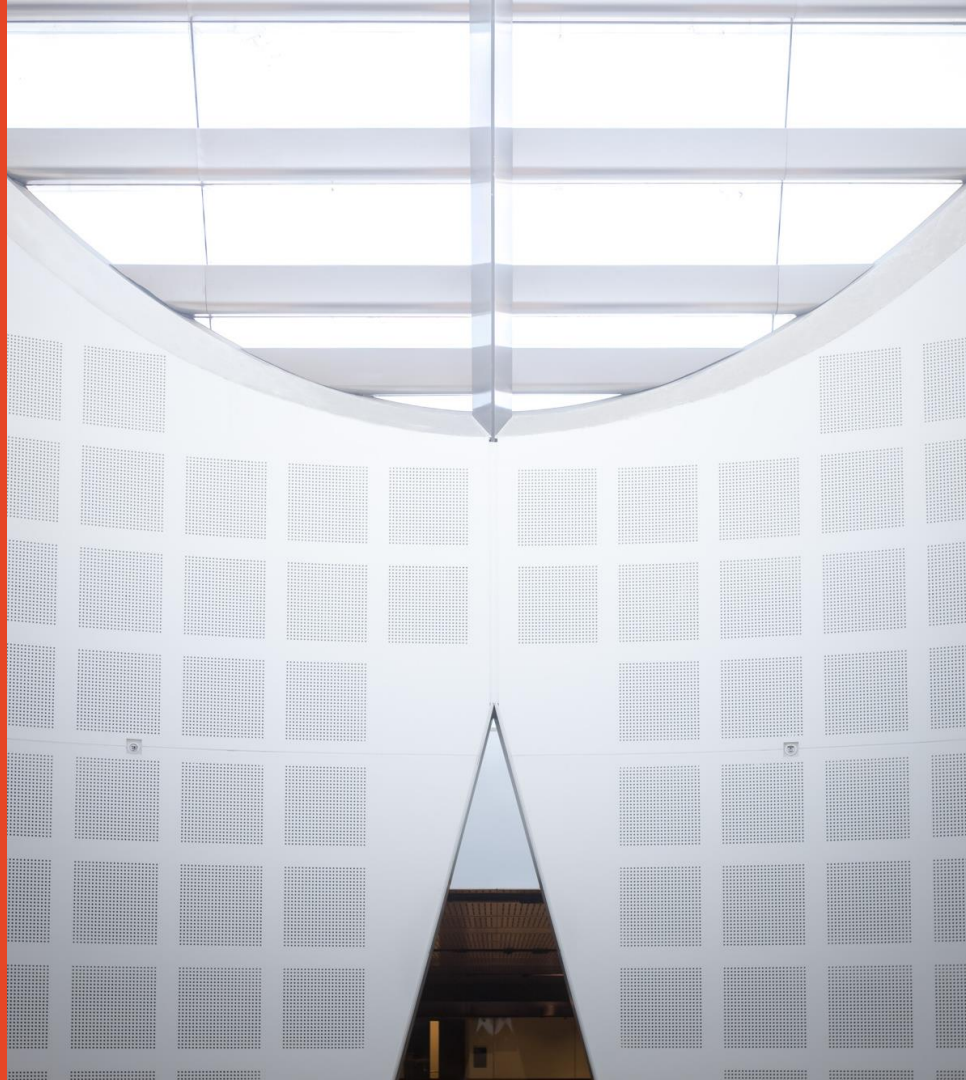
Software Design and Construction 2
SOFT3202 / COMP9202
Advanced Design Patterns (GoF)

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY



Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

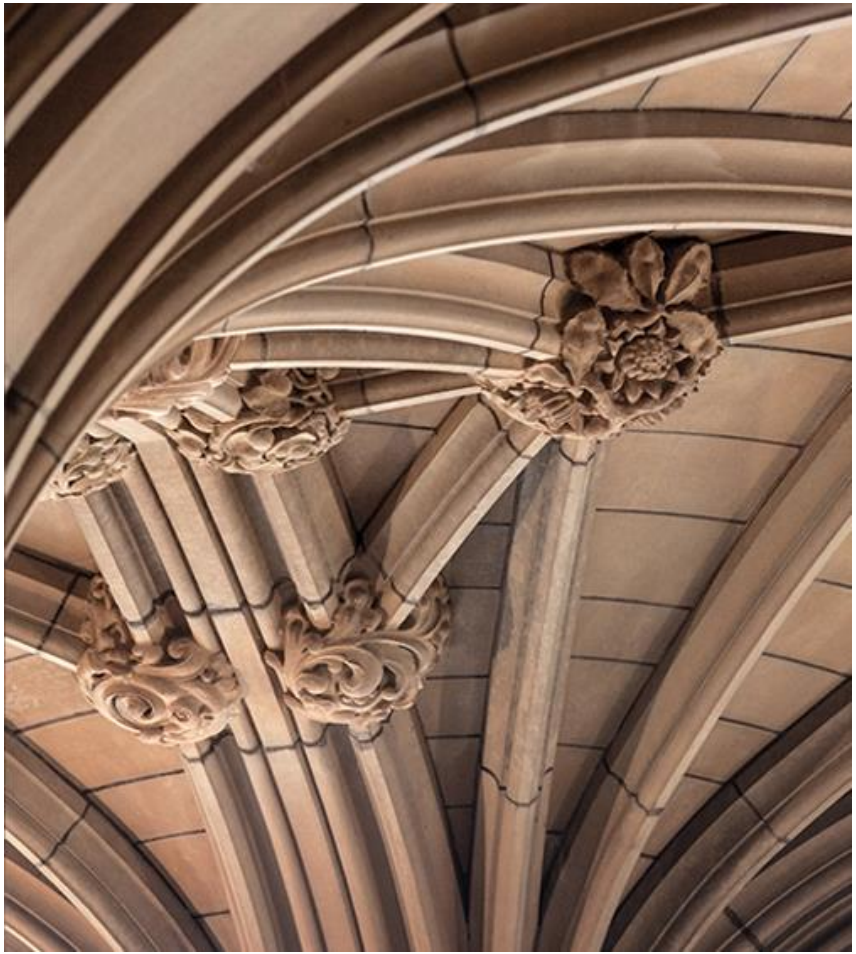
Do not remove this notice.

