Pattern-Based Software Development

A Hands-on Introduction with Models and Java

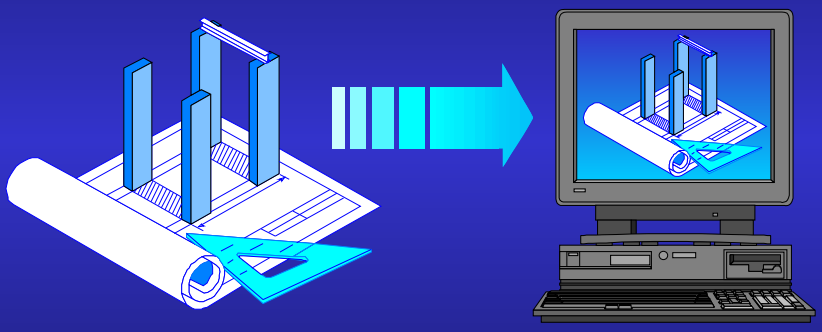
Ankara
January/February 2015
(Not for redistribution)

Curbralan Ltd
www.curbralan.com

Contents

Pattern-Based Software Development: A Hands-on Introduction with Models and Java, version 07020I.

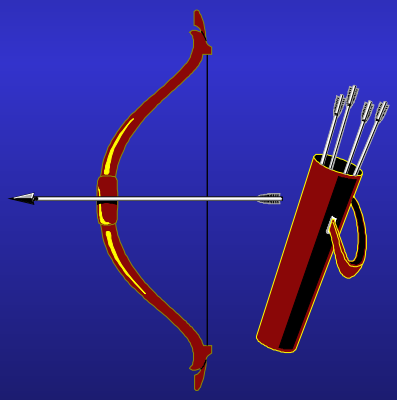
Course Introduction

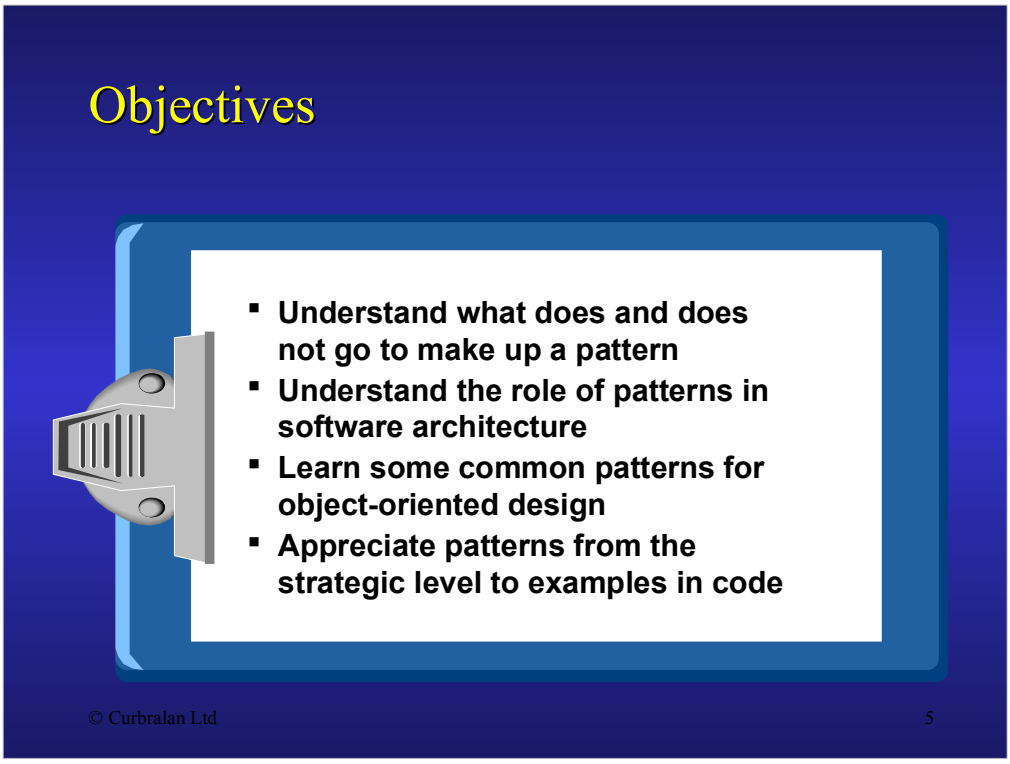


Pattern-Based Software Development

Course Introduction

- Objectives
  - Define the scope and purpose of this course
- Contents
  - Objectives
  - Prerequisites
  - Questions



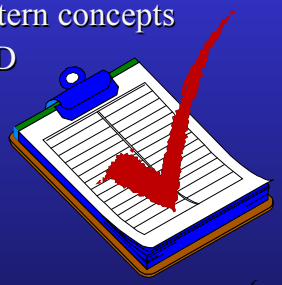


Non-trivial systems include many recurring design problems whose solutions are commonly repeated from place to place, and from programmer to programmer. The essence and basic structure of a solution may be repeated many times, even though the realisation is different in each case. Patterns offer a technique for capturing design and architecture, presenting and communicating architectural knowledge at all levels of a system, from broad-brush architecture to programming language detail, allowing experience to be understood and distilled. Patterns allow developers to work on, share and understand designs, but are not a basis for automation of design; frameworks and libraries present code-level commodities often built on common patterns.

The *Pattern-Based Software Development* course introduces patterns from the ground up, presenting principles as well as concrete examples. It develops understanding through lecture, discussion and hands-on exercises.

Prerequisites

- Essential...
  - Hands-on experience of object-oriented development
  - Some familiarity with UML
  - Knowledge of Java
- Useful...
  - Previous exposure to patterns and pattern concepts
  - Previous experience of JUnit and TDD

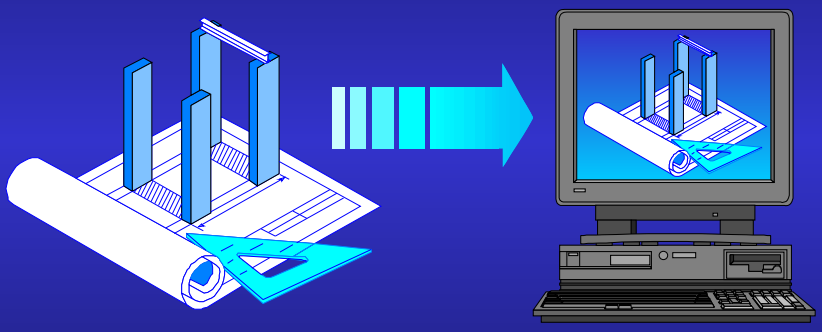


This course is suitable for software developers familiar with object-oriented principles and practices. Some knowledge of UML is assumed, as is hands-on programming experience in a modern language with support for object orientation. Any previous exposure to patterns, whether pattern concepts in general or specific patterns, is beneficial. Likewise, techniques such Test-Driven Development and experience of JUnit is also considered beneficial.

Any Questions?

- Please feel free to ask questions at any time
  - The surest way of having your questions answered is to ask them!

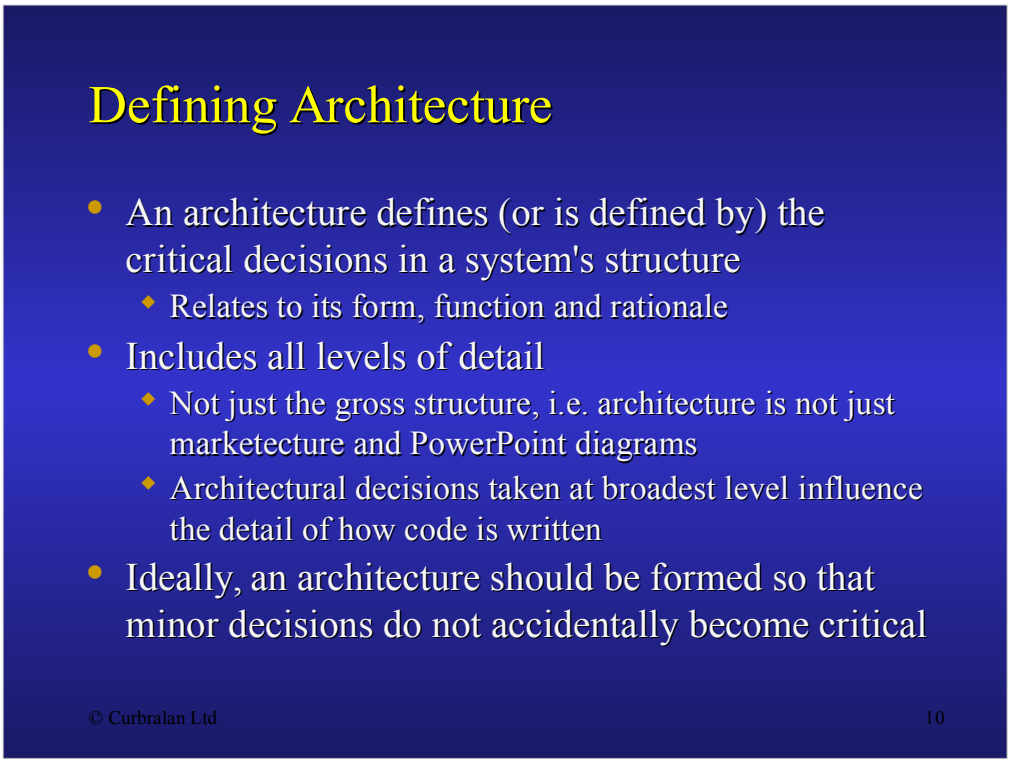
Software Architecture



Pattern-Based Software Development

Software Architecture

- Objectives
  - Outline what is meant by the term software architecture
- Contents
  - Defining architecture
  - Unmanaged dependencies
  - Stability and change
  - Overdesign and sufficiency
  - Models and patterns



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

> 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 [Unified Modeling Language User Guide] defines 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 may be accidental rather than intentional, although there is always agency involved in either outcome.

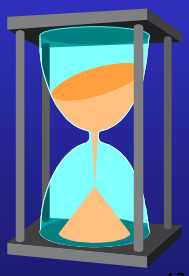
Unmanaged Dependencies

- The dependency structure can weigh down a system and its future prospects
  - Becomes harder to understand
  - Becomes harder to integrate
  - Becomes harder to extend
  - Becomes harder to test
  - Becomes harder to fix
  - Becomes harder...
- Therefore, partition to minimise dependencies
  - Low coupling between components
  - High cohesion within a component

Lower comprehensibility, higher build times, higher bug rates, and so on become the battle in systems with spaghetti dependencies, and each addition of functionality works against rather than with the system. Put in simple terms, poor system structure costs time, money and other organisational attributes. Therefore any effort to improve structure should be treated as investment and a proper part of a software system's lifecycle.

Stability

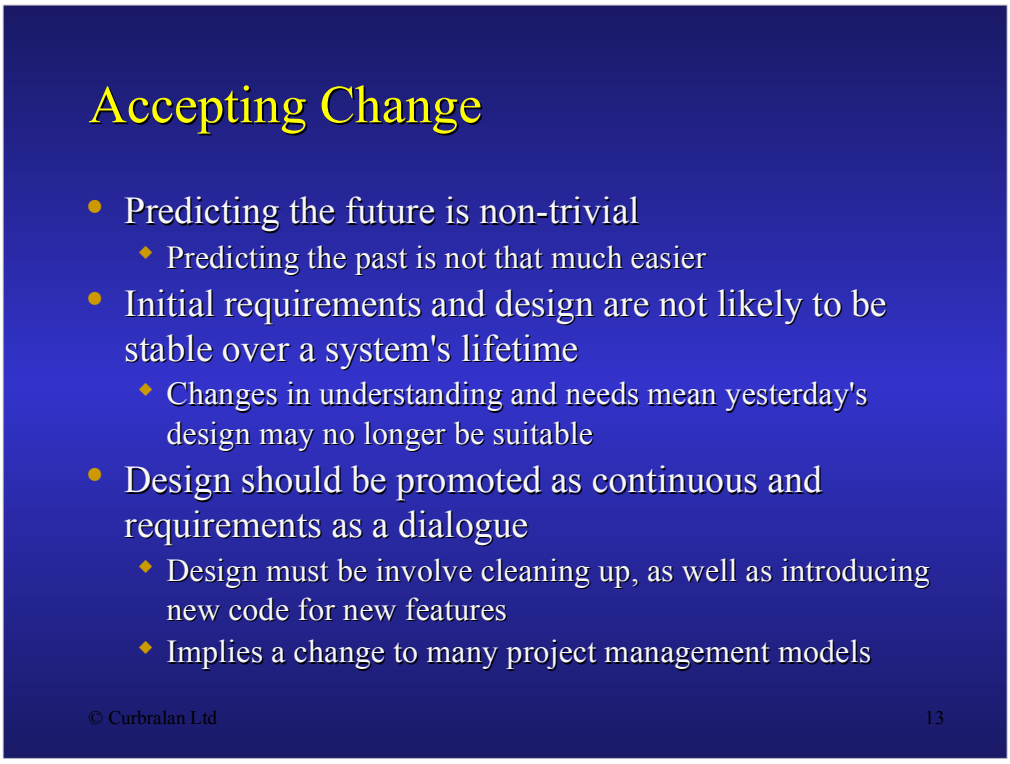
- Dependencies should be on more stable elements with the same rate of change
  - Put things together that change together
- Interfaces should be more stable than their implementations
  - Either because of good design or because of fear of change
- A system's partitioning can be with respect to rate of change or variability
  - Not just in terms of abstraction or technology choices



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. The idea of such time-ordered coupling 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 that fills the space).

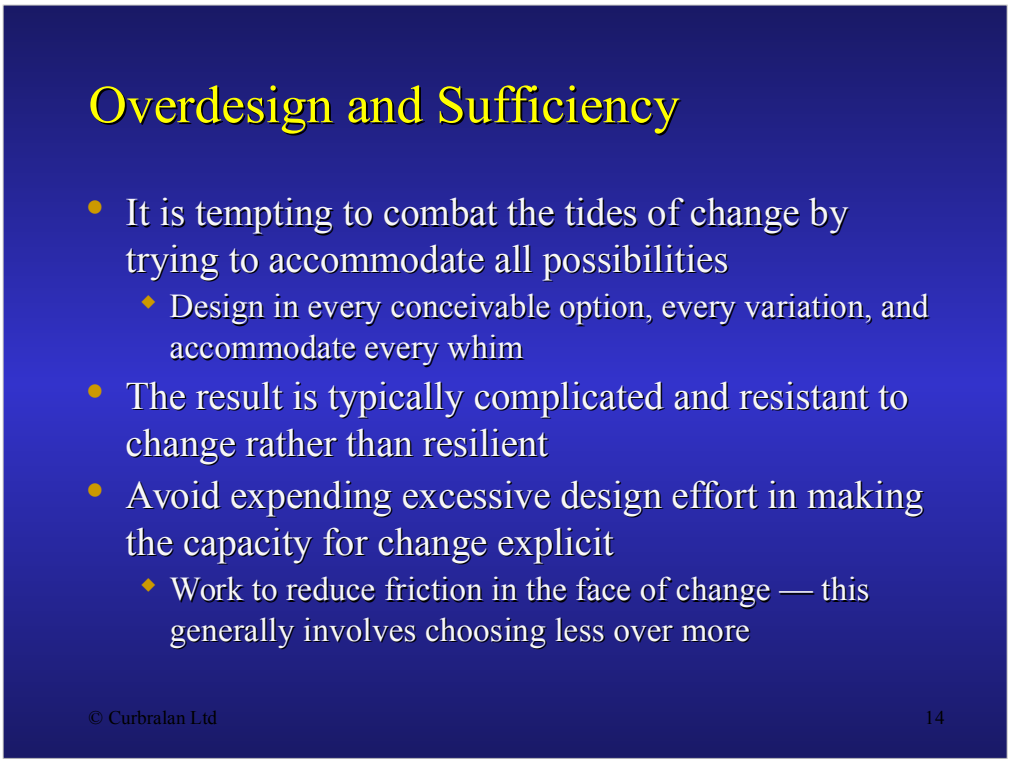
It is therefore tempting to program only in the future tense, adding complexity by building for possibilities that may never happen, but it is also tempting to program only in the here and now, ignoring the possibility of and processes for change, and therefore being surprised and unprepared when change occurs.

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. The more abstract something the more stable it should or must be, i.e. program to interface not an implementation.



It is a mistake to consider a piece of code that works as fixed and immutable, never to be changed again. As development progresses code can nearly always be improved — duplicate code across a system duplicated, unused code removed, etc — so a discipline for doing so should be considered a part of ordinary development.

But accepting that change happens is not the same as accommodating creeping featurism or bending over backwards to all requests for change. Part of accepting change is to accept it as a first class element of the development process. This does not mean that there should be no push back against arbitrary change.



How can a design handle inevitable change? Can design embrace all possible futures without requiring further change? Such master-plan approaches often lead to excessively large and complex frameworks that are grounded in fiction rather than fact, which means that they will inevitably have to be changed to accommodate unanticipated requirements and feedback. Changing such behemoths is harder than changing lighter, streamlined software honed to its task. On the compromise of design, David Pye offers the following:

> 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.

Generalisation and reuse are often seen to be virtues in their own right, but without the context of a real system — or, to be precise, many systems — generalisation and reuse make little sense as design criteria. Often, when reuse or generality are taken as design guidelines, the resulting design is full of unnecessary and unreasonable options that are difficult to understand and hard to use. Code can become general but be conceived of as specific if designed with a few principles in mind. Speculative generalisation is more often than not a failure rather than a success.

Vision and Communication

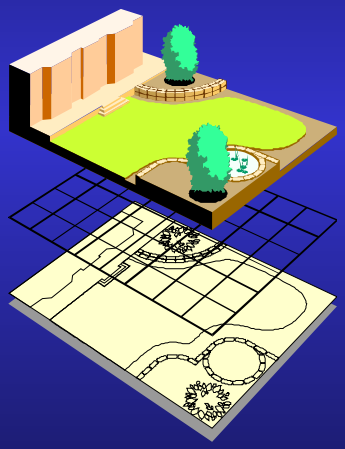
- An architect has responsibility to formulate and articulate an architecture
  - Structure, technologies, tools, roles, etc
- But it is not enough just to have the vision of a system's architecture
  - It's no good as a secret
  - Architecture and style need to be communicated and nurtured, which includes both preservation and evolution

An architect needs to establish the vision the system in co-operation with analysts, the customer or some other representative of the customer, other developers and any other stakeholders.

Where an analyst will establish a more functional perspective of a system, the architect will focus more on the structural design side, including questions of how the system is to be developed — tools, scheduling, etc — and deployed. The vision of an architecture needs to be clear, communicated, shared and sustainable to be of use in the long term. The high-level view of the system plus some critical details need to be a part of this vision — a vague, prophetic vision that needs deciphering is of little use.

The Role of Models

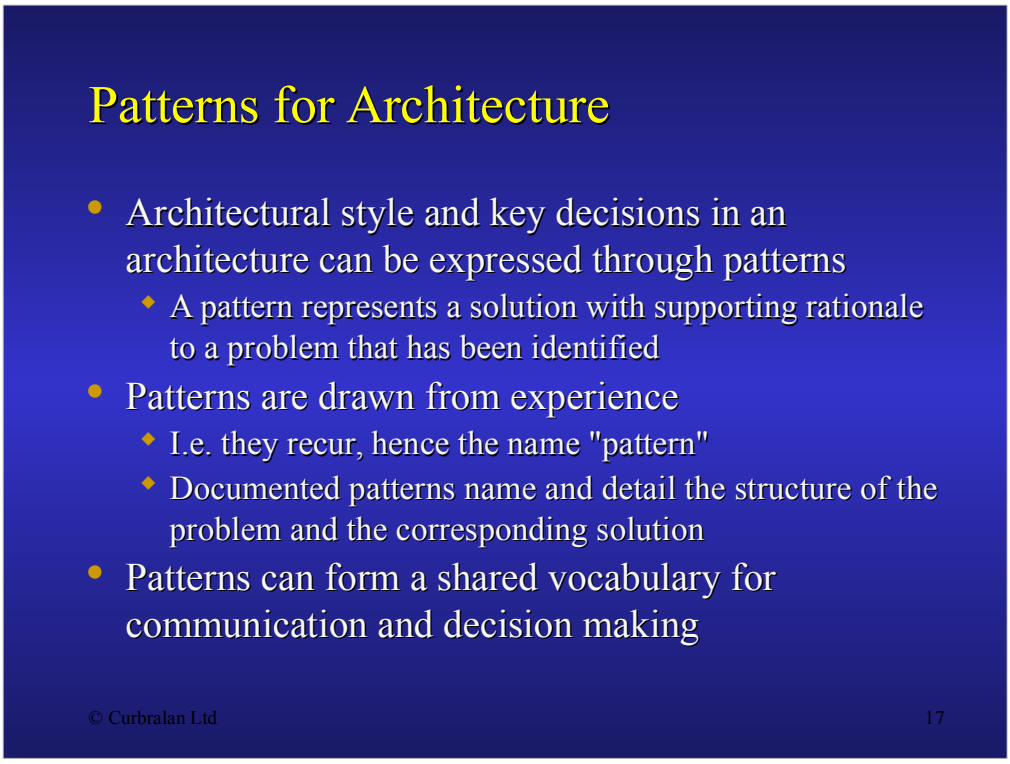
- A model is an abstraction from a point of view for a purpose
  - But don't confuse the map with the territory
  - Distinguish between problem models and solution models
- An architect can use models to communicate many of the key features of an architecture
  - E.g. UML or ad hoc notations



So what exactly is a model? There are many definitions, of which the following captures a lot of what is commonly understood in software [The Unified Modeling Language Reference Manual, Rumbaugh et al]:

> 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.

Michael Jackson [Software Requirements and Specifications] 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.



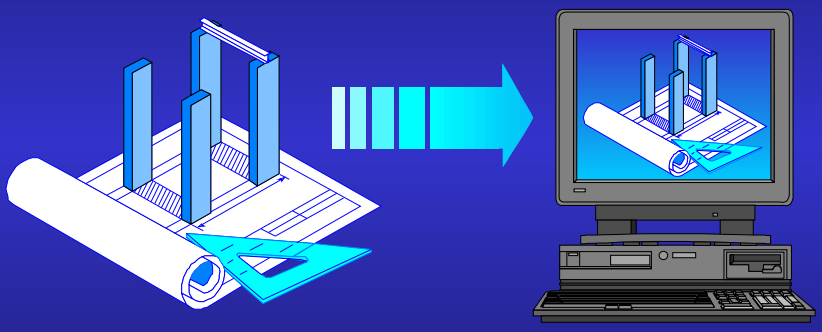
Good patterns are recurring solutions to similar problems that are known to work, i.e. they are empirically based and drawn from experience rather than invented for their own sake. That they are documented pieces of successful design experience and not single instances of design is a significant distinction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

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

Exercise: Architectural Decisions

- List some of the key architectural decisions that have influenced projects you have worked on
  - Include both intentional architectural decisions and accidental ones, i.e. simple localised design decisions that later turned out to have architectural significance
- Was the decision beneficial in the long term, neutral, problematic or of mixed benefit?
  - A neutral feature of an architecture is one that had no obvious positive or negative effect, or where the outcome would have been the same no matter what decision had been taken

Core Pattern Concepts



Pattern-Based Software Development

Core Pattern Concepts

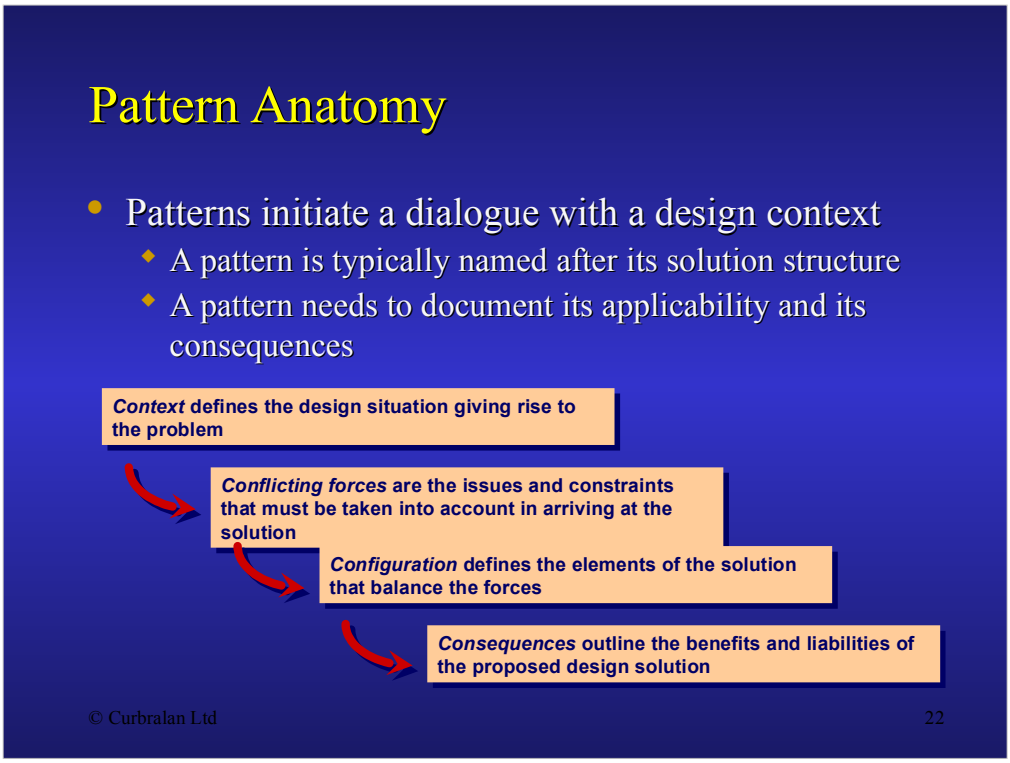
- Objectives
  - Introduce vocabulary and ideas behind software patterns
- Contents
  - Patterns in software architecture
  - Pattern anatomy
  - The role of patterns
  - Essential pattern form elements
  - Common pattern resources

Patterns in Software Architecture

- A pattern documents a reusable solution to a problem within a given context
  - Captures all the forces that define a design problem and the context that is around the design problem
  - Proposes a solution that resolves forces in the problem, and outlines the consequences of applying the solution
- A pattern represents a fragment of design that typically cuts across modularity and scale
  - It represents a reusable idea that can be used to guide the design of code, rather than a reusable piece of code

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

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



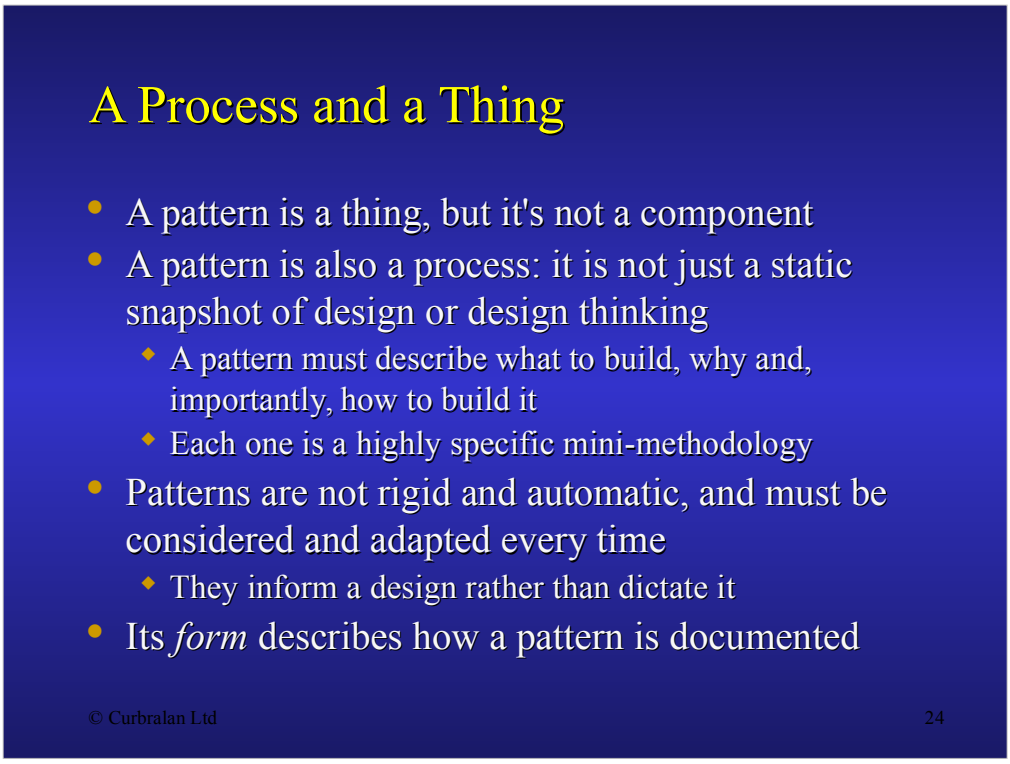
An individual pattern can be considered a study in design and a fragment of design vocabulary. A good pattern tells (and retells) a "successful software engineering story" — albeit a short one.

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

The Role of Patterns

- Patterns establish a design vocabulary
  - Useful for describing architectures, new or old
  - Useful for generating new architectures
- Patterns represent reuse of design knowledge drawn from imagination and experience
  - Good patterns are drawn from real systems and practice
- Patterns may be used consciously and explicitly or unconsciously and tacitly

It does not take explicit, taught knowledge of certain named patterns to use them, but it does take such knowledge to be able to talk about them and communicate them simply. Because patterns are recurring and experience based it could be said that all developers work with patterns. The difference is between developers that recognise those patterns and see how they fit together as a style of design that they can articulate and those who hold those patterns in partial and private form.

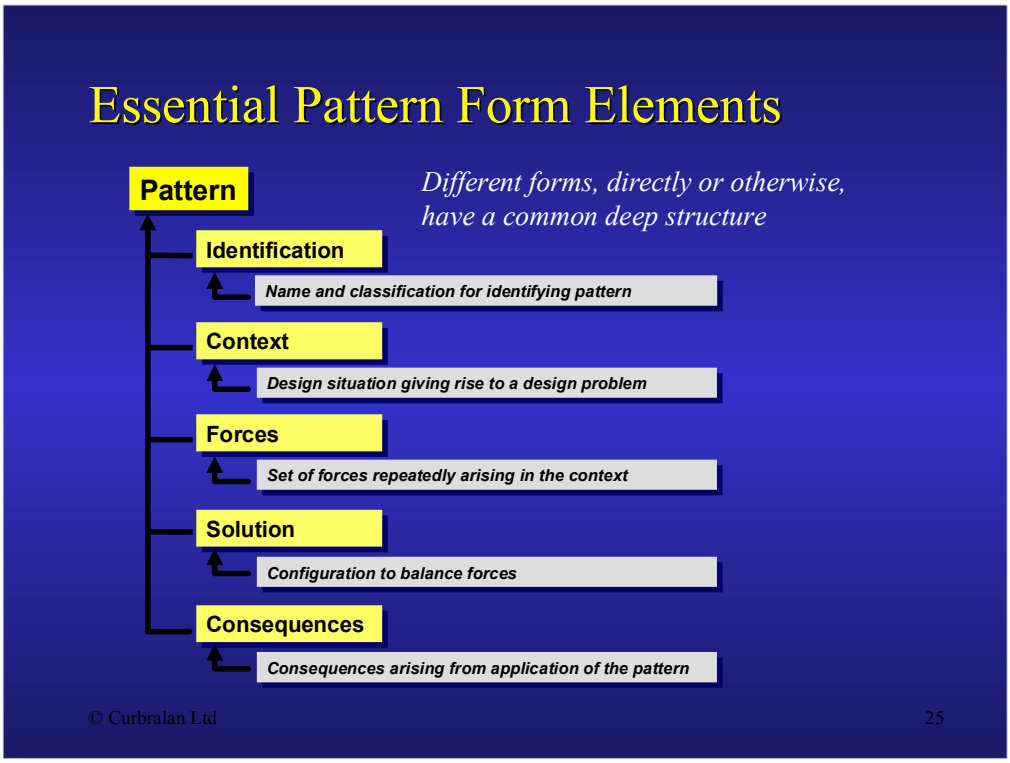


A pattern is sometimes said to be both a process and a thing. This means that a pattern defines not only a structure, but also the reasoning used to arrive at that structure and the details of how to put in place that structure. Many developers mistake patterns to be just structure without the rationale and process of implementation. A single given structure may satisfy many different goals, and consequently belong to many different patterns.

