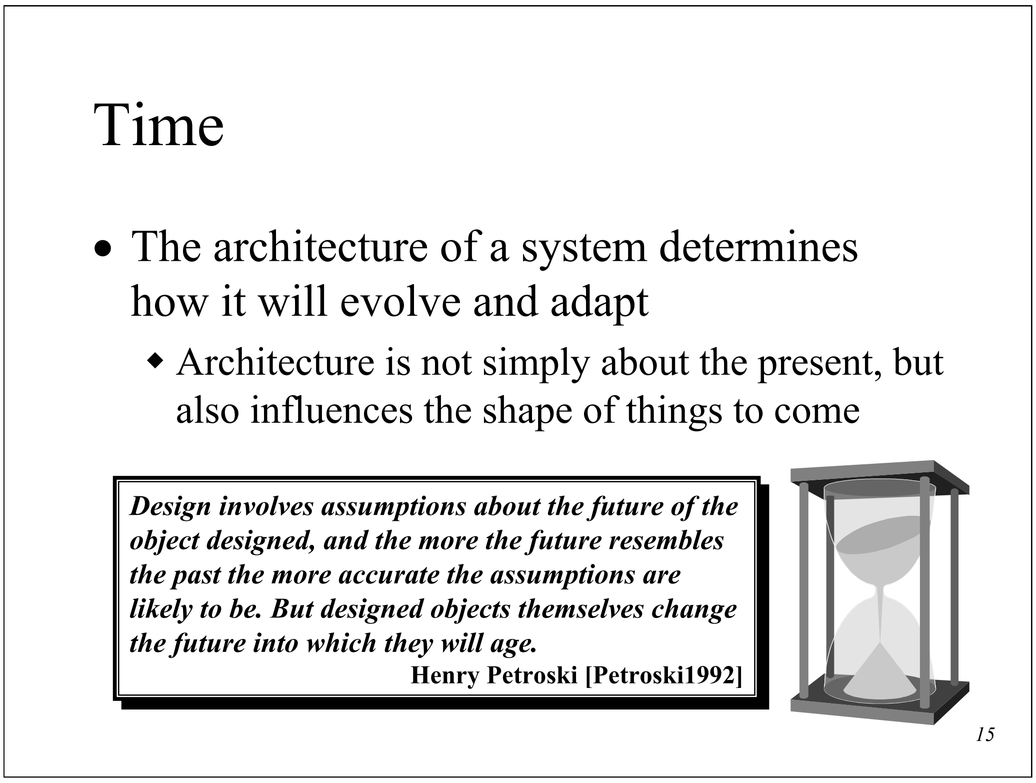


Interfaces define a contract between client and supplier of an abstraction [Meyer1997]. In C an interface is considered to be the type and functions deployed in a header file. In IDLs the definition of interface seems self explanatory, but must also take into account file partitioning. Java has explicit 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.

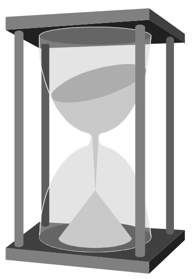
Interfaces should be expressive, consistent and complete. A counterexample of expressiveness is the constructor for Java's `FileWriter` class which requires a `boolean` to indicate whether or not a file should be opened for appending. The use of such flags is inexpressive. A more appropriate interface would be to offer named class Factory Methods [Gamma+1995] `openForAppending` and `openForWriting`. 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. Completeness is a relative rather than absolute concept, and therefore we must ask "complete with respect to what?" when we consider completeness of interfaces. A quest for completeness often leads to unmanageable kitchen sink interfaces that lack focus, e.g. C++'s `basic_string` template class.



Time

- The architecture of a system determines how it will evolve and adapt
  - Architecture is not simply about the present, but also influences the shape of things to come

Design involves assumptions about the future of the object designed, and the more the future resembles the past the more accurate the assumptions are likely to be. But designed objects themselves change the future into which they will age.
Henry Petroski [Petroski1992]



