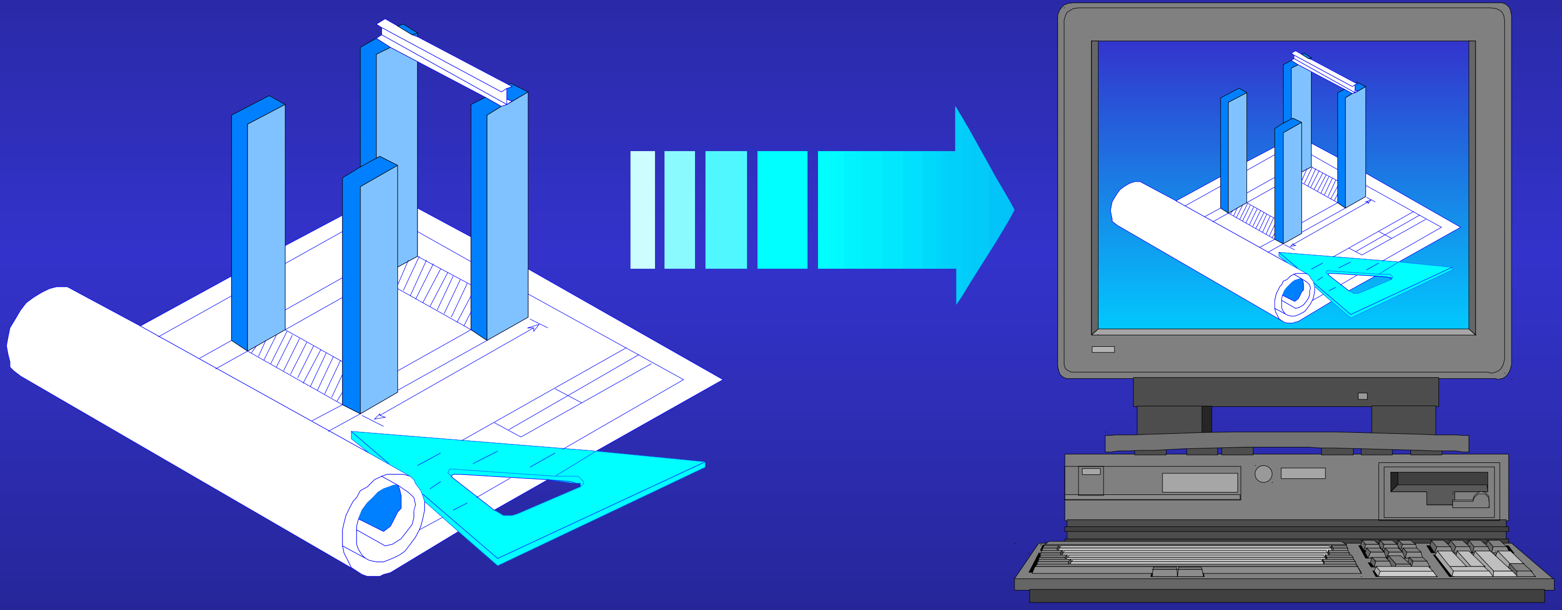
Pattern-Based Software Development

A Hands-on Introduction

Curbralan Ltd

www.curbralan.com

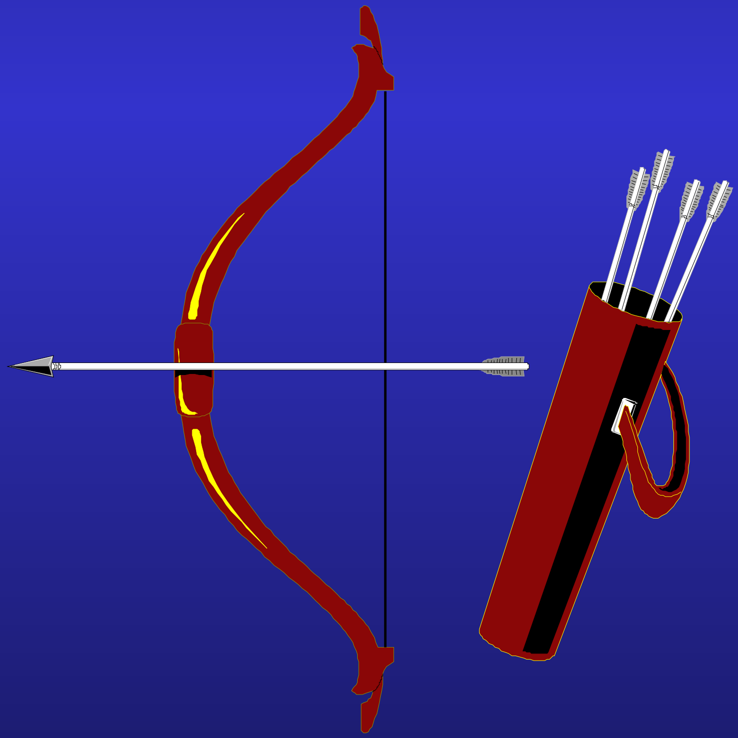
Course Introduction



Pattern-Based Software Development

Course Introduction

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

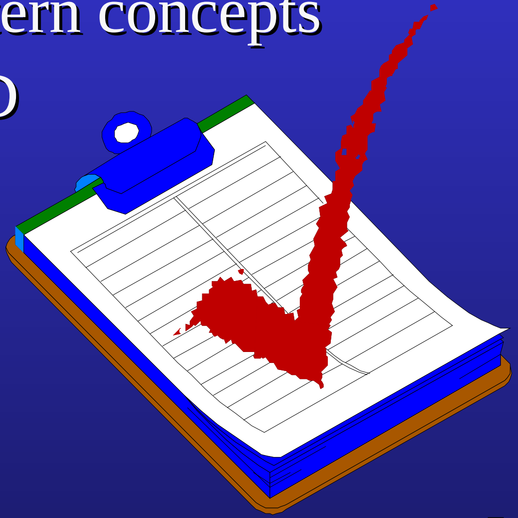


Objectives

- Understand what does and does not go to make up a pattern
- Understand the role of patterns in software architecture
- Learn some common patterns for object-oriented design
- Appreciate patterns from the strategic level to examples in code

Prerequisites

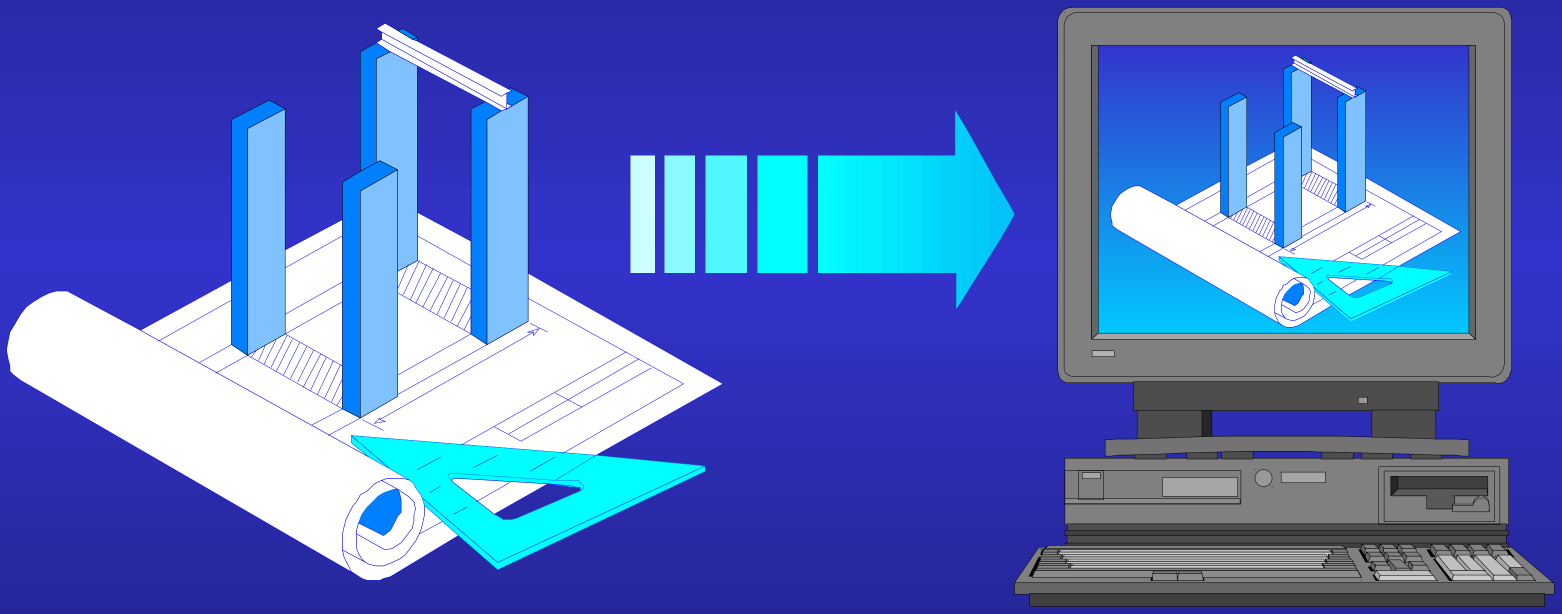
- Essential...
  - Hands-on experience of object-oriented development
  - Some familiarity with UML
  - Knowledge of Java
- Useful...
  - Previous exposure to patterns and pattern concepts
  - Previous experience of JUnit and TDD



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

Defining Architecture

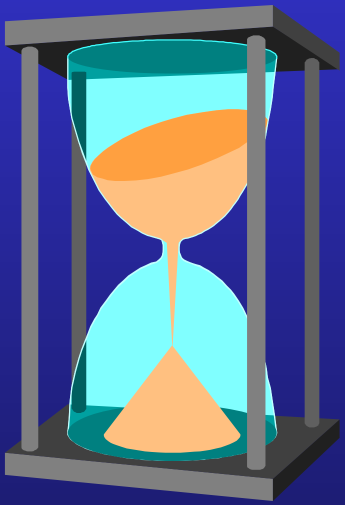
- An architecture defines (or is defined by) the critical decisions in a system's structure
  - Relates to its form, function and rationale
- Includes all levels of detail
  - Not just the gross structure, i.e. architecture is not just marketecture and PowerPoint diagrams
  - Architectural decisions taken at broadest level influence the detail of how code is written
- Ideally, an architecture should be formed so that minor decisions do not accidentally become critical

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

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



Accepting Change

- Predicting the future is non-trivial
  - Predicting the past is not that much easier
- Initial requirements and design are not likely to be stable over a system's lifetime
  - Changes in understanding and needs mean yesterday's design may no longer be suitable
- Design should be promoted as continuous and requirements as a dialogue
  - Design must be involve cleaning up, as well as introducing new code for new features
  - Implies a change to many project management models

Overdesign and Sufficiency

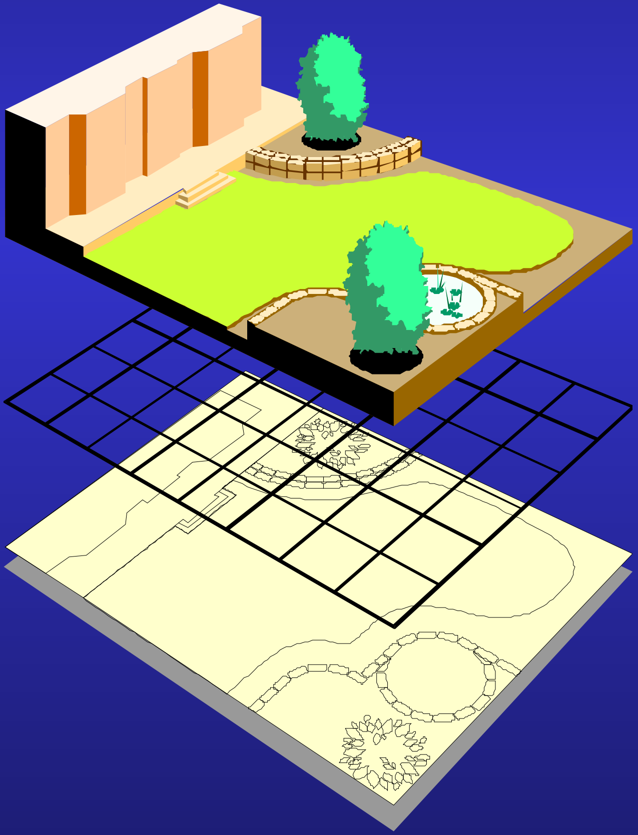
- It is tempting to combat the tides of change by trying to accommodate all possibilities
  - Design in every conceivable option, every variation, and accommodate every whim
- The result is typically complicated and resistant to change rather than resilient
- Avoid expending excessive design effort in making the capacity for change explicit
  - Work to reduce friction in the face of change — this generally involves choosing less over more

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

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



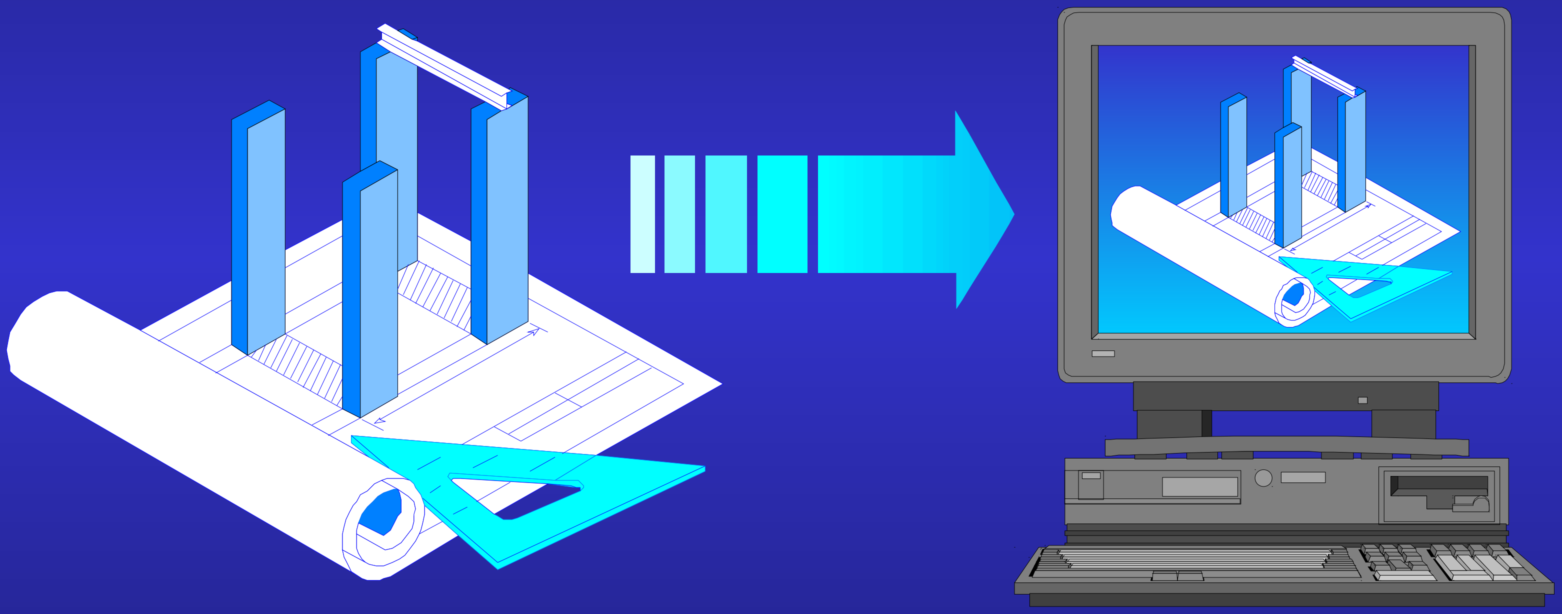
Patterns for Architecture

- Architectural style and key decisions in an architecture can be expressed through patterns
  - A pattern represents a solution with supporting rationale to a problem that has been identified
- Patterns are drawn from experience
  - I.e. they recur, hence the name "pattern"
  - Documented patterns name and detail the structure of the problem and the corresponding solution
- Patterns can form a shared vocabulary for communication and decision making

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

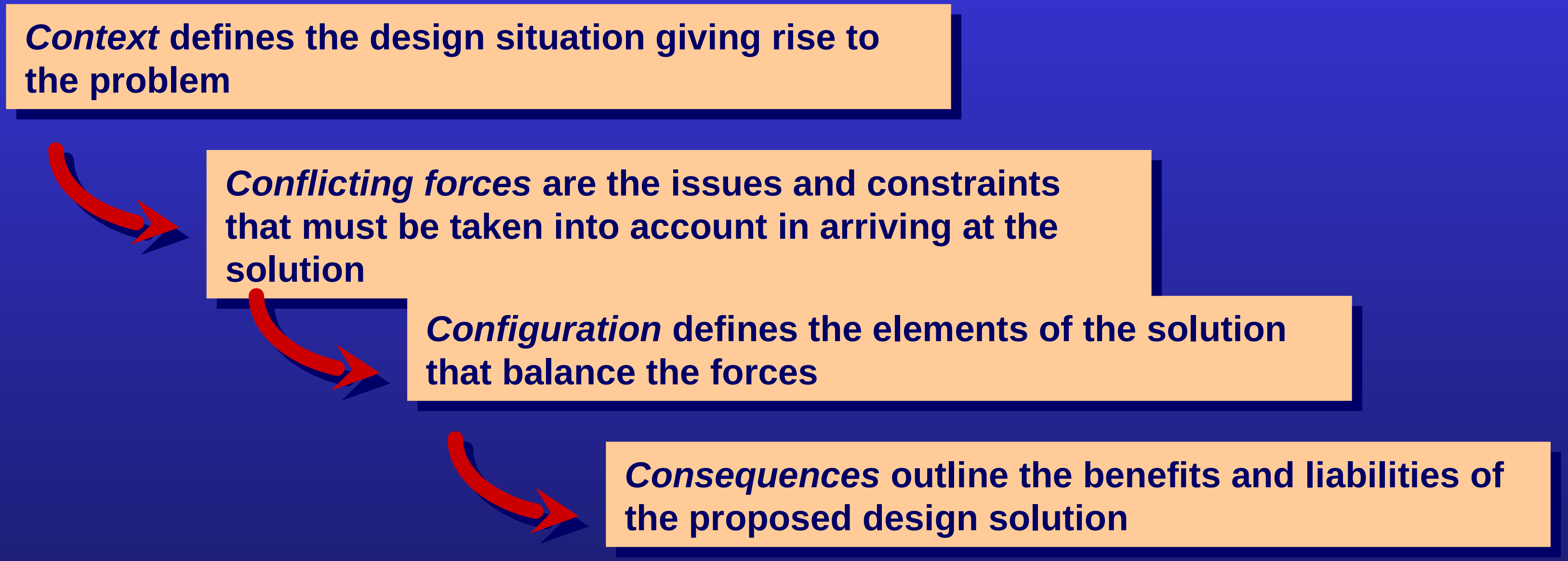
- Objectives
  - Introduce vocabulary and ideas behind software patterns
- Contents
  - Patterns in software architecture
  - Pattern anatomy
  - The role of patterns
  - Essential pattern form elements
  - Common pattern resources

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

Pattern Anatomy

- Patterns initiate a dialogue with a design context
  - A pattern is typically named after its solution structure
  - A pattern needs to document its applicability and its consequences



Context defines the design situation giving rise to the problem

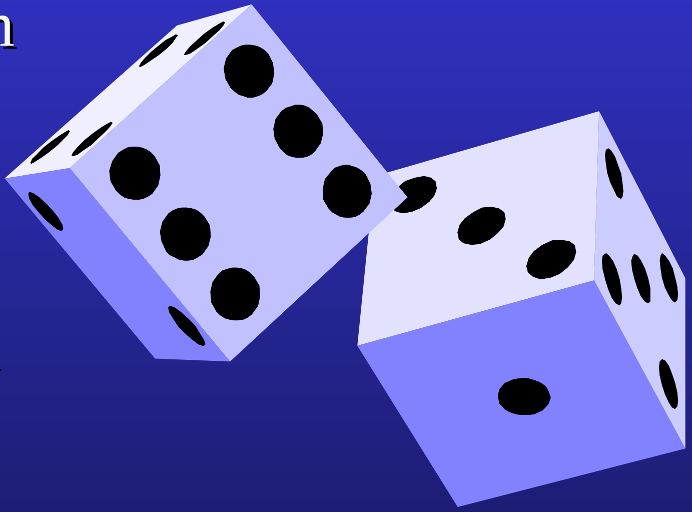
Conflicting forces are the issues and constraints that must be taken into account in arriving at the solution

Configuration defines the elements of the solution that balance the forces

Consequences outline the benefits and liabilities of the proposed design solution

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

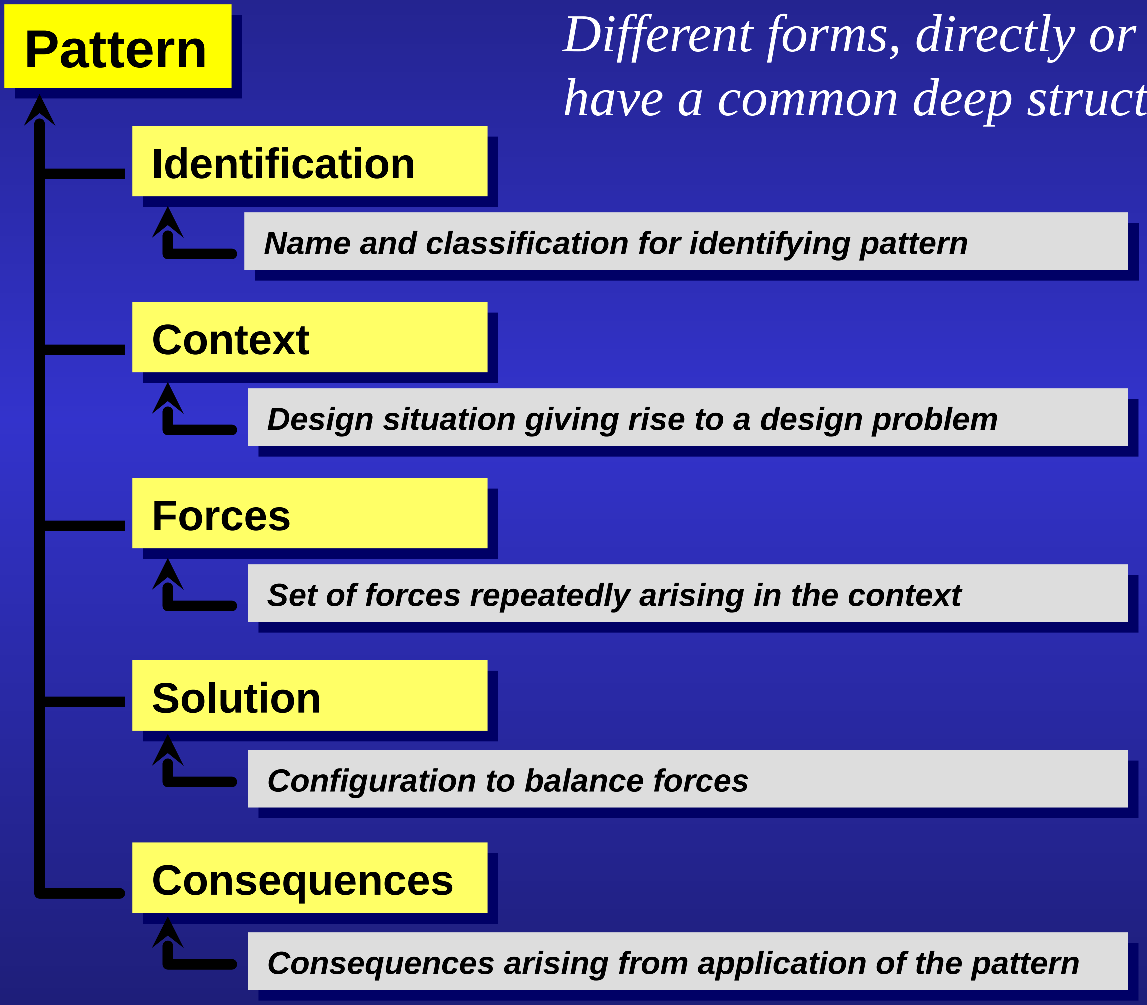


A Process and a Thing

- A pattern is a thing, but 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 to build it
  - Each one is a highly specific mini-methodology
- Patterns are not rigid and automatic, and must be considered and adapted every time
  - They inform a design rather than dictate it
- Its *form* describes how a pattern is documented

Essential Pattern Form Elements

Different forms, directly or otherwise, have a common deep structure



Pattern

Identification

Name and classification for identifying pattern

Context

Design situation giving rise to a design problem

Forces

Set of forces repeatedly arising in the context

Solution

Configuration to balance forces

Consequences

Consequences arising from application of the pattern

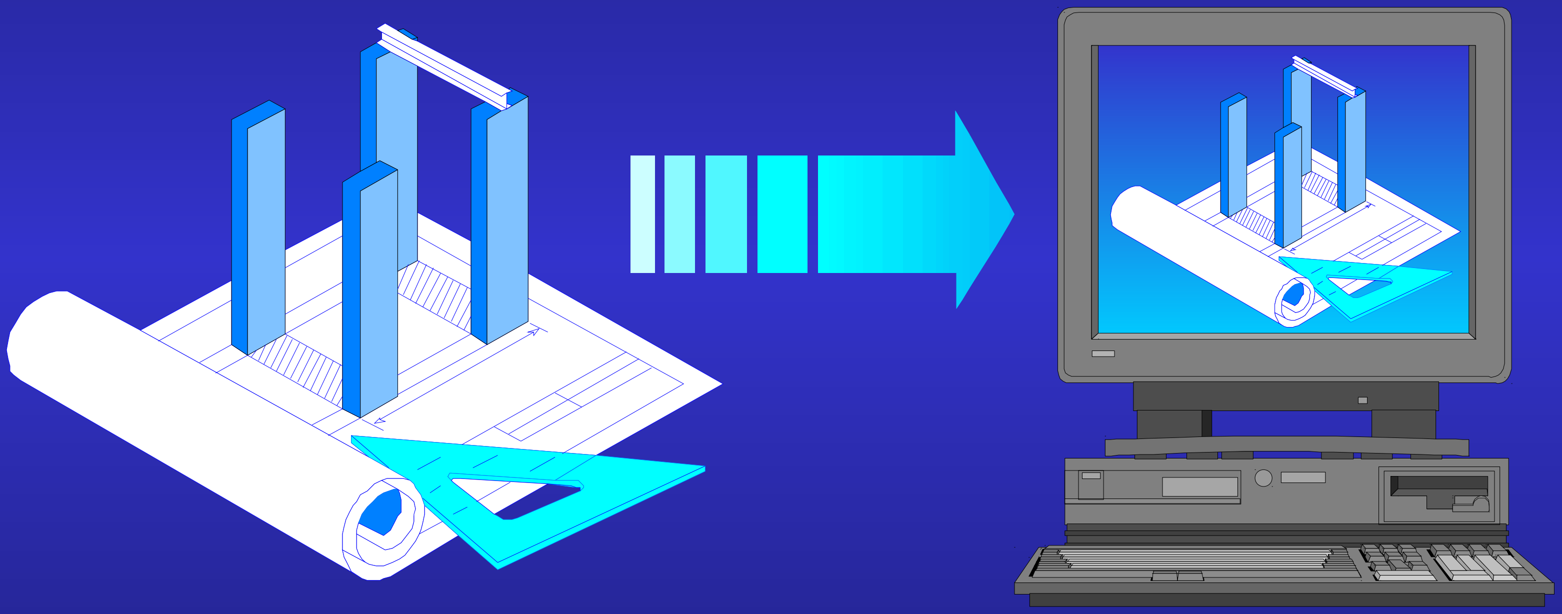
Common Pattern Resources

- The *Gang of Four* (GoF) patterns are best known
  - A catalogue of common OO design patterns
  - There are many books that re-present GoF
- The *Pattern-Oriented Software Architecture* (POSA) series focus on large-scale systems
  - Have evolved from catalogue to pattern language
- Patterns have gone much further than the basic GoF subset and format
  - Documented in numerous domains and many formats
  - A great deal of pattern literature is available online and published in books and articles

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?