A pattern form is the template or style that a pattern is presented in when written up. It primarily involves text, as this details the rationale and motivating details most clearly. Additionally, pattern text is often supported by diagrams in a semi-formal notation, such as UML. The most common diagrams tend to be class diagrams, showing the static and 'spatial' relationships between classes and their instances, or object diagrams, showing a snapshot of an instance configuration.

Although such static configuration diagrams constitute the majority, there are many patterns whose structure is temporal or control oriented, and hence additional or alternative diagrams (such as collaboration or sequence diagrams) or notation are more appropriate. Such notation includes code, pseudocode or more rigorously in a formal notation.

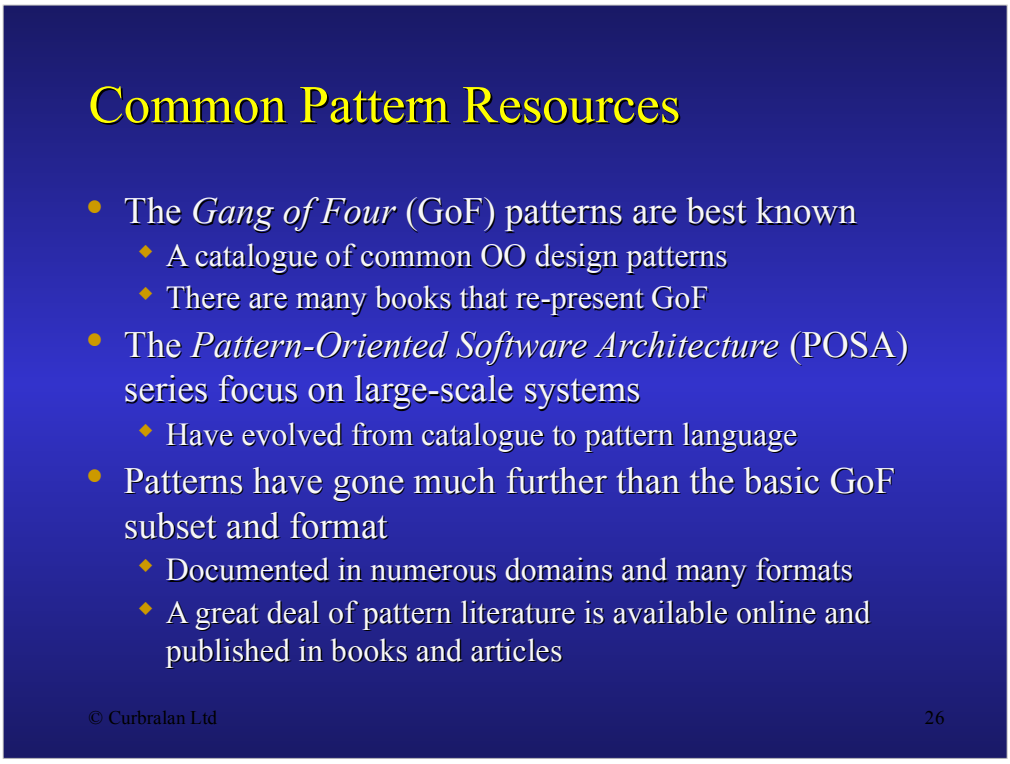
Of course, where patterns do not describe a technical design such diagrams would not be useful. Where patterns focus on code-centric idioms, a diagram may be awkward or less communicative than code. Even for designs that could be expressed using UML, it is not inevitable that UML is the right choice: a more ad hoc sketched form or a different schematic notation would be more appropriate.



There are many different schema forms for rendering patterns. At the more formal end of the pattern form spectrum, there is the template or schema style adopted by the Gang of Four. At the other is the more literary original Alexandrian (or Alexanderian) form and the Portland form, in which the majority of patterns in the Portland Pattern Repository are documented.

That a pattern has an indexing component, or key, is also an essential element: patterns would be useless if we had no way of referring to them either by their name or by some other more elaborate categorisation. The Gang of Four identified a useful set of purpose based categories for grouping and finding patterns: creational, structural and behavioural.

The use of examples is an important part of communicating patterns, and in describing a pattern it is often the case that a motivating example will illustrate it better than a model that speaks only in the most abstract terms.



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

Patterns can be identified and applied in a broad range of domains, highlighted especially by their origins in the architecture of the built environment before being adopted in software architecture.

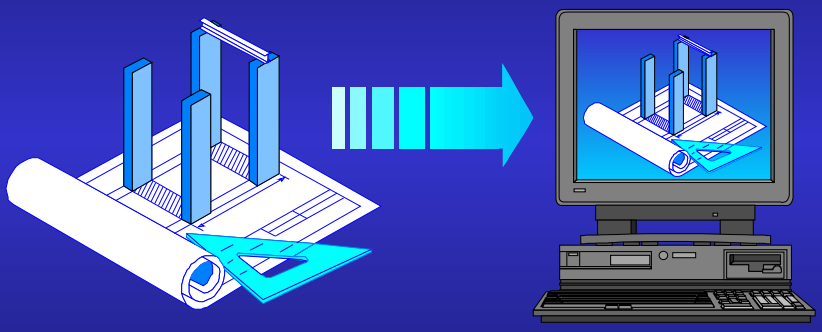
The particular patterns documented by the "Gang of Four" are general purpose, common and focused on OO design. In spite of this, there is a lingering perception by some developers that the patterns in their book are all that there are in this rich field, and that this level of design is their only context of relevance.

That this is not the case can be quickly dispelled by picking up any of the PLoPD books [Pattern Languages of Program Design volumes 1, 2, 3, 4 and 5], surfing [http://hillside.net], sifting the Pattern Almanac [Linda Rising] or browsing any number of development magazines and journals.

Exercises: Recurring Solutions

- Have you seen any familiar recurring solutions in your own code?
  - What problems did they solve?
  - What were the alternatives?
- Have you seen any other effective patterns in what you do?
  - For example, development process or practices
- Have you seen any patterns misapplied?

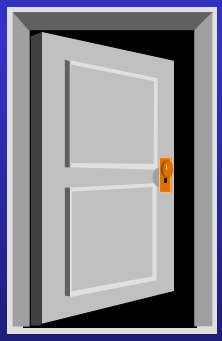
Introductory Pattern Examples



Pattern-Based Software Development

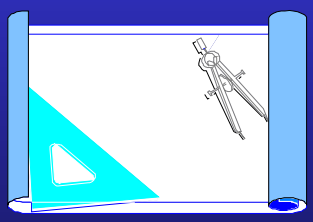
Introductory Pattern Examples

- Objectives
  - Introduce patterns by example
- Contents
  - General design patterns in OO
  - The Composite pattern
  - The Proxy pattern
  - Patterns beyond objects



General Design Patterns in OO

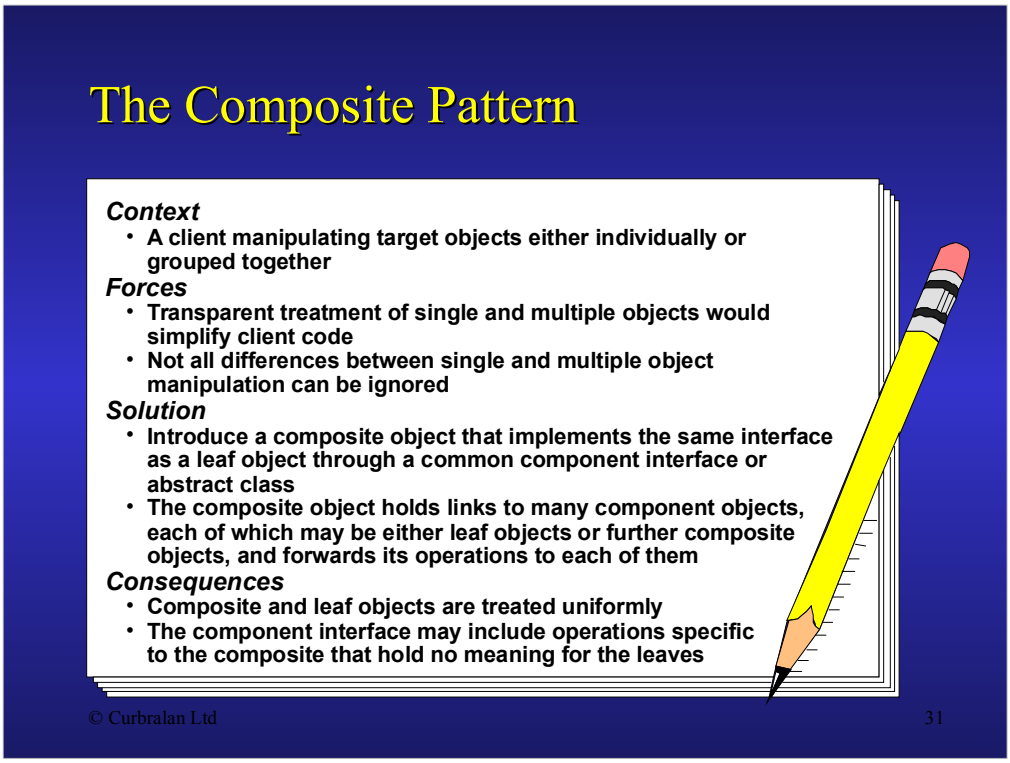
- General design patterns document common frameworks and application collaborations
  - Typically independent of programming language
  - Originally documented within the technical domain of OO, but many more have since been documented
- There may be alternative solutions to the same problem that are more or less appropriate
  - Depending on the trade-offs identified for each pattern



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. Therefore design is a creational and intentional act: the conception and construction of a structure on purpose for a purpose. Thus design embraces all levels of detail in the system, not simply the intermediate level of classes and their relationships. Although the terms design and design patterns have commonly come to mean general, intermediate-level design, it is perhaps better to distinguish this category by referring to them as general design patterns.

Such general design patterns may be used to understand the design of existing frameworks – where Carolyn Morris defines a framework as "a skeleton on which a model of work is built" – and libraries or a knowledge of patterns may be used directly in the construction or refinement of a framework or library. They address the problems at the micro-architectural level as opposed to either the language or the (macro-)architectural scale [Buschmann et al, Pattern-Oriented Software Architecture Volume 1].

The Gang of Four's work [Gamma, Helm, Johnson and Vlissides, Design Patterns] is an example of a catalogue of general-purpose design patterns. Although a number of the examples are based on GUIs, the patterns themselves can and have been found and applied across all manner of systems. Domain-specific patterns are those found within a particular sector of industry, e.g. telecoms, finance, etc. These are more detailed than the domain specific patterns one might find in analysis. A number of design patterns have as their applicability the context of a given architecture. For instance, some are particularly suited to the design, implementation and documentation of distributed systems.



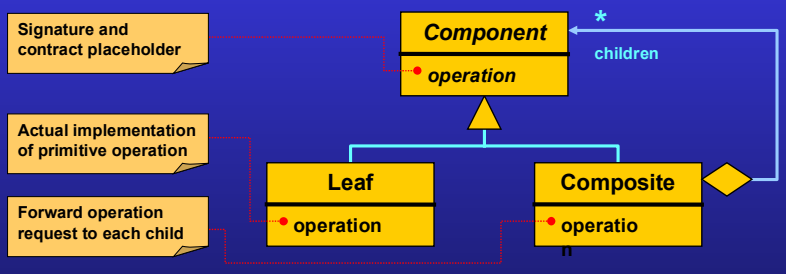
The Composite structural pattern captures the essence of the problem and resulting design. Its intent is to

> Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Composite is effectively a recursive variant of the Whole–Part pattern [Pattern-Oriented Software Architecture, Volume 1, Frank Buschmann et al].

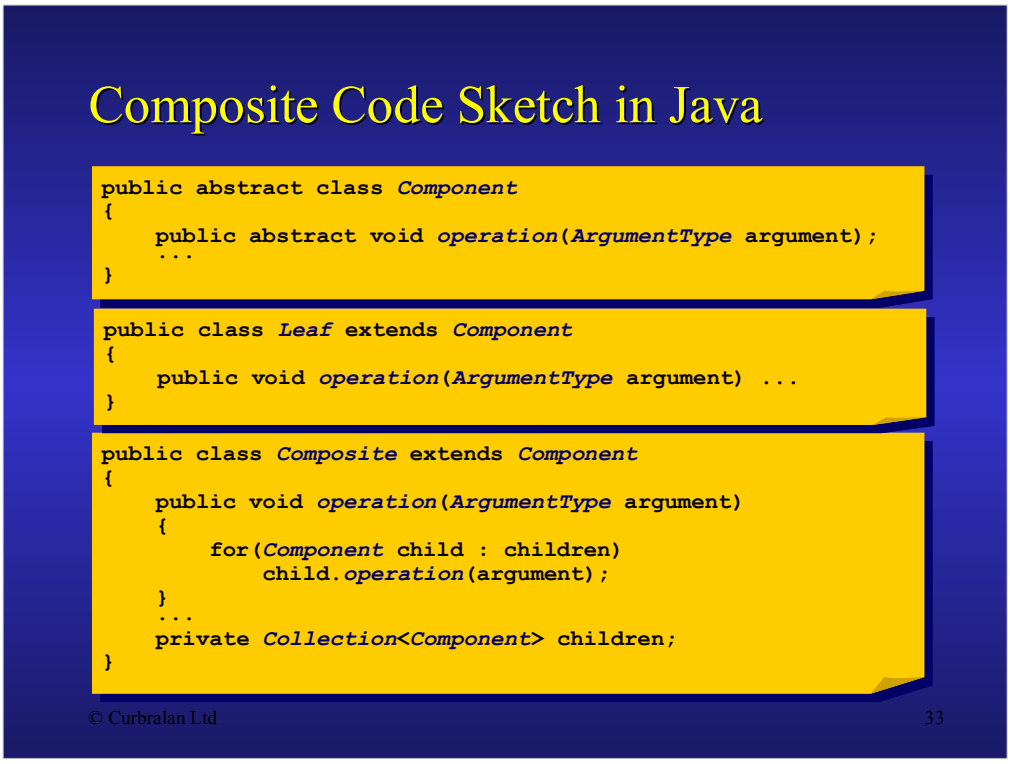
Typical Composite Configuration

- Common configurations for the pattern solution may be illustrated with diagrams
  - Sometimes a class diagram, but other diagrams and non-UML forms may be more appropriate



A significant consequences of Composite is that it may lead to the common superclass's interface being fattened in order to accommodate operations that are specific to either leaf classes or, more typically, the composite class in order to achieve uniform substitutability. This clearly creates a tension between the desire for uniform treatment of a collection of objects and the desire to have minimal, cohesive interfaces.

The Composite structure is one that is now often taken for granted: hierarchical directory structures, graphical grouping, organisational hierarchies, etc. However, its application is not always so obvious and immediate: it often requires clarification of terminology or refactoring an interface to see.



Patterns that are code-centric or can be realised easily and directly in code are normally accompanied with code examples or fragments of code that help the reader to understand the configuration of the pattern and its consequences.

It is important to note that while diagram and supporting code may be used to illustrate or support a pattern, they do not on their own constitute the whole pattern. Capturing the forces in the problem, the players in the solution, and the consequences of the solution, i.e. pros and cons, is essential in any pattern.

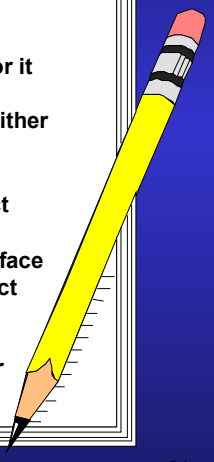
The Proxy Pattern

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

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

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

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



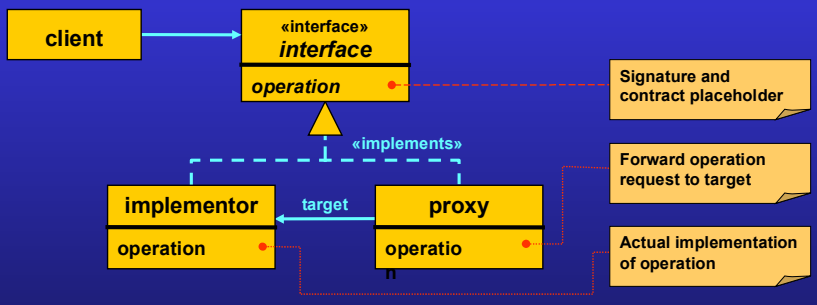
The use of a proxy object resolved the problem of transparent distributed procedural communication in the motivating example. A similar solution can also be found in other object systems, suggesting that there is a pattern that can be generalised. The intent of the Proxy pattern is given as

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

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

Proxy Configuration Sketch

- Proxy is typically described in terms of an interface and a class hierarchy
  - However, proxies are not just an OO concept



There are many variants of Proxy which can be considered specialisations of the basic pattern:

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

Patterns beyond Objects

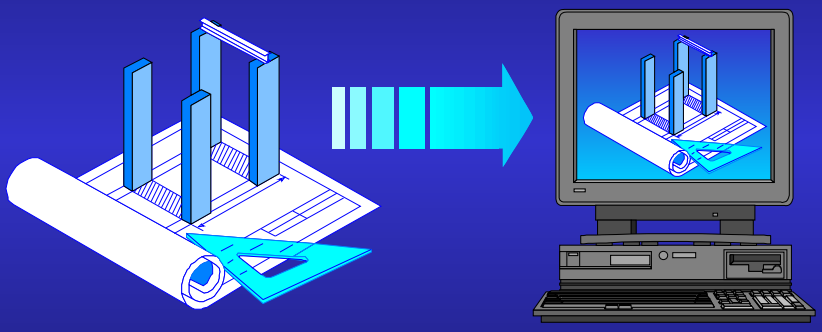
- Patterns are not just confined to OO design
  - They have been used to characterise distributed and component-based architectures
  - Analysis patterns focus on modelling problem domains
  - Organisational and development-process patterns have captured effective development practices
  - Many programming language specific idioms can be documented as patterns, with the programming language being their defining context
- Patterns are not just individual fragments of design
  - A whole design will typically follow many patterns

Patterns are not just about objects, or even buildings. They have been used to document many forms of architectural knowledge, large and small, where there needs to be some understanding of the forces and consequences involved in adopting one solution over another.

Organisational patterns for software development have been the focus of work for James Coplien and Neil Harrison [Organizational Patterns of Agile Software Development]. These patterns capture the essence of working collaborations between groups of people. For example, the pattern of Organisation Follows Architecture is also known as Conway's Law, although Conway's Law is a better name for the forces identified within the pattern. For a well established product it simplifies communication and change to align the structure of an organisation around the architecture of the product. Organisation Follows Market notes that organisations serving several distinct markets should structure themselves along market lines. However, to avoid setting up exclusive project fiefdoms that share nothing, the common features of all systems should be managed by a hub, framework team.

Martin Fowler [Analysis Patterns] describes a number of domain related analysis patterns. For instance, Posting Rule Execution describes the interaction between accounts, transactions and posting rules under the domain heading of Inventory and Accounting. The Quote pattern combines two or more prices (e.g. a bid and an offer price) into a single object which introduces a new fundamental type to the trading domain. Although there are clearly some commonalities with patterns found in design, the language and bias of these is very much of the problem rather than solution domain.

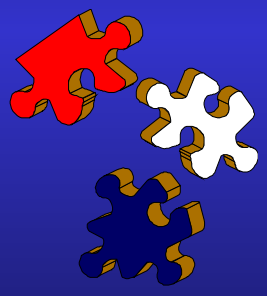
Combining Patterns



Pattern-Based Software Development

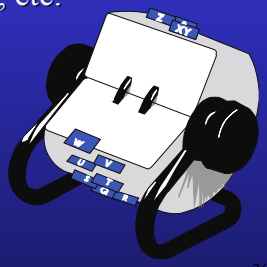
Combining Patterns

- Objectives
  - Demonstrate how patterns combine by contributing roles to a design
- Contents
  - Pattern catalogues
  - Pattern communities
  - Classes and patterns in JUnit
  - From individual to multiple patterns
  - The Visitor pattern
  - Pattern families
  - Pattern stories and languages



Pattern Catalogues

- A pattern catalogue holds individual patterns
  - Not necessarily interconnected
- Pattern catalogues may be organised by...
  - *Problem domain*: finance, telecoms, etc.
  - *Solution domain*: distribution, concurrency, N-tier architecture, RDBMS, Java, C++, C#, etc.
- Catalogues and other collections represent the most common presentation of multiple patterns



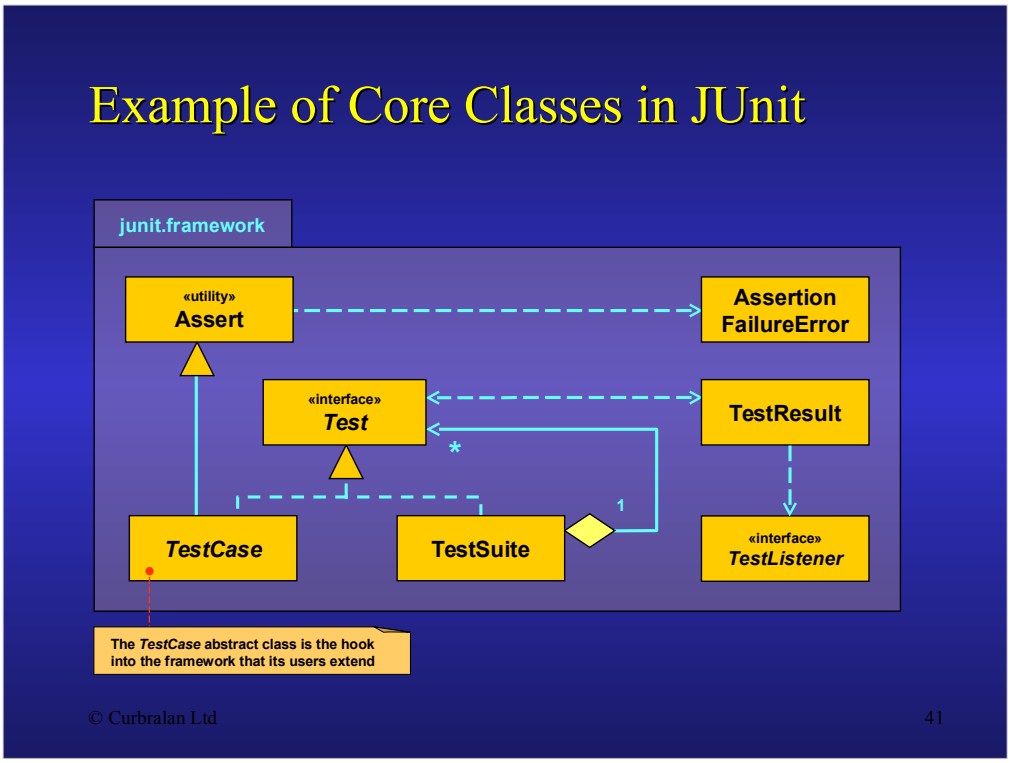
The Gang of Four's work is an example of a catalogue of general purpose OO design patterns. Although a number of the examples are based on GUIs, the patterns themselves can and have been found and applied across all manner of systems. The catalogue is structured in terms of pattern category (creational, structural and behavioural). Similarities are noted between some patterns, and some of the usage relationships are highlighted within the catalogue.

Domain specific patterns are those found within a particular sector of industry, e.g. telecoms, finance, etc. These are more detailed than the domain specific patterns one might find in analysis.

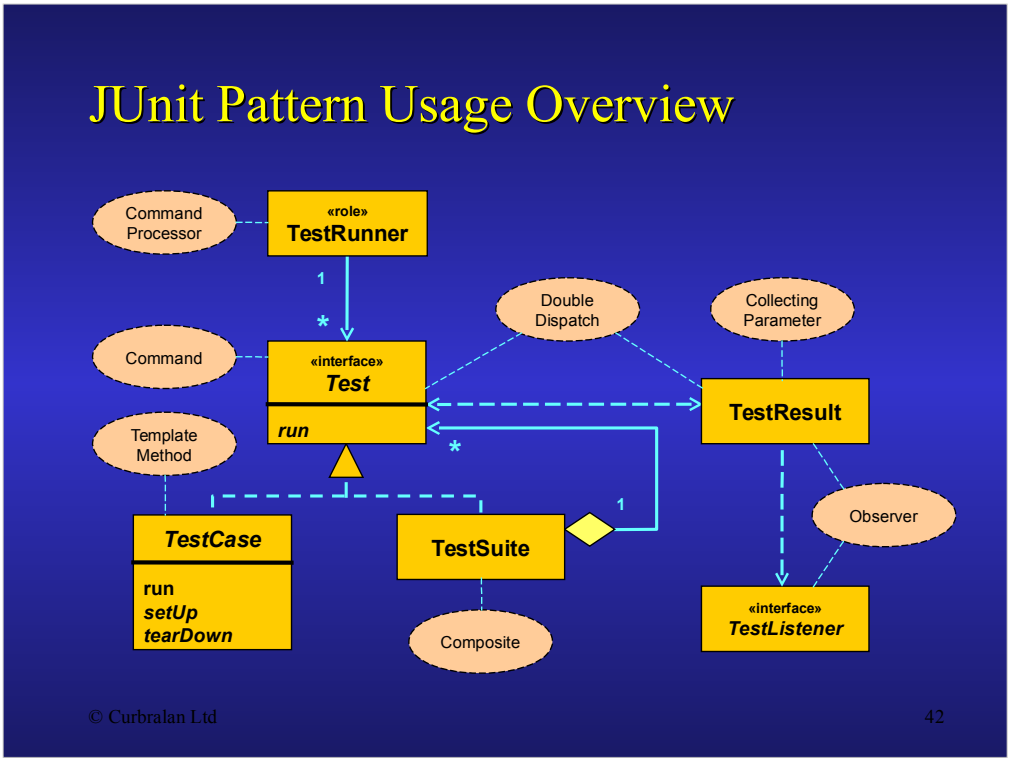
Pattern Communities

- Individual patterns can be used in isolation with some degree of success
  - Represent foci for discussion or point solutions
- However, patterns are in truth gregarious
  - They're rather fond of the company of patterns
  - To make practical sense as a design idea, patterns enlist other patterns for expression and variation
- Compound patterns (a.k.a. pattern compounds) capture common recurring pattern subcommunities
  - In truth, most patterns are, at one level or another, compound

No pattern is truly an island. Some patterns suggest the use of follow-on patterns. Other patterns employ simpler patterns as part of their make up. A compound pattern identifies the recurrence of a particular community of patterns as a whole, i.e. a pattern in its own right. Not all pattern communities can be considered, or are usefully considered, compound patterns, but the notion that some common designs are obviously composed of smaller patterns is a useful one.



The core of JUnit, an open source unit-testing framework for Java, is found in the `junit.framework` package. Other common extensions are found in `junit.extensions`. These are the only two packages that have proper, associated *javadoc* documentation. Other standard packages include the runner packages.



JUnit's design can be understood in terms of a number of common design patterns. The diagram omits some of the patterns described here. The description below is a sketch, further omitting method specifics and some collaborations to convey the general picture without getting lost in detail.

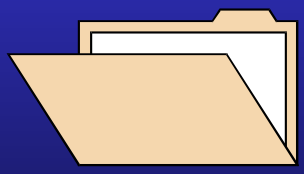
The core of the framework is characterised by a Composite that allows a `TestSuite` to both be a `Test` and hold many `Test` objects. The `Test` hierarchy is itself an implementation of the Command pattern, where an object is used to represent an action. Although there is no core runner type specified, its role is implied by the framework, and so has been shown in the diagram above. A runner acts as a Command Processor that executes Command objects, in this case instances of the `Test` hierarchy.

The main extension point of the framework is the `TestCase` class, which is abstract. This is the class from which user test cases will inherit, and is of most interest to JUnit users. `TestCase` has a Template Method that runs the `setUp` and `tearDown` Hook Methods, which may be overridden in subclasses. The `TestCase` class also has a Pluggable Selector in combination with Reflection to allow calling a named test method.

A `TestResult` is used to record the results of executing one or more tests. It is a Collecting Parameter that is passed in to accumulate test results rather than a new object that is created and returned as a result of executing a test. The relationship between `TestResult` and `TestCase`, as implied by the `Test` interface, uses Double Dispatch to effect execution: when a test runs it will call a `TestResult` instance to call it back and run actual the test. The Double Dispatch is used because multiple `TestListener` objects may be registered with a `TestResult` as Observers of the test exeution. They need to be notified of the execution and its results, and this is the responsibility of the `TestResult` and not the `Test` hierarchy.

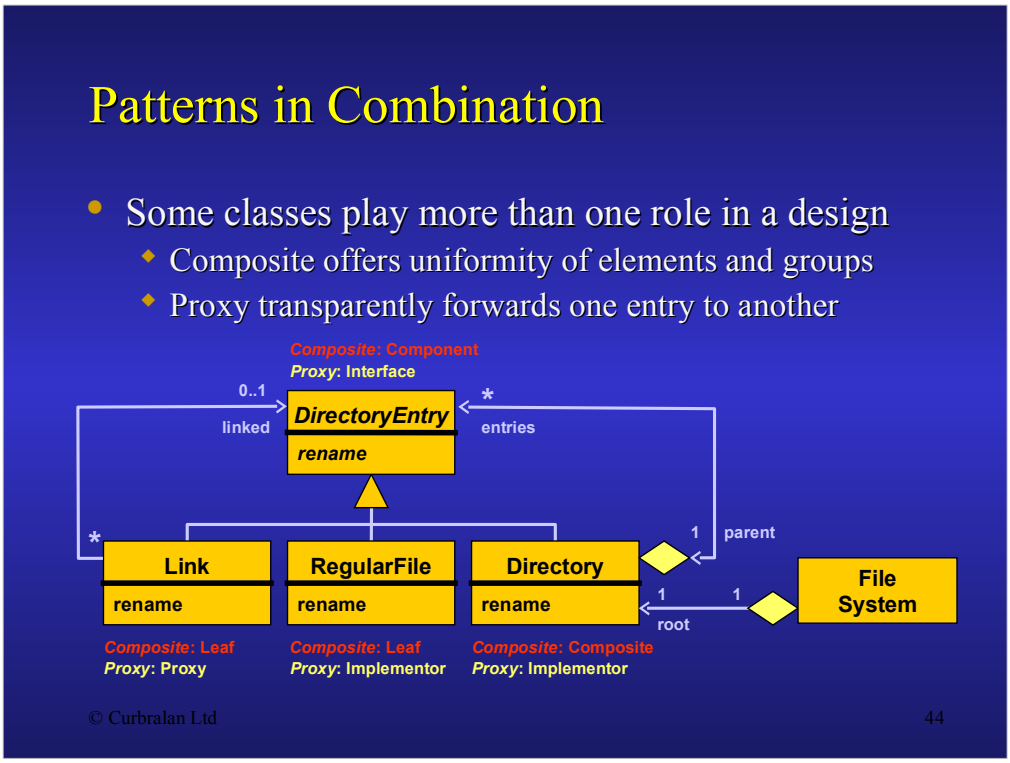
From Individual to Multiple Patterns

- Patterns are not just isolated fragments of design
  - A class can play different roles in different patterns
- For example, consider how a file system can be modelled with classes...
  - Assuming that directories may contain ordinary files, links and other directories...
  - Which belong to at most one directory...
  - And may be renamed, removed, moved or copied...
  - And the file system has a single root directory



The solution to this problem brings together a number of the notational concepts for both object and class relationships. The basic problem can be rephrased more generally as how to allow an arbitrary grouping of elements and at the same time uniform treatment of grouped and primitive elements. There is also the need to treat a link (also known as a shortcut or symbolic link) transparently as the file or directory to which it refers.