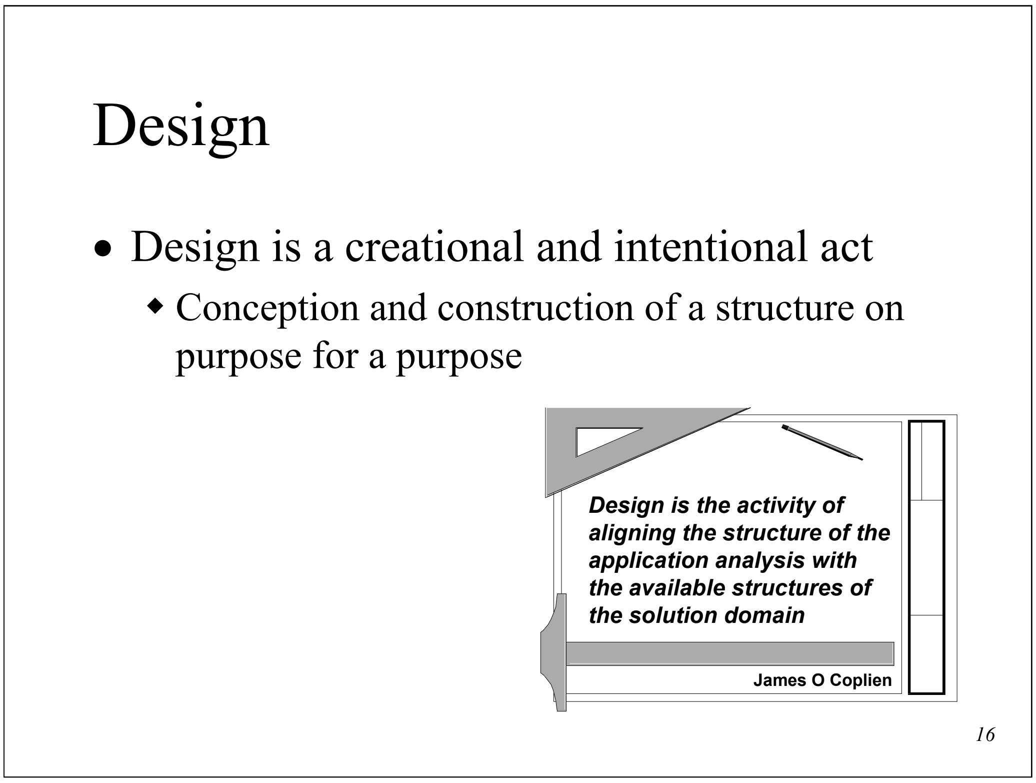
15

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]. 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. Adopting an appropriate mindset is a challenge [Petroski1992]:

> *Engineering deals with lifetimes, both human and otherwise. If not fatigue or fracture, then corrosion or erosion; if not war or vandalism, then taste or fashion claim not only the body but the very souls of once-new machines. Some lifetimes are set by the intended use of an engineering structure. As such an offshore oil platform may be designed to last for only the twenty or thirty years that it will take to extract the oil from the rock beneath the sea. It is less easy to say when the job of a bridge will be completed, yet engineers will have to have some clear idea of a bridge' lifetime if only to specify when some major parts will have to be inspected, serviced or replaced. Buildings have uses that are subject to the whims of business fashion, and thus today's modern skyscraper may be unrentable in fifty years. Monumental architecture such as museums and government buildings, on the other hand, should suggest a permanence that makes engineers think in terms of centuries. A cathedral, a millenium.*

There we can see an additional quality in an architecture [Coplien1999]:

> *A good architecture encapsulates change.*



Design

- Design is a creational and intentional act
  - Conception and construction of a structure on purpose for a purpose

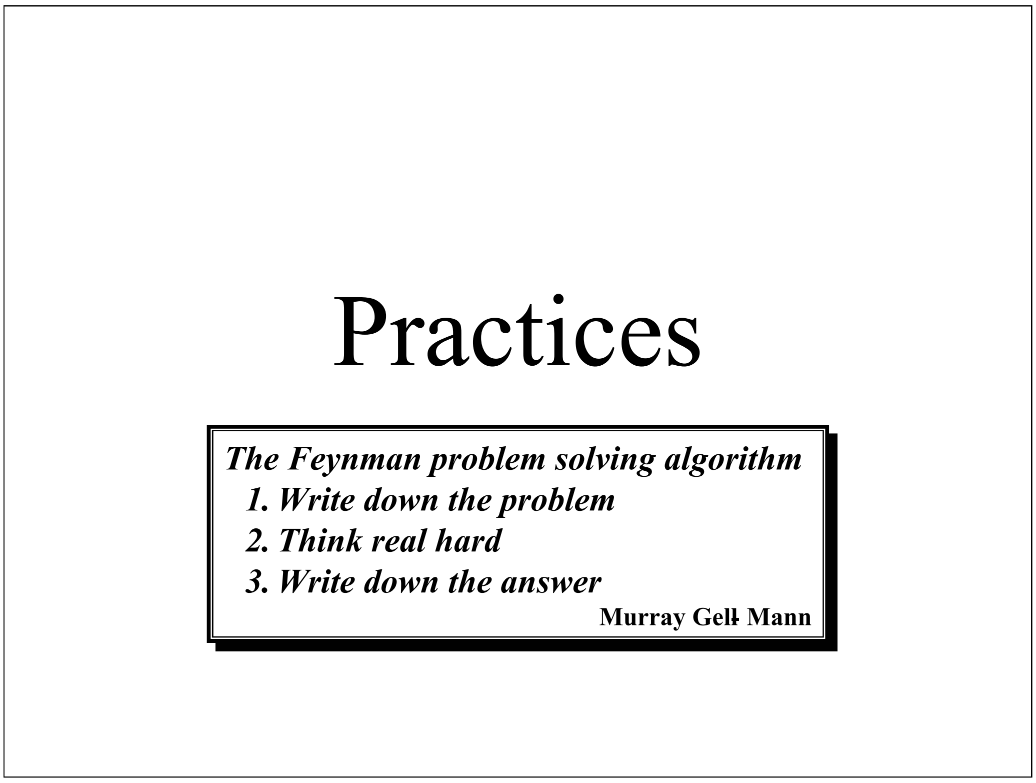
Design is the activity of aligning the structure of the application analysis with the available structures of the solution domain