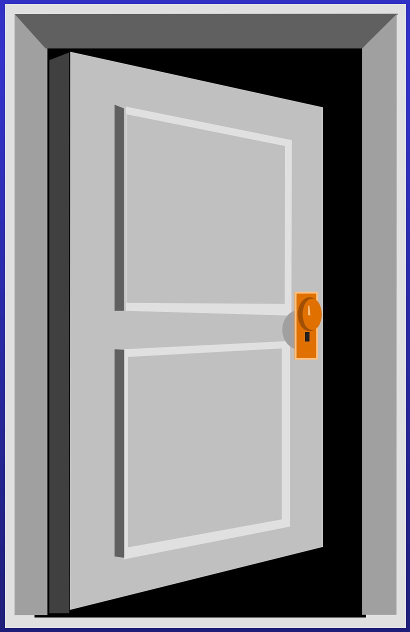
Introductory Pattern Examples



Pattern-Based Software Development

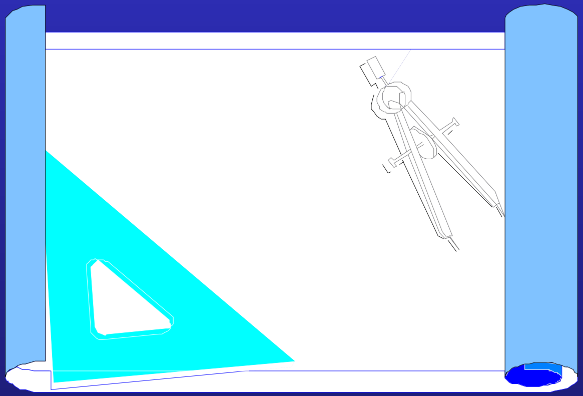
Introductory Pattern Examples

- 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 Composite Pattern

Context

- **A client manipulating target objects either individually or grouped together**

Forces

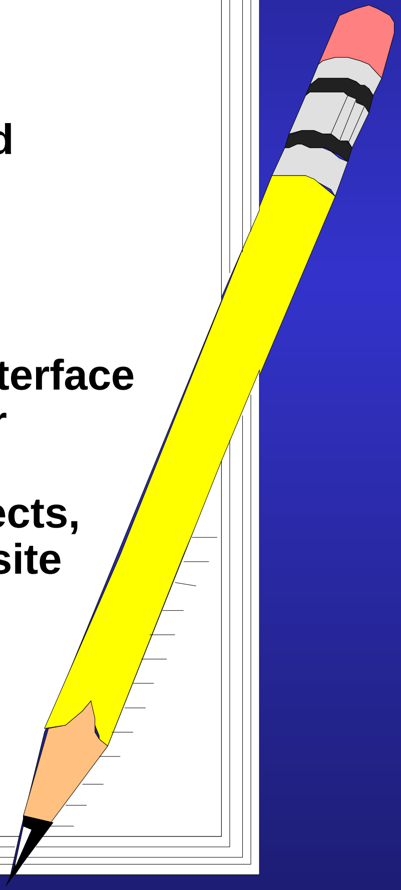
- **Transparent treatment of single and multiple objects would simplify client code**
- **Not all differences between single and multiple object manipulation can be ignored**

Solution

- **Introduce a composite object that implements the same interface as a leaf object through a common component interface or abstract class**
- **The composite object holds links to many component objects, each of which may be either leaf objects or further composite objects, and forwards its operations to each of them**

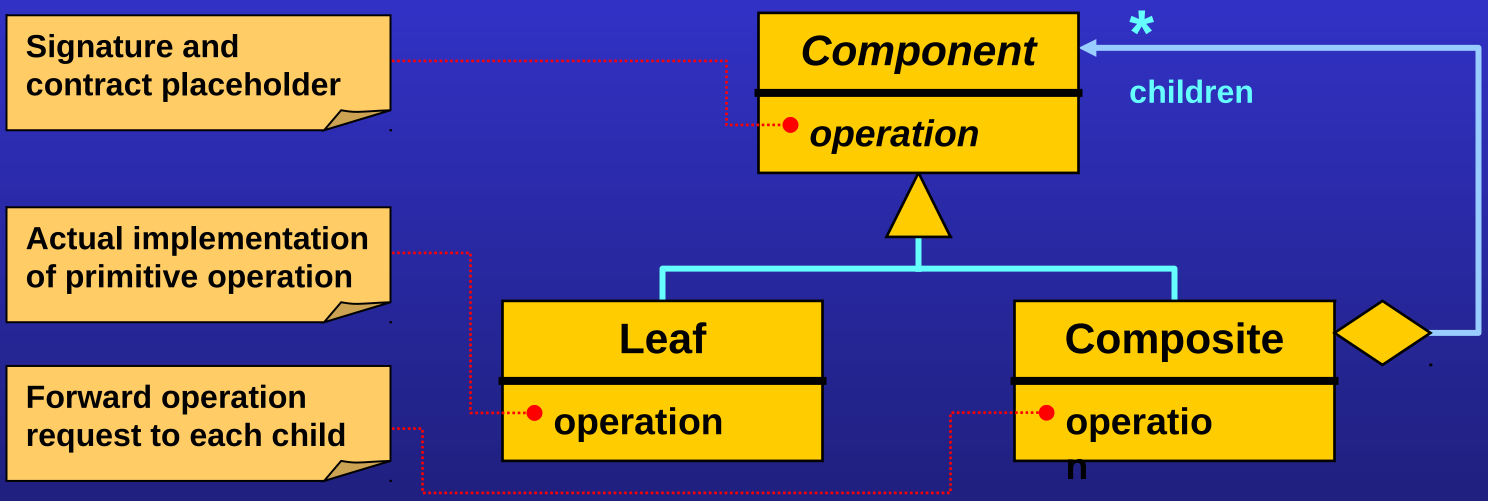
Consequences

- **Composite and leaf objects are treated uniformly**
- **The component interface may include operations specific to the composite that hold no meaning for the leaves**



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


Signature and contract placeholder
Actual implementation of primitive operation
Forward operation request to each child
Component
operation
*
children
Leaf
operation
Composite
operatio
n

Composite Code Sketch in Java

```java
public abstract class Component
{
    public abstract void operation(ArgumentType argument);
    ...
}
```

```java
public class Leaf extends Component
{
    public void operation(ArgumentType argument) ...
}
```

```java
public class Composite extends Component
{
    public void operation(ArgumentType argument)
    {
        for(Component child : children)
            child.operation(argument);
    }
    ...
    private Collection<Component> children;
}
```

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

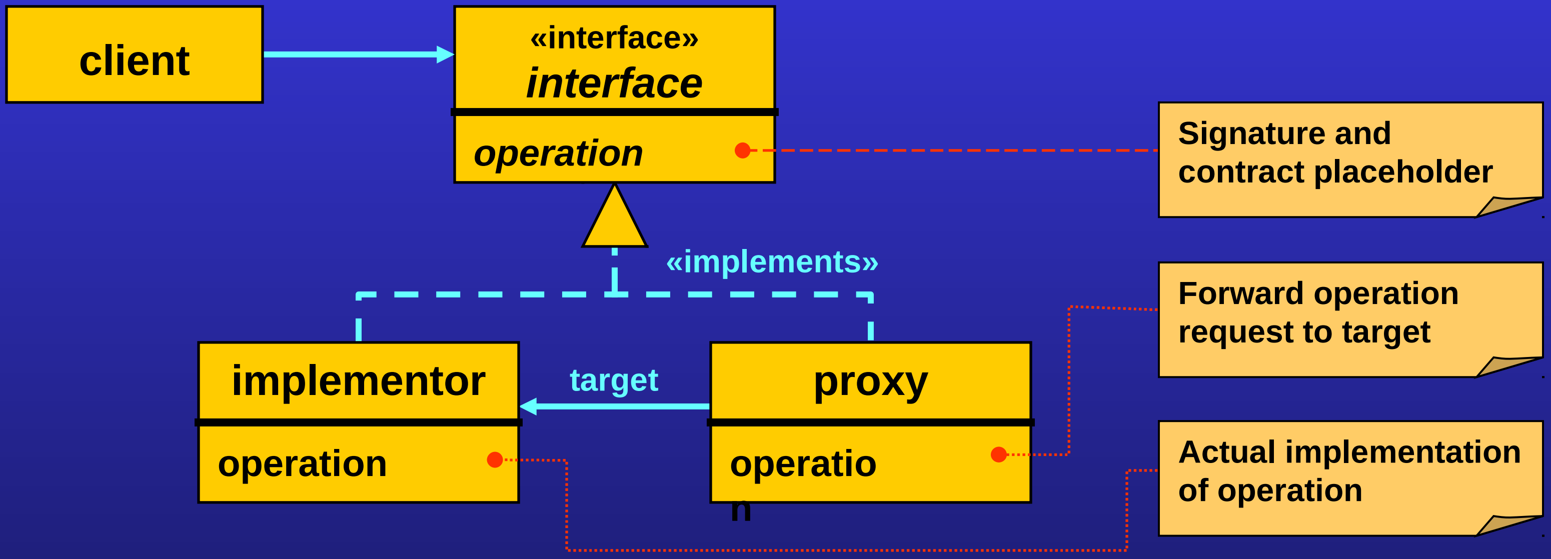
- Provide a proxy that stands in for the actual target object

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

Proxy Configuration Sketch

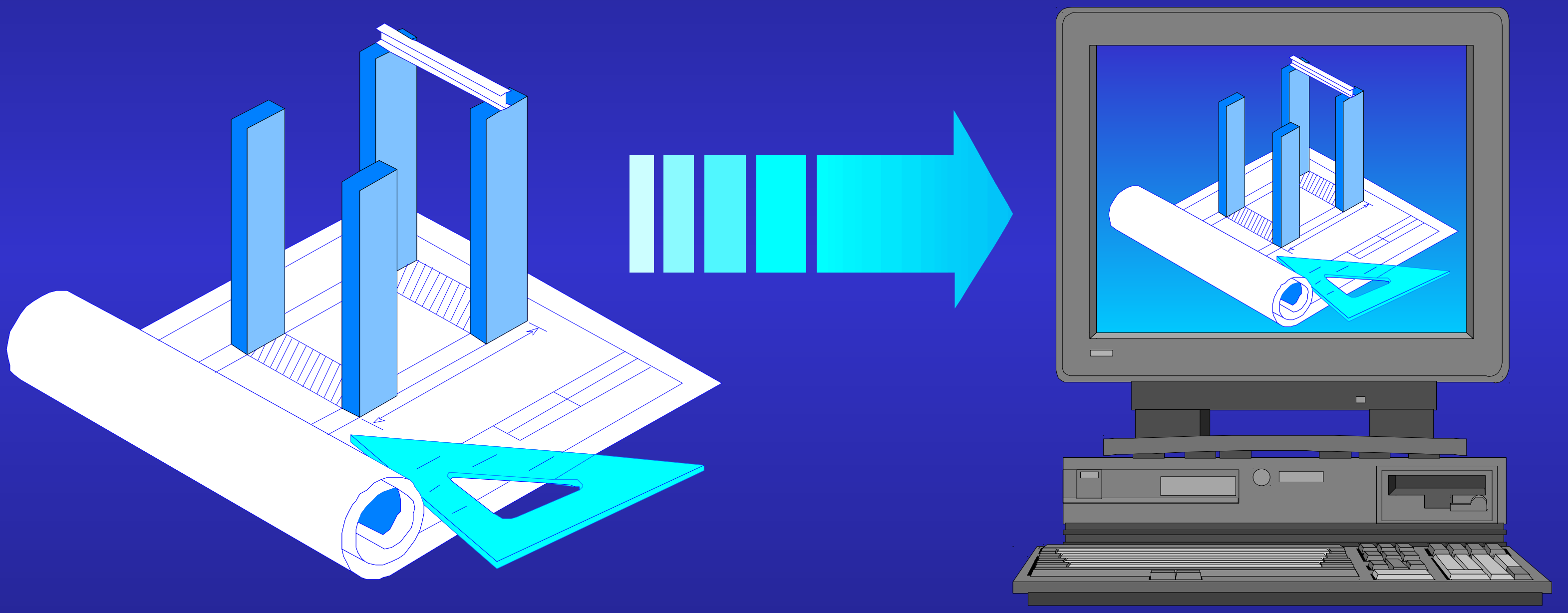
- Proxy is typically described in terms of an interface and a class hierarchy
  - However, proxies are not just an OO concept


client
«interface»
interface
operation
«implements»
implementor
operation
target
proxy
operatio
n
Signature and contract placeholder
Forward operation request to target
Actual implementation of operation

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

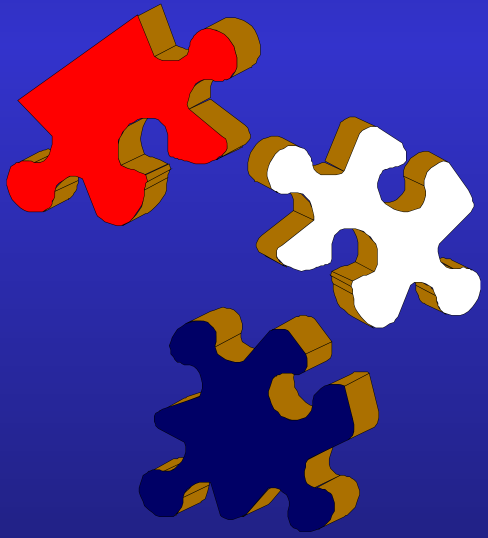
Combining Patterns



Pattern-Based Software Development

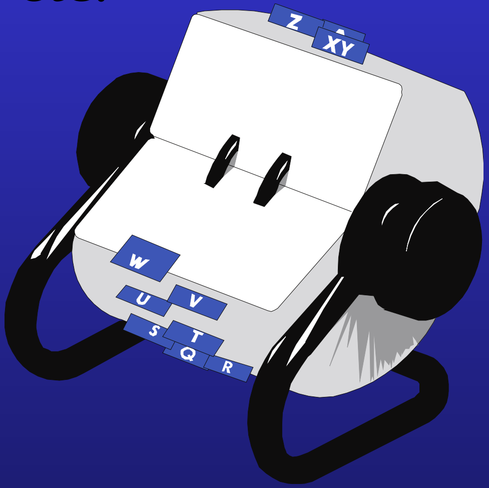
Combining Patterns

- 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

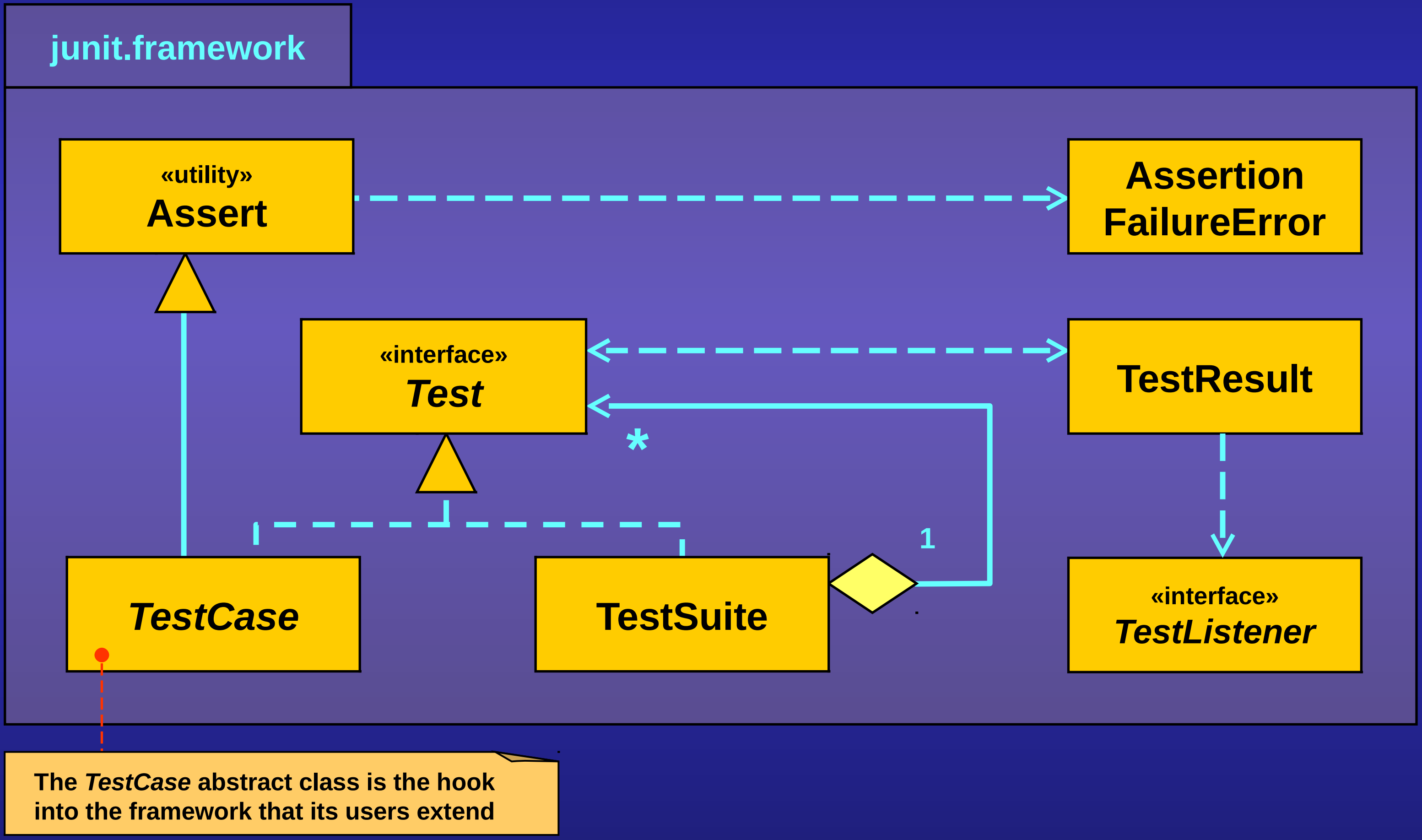
- A pattern catalogue holds individual patterns
  - Not necessarily interconnected
- Pattern catalogues may be organised by...
  - *Problem domain*: finance, telecoms, etc.
  - *Solution domain*: distribution, concurrency, N-tier architecture, RDBMS, Java, C++, C#, etc.
- Catalogues and other collections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



Pattern Communities

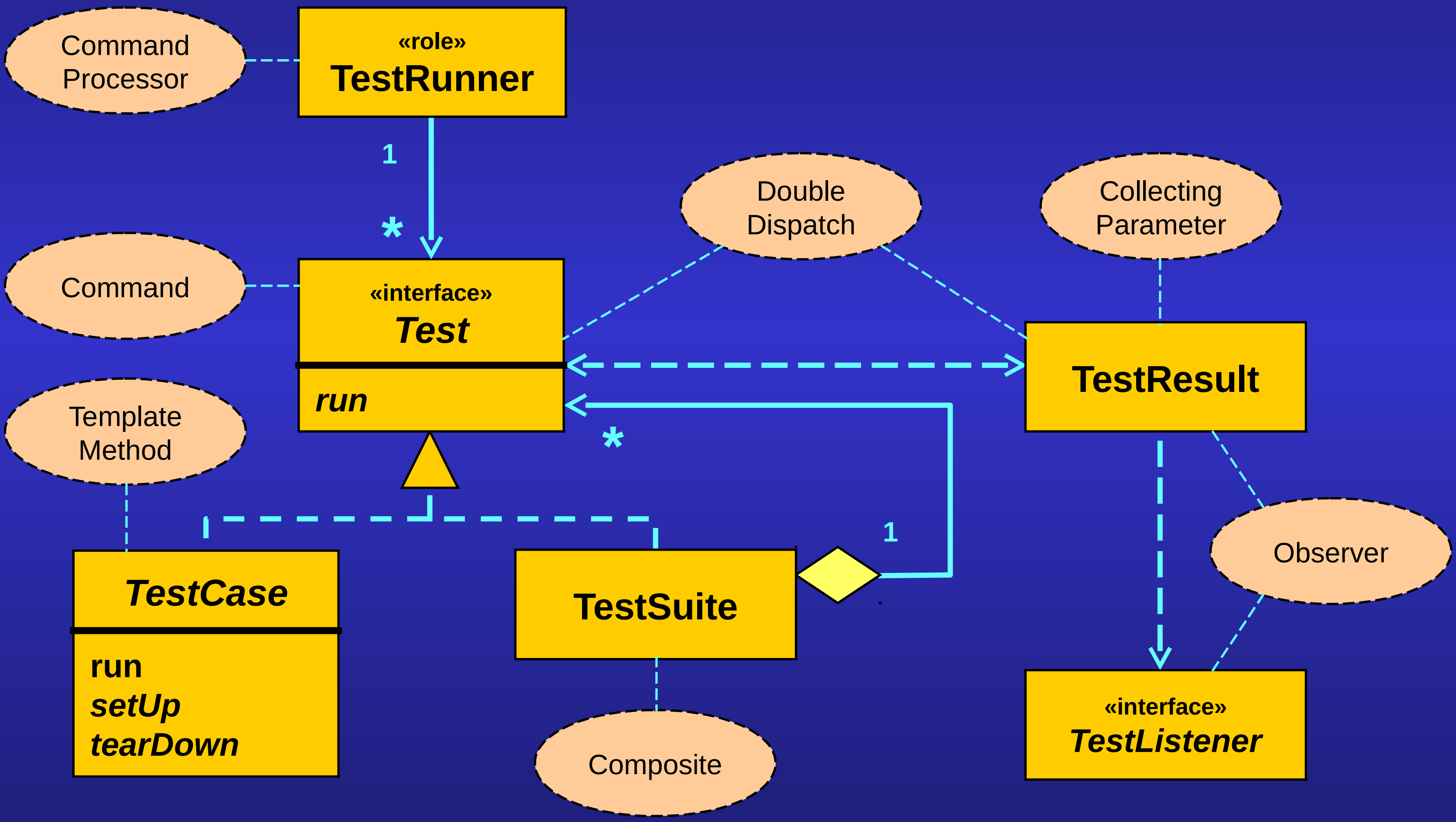
- Individual patterns can be used in isolation with some degree of success
  - Represent foci for discussion or point solutions
- However, patterns are in truth gregarious
  - They're rather fond of the company of patterns
  - To make practical sense as a design idea, patterns enlist other patterns for 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

Example of Core Classes in JUnit



The *TestCase* abstract class is the hook into the framework that its users extend

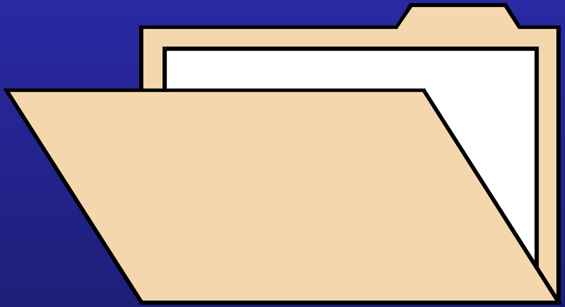
JUnit Pattern Usage Overview



© Curbralan Ltd

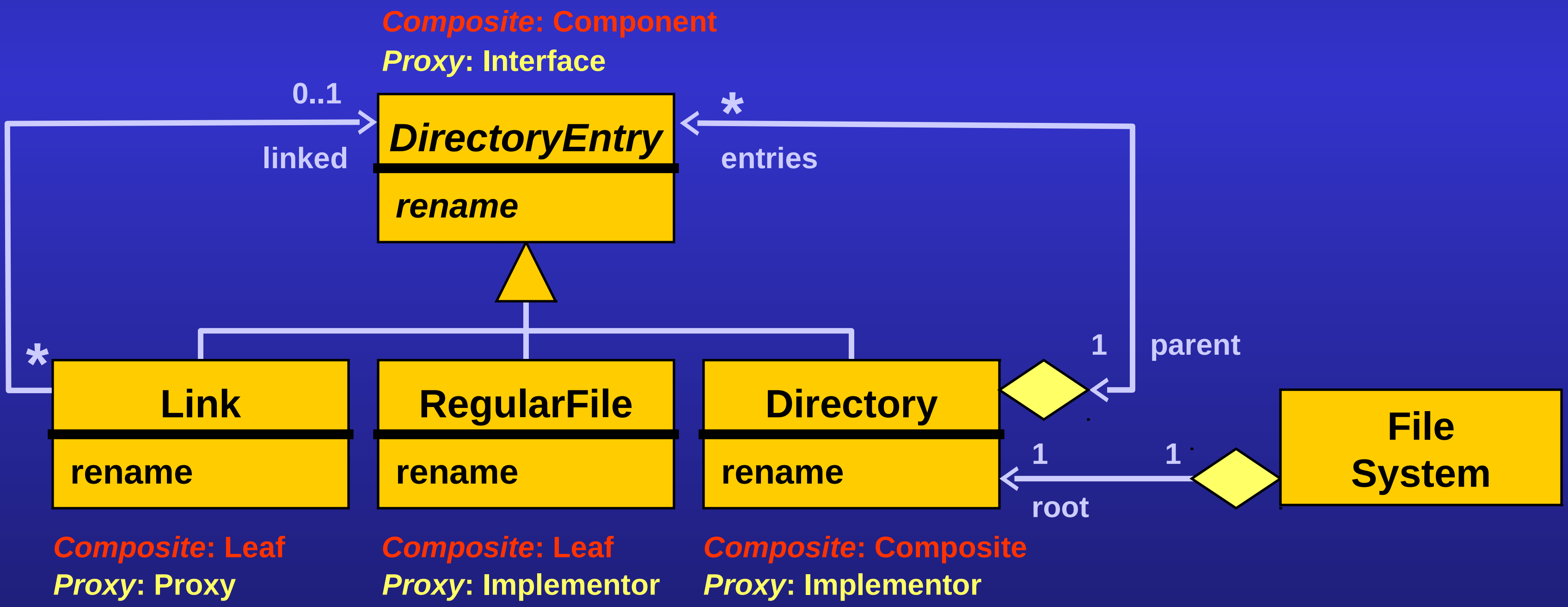
From Individual to Multiple Patterns

- 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



Patterns in Combination

- Some classes play more than one role in a design
  - Composite offers uniformity of elements and groups
  - Proxy transparently forwards one entry to another