Agenda

- GoF Design Patterns
  - Flyweight
  - Bridge
  - Chain of Responsibility

Flyweight Design Pattern

Object Structural



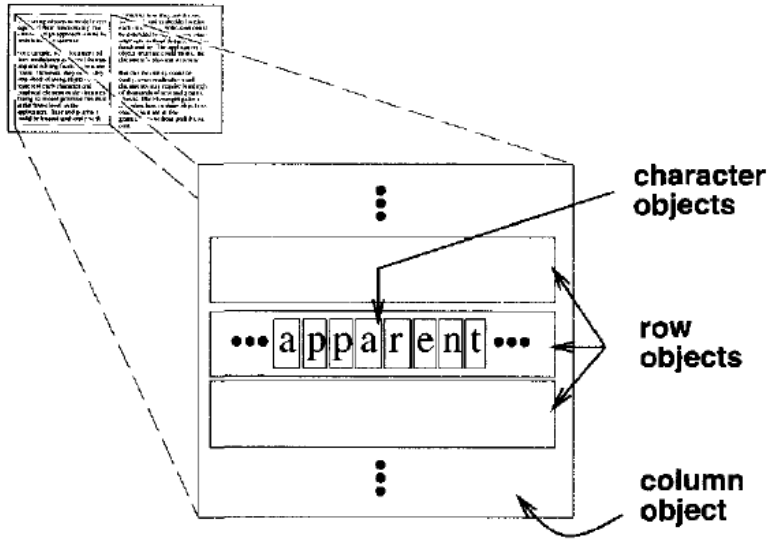
THE UNIVERSITY OF SYDNEY

Motivation – Text Editor Application

- Design of document editor system

- Use objects to represents tables/figures
- Use object for each character

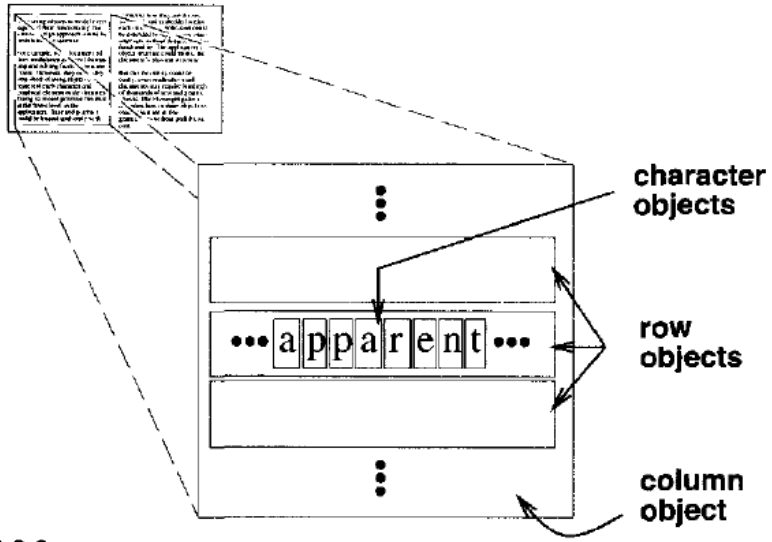
- Discuss benefits and drawbacks ?

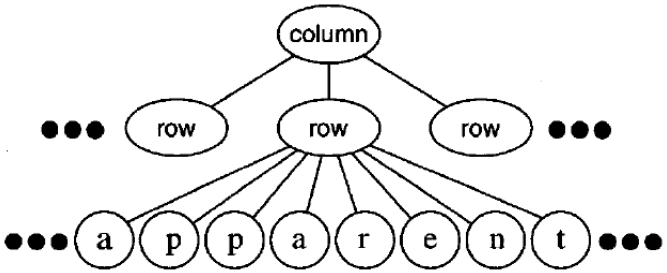


Flyweight – Motivation

- Flexibility at fine granular level - uniform formatting and processing
- Support new character set without impacting other functions (extensibility)

- Very high cost (memory)





Motivation – Text Editor Application

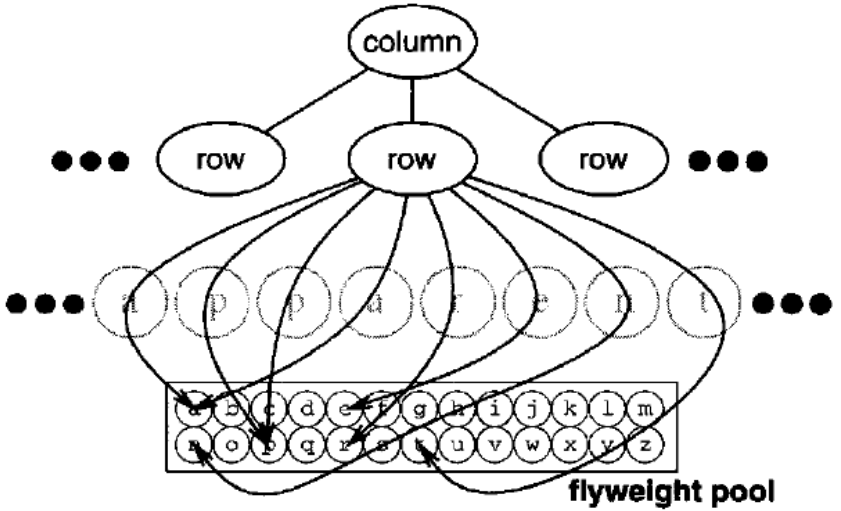
- An object for each letter of the alphabet
- Shared object than can be used in multiple contexts simultaneously
- What about object's state?
  - Character code
  - Character position (coordinates)
  - Typographic style
- Given the sharing aspect, how the above states should be stored?