James O Coplien

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

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

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



Practices

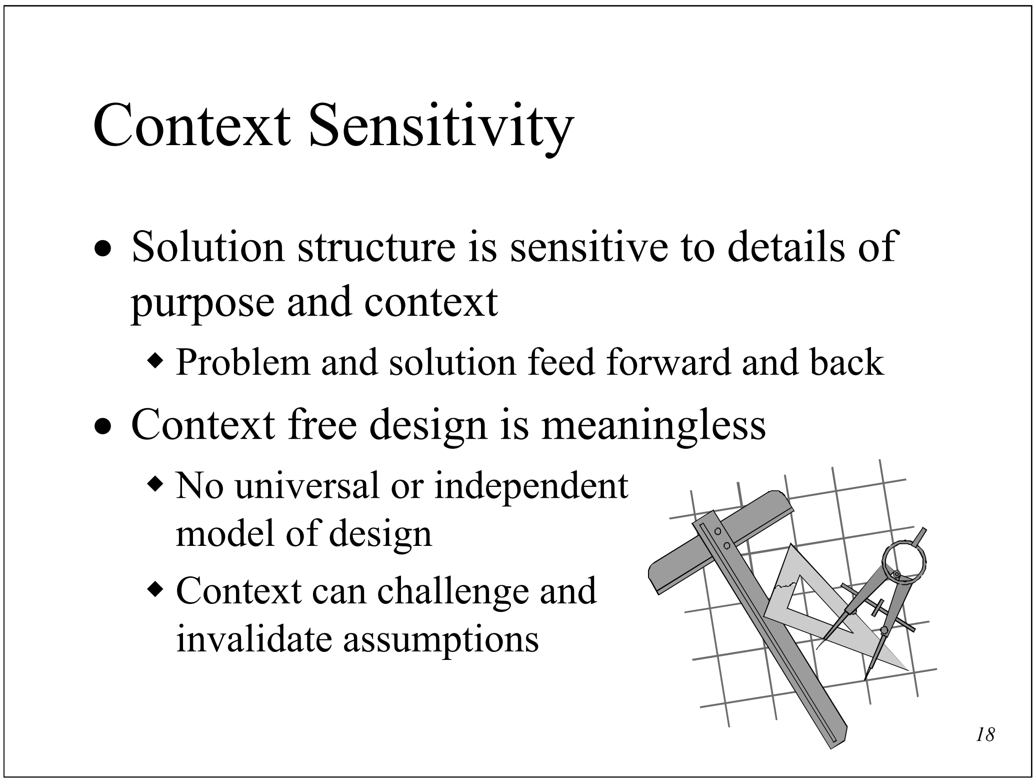
The Feynman problem solving algorithm
1. Write down the problem
2. Think real hard
3. Write down the answer
Murray Gell‑Mann

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 attempts to establish a bridge of understanding that starts in the problem domain and ends in the solution domain. It is also recursive: the solution to a problem at one level or stage reveals itself to be the problem domain for the next.

So when does design happen? Or is that not a meaningful question? Perhaps it is easier to bound design in time – between the conception and realisation of a system – and let methodologists assert their preferences and definitions! And what practices should be followed? Again, much of the emphasis will come from a paradigm, but this section includes (non-exhaustively) some practices and mindsets that have been shown to be successful.