© Curbralan Ltd

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 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);
    ...
}
```

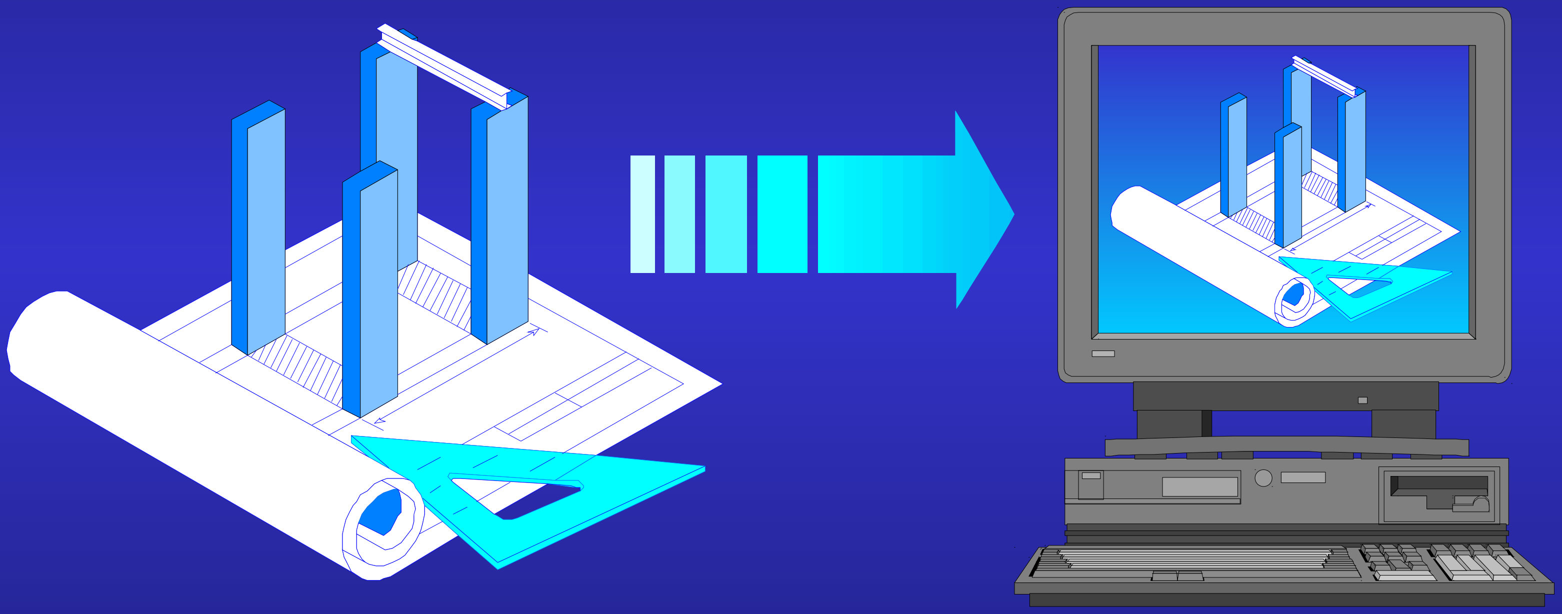
Pattern Families

- 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

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

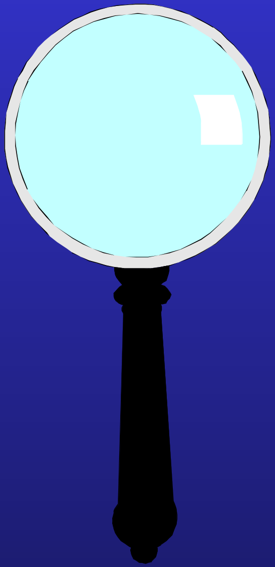
Pattern Context Dependency



Pattern-Based Software Development

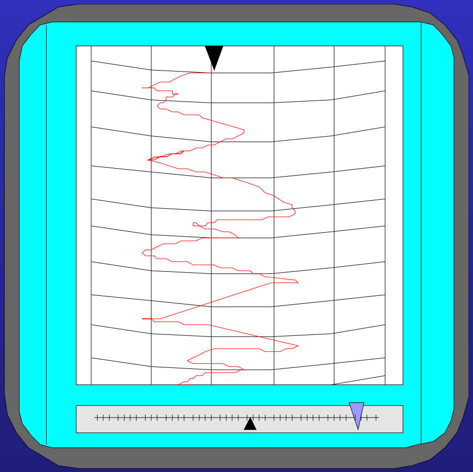
Pattern Context Dependency

- 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



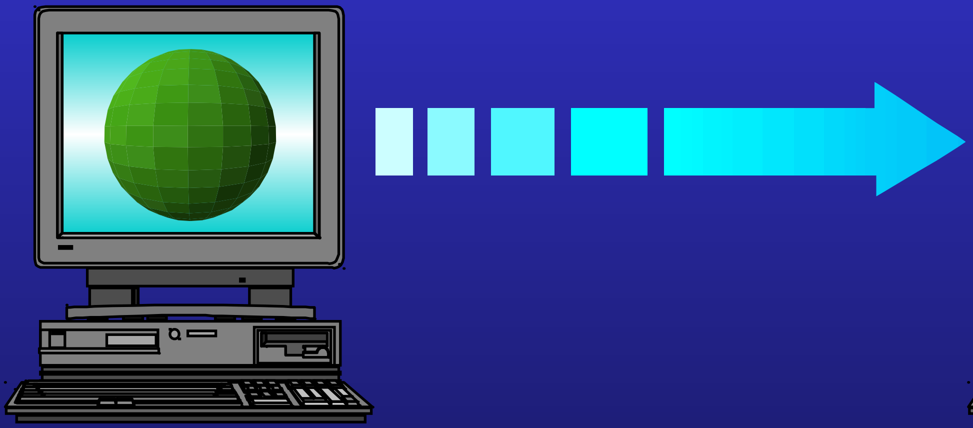
Context Sensitivity

- 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



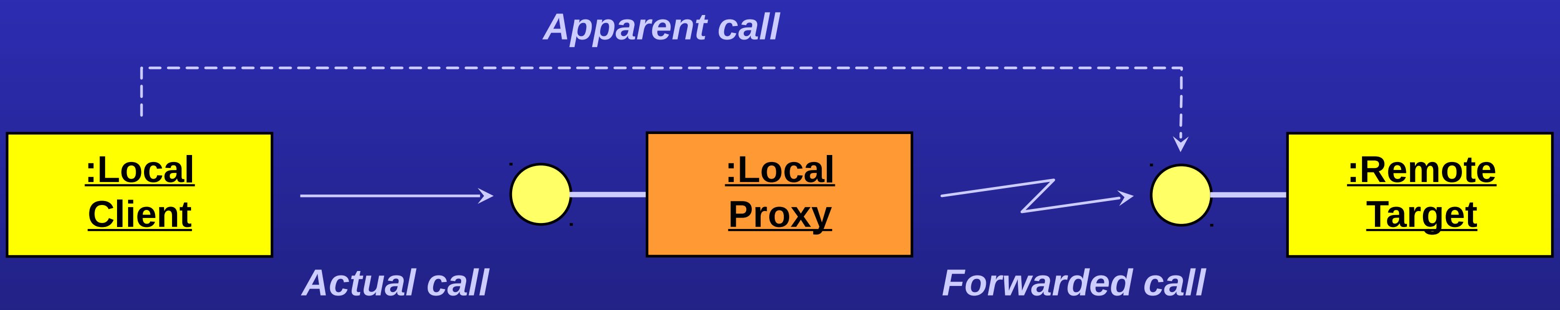
Remote Procedure Call Models

- 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



The Client Proxy Pattern

- Problem...
  - How can you communicate transparently with an object in a different address space?
- Solution...
  - Provide a placeholder to forward the request, and have it support the same interface as the remote target object

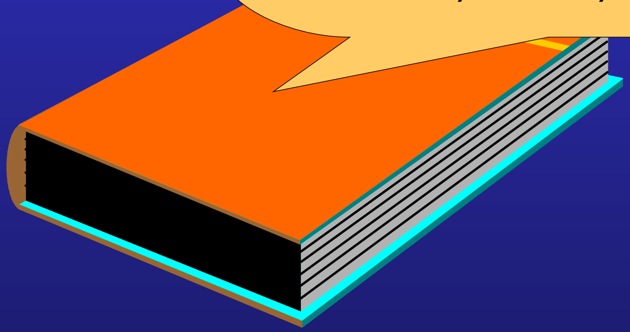


Strategic and Tactical Patterns

- Patterns can be applied at different levels in a system to different effect
  - *Strategic*: e.g. Proxy can be used as the basis of a distributed middleware architecture
  - *Tactical*: e.g. Proxy can be applied to wrap access to some local functionality conveniently
- Other classification schemes have been used to capture describe pattern granularity
  - However, these are not always consistent or successful as they often try to pigeonhole a pattern in one category

Idioms

- An idiom is often considered to be a pattern specific to a language, language model or technology
  - It has the language as part of its context
  - Addresses a particular design problem or solution native to a language

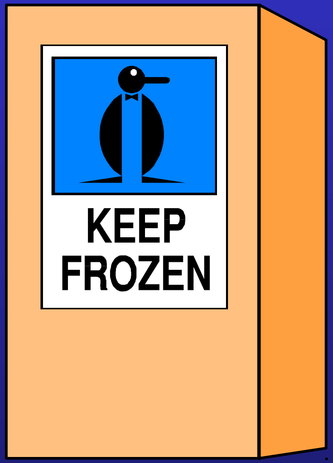


idiom

- **Linguistic usage that is grammatical and natural to native speakers**
- **The characteristic vocabulary or usage of a specific human group or subject**
- **The characteristic artistic style of an individual, school, etc.**

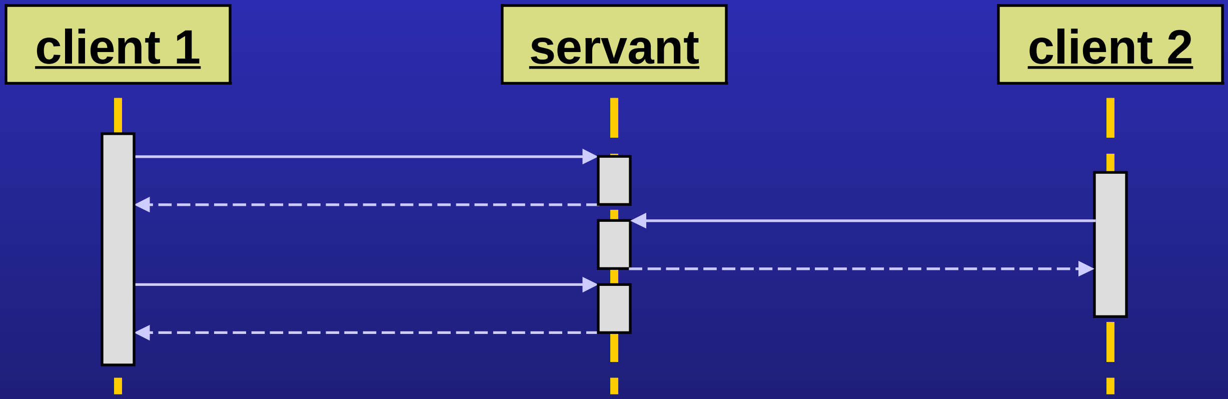
The Immutable Value Pattern

- 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



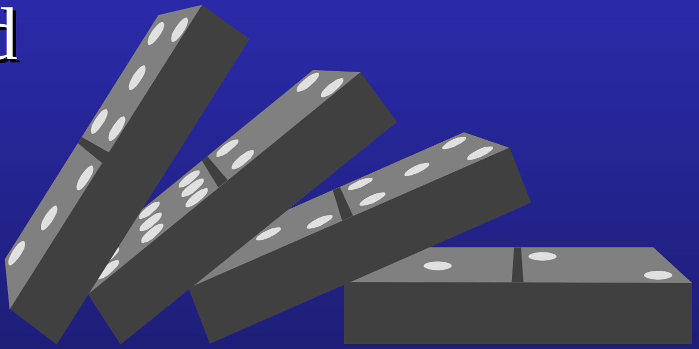
Concurrency and Method Calls

- 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



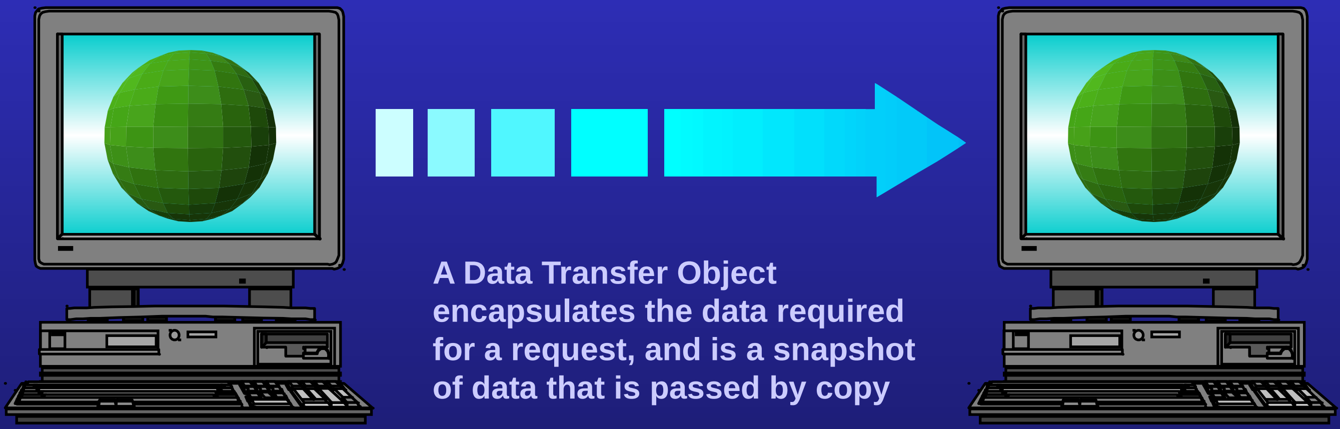
The Combined Method Pattern

- 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



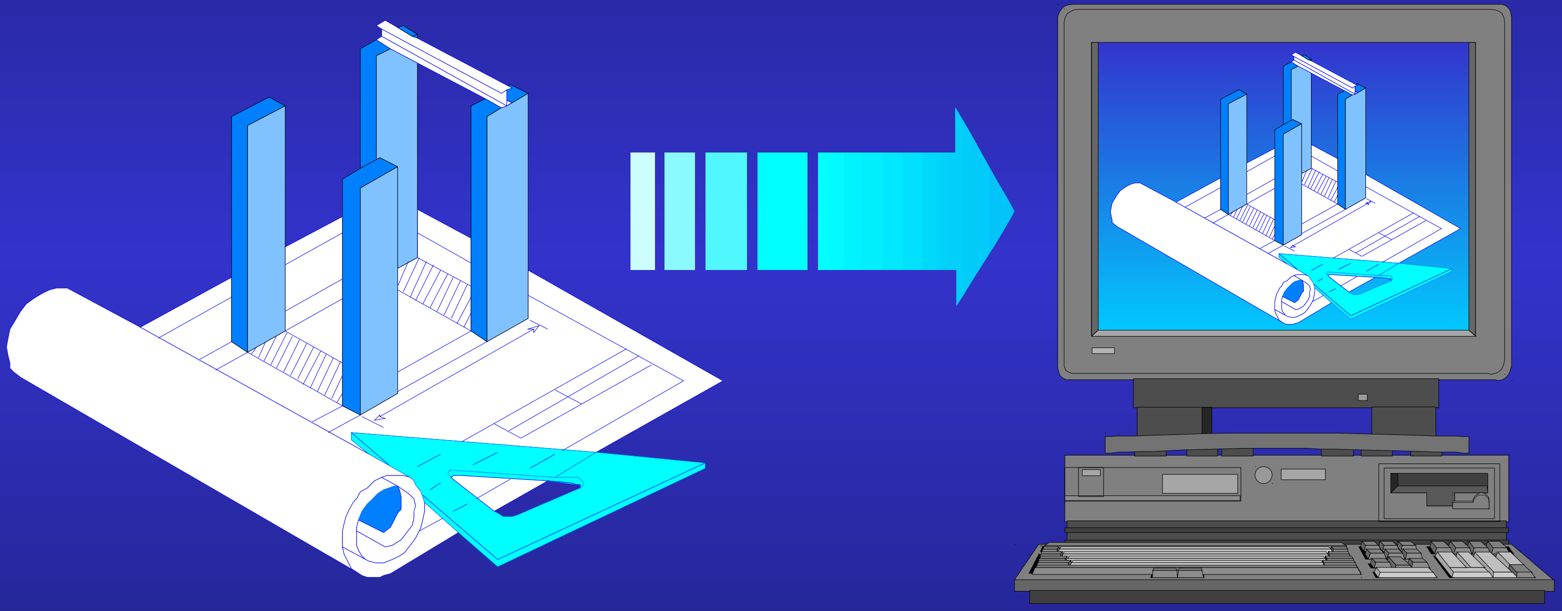
The Data Transfer Object Pattern

- 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



A Data Transfer Object encapsulates the data required for a request, and is a snapshot of data that is passed by copy

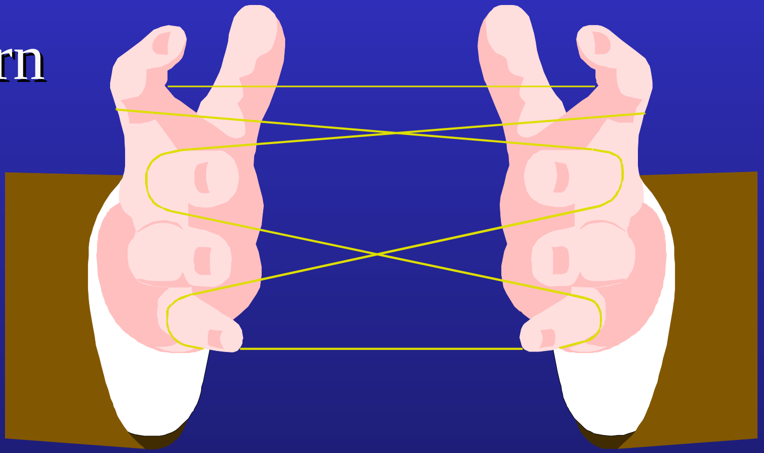
Patterns for Decoupling



Pattern-Based Software Development

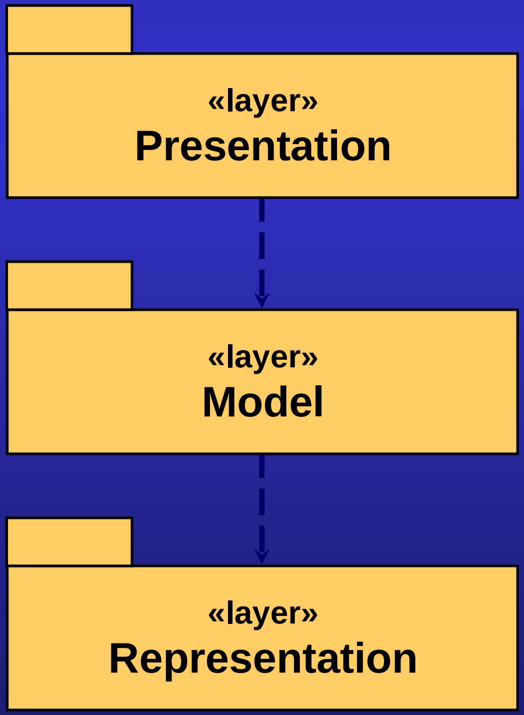
Patterns for Decoupling

- 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

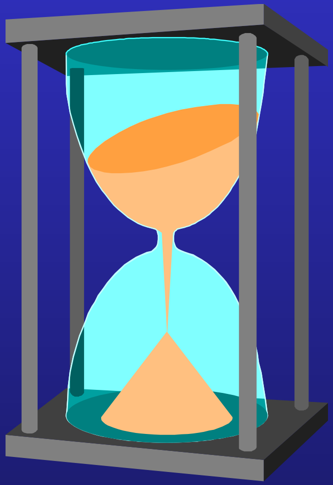


Porous Layers

- 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

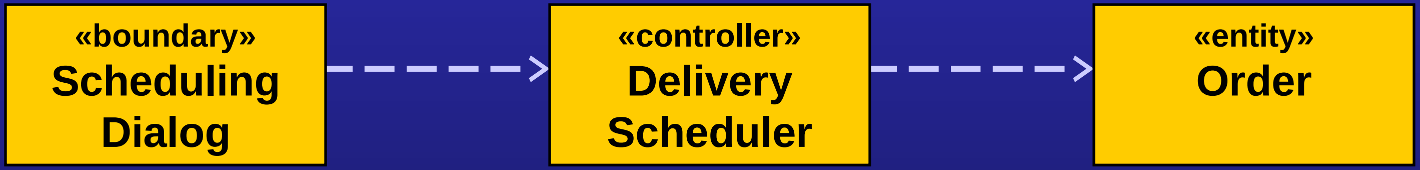
Time-Ordered Layering

- 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

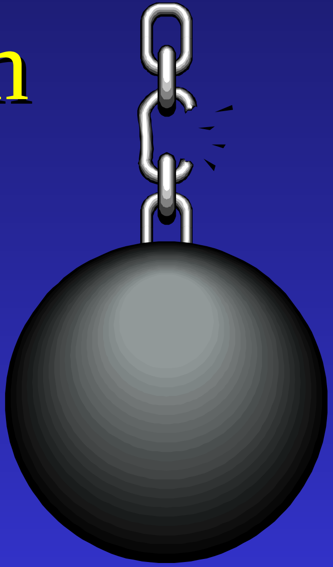


Boundary–Controller–Entity Layers

- 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



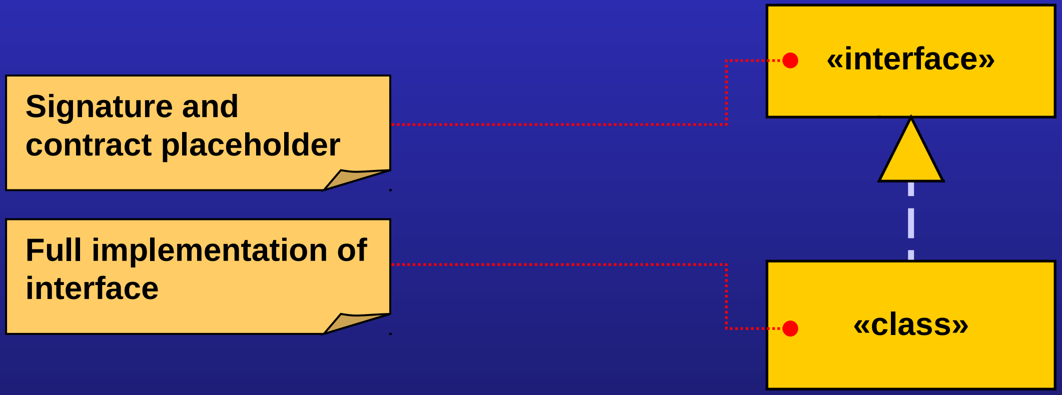
The Fragile Base Class Problem



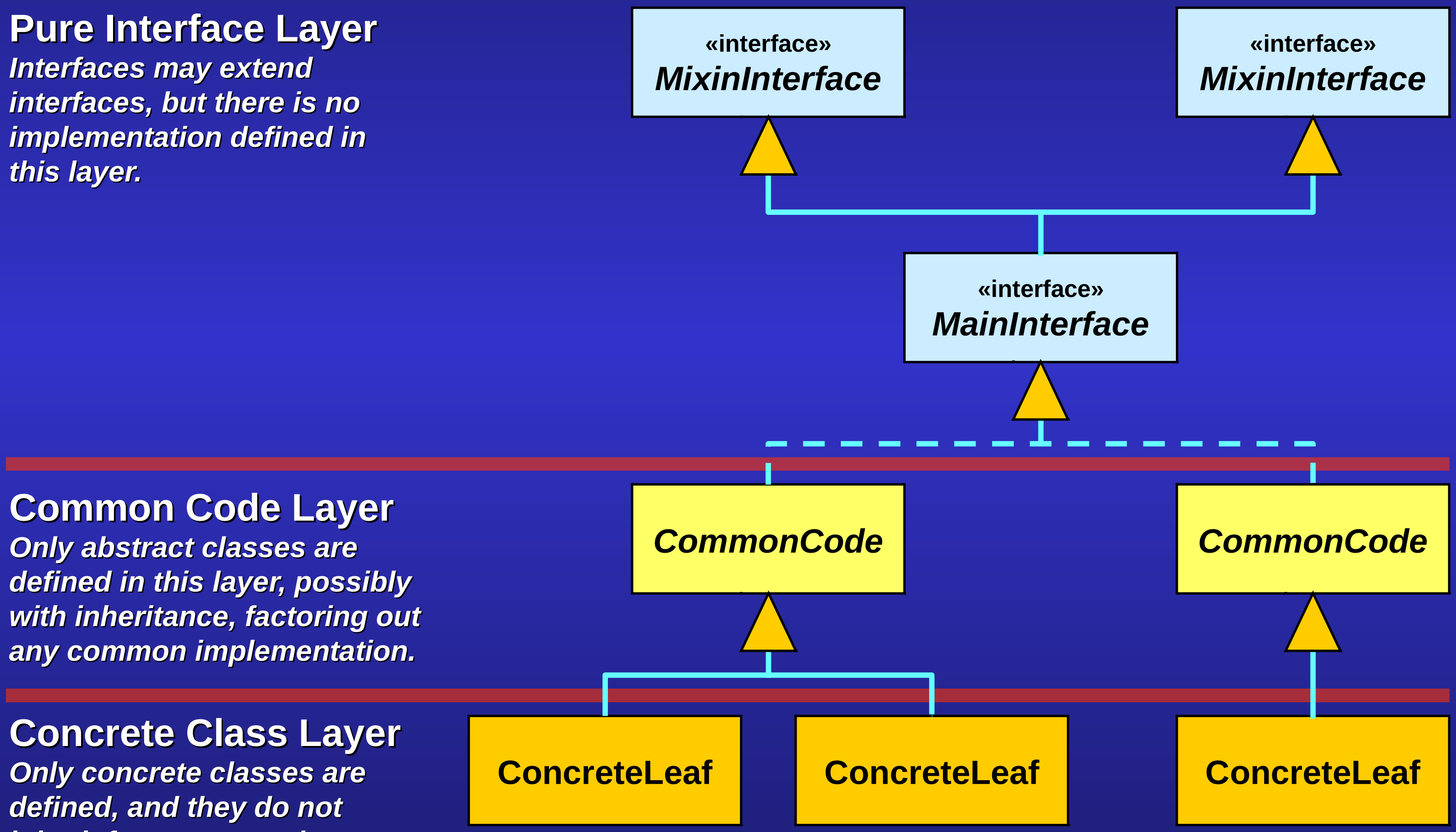
- A subclass has a strong dependency on its superclass
  - Not as fully encapsulated or decoupled as delegation-based relationships
  - Inheritance is essentially the strongest form of coupling in an OO system
- Superclass evolution is a problem...
  - Release-to-release interface compatibility
  - Private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

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


Signature and contract placeholder
Full implementation of interface
«interface»
«class»

Pure Roots, Concrete Leaves



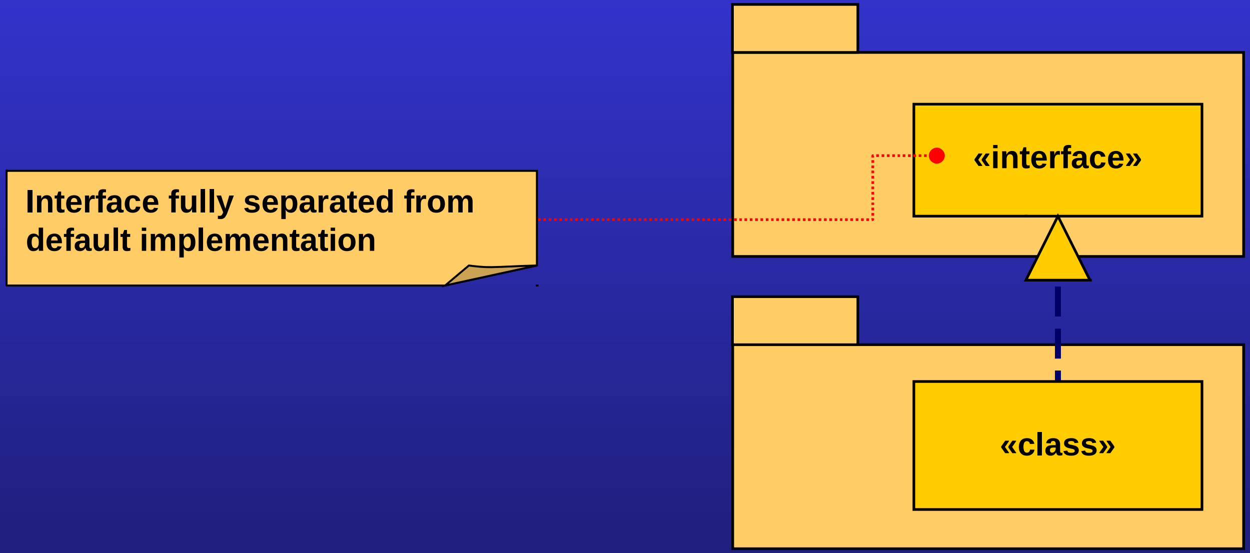
Pure Interface Layer
Interfaces may extend interfaces, but there is no implementation defined in this layer.

Common Code Layer
Only abstract classes are defined in this layer, possibly with inheritance, factoring out any common implementation.

Concrete Class Layer
Only concrete classes are defined, and they do not inherit from one another.

The Separated Interface Pattern

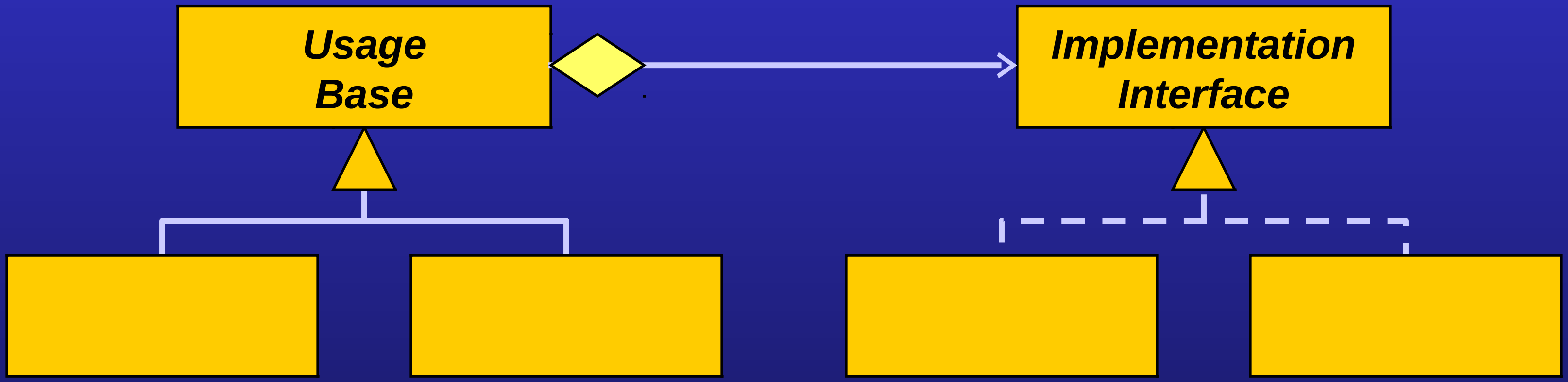
- The next step beyond making an interface explicit is to separate it from the implementing classes
  - This allows interface and implementation to be published, deployed and implemented separately



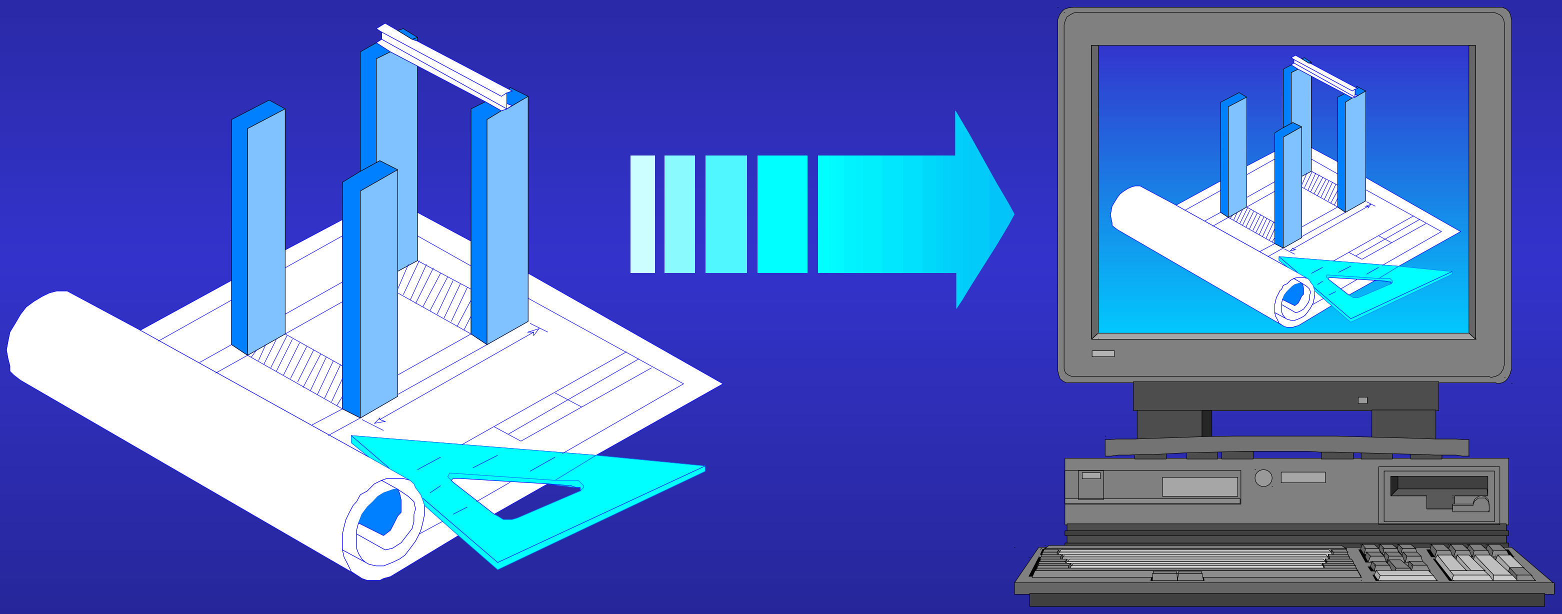
Interface fully separated from default implementation

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



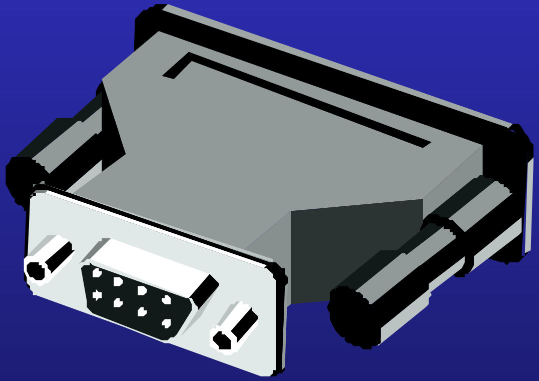
Patterns for Adaptation



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

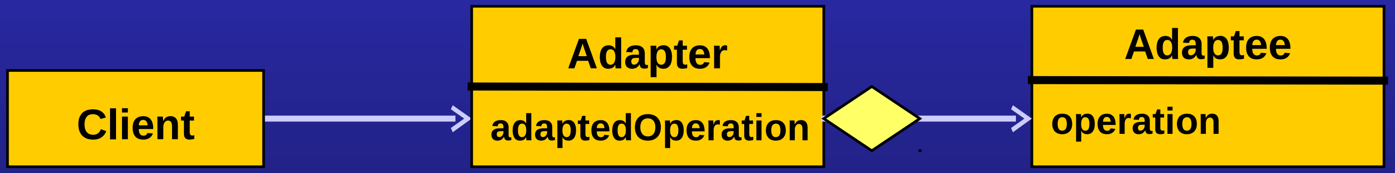


Adapter Patterns

- 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

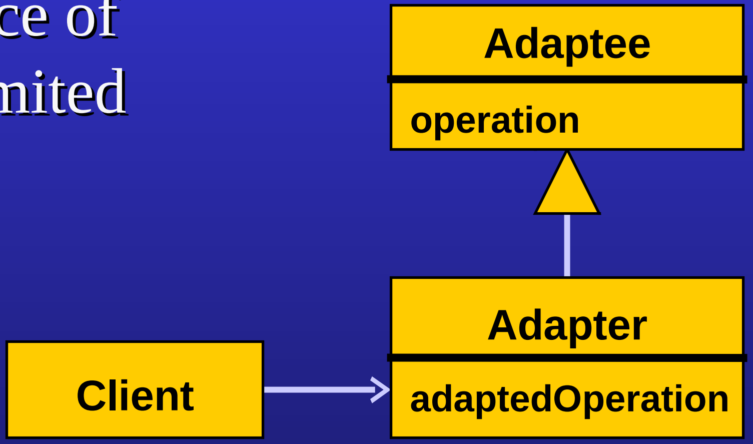
The Object Adapter Pattern

- 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



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 Adapter Patterns in Practice