Motivation – Text Editor Application

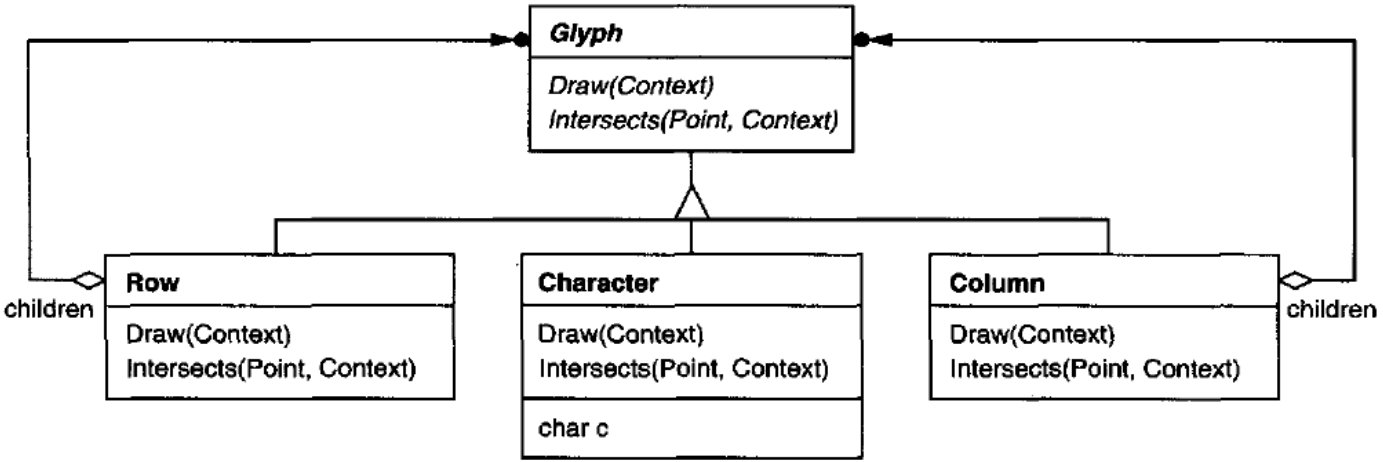
- An object for each letter of the alphabet
- Shared object than can be used in multiple contexts simultaneously
- What about object's state?
  - Character code (intrinsic/shared)
  - Character position (extrinsic/not shared)
  - Typographic style (extrinsic/not shared)
- Given the sharing aspect, how the above states should be stored?
- Intrinsic state: shared and thus stored in the shared object
- Extrinsic state: cannot be shared as it depends on the context (client's responsibility)

Text Editor Application – Flyweight Objects

- One shared flyweight object per character which can appear in different contexts in the document structure



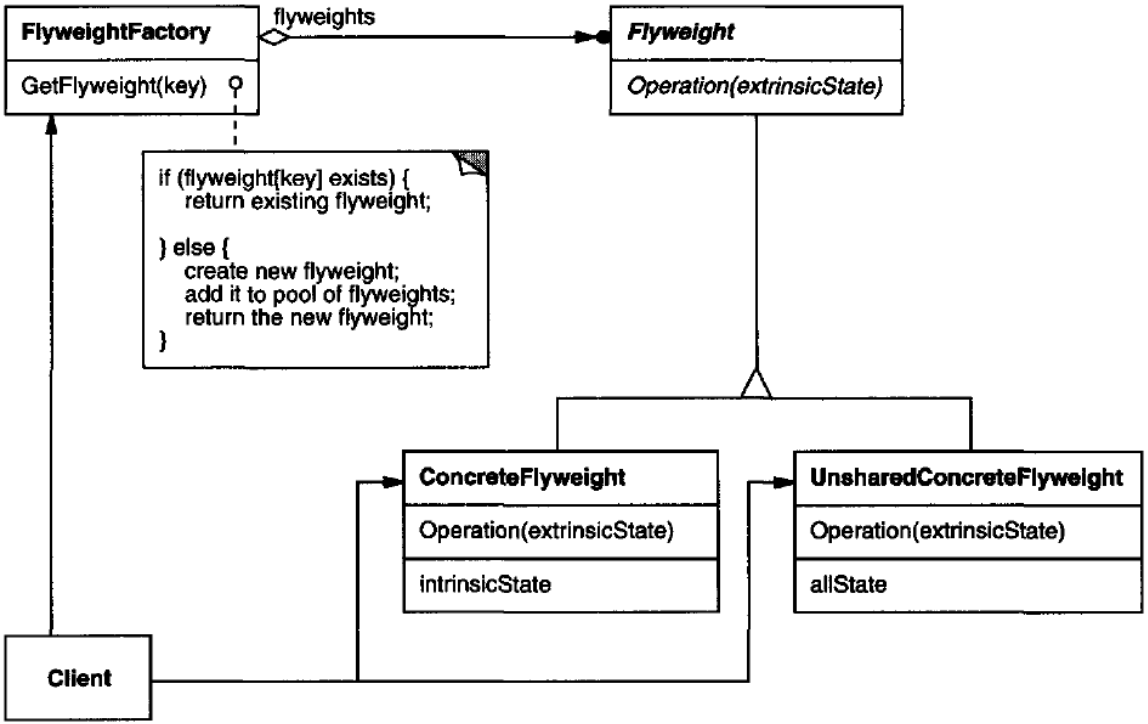
Text Editor Application – Flyweight Design



Flyweight Pattern

- Object structural
- Intent:
  - Use sharing to support large numbers of fine-grained objects efficiently
- Applicability:
  - Large number of objects are used
  - Storage costs are high
  - Most object state can be made extrinsic
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed
  - The application does not depend on object identity

