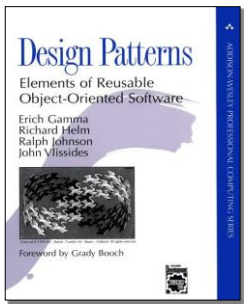
Lecture 5
Design Patterns

Dr Peter T. Popov
Centre for Software Reliability

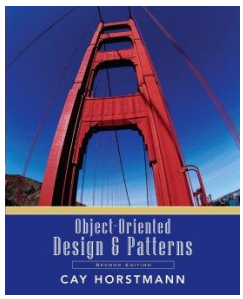
25th October 2018

Design Patterns



Based largely on two books:

1. Erich, Gamma, Richard Helm, Ralph Johnson and John Vlissides, "Design Patters: Elements of Reusable Object-Oriented Software". (known as the "Gang of Four", GoF)
2. Cay Horstmann, "Object-Oriented Design & Patterns", 2nd Edition.



2

OO Design is a form of Engineering

Engineers
- Do not work from the first principle
- They reuse ***proven designs***
- There are standards that engineers are expected to follow

Even playwrights rarely design their plots from scratch
- They use 'patterns', like 'Tragically Flawed Hero' (Macbeth, Hamlet, etc.).

Why should a Software Engineer (OO designer) be different?

3

Motivation & Concept

UML emphasizes ***notations***
- Fine for specification, documentation

Object oriented Design is more than just drawing diagrams
- Design patterns improve product quality
- Design patterns improve maintenance
  - Even well designed systems over time become difficult to maintain
- Design patterns improve documentation

Good OO designers rely on lots of ***experience***
- At least as important as syntax
- Expert OO designers are experts because they are ***aware of good designs***
- Good designs solve specific design problems by making designs:
  - More flexible
  - Elegant (elegance is important quality!)
  - Reusable
- Experienced designers use design patterns to get the 'right' design ***more quickly***.

4

A *Design Pattern…*

"Each pattern describes a ***problem*** which occurs over and over again in our environment, and then describes the ***core of the solution*** to that problem, in such a way that you can use this solution a ***million times over***, without ever doing it the same way twice".

Christopher Alexander

This definition applies to software, too.

"Patterns are descriptions of communicating objects and classes that are customised to solve general design problem in a specific context."

The "Gang of Four"

5

A *Design Pattern…*

- abstracts a recurring *design structure*
- comprises classes and/or objects
  - dependencies
  - structures
  - interactions
  - conventions
- names & specifies the design structure explicitly
- distills design experience

6

Antipatterns…

- Designs that are so bad that should be avoided at all cost
  http://sourcemaking.com/antipatterns/software-development-antipatterns
- Examples
  - Blob – a class that gobbled up many different responsibilities
  - Poltergeist - spurious class whose objects are short-lived and do carry no significant responsibilities

7

Design patterns: Four Basic Parts

1. Name
2. Problem (including "forces")
3. Solution
4. Consequences & trade-offs of application

Language - implementation-independent

- There are design language specific and domain specific (concurrency, GUI, database connectivity, etc.) patterns

A "micro-architecture", but ***not ready solutions***.

- Must be adapted to a specific context.

8

Goals with Design Patterns

Codify good design
- distil & generalise experience
- aid to novices & experts alike

Give design structures ***explicit names***
- common vocabulary
- reduced complexity
- greater expressiveness

Capture & preserve design information
- articulate design decisions succinctly
- improve documentation

Facilitate restructuring/refactoring
- patterns are interrelated
- additional flexibility, promote ***loose coupling***

9

Benefits of Patterns

- *Design* reuse
- Uniform design vocabulary
- Enhance understanding, restructuring, & team communication
- Basis for automation
  - Visual Paradigm supports design patterns
    - Offers a number of tutorials about how to use Design Patterns
    - One can document one's own pattern and "store" it for future reuse
- Abstracts away from many unimportant details
  - Patterns are ***not*** ready solutions.
  - Have to be adapted to a specific context.