Context Sensitivity

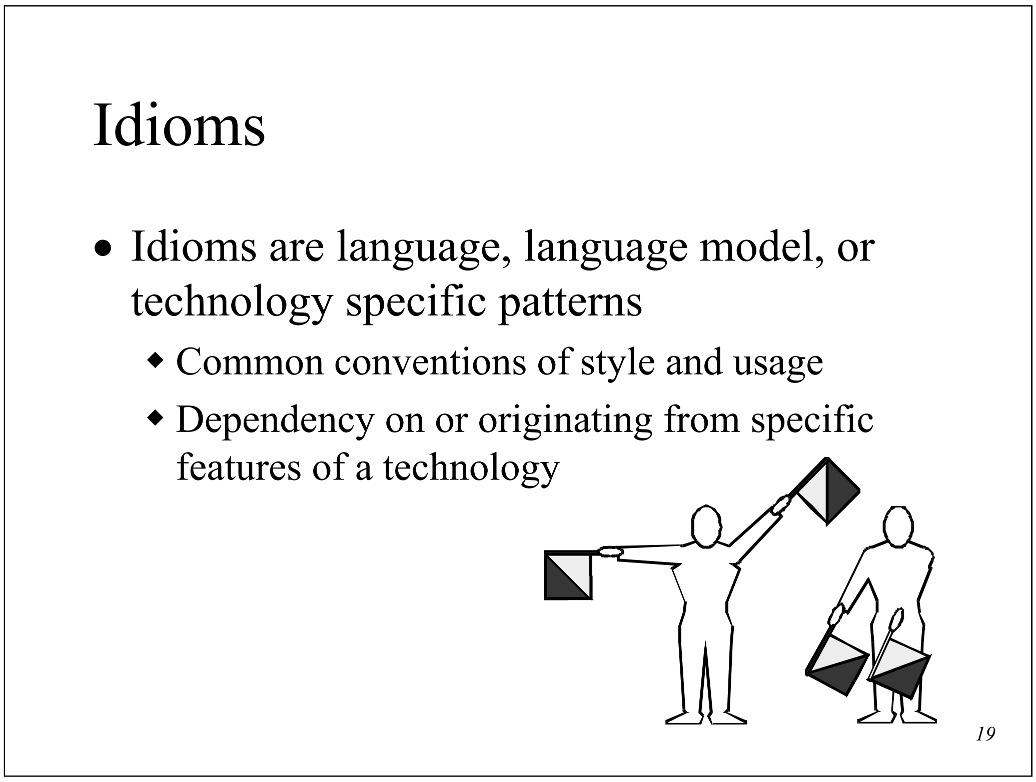
- Solution structure is sensitive to details of purpose and context
  - Problem and solution feed forward and back
- Context free design is meaningless
  - No universal or independent model of design
  - Context can challenge and invalidate assumptions

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.

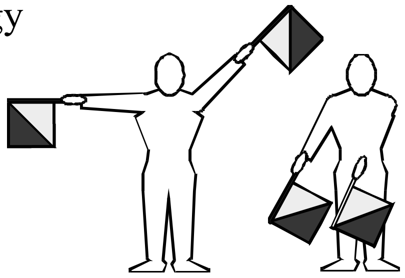
On the compromise of design, David Pye is quoted in [Petroski1992]:

> *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.*



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, literally, what the locals speak. In this case techniques applied by the users in a programming language culture. 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 has documented a great 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).

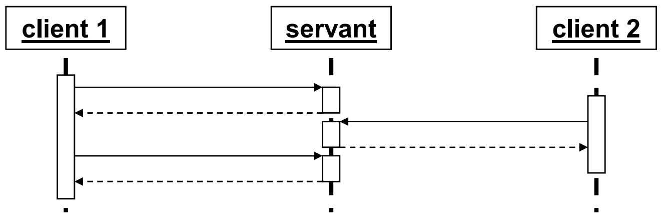
Some idioms, such as those for naming, can be transferred easily across languages. Others depend on features of a language model and are simply inapplicable when translated: strong and statically checked type system (e.g. C++ and Java) versus a looser, dynamically checked one (e.g. Smalltalk and Lisp); 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. For example, a great many C++ libraries have suffered from inappropriate application of Smalltalk idioms, and a great many Java designs have been compromised by C++ style – syntactic similarity between languages can be deceptive.

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.

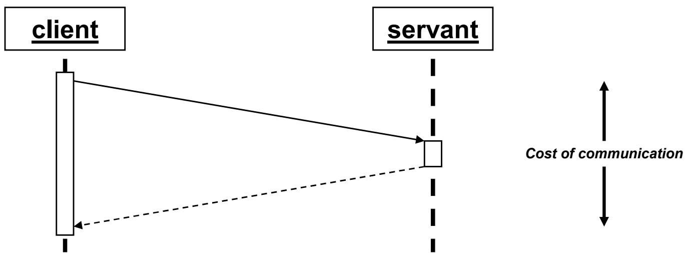
Note that attempting to generalise concurrent programming practices from sequential programming is the wrong way: sequential programming is a limited case of concurrent programming, and not vice-versa. This means that techniques such command/query separation and programming by contract [Meyer1997] do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

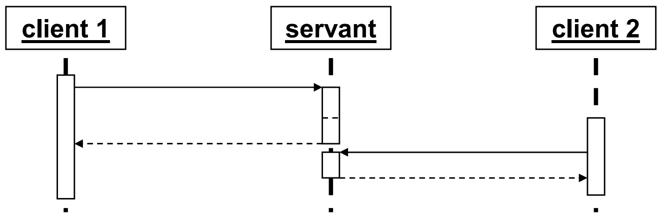
In addition to all of the issues raised with concurrency, there is additional cost involved in remote operation calls. When sketched out on a sequence diagram, sequences of property *get*s and *set*s suffer the 'sawtooth' effect, spending more time in communication than they do in performing useful work.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

Context Can Affect Interface

- Define compound rather than primitive operations based on common usage
  - Combined Operation for operation sequences
  - Combined Attributes for attribute groups
  - Batch Operation handles repetition



The overriding principle in concurrent and distributed 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. It is also worth noting that synchronisation incurs an overhead and reduces the concurrency of a system, and so frequent locking and long locks are not considered wise strategies.