Flyweight – Structure



Flyweight – Participants & Collaboration

- Flyweight (Glyph)
  - Interface for extrinsic state
- ConcereteFlyweight (Charcterer)
  - Implements Flyweight interface adding intrinsic state
- UnsharedConcereteFlyweight (Row, Column)
  - Make some concrete flyweight subclasses unshared
- FlyweightFactory
  - Creates and manages flyweight objects and ensure proper sharing
- Client
  - Maintains a reference to flyweights

Flyweight Consequences

- Benefits
  - Efficiency: save memory at run-time (sharing objects, intrinsic state)
  - Consistency: centralized objects' state

- Drawback
  - Un-time costs to transfer find and/or compute extrinsic stae
  - All objects are controlled identically

Flyweight – Implementation

- Extrinsic state and efficient storage
  - If there are as many different kinds of extrinsic state as there are objects before sharing, then removing it from shared objects won't reduce storage costs

- Managing shared objects
  - FlyweightFactory objects often use an associative store to let clients look up flyweight of interests
  - Sharing implies reference counting or garbage collection to reclaim a flyweight's storage when it's no longer needed, especially when number of flyweights is large

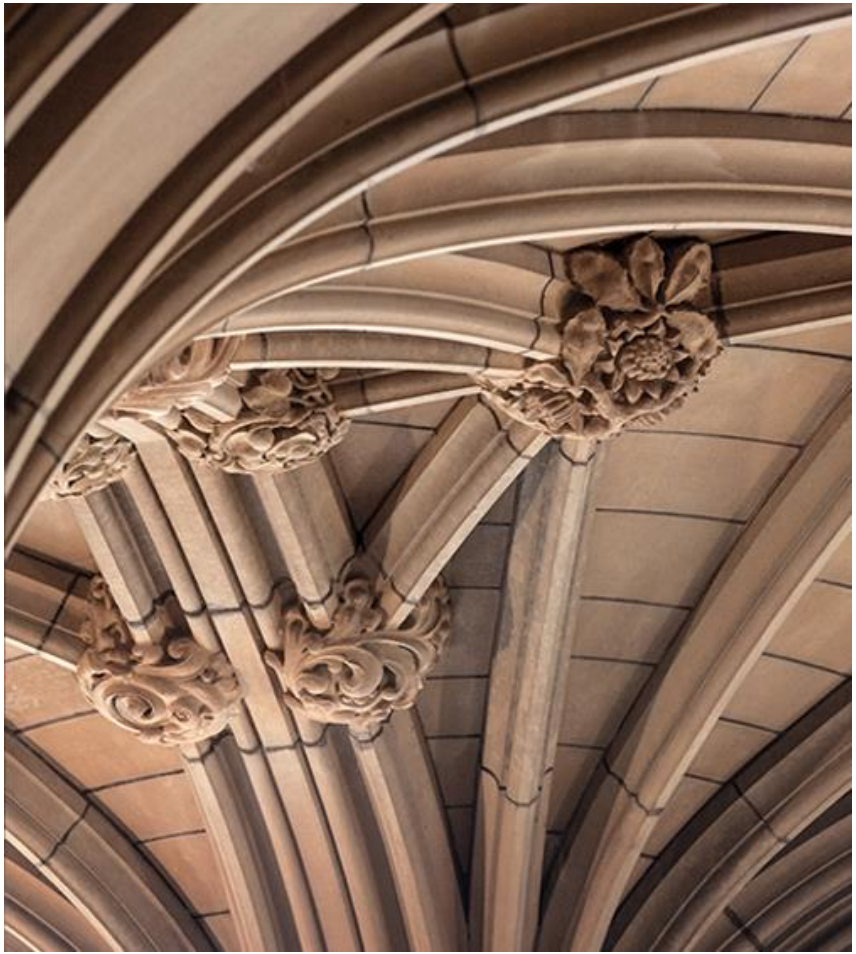
Flyweight – Related Patterns

- Composite
  - Flyweight often combined with the *composite* pattern to implement a hierarchical structure as a graph with shared nodes
    - Leaf nodes cannot store a pointer to their parent (passed)

- State and Strategy Patterns
  - Flyweight often implement *state* and *strategy* objects as flyweights

Bridge

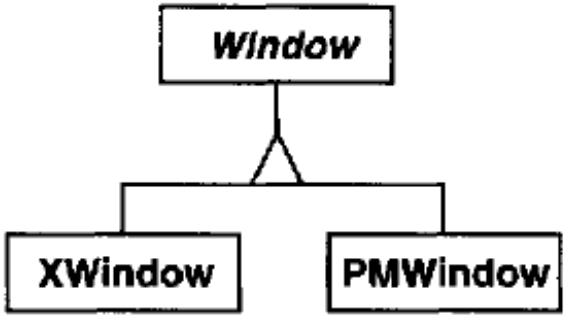
Object Structural



THE UNIVERSITY OF SYDNEY

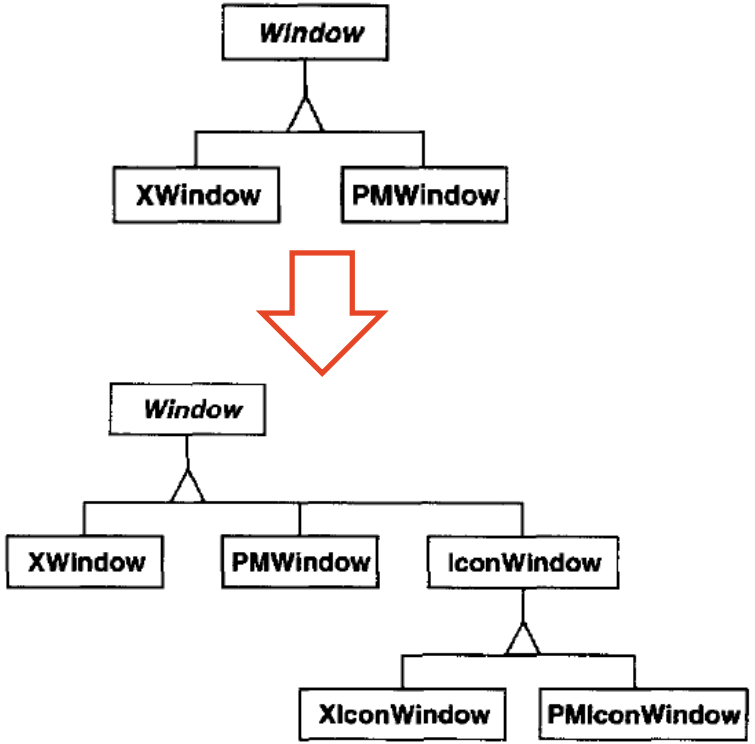
Motivating Scenario

- Portable window abstraction in a user interface toolkit
- Abstraction to allow writing applications that work in different platforms (e.g., Windows, IBM)
- Design using inheritance (right diagram)
- Good/bad design? Why/Why not?