Remember that conventional file systems have a fixed root, but otherwise their files and directories can be moved around.



The essence of the solution is that groups and primitives share a common interface that defines operations that are common to all directory entries, regardless of whether they represent directories or regular files. The directory is thus substitutable for any directory entry. It holds a collection of directory entries, which may in turn be regular files or directories, and to which operations are delegated.

The salient features of this common problem and solution pair may be understood and generalised as the Composite pattern. The challenges in identifying this recursive object structure are finding appropriate names (i.e. file is too broad a term in that it ambiguously means directory entry or regular file, the exact meaning depending on conversational context) and the entry point for aggregation from outside, which is often the composite element itself, i.e. a directory.

In addition, a link acts as a forwarder for a directory entry, which is the Proxy pattern.

The Problem of Type Laundering

- Explicit type selection is often a sign of missing polymorphism
  - Brittle in the face of change

```
public void example(Component component)
{
    if(component instanceof Composite)
    {
        Composite composite = (Composite) component;
        ...
    }
    else if(component instanceof Leaf)
    {
        Leaf leaf = (Leaf) component;
        ...
    }
    else
    {
        ... // error!
    }
}
```


The use of `instanceof` to switch on an object's type to determine additional behaviour is a brittle and verbose approach that hints at missing polymorphism. The problem arises when the polymorphism is genuinely external to the class hierarchy of interest.

The Visitor Pattern

- Polymorphism can be used to extend behaviour outside the main class hierarchy
  - Double dispatch bounces a call from object to object

```
public interface Visitor
{
    void visit(Composite composite);
    void visit(Leaf leaf);
}
```

```
public abstract class Component
{
    public void accept(Visitor visitor);
    ...
}
```


The Visitor pattern, normally using the Double Dispatch pattern, allows behaviour to be extended in a relatively stable fashion using polymorphism in two hierarchies: the component hierarchy and a plug-in hierarchy of Visitors. Type laundering is automatic and castless:

```
public class Leaf extends Component
{
    public void accept(Visitor visitor)
    {
        visitor.visit(this);
    }
    ...
}
public class Composite extends Component
{
    public void accept(Visitor visitor)
    {
        visitor.visit(this);
    }
    ...
}
```

Note that to be effective, this pattern requires stability of the component class hierarchy.

Pattern Families

- Patterns that compete with one another can be said to form a pattern family
  - They differ slightly in the details of their forces, but often significantly in their structure...
  - But they share some degree of common intent
- Recognising the commonality helps to enrich a design vocabulary
  - Helps to avoid single-hammer syndrome

Many patterns can be seen as bound by a common core intent, and yet they differ significantly in the expression of their solution structure or in the fine detail of their forces. A common mistake in the pattern literature is to mistake a pattern family as a whole pattern, such as the Gang of Four's classification of the fundamentally distinct Object Adapter and Class Adapter as one pattern, Adapter, or the attempt to unify the distinct Iterator and Enumeration Method solutions under a single pattern, Iterator.

Pattern Stories and Languages

- Pattern stories capture examples of common sequences of pattern application
  - One pattern follows another to produce a given system or subsystem example
  - Pattern stories can illustrate example designs or design ideas
- A pattern language connects many patterns together in a more general fashion
  - From a pattern language can flow many stories
  - A pattern language can characterise an architectural style
  - Most pattern works have not matured to this level

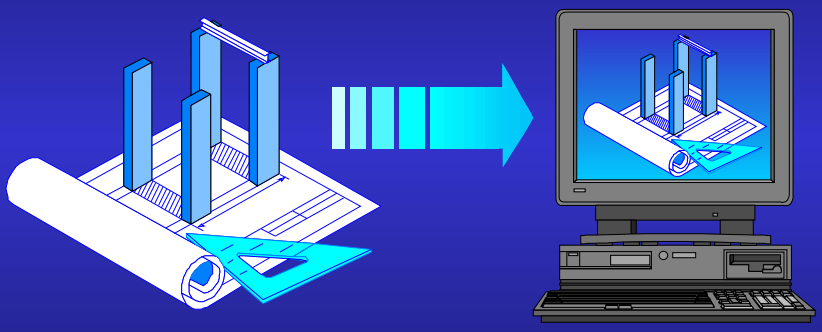
Any given design can be seen as an interlocking and flowing set of decisions — context leads to problem, is resolved by solution, establishes new context, which in turn... and so on. It is worth capturing such a flow as it outlines the elements and unfolding of a design.

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

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

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

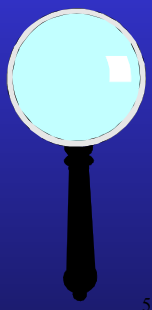
Pattern Context Dependency



Pattern-Based Software Development

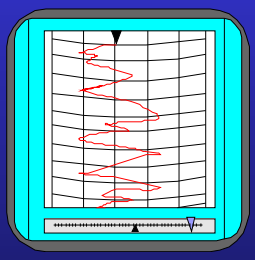
Pattern Context Dependency

- Objectives
  - Illustrate how patterns are driven and dependent upon the surrounding design context
- Contents
  - Context sensitivity
  - The Client Proxy pattern
  - Strategic and tactical patterns
  - Idioms
  - The Immutable Value pattern
  - The Combined Method pattern
  - The Data Transfer Object pattern



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
  - E.g. distribution, persistence, concurrency, asynchronous events

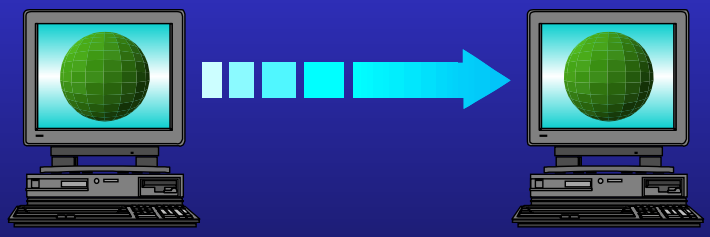


Design is not simply a feed-forward process into which an analysis is fed, a handle turned, and a suitable implementation spat out. There are those who maintain such a view, but close inspection of what they define as analysis reveals synthesis: construction detail and compromises relevant only to the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

Hence, the importance that good pattern forms place on the notion of establishing the context and forces that drive and support the proposed solution.

Remote Procedure Call Models

- The conventional call–response approach of in-process procedural programming can be extended
  - Hence, the concept of the Remote Procedure Call (RPC), which has also been extended to include object methods
  - Communication is initiated through proxies that are part of a middleware layer

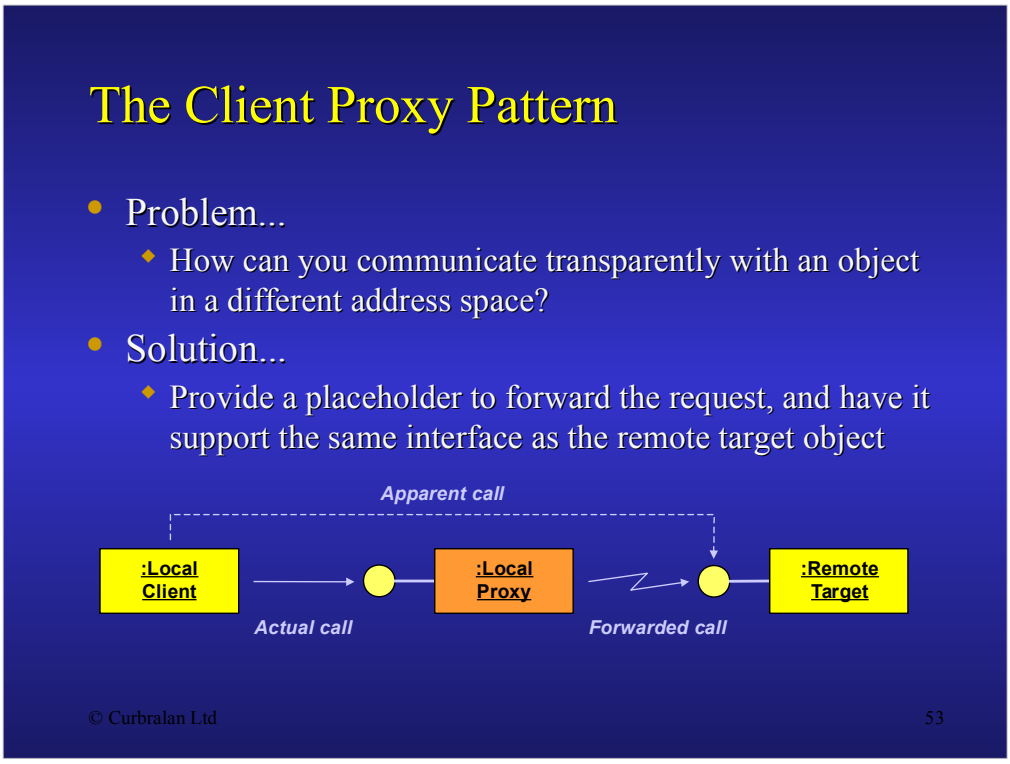


Distributed object computing (DOC) has evolved out of the work done on higher-level approaches to programming networks.

The RPC (Remote Procedure Call) model has offered a high-level, transparent, procedural mechanism for supporting calls to functions in remote processes. This level of structured messaging is above the ad hoc level of networking with sockets. The two key standards in this area have been OSF's DCE and Sun's ONC. However, the unit of a function is often too fine a one to meaningfully structure many systems. At the same time, object-orientation offers some attractive structuring principles that support distribution.

The aim of any distributed computing model is to provide a simple and uniform model for communicating with distributed units. In other words, to address the needs of development at a level appropriate to the application programmer, i.e. the application layer, rather than dropping down to a more systems programming level, e.g. socket programming at the transport layer. Code that mixes its levels in such a fashion is often difficult to interpret and closed to change.

Objects provide an ideal component for distribution: they are viewed as encapsulated communicating entities with public interfaces composed of operations. In a distributed environment the encapsulation of an object is complete, and not merely by convention as it is in typical programming environments. There is genuinely no access to the representation, and this is true down to the level of the actual platform of execution.



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

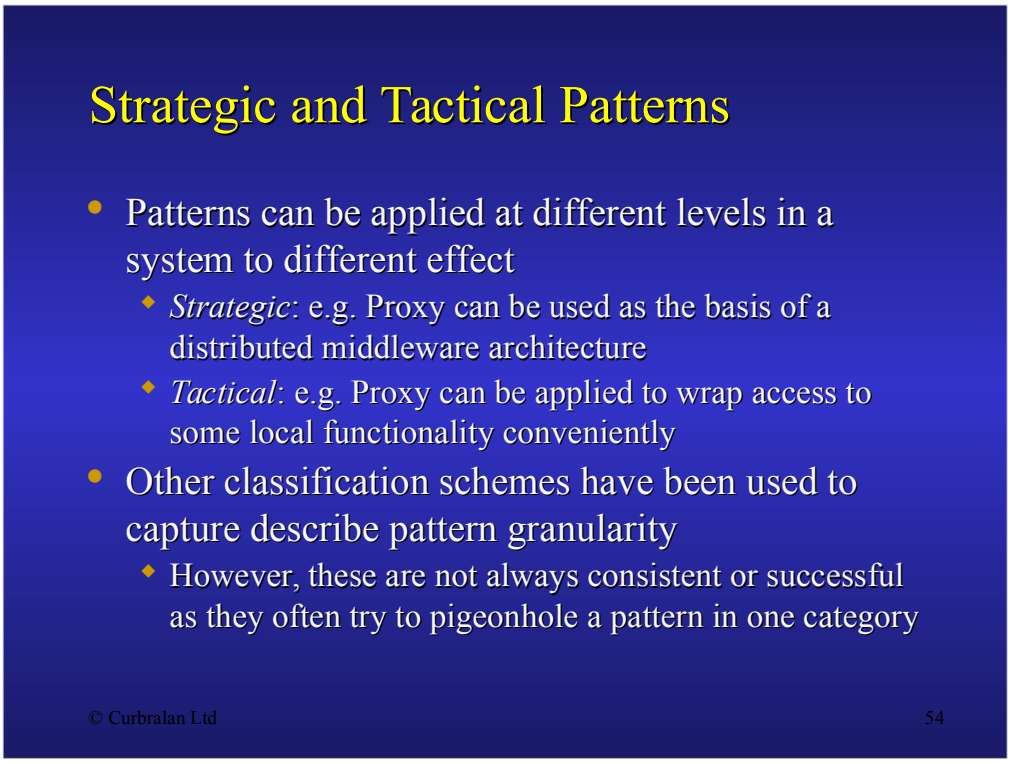
The essence of the solution is that an interface describing the desired behaviour of an object is implemented in two parts: one part is the concrete implementation; the other is the part responsible for any forwarding and translation. Each operation in the proxy forwards to the actual, associated target.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

We see this design solution directly in, and as a core part of, many distributed system implementations, including CORBA, RMI, COM and .NET Remoting. The principle of delegation is used to forward and translate messages across address spaces and representations. The alternative names Surrogate and Ambassador are also suggestive of the basic concept behind this pattern.

The use of a proxy object resolved the problem of transparent method calling between objects where an additional layer of control is introduced. The intent of the Proxy pattern [Design Patterns] is given as

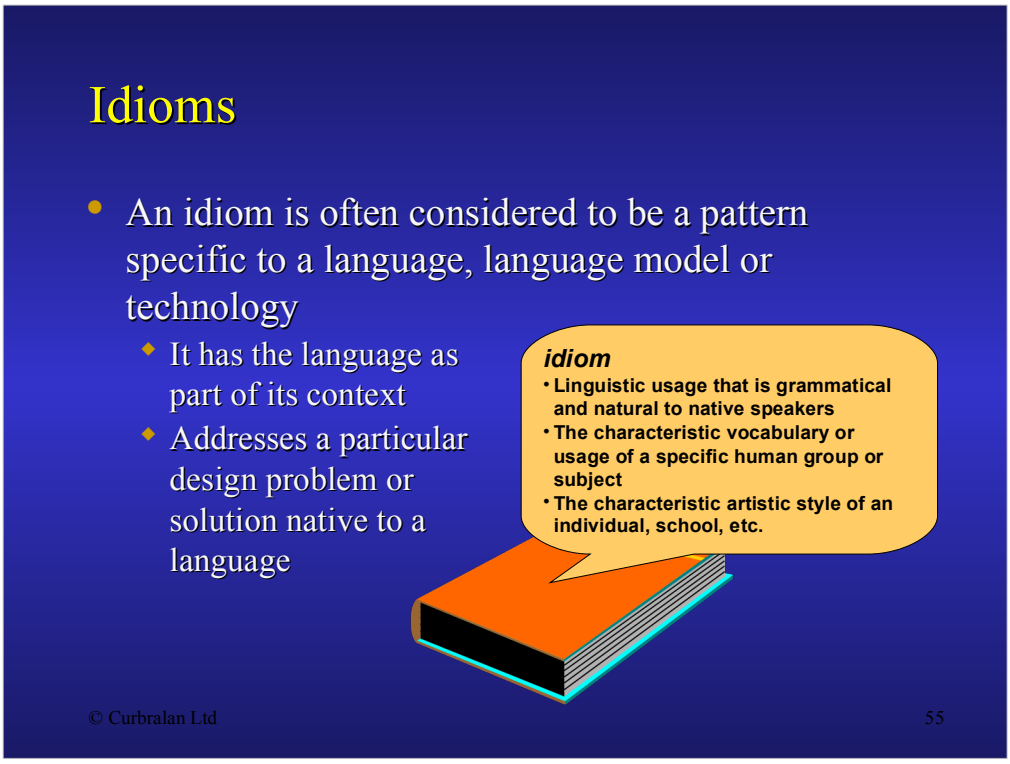
> Provide a surrogate or placeholder for another object to control access to it.

The alternative names Surrogate and Ambassador are also suggestive of the basic concept behind this pattern. The form above illustrates the key features of the pattern.



There are many ways of viewing patterns, and one of the earliest is with respect to scale: architectural patterns, design patterns and idioms. However, there are dual concerns over this classification. In terms of vocabulary, architecture is not just a macroscopic issue and design is not defined by level of detail. In terms of scale, some patterns cross these levels, and may be at one level or another depending on their use.

Viewing pattern applications, rather than just patterns, in terms of their strategic effect or their tactical effect is often more useful. The strategic–tactical metaphor can also be extended to embrace the idea of logistical and operational patterns.



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

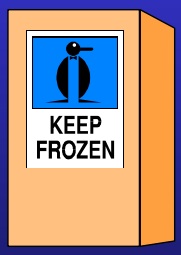
Some idioms, such as those for naming, can sometimes be transferred easily among languages. Others depend on features of a language model and are simply inapplicable when translated, such as between a strong and statically checked type system (e.g. C++ and Java) versus a looser, dynamically checked one (e.g. Smalltalk and Lisp).

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

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

The Immutable Value Pattern

- When objects are implicitly or explicitly shared aliasing causes problems
  - Copying must be implemented manually to ensure that changes are not unexpected
  - Methods must ensure they are synchronised if the object is shared between threads
- Therefore...
  - Don't have modifiable state!
- This simpler solution bypasses the problem and is a common Java idiom



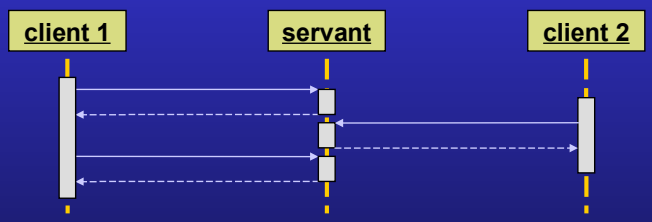
How can you share objects and guarantee no side effect problems? Various forms of copying can be used to to minimise aliasing issues. However, this can be error prone and may lead to excessive creation of small objects, especially where values are frequently queried or passed around.

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

The issues can be resolved by sharing an object that cannot be modified. The absence of any possible state changes means there no reason to synchronise. Not only does this make Immutable Value objects implicitly thread safe; the absence of locking means that their use in threaded environment is also efficient.

Concurrency and Method Calls

- Synchronisation is normally required to ensure consistent and coherent state change and access
  - This mismatch in granularity and coherence cannot be handled adequately a calling client
- The opportunity for lock-free code is not always present or easy to take advantage of

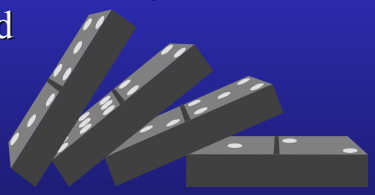


Concurrency introduces a design context unfamiliar to many developers, and one fraught with subtleties. If the consequences of distribution are not fully appreciated (i.e. the developer must genuinely grok them rather than pay lip service to them), this subtle design context becomes a subtle debugging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

Property-style programming, whether through the use of an actual property mechanism (e.g. as in .NET) or simple operations relating to attribute-like values (e.g. paired `get` and `set` operations as found in the JavaBeans model), has become popular in recent years. Newcomers to OO tend to design interfaces that have just enough encapsulation, i.e. each instance variable has corresponding set and get method. The result of this approach can be generously described as structification. A sequence of property operations may 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.

The Combined Method Pattern

- Choose a coarser granularity for object operations to solve coherence issues
  - Aligns and groups the unit of failure, synchronisation and common use
  - Can improve the encapsulation of object use, isolating the client from unnecessary details
- Therefore, the presence of concurrency affects the design of class interfaces and the approach to partitioning methods

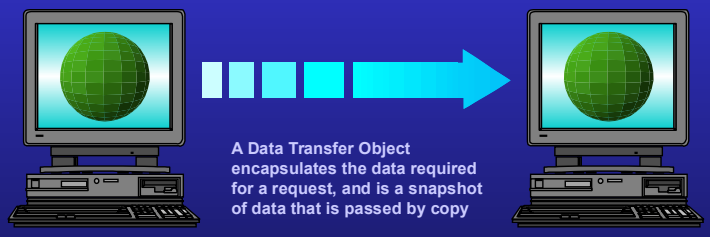


The presence of concurrency should be made as transparent as is reasonable. The user should not be concerned with having to manage synchronisation locks externally. The simplistic solution is to adopt an explicit acquisition–release approach to all interactions, which certainly requires visible change to user code and is not a relevant model for sequential systems, and is therefore anything but transparent.

For large and complex operation sequences involving the use of many objects, a transaction processor is appropriate; for small common operations on a single object, a transaction processor is overkill.

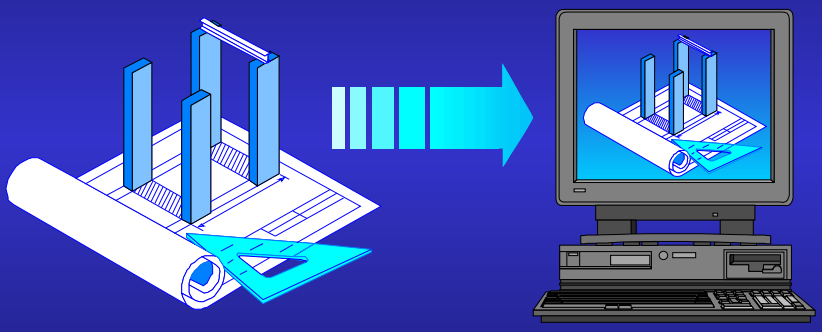
The Data Transfer Object Pattern

- Encapsulate data used for requests and results in an object that is copied across the network
  - Therefore, clients work with a copy of value data for and from a request, rather than with the actual remote objects that provide the various values
  - Also a threading solution, often with Immutable Values



The Data Transfer Object pattern can be seen as an alternative to the Combined Method, or even as a variant expression of the pattern. In essence, the Data Transfer Object represents an argument list and call context as a single object that is copied from one tier to another.

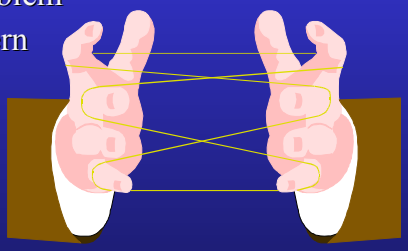
Patterns for Decoupling



Pattern-Based Software Development

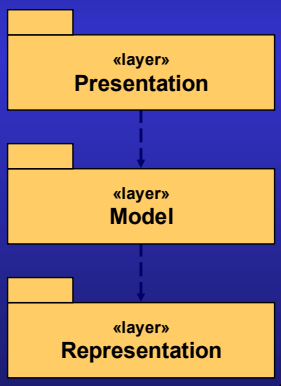
Patterns for Decoupling

- Objectives
  - Present some issues with coupling in a system along with a number of solutions for loosening coupling
- Contents
  - The Layers pattern and variations
  - The Fragile Base Class problem
  - The Explicit Interface pattern
  - The Separated Interface pattern
  - The Bridge pattern



The Layers Pattern

- Layers encapsulate different levels in a system with respect to one or more of...
  - Levels of abstraction
  - Rate of change
  - Development skills
  - Technology
  - Organisational structure
  - Geography
- Layers can be expressed in terms of packages, components or less formally



The Layers pattern [Buschmann et al, Pattern-Oriented Software Architecture Volume 1: A System of Patterns] 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 both from full encapsulation and from strict assignment of roles to layers (as opposed to a hierarchy 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. Layers may be strict or non-strict, but never cyclic.

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.

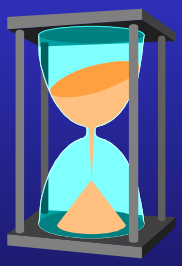
Porous Layers

- The layering in some architectures can be too laissez faire
  - The layers are porous because low-level or API-specific types bubble up to the higher layers
- Coupling between presentation and application logic should be reduced
  - Concepts internal to the representation layer should not be used or exposed higher up
  - Technology separation improves testability and adaptability

The problem of porous layering is a common one in architectures that are only layered in gesture but not in effect. For example, the use of Win32 API types in the business layers is a common problem. The concepts being described do not depend on the API, and yet they hook into its types and mechanisms. Regardless of portability (not always an immediate concern for many systems), porous layering has the effect of increasing build times and complicating testing. Something as simple as #include <windows.h> in an inappropriate layer can be a nuisance.

Time-Ordered Layering

- 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
- Different parts of a system are subject to different rates of development change
  - Layering should respect such change
  - This is something that can be monitored over a code base's lifetime, and the code refactored accordingly



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, explored by Alan O'Callaghan, is based on the idea of shearing layers of change in buildings. Stewart Brand, in How Buildings Learn, 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).

Without taking it too far, we can draw the analogy in software between the essentially stable parts of a system, i.e. the core business and data of a system, and the more volatile parts, such as the presentation layer.

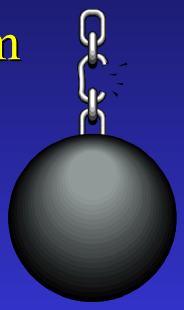
Boundary–Controller–Entity Layers

- This architecture represents a layered use–case-driven separation of concerns
  - A *boundary* object acts as an interface — visual or otherwise — between actor and system
  - A *controller* expresses the behaviour of a use case
  - An *entity* represents an artefact from the domain information model affected by the actions of a use case
- Creates onion-ring rather than stacked layering

A useful partitioning, driven from the perspective of combing a use case model with that of an informational domain model, is to provide objects at each of the conceptual areas of concern:

- Boundary objects manage the behaviour at the edge of the system, i.e. between the system and an actor. They may be visual or they may be wire-level wrappers.
- Controller objects articulate the behaviour and flow of a use case, acting as a bridge between the outside world and the conceptual domain objects on the inside of the system.
- Entity objects represent the artefacts of the domain, i.e. the 'stuff' of the system, that are affected by the actions described in use cases.

The Fragile Base Class Problem



- A subclass has a strong dependency on its superclass
  - Not as fully encapsulated or decoupled as delegation-based relationships
  - Inheritance is essentially the strongest form of coupling in an OO system
- Superclass evolution is a problem...
  - Release-to-release interface compatibility
  - Private changes lead to public builds
  - Subclass semantics when superclass is modified

There is a strong structural dependency upward from sub- to superclasses: a subclass cannot be defined without full reference to its superclass. Modifications to a subclass can therefore lead to something of a ripple effect especially if implementation is being inherited. This is known as the fragile base class problem.

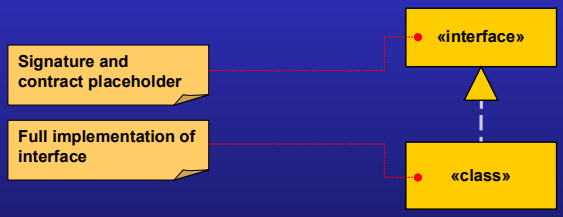
Clemens Szyperski classifies two different aspects of the problem: the syntactic fragile base class problem and the semantic fragile base class problem.

The syntactic problem is concerned with source code compatibility effects of modifying a base class. For instance, the use of protected is often questioned, especially for data. This creates even stronger dependencies between sub- and superclasses. In systems that have wide distribution, such as frameworks, this is almost equivalent to making the data public! The use of protected should be considered carefully, with ordinary instance data considered the least acceptable feature for this visibility. Java programmers should also be aware that protected also offers package visibility and not just subclass visibility of a feature.

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

The Explicit Interface Pattern

- Reduce client dependencies on non-public features in a package, subsystem or class hierarchy by...
  - Separating pure interfaces from classes
  - Providing appropriate creational facilities
- Fully decoupling usage type (interface) from creation type (concrete class) reduces dependencies

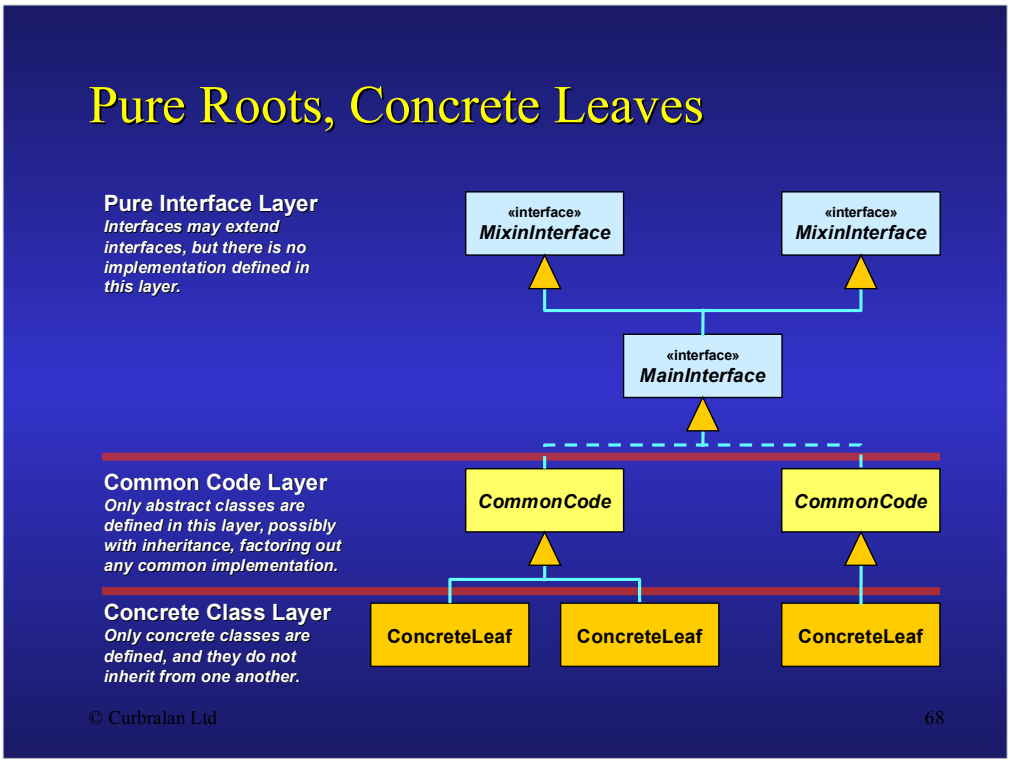


It is often the case that developers slap an interface onto an implementation as an afterthought. Such an approach stems from the view that an interface does not actually do anything, and therefore procedural code is more significant both in terms of its effect and bulk. Hence less effort is invested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

Although an interface requires far 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's 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 as failures of a component to satisfy an interface they can be fixed without adversely affecting clients. Modifications to poorly designed interfaces, however, will break clients that have been written against (and have worked around) them; such changes will be seen as causing problems rather than fixing them.

A client can only use the public code of a class, but a class definition typically includes more than simply the public usage interface. This can lead to unnecessary dependencies on users of a package. Explicit Interface introduces a 'vertical' separation between the usage type of an object, i.e. interface, and its actual underlying class.

The knowledge of the actual class is required in only one place, i.e. at the point of the creation, leaving all other usage code unaffected. Note that introducing any separation in structure must also be balanced by appropriate consideration of creation issues.



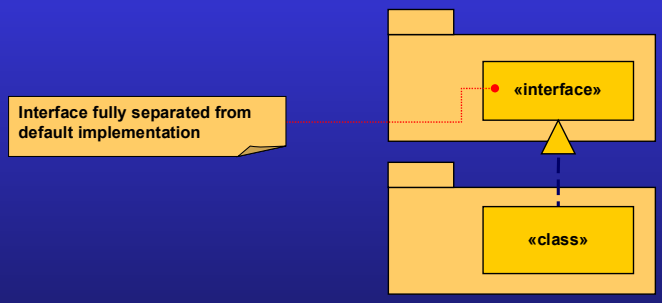
The advice to avoid inheriting from concrete classes and ensure that the tree of inheritance has concrete leaves only is not new, but it can be taken further: make the root of the hierarchy fully abstract, i.e. in languages that support it, use an interface at the root, and for languages that don't, ensure that no data is defined and only methods that are abstract and public are declared. The root of the hierarchy becomes far more stable as there is no implementation that can change. It is clearer because it is uncluttered by incidental default implementation guesses. If you want to provide some kit parts that make further derivation easier, they can be provided as a variety (not necessarily just one) of abstract classes the next level down.