```
public class Adaptee ...
{
    public ... method(...) ...
    ...
}
```

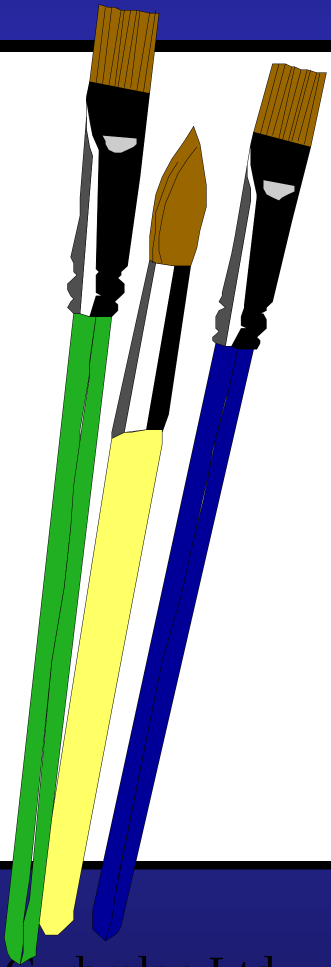
```
public class Adapter ...
{
    public ... adaptedMethod(...)
    {
        ... adaptee.method(...);
    }
    ...
    private Adaptee adaptee;
}
```

Object Adapter

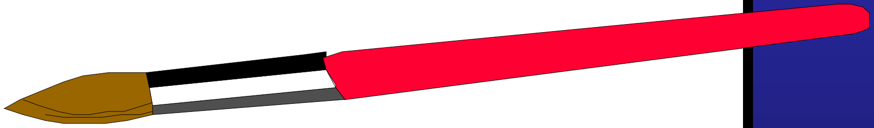
```
public class Adapter extends Adaptee ...
{
    ...
}
```

Class Adapter

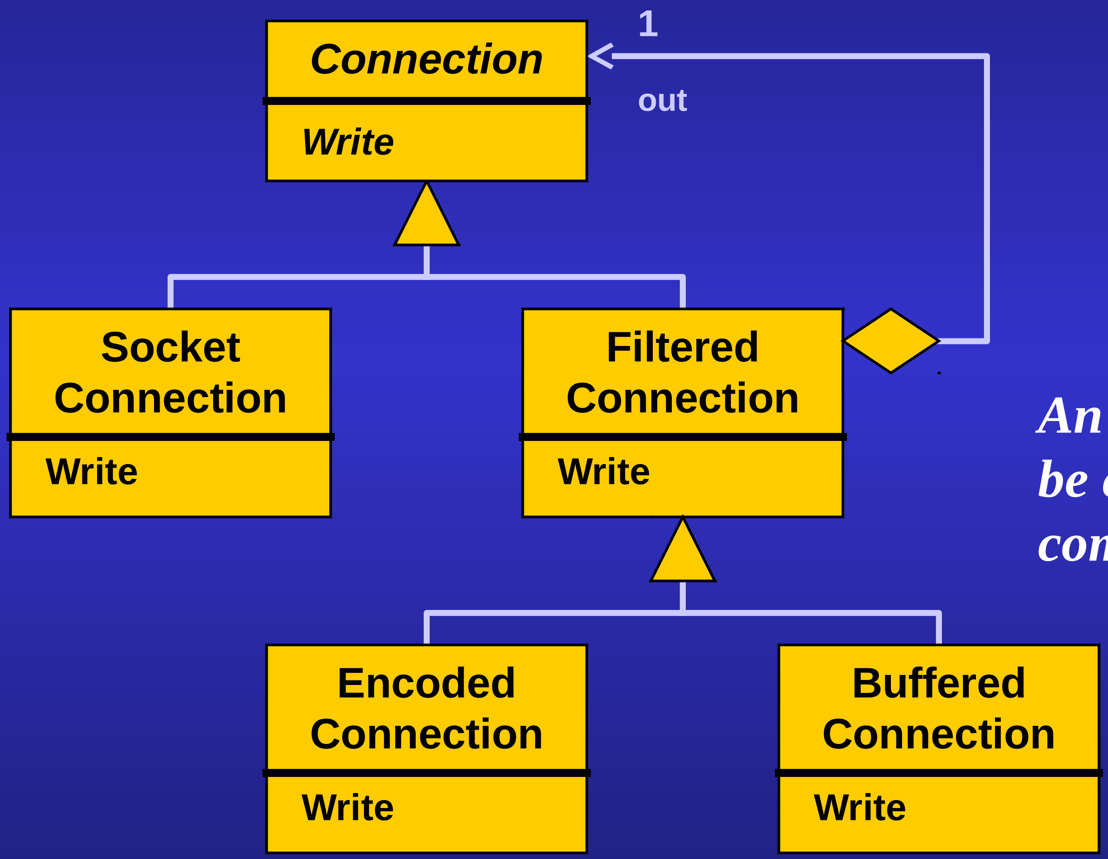
The Wrapped Adapter Pattern