One common resolution of concurrency issues is the Combined Operation [OMG] idiom, i.e. attributes are queried and set in related groups, reducing synchronisation cost and complexity, and ensuring coherence. In other words, common usage of attributes is made atomic rather than the individual attributes' manipulation. This can be taken further, at the cost of introducing control coupling, by supporting flag options on an operation that offer more fine grained control [Box+1999].

This can be supported and made more convenient with Combined Attributes that group information details, i.e. properties or attributes, of an interface together in a `struct` type (or other value object) provided by the interface.

Not only do such idioms make interfaces implicitly safer with respect to concurrency and distribution, it also makes them more self descriptive: rather than being presented with a bucket of primitive 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.

Constraints

- Constraints bound the meaningful behaviour of a system
  - Can be realised in types, exceptions, language features, etc.
- Constraints can be liberating
  - Ensuring what's true is true and what's not is not frees rather than binds a developer

$$\boldsymbol{F = G\,\frac{m_1 m_2}{r^2}}$$

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. Preconditions can only be weakened if class invariants and postconditions are preserved or strengthened [Meyer1997, Szyperski1998].

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` is 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.

Minimalism

- Additional options and features can lead to confusion rather than clarity
  - Overachieving interfaces are weaker and more complex not stronger and simpler

The difference between a good and a poor architect is that the poor architect succumbs to every temptation and the good one resists it.

Ludwig Wittgenstein

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:

The minimum could be defined as the perfection that an artefact achieves when it is no longer possible to improve it by subtraction. This is the quality that an object has when every component, every detail, and every junction has been reduced or condensed to the essentials. It is the result of the omission of the inessentials.

[Pawson1996]

I have made this letter longer than usual, only because I have not had the time to make it shorter.

Blaise Pascal

[Rule] 17. Omit needless words

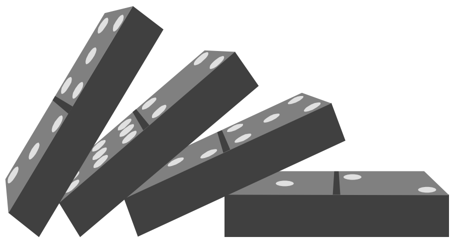
Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all his sentences short, or that he avoid all detail and treat his subjects only in outline, but that every word tell.

Many expressions in common use violate this principle.

[Strunk+1979]

Dependency Management

- Partition to minimise dependencies
  - Low coupling and high cohesion
- Put things together that change together
  - Where dependencies exist they should be on stable elements



One of the features that typifies any architecture driven approach is the management of dependencies in a design. 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.

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.

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].

Decoupling

- Interface Decoupling
  - Separate client usage interface from creational implementation class
- Role Decoupling
  - Depend on the interface, the whole interface, and nothing but the interface
- Dependency Inversion
  - Can be used to break cyclic dependencies

Interface Decoupling is a pattern that introduces a *vertical* separation, i.e. a separation through inheritance rather than composition. An interface is fully abstract, defining a *usage type* [Barton+1994] and modifications in the implementation should not affect it, whereas a concrete class provides implementation details and is the *creation type* [Barton+1994], i.e. it is only visible at the point of creation of an object. Interface Decoupling is a natural property of CBD architectures, such as COM and CORBA, and is easy to express in Java because of its explicit support for interfaces. In C++ the support comes idiomatically, e.g. through the use of fully abstract base classes without any implementation features. C++ also supports horizontal decoupling techniques, principally through the use of forward declared, opaque types.

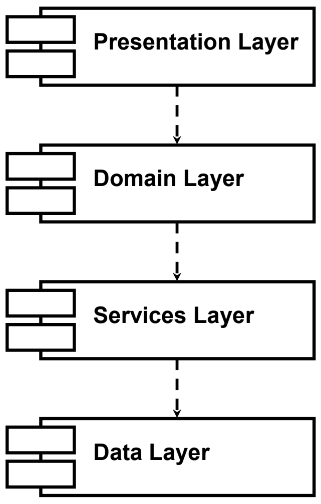
Role Decoupling [D'Souza+1999] goes a step further by factoring out smaller interfaces based on usage. This is effectively interface normalisation, and reduces dependencies with respect to operations in an interface [OMG]:

> *Interface inheritance (subtyping) is used whenever one can imagine that client code should depend on less functionality than the full interface. Services are often partitioned into several unrelated interfaces when it is possible to partition the clients into different roles. For example, an administrative interface is often unrelated and distinct in the type system from the interface used by "normal" clients.*

Dependency Inversion [Martin1996] supports inversion of dependencies between components. This can be used to reverse the dependency of a stable component on a less stable one, and to break cyclic dependencies across package boundaries.