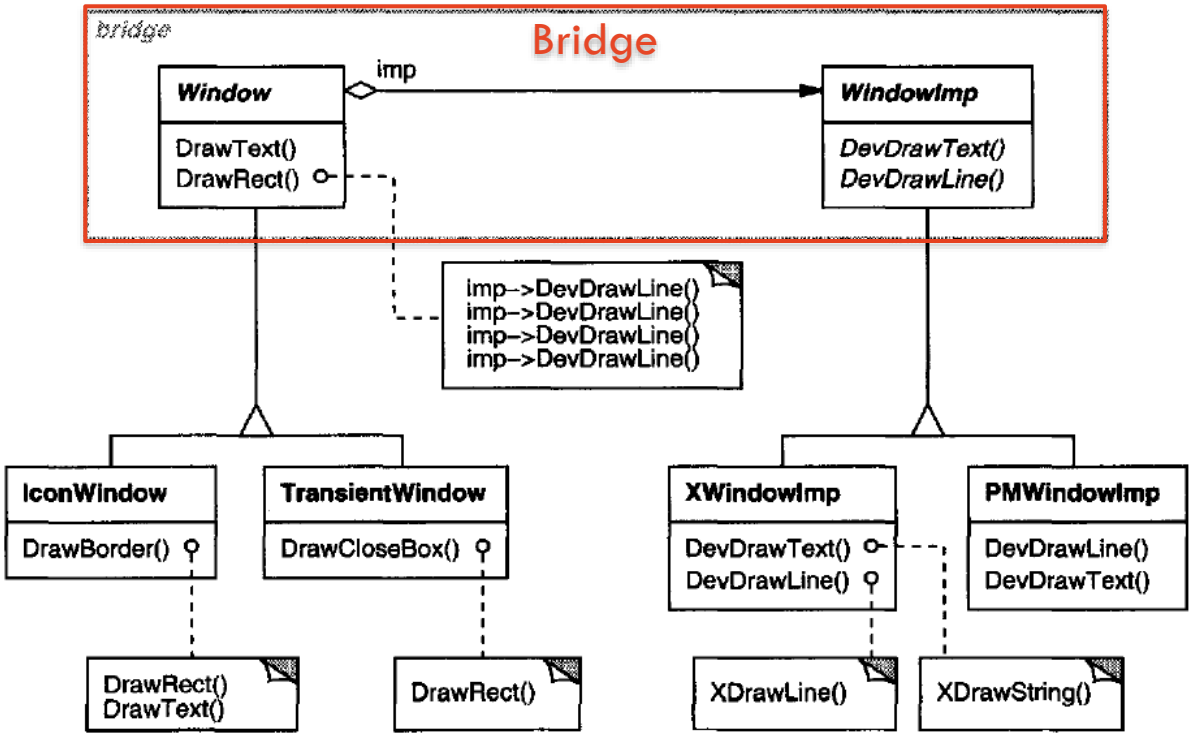


Motivating Scenario – Design with Inheritance

- Extend window abstraction to cover different implementation, BUT:
- Implement many classes in the hierarchy
- Strong binding between abstraction and binding(client code is platform-dependent)