```
public class Adapter ...
{
    public ... adaptedMethod(...)
    {
        ... adaptee.method(...);
    }
    ...
    private Adaptee adaptee =
        new Adaptee()
        {
            ... // override methods
        }
}
```



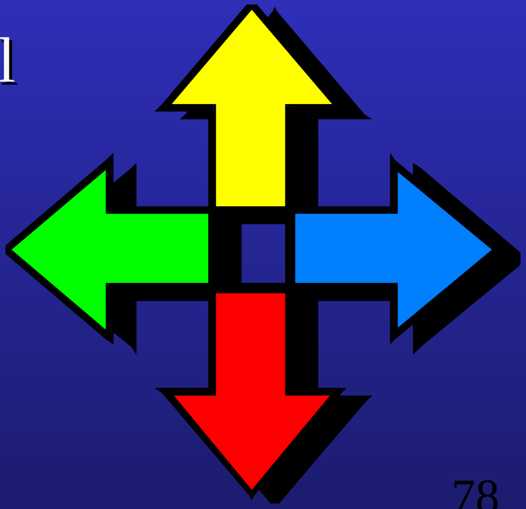
The Decorator Pattern



An object's capabilities may be extended through object composition and chaining.

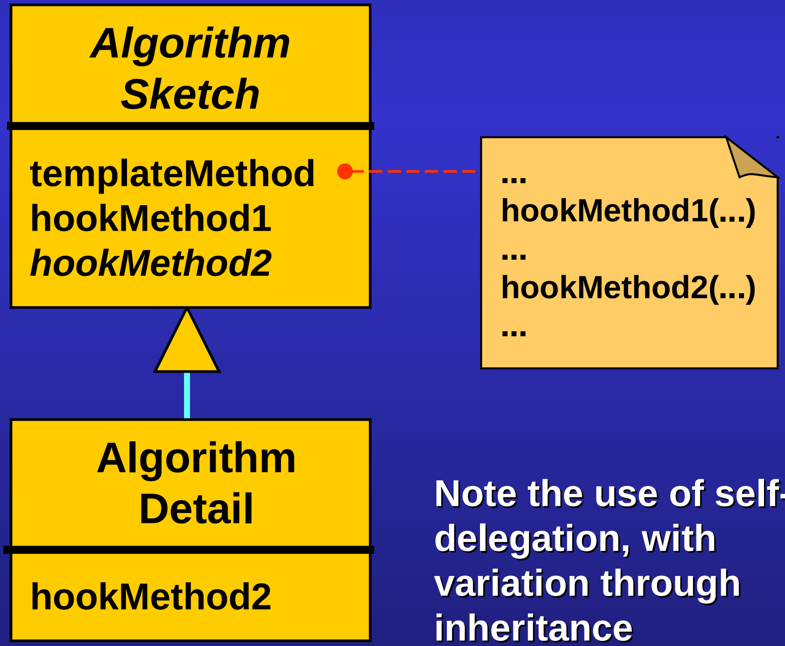
Proxy, Decorator or Object Adapter?

- 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



The Template Method Pattern

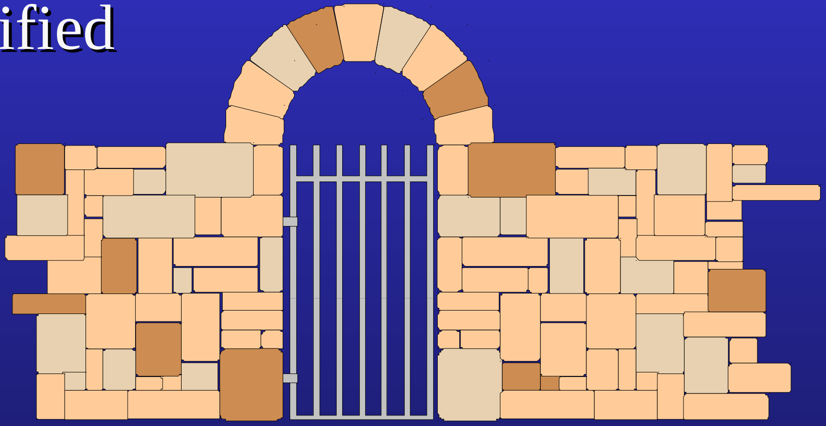
- Captures commonality of control-flow and separates it from variability of detail
  - Commonality is captured in a root class, and variability is deferred to methods overridden in subclasses



Note the use of self-delegation, with variation through inheritance

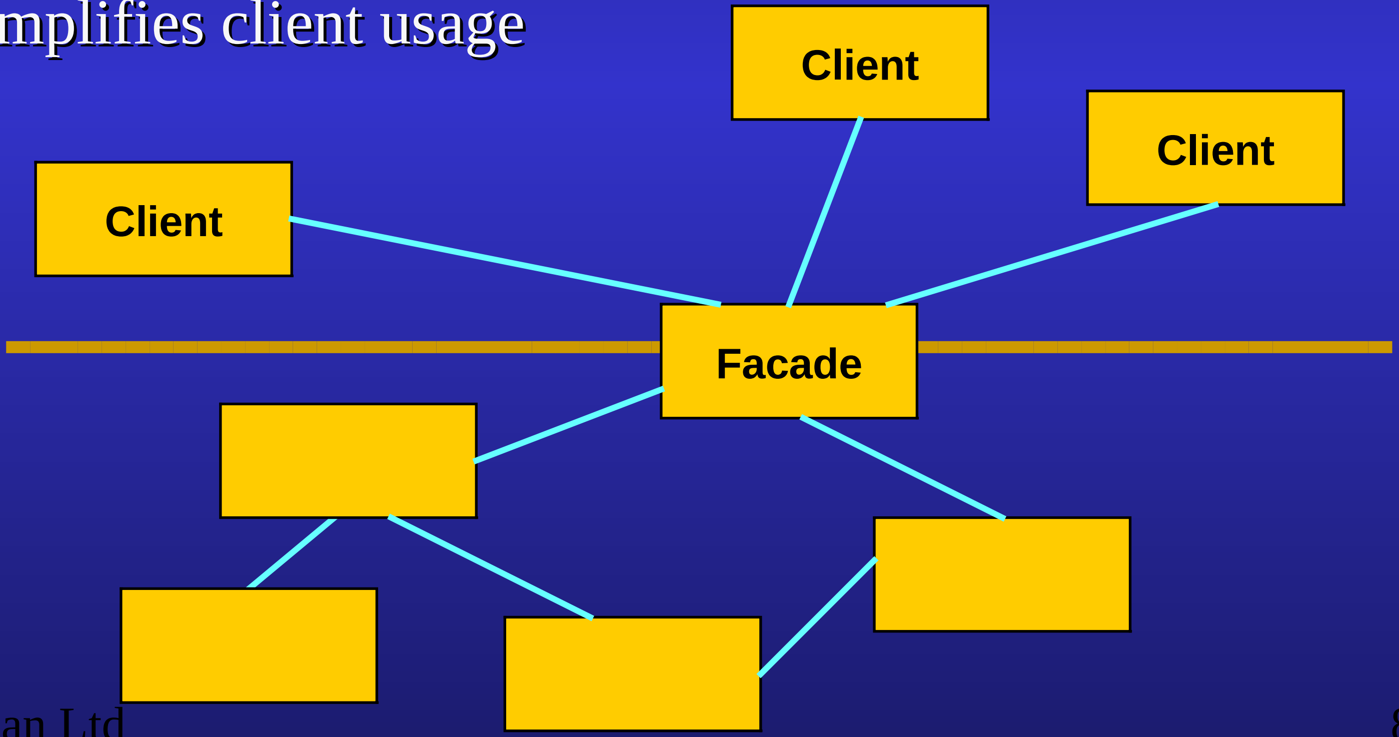
The Facade Pattern

- 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



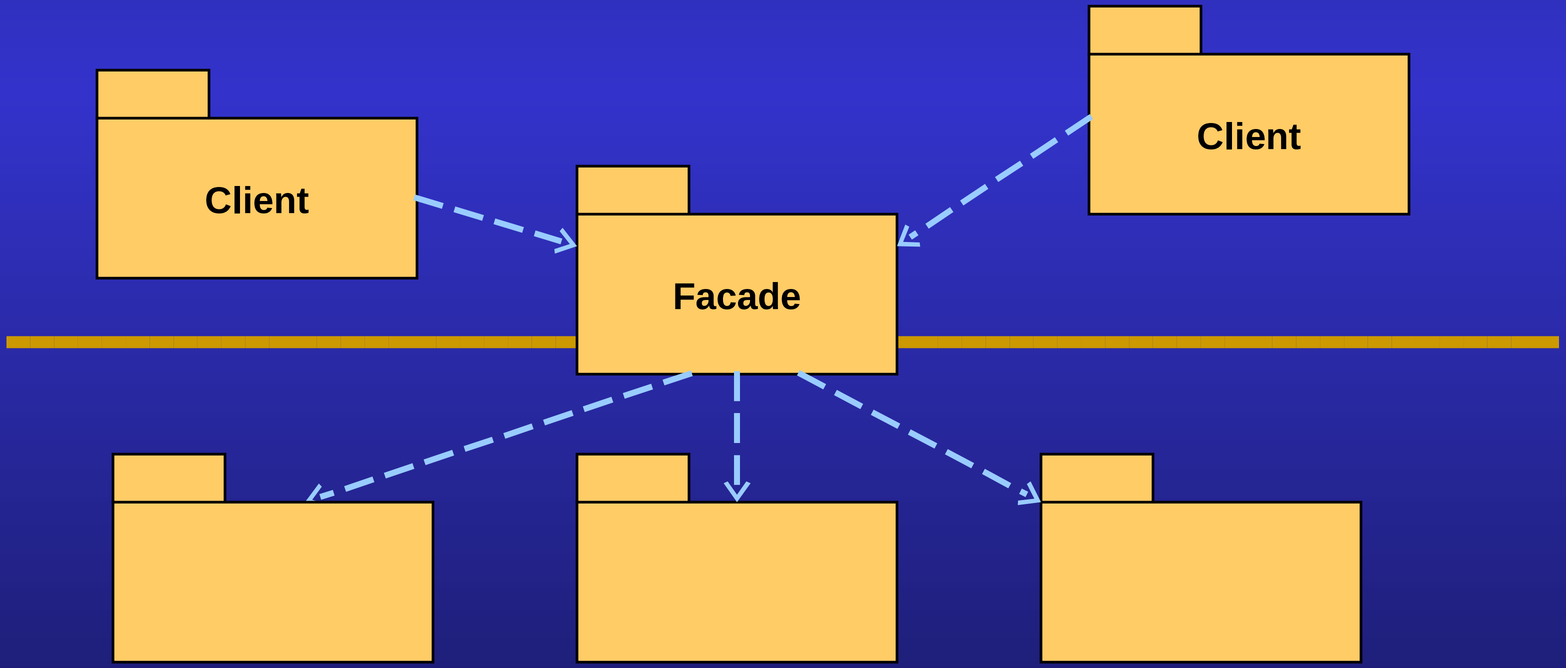
Facade Objects

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

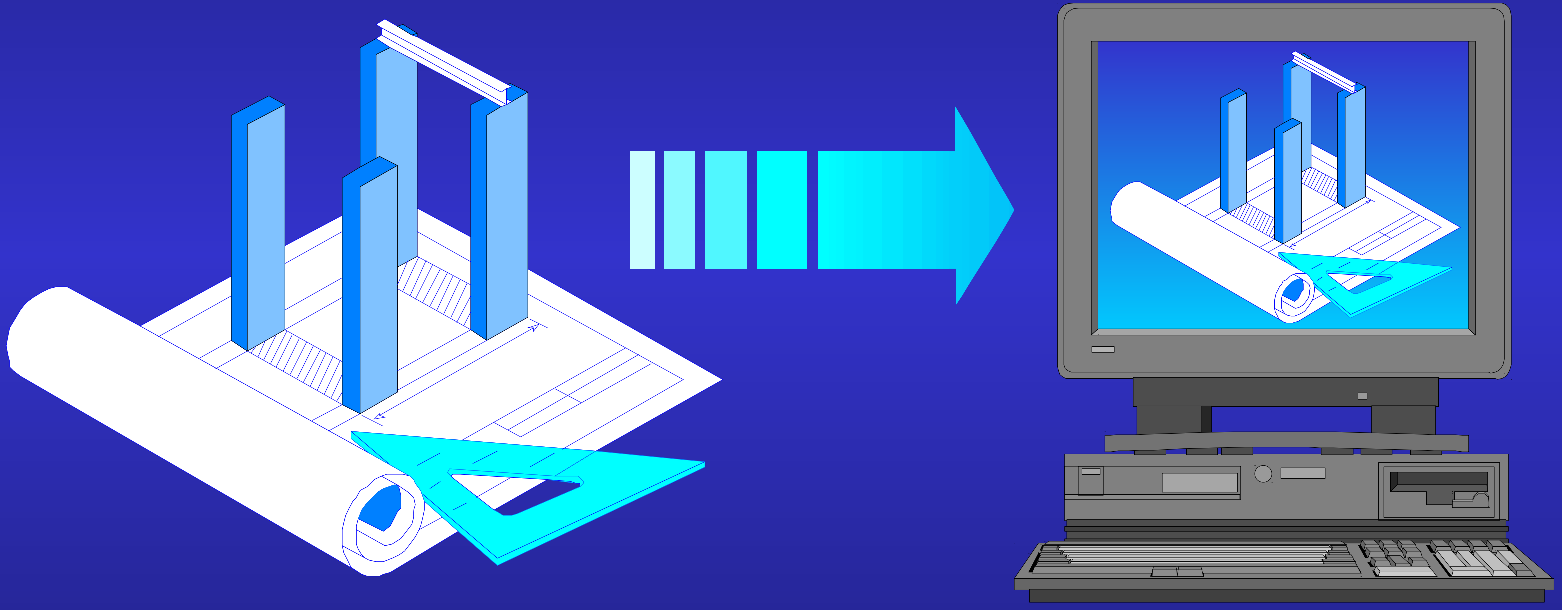


Facade Packages

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



Patterns for Object Management



Pattern-Based Software Development

Patterns for Object Management

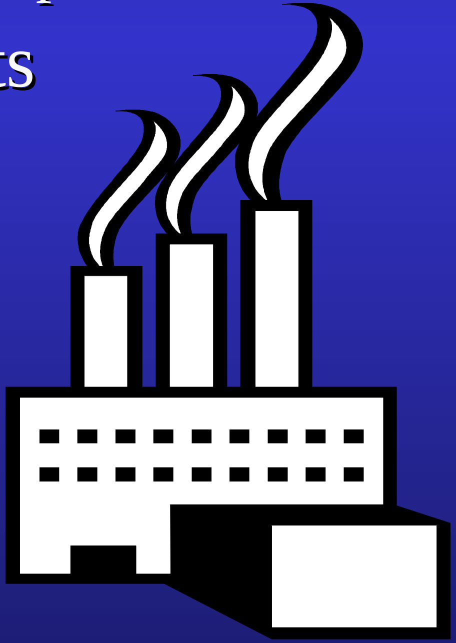
- Objectives
  - Present the issues that motivate the use of and policies for object factories and managers
- Contents
  - Object factories
  - The Factory Method pattern
  - The Disposal Method pattern
  - The Manager pattern
  - The Leasing pattern
  - The Evictor pattern

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?

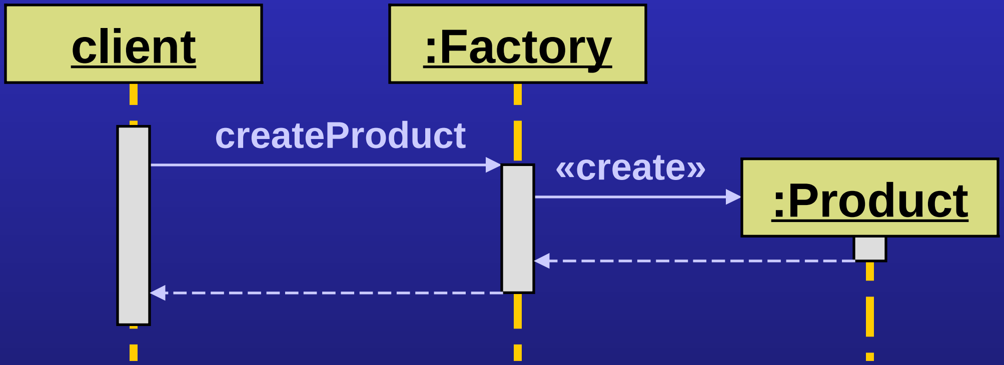
Object Factories

- 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



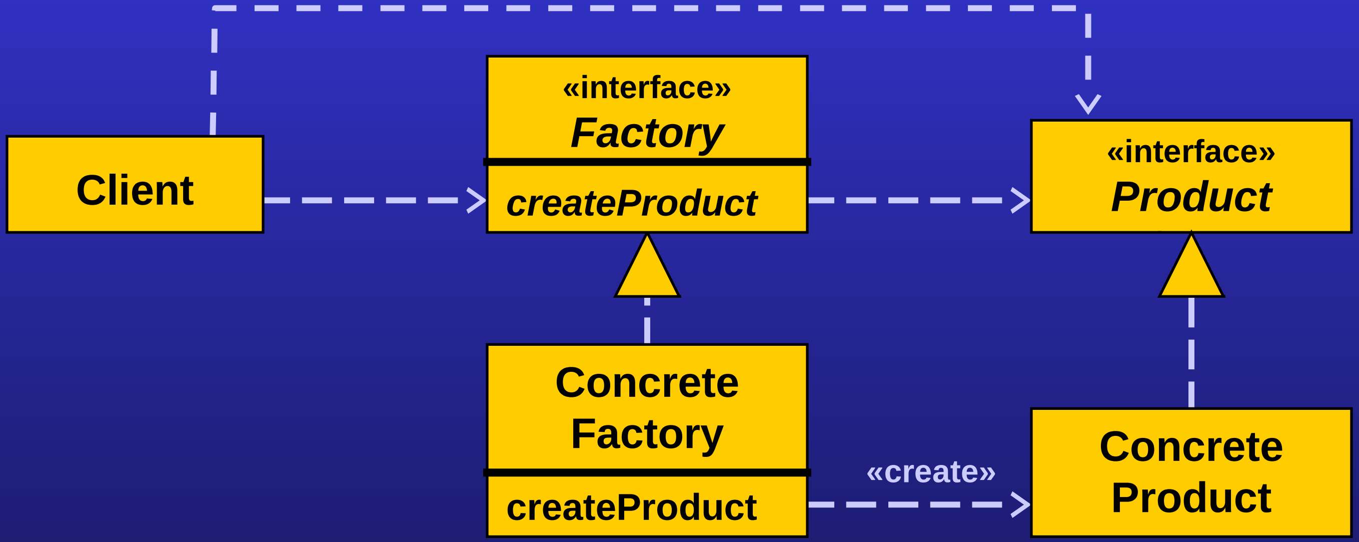
The Factory Method Pattern

- 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



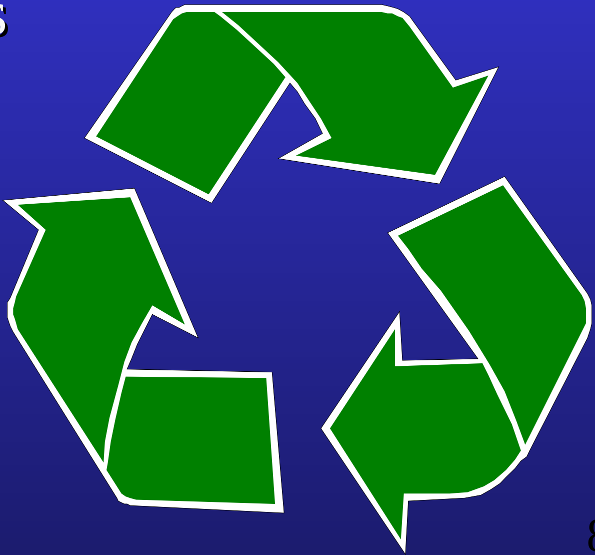
Typical Factory Configuration

- 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



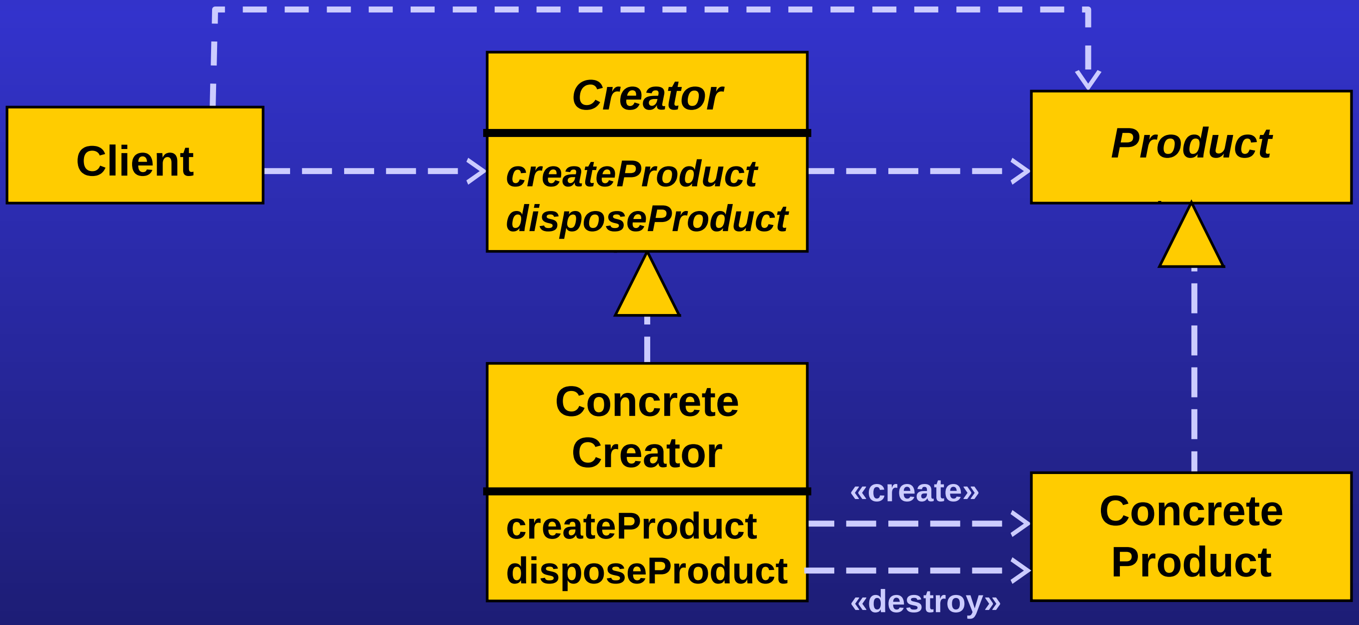
The Disposal Method Pattern

- 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



Disposal Method Sketch

- Reflecting the use of Factory Method...
  - Responsibility for disposal — destruction or recycling — is collocated with the creation
  - Disposal extends the role of the creating object



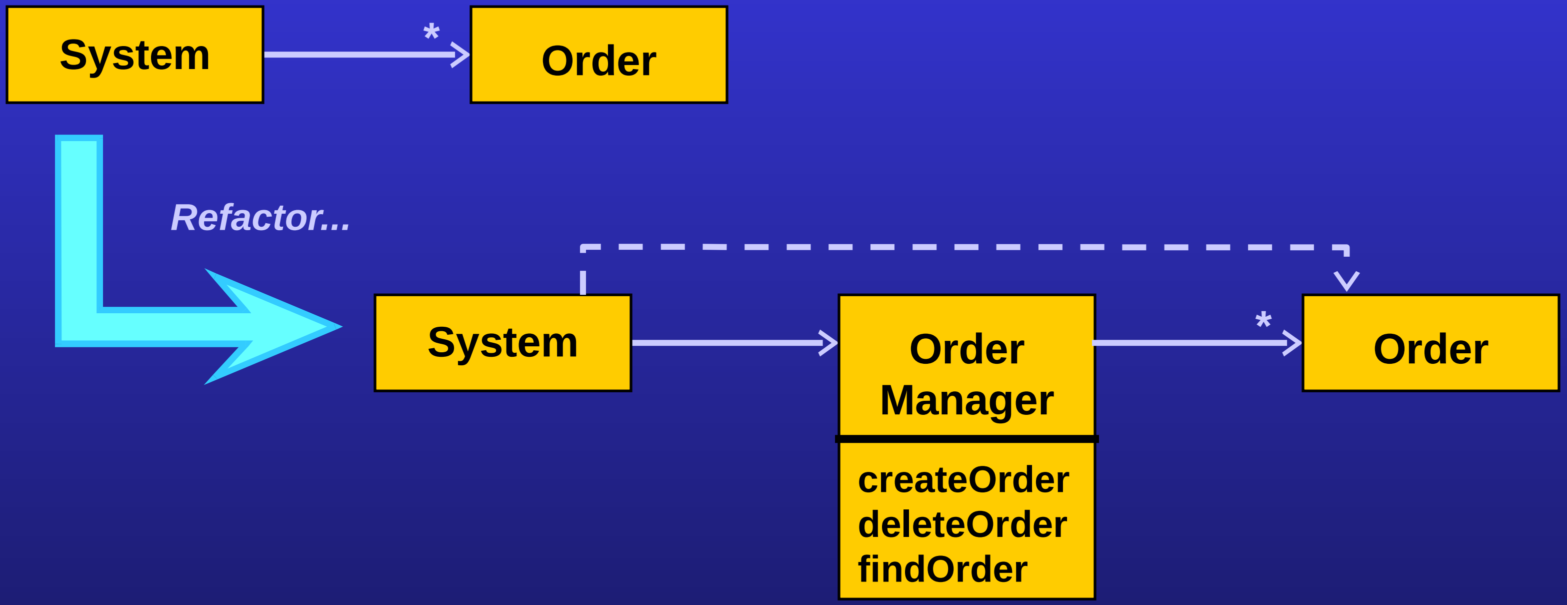
The Manager Pattern

- 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

© Curbralan Ltd

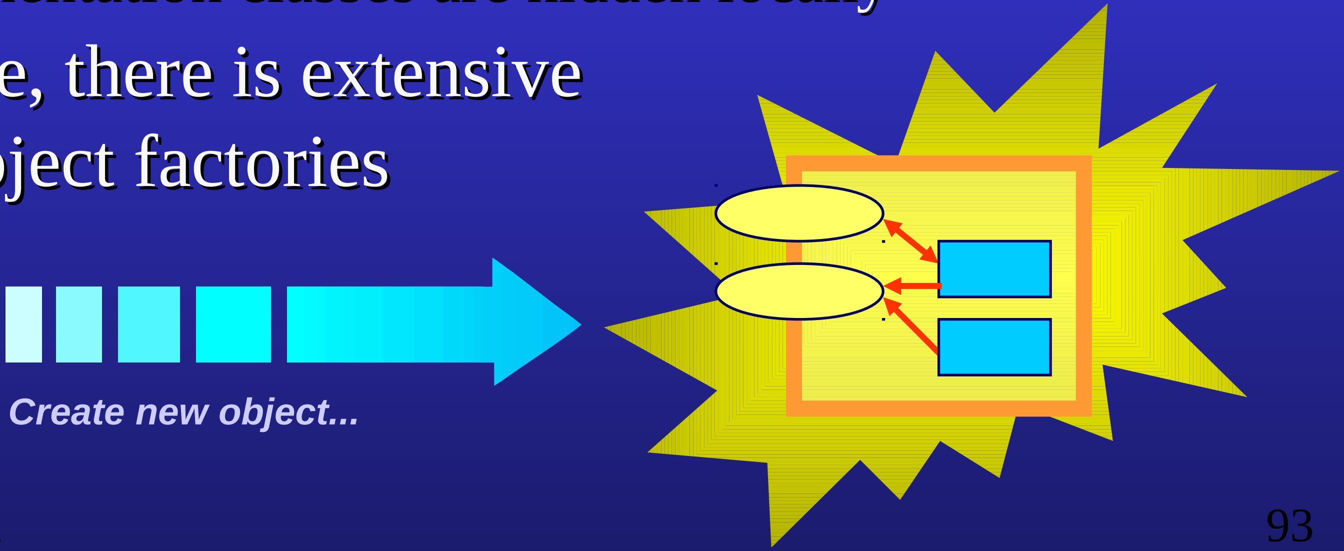
Introducing a Manager

- 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



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

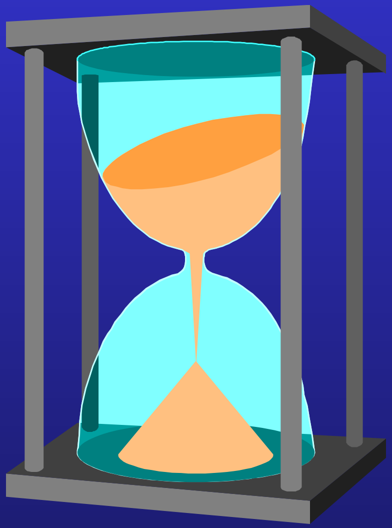


Different Lifetime Models

- *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

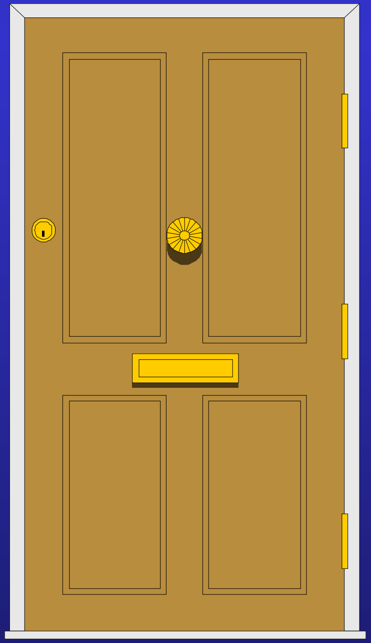
The Leasing Pattern

- 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

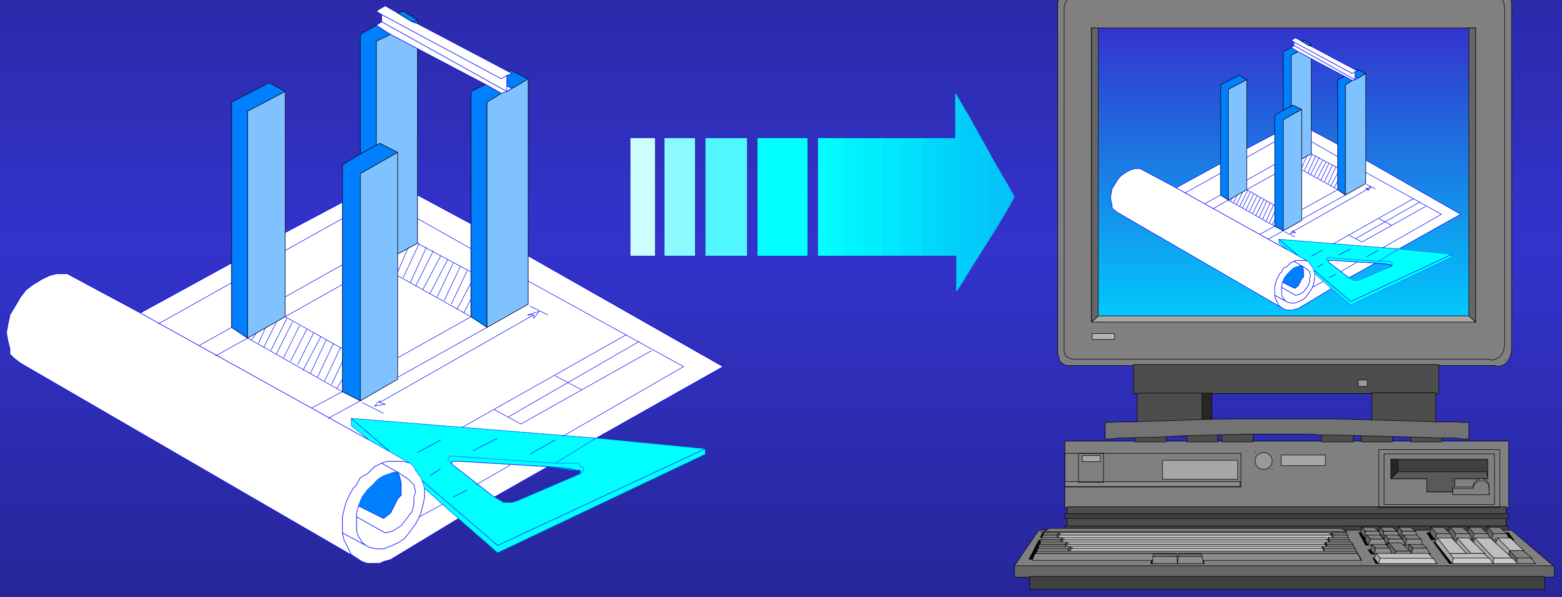


The Evictor Pattern

- 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



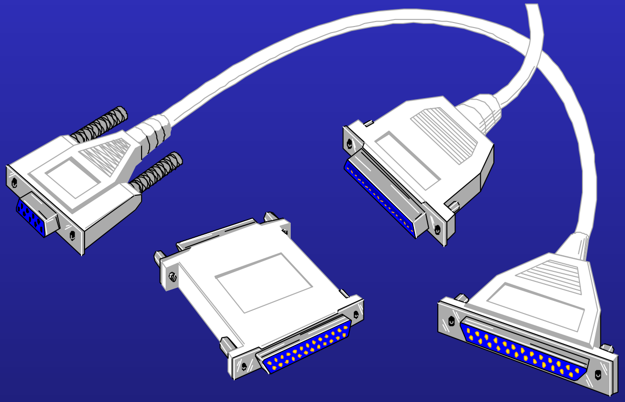
Patterns for Pluggability



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



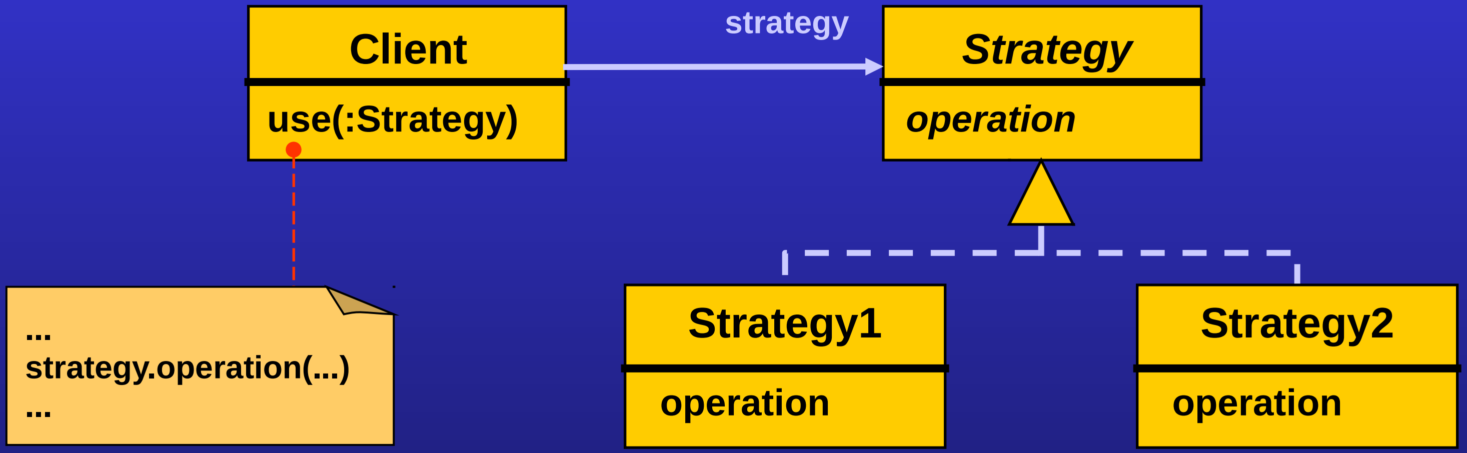
© Curbralan Ltd

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

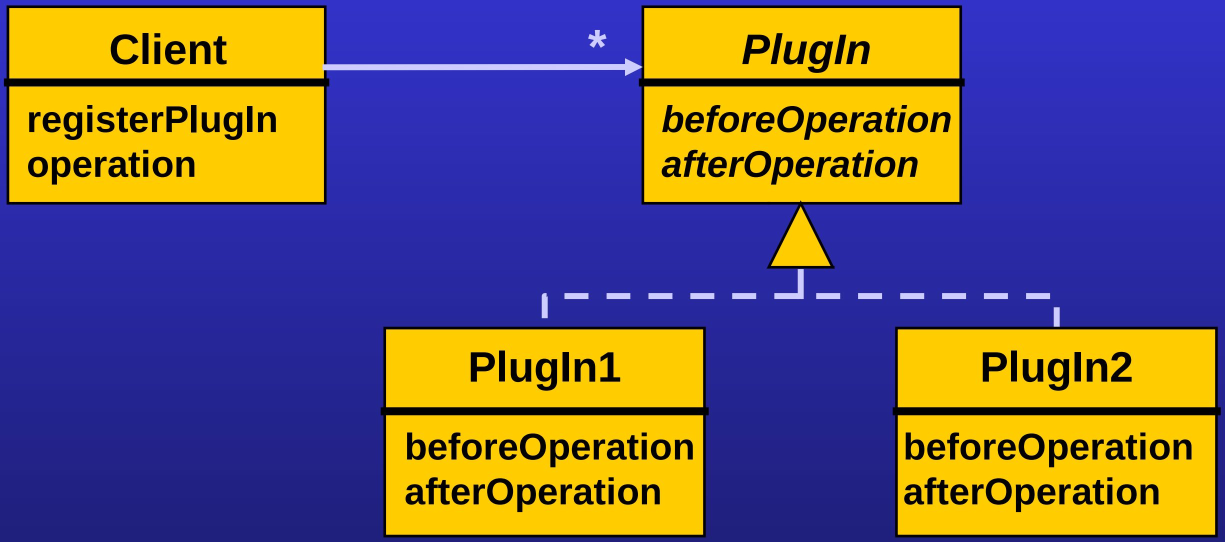
Strategy Configuration Sketch

- 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



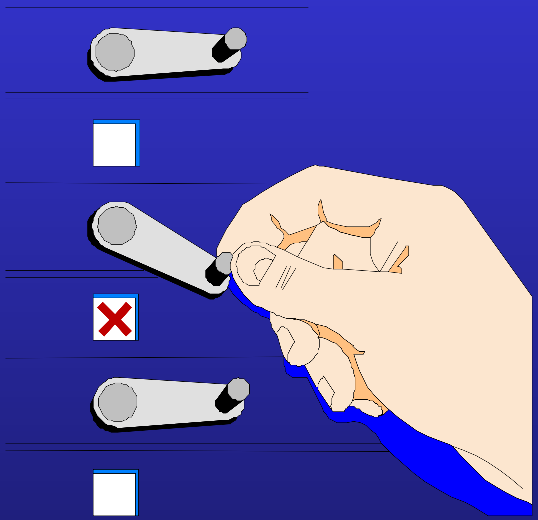
The Interceptor Pattern

- 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



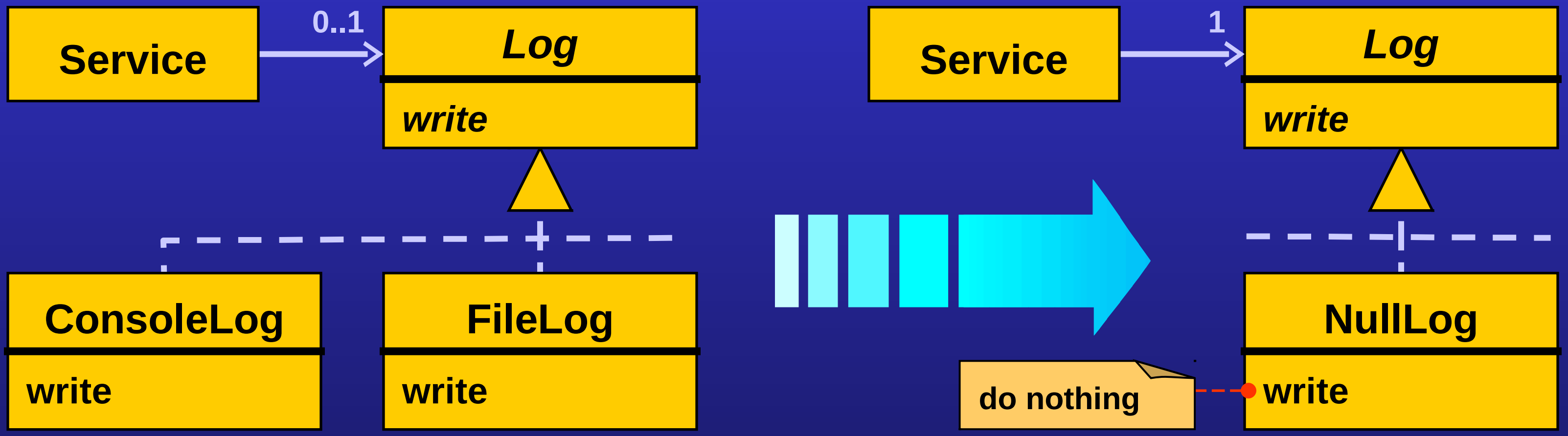
Optional Object Relationships

- 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



The Null Object Pattern

- A Null Object can be used to replace a null relationship, simplifying calling code
  - A Null Object is an implementation of a polymorphic interface where each method does nothing
  - The initialiser of the relationship needs to ensure that a Null Object is used instead of a null reference

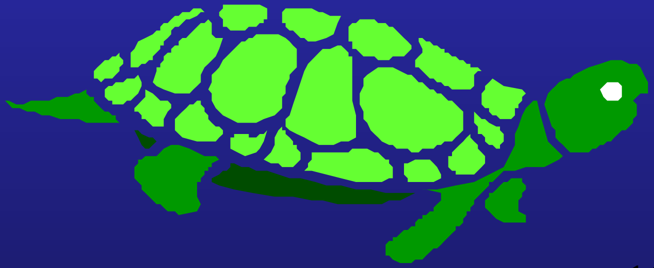


The Context Object Pattern

- 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*

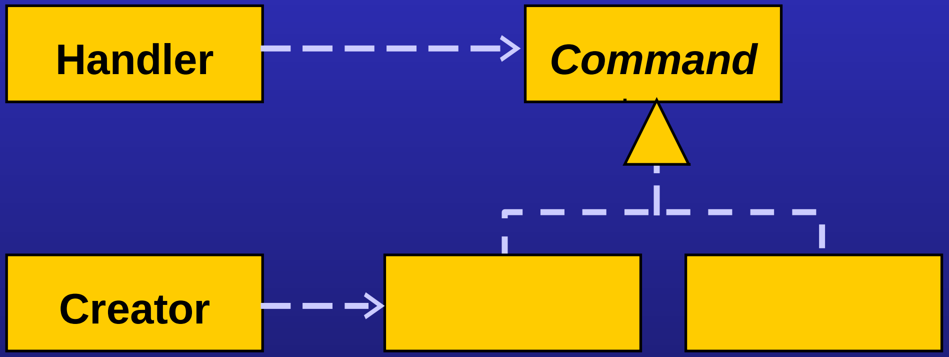
The Mock Object Pattern

- 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



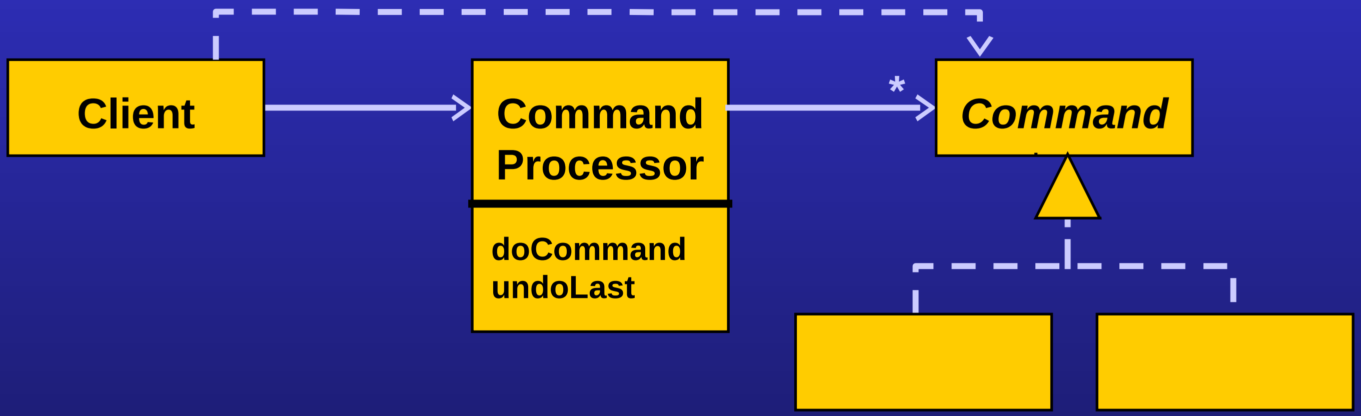
The Command Pattern

- 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



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



© Curbralan Ltd

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;
}
```