10

Design Patterns Classification

- Design Patterns vary in their ***granularity*** and ***level of abstraction***
- 2 - criteria classification (proposed in the "Gang of Four" book):
  - Purpose:
    - ***Creational*** – concerned with the process of object creation
    - ***Structural*** - deal with the composition of classes or objects
    - ***Behavioural*** – characterise the ways in which classes or objects interact and distribute responsibilities
  - Scope – if the pattern applies primarily to classes or objects.
    - Class patterns deal with relationships between classes and their sub-classes. Relationships are established through inheritance, i.e. they are static – fixed ***at compile time***
    - Object patterns deal with object relationships, which can be changed at ***run time*** and are more dynamic.

11

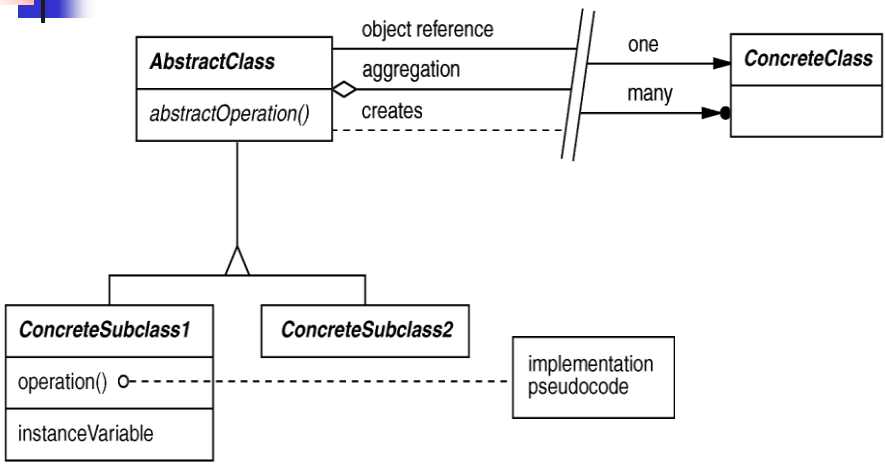
Design Patterns Classification

		Purpose		
		Creational	**Structural**	**Behavioural**
Scope	**Class**	Factory Method	***Adapter (class)***	Interpreter Template Method
	Object	***Abstract Factory*** Builder Prototype Singleton	***Adapter (object)*** ***Bridge*** ***Composite*** ***Decorator*** ***Facade*** Flyweight ***Proxy***	***Chain of Responsibility*** ***Command*** ***Iterator*** ***Mediator*** Memento ***Observer*** State ***Strategy*** ***Visitor***

The patterns shown in bold italic are covered in the lecture.

12

Modified UML/OMT Notation



This slide is useful in case you read the "Gang of Four" book. The notations used in the book differ from those adopted by UML 2.X.

13

OBSERVER object behavioral

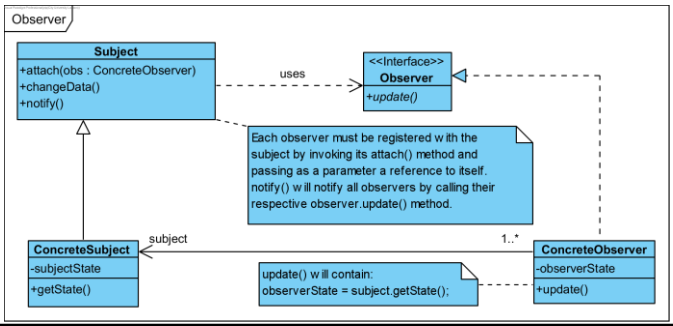
Intent

Defines a one-to-many dependency between objects so that when one object changes state, all dependents are notified & updated. Known as publish-subscribe.

Applicability

- an abstraction that has two aspects, one dependent on the other
- a change to one object requires changing untold others
- an object should notify unknown other objects of the change

Structure

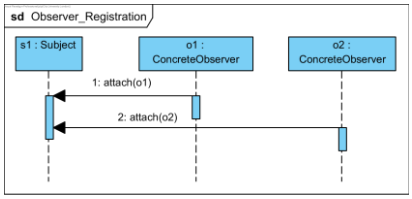


14

OBSERVER (2) Collaboration

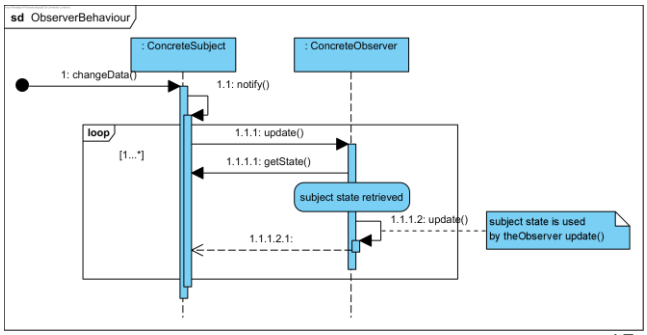
Registration

Each observer must register with the subject.