Layering

- Layers are encapsulated levels in a system
- Systems may be layered with respect to...
  - Levels of abstraction
  - Rate of change
  - Development skills
  - Organisational structure



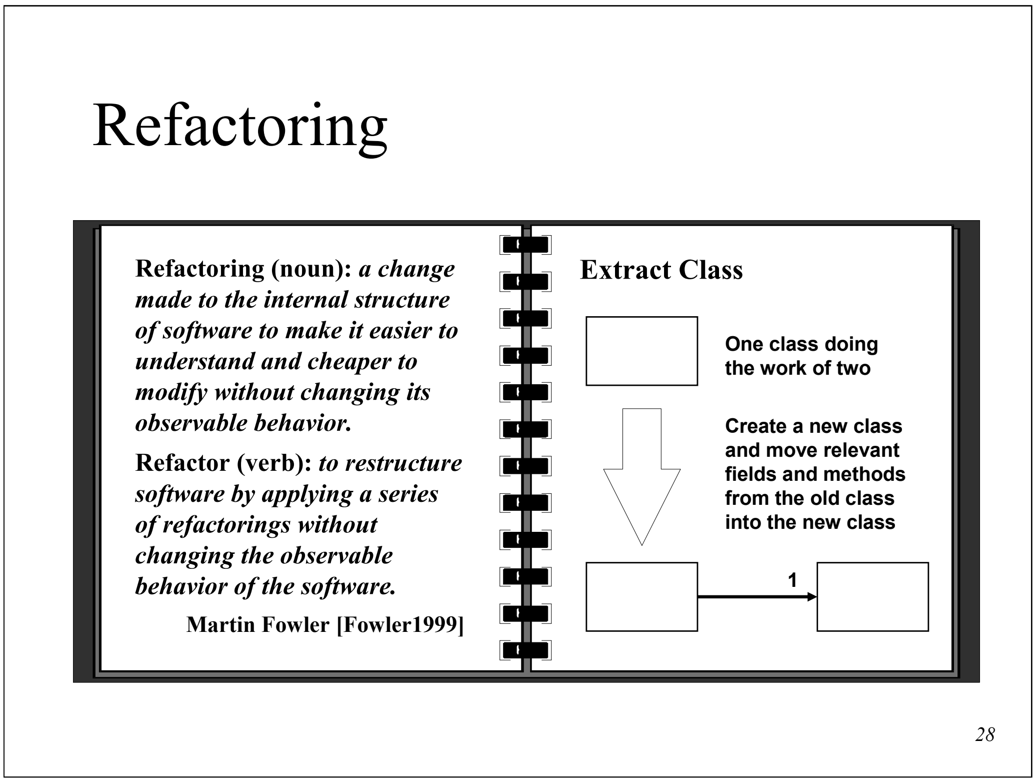
The Layers pattern [Buschmann+1996] may be summarised as:

> *The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.*

The use of encapsulated system layers can be seen as a disciplined use of delegation to construct complex components out of simpler ones. The discipline arises from both full encapsulation (to be contrasted with Façade [Gamma+1995], which otherwise has certain similarities) and strict assignment of roles to layers (as opposed to a heterarchy of mixed layers), which serves to decouple one layer from another. Naturally, as with any decoupling, there may be an overhead so this must be considered when balancing up the options.

The concept of change is an important one in systems with many levels, whether or not the system is explicitly layered or merely its semantics can be understood in terms of layers (although the suggestion is that these should be made explicit in the architecture). It is often the case that different layers in a system are subject to different forces – for instance, technology or requirements drift – and these produce different sheering rates.

The idea of such time ordered layering in software [O'Callaghan1998, Dai+1999] is based on the idea of shearing layers of change in buildings [Brand1994], which identifies six components with progressively more rapid rates of change: *site* (geographical setting); *structure* (foundation and load bearing elements); *skin* (exterior surfaces); *services* (e.g. electricity, water); *space plan* (interior partitioning and layout); *stuff* (that which fills the space).



Refactoring

Refactoring (noun): ***a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.***

Refactor (verb): ***to restructure software by applying a series of refactorings without changing the observable behavior of the software.***

Martin Fowler [Fowler1999]

Extract Class

One class doing the work of two

Create a new class and move relevant fields and methods from the old class into the new class