The Block Pattern

- Passing a block of code as an object...
  - Use anonymous inner classes
  - Code dynamically bound to its context
  - Simplifies use of task–object-based idioms
- A Java idiom for expressing simple Commands

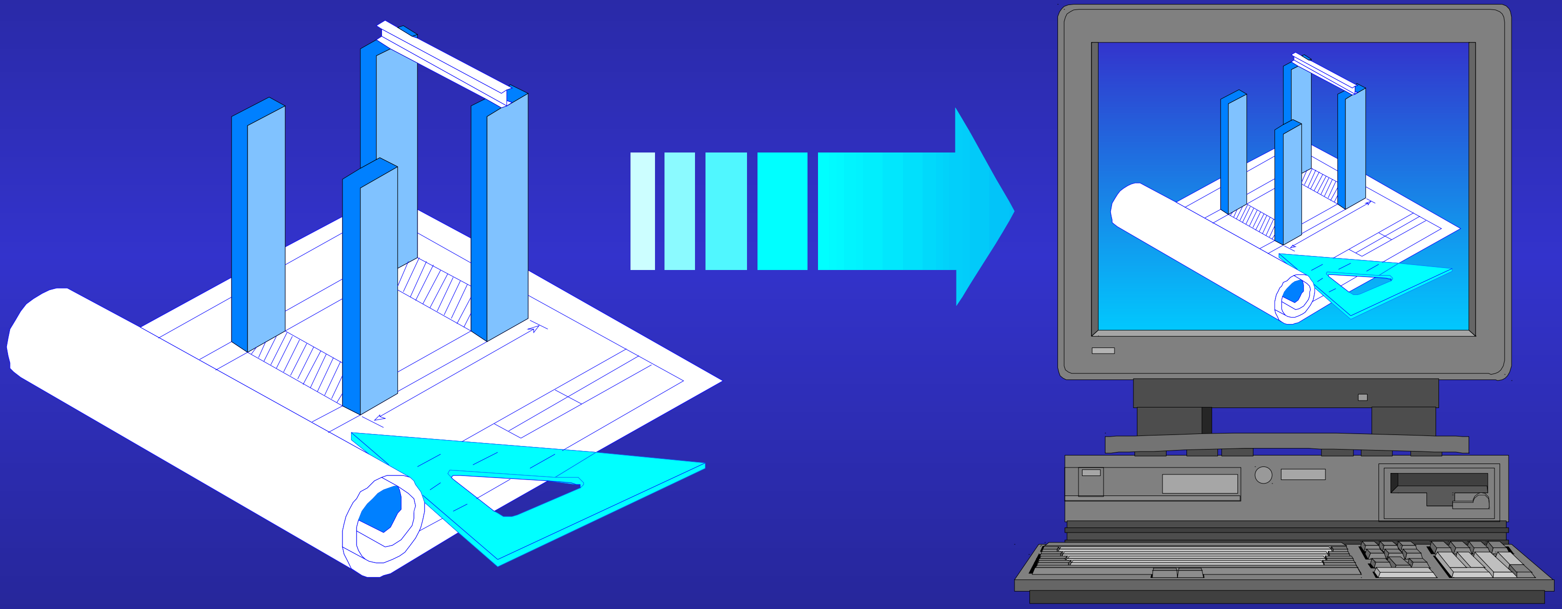
```
class Timer
{
  public Timer(
    int when, Command what) …
  public void start() …
  …
}
```

Callback on expiry

Set up timer callback

```
new Timer(delay,
  new Command() {
    public void execute()
      { block of code }
  }).start();
```

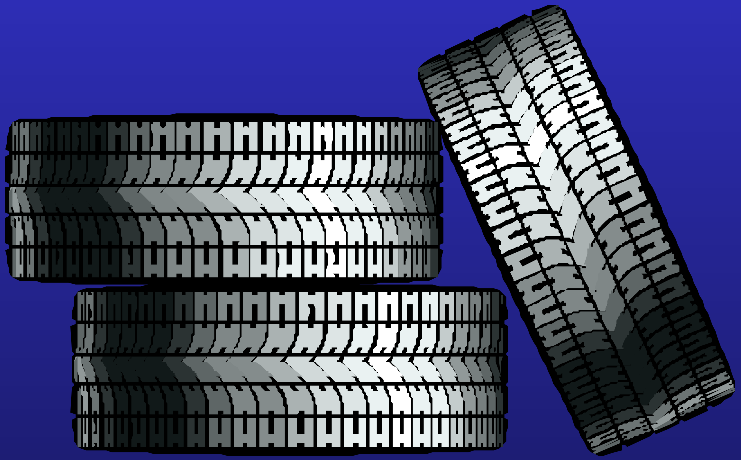
Patterns for Iteration



Pattern-Based Software Development

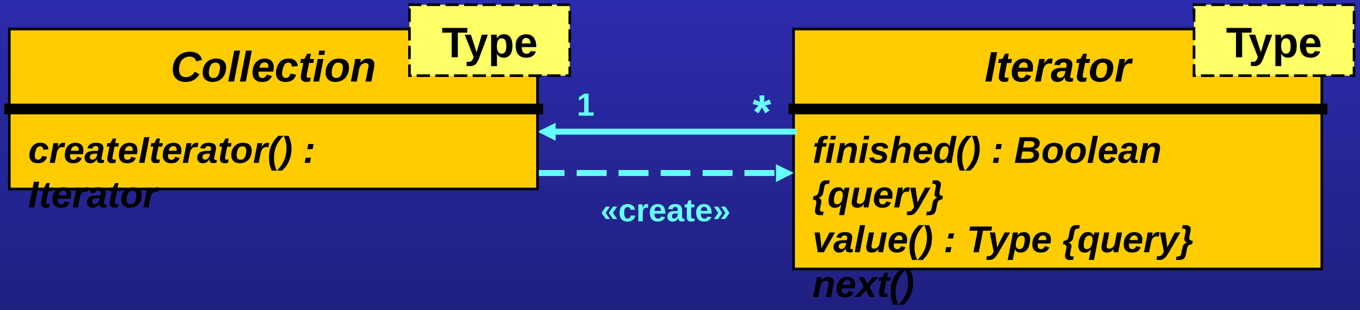
Patterns for Iteration

- 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



Standard Iterators

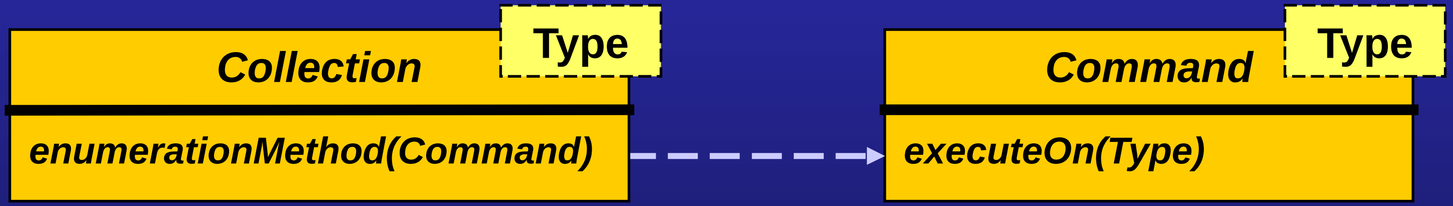
- Java library offers polymorphic iterators
  - Support for *for*-each loops exists for *java.lang.Iterable* classes, which return *java.lang.Iterator*

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

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

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



© Curbralan Ltd

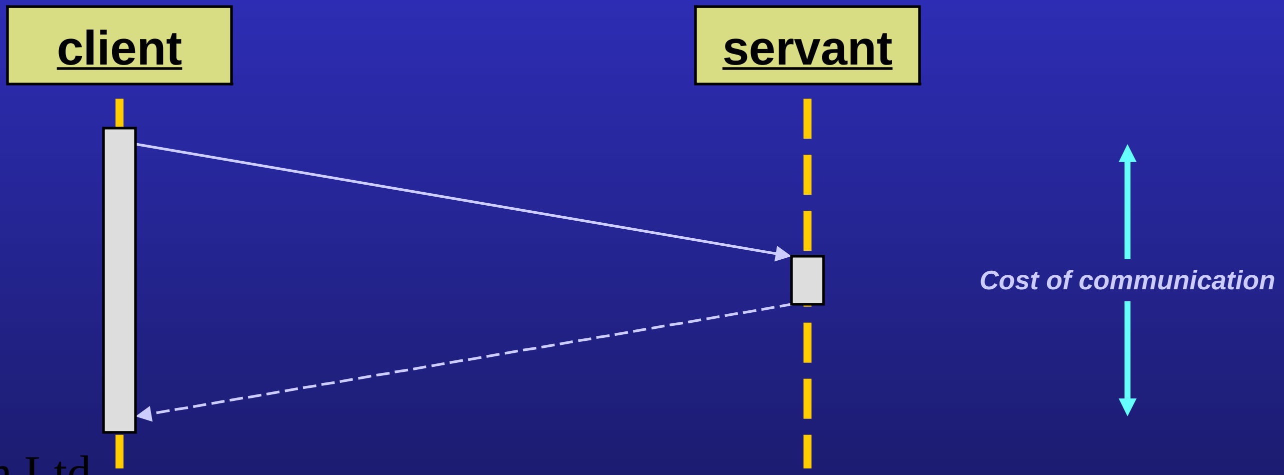
Enumeration Method in Java

- 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); }
            });
    }
    ...
}
```

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



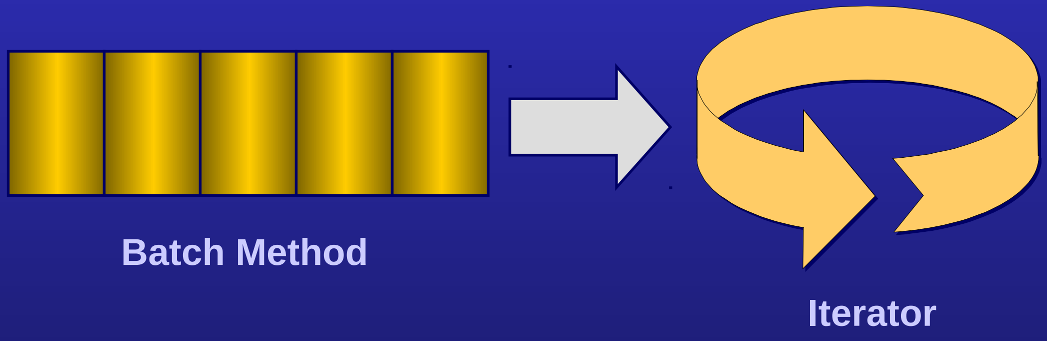
The Batch Method Pattern

- Iterated simple methods use up bandwidth
  - Spending far more time in communication than in computation
- So, provide the repetition in a data structure
  - More appropriate granularity that reduces communication and synchronisation costs

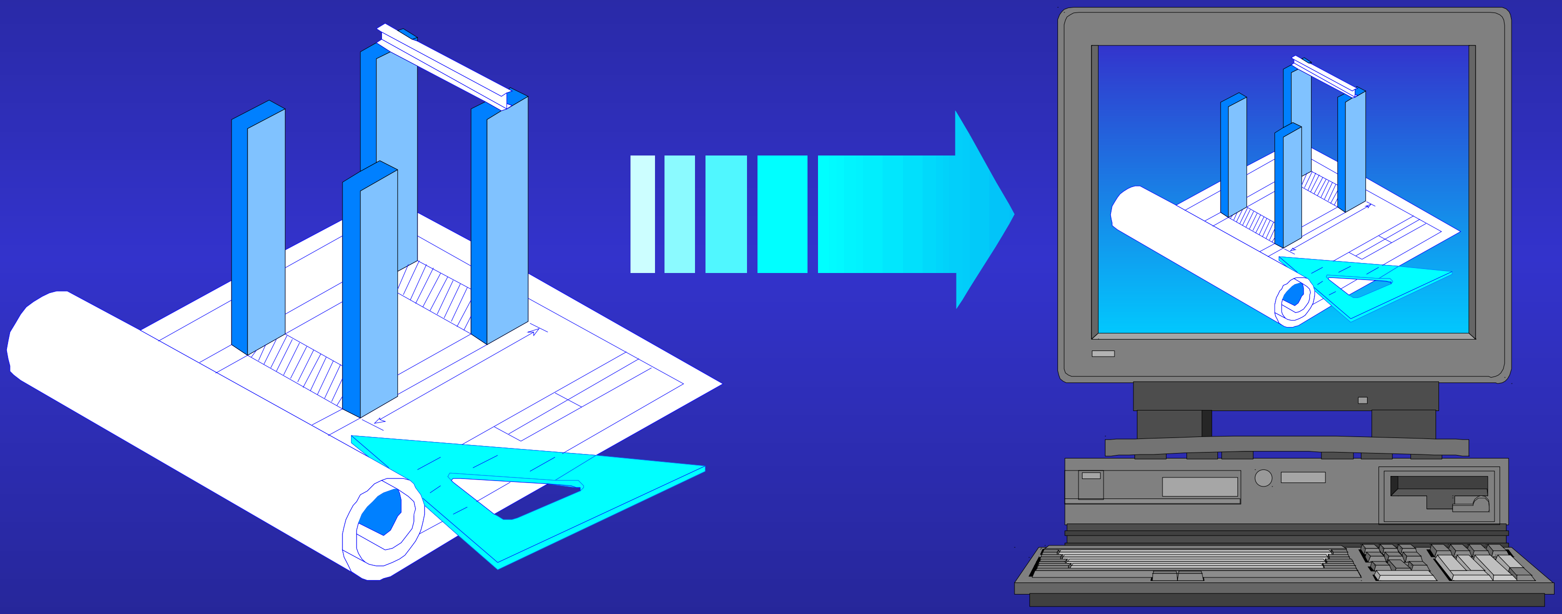
```
interface RemoteDictionary<Key, Value> ...
{
    Value lookupValue(Key key);
    Value[] LookupValues(Key[] keys); // batch method
    ...
}
```

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



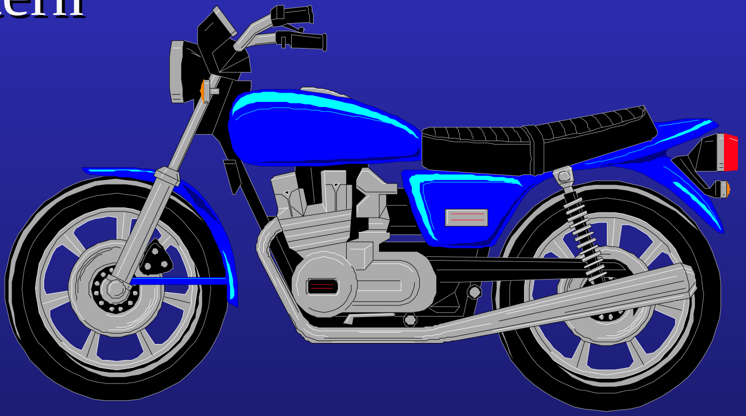
Patterns for Object Lifecycles



Pattern-Based Software Development

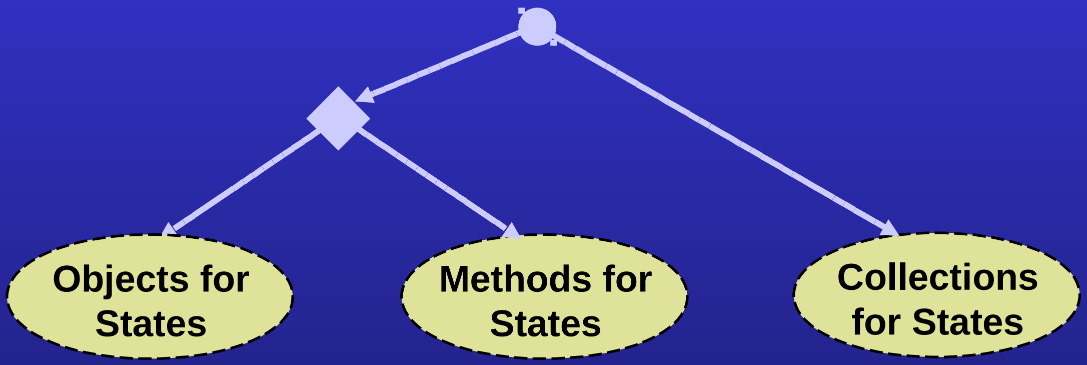
Patterns for Object Lifecycles

- 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

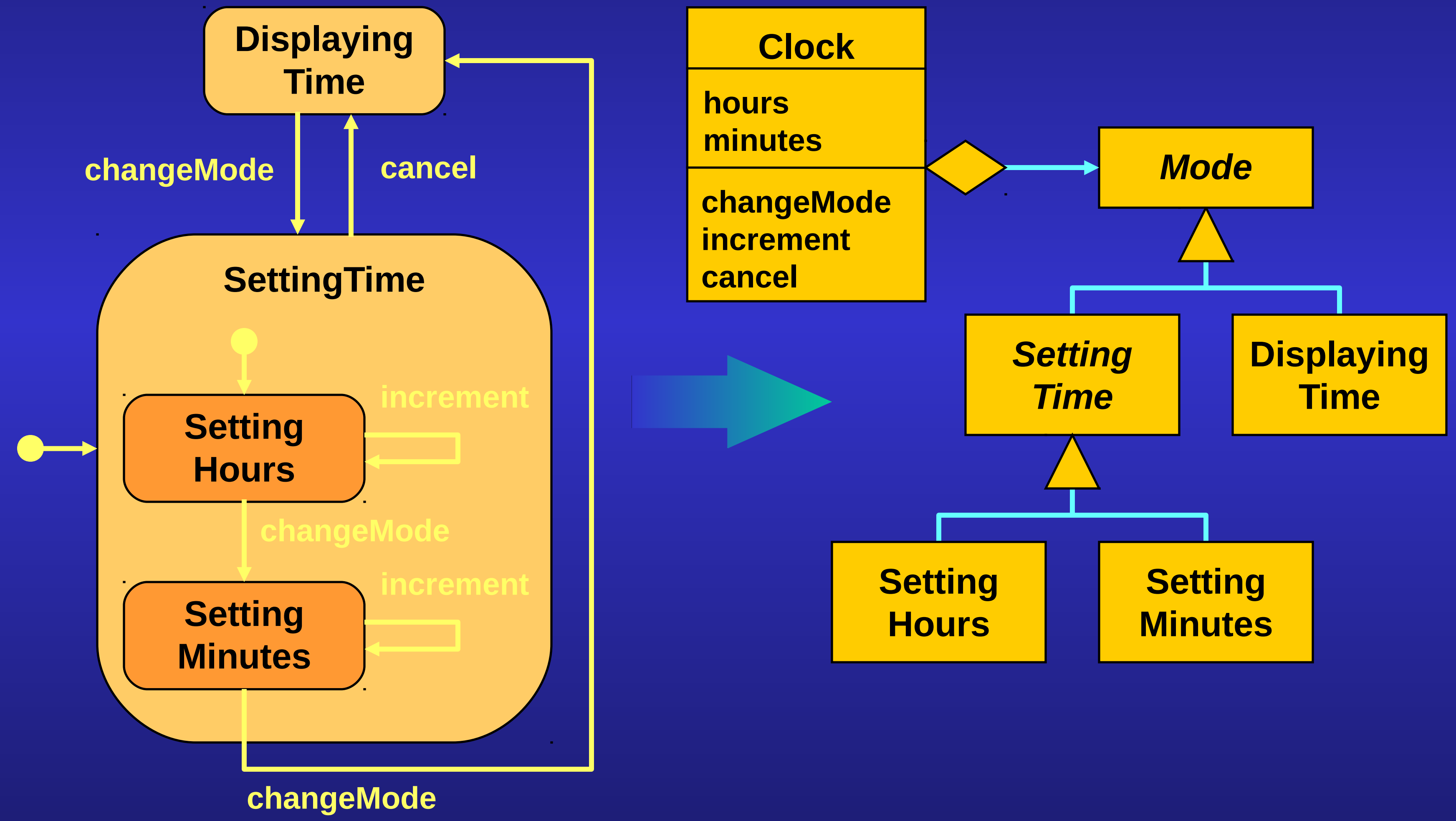


Modal Behaviour

- 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



The Objects for States Pattern



Displaying Time

changeMode

cancel

SettingTime

Setting Hours

increment

changeMode

Setting Minutes

increment

changeMode

Clock

hours
minutes

changeMode
increment
cancel

Mode

Setting Time

Displaying Time

Setting Hours

Setting Minutes

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;
}
```