Notification

All registered observers are notified of state change and must synch with the state change.



15

OBSERVER (3) object behavioral

Consequences

+ modularity: subject & observers may vary independently
+ extensibility: can define & add any number of observers
+ Support for broadcast communication: no need to specify a receiver of the notification
− unexpected updates: observers don't know about each other. If an observer changes the subject this would trigger a cascade of updates to observers.

Implementation

- subject-observer mapping *has to be implemented somehow*
- Observing more than one subject. The subject may pass itself as a parameter of notify().
- Dangling references, e.g. deleting a subject: the deleted subject may notify its observers .
- Avoid observer specific update protocols: the push (details about the change are pushed by the subject) & pull (the observer asks for details, as in the sequence diagram) models
- registering interest in modifications explicitly (as this may trigger unexpected updates)

Known Uses

- Smalltalk Model-View-Controller (MVC)
- InterViews (Subjects & Views, Observer/Observable)
- Pub/sub middleware (e.g., CORBA Notification Service, Java Messaging Service, OMG DDS)
- Java notify() – awaking the sleeping threats.

16

COMPOSITE object structural

Intent

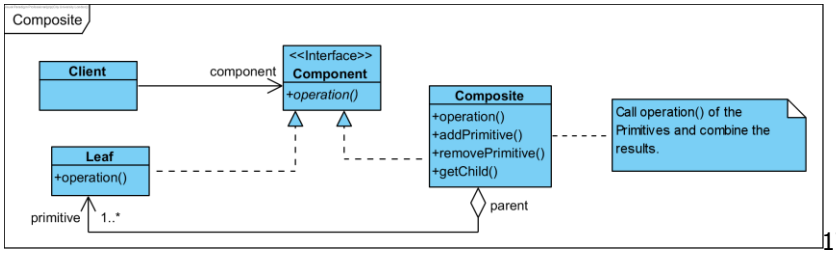
Composes an object into tree structure to represent whole – part. Clients treat individual objects and compositions of objects ***uniformly***.

Applicability

objects must be composed recursively,

and clients are able to ignore the difference between individual & composed elements.

Structure



17

COMPOSITE (2) object structural

Consequences

+ uniformity: treat components the same regardless of complexity
+ extensibility: new Component subclasses work wherever old ones do
– overhead: might need prohibitive numbers of objects

Implementation

- explicit parent references from children to their parent can simplify the traversal and management of composite structure
- sharing components. Component with more than one parent creates problems.
- responsibility for deleting children. This is an issue if the class hierarchy already offer operations for dealing with this.

Known Uses

- Directory structures on UNIX & Windows
- Naming Contexts in CORBA
- MIME types in SOAP

18

STRATEGY object behavioral

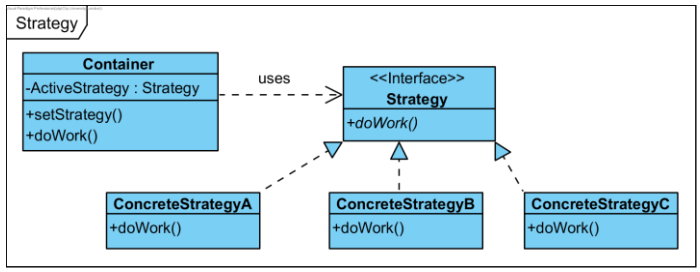
Intent

Define a family of algorithms, encapsulate each one, & make them interchangeable to let clients & algorithms vary independently