Better Design – using the Bridge

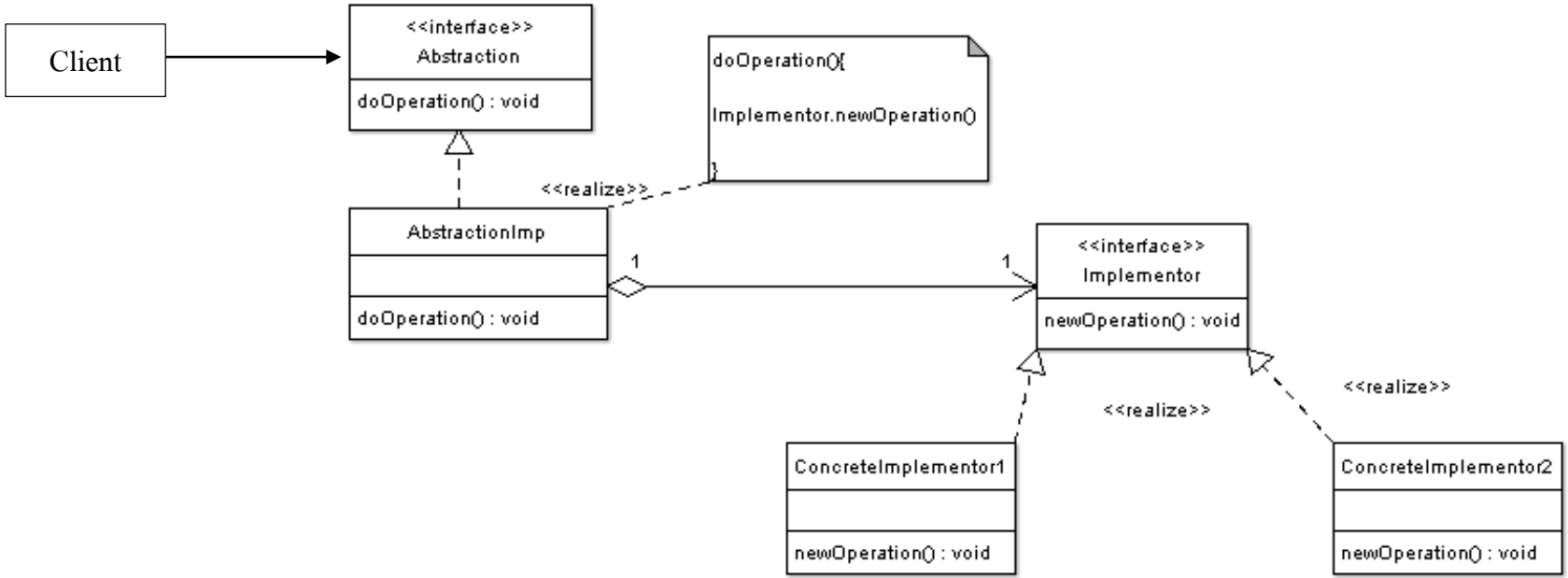


Bridge Pattern (Handle or Body)

- Avoid permanent binding between an abstraction and its implementation
- Abstractions and their implementations should be extensible
- Changes in an abstraction's implementation should not impact its client
- Large number of classes involved
  - Split into two class hierarchies ("nested generalization")
- Share an implementation on multiple objects and make the client unaware of it

Bridge Pattern – Participants and Collaboration

–



Bridge Pattern – Participants and Collaboration

- Abstraction (Window)
  - Defines the abstraction's interface and maintains a reference to an object of type Implementor
- RefinedAbstraction (IconWindow)
  - Extends the interface defined by Abstraction
- Implementor (WindowImp)
  - Defines the interface for implementation classes.
  - The Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives
- ConcreteImplementor (XWindowImp, PMWindowImp)
  - Implements the Implement or interface and defines its concrete implementation.
- Client
  - Abstraction forwards client requests to its Implementor object.

Bridge Pattern – Consequences

- Decoupling interface and implementation
  - Implementation can be configured at run-time
  - Reduce compile-time dependencies on implementation
  - Better structured design
- Improve extensibility
  - Abstraction and implementor can be extended independently
- Hiding implementation details from clients
- Increased complexity!
  - Two hierarchies to grow and to manage

Bridge Pattern – Implementation

- One implementor
  - Abstract implementor class isn't necessary if there's only one implementation
  - It's still useful when a change in the implementation of a class must not affect its existing clients

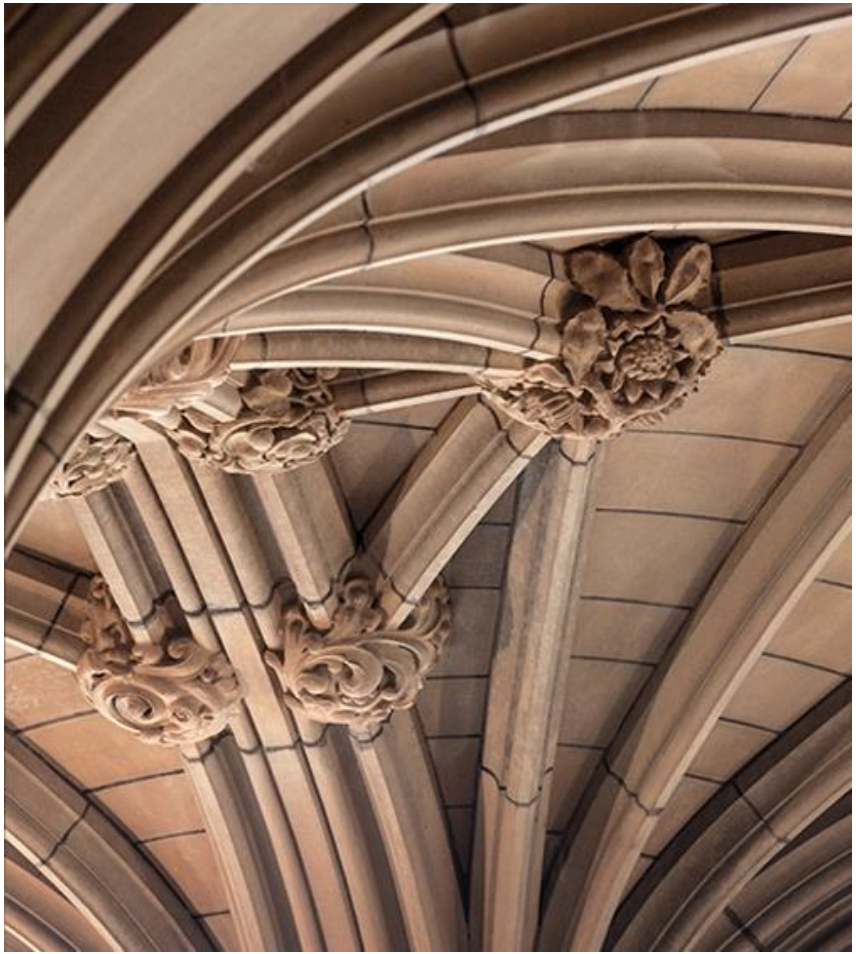
- Creating the right implementor object when there is more than one
  - Abstraction's constructor if it knows about all ConcreteImplementor classes
  - A collection class supports multiple implementations, decide by the collection's size
    - Use linked list for a small collection
    - Use a hash table for a large collection
  - Default implementation which can be changed according to usage

Bridge Pattern – Related Patterns

- Abstract Factory
  - Can create and configure particular Bridge
- Adapter
  - Aims at making un-related classes work together (after design consideration)
  - Bridge focuses on making abstraction and implementations vary independently (during design)

Chain of Responsibility (CoR)