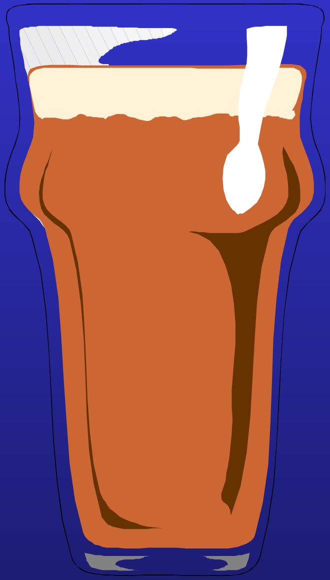
Stateless Mode Objects

- The context object can be passed explicitly
  - Behavioural objects are stateless
  - Such objects are shareable
  - Reduces creation costs

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

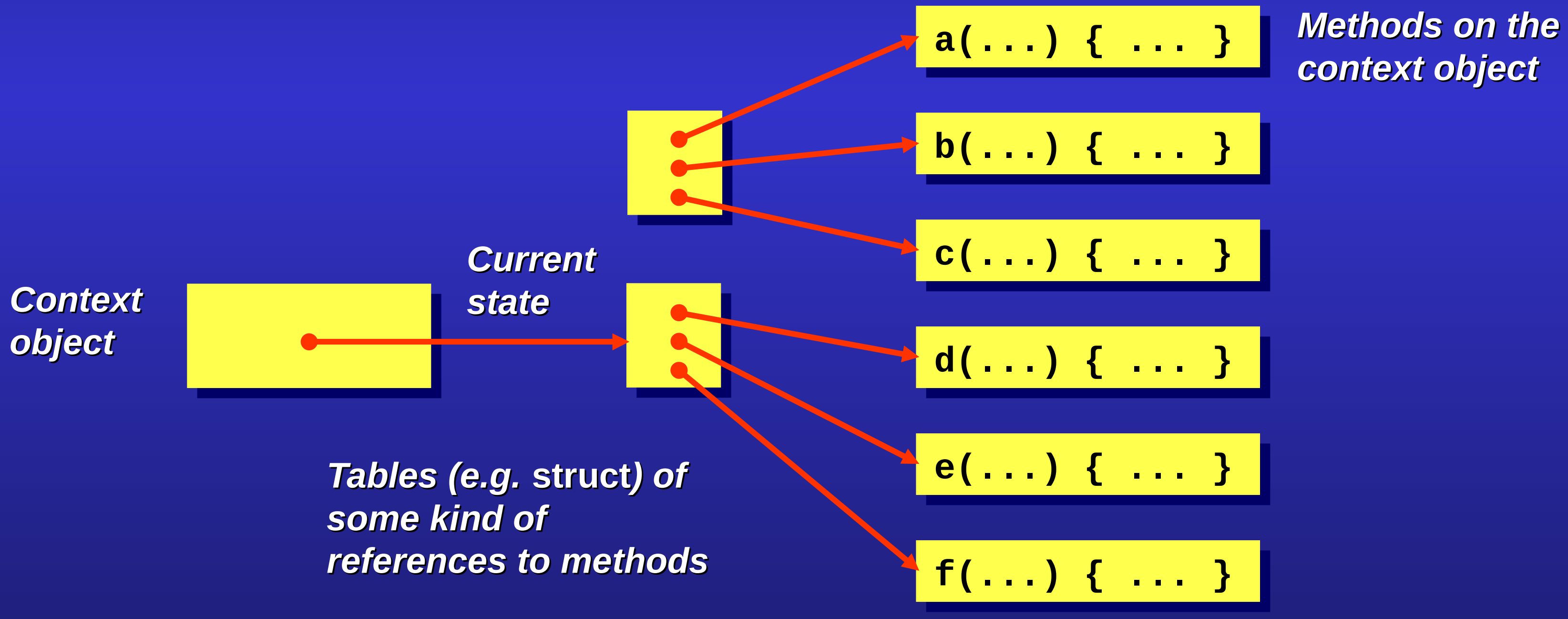
The Methods for States Pattern

- 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



Referencing Methods in a State

- All calls on the context object are forward to the appropriate method via the table
  - Effectively, DIY *vtables*



Methods on the context object

Current state

Context object

Tables (e.g. struct) of some kind of references to methods

Collections of Objects

- How should state be represented for objects that are managed collectively?

```
public class SaveableObjectManager
{
    public void saveChanges() ...
    ...
    private Collection saveableObjects;
}
```

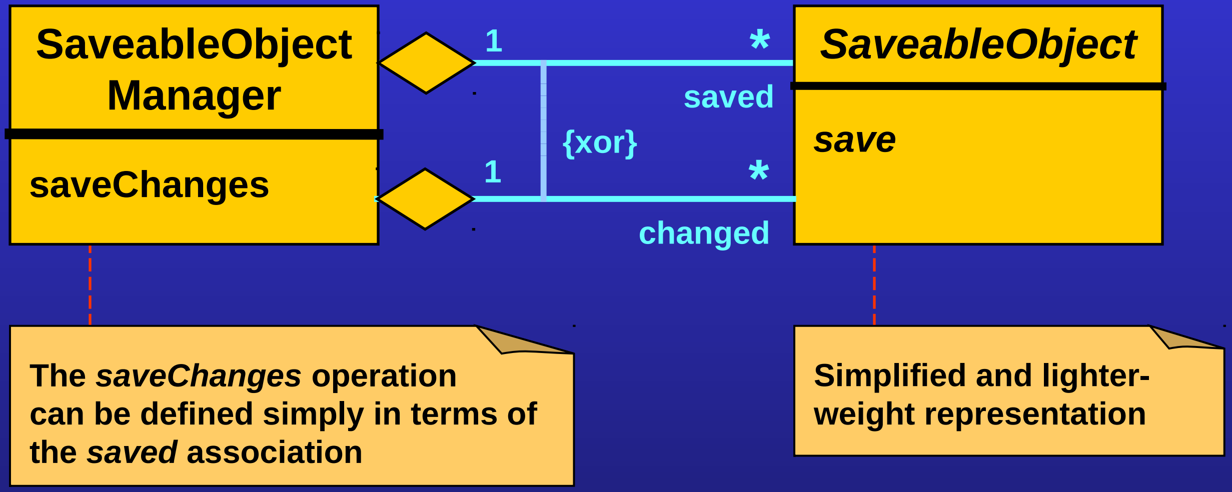
Manager object responsible for controlling many other objects

```
public abstract class SaveableObject
{
    public abstract void save();
    ...
}
```

Object with simple lifecycle model, i.e. transition between *saved* and *changed* states

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

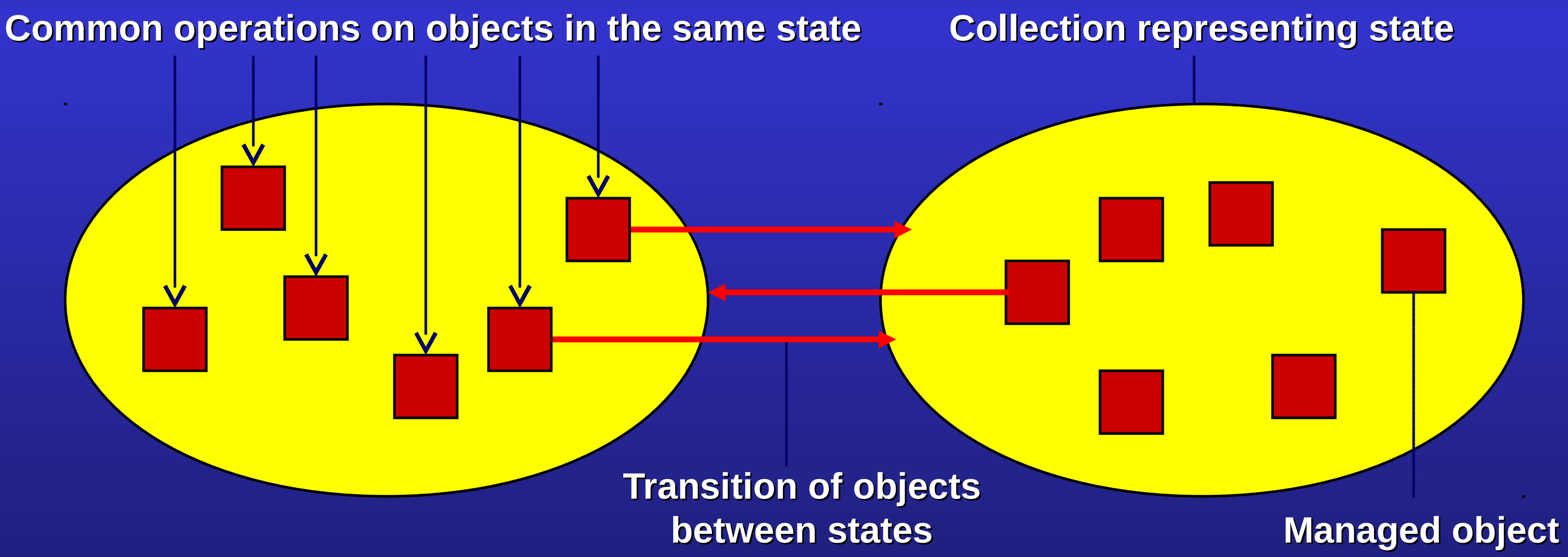


The *saveChanges* operation can be defined simply in terms of the *saved* association

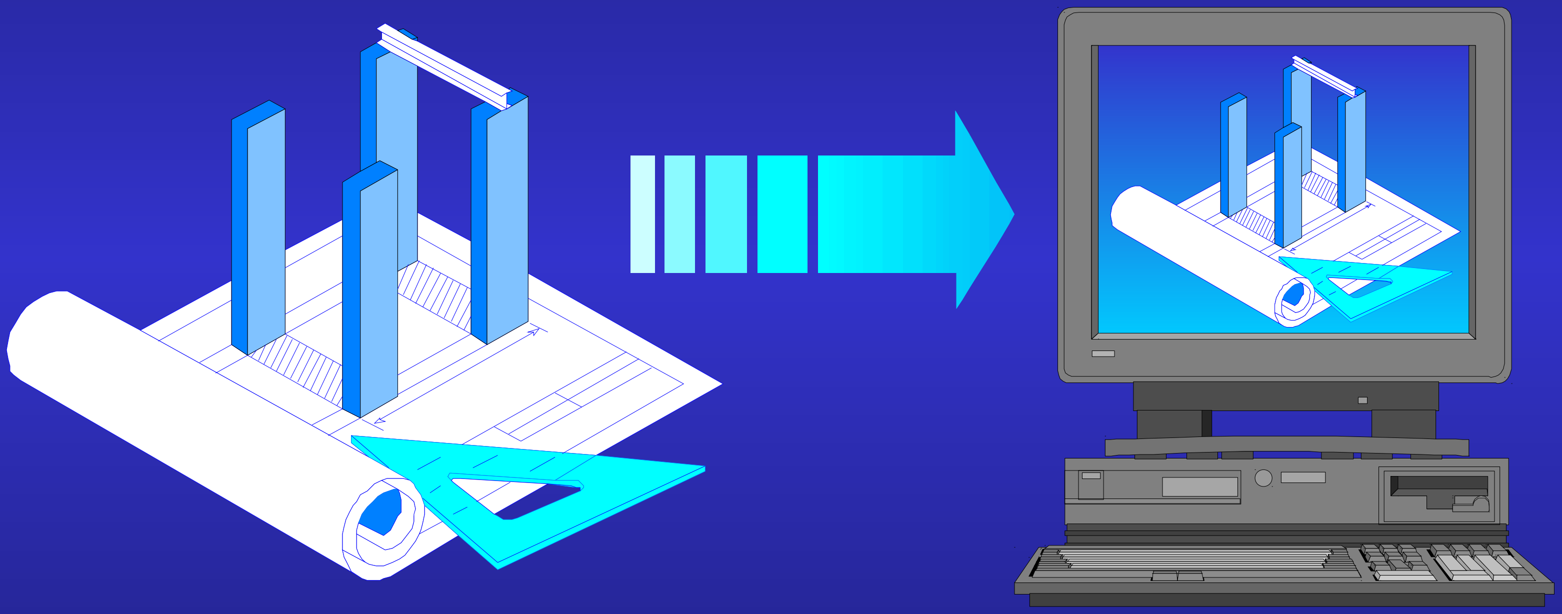
Simplified and lighter-weight representation

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


Common operations on objects in the same state
Collection representing state
Transition of objects between states
Managed object

Patterns for Notification



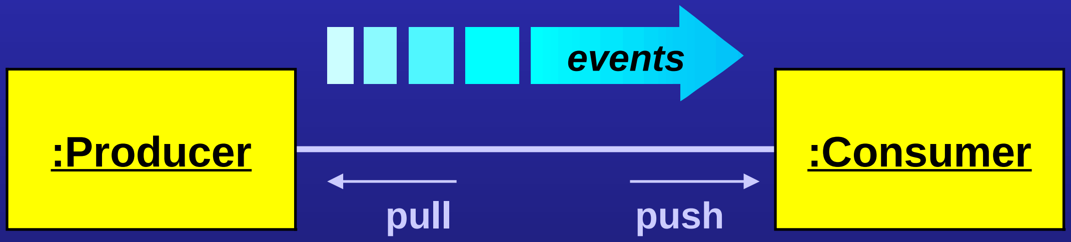
Pattern-Based Software Development

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



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

```
interface Producer
{
    Event pull();
    Event tryPull();
    ...
}
```

Blocking

Non-blocking

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

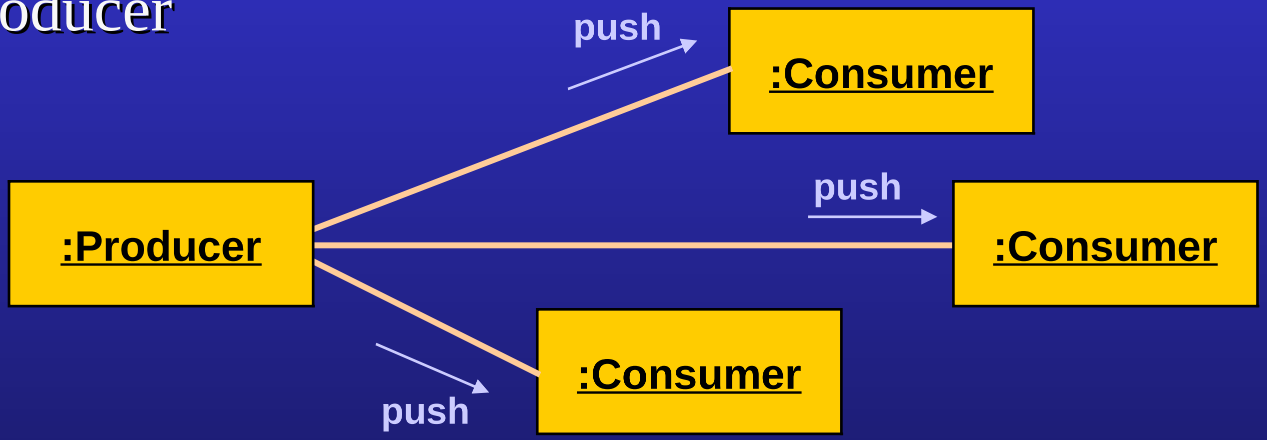
General notification

Specific notification (alternative)