Applicability

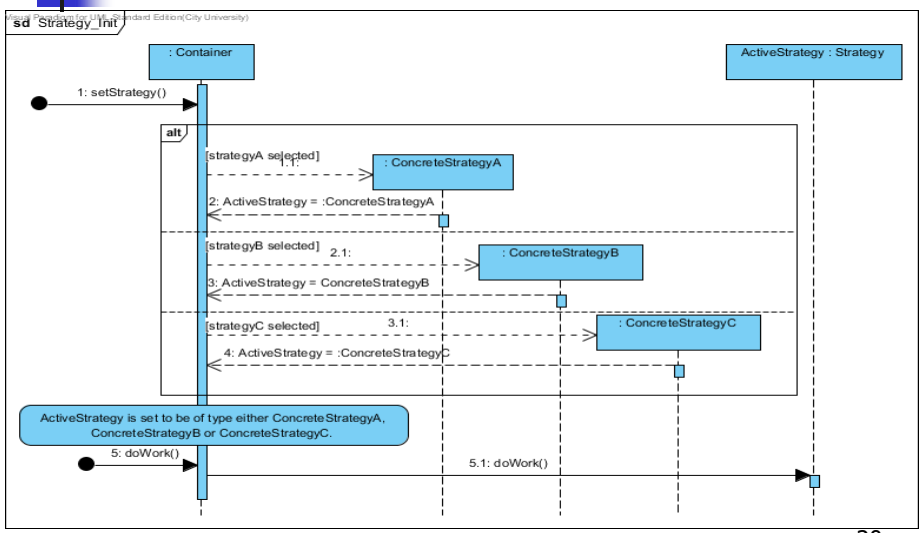
when an object should be configurable with *one of many algorithms*, *and* all algorithms can be encapsulated, *and* one interface covers all encapsulations

Structure



19

STRATEGY (2): Collaboration



20

STRATEGY (3) object behavioral

Consequences

+ greater flexibility, reuse
+ can change algorithms *dynamically*
− strategy creation & communication overhead (may be significant)
− inflexible Strategy interface
− semantic incompatibility of multiple strategies used together

Implementation

- exchanging information between a Strategy & its context
- static strategy selection via parameterized types

Known Uses

- InterViews text formatting
- RTL register allocation & scheduling strategies
- The ACE ORB (TAO) Real-time CORBA middleware

See Also

- Bridge pattern (object structural)

21

DECORATOR object structural

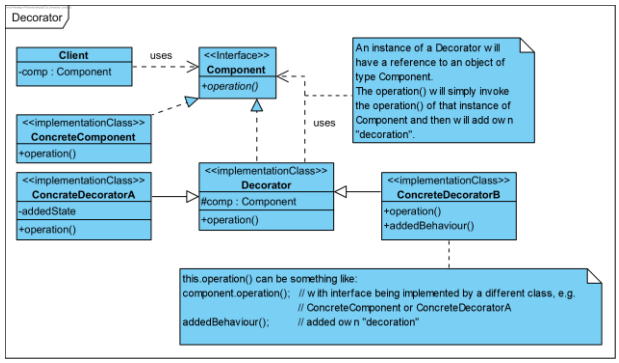
Intent

Transparently augments objects with *new responsibilities dynamically*

Applicability

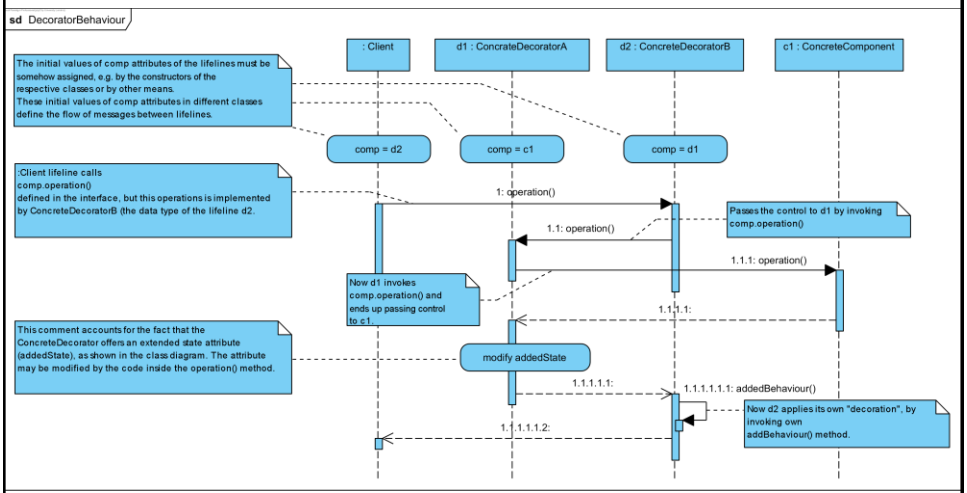
- when extension by sub-classing is impractical
- for responsibilities that can be added & withdrawn dynamically