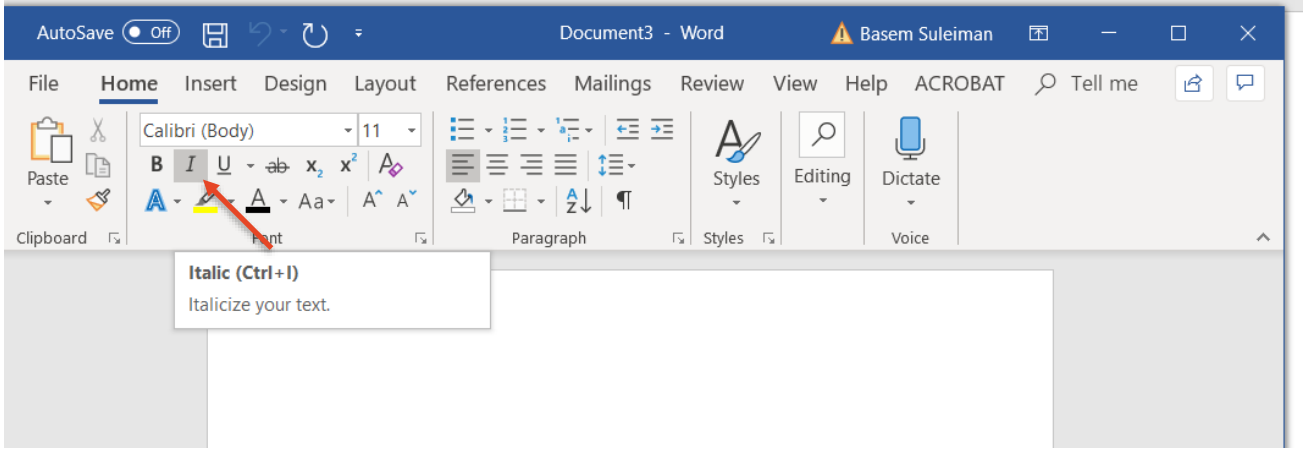
Object Behavioural



THE UNIVERSITY OF SYDNEY

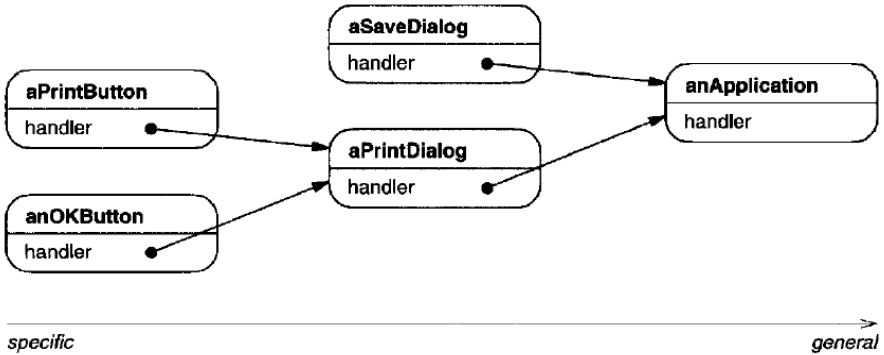
Motivating Scenario – GUI with Help

- GUI with a help facility where a user gets help information by clicking on it
- Help information dependent on the interface's context (context-sensitive)
  - Button in dialog box vs. button in a window
  - Display general help info. About the immediate context in case no specific help exits



GUI with Help – Potential Design

- Organize help info. according from the most specific to the most general
- Several UI objects, one per help request
- **Discuss the prose/cons of this design.**



GUI with Help – Potential Design

- It helps to serve different types of help requests
- However, the object that ultimately provides the help isn't known explicitly to the object that initiates the help request (strong coupling)
- So, we need a way to decouple the object that initiates the help request from those that might provide the help information

Better Design – Chain of Responsibility (CoR)

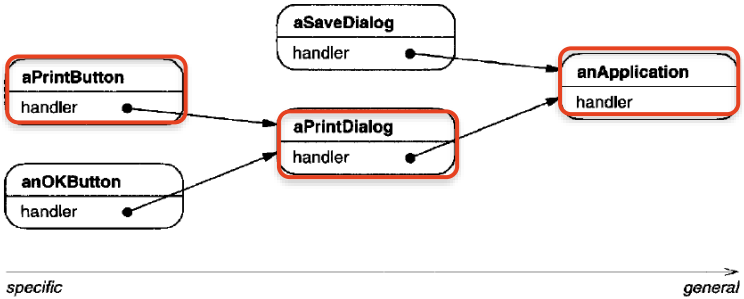
- Provide multiple objects a chance to handle a request
  - Pass the request along a chain of objects until one handles it

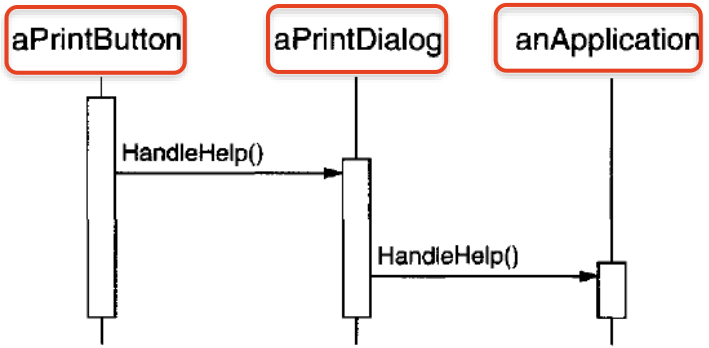
- First object receives the request either handles it or forward it to the next candidate on the chain, and so on so forth

- The request has an ***implicit receiver*** as the requester object has no explicit knowledge of the handler object

Better Design – Chain of Responsibility (CoR)