This consideration is a relatively easy one to put into practice: it is easy to spot a hierarchy without a fully abstract base, and trivial to refactor — the Extract Interface refactoring that fits the bill precisely.

Perhaps more contentious is the consideration that a non-abstract method should not be overridden. In other words, once defined a method is never overridden, and code is never uninherited. Although not always practical, this practice has the benefit of making class hierarchies more loosely coupled and simpler to understand.

The Separated Interface Pattern

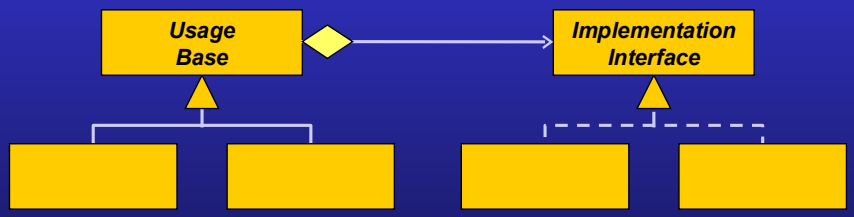
- The next step beyond making an interface explicit is to separate it from the implementing classes
  - This allows interface and implementation to be published, deployed and implemented separately



There is a tendency to assume that packages must necessarily contain concrete classes. However, a package (or other clustering unit) can serve many purposes in a system. The Separated Interface takes the separation of interface from implementation class a step further by promoting the separation at the subsystem or deployment level, so that interfaces are packaged and focused on quite separately from any implementation packages. This loosening of coupling is essential for plug-in architectures, but is also effective in other systems.

The Bridge Pattern

- How can different usage models and different underlying implementations be combined?
- Separate usage and implementation variations into two separate hierarchies
  - One represents the part expressing the domain concept, the other represents the implementation variations

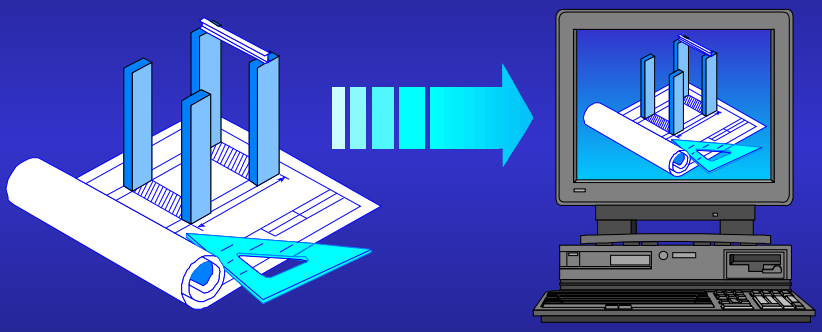


The intent of the Bridge pattern [Design Patterns], also known as Handle–Body, is given as

> Decouple an abstraction from its implementation so that the two can vary independently.

Variation in usage that is independent of implementation details, which may also vary, is catered for by two connected hierarchies, one delegating to the other. The Bridge separation is also applicable when there is no such runtime variation, and there is only a concrete handle class and a concrete body class and some other need for variation and control.

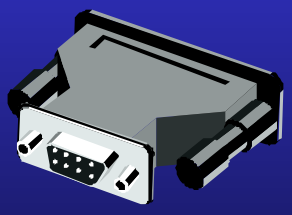
Patterns for Adaptation



Pattern-Based Software Development

Patterns for Adaptation

- Objectives
  - Present a number of patterns focused on adapting classes and objects to specific uses
- Contents
  - The Object Adapter pattern
  - The Class Adapter pattern
  - The Wrapped Adapter pattern
  - The Decorator pattern
  - The Template Method pattern
  - The Facade pattern



Adapter Patterns

- Problem...
  - How can a class be reused when it does not match a required or expected interface?
- Solution...
  - Define a new class that adapts the original class
- A common motivation brings two separate patterns into competition
  - The Object Adapter pattern, based on delegation
  - The Class Adapter pattern, based on inheritance

Wrapping one object up to be treated as another object is one of the commonest forms of forwarding in OO systems, and is sometimes seen as the most practical form of reuse, i.e. adapt existing classes to fit within a framework rather than bend the new classes to fit the old. The motivation for this varies — e.g. use of unstable third party code, legacy code, layering — but it is not always necessary to adapt existing code to new when the existing code forms the application framework, e.g. adapting Swing is non-trivial.

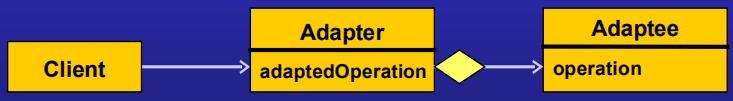
The Gang of Four consider there two be one adaptation pattern, Adapter, which has two main variants, Object Adapter and Class Adapter. The common intent is given as

> Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

However, the difference in resulting structures, applicability and trade-offs of the two variations stretches the notion of them as conjoined patterns to the limit. They are profoundly different patterns that share common motivation, but are otherwise in competition as distinct design solutions, not simply variations on a cohesive theme.

The Object Adapter Pattern

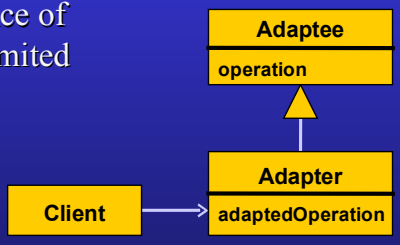
- Interface mismatch between classes can be resolved by adding an additional object
  - Adapter object supports relevant interface
  - Adapter object forwards operation calls as necessary to underlying adaptee object
- Two forms of adaptation that can be offered...
  - Syntactic adaptation and semantic adaptation



An Object Adapter aggregates the adaptee object and forwards requests to it as necessary. This may also involve some additional effort in translation, for instance where there is no one to one correspondence between adapter and adaptee methods and dependent data types. Whether the adapter is explicitly given the adaptee at runtime, i.e. as an argument when the object is created and initialised, or whether it fully hides the existence of the adaptee, i.e. it creates it on initialisation, is a decision the developer must make. In the first case shared aggregation is potentially the result and the creation of the adapter object is no longer fully transparent with respect to what it is adapting, although it can be used more easily in adapting class hierarchies. In the second case composition is handled by the adapter, the existence of the adaptee being fully hidden from the user.

The Class Adapter Pattern

- Using inheritance to adapt the adaptee...
  - Supports overriding of operations
  - Allows access to *protected* features
  - Results in a single object
- However...
  - Difficult to use if inheritance of implementation is either limited or not supported
  - Typically makes adaptee features *public*

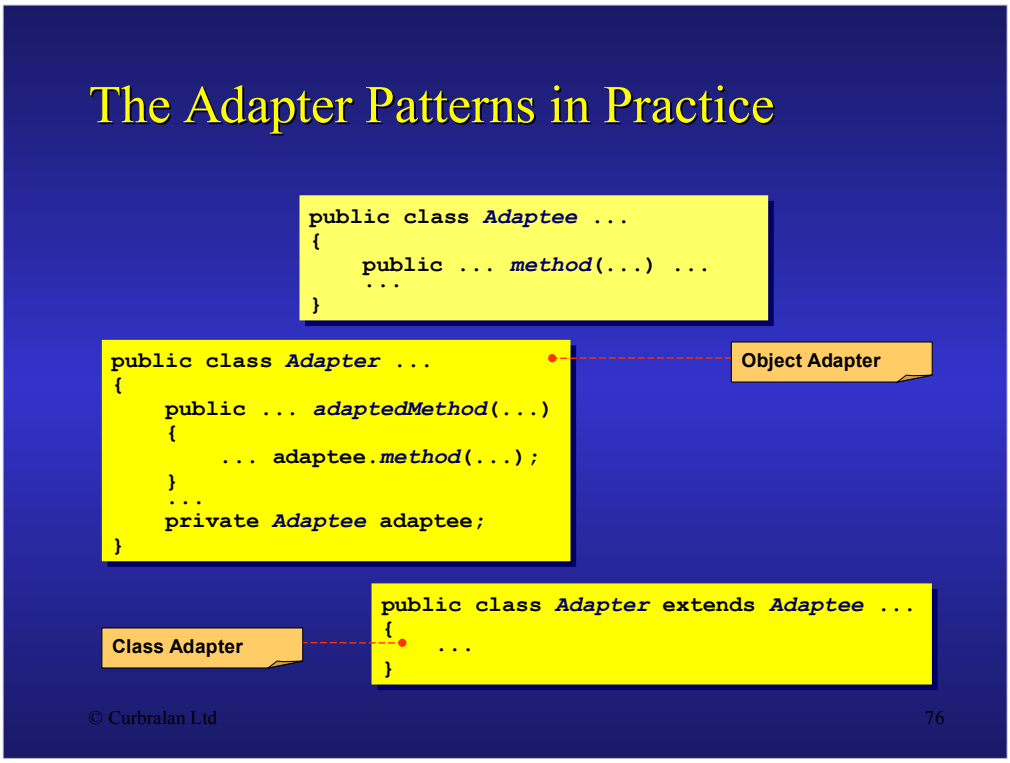


The Object Adapter form is by far and away the most commonly recommended adaptation pattern, but it still has limitations:

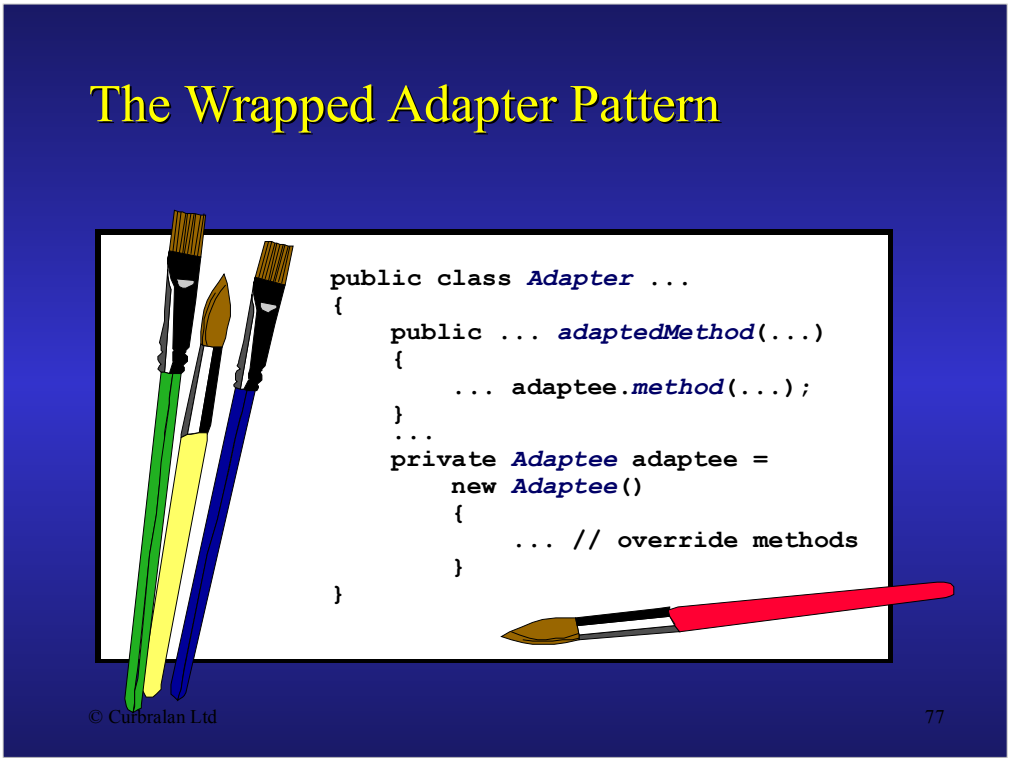
- Forwarding the operation calls, especially if they are almost one for one and many of the names are the same, can be tedious.
- If the adaptee needs to know its identity, the link must be bidirectional.
- If an operation in the adaptee class needs to be overridden, for instance if it is an abstract class, this must be done outside the adapter class if at all, which further undermines the transparency of the solution.
- Any protected operations and attributes that need to be accessed cannot be.

The first case is a matter of convenience, and might be considered one of those facts of life; bending designs to save a little typing at the potential expense of clarity and encapsulation is rarely a good idea. The second is more serious as it increases the complexity of the code. However, the third and fourth issues may well prove to be show stoppers.

A solution to this is to use inheritance, so that the adapter class inherits from the adaptee class. Inheritance means that at runtime only one object will be supporting adapter and adaptee roles, whereas an Object Adapter potentially results in two separate objects. However, as a compile time relationship, the inheritance means that the adaptation cannot be modified at runtime. Two fundamental issues with Class Adapter of which developers should be both aware and wary: increased coupling and potential violation of the substitution principle, because the derived class cannot uninherit undesirable functionality from its base class; limitations on the use of inheritance of implementation.

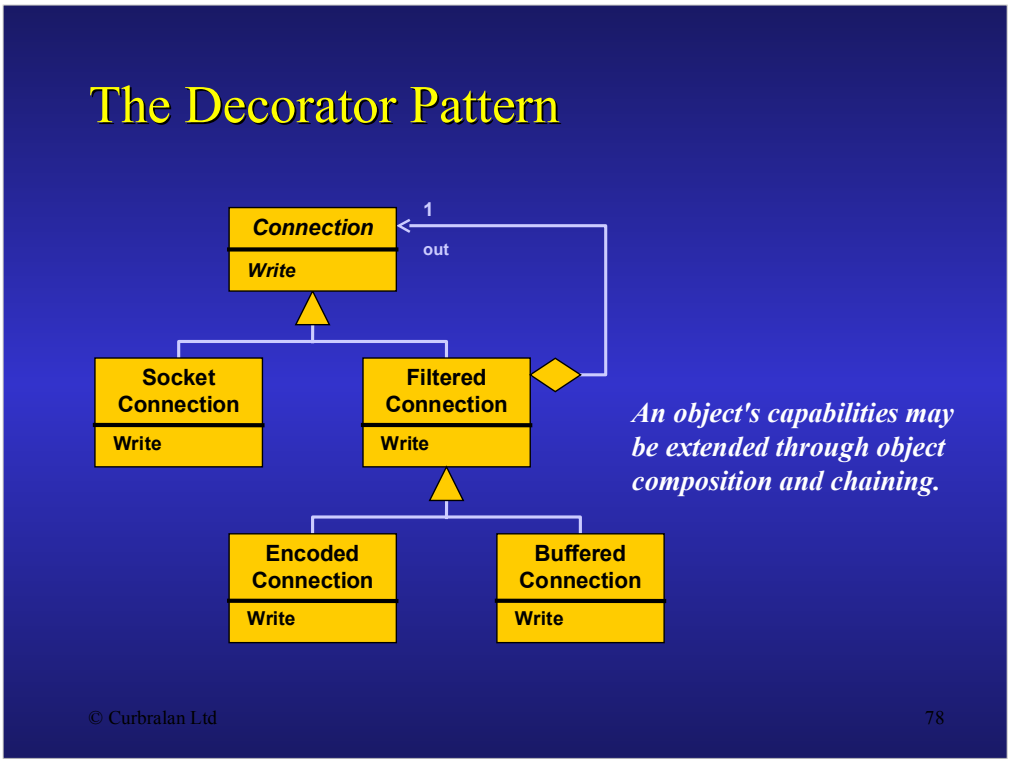


The two Adapter patterns examined have quite distinct appearances and trade-offs in code. Importantly, the temptation of the Class Adapter pattern is that it appears initially more economic, but this belies its problems.



Java allows an alternative Adapter variation which has many of the capabilities of Object Adapter and Class Adapter, but without Class Adapter's potential drawbacks. It is a combination of both Object Adapter and Class Adapter made convenient through the use of inner classes:

- Externally the adapter appears full encapsulated, as methods are forwarded as necessary to an adaptee object, i.e. this aspect is that of Object Adapter. This means that inheritance of implementation is available for other uses, and the interface of the adaptee does not interfere with that of the adapter.
- The adapter–adaptee relationship is that of composition: The adapter creates the adaptee object, does not share it or pass it out, and is notionally responsible for its termination. This means that a Wrapped Adapter is exclusively a compile-time wrapper rather than a runtime wrapper (although clearly some variation is possible through the use of another layer of adaptation).
- The adaptee is itself an adapter — a Class Adapter — inheriting from the true class that is to be adapted. Any method overriding and protected access is supported as necessary. Because this class is an inner class it also has the benefit of access to the main adapter's methods and fields.



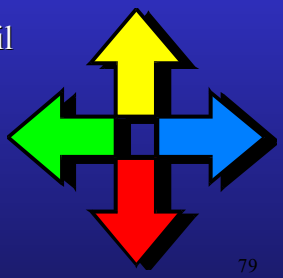
The intent of the Decorator pattern is given as

> Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

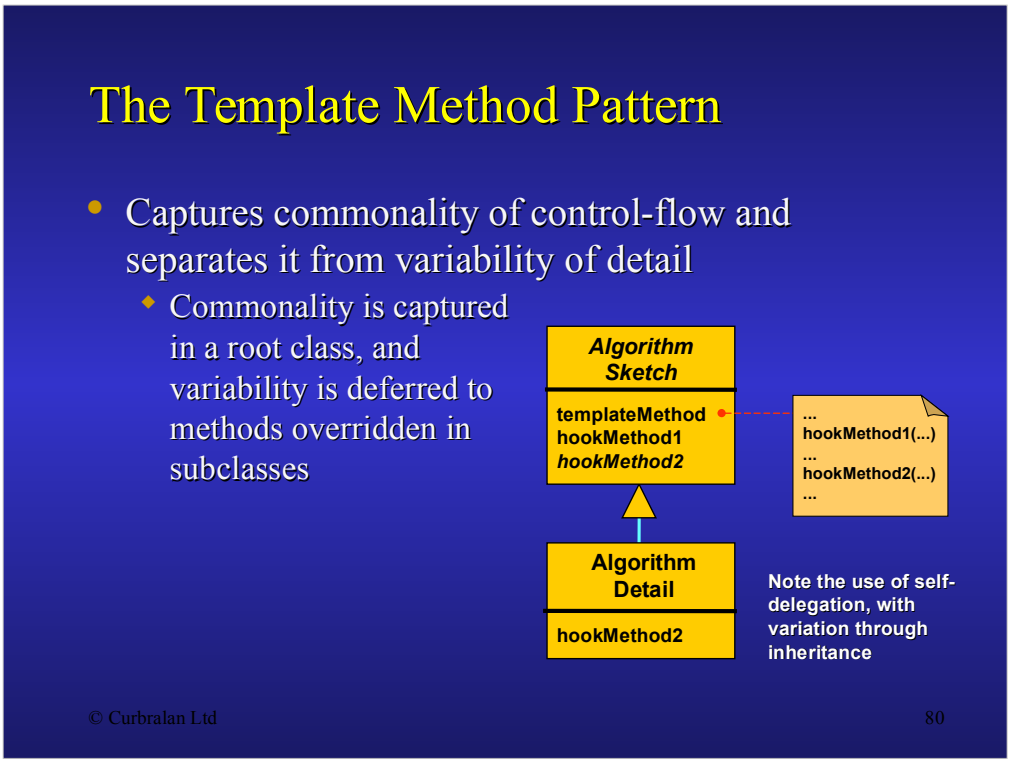
By forming a chain of objects conforming to the same interface, each adding something to a method invocation before passing the call on to the next in the chain, behavioural extension can be achieved through object composition. This can be more flexible and lead to more rapidly assembled alternatives than an equivalent inheritance-based approach. The delegation-based approach more clearly encapsulates each responsibility, leading to higher composability and testability.

Proxy, Decorator or Object Adapter?

- There can be confusion between many of the common patterns that are based on delegation
  - Similar structure and use of forwarding, but sometimes subtly different intent
- But there are sufficient differences to distinguish one pattern from another...
  - *Proxy*: same interface and hidden detail
  - *Decorator*: same interface with extras
  - *Object Adapter*: different interface



Many patterns have a similar appearance in terms of their class structure, and this can cause confusion. However, what distinguishes one pattern from another is not just a matter of structure, but also a matter of purpose. It is important to know the intent of a pattern to make the most sense of it.



The pattern is based on polymorphic delegation to oneself; in effect, implicit forwarding through `this`. The structure of an algorithm is inverted, so that the high level view of the algorithm is sketched in the superclass, which is typically abstract, and in the deferred functionality implemented in the subclass.

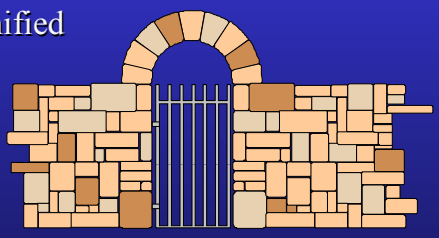
The polymorphism works within the class so that the primitive operations of the Template Method are typically `protected` rather than being made available for public use (although there are cases when this makes sense). The Template Method itself should often be declared `final`: it is difficult to override correctly, both preserving the integrity of the class and avoiding any attempt to *uninherit* functionality.

A dependency between the sub and superclasses is introduced. It is based on contractual obligation: the subclass is coupled directly to the superclass through inheritance; the superclass is coupled to the implied behaviour of the subclass.

The class author can provide default implementation in the superclass, but excessive use of this capability makes class hierarchies hard to understand and change rather than more convenient.

The Facade Pattern

- Problem...
  - For broad requirements, a sufficient interface can sometimes be quite complex to use and understand
  - Common tasks should be simple, hard things should be possible
- Solution...
  - Provide a higher-level unified interface to set of more intricate interfaces



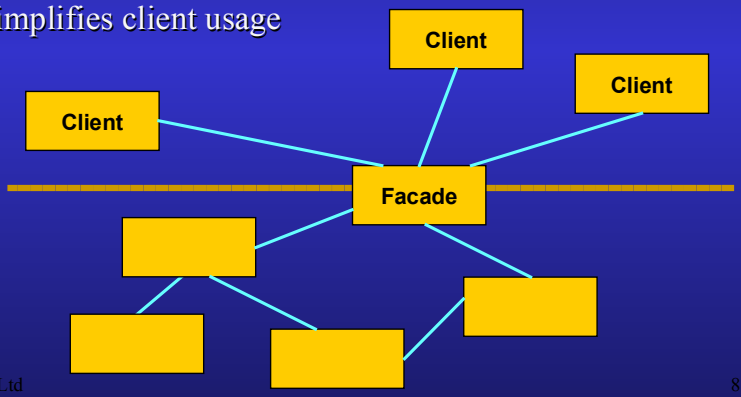
A subsystem, or set of subsystems, may well provide complete functionality to address a particular problem, but the framework provided may be difficult to use because of its comprehensiveness. The intent of the Facade pattern is given as [Design Patterns]:

> Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

In other words, there is a two-tier usage model: the common and simpler interface, as well as the less common but more powerful interface. The power-user option is what differentiates this pattern from the idea of providing full subsystem encapsulation.

Facade Objects

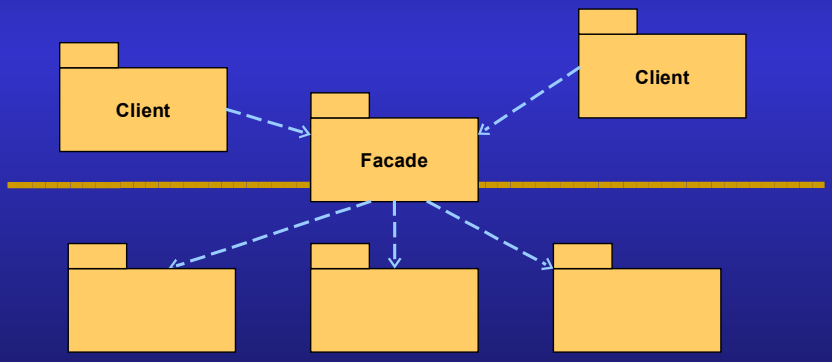
- A Facade object acts as a co-ordinator for a community of objects
  - Does not necessarily encapsulate them
  - Simplifies client usage



The Facade pattern, as originally described, is an object structural pattern. It simplifies usage of object communities by providing a single 'gatekeeper' object. This object takes on responsibility for controlling the other objects, often encapsulating a particular model of usage.

Facade Packages

- A Facade package acts as a simplifying wrapper for one or more other packages
  - Simplifies client usage from an API perspective



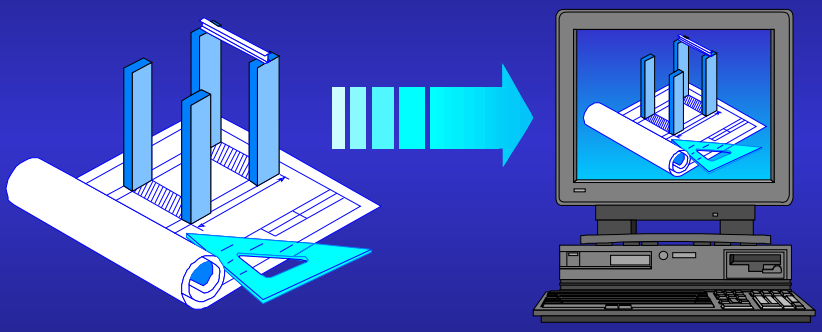
The Facade idea has also been applied more generally to whole packages and APIs. A simplified set of options and easier interface can have a simplifying effect on the use of whatever services are encapsulated, and may lead to greater uptake.

UML provides a slightly narrower definition of a *«facade»* package:

> A facade is a stereotyped package containing only references to model elements owned by another package. It is used to provide a 'public view' of some of the contents of a package. A facade does not contain any model elements of its own.

However, be aware that the contents of the Facade package should not be guessed at: they should be derived from understanding the common use cases associated with the full API. By way of balance, if the full API is never used, it suggests that its fuller feature set may have been a consequence of speculative generalisation.

Patterns for Object Management



Pattern-Based Software Development

Patterns for Object Management

- Objectives
  - Present the issues that motivate the use of and policies for object factories and managers
- Contents
  - Object factories
  - The Factory Method pattern
  - The Disposal Method pattern
  - The Manager pattern
  - The Leasing pattern
  - The Evictor pattern

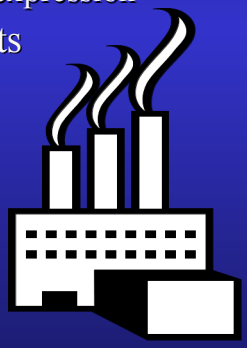
Consequences of Separating Classes

- Dependency management in a system often leads to introducing separations
  - Delegation often used to divide responsibility that would otherwise overburden individual classes and methods
- These separations need to be stable
  - Otherwise they introduce the "shotgun maintenance" problem: each change requires fixing multiple classes
- Separations introduced for decoupling also create a tension in a design that is based on encapsulation
  - How are objects created if their concrete type is hidden?

Dependency management is not for free: decoupling requires some investment of effort and clarity of thinking and purpose. The improvement in stability and testability of a system, however, is certainly worth the effort. However, a more decoupled approach to development tends to encourage a style that some developers find quite different: increased use of interfaces and helper objects, and decreased use of new expressions and concrete types in public method signatures.

Object Factories

- Knowledge for creation of an object can be delegated and encapsulated
  - Exact detail of creation is hidden from the caller, so the caller is not responsible for the *new* expression
- Object factories create other objects
  - Being an object factory may be the responsibility of a single operation, a whole object or a class

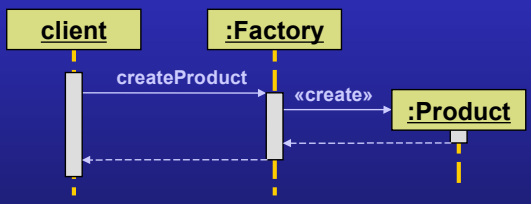


The separation of interface from implementation can lead to creational issues. If the client that uses an object through an interface is also aware of its concrete type and is the party responsible for creating it, there is no issue. However, if the client is only aware of the usage interface rather than the concrete creation type, but is also responsible for instigating the act of object creation, there are apparently opposing design forces that need to be resolved.

There are many patterns that address creation issues by encapsulating creation knowledge in some way. Objects that are responsible for creating other objects are known, for fairly obvious reasons, as factories.

The Factory Method Pattern

- Problem...
  - Working through an abstract class, how can you create a related object without knowing its concrete class?
- Solution...
  - Provide a method for creation at the interface level of an object that performs actual creation



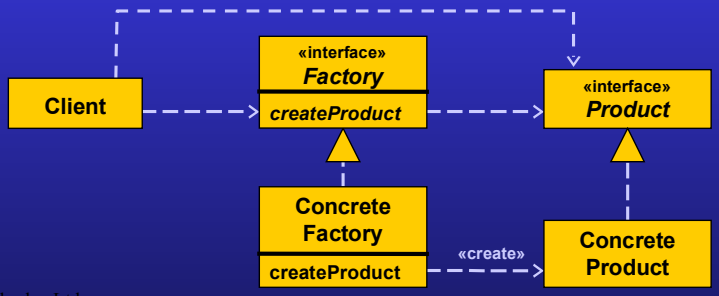
Parallel hierarchies are often created using the Factory Method pattern [Design Patterns]. The intent for this pattern is summarised as

> Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Thus, the idea of creation is contracted out at the interface level, but its realisation is deferred to one or more subclasses.

Typical Factory Configuration

- Mirroring the hierarchy of what is created...
  - Interfaces for creation is provided: a factory interface and a product interface
  - Actual creation from a concrete class is deferred to an implementing subclass of the factory interface



Generically we can see that in the two hierarchies, one is the product hierarchy:

A product interface defines how to use a created object. In the purest case this will be an interface, but it may alternatively be an abstract classes including some implementation.

A concrete product class is the type instantiated against the interface.

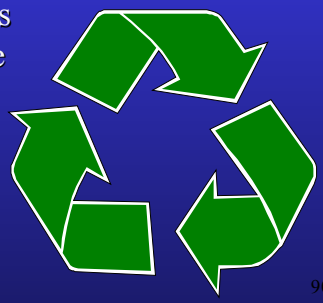
And the other is the creator or factory hierarchy:

The creator interface defines how to create instances that implement the product interface. As with the product, this may be either an interface or an abstract class.

The concrete creator class supports the creator interface and implement the creation of concrete product objects.