Structure



22

DECORATOR (2): Collaboration



23

DECORATOR (3) object structural

Consequences
- \+ responsibilities can be added/removed at run-time (by assigning the value of #comp attribute)
- \+ avoids subclass explosion
- \+ recursive nesting allows multiple responsibilities
- – identity crisis – decorated component is ***not identical to the component itself***.
- – composition of decorators is hard if there are side-effects.

Implementation
- interface conformance
- use a lightweight, abstract base class for Decorator
- heavyweight base classes make Strategy more attractive

Known Uses
- Java I/O classes (streams)

24

ABSTRACT FACTORY object creational

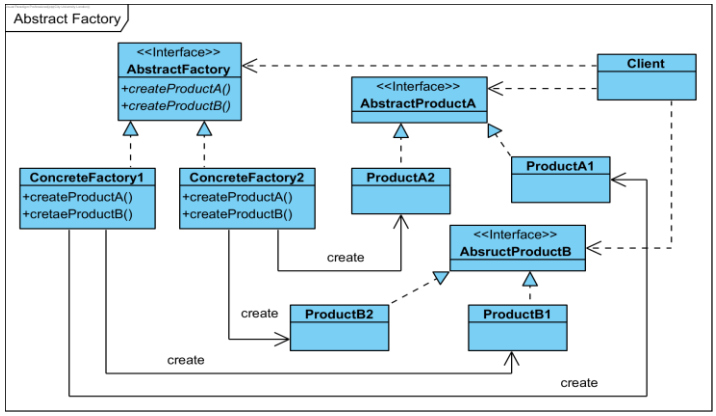
Intent

create families of related objects without specifying subclass names

Applicability

when clients cannot anticipate groups of classes to be instantiated

Structure



25

ABSTRACT FACTORY (2) object creational

Consequences

+ flexibility: removes type (i.e., subclass) dependencies from clients
+ abstraction & semantic checking: hides product's composition
– hard to extend factory interface to create new products

Implementation

- factories as singletons. An application really needs only one instance of a concrete factory. So usually best implemented as a Singleton.
- parameterization as a way of controlling interface size
- configuration with Prototypes, i.e., determines who creates the factories
- abstract factories are essentially groups of factory methods

Known Uses

- AWT Toolkit
- The ACE ORB (TAO)

26

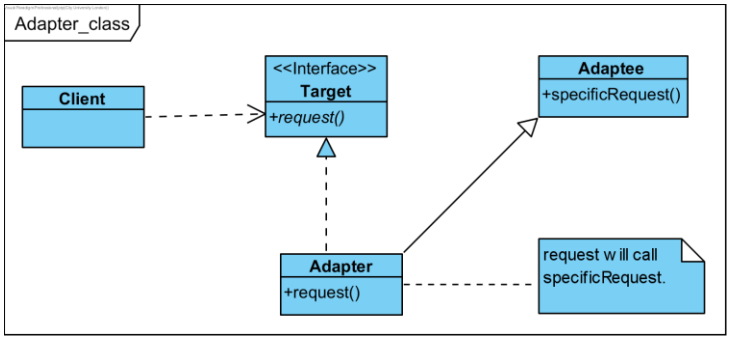
ADAPTER class/object structural

Intent

Convert the interface of a class into another interface the client expects.
Concept similar to Power Plug Adapters.