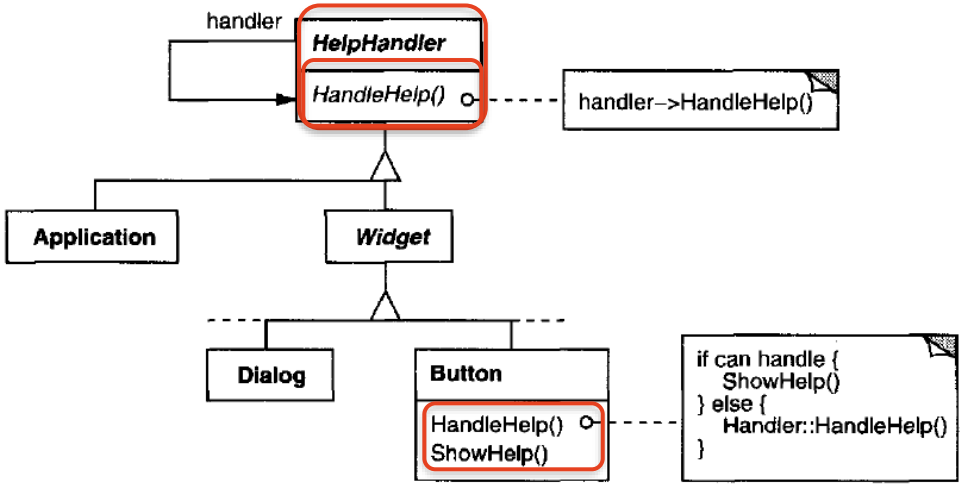
- Solution details
  - User clicks the "*Print*" button's help (contained in *PrintDialog* instance)
  - PrintDialog knows the object it belongs to
  - The client (request issuer) has no direct reference to the object that ultimately realizes it





Better Design – Chain of Responsibility (CoR)

- How to ensure implicit receiver?
  - Each object shares *common interface* for handling requests and accessing its successors on the chain
- Classes that want to handle help requests can make HelpHandlera parent
- HelpHandler's HandleHelp forwards the request to the success by default
- Subclasses can override this operation to provide help under the right conditions



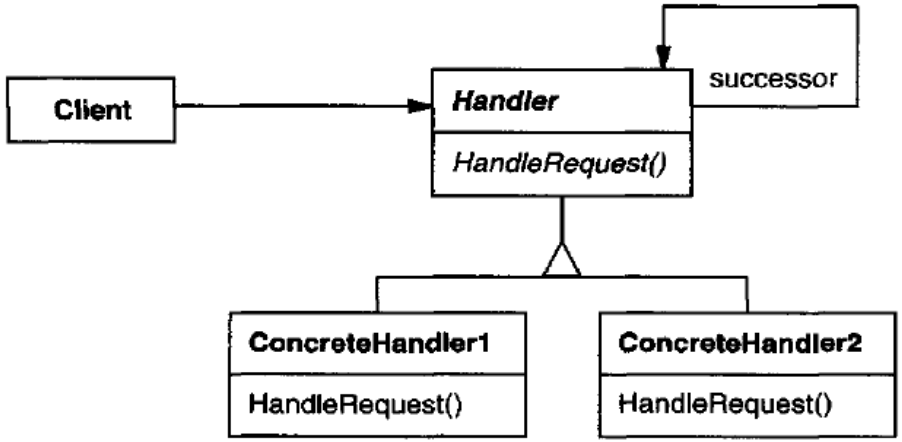
Chain of Responsibility Pattern

- Intent
  - Avoid coupling the sender of a request to its receiver
  - It allows more than one object a chance to handle the request
  - Chain the receiving objects and pass the request along the chain until an object handles it

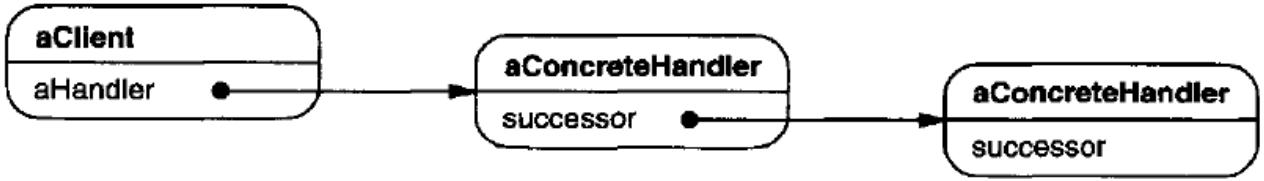
- Use
  - More than one object may handle a request, and the handler should be ascertained dynamically
  - Hide the receiver (explicitly) when a request should be issued to one of several objects
  - The handling behavior should be specified dynamically

CoR Pattern – Structure

Class structure



Typical object structure



CoR Pattern – Participants and Collaboration

- Handler (HelpHandler)
  - Defines interface for handling requests
  - May implement the successor line
- ConcreteHandler (PrintButton, PrintDialog)
  - Handles requests it is responsible for
  - forwards the request to its successor if it cannot handle it
- Client
  - Initiates a request which will be propagated along the chain until a ConcereteHandler takes responsibility for handling it

CoR Pattern – Consequences

- Reduced Coupling
  - Objects in the chain does not have explicit knowledge about each other
- Flexibility in distributing responsibilities among objects
  - Can add/change responsibilities at run-time
- Requests could be unhandled
  - There's no guarantee that a request could be handled

CoR Pattern – Implementation (1)

- Declaring child management operations: which classes declare the child management operations (add, remove) in the composite class hierarchy:
  - Define child management interface at the class hierarchy's root
    - Allows treating all components uniformly (transparent)
    - Clients may add/remove objects from leaves (not safe)
  - Child management in the composite class
    - Add/remove objects at the compile-time in statically typed languages (safe)
    - Leaves and composites have different interfaces (not transparent
  - Transparency over safety

CoR Pattern – Implementation (2)

- Child ordering
  - Ordering on the children of composite is important in some designs
  - Composites represents parse trees then compound statements can be instances of a composite whose children must be ordered to reflect the program
  - Design child access and management interfaces carefully to manage the sequence (use iterator pattern)

CoR Pattern – Related Patterns

- Composite
  - In CoR pattern, a component's parent can act as its successor and hence the use of Composite pattern

References

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

- OO Design, Online: [https://www.oodesign.com/bridge-pattern.html]

W7 Tutorial: Practical Exercises/coding
W7 Lecture: Enterprise Design Patterns
Testing Assignment A2

THE UNIVERSITY OF SYDNEY