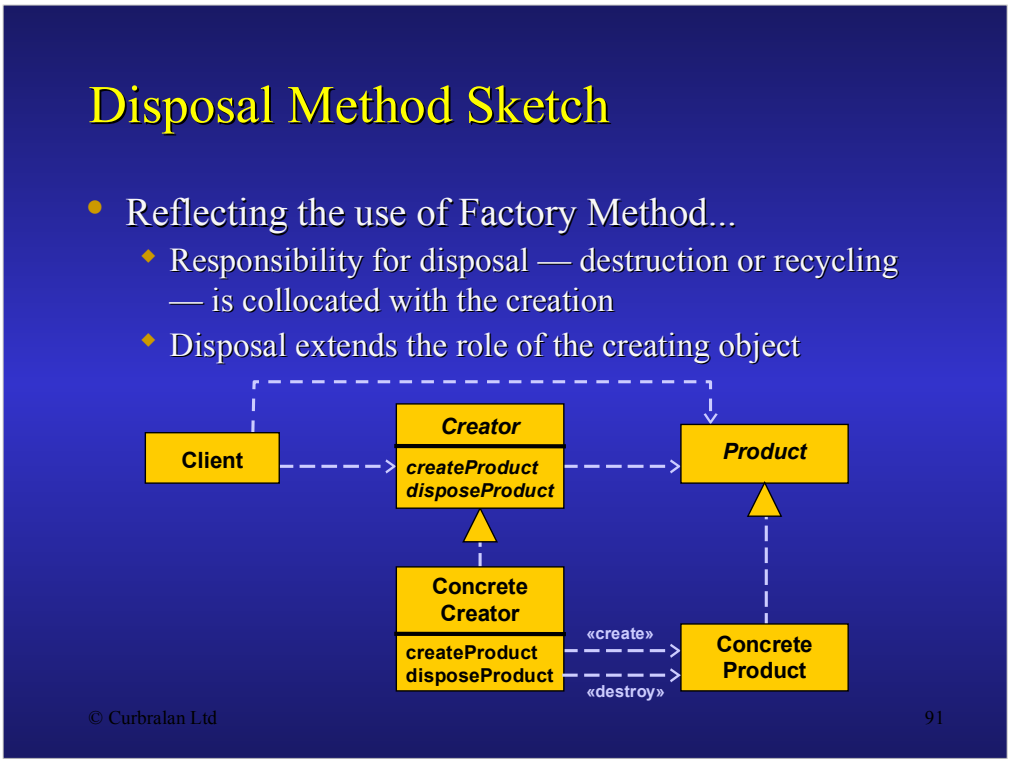
The Disposal Method Pattern

- Problems...
  - Who cleans up factory products after use?
  - How can you reduce dynamic memory usage?
- Solution...
  - Returned the created object back to its creator
  - The disposal is as encapsulated as the creation, so the object may be pooled for later reuse or discarded



Java and C# programmers leave objects lying around to be collected automatically by the Garbage Collector. It is, however, naïve to assume that a GC system solves all memory and resource management issues out of the box. Frequent creation of fine-grained objects, such as iterators, can potentially lead to inefficient use of resources. Inevitably performance will suffer.

Mirroring Factory Method, Disposal Method answers the question of who is responsible for the disposal of a factory created object by returning the object to the factory, thus localising all the knowledge of creation and disposal in a single place.



Disposal Method may be implemented either by giving the factory the object back directly:

```
public interface Creator
{
    Product create();
    void dispose(Product toDisposeOf);
}
```

Or by providing a method in the object that returns it to its creator:

```
public interface Product
{
    void dispose();
}
```

Importantly, because Disposal Method hides the policy of the factory, objects can be cached and recycled to save on the cost of heap management.

The Manager Pattern

- Problem...
  - How can a group of instances be managed so that management does not intrude on the client?
- Solution...
  - Introduce a manager object that handles the responsibility for managing instances, including creation, disposal and finding
  - The client accesses the instances via the manager object

The intent of the Manager pattern is given as

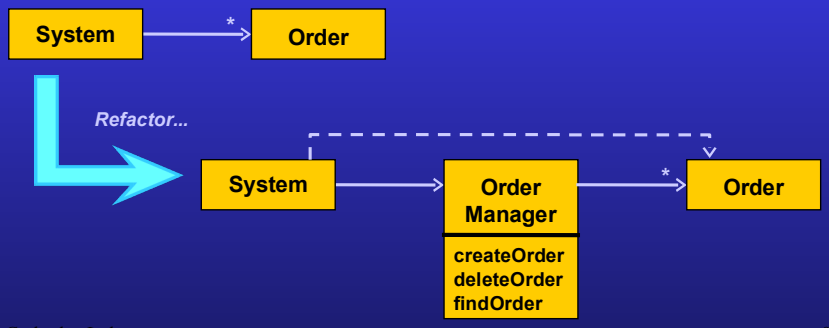
> The Manager design pattern encapsulates management of the instances of a class into a separate manager object. This allows for variation of management functionality independent of the class and for reuse of the manager for different object classes.

It can be where the management responsibility for instances should lie. Where objects are held in collections that are accessed by a client, it is often the client code that performs all the access management and manipulation details – creation and deletion, persistence management, searching, etc. This can lead to code duplication and client objects that are uncohesive – being seemingly involved in the details of other objects rather than in their principle purpose. It also represents poor encapsulation of responsibility, so that any change to the management details will affect multiple classes.

Introducing a Manager object allows the responsibility to be clearly located in one place. This can provide the principal point of contact – the main service abstraction – for using the underlying objects, in effect providing a more intelligent version of a collection.

Introducing a Manager

- Existing code can be refactored to introduce manager objects
  - Tends to simplify both client and target objects, decluttering objects higher up in the control hierarchy



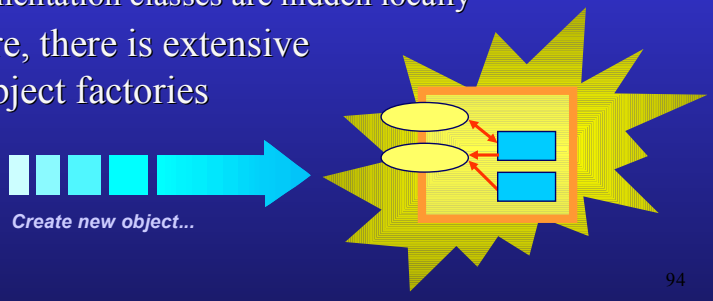
Manager objects typically manage collections of objects, being responsible for their creation and destruction, their lookup, and their general well being. Manager interfaces often sport a Factory Method, although not necessarily as a polymorphic operation, and also a Disposal Method.

Some Manager objects, as with some factories, are natural and do not have to be introduced consciously into a design — e.g. consider the relationship between a bank and its accounts — whereas others need to be introduced to take responsibility away from overly complex classes. There is often a close correspondence between Manager object classes and entity object classes, i.e. objects that are significant and persistent in a system. In a simple architecture there may be a direct correspondence between entity object — and therefore Manager — and relational table.

In the Enterprise JavaBeans architecture, the home interface plays the role of a Manager. It offers services for the creation and querying of objects, handling their persistence as necessary.

Distributed Object Lifetimes

- Distributed systems introduce a number of additional lifetime management issues
  - In heterogeneous systems object creation and destruction policies cannot be platform based
  - Interfaces are the currency of distributed systems, but implementation classes are hidden locally
- Therefore, there is extensive use of object factories



The simplicity of a new expression to create an object does not generally translate from local to remote usage. In a heterogeneous distributed system, the language types available on the target platform may not be available at the calling platform, and so an instantiation would not be able to name the concrete class. There is also the added complication of how to manage remote placement, which, given the aims of location transparency in a distributed system, are probably inappropriate.

The issues of remoteness and independence are addressed for distributed object creation by introducing object factories and, by extension, object managers. Only the "roots" of the system need to be made available via some other means, such as publicly named lookup service, e.g. the CORBA Naming Service.

Distributed systems emphasise interfaces as the means for establishing what objects are prepared to talk about. This is typically the case even in homogeneous systems, such as RMI. In other words, concrete classes do not exist at the level of distribution and are hidden locally. Again, they cannot be nominated as the template for instantiation.

In a heterogeneous system mechanisms, such as GC, cannot be easily assumed, as target platforms may have alternative arrangements for terminating object lifetime. Partial failure in a distributed system means that any mechanisms that requires explicit notification of use and non-use is likely not to end an object's life when it is no longer being used.

Different Lifetime Models

- *Client-centric*: Client dictates life of server object, but requires help to handle rogue clients, e.g. COM
  - For example, use of Disposal Method pattern, possibly combined with reference counting
- *Server-centric*: the "server knows best" means that the client has no implicit control, e.g. CORBA
  - For example, use of Evictor pattern
- *Shared responsibility*: client and server collaborate, e.g. RMI
  - For example, use of Leasing pattern

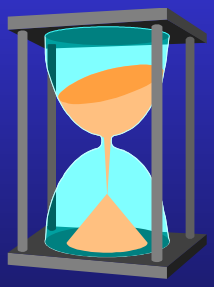
Different distributed system middleware architectures adopt different lifetime models. Lifetime models can require explicit support, requiring the involvement of the client and server programmers, or can be fully encapsulated by the middleware.

Client-centric systems assume that an object on the server exists only to serve the clients using that server. In principle, when there are no clients using that object, the object can be cleared automatically. If the client programmer must manage the interest registration and deregistration process, server objects are susceptible to rogue clients. Even when encapsulated, this approach suffers from the problem of reliability: a crashed client will not have unregistered its interest. Such an architecture is fundamentally flawed for distribution purposes, and requires additional infrastructure support. COM had to introduce a ping message to check whether clients were still alive or not. This is one of many hacks that had to be added to take a non-remote architecture and make it remotable.

A server-centric architecture, such as CORBA, does not care what the client thinks, and it is entirely up to the server as to what it does with its objects. If an object is no longer available, the client will be notified in no uncertain terms using an exception. This may seem quite a hard-line, which is why CORBA designers often expose lifetime management requests in their IDL. The Portable Object Adapter (POA) model for CORBA also distinguishes between transient and persistent objects. A transient object lives for a server session (at most) whereas a persistent one is effectively eternal.

The Leasing Pattern

- A client leases an object for a period of time
  - If the lease expires, the server drops the object
  - If the lease is renewed, the server retains the object until at least the next expiry
- Consequences of living on borrowed time...
  - Addresses problem of rogue clients
  - Emphasis on client-side framework

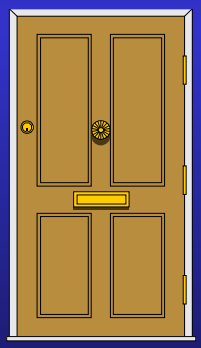


It is possible for clients and servers to share the responsibility for lifetime management. Distributed garbage collection presents a number of design challenges, but represents an example of whole-system co-operation. RMI is based on this. There are a number of issues with distributed GC, including the forgetfulness and vagaries of GC, but again the spectre of partial failure raises its head to steal the show. How can a globally broadcast "are you alive" request be avoided?

The Leasing pattern, as used in RMI and Jini, presents a solution to the problem, based on the idea of the dead man's switch. The interest in the server object is only kept alive whilst the client explicitly maintains it. If the client dies, then so does the interest. To be precise, the mechanism used is that object interest is subject to lease, which is expires. If the client does not renew the lease before expiry, the interest is dropped. When no client has interest, the object is dropped. It is possible that a renewal may be missed if the client is heavily loaded, in which case a non-existent object will announce its absence through via exception.

The Evictor Pattern

- The server holds a fixed-size pool of active objects
  - Each request is directed to an object in the pool or...
  - An object is dropped and a new one paged in to satisfy the request
- Usage is logged for each object in LRU or LFU strategies

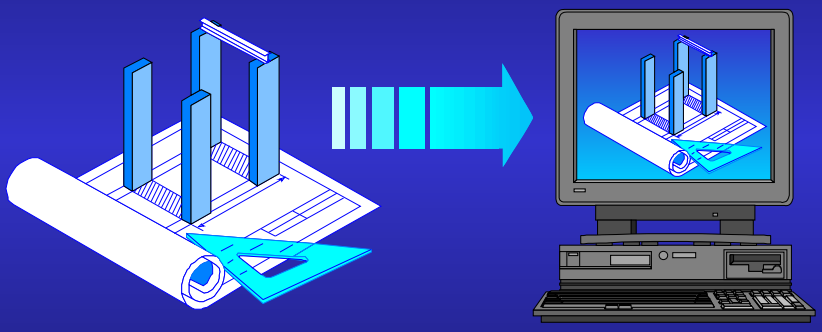


CORBA persistent objects are, in principle, immortal. A CORBA object is a virtual object: it does not actually exist except conceptually. Requests on CORBA objects are handled by servants. Often there is a simple one to one relationship between servant and CORBA object, and in these cases we often speak casually and freely of the servant being the CORBA object. However, thanks to the POA architecture, we can also handle each operation request individually or reuse servants for different CORBA objects, either over time or at the same time, e.g. if they are stateless service objects.

The adaptable features of the POA become more significant when we consider the need to control memory usage. In a server representing many CORBA objects, a one-to-one mapping may be a resource hog. The Evictor pattern [Henning+1999] provides a server-side framework that aims to conserve resources according to server policy.

Active objects have their servants held in an Object Pool. The pool dispatches requests to these objects. If there is a request for an object not currently served by the pool, and the pool is deemed to be full, an existing servant is either dropped or repurposed to handle the request. The eviction criterion could be least recently used (LRU), least frequently used (LFU), memory usage or some other priority assigned to the objects.

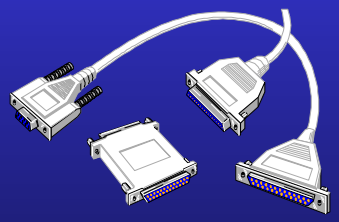
Patterns for Pluggability



Pattern-Based Software Development

Patterns for Pluggability

- Objectives
  - Present patterns that support plug-in style flexibility
- Contents
  - The Strategy pattern
  - The Interceptor pattern
  - The Null Object pattern
  - The Context Object pattern
  - The Mock Object pattern
  - The Command pattern
  - The Command Processor pattern
  - The Block pattern



The Strategy Pattern

- How can different algorithms for the same task be supported with resorting to *switch* code?
- Introduce an interface that captures the algorithm usage, and implement accordingly
  - For example, a formatting interface can be separated from the possible implementations
- Offers a delegation-based alternative to subclassing the class defining the algorithm
  - Often too narrowly described in terms of expressing just a family of algorithms through an interface: related parts of an algorithm may also be expressed

How can different algorithms for the same task be supported with resorting to case code? Flags offer a clumsy form of polymorphism. Therefore, introduce an interface that captures the algorithm usage, and implement accordingly.

Rather than seeing them as distinct, many patterns can adopt roles in common with Strategy. For example, the use of a Null Object to express a Strategy is a common application.

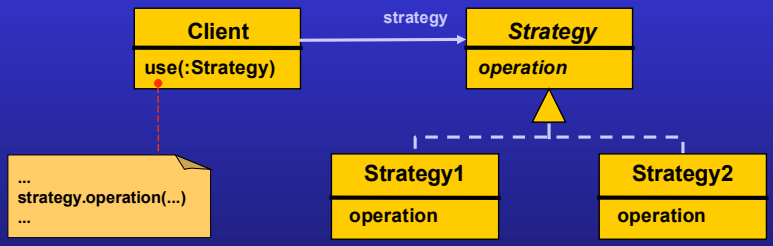
The intent of the Strategy pattern [Design Patterns] is given as

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

In many cases Strategy replaces the use of explicit switch code. Its use can simplify testing, comprehension and extensibility.

Strategy Configuration Sketch

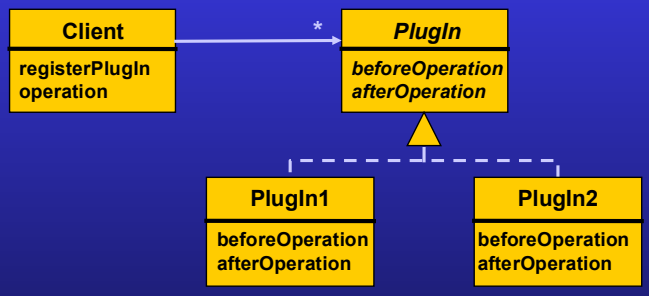
- The canonical form of Strategy is expressed as a class hierarchy with runtime polymorphism
  - However, can also be expressed through more ad hoc polymorphic mechanisms, e.g. reflection or templates



Although Strategy is most often considered in the context of runtime polymorphism, it plays a dominant role in generic and generative programming in C++ expressed as compile-time polymorphism.

The Interceptor Pattern

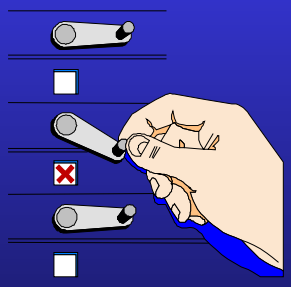
- An object's basic behaviour can be extended by adding plug-ins that are called on certain actions
  - Often used for adding extra-functional behaviour, such as instrumentation and logging



The Interceptor pattern supports non-intrusive extension of an object's capabilities by calling out to plug-ins. For many cases, Interceptor is less intrusive and more orthogonal than Decorator and Template Method, which tend to introduce coupling or object management problems. Interceptor can be considered to be based on Strategy and Observer, and is a common feature of component architectures and an increasingly common feature of frameworks. It can be said to support an aspect-like approach to extensibility.

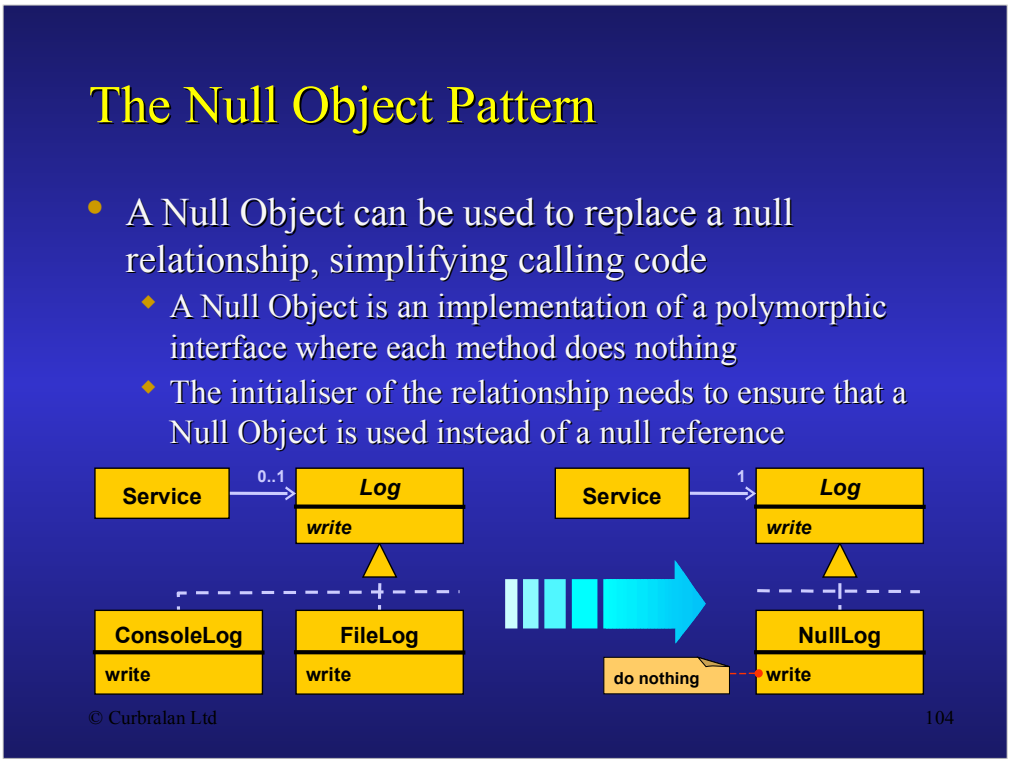
Optional Object Relationships

- What is the best approach to implementing optional object relationships?
  - Optional relationships have multiplicity *0..1*
- The commonest solution is to use a null reference for the zero case
  - But programmers always need to check explicitly for null before use
- Polymorphism is an alternative mechanism for selection that is sometimes more transparent



An optional object relationship has multiplicity 0..1 meaning that a link can be present or not. In terms of programming this commonly means that a pointer or reference in a language will be used to represent the link, and that when it is null it represents the 0 case. For storage in a database, a null foreign key indicates this case.

Before following a link it must therefore be checked. In some cases the use of a null is notionally transparent, i.e. something is ignored if it is not there. In others it implies a very different state of an object and leads to fundamentally different behaviour.



There is a great deal of procedural clunkiness in code such as

```
if object reference is not null
    call method on object reference
```

This can be repetitive, as well as error prone; it is possible to forget to write such a null guard. The Null Object pattern ensures that the reference need never be null, and hence there is no need to check for nullness:

```
call method on object reference
```

This radically simplifies the code. Substituting a stateless object that does nothing or takes the appropriate default action offers a powerful solution, normally based on inheritance. The Null Object pattern is more common than many people realise: consider the role of the null file device on many operating systems (/dev/null on Unix and NUL on Microsoft systems), or the no-op machine instruction.

In the example above, logging is optional. Allowing nulls leads to usage in long methods such as

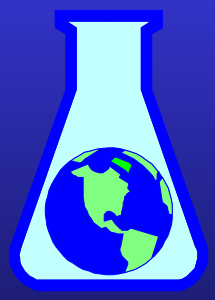
```
if log reference is not null
    write initial message to log
carry out service
if log reference is not null
    write next message to log
clean up service
```

The Null Object version is simpler and more direct:

```
write initial message to log
carry out service
write next message to log
clean up service
```

The Context Object Pattern

- How can objects in different parts of a system gain access to common facilities?
  - Keeping in mind the goal of loose coupling, which supports extensibility, comprehensibility, testability, etc.
- Pass execution context for a component — whether a layer or a lone object — as an object
  - Avoids tedium and instability of long argument lists
  - Avoids explicit or implicit global services, e.g. Singletons or other uses of *static*



Whether in the client or the server, application code using a middleware framework will need to gain access to the execution context of that server. There is a strong temptation to make such context global in some way. This temptation should be resisted. It is easier to keep control of dependencies and to manage isolation of services and features by capturing the context in an object and passing it explicitly to where it is needed.

The Mock Object Pattern

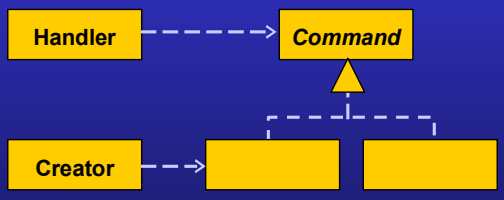
- Provide decoupled interfaces for outgoing dependencies
  - Call-out interfaces that encapsulate dependencies such as configuration, factories and database connections
- Can use an instance of the production class or substitute a mock object for testing
  - Allows I/O to be tested without actually performing external I/O and supports isolation testing and replacement

The principle here is that a system's configuration should be parameterized from above rather than from below: factories, configuration info, logging objects, etc. should be passed in from the root of the program downwards, and should not be picked up as facilities hardwired into the lower layers of a system, e.g. using globals, public statics or Singleton objects. This means that, in general, interfaces should be used and globals and their kin avoided.

Consider, how would you test that code using a factory interface responded to failed creation correctly? What you want to do is inject faults into the system, so that the factory fails to create an object. In C, C++ or C# you might consider introducing a pre-processor hack: conditionally compile a failing test version and a normal release version. The test version of the factory would throw exceptions instead of creating an object. Java does not have a pre-processor, so you are forced to create some kind of parameterisation for the factory to behave one way or another, either external configuration picked up at runtime or a runtime flag that can be set, or to perform clever tricks with dynamic class loading. All of these options are particularly unattractive. The solution is to pass in a Mock Object that is a failing factory. The mock factory injects faults for testing, taking advantage of this interface-given flexibility.

The Command Pattern

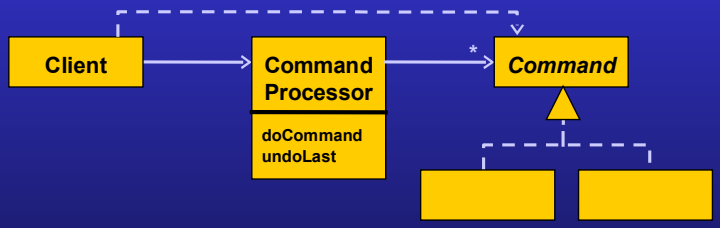
- Abstracts a function call and calling context into an object that can be passed around
  - Decoupling the decision of what to execute from the decision of when to execute, inverting the control flow
- Concrete classes define the context needed for executing the requested command



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? Treat the request for functionality as an object in its own right. Concrete Command classes define the exact behaviour, whereas the handler sees only the Command interface.

The Command Processor Pattern

- Command Processor complements Command by handling execution policy
  - An additional and complementary micro-architecture for Command
  - Additional management responsibilities should clutter neither the Command hierarchy nor the client code



The most cohesive solution, rather than polluting client code with Command management details or cluttering Command code with inter-Command handling, is to introduce an additional manager object, a Command Processor.

The use of a Command Processor qualifies as a separate design decision. It is not always the case that a Command would require a Command Processor, but when it does there are clear trade-offs and a separate design role is introduced.

A Command Processor

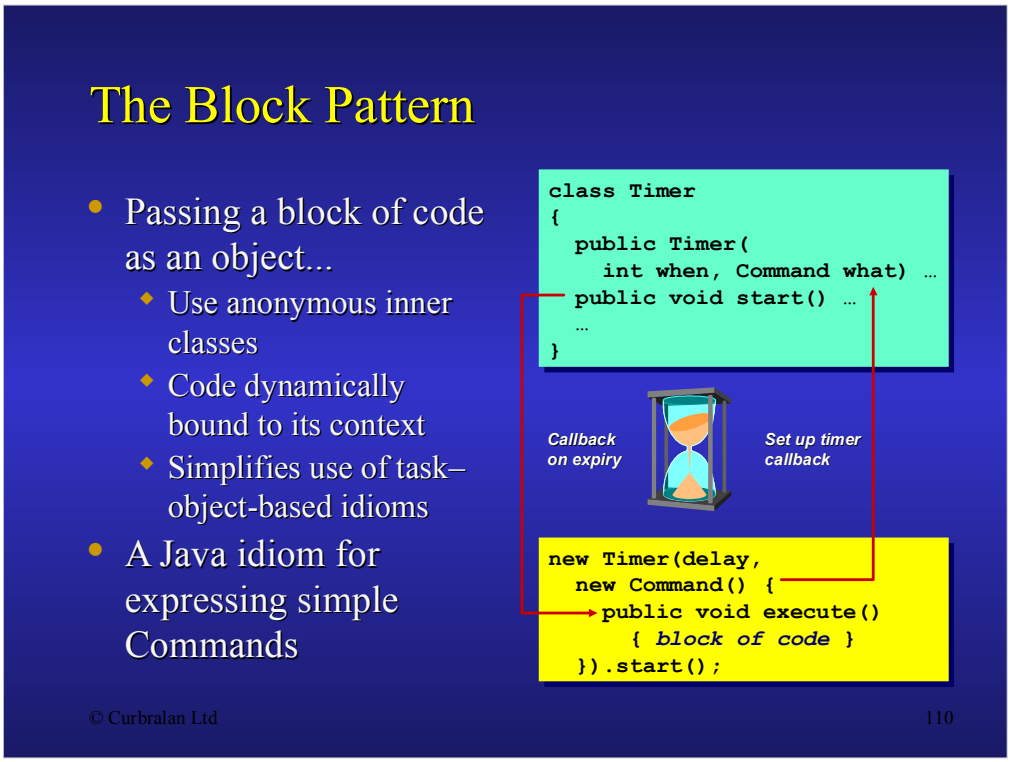
- A separate command processor can take responsibility for command objects
  - Execution and management are encapsulated

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

```
public class CommandProcessor
{
    public void doCommand(Command command) ...
    public void undoLastCommand() ...
    ...
    private Stack<Command> history;
}
```


Building on a single Command undo, last-in/first-out ordering is needed to undo actions in the proper sequence:

```
public interface RecoverableCommand extends Command
{
    void undo();
}
public class CommandProcessor
{
    public void doCommand(Command command)
    {
        command.execute();
        if(command instanceof RecoverableCommand)
            history.push(command);
    }
    public void undoLastCommand()
    {
        history.pop().undo();
    }
    ...
    private Stack<Command> history = new Stack<Command>();
}
```



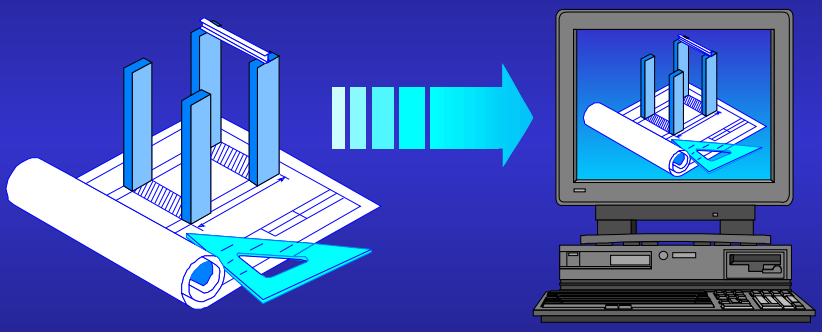
One of the key differences to the Java language between version 1.1 and 1.0 of the JDK was the addition of inner classes. Superficially inner classes look like the nested classes of C++. However, the resemblance ends there. Objects of inner classes can bind to the variables within the scope of their definition. For local inner classes, i.e. those defined in block scope rather than within a class, these variables must be declared `final`.

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

In the example above an object of a local anonymous inner class derived from `Command` is passed to a simple implementation of a timer object for execution on expiry. A similar example would be the use of the Block idiom with the thread `Thread` class as executor and `Runnable` as the base interface for the block.

Inner class go further than simply single method classes, and are the basis for easy use of the current library's event-handling model.

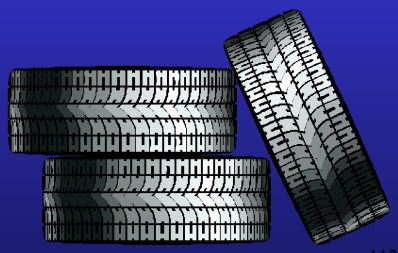
Patterns for Iteration



Pattern-Based Software Development

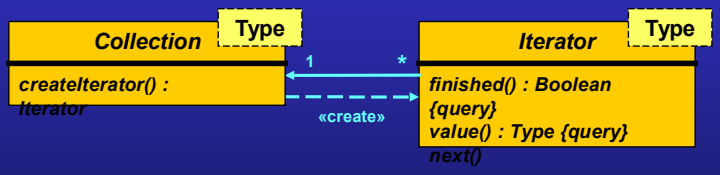
Patterns for Iteration

- Objectives
  - Present patterns for encapsulating iteration across aggregate objects
- Contents
  - The Iterator pattern
  - The Enumeration Method pattern
  - The Batch Method pattern
  - The Batch Iterator pattern



The Iterator Pattern

- Classically defined in terms of separating the responsibility for iteration from its target
  - The knowledge for iteration is encapsulated in a separate object from the target
  - The target object is normally a collection, but this is not necessarily the case



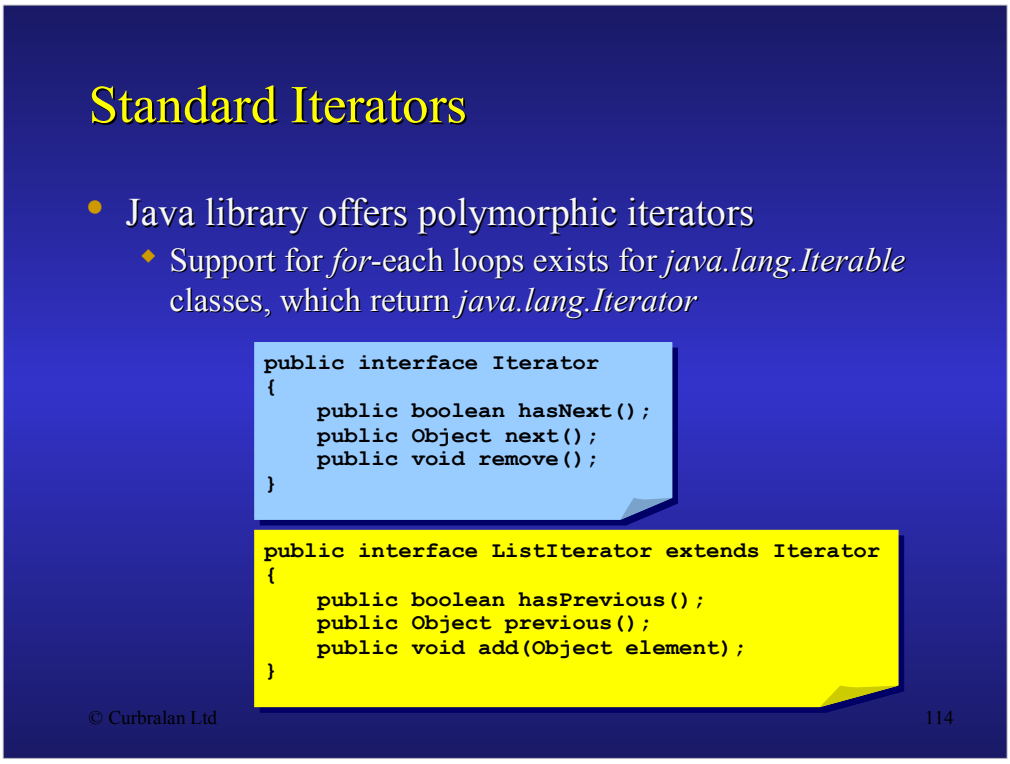
Given that a collection object is encapsulated, how can one conveniently access all of its elements in turn without knowledge of its internal structure? An unsatisfactory approach is to embed the concept of a position cursor within the collection. The result is a design with poor cohesion (a collection now supports peripheral features not related to collecting) and can be awkward to use, e.g. multiple traversal is not supported.