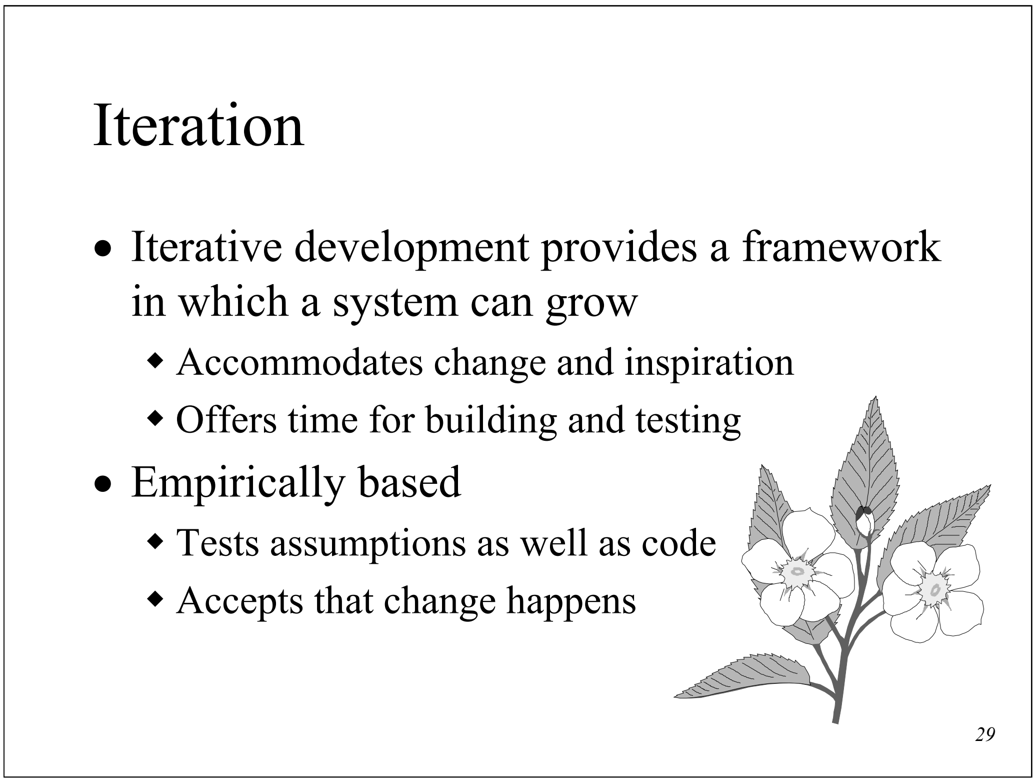
1

Predicting the future is non-trivial. Refactoring [Fowler1999] offers a pragmatic but sound route from the past to the future with a focus on the present that accommodates change without the need for either clairvoyance or breakage. It keeps a system's architecture clean as a system evolves beyond its original scope and functionality. Importantly, refactoring can keep the size of a system in check and its actual complexity low, benefits which pay for themselves many times over.

For refactoring to be successful requires a comprehensive set of automated unit tests that can be run and rerun to confirm the (absence of) effect of each refactoring, fine grained use of version control, and a well defined set of refactoring practices.

Just as the body renews itself, such revision should be considered healthy. When a system and its associated development culture loses its ability to change and be changed it becomes stagnant and dies (or goes senile at the very least). Such practices are common to many domains [Strunk+1979]:

> *[Reminder] 5. Revise and rewrite*
>
> *Revising is part of writing. Few writers are so expert that they can produce what they are after on the first try... Do not be afraid to seize whatever you have written and cut it to ribbons; it can always be restored to its original condition in the morning, if that course seems best. Remember, it is no sign of weakness or defeat that your manuscript ends up in need of major surgery. This is a common occurrence in all writing, and among the best writers.*



Iteration

- Iterative development provides a framework in which a system can grow
  - Accommodates change and inspiration
  - Offers time for building and testing
- Empirically based
  - Tests assumptions as well as code
  - Accepts that change happens