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

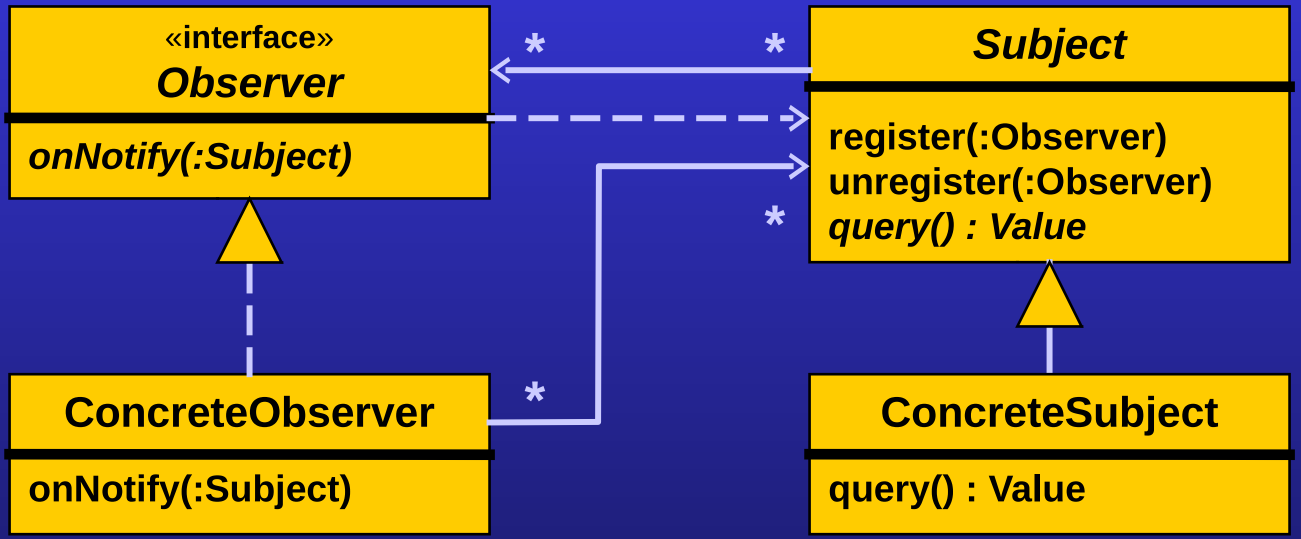
Multiple Notification

- 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



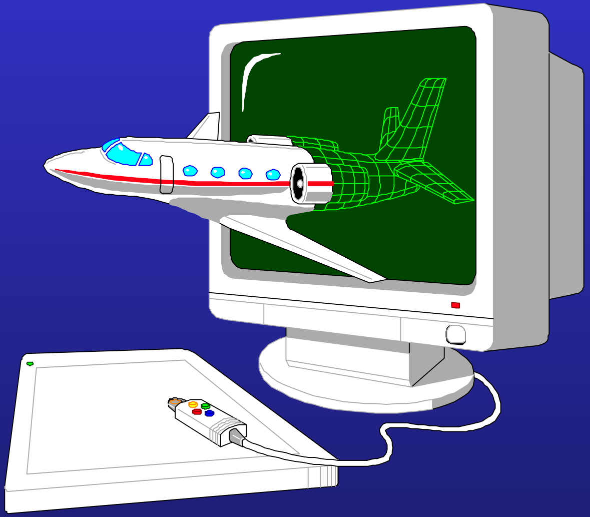
The Observer Pattern

- 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



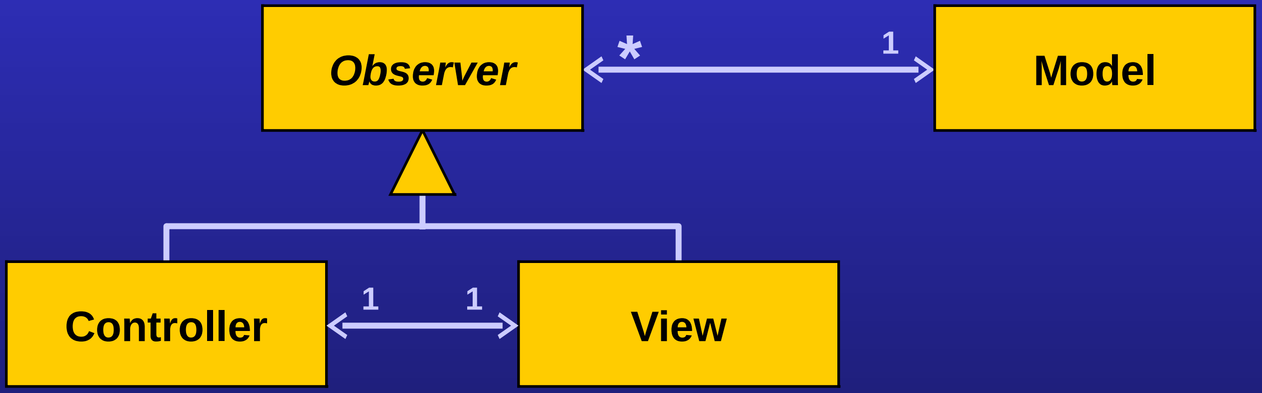
The Model–View–Controller Pattern

- 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



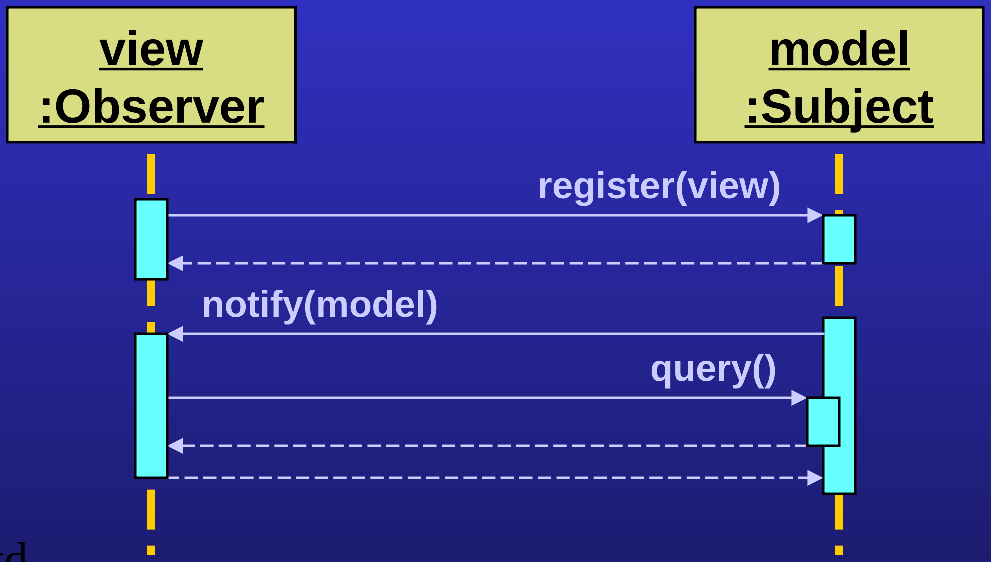
Typical MVC Configuration

- 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



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

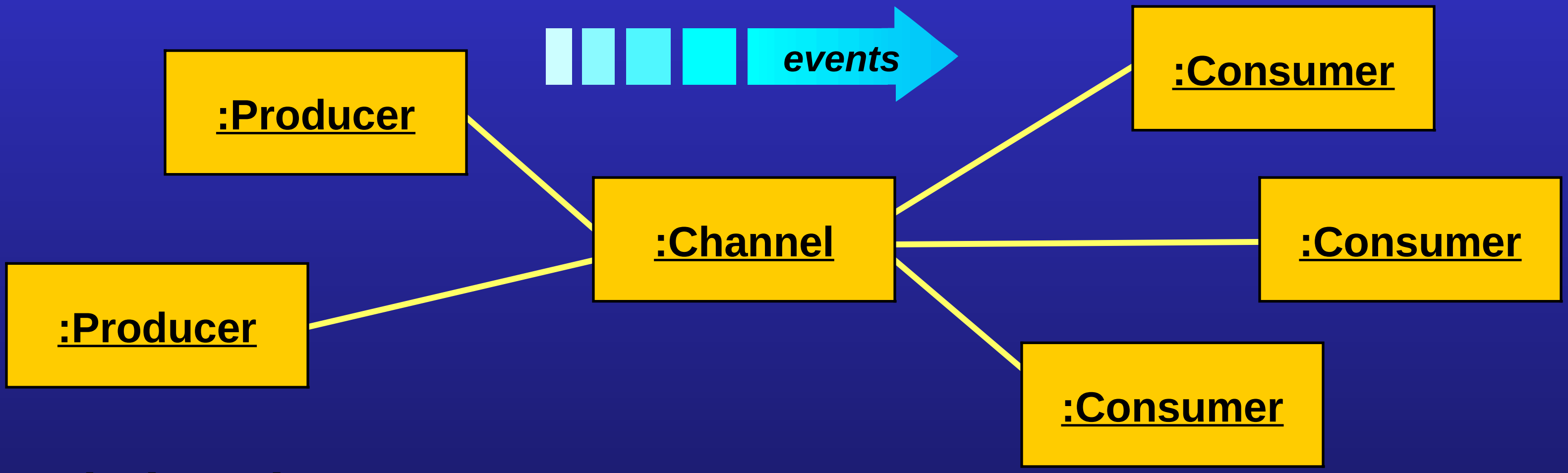


Combined and Batched Notifications

- 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

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

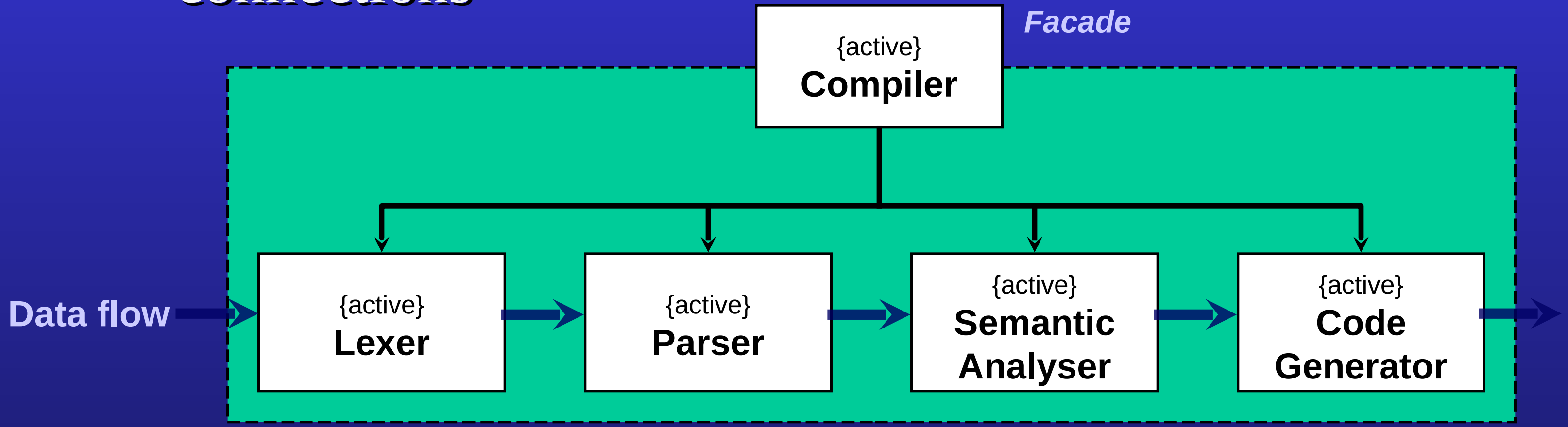


Message Queues

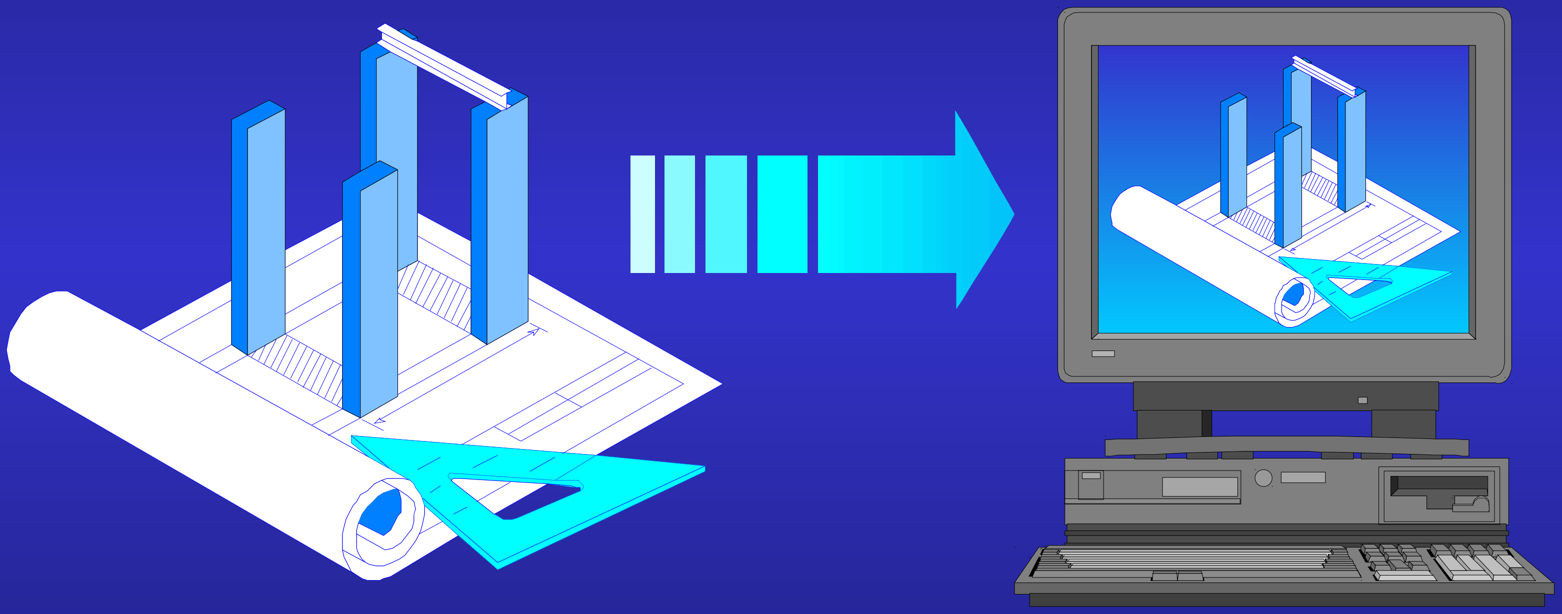
- A variation on eventing and publish–subscribe is point-to-point, asynchronous messaging
  - In contrast to classic publish–subscribe, sender and receiver identity are normally significant for communication via asynchronous message queues
- Message queues are used for their persistence, resilience and 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

The Pipes and Filters Pattern

- Divides transformation based system into independent processing steps connected by pipes
  - These individual steps are at least conceptually concurrent, and may actually be concurrent
  - Often modelled as a pipeline of processes with connections



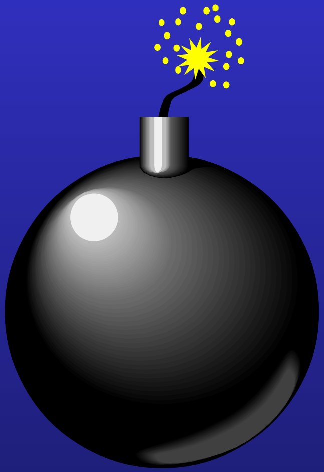
Pattern Pitfalls



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

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

Reference Models and Blueprints?

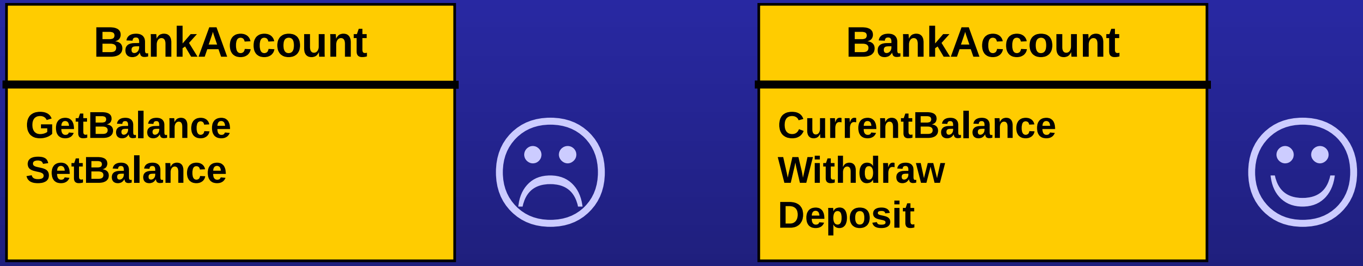
- A motivating example is sometimes taken to be a pattern's reference implementation
  - Copy-and-paste rarely gives a suitable solution, adaptation is always needed to fit the design together and have it make sense in context
- That a pattern is a blueprint or simply parameterized design is a common 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

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

The Getters and Setters "Pattern"

- A classic dysfunctional pattern
  - Pairing every private field with a *get* with a *set* method
- Appeals to a false symmetry
  - A *get* does not have the opposite effect to a *set*
- In some interfaces, *gets* and *sets* end up matched, but this is a consequence of other requirements
  - Not a design in itself



BankAccount

GetBalance
SetBalance

☹

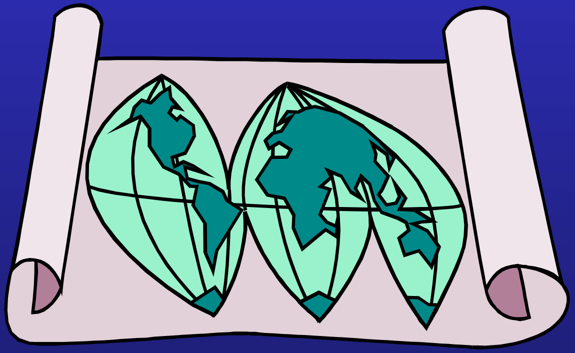
BankAccount

CurrentBalance
Withdraw
Deposit

☺

The Singleton Pattern (and Avoiding It)

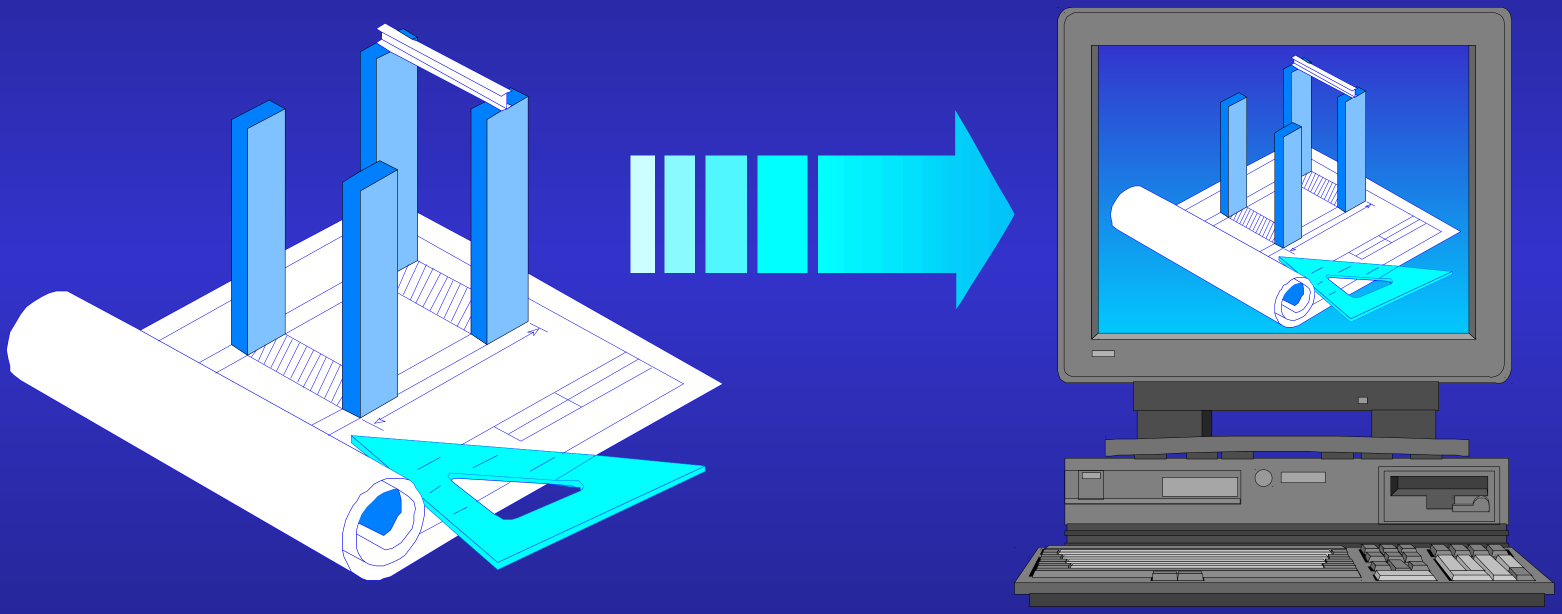
- The notion is that a Singleton object enforces a creational constraint on an object type
  - Only a single instance of a class can be created, and public, global access is provided to it via a *static* method
- Singleton is (very) frequently misused
  - Singleton is sometimes seen as a legitimate way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



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

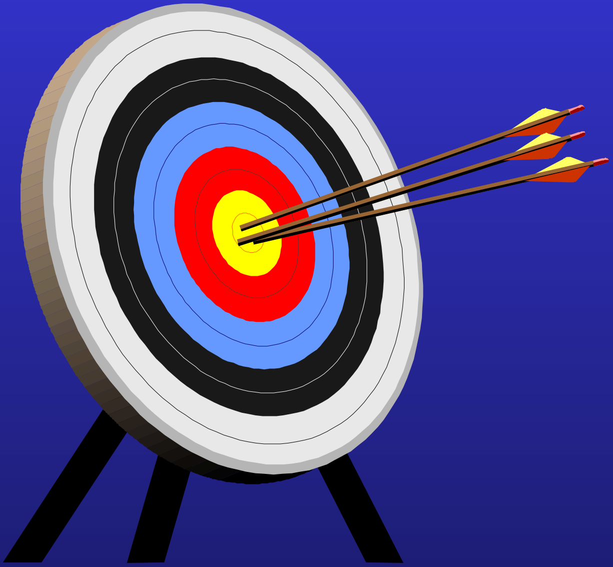
Course Outroduction



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

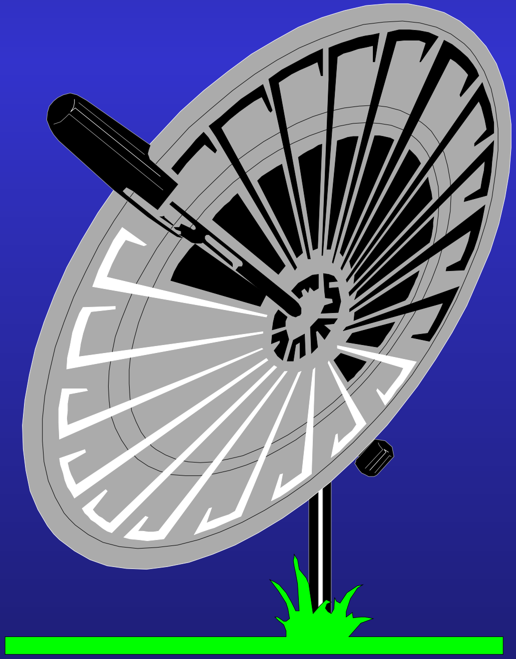


Objectives Revisited

- Understand what does and does not go to make up a pattern ✓
- Understand the role of patterns in software architecture ✓
- Learn some common patterns for object-oriented design ✓
- Appreciate patterns from the strategic level to examples in code ✓

Moving Beyond

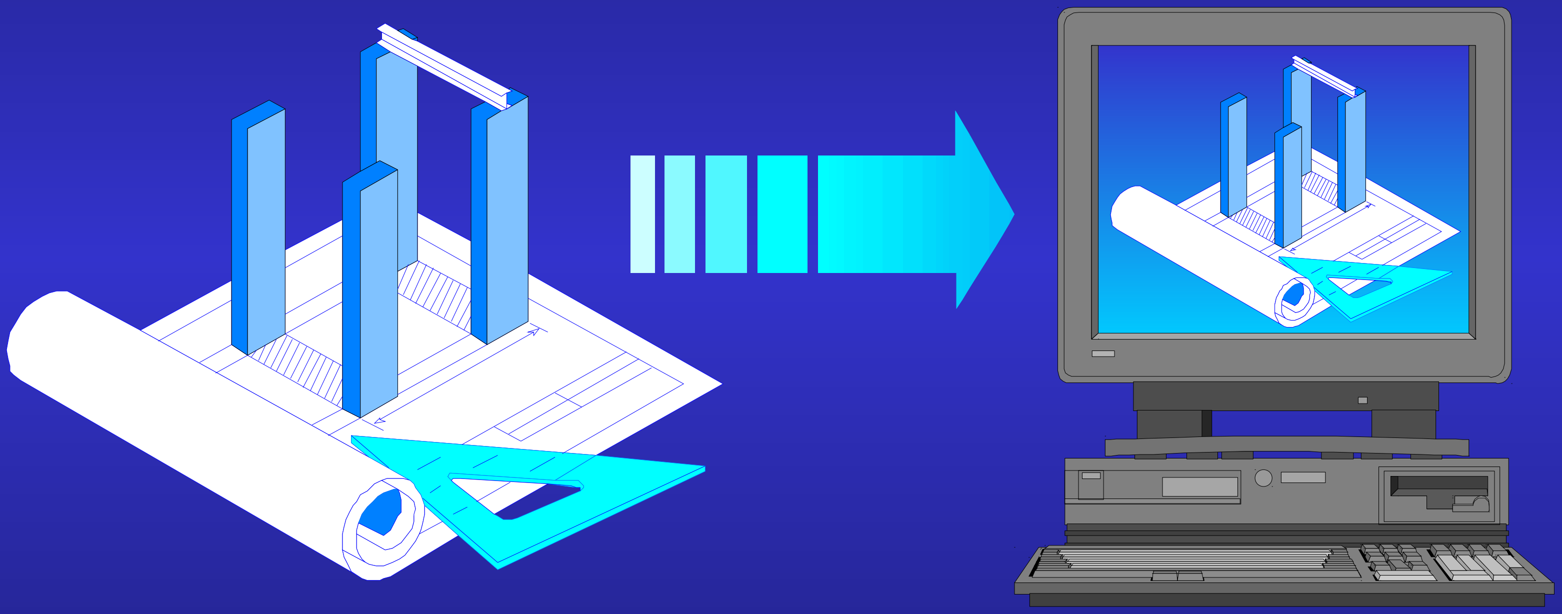
- The course represents a beginning not an end to understanding patterns
  - Hands-on experience and further reading is recommended
- Related areas include...
  - Development process
  - Object-oriented modelling
  - Programming style



Summary

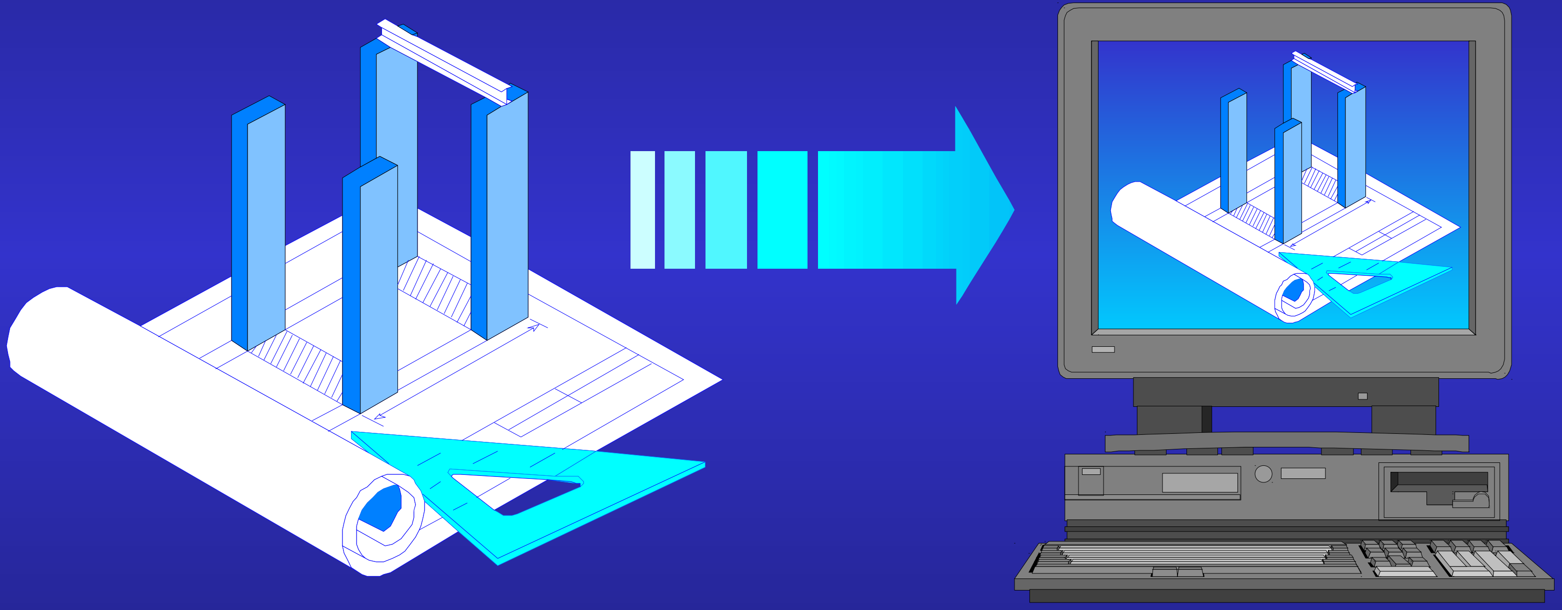
- Patterns provide a design vocabulary
  - Recurring and reusable architectural ideas
- They distil successful design experience
  - Solution structure is sensitive to details of purpose and context
- No pattern is an island
  - Patterns may relate to one another in intent, structure or sequence

Further Reading



Pattern-Based Software Development

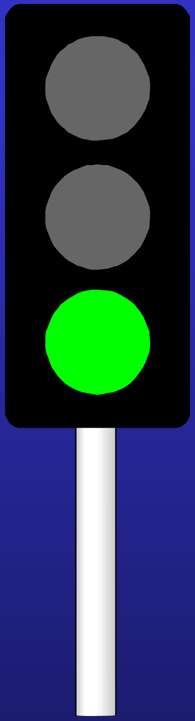
Unit Testing with JUnit



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

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

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

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

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

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));
    }
    ...
}
```

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

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) ...
    ...
}
```

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

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

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) ...
    ...
}
```

fail

- *fail* is, in effect, an unconditional assertion
  - There is no condition, it always fails and will do so by throwing an *AssertionFailedError*
  - Useful for marking paths in tests that should be unreachable if the tested code is correct

```
public class Assert
{
    ...
    public static void fail() ...
    public static void fail(String message) ...
    ...
}
```

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)
{
}
```

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() ...
    ...
}
```

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

Defining Test Cases in JUnit 4

```
import org.junit.Test;
import static org.junit.Assert.*;

public class RecentlyUsedListTest
{
    @Test public void initialListIsEmpty()
    {
        RecentlyUsedList list = new RecentlyUsedList();

        assertTrue(list.isEmpty());
        assertEquals(0, list.size());
    }
    @Test public void singleItemInsertionIntoEmptyList()
    {
        RecentlyUsedList list = new RecentlyUsedList();
        list.add("Bristol");

        assertFalse(list.isEmpty());
        assertEquals(1, list.size());
        assertEquals("Bristol", list.get(0));
    }
    ...
}
```

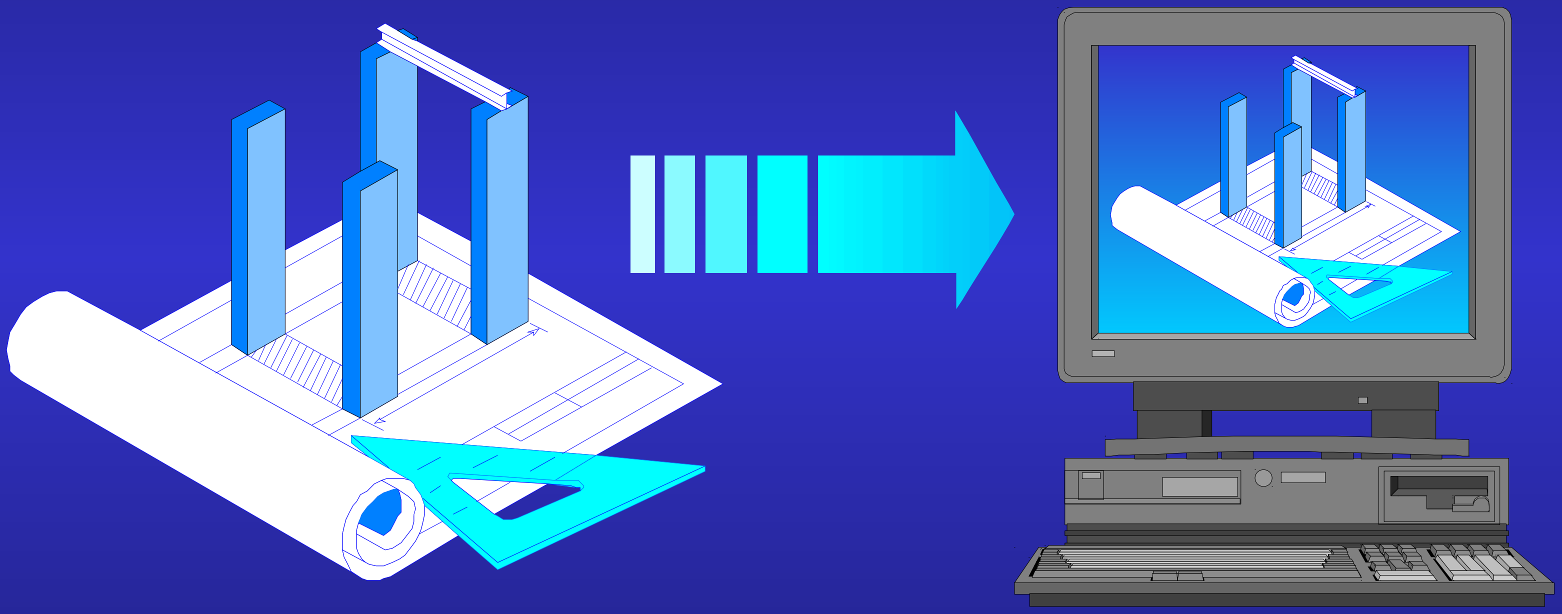
Assertions in JUnit 4

- 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.**

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

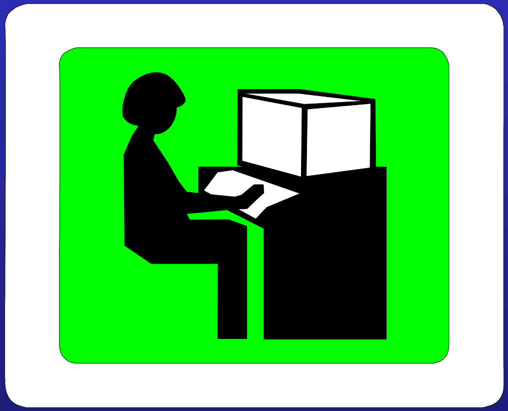
Workshop



Pattern-Based Software Development

Workshop

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

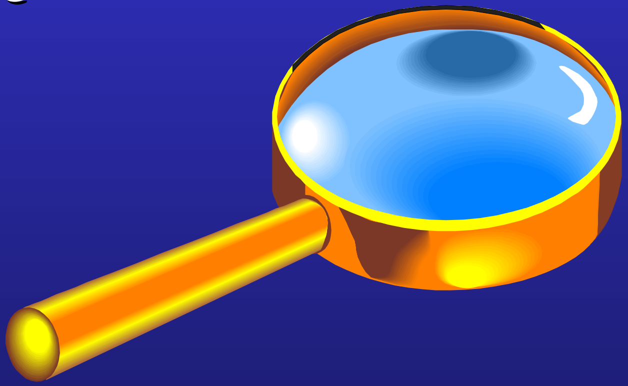


Workshop Guidelines

- 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

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



Exercise: Note Organiser

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