One solution is the Iterator pattern illustrates a clear separation of responsibilities and concerns with respect to objects: a collection object collects; an iterator object iterates. As a solution, iterators have been around for some time.

There are four basic properties plus one optional one that any iterator collaboration must support:

- Initialisation of the iteration (essential).
- Access to the current iterated value (essential).
- Advancing the iteration to the next value (essential).
- The ability to determine if the iteration is complete (essential).
- Finalisation of the iteration (optional).

These individual requirements may be mapped one for one with operations, or they may be folded into fewer operations that involve side effects. In a direct mapping it is worth noting that these functional requirements correspond one for one with the structural elements of C, C++, C# and Java's `for` loop, with the possible exception of the final clean up, which is often either a no-op or invisible.



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

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

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

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

The Enumeration Method Pattern

- The collection receives a Command, which it then applies to its elements
  - An inversion of the basic Iterator design
- Iteration is encapsulated within the collection, so the client is not responsible for loop housekeeping
  - Additional iteration-related policies, such as synchronisation or rollback, can be encapsulated
  - Often used in conjunction with Visitor

Enumeration Method is a fundamental iteration pattern that encapsulates the actual control flow of the traversal loop. It is essentially an inversion of Iterator. Rather than execute the action for each element in a collection outside of the collection, a method can be provided that applies an action to each element in turn.

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

In some senses, Enumeration Method is higher level: it captures and names common traversals for the programmer to use directly. It may also be combined transparently with acquisition and release patterns to ensure transactional and thread-safe behaviour, i.e. atomic, exception safe and consistent.

There are strong similarities between Enumeration Method and the Template Method pattern that encapsulates an algorithm at the base of an inheritance hierarchy, as well as the Visitor and Execute-Around Method patterns.

Enumeration Method in Java

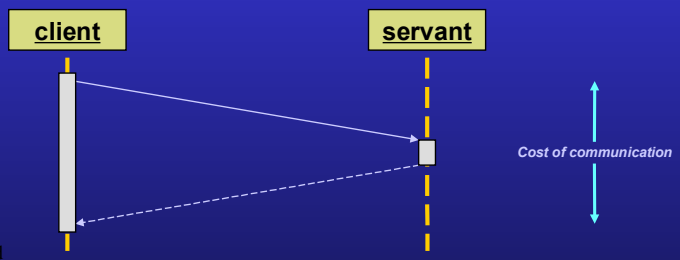
- Block is often applied in conjunction with Enumeration Method
  - Efficiency on a par with Iterator

```
public class Example
{
    public void example(Collection collection)
    {
        collection.forEachDo(
            new Collection.Operation() {
                public void execute(Object element)
                    { System.out.println(element); }
            });
    }
    ...
}
```


In Java, barring some of the syntactic overhead, using a Block can be an effective way to pass Command context into an Enumeration Method.

Remote Calls

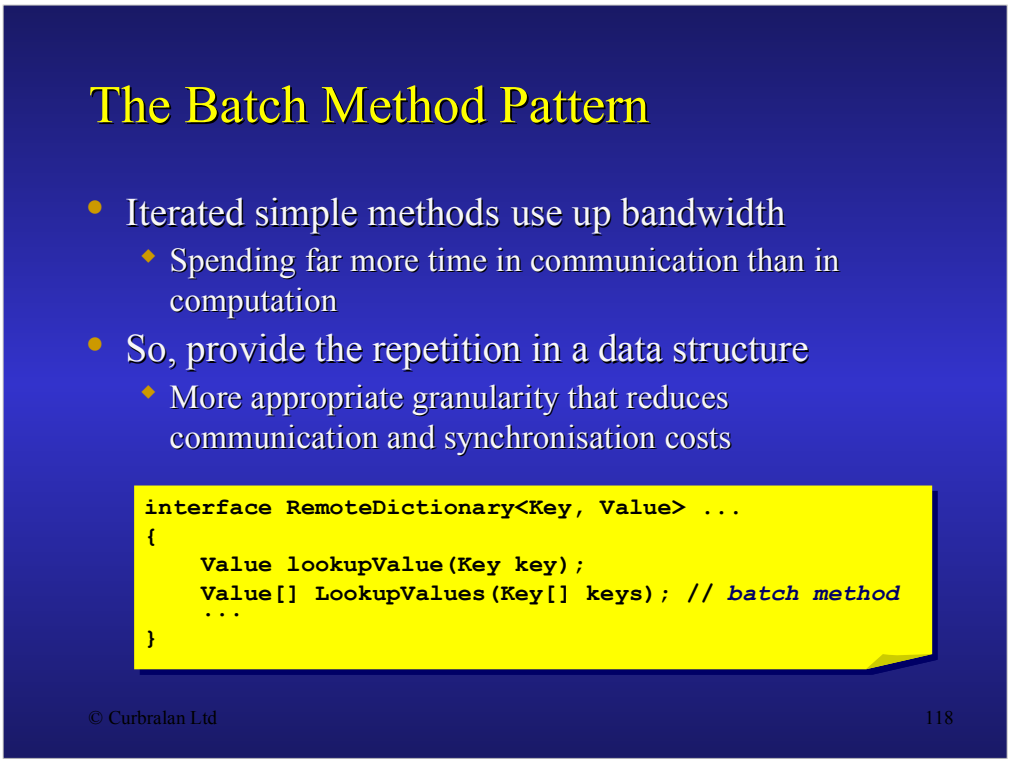
- In distributed systems, method calls are no longer trivial and concurrency is implicit
  - Communication can dominate computation
  - Partial failure is almost inevitable
  - Assumptions about method design used for in-process calls no longer hold



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 can dominate the behaviour of the system.

In addition to all of the issues raised by concurrency, there is additional cost involved in remote operation calls. When sketched out on a sequence diagram, sequences of property gets and sets suffer the 'sawtooth' effect, spending more time in communication than they do in performing useful work. It is the ratio of communication to computation that is an important consideration in distributed computing.

The scope for failure is even greater in a distributed system than in a local threaded 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.



Iterator allows traversal of a sequence in an abstract and controlled fashion. Defining a Combined Iterator — i.e. an Iterator whose method interface is in terms of a Combined Method — addresses the basic concurrency issues. However, performance issues arise in a distributed environment for repeated short operations, such as a loop repeating on a relatively simple operation. Again, the sawtooth effect occurs, incurring a great deal of communication overhead for relatively little computational benefit.

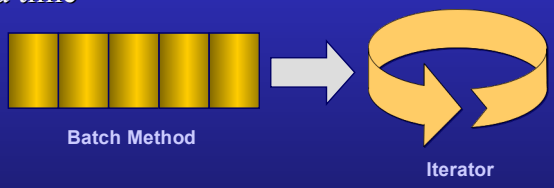
The solution is to 'slurp' as much across as possible in a sequence, which is the essence of the Batch Method pattern. This is in many respects another inversion of Iterator, and it is also related by both similarity and inversion to Enumeration Method. The basic structure has placed the repetition in the data structure.

The Batch Method — applied in `lookupValues` as opposed to the sequentially minded `lookupValue` — deals with a sequence of results that corresponds to a sequence of inputs: rather than declare a single parameter ref as a sequence of input-result pairs, the inputs and results should be striped into separate sequences, one in and one out. This avoids the cost of sending unset or redundant data to and fro.

It is possible that the lookup shown may not succeed and so either an exception handling policy must be adopted if inability to find is considered a failure, or some kind of out-of-band value must be used.

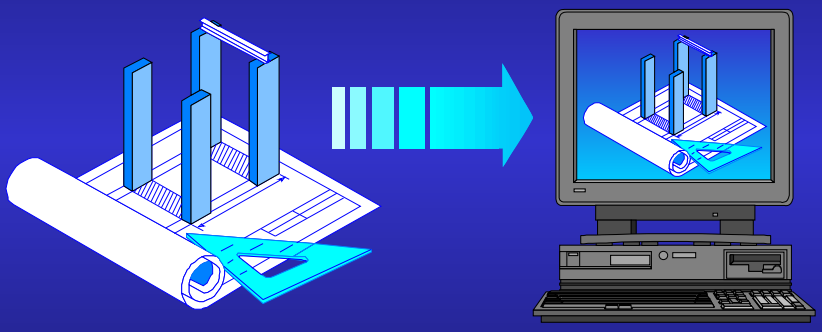
The Batch Iterator Pattern

- Iterating over individual values in a server wastes both time and bandwidth
  - But Batch Method can cause client to block for too long
- Batch Iterator is a compound pattern resulting from combining both Iterator and Batch Method
  - An Iterator uses a Batch Method to pull multiple values at a time



Batch Method offers a compromise in granularity and control, allowing a caller to step through a collection in strides greater than one step but less than the whole. It allows a client to control flow, chunking and caching of data, which may be more suitable for displaying lists of values on screen when only a subset will be viewable at any one time.

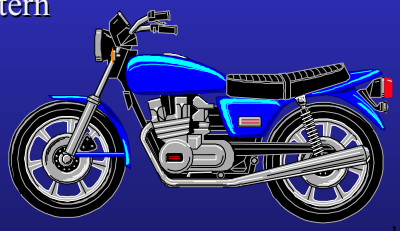
Patterns for Object Lifecycles



Pattern-Based Software Development

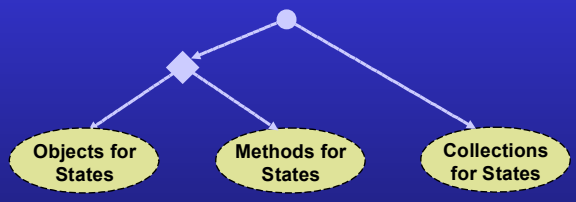
Patterns for Object Lifecycles

- Objectives
  - Present patterns for object modal behaviour
- Contents
  - Modal behaviour
  - The Objects for States pattern
  - Stateful and stateless mode objects
  - The Methods for States pattern
  - The Collections for States pattern



Modal Behaviour

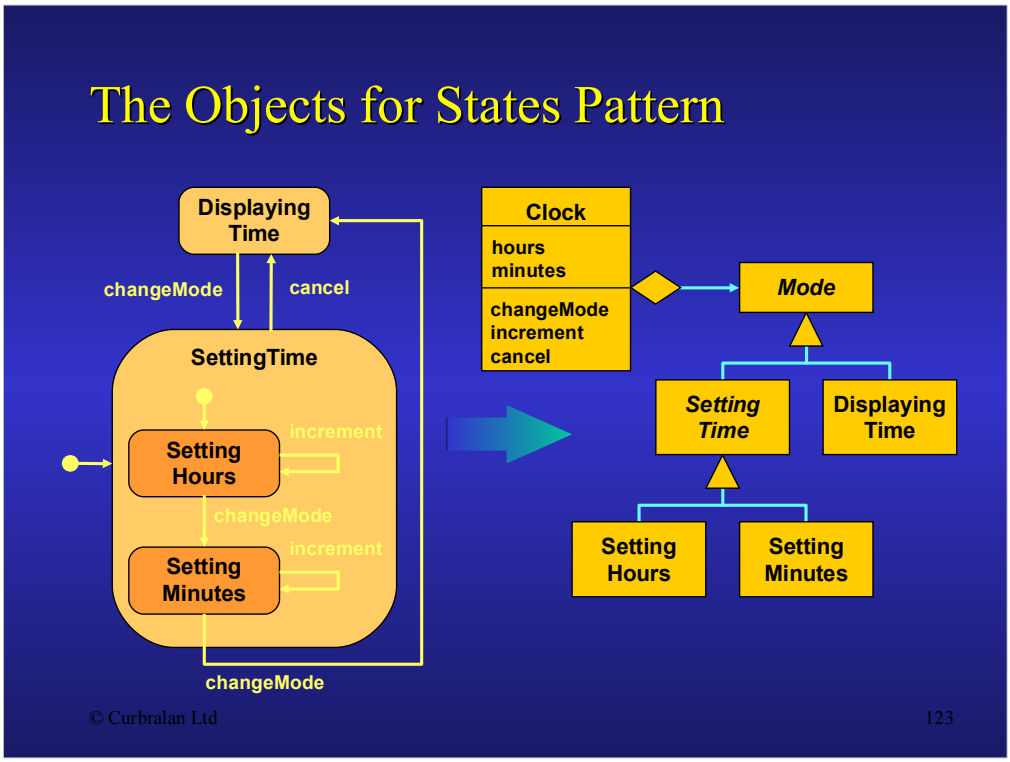
- There are a number of successful recurring solutions for expressing modal behaviour
  - These patterns focus primarily on how modal state is represented, and less on the mechanics of transition between modes



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. 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.

However, in spite of its common name, *the* State pattern — documented by the Gang of Four — is not the only game in town. A common misconception is that any problem associated with a state model should necessarily be realised using the State pattern. The name *State* suggests the problem domain rather the solution, and as such appears to address all variations. As with variables, naming is important: should be memorable, usable and distinct; should be based on something that characterises the proposed solution rather than the problem. The Gang-of-Four naming style is based on choosing the name of a class role in the solution. The name State is taken from the root of the (often stateless!) class hierarchy that defines modal behaviour. The synonym listed in *Design Patterns*, Objects for States, should be preferred as it is less ambiguous and more precise.



The Objects for States 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. Method calls on the context are forwarded to the state object; responsibility for implementing behaviour in a particular state is therefore delegated. The transition to a different state means replacement of the behaviour object by another of a different class. Polymorphism replaces the explicit conditional lookup.

Transitions between states can be managed either by the behavioural state objects themselves, or by a centralised mechanism, such as a table lookup.

Although a relatively simple example, the clock lifecycle shown above illustrates a number of statechart features, including the use of superstates to generalise description of behaviour. The specialised mode correspond to specialised classes in the Objects for States configuration.

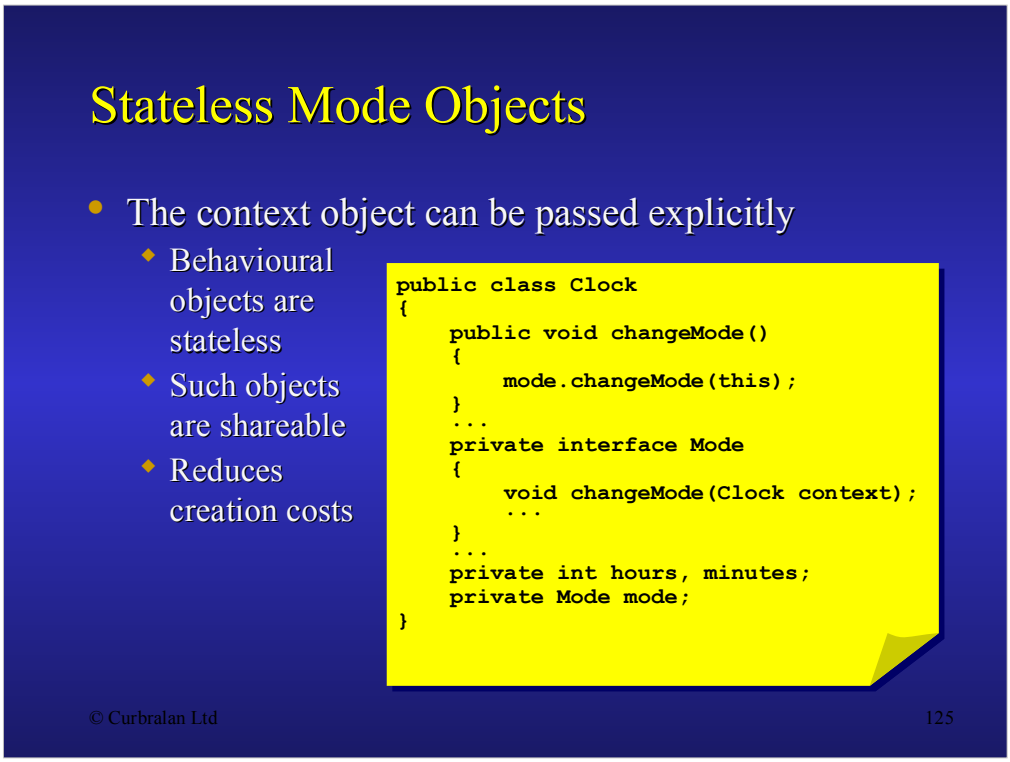
Stateful Mode Objects

- In Java, an object's fields are visible to instances of its inner classes
  - Simplifies access to context state

```
public class Clock
{
    public void changeMode()
    {
        mode.changeMode();
    }
    ...
    private interface Mode
    {
        void changeMode();
        ...
    }
    ...
    private int hours, minutes;
    private Mode mode;
}
```


The simplest and most direct approach is to use inner classes so that the context object's state is implicitly visible to the behaviour classes:

```
public class Clock
{
    public void changeMode() { mode.changeMode(); }
    public void increment()  { mode.increment();  }
    public void cancel()     { mode.cancel();     }
    private interface Mode
    {
        void changeMode();
        void increment();
        void cancel();
    }
    private class DisplayingTime implements Mode ...
    private abstract class SettingTime implements Mode
    {
        public void cancel() { mode = new DisplayingTime(); }
    }
    private class SettingHours extends SettingTime ...
    private class SettingMinutes extends SettingTime
    {
        public void changeMode() { mode = new DisplayingTime(); }
        public void increment()  { ++minutes; }
    }
    private int hours, minutes;
    private Mode mode = new SettingHours();
}
```



An alternative and more efficient approach is to use stateless `static` behaviour classes, passing the context in. Stateless objects can be shared:

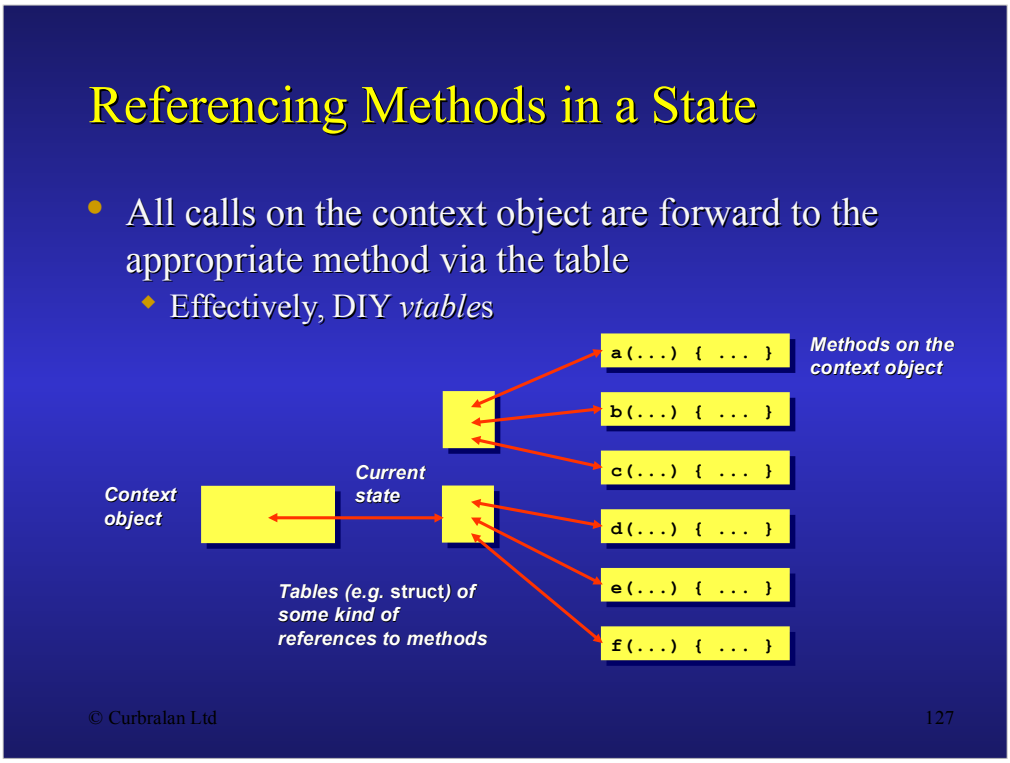
```
public class Clock
{
    public void changeMode() { mode.changeMode(this); }
    public void increment()  { mode.increment(this);  }
    public void cancel()     { mode.cancel(this);     }
    private interface Mode
    {
        void changeMode(Clock context);
        void increment(Clock context);
        void cancel(Clock context);
    }
    private static class DisplayingTime implements Mode ...
    private static abstract class SettingTime
        implements Mode ...
    private static class SettingHours extends SettingTime ...
    private static class SettingMinutes extends SettingTime
    {
        public void changeMode(Clock context)
            { context.mode = displayingTime; }
        public void increment(Clock context) { ++context.minutes; }
    }
    private int hours, minutes;
    private Mode mode = settingHours;
    private static final Mode
        displayingTime = new DisplayingTime(),
        settingHours   = new SettingHours(),
        settingMinutes = new SettingMinutes();
}
```

The Methods for States Pattern

- Objects for States can lead to a proliferation of classes and a complex design
  - Not inappropriate in all cases, but in many the design can be overly complex
- Methods for States represents each state as a table of method objects
  - In Java, this requires the use either of fine-grained Command objects or use of reflection, which make this pattern harder and less appropriate to apply than in other languages

The intent of the Methods for States pattern is to encapsulate modal behaviour of an object within a single class. In stateful objects with strongly modal lifecycles, the behaviour of a given method can be history sensitive, differing significantly according to the current state of the object. Simple lifecycle models can be implemented in terms of flags and conditional statements in each method; an approach whose code comprehensibility scales poorly. More sophisticated modal behaviour can be realized through object delegation, drawing on a community of dependent classes to express the behaviour. However, the larger the community, the more pronounced coupling and comprehensibility problems become.

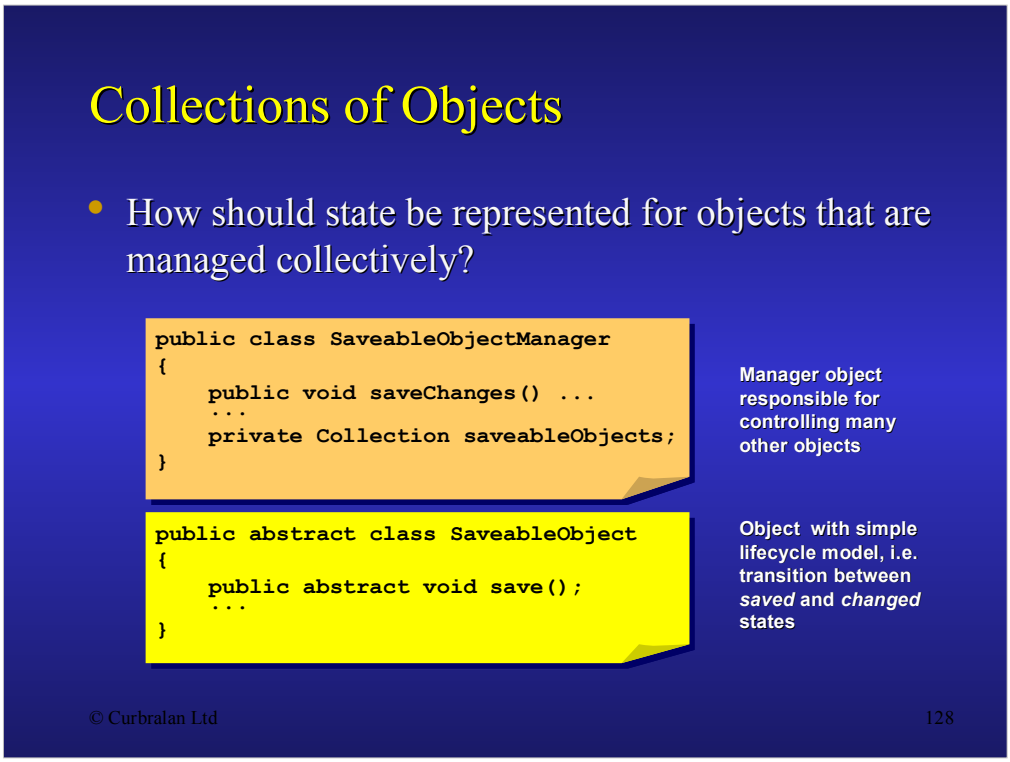
Using Methods for States a class is able to express all of the different behaviours as ordinary methods. It can do so without either the control coupling and reduced readability of large conditional statements or a large supporting cast of ancillary classes. Indirection, based on referring to methods as objects, is used to both represent the state and dispatch from a public method request to the correct underlying method for the state.



Methods for States represents an object's mode as a simple data structure containing references to methods. Each history-sensitive public method of the object forwards a call, along with any arguments received, to a corresponding entry in the data structure. Each different behaviour for the object is implemented as its own private method. Each mode is associated with its own data structure instance, which holds references to the relevant private methods.

The method references may be true direct method references, such as member function pointers in C++, or they may take the form of the symbolic method names that are resolved using reflection. In languages such as Ruby and C# there are different approaches that yield the same result. However, in Java there is no convenient correspondent (reflection is one possibility, Command objects another), and so the pattern is not as easily applicable, and Objects for States proves easier in almost all cases.

The data structure holding the method references can be a record-like data structure with named fields, such as a C# `struct`. Alternatively, a dictionary object can be used to look up the private method reference corresponding to each history-sensitive public method. In effect, this configuration emulates the normal method lookup table (*vtable*) mechanism, with a little added customization, evolution, and intelligence. Where only a single public method is history sensitive, no intermediate data structure is needed to represent the mode: a single method reference will suffice. Global, module, or class-wide variables can be used to hold the single instance of the data structure (or method reference) required for each mode.



Consider the case when you are managing a collection of similar objects that have lifecycles with only a few distinct states. These objects are often operated on collectively with respect to the state they are in.

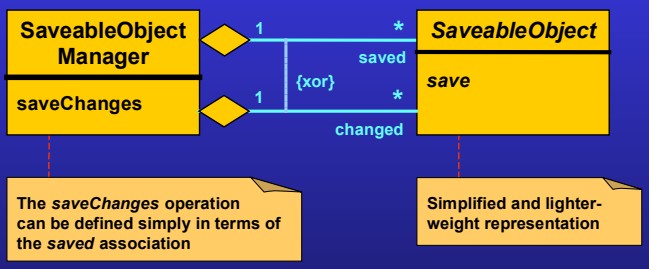
The objects, whose lifecycle and state is principally of interest to the object that manages them, may be modelled as individual state machines. However, in many ways the objects are independent of this state model, which is a view their manager has of them. What is a suitable model for the collected objects and their managing objects that emphasises and supports this independence? How is the responsibility for state representation and management divided between the collected objects and their managing objects?

Consider an application that holds a number of graphical objects that may be manipulated by a user, such as a CAD tool or graphical editor. The objects manipulated by the user are the workpieces, and these may be saved to some kind of persistent storage, e.g. some kind of database. These objects all share a simple lifecycle model, in addition to their type specific state.

For such a simple state model we can assume that the use of the Objects for States pattern is inappropriate. Instead we could use a simpler flag-based approach, representing state as an attribute. However, this means that to perform an operation, such as `saveChanges`, all objects must be iterated over and individually queried. This is inefficient in terms of both expression and performance.

The Collections for States Pattern

- Partition objects into separate collections with respect to the state they are
  - State classification is extrinsic, in terms of collection membership, rather than intrinsic



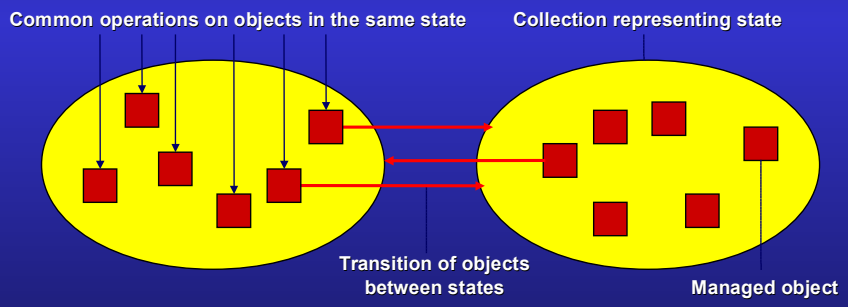
The Collections for States pattern expresses the state model outside the collected objects. Each state of interest is represented by a separate collection that refers to all objects in that state. The manager object holds these state collections and is responsible for managing the lifecycle of the objects: when an object changes state, the manager ensures that it is moved from the collection representing the source state to the one representing the target state.

Traversal and operation on all objects in a given state is greatly simplified as well as optimised: for operations on objects in a given state, only the collection representing that state is traversed:

```
public class SaveableObjectManager
{
    public void saveChanges()
    {
        for(SaveableObject workpiece : changed)
            workpiece.save();
        saved.addAll(changed);
        changed.clear();
    }
    ...
    private Collection<SaveableObject> saved, changed;
}
```

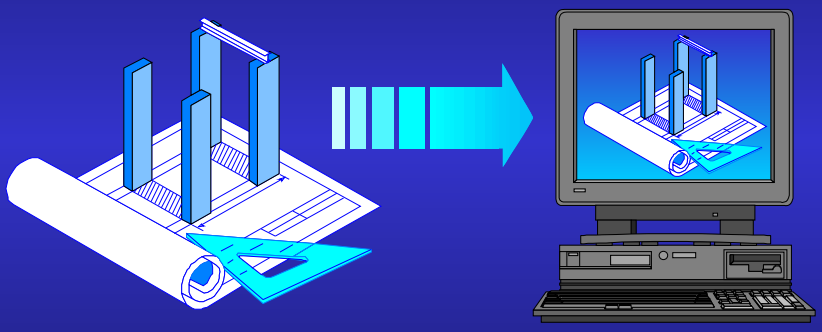
A Sketch of the Design Concept

- Schematically the concept of collection as enclosure or classifier can be made clear
  - In this case by using a non-UML notation



Note that Collections for States is not a replacement for Objects for States or Methods for States: they represent design alternatives based on different context and problem detail. Collections for States can be used to complement either Objects for States or Methods for States, which are contradictory, for optimisation.

Patterns for Notification



Pattern-Based Software Development

Patterns for Notification

- Objectives
  - Describe the common roles in a notification relationship along with various patterns that support notification
- Contents
  - Event flow
  - The Observer pattern
  - The Model–View–Controller pattern
  - Re-entrancy, concurrency and distribution
  - The Event Channel pattern
  - The Pipes and Filters pattern

Event Flow

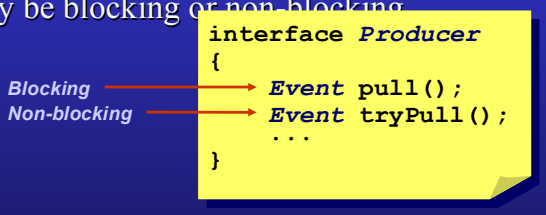
- Events associate data and control
  - Notification is based on information about change
  - Event propagating architectures have a great deal in common with message-based architectures
- Two roles in any notification relationship...
  - The *producer* is the source of the events
  - The *consumer* of the sink for the events

Events indicate an external stimulus or a change of state in an object or group of objects. An event may be associated with data, such as change information or flags. Where an event should alter the state of another object or initiate some processing, there needs to be a mechanism for propagating the event from its producer to an interested consumer.