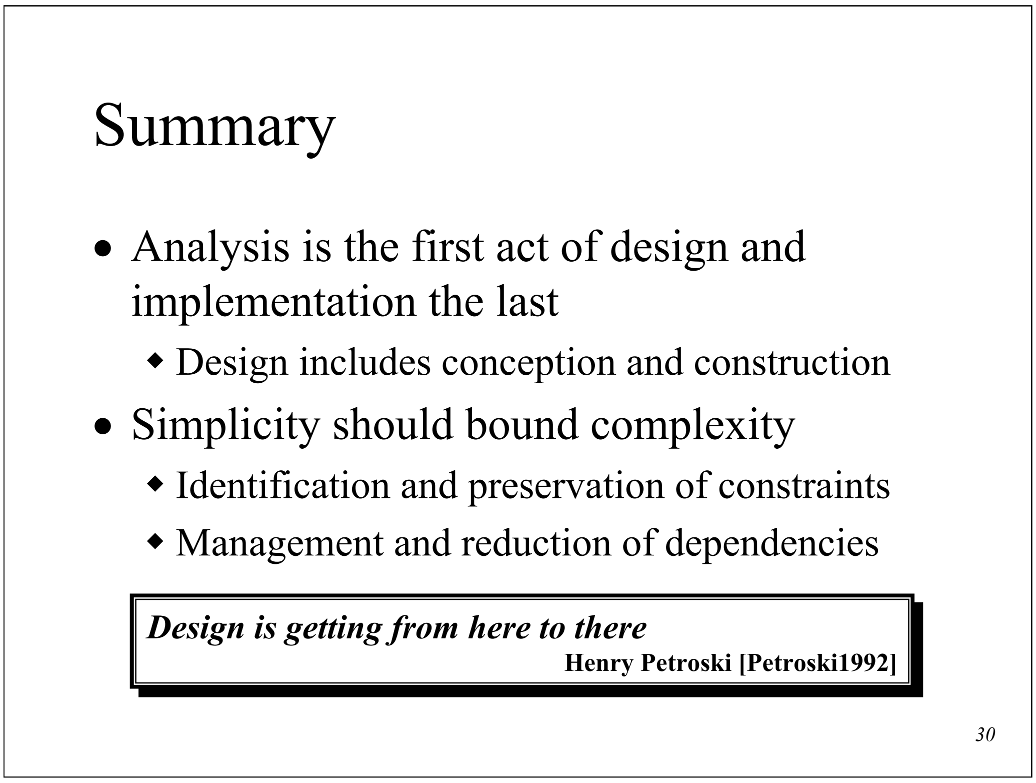
29

Many books on analysis would have you believe that once analysis is complete, implementation is trivial, and design is only a simple step on the road from to the other. It is hard to conceive of something that is further from practical reality than this point of view. At the opposite extreme are programmers and authors that believe that any formal analysis is irrelevant, and design is merely a way of diagramming code: heroic programming and pure technical knowledge will overcome any apparent obstacles and shortcomings in a project.

Project risk management and full discussion of methodologies and lifecycles are beyond the scope of this talk, suffice to say that such compartmentalised views have been shown as inadequate in supporting the development of complex software. Many attempts at capturing the essence of a successful development process in a simple form have often ended up more simplistic than simple: it is easier to prescribe a rigid process than a dynamic one.

The steps – whatever steps are decided upon – of software development are intimately linked, and as such form a chain. As with any chain, the development lifecycle is only as strong as its weakest link. The best laid architecture can be undermined by poor implementation, and the best implementation technology held back by poor analysis.

So it is with design, which is a non-trivial transformation whether formally or informally tackled. Until recently what has been elusive is a set of concepts to support successful design. Effective design runs more like a blackboard process [Buschmann+1996] than a simple lifecycle step: with whole solutions arising out of incremental, iterative and empirical solution of partial problems.



Summary

- Analysis is the first act of design and implementation the last
  - Design includes conception and construction
- Simplicity should bound complexity
  - Identification and preservation of constraints
  - Management and reduction of dependencies

Design is getting from here to there
Henry Petroski [Petroski1992]

Respect for the developer must be one of the golden rules for a development process, and hence an attitude to design. Attempts to address software development schedules by code generation are typically not useful. Code generation falls into two basic categories: generation of black box code that is complex and in a different layer, e.g. generation of proxies in any CBD system; generation of code to save typing (e.g. header file generation from a UML diagram) or work around poor library design (e.g. MFC). The latter approach solves the wrong problem: typing is not the bottleneck in software development, and as development is a thinking profession removing thinking time is like removing preparation time for a performance artist – looks great on the schedule, but is self defeating in practice.

Design is a human activity and is complex enough without attempting to solve the wrong problem or dehumanise it. The automation of design is believed by some to be the holy grail of development. However, this ignores the experience of other professions, where tools support design rather than replace it or dictate it. As Don Olson observes [Rising1998]:

> *An interesting tactic of this bunch is to point out that people used to code only in assembly languages and now work in higher level languages. From this they extrapolate that as we continue to evolve languages, eventually specification languages will result, which allow us to specify at a comfortably high level whatever it is we have to build. Unfortunately, this is pure crap.*

A more worthwhile aim is to identify the things we do well, and reflect on and communicate them. Design is a journey worth taking, not a cancer to be cured.

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

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

[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.

[Booch+1999] Grady Booch, James Rumbaugh and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[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.

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

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

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

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

[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.

[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.

[Henney1999] Kevlin Henney, "For the Sake of Simplicity", *EXE*, July 1999.

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

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

[Kernighan+1979] Brian W Kernighan and P J Plauger, *The Elements of Programming Style*, 2nd edition, McGraw-Hill, 1979.

[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.

[Martin1996] Robert Martin, "The Dependency Inversion Principle", *C++ Report*, SIGS, May 1996.

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

[O'Callaghan+1996] Alan O'Callaghan and Clazien Wezeman, "Patterns for Legacy Systems' Migration to Object Technology", *Getting the Best out of Patterns and Architectures*, Unicom, 1996.

[OMG] *CORBAservices: Common Object Services Specification*, OMG, `http://www.omg.org`.

[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.

[PLoP1998] Edited by Robert Martin, Dirk Riehle and Frank Buschmann, *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

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

[Rumbaugh+1999] James Rumbaugh, Ivar Jacobson and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[Shaw+1996] Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

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

[SunTzu1963] Sun Tzu, *The Art of War*, translated by Samuel B Griffith, Oxford University Press, 1963.

[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.