The event flow is always from producer to consumer, but the control flow need not be. There consumer may have the event pushed onto it by the producer or it may pull it explicitly from the producer.

The Pull Model

- This is the polling model, where the consumer calls into the producer to query for events
  - Control flow is in opposite direction to the event flow
- Polling gives the consumer full control of execution policy, but it has to do all of its own dispatching
  - Polling may be blocking or non-blocking



In the pull model, the consumer has the thread of control and polls the producer for event data. Polling may be blocking. Alternatively, non-blocking polls allow better CPU usage through simple multi-tasking. The pull model requires the consumer to know something about the producer's interface, but not necessarily the other way around.

The Push Model

- This is the traditional callback model, where the producer calls into the consumer directly
  - Control flow is in the same direction as the event flow
  - Explicit Interfaces may be used to express the callback interface
- Event notification may be general or type specific

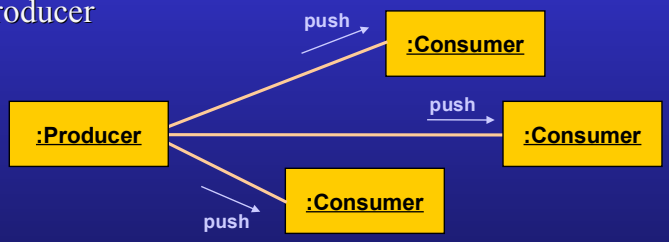
General notification
Specific notification (alternative)

```
interface Consumer
{
    void push(Event event);
    void onSomeEvent(SomeEvent event);
    ...
}
```


In the push model, the producer has the thread of control and initiates callbacks on the consumer to provide it with event data. The consumer is reactive in this relationship. There are two basic interface styles for callbacks. In the general approach, a narrow interface through which all event data is passed is defined, leaving the consumer to sort one event type out from another. Alternatively, event types can be differentiated by method. The push model requires the producer to know something about the consumer's interface, but not necessarily the other way around.

Multiple Notification

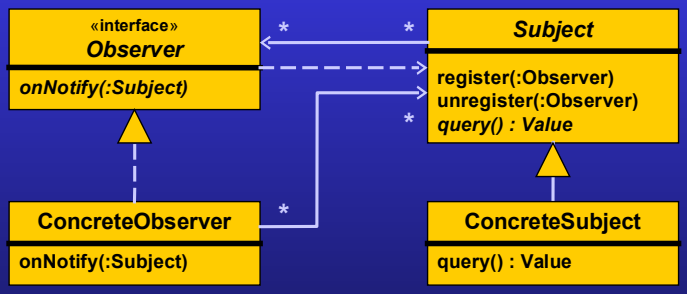
- A single event producer may be associated with multiple event consumers
  - Introduces issues of dependency and relationship management if push model is used
  - But apart from this issue, consumers can be lightweight, with all the implementation complexity gathered in the producer



Notification is sometimes point to point, between a single producer and a single consumer. More often, however, it involves a single producer and multiple consumers. This means that there needs to be management of this one-to-many relationship, and that there may potentially be a cyclic relationship between producer and consumer types: the consumer may hold a pointer back to the producer or it may be passed one by the consumer on callback.

The Observer Pattern

- Observer decouples controlling and viewing a model from the model itself
  - Basis of many notification designs, from OO framework Model–View to message-based Publisher–Subscriber



The intent for the Observer pattern is

> Define a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically.

Two immediately relevant situations for using Observer are given:

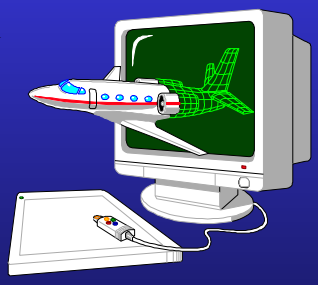
> When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
>
> When an object should be able to notify other objects without making assumptions about who these objects are.

The basic configuration is defined in terms of an abstract subject class, from which an actual business object class is derived, and an observer class, from which a user interface object may be derived. The framework provides for update notification between them. Multiple Observer objects can now be supported, for instance making user interface design more flexible – multiple, consistent views of the business objects can be handled simultaneously.

The Observer pattern is also known as Publisher–Subscriber. It lies at the heart of the Model–View–Controller framework-level pattern and its many variations.

The Model–View–Controller Pattern

- MVC is used in many interactive architectures
  - Traditionally focused on UI frameworks, but also applicable in other situations
- There are three main roles played by objects in the MVC architecture...
  - *Model* objects are the informational objects being observed
  - *View* objects observe model objects
  - *Controller* objects control how view objects respond to events



How can user interface logic be separated from the information to be presented? Any solution should take into account the following forces: the same information can be represented in different views, each of which may render the information differently; views must reflect changes to the information immediately (or promptly within an appropriate tolerance); information must respond immediately (or promptly) to changes initiated in the views; user interface should be decoupled from the information so that it can be changed independently, e.g., for different look and feel.

Model–View–Controller is an architectural pattern implemented originally in a Smalltalk event-driven GUI framework. It is now a common framework pattern that can be found both within and outside GUI frameworks. MVC is a course-grained pattern built from more granular design patterns. The overall architecture emerges from these smaller collaborations.

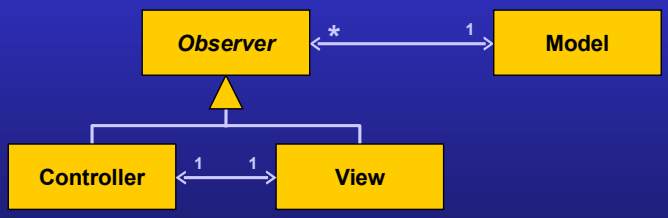
MVC is often applied to separate presentation management from core program logic [Buschmann+1996]:

> The Model–View–Controller architectural pattern (MVC) divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

Although commonly found and discussed in the context of GUI applications, it can be used in other interactive and event-driven environments.

Typical MVC Configuration

- Observer provides the core for MVC
  - MVC further partitions the responsibilities with inter-observer relationships
  - Many frameworks claiming to implement MVC actually merge view and controller roles, so they are more simply classified as Observer or Model–View architectures



The Model–View–Controller architecture can be expressed in terms of a number of common, OO framework patterns [Gamma+1995]:

Observer: Decouples views from models and keeps different views in synch with one another and the model. Views handle presentation logic and model objects represent the core functionality and information.

Composite: Views can be nested, so that nested and nesting views are treated uniformly.

Strategy: Different algorithms or policies can share the same interface, decoupling them from their client. A view object uses a controller object to determine how to respond to events.

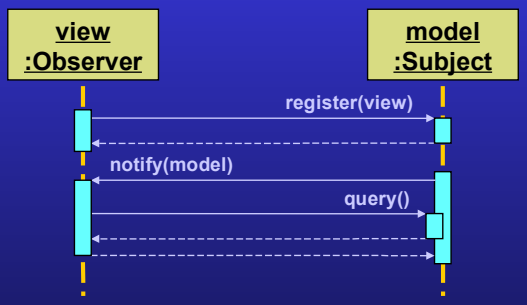
Factory Method: Decouples creation of an object that satisfies a particular interface from its class. This pattern is used in MVC to provide the default controller class for a view.

Decorator: Capabilities can be added to an object by communicating with it via an object with the same interface that adds the capability and then forwards the request. Typically used to add display features to a view.

Each smaller pattern addresses a particular set of issues, extending the flexibility of the architecture in a particular direction.

Re-entrancy

- Event notification from a subject object to its observer clients is handled through callbacks
  - Client may then call back in to subject, so subject must be in a valid state and have completed any changes before issuing the notification



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 details the basic callback form for one to many dependencies; Model–View–Controller 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.

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, 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.

Combined and Batched Notifications

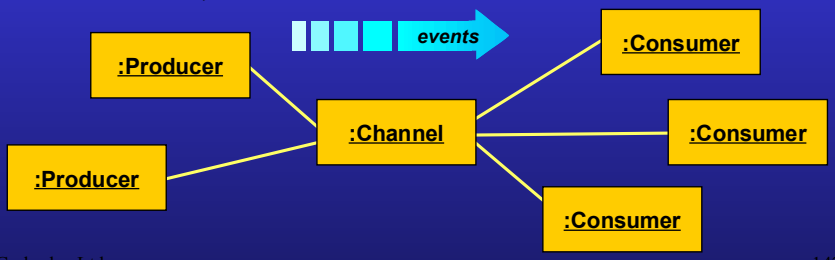
- Concurrency and distribution can affect the definition of a suitable notification interface
  - Granularity of conversation between producer and consumer needs to be reconsidered
- Notification may include details of changed data
  - Use of Combined Method saves the observer having to call back into the subject
- Multiple notifications may be grouped together
  - Use of Batch Method reduces number of calls
  - Notification is not necessarily immediate

In a concurrent system, either single-process multithreaded or multi-process, there are coherence and efficiency issues related to requiring a push notification to be followed by an information pull by the observer from the subject. A Combined Method delivers both notification and event data, rather than leaving it to the observer to figure out.

In a distributed system, whether between different processes on the same host or different processes on different hosts, many single notifications from subject to observer can flood the bandwidth with many trivial messages. A more efficient use of bandwidth is to use a Batch Method for notification. The subject may choose to batch up calls according to number, time or some other parameterisable quality-of-service value. It may also be designed to receive changes in batch.

The Event Channel Pattern

- Greater decoupling in execution policy is made possible by introducing event queues
  - Can have many producers and many consumers
  - Can support separate threading of notification
  - Producers are anonymous with respect to their consumers, and vice versa

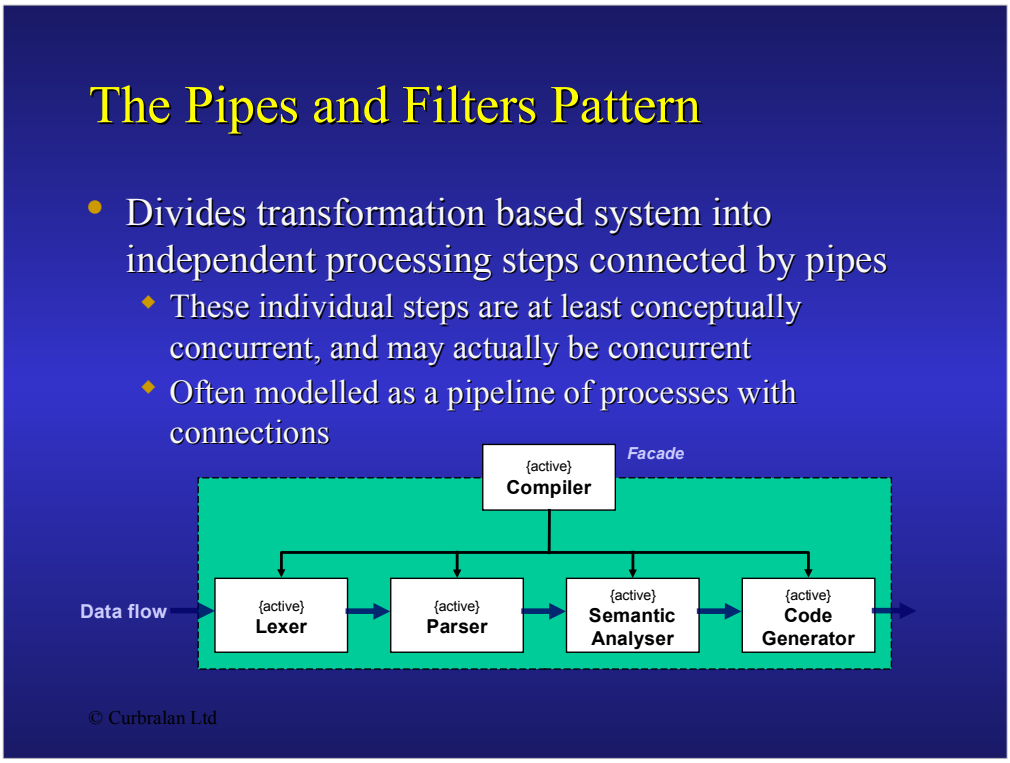


Further decoupling is also possible by introducing an intermediate message or event queue. The decoupling may be in terms of types, numbers of producers and consumers (accommodating many of each), identities of producers and consumers, execution (i.e. it may be asynchronous rather than synchronous), or any combination of these.

Message Queues

- A variation on eventing and publish–subscribe is point-to-point, asynchronous messaging
  - In contrast to classic publish–subscribe, sender and receiver identity are normally significant for communication via asynchronous message queues
- Message queues are used for their persistence, resilience and asynchronous properties
  - Point-to-point communication with non-blocking calls
  - The sender sends a message (a kind of Command) as a fire-and-forget call and picks up results explicitly via a *future* (a kind of Proxy) or via a callback

Mainstream programming languages tend to be tied to a very sequential and procedural view of computation, which makes the natural model for working with APIs procedural, based on the synchronous call-and-return control flow that follows from stack-based architectures. However, providing a synchronous API over a distributed infrastructure does not always scale. Opportunities for concurrency and absorption of latency are missed, making many RPC-based approaches scale poorly. A loosening of the procedural approach leads to the notion of asynchronous messaging. However, as this is not a native model for programming languages such as Java, there is a need to express the mechanisms through a framework that is manipulated explicitly via an API, so that the use of messaging is perhaps not as transparent as that of RPCs, but the operational benefits outweigh any loss of transparency.



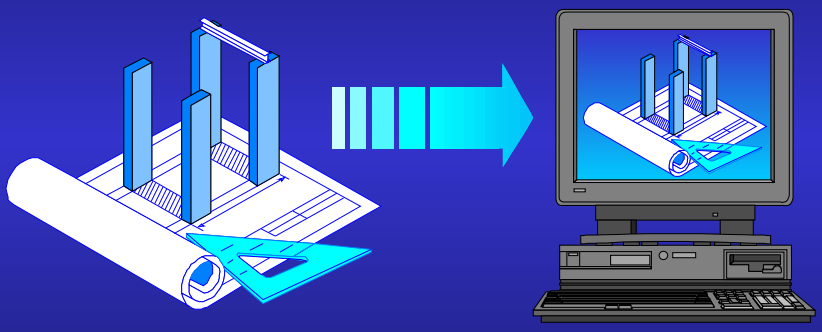
The Pipes and Filters architecture is a common and widely understood architecture [Frank Buschmann et al, Pattern-Oriented Software Architecture, Volume 1]. It underpins transformation oriented systems such as compilers, signal processors, batch sequential systems, etc. Its most popular manifestation is as the pipeline, which is a strict linear chain of transformations.

Pipes and Filters provides a framework for decoupling processing components, supporting simple composition, enhancement, reuse and parallelism. An example of enhancement would be the addition of additional optimising stages in the compiler shown above, and further opportunity to modify the optimisation strategy by plugging an alternative component. The most obvious drawback of Pipes and Filters is that it cannot easily handle interaction. The emphasis placed on the transformation of data flows may lead to a great deal of effort directed at the interpretation and rendering of the data streams at each filter.

There is a very strong relationship between architectural patterns and Jackson's Problem Frames. A problem frame describes the general shape of a problem in terms of the system to be built, the application domains, and the relationships and constraints on them. Each problem frame is associated with a method for solving it. The Transformation frame most corresponds closely to Pipes and Filters, identifying the problem in terms of transformation of sequential streams bound by a grammar.

Clearly problems do not always fit into single frames. Partitioning a problem into multiple frames helps inform decisions on architecture, such as whether to apply object-orientation strategically or tactically. For instance, an optimiser lends itself to the use of a Blackboard architecture; an IDE can be typified by the Workpieces frame; the application of the Facade pattern provides a simple entry point into the compiler system shown, e.g. in the case of traditional C compilers each major component of the compiler is accessible separately from the driver program.

Pattern Pitfalls



Pattern-Based Software Development

Pattern Pitfalls

- Objectives
  - Close with some considerations of the common ways in which patterns are misunderstood or misapplied
- Contents
  - Common pitfalls
  - Pattern applicability and quality
  - Reference implementations and blueprints
  - Dysfunctional patterns and applications
  - The Getters and Setters "pattern"
  - The Singleton pattern (and avoiding it)

Common Pitfalls

- Many pitfalls in the understanding and application of the pattern concept and specific patterns
  - The assumption that there are only 23 design patterns (the Gang of Four book)
  - The assumption that design patterns are related only to object orientation
  - A class diagram that demonstrates a pattern solution is the pattern itself
  - Assuming that anything that is identified as a pattern is necessarily good
  - Applying a pattern out of context

Patterns apply to all styles of development, not just object orientation. Although they have been popularised most in the OO community, they started out in building architecture, first emerged in software in GUI design, later in C++ idioms, and then were picked up when most strongly when considered in the context of OO design. However, they have since been applied to telecommunications, analysis, web design and numerous other diverse domains.

A common assumption is that the class diagram is the pattern. This is clearly untrue for a number of reasons, not least of which is the obvious refutation that patterns that are not expressed in terms of classes will clearly not have a class diagram! A pattern is a whole that includes the context of the problem all the way through to the consequences of the solution: a diagram forms only one element. There are often cases when two class diagrams look the same, but represent quite different solutions, and therefore quite different patterns: the motivation, the problem being solved, determines the pattern, not just the solution.

And one of the problems that plagues many projects is to assume that the only design models suitable are the ones in the Gang of Four book, and just apply them willy-nilly without consideration of the design problem. Patterns only represent good designs for particular problems: beyond the scope of these problems, such designs often represent poor solutions.

Pattern Applicability and Quality

- The recurrence of design idea is not necessarily an indication of its inherent quality
  - Software development is too often led by fashion rather than sound reasoning
  - Not all patterns are by necessity good patterns and not all applications of good patterns are necessarily good
- It is important to understand what problem a pattern solves, in what context this applies and what the consequences are
  - Patterns applied as a hammer-like tool make code more complex and obscure, not simpler and more elegant

It is often assumed that "pattern" is synonymous with "good", but this is not the case. Of course, it makes sense to focus on the good stuff, although there are cases for learning from counterexample, but the heart of a pattern is its recurrence. However, not all that recurs in design is necessarily a good pattern: many patterns can be considered bad. Indeed, the original inspiration for patterns, in the architecture of the built environment, was precisely because many patterns used for architecture in the 1950s, 1960s and 1970s were considered bad and that good patterns needed documenting. Some have chosen to rebrand this concept somewhat sensationally as "anti-patterns", but "bad patterns" is sufficient and more traditional.

Another problem for developers is the application of perfectly good patterns out of their context of applicability. The wrong design is the wrong design, no matter how good it is elsewhere.

Reference Models and Blueprints?

- A motivating example is sometimes taken to be a pattern's reference implementation
  - Copy-and-paste rarely gives a suitable solution, adaptation is always needed to fit the design together and have it make sense in context
- That a pattern is a blueprint or simply parameterized design is a common mistake
  - It is possible to generate code from such an approach, but at the expense of constraining the variation in the pattern, at which point it is no longer a "blueprint" of the pattern, just a parameterized application of it

A pattern is both a process and a thing. However, although a pattern is a thing, it's not a component. A pattern is also a process: it is not just a static snapshot of design or design thinking. A pattern must describe what to build, why and, importantly, how; each one is a highly specific mini-methodology. Therefore, patterns inform rather than dictate a design — they are not rigid and automatic, and must be considered and adapted every time. The reference or blueprint view of patterns has limited use and tends to miss the whole point of patterns as a vocabulary for thinking, reasoning and communicating.

Dysfunctional Patterns

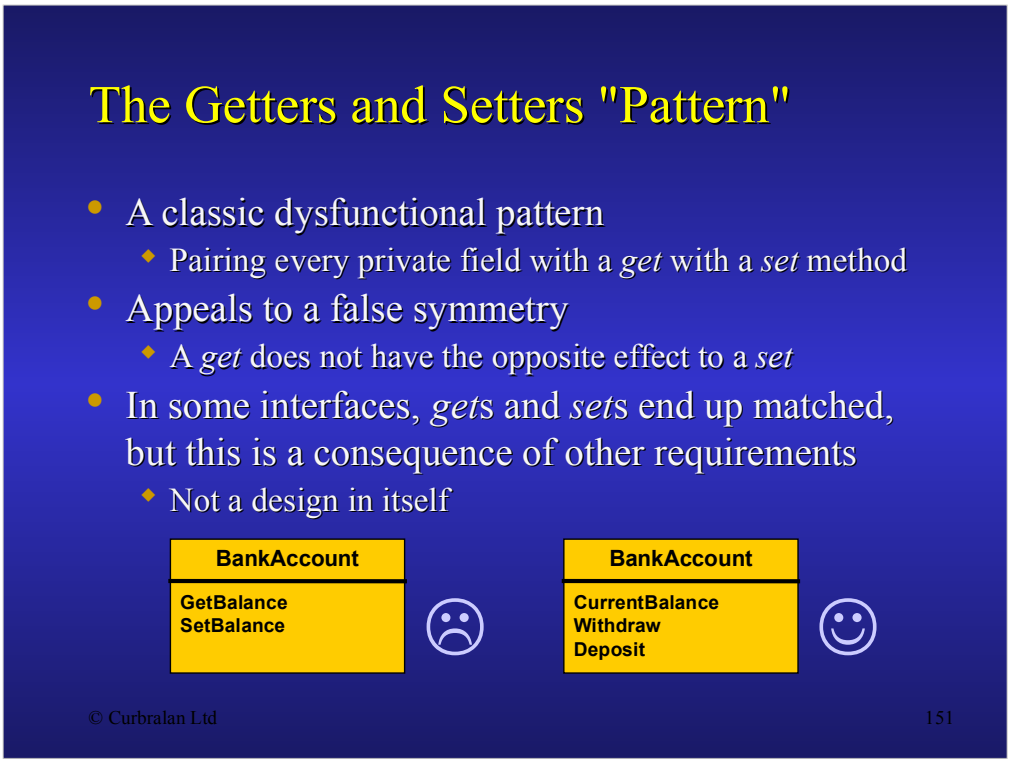
- A dysfunctional pattern is a design that recurs but whose forces and consequences are out of balance
  - E.g. the so-called Getters and Setters pattern
  - Many have the plausible appeal of solutions — but are often in search of a problem to solve
- Sometimes patterns are applied out of their appropriate context
  - Regardless of other qualities, the wrong solution is still the wrong solution
  - Singleton suffers this problem — it is rarely applicable

Any pattern describes a dialog with a design situation. It explores the context of a design problem, whether large scale or fine grained, enumerating the forces that drive and buffet the design. A pattern moves on to describe a solution. But, importantly, a pattern does not end there. The dialog continues by describing the consequences of applying the proposed solution, detailing the resulting context. What forces were balanced? What was left unresolved? What benefits have arisen? What liabilities have been incurred?

Patterns can be considered either "good" or "bad". When programmers normally talk about patterns there is an implicit assumption that they are talking about patterns that improve the quality of their systems, an implicit assumption that documented patterns are, almost by definition, of the "good" variety, so that anything that is a "bad" pattern is not a pattern.

However, if "bad" patterns are also considered to be patterns — the case originally — we can understand their dysfunctionality by trying to consider each as an unfulfilled whole — a whole with holes. A "good" pattern is one whose forces are both genuine and well matched by its consequences. A "bad" pattern, on the other hand, is one whose forces are incomplete and whose consequences are out of balance.

A pattern can also become dysfunctional if it becomes applied out of context, this newer application becoming the recurring use that holds in people's minds. This is in part the case with Singleton and the common use of Class Adapter as the first port of call for reusing code — the classic marketectural recommendation that inheritance increases productivity by encouraging reuse of existing code.

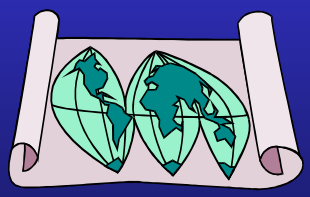


Some developers have used the phrase "the Getter and Setter pattern" assuming that the recurrence of the interface protocol is automatically, of itself, a good pattern. More experienced designers would reject both the "good"-ness of this design and its "pattern"-ness.

In truth, patterns, whether good or bad, have a recurring nature and recognisable form. By trying to understand Getters and Setters as some kind of pattern we can see it more easily in code; we can better understand why it is not a good pattern; most importantly, we can see how we might better address the design problem — preferably with a solution rather than another problem. This is not to say that objects should never have paired get and set methods — but if there is any guideline here, it is probably that any set methods should be accompanied by get methods, rather than the other way around — just to point out that when useful, such a pairing is a consequence of other design decisions and less a decision in itself.

The Singleton Pattern (and Avoiding It)

- The notion is that a Singleton object enforces a creational constraint on an object type
  - Only a single instance of a class can be created, and public, global access is provided to it via a *static* method
- Singleton is (very) frequently misused
  - Singleton is sometimes seen as a legitimate way of doing "OO global variables"
- Complicates unit testing and thread safety



The Singleton pattern, although one of the most commonly referred to patterns, is perhaps one of the most abused, misunderstood and incorrectly implemented patterns. It is often seen as giving legitimate licence to global variables, whose limitations are well documented. Some developers assume that since there are many objects that happen to have one instance in a particular program, they should apply Singleton and a global point of access.

The Singleton pattern is not simply about constraining instance numbers: it also encapsulates and defers the creation of instances until the point of demand. This is useful where it would be wasteful to create an instance outright (e.g. in a `static` initialiser) or difficult (e.g. an object per thread).

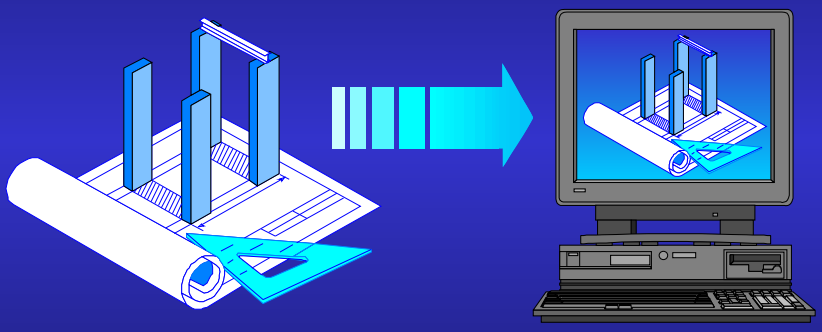
Methods on a Singleton are instance methods; only the instance access is a static. A common misimplementation is to assume that the class is the instance and have all methods as static. This emulates a module rather than a Singleton.

Alternatives to Using Singleton

- Singleton is often used to avoid passing a dependency around explicitly
  - Essential dependencies should be made explicit in an object's interface, not brushed under the carpet
  - If the objection is that an object would need to be passed everywhere, that is a clear indication the architecture is tightly coupled, and Singleton is a salve not a solution
- External dependencies should be encapsulated and passed around via an Explicit Interface
  - This is a key characteristic of patterns such as Context Object, Strategy and Mock Object

The widespread use of Singleton in an architecture is often an indication of tight coupling and/or "patternitis". Many reasons — often better characterised as excuses — are often given for this, but in the final reckoning there is most often an alternative solution that relies on a more cohesive object partitioning that is more flexible, stable and testable in the long term.

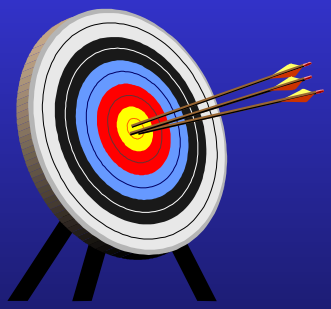
Course Outroduction

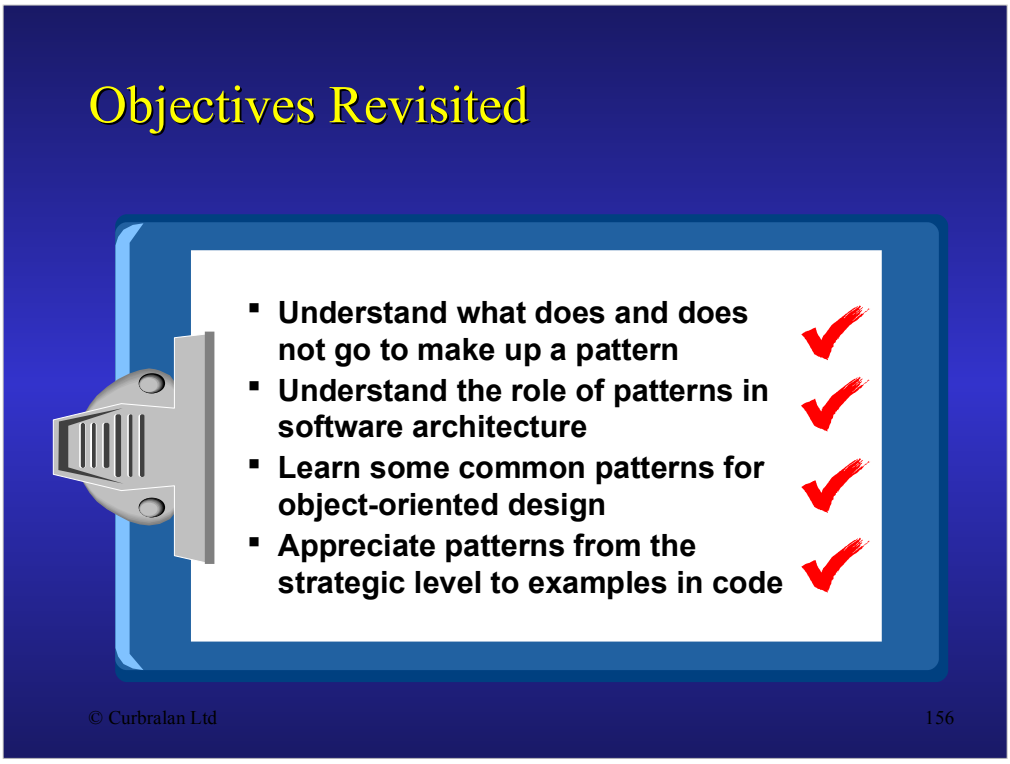


Pattern-Based Software Development

Course Outroduction

- Objectives
  - Wrap up and reflect on the course
- Contents
  - Check back against the objectives
  - Moving beyond the course
  - Summary

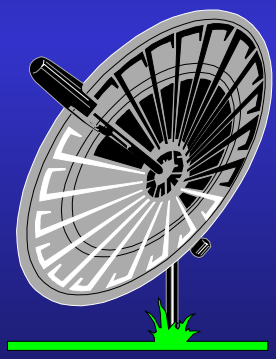




The main objectives of the course have hopefully been met for you, through a combination of lecture material, supporting notes, additional comments and questions. Patterns have been introduced as a way of capturing architectural knowledge and reasoning about design, not just as a shopping list of design templates. This should put you in good stead for taking them further and into a practical context.

Moving Beyond

- The course represents a beginning not an end to understanding patterns
  - Hands-on experience and further reading is recommended
- Related areas include...
  - Development process
  - Object-oriented modelling
  - Programming style



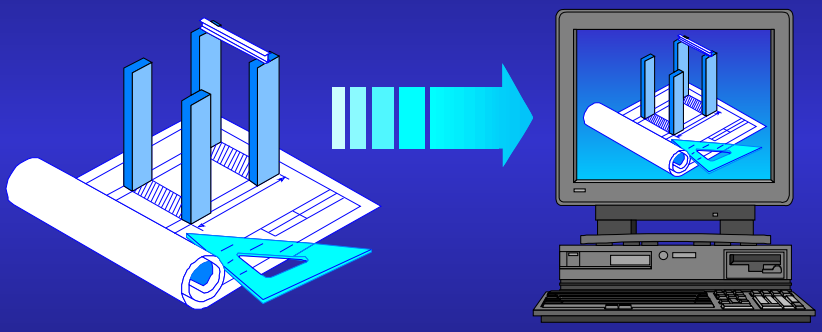
The course includes information useful for extending design vocabulary and reasoning through patterns. However, to make the most out of this requires practical experience, discussion and further reading, either online or treeware.

These techniques do not stand on their own, so it is important to see how they fit with other technologies, systems and techniques.

Summary

- Patterns provide a design vocabulary
  - Recurring and reusable architectural ideas
- They distil successful design experience
  - Solution structure is sensitive to details of purpose and context
- No pattern is an island
  - Patterns may relate to one another in intent, structure or sequence

Further Reading



Pattern-Based Software Development

Object-Orientation and Component-Based Development

Object Technology: A Manager's Guide, 2nd edition, David A Taylor, Addison-Wesley, 1998.

Object-Oriented Software Construction, 2nd edition, Bertrand Meyer, Prentice Hall, 1997.

Cetus Links: Object-Orientation, `http://www.cetus-links.org`.

Component Software: Beyond Object-Oriented Programming, 2nd edition, Clemens Szyperski with Dominik Gruntz and Stephen Murer, Addison-Wesley, 2002.

Essential COM, Don Box, Addison-Wesley, 1998.

Effective COM: 50 Ways to Improve your COM and MTS-Based Applications, Don Box, Keith Brown, Tim Ewald and Chris Sells, Addison-Wesley, 1999.

Advanced CORBA Programming with C++, Michi Henning and Steve Vinoski, Addison-Wesley, 1999.

Object Management Group, `http://www.omg.org`.

UML

UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edition, Martin Fowler, Addison-Wesley, 2004.

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

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

UML Components, John Cheesman and John Daniels, Addison-Wesley, 2000.

The Object Constraint Language: Precise Modeling with UML, Jos Warmer and Anneke Kleppe, Addison-Wesley, 1999.

UML Resource Center, `http://www.rational.com/uml`.

Development Process

Object Solutions: Managing the Object-Oriented Project, Grady Booch, Addison-Wesley, 1996.

Surviving Object-Oriented Projects: A Manager's Guide, Alistair Cockburn, Addison-Wesley, 1998.

The Unified Software Development Process, Ivar Jacobson, Grady Booch and James Rumbaugh, Addison-Wesley, 1999.

Manifesto for Agile Software Development, `http://agilemanifesto.org`.

DSDM Consortium, `http://www.dsdm.org`.

Extreme Programming Explained: Embrace Change, 2nd edition, Kent Beck, Addison-Wesley, 2005.

XProgramming.com, `http://www.xprogramming.com`.

Agile Software Development with Scrum, Ken Schwaber and Mike Beedle, Prentice Hall, 2002.

Agile Project Management with Scrum, Ken Schwaber, Microsoft Press, 2004.

Scrum Development Process, `http://www.controlchaos.com`.

Lean Software Development, Mary Poppendieck and Tom Poppendieck, Addison-Wesley, 2003.

Organizational Patterns of Agile Software Development, James O Coplien and Neil Harrison, Prentice Hall, 2005.

Slack, Tom DeMarco, Broadway Books, 2001.

The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, Eric S Raymond, O'Reilly, 2000, `http://www.tuxedo.org/~esr/writings/cathedral-bazaar`.

Open Sources: Voices from the Open Source Revolution, edited by Chris DiBona, Sam Ockman and Mark Stone, O'Reilly, 1999.

Patterns of Software: Tales from the Software Community, Richard P Gabriel, Oxford, 1996.

Design and Architecture

Hillside Group Patterns Home Page, `http://hillside.net`.

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

Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal, Wiley, 1996.

Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects, Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, Wiley, 2000.

Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management, Michael Kircher and Prashant Jain, Wiley, 2004.

Pattern Hatching: Design Patterns Applied, John Vlissides, Addison-Wesley, 1998.

Core J2EE Patterns: Best Practices and Design Strategies, 2nd edition, Deepak Alur, John Crupi and Dan Malks, Prentice Hall, 2003.

Patterns of Enterprise Application Architecture, Martin Fowler, Addison-Wesley, 2003.

Small Memory Software: Patterns for Systems with Limited Memory, James Noble and Charles Weir, Addison-Wesley, 2001.

Pattern Languages of Program Design, edited by James O Coplien and Douglas C Schmidt, Addison-Wesley, 1995.

Pattern Languages of Program Design 2, edited by John M Vlissides, James O Coplien and Norman L Kerth, Addison-Wesley, 1996.

Pattern Languages of Program Design 3, edited by Robert Martin, Dirk Riehle and Frank Buschmann, Addison-Wesley, 1998.

Pattern Languages of Program Design 4, edited by Neil Harrison, Brian Foote and Hans Rohnert, Addison-Wesley, 2000.

Domain-Driven Design, Eric Evans, Addison-Wesley, 2004.

Generative Programming, Krzystof Czarnecki and Ulrich W Eisenecker, Addison-Wesley, 2000.

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

The Design of Everyday Things, Donald A Norman, MIT Press, 1988.

The Nature and Aesthetics of Design, David Pye, Herbert Press, 1978.

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

The Timeless Way of Building, Christopher Alexander, Oxford, 1979.

Development Practice

Test-Driven Development: By Example, Kent Beck, Addison-Wesley, 2003.

Pragmatic Unit Testing in Java with JUnit, Andrew Hunt and David Thomas, The Pragmatic Bookshelf, 2003.

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

Refactoring Home Page, `http://www.refactoring.com`.

Working Effectively with Legacy Code, Michael C Feathers, Prentice Hall, 2005.

The Practice of Programming, Brian W Kernighan and Rob Pike, Addison-Wesley, 1999.

The Pragmatic Programmer, Andrew Hunt and David Thomas, Addison-Wesley, 2000.

Practices of an Agile Developer, Venkat Subramaniam and Andy Hunt, Pragmatic Bookshelf, 2006.

Writing Solid Code, Steve Maguire, Microsoft Press, 1993.

The Programmer's Stone, Alan G Carter and Colston Sanger, `http://www.reciprocality.org/Reciprocality/r0`.

Curbralan, various articles and links, `http://www.curbralan.com`.

Requirements and Analysis

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

Problem Frames, Michael Jackson, Addison-Wesley, 2001.

Structuring Use Cases with Goals, Alistair Cockburn, `http://members.aol.com/acockburn/papers/usecases.htm`.

Writing Effective Use Cases, Alistair Cockburn, Addison-Wesley, 2001.

Analysis Patterns: Reusable Object Models, Martin Fowler, Addison-Wesley, 1997.

C++

Accelerated C++, Andrew Koenig and Barbara E Moo, Addison-Wesley, 2000.

C++ Primer, 3rd edition, Stanley B Lippman and Josée Lajoie, Addison-Wesley, 1998.

Thinking in C++ Volume 1: Introduction to Standard C++, 2nd edition, Bruce Eckel, Prentice Hall, 2000.

The C++ Programming Language, 3rd edition, Bjarne Stroustrup, Addison-Wesley, 1997.

The C++ Standard Library: A Tutorial and Reference, Nicolai M Josuttis, Addison-Wesley, 1999.

Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd edition, Scott Meyers, Addison-Wesley, 2005.

Effective STL: 50 Ways to Improve Your Use of the Standard Template Library, Scott Meyers, Addison-Wesley, 2001.

Modern C++ Design, Andrei Alexandrescu, Addison-Wesley, 2001.

Exceptional C++, Herb Sutter, Addison-Wesley, 2000.

Curbralan, various articles and links, `http://www.curbralan.com`.

Java

Thinking in Java, 2nd edition, Bruce Eckel, Prentice Hall, 2000, `http://www.mindview.net/Books/DownloadSites`.

The Java Programming Language, 3rd edition, Ken Arnold, James Gosling and David Holmes, Addison-Wesley, 2000.

Java in a Nutshell, 4th edition, David Flanagan, O'Reilly, 2002.

Java Enterprise in a Nutshell, 3rd edition, Jim Farley and William Crawford, O'Reilly, 2006.

Concurrent Programming in Java: Design Principles and Patterns, 2nd edition, Doug Lea, Addison-Wesley, 2000.

Effective Java, Joshua Bloch, Addison-Wesley, 2001.

The Elements of Java Style, Allan Vermeulen, Scott Ambler, et al, Cambridge, 2000.

Curbralan, various articles and links, `http://www.curbralan.com`.

Dynamic Languages

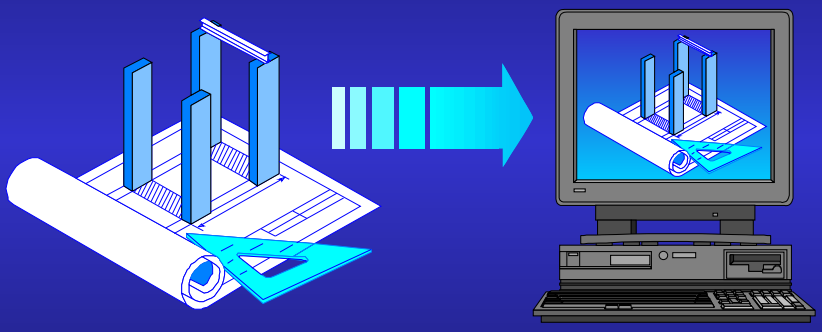
Programming Ruby, David Thomas and Andrew Hunt, Addison-Wesley, 2001.

Ruby in a Nutshell, Yukihiro Matsumoto, O'Reilly, 2002.

Ruby Home Page, `http://www.ruby-lang.org/en/index.html`.

Smalltalk Best Practice Patterns, Kent Beck, Prentice Hall, 1997.

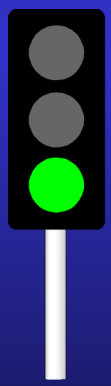
Unit Testing with JUnit



Pattern-Based Software Development

Unit Testing with JUnit

- Objectives
  - Provide an overview of the freely available JUnit testing framework
- Contents
  - TDD development cycle
  - Example-based test cases
  - Writing tests with JUnit 3.8
  - Writing tests with JUnit 4



Test-Driven Development Cycle

- Write tests along with production code
  - Test-first style is based on first writing a test case for new functionality (fails or fails to compile)...
  - Then writing production code that makes the test pass (no matter how simple)...
  - And then refactoring the code to make the next change simpler
- Test cases are used to scope design, offer instant feedback on code and regression test

TDD has emerged from the many practices that form the core of Extreme Programming. The focus is on the code-centric practices in the micro process rather than driving the macro process. Most commonly, TDD is used other macro-process models than XP, e.g. Scrum, RUP, DSDM.

Testing is considered to be about promoting visibility, building confidence and defining completion of what is essentially an invisible act. Testing is considered to be a part of development, rather than something that follows it as an extra.

Part of what distinguishes TDD from just being unit testing — of which there are many approaches — is the motivation and reasoning behind the tests. Unit tests are not planned in advance and they are not written and executed some indefinite time after the units under test. They are written alongside the unit being written, so that code and test writing is interleaved. This offers a visible measure of progress for the individual developer and keeps the interface of the unit focused, avoiding unnecessary generalisation. As such, active test writing supports the design of the code directly rather than being a passive afterthought.

Example-Based Test Cases

- The structure of individual test cases is to demonstrate functionality by example
  - Therefore, the tests are functional as opposed to performance or other operational tests
- Tests focus on, and use, the interface of a unit and not its internal structure
  - Examples with specific values are used, not test cases based on the code's internal structure

TDD uses a scenario-based approach for defining the behavioural contract for a unit of code. Examples that should succeed are written in the same language as the code unit, and demonstrate a simple or complex sequence of code using that unit and then some required outcomes. This means that the tests are functional by nature rather than performance or resource focused.

JUnit

- JUnit is a widely used, open source unit-testing framework for Java
  - Developed originally by Erich Gamma and Kent Beck, based on Kent Beck's SUnit for Smalltalk
  - Part of the broader xUnit family, which includes frameworks for C++, .NET, Python, etc
- JUnit can be run as a console or GUI application, or as part of an IDE

There are many members of what is known as the xUnit family, including versions for C++ (CppUnit) and Python (PyUnit). However, perhaps the best known is the Java flavour, JUnit. Although it is not hard to write a test framework with the minimal functionality that JUnit has, the straightforward framework model and free availability of JUnit has led to its widespread adoption in TDD circles. It is integrated with many IDEs.

JUnit Versions

- JUnit has long been stable at version 3.8
  - This is a benefit to tools that depend on JUnit...
  - But stunts the evolution of the tool in line with experiences learnt from JUnit and other tools
- JUnit 4 is a more recent and offers a programmatically different approach
  - Based on a less-intrusive annotation interface, in contrast to an inherited approach, borrowing ideas from other frameworks, such as NUnit and TestNG

The stability of JUnit has been a mixed blessing. It has accumulated some historical aspects to its interface that can be considered less than idiomatic from a modern Java perspective, and there are common concerns that have not been addressed and cannot easily be addressed given its existing design — particularly for integration testing. JUnit 4 offers backward compatibility but also some degree of innovation.

Test Cases in JUnit 3.8

- A test case class defines multiple tests to be executed independently in their own instances
  - Test case classes normally named with a *Test* suffix and must subclass *junit.framework.TestCase*
  - Test methods need to be named with a *test* prefix
- A test case class is effectively a test suite and the class can be executed as by a test runner
  - IDEs, such as Eclipse, have their own built in test runner with a GUI

A test case instance is created with the name of the test method that it is to be run, so all `TestCase` subclasses should support at least a single-argument constructor that takes a string and initialises the superclass with that string.

A test suite represents and runs a collection of tests: tests may be individual test cases or other suites. A test suite can be constructed manually by adding named test cases one at a time. This is explicit but, for the common case, typically error prone. It becomes inconvenient because for every new test method added, an `addTest` call with a correctly named test case instance must be made. Given the inevitable copy-and-paste nature of this task, it can be frustrating to discover that a new test apparently passes when it shouldn't, only to discover that it was a typo in the test suite setup. A simpler and more direct approach to populating a test suite is to create a new suite and pass in the class of the test case class. The constructor then uses reflection to pull out all of the methods whose name begin with `test` and that take no arguments. There is also an `addTestSuite` method that does the same.

A test runner takes a nominated suite and executes it. A suite is extracted either by calling a static `suite` method or by finding all `test` methods in the class. IDEs normally include a runner and automate this, so programmers don't need to deal directly with suites, just test-case classes.

Defining Test Cases in JUnit 3.8

```
import junit.framework.TestCase;
public class RecentlyUsedListTest extends TestCase
{
    public void testInitialListIsEmpty()
    {
        RecentlyUsedList list = new RecentlyUsedList();
        assertTrue(list.isEmpty());
        assertEquals(0, list.size());
    }
    public void testSingleItemInsertionIntoEmptyList()
    {
        RecentlyUsedList list = new RecentlyUsedList();
        list.add("Bristol");
        assertFalse(list.isEmpty());
        assertEquals(1, list.size());
        assertEquals("Bristol", list.get(0));
    }
    ...
}
```


Writing a set of unit tests is therefore a simple matter of inheriting from `junit.framework.TestCase` and implementing public methods prefixed with `test` and taking no arguments. Test methods set up situations and assert on outcomes using the standard set of JUnit assertions. Note that no constructor is required for the typical case of usage and that tests are isolated from one another in terms of effect.

Assertion Methods

- The *Assert* class is a commodity class
  - Can be used as a module, so can use *import static*, or as a superclass, which it is for the *TestCase* class
- It defines a number of *static* assertion methods
  - They are generally focused on simple comparisons
  - They come in two flavours: with or without an additional argument for a message
  - They throw *AssertionFailedError*, or a suitable subclass, in the event of a failed assertion

The `Assert` class is a stateless, `static`-only class not intended for instantiation. It uses a `protected` constructor to enforce this constraint, as opposed to being declared `abstract`. Its assertion methods can be used independently, in a module fashion, i.e. import static `junit.framework.Assert.*`, but within the `TestCase` hierarchy is its most common use.

Note that `AssertionFailedError` is, as its name suggests, a proper error rather than either an unchecked runtime exception or checked exception. This means neither it nor `Error` should be caught explicitly in user code.

assertTrue and *assertFalse*

- The *assertTrue* method can be considered the most primitive assertion needed
  - *assertFalse* provides its natural complement

```
public class Assert
{
    ...
    public static void assertTrue(boolean condition) ...
    public static void assertTrue(
        String message, boolean condition) ...
    public static void assertFalse(boolean condition) ...
    public static void assertFalse(
        String message, boolean condition) ...
    ...
}
```


Historically, `assertTrue` was simply called `assert`. However, this clashed with the Java `assert` keyword introduced in J2SDK 1.4, so it was renamed. `assert` was deprecated in JUnit 3.7 and then withdrawn in JUnit 3.8. This also explains why the `assertFalse` method was missing from JUnit 3.7 — an annoying asymmetry in the design of the interface.

assertEquals

- The *assertEquals* methods compare expected against actual values
  - Objects are compared using the *equals* method and simple values using the == operator

```
public class Assert
{
    ...
    public static void assertEquals(
        Object expected, Object actual) ...
    public static void assertEquals(
        int expected, int actual) ...
    public static void assertEquals(
        double expected, double actual, double delta) ...
    ...
}
```

Other overloads omitted for brevity

The main workhorse of the `Assert` class is the family of `assertEquals` methods. There are currently no `assertNotEquals` methods, so if such assertions are required they must be phrased explicitly using `assertTrue` or `assertFalse`. Similarly, there are no assertions for relational comparisons.

Although not shown above, all variants of `assertEquals` include an overload that takes a message as the first argument. There are variants for all of the built-in value types. The overload for `float` includes a delta and there is also a specific overload for `String` that is distinct from the one for `Object`.

assertSame and *assertNotSame*

- *assertSame* and *assertNotSame* are based on identity comparison of objects
  - Whereas *assertEquals* on objects uses the *equals* method for comparison

```
public class Assert
{
    ...
    public static void assertSame(
        Object expected, Object actual) ...
    public static void assertNotSame(
        Object expected, Object actual) ...
    ...
}
```

Other overloads omitted for brevity

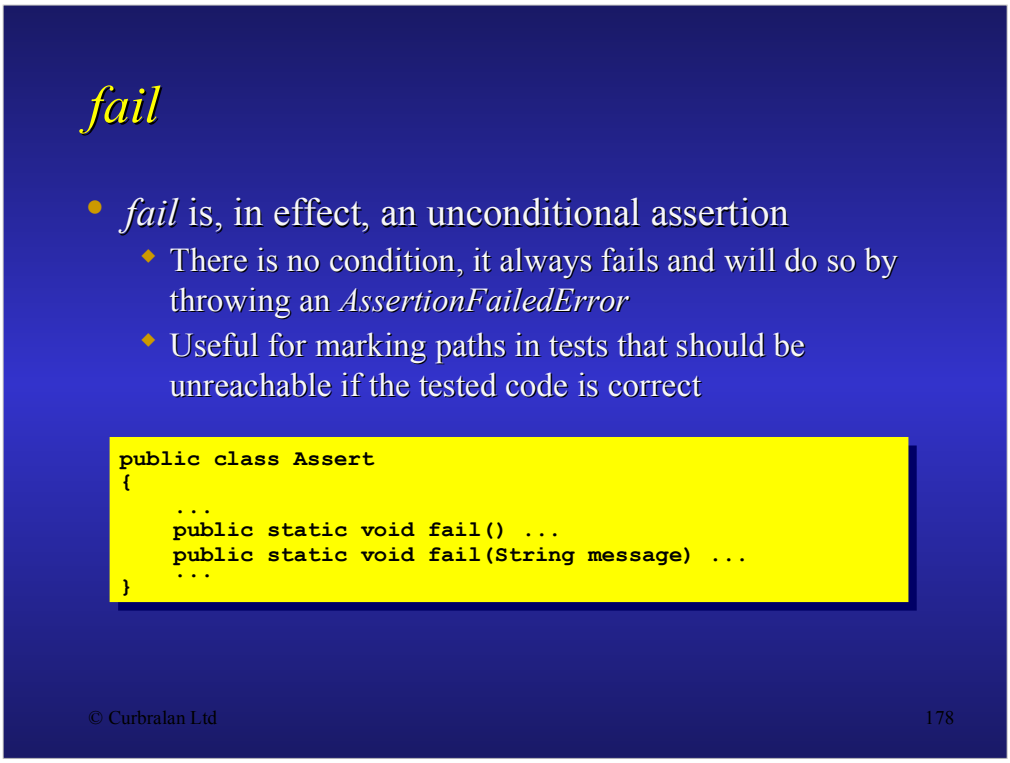
Although not shown above, both `assertSame` and `assertNotSame` have overloads that that take a message as the first argument. There are no variants for the built-in value types because they would be equivalent to the corresponding `assertEquals` methods. Note that the `assertNotSame` method is missing from versions of JUnit before 3.8.

assertNull and *assertNotNull*

- The *assertNull* and *assertNotNull* methods cater for a common requirement
  - More descriptive than using *assertSame* and *assertNotSame* or *assertTrue* and *assertFalse*

```
public class Assert
{
    ...
    public static void assertNull(Object object) ...
    public static void assertNull(
        String message, Object object) ...
    public static void assertNotNull(Object object) ...
    public static void assertNotNull(
        String message, Object object) ...
    ...
}
```


The `assertNull` and `assertNotNull` address a common need that would otherwise have to be met by using `assertSame(object, null)` and `assertNotSame(object, null)`.



The `fail` methods are classified as assertions, but they are odd assertions, in effect equivalent to `assertTrue(false)`. Object states and outcomes that can be tested by predicates work fine with the other assertions, but for some tests it can be easier simply to mark what should be an unreachable part of a test with a `fail`. For example, the `default` case of a `switch` statement if only the explicitly labelled cases should ever be executed.

Testing Correctness of Exceptions

- There are two approaches to testing the correctness of exceptional outcomes
  - Use the *fail* method
  - Create an *ExceptionTestCase* to wrap the call to a test method

```
try
{
    new Date(2001, 7, 32);
    fail("Invalid date should throw a RuntimeException");
}
catch(RuntimeException caught)
{
}
```


The `fail` approach has the benefits of being explicit, requiring no additional test suite set up and allowing more than one exception outcome to be asserted in a test method. However, frequent use can make the code verbose.

The `ExceptionTestCase` approach allows a test to be added to a suite and have the outcome of that test matched with an exception type, so that the test fails if an exception of that type is not thrown:

```
suite.addTest(
    new ExceptionTestCase(
        "testInvalidDate", RuntimeException.class));
```

The `ExceptionTestCase` class is defined in the `junit.extensions` package, rather than in `junit.framework`.

Overriding *setUp* and *tearDown*

- Hook methods in the *TestCase* class can be overridden to run before and after a test
  - By default *setUp* and *tearDown* do nothing
  - *setUp* can be used for common initialisation code and *tearDown* is most useful in integration testing

```
public abstract class TestCase ...
{
    ...
    protected void setUp() ...
    protected void tearDown() ...
    ...
}
```

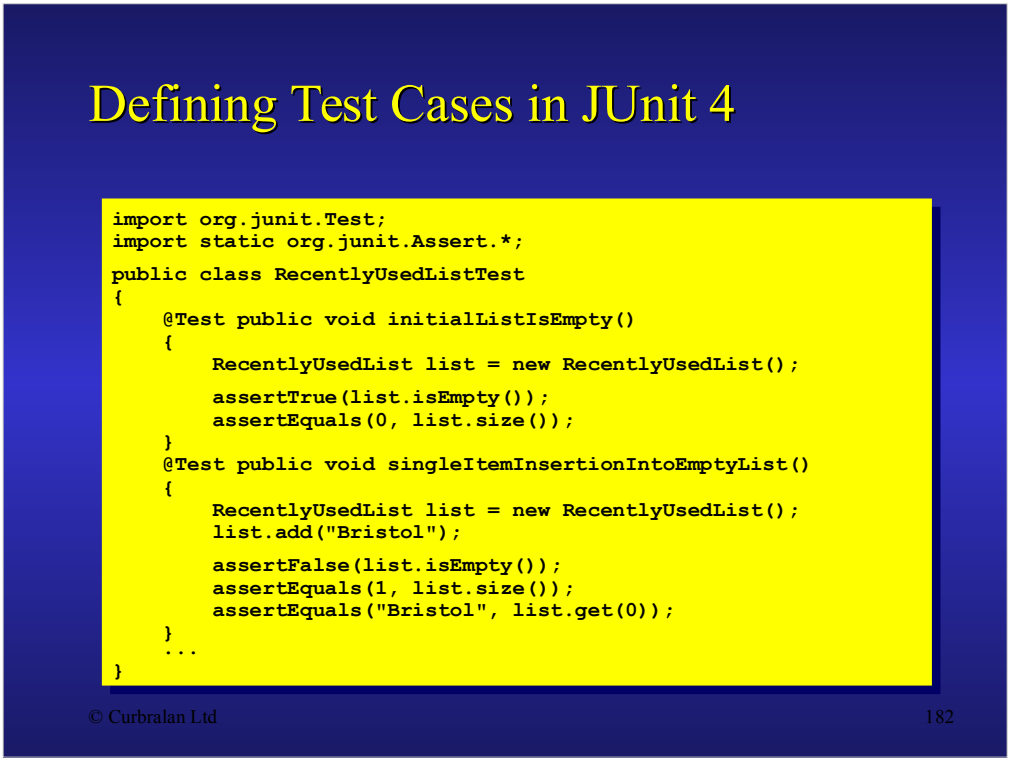

The relationship between the Command running methods in the `TestCase` class and the `setUp` and `tearDown` methods is that of Template Methods.

If the initialisation common is simple and not liable to throw an exception, for example setting the values of a few fields, there is no need to override the `setUp` method: simple field initialisers suffice. Introducing a `setUp` method in such circumstances is not particularly appropriate. A good rule of thumb is that if you need a `tearDown`, you should probably also have a `setUp`. Another rule of thumb is that if you need `tearDown`, the test in question is an integration rather than a unit test, or it is working around an inappropriate dependency in the code, such as a Singleton.

Test Cases in JUnit 4

- In contrast to classic JUnit, JUnit 4 does not require any inheritance for test case classes
  - They are just ordinary classes, typically with a *Test* suffix in their name
- Test methods are annotated using *@Test*
  - This is from the *org.junit* package
  - There is no need to prefix the test method with *test*
- May need to use *JUnit4TestAdapter* to execute test cases in old test runners

The key difference between classic JUnit and JUnit 4 is that JUnit 4 is based on a less intrusive framework style, one based more fully on reflection than on inheritance. JUnit 4 takes advantage of JDK 5's support for annotations, so it will not run with older Java compilers.



Writing a set of unit tests in JUnit therefore requires no inheritance or special status for the test fixture class. An ordinary class with public methods that take no arguments and are annotated with `@org.junit.Test` is all that is required. The standard set of JUnit assertions is accessed using `import static`.

Assertions in JUnit 4

- The assertions supported are the same limited set as found in earlier versions of JUnit
  - I.e. *assertTrue*, *assertFalse*, *assertEquals*, *assertSame*, *assertNotSame*, *assertNull*, *assertNotNull* and *fail*
  - The assertions are defined in *org.junit.Assert*
- However, in use the assertion methods are normally imported
  - I.e. *import static org.junit.Assert.**

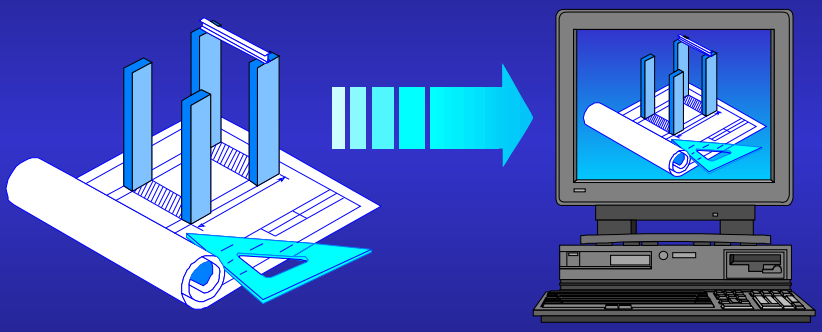
The assertions in JUnit 4 are the same as those in JUnit 3.8, which unfortunately means that a number of useful assertion methods are still missing, e.g. `assertNotEquals`. The main difference is that they are defined in `org.junit` and that their usage is normally through `import static` rather than inheritance or full package qualification.

Other Annotations

- A test method can be ignored
  - Use the *@Ignore* annotation, which optionally takes a string describing why the method is ignored
- Code can be executed before and after each test method is executed
  - Use the *@Before* and *@After* annotations
- Expected exceptions can be annotated
  - Use *@Test(expected=...)* on a method

JUnit 4 supports a number of other annotations that help to simplify the expression of testing logic, and move away from the inheritance-based view of classic JUnit. That said, although the ability to specify expected exceptions is convenient, it is overly strict in expecting an exact type match.

Workshop



Pattern-Based Software Development

Workshop

- Objectives
  - Provide details for the two workshop exercises
- Contents
  - Workshop guidelines
  - Exercise: note organiser
  - Exercise: displaying and filtering files

Workshop Guidelines

- The workshop approach is based on common agile and OO development practices
  - Sketch out models using appropriate UML diagrams
  - Break down the requirements into smaller objectives and prioritise them
  - Determine patterns that help with each objective, but do not restrict this just to patterns in the course
  - Aim to use TDD to elaborate and validate code detail
  - Refactor as problems or new design opportunities become obvious
  - Program in pairs

It is worth starting each exercise by understanding and walking through the requirements and likely solution with appropriate, lightweight UML models. Use cases can help with reorganising and prioritising requirements. Prioritisation should guide the order of development, i.e. higher-priority items before lower priority ones.

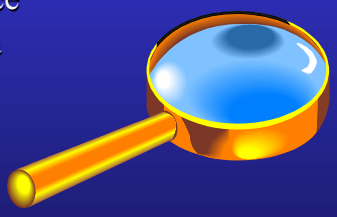
Make sure that you build up the tests iteratively and incrementally, rather than coding out all of the tests or all of the implementation at once, and only then tackling the implementation or the tests. Test the simplest things first and then build up to more sophisticated tests. Assume that you will refactor, so that your implementations can be almost ridiculously simple to pass the initial tests, and become more complex as the tests demand and grow.

Also ensure that you provide no more features than make sense for the problem — it is easy to get carried away and add features that are not strictly necessary! Remembering that you should not introduce a feature that you do not intend to test should keep surplus features in check.

As you develop your tests, keep in mind that it is often helpful and clearer to group tests into separate methods rather than holding them all in a single, long method. Make sure that in each example the code you are implementing is in a separate source file from the source file containing the tests. In terms of coding guidelines, try to follow common Java naming conventions for class and method names where appropriate.

Exercise: Displaying and Filtering Files

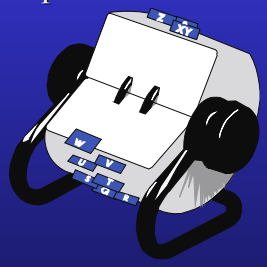
- Consider a set of classes that allow a text file to be printed out and filtered in various simple ways
  - Print out the content of a file stream directly to an output stream
  - Add line numbers
  - Shift case to either upper or lower case
  - Trim leading and trailing space
  - Only print lines that contain a particular string
  - Ignore the input
  - Pretty print code



Shown above is just a selection of the possible filtering and display options available. These should be selectable at runtime. Note that by "pretty print", simple and consistent indentation with respect to opening and closing curly braces is assumed.

Exercise: Note Organiser

- Consider a simple package that allows a user to enter and organise textual notes
  - They can enter text under a title or they can link to a named file
  - They can group notes together under common headings, and also further categorise these groups
  - They can remove items
  - They can search and list items



There is no single intended use for the organiser: a user can choose to use it as an address book, a diary, arbitrary notes, etc. Although it is not a requirement to provide even a text-driven console UI, this may be a useful supplement to automated unit tests.

Exercise: Spreadsheet

- Consider a simple package that allows a user to enter values in cells in a grid
  - A cell may be a constant value or a simple formula based on other cells' values
  - A formula can be addition (+) or subtraction (-) of two values
  - Cells get updated when their dependencies change
  - Ensure that only changed cells get recalculated
  - Consider how to support multiple views of the grid