Also known as ***Wrapper.***

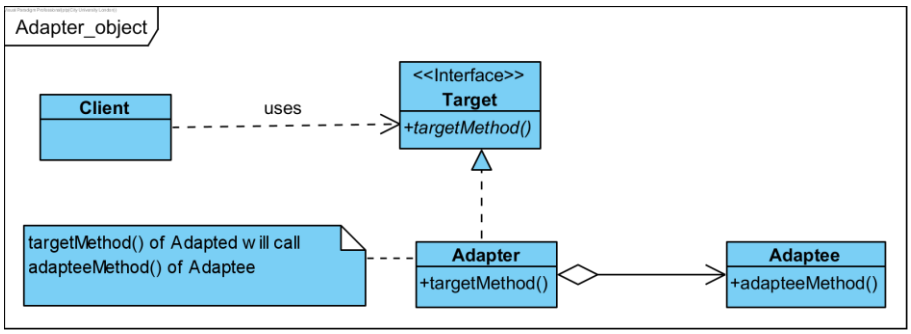
Structure (class Adapter)



27

ADAPTER (2) class/object structural

Structure (object Adapter)



28

ADAPTER (3) class/object structural

Consequences (class Adapter)

+ Introduces only one object, and no additional indirection (reference) is needed to get to the Adaptee.
– Adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence the ***adapter won't work with sub-classes of Adaptee***.

Consequences (object Adapter)

+ A single adapter can work with Adaptees and all its sub-classes. The Adapter can add functionality to all Adaptees at once.
– *Overriding* Adaptee behaviour will require sub-classing Adaptee and making Adapter *refer* to the sub-class rather than the Adaptee itself.

Known Uses

- Java stream library
- ET++
- Pluggable Adapters are common for ObjectWorks/Smalltalk
- Bertrand Mayer's "Marriage of Convenience" is a form of class Adapter.

29

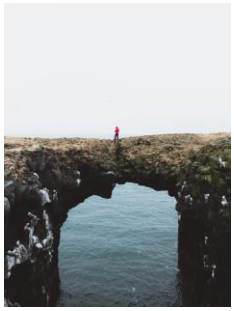
BRIDGE object structural

Intent

decouples an abstraction (interface) from its (physical) implementation(s) so that the two can vary independently.

Applicability

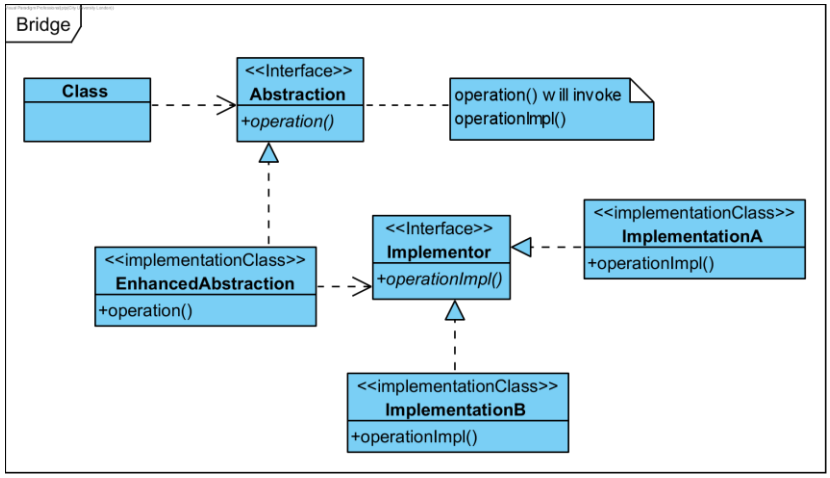
- when interface & implementation should vary ***independently***
- require a uniform interface to interchangeable class hierarchies
- Implementation may also vary dynamically (i.e. at run time)



30

BRIDGE (2) object structural

Structure



31

BRIDGE (3) object structural

Consequences

- \+ abstraction interface & implementation are independent
- \+ implementations can vary dynamically
- – one-size-fits-all Abstraction & Implementor interfaces

Implementation

- sharing Implementors & reference counting
- creating the right Implementor (often use factories)

Known Uses

- libg++ Set/{LinkedList, HashTable}
- AWT Component/ComponentPeer

32

FAÇADE object structural

Intent

Provide a unified interface to a set of interfaces in a subsystem.

Façade defines a higher-level interface that makes the subsystem easier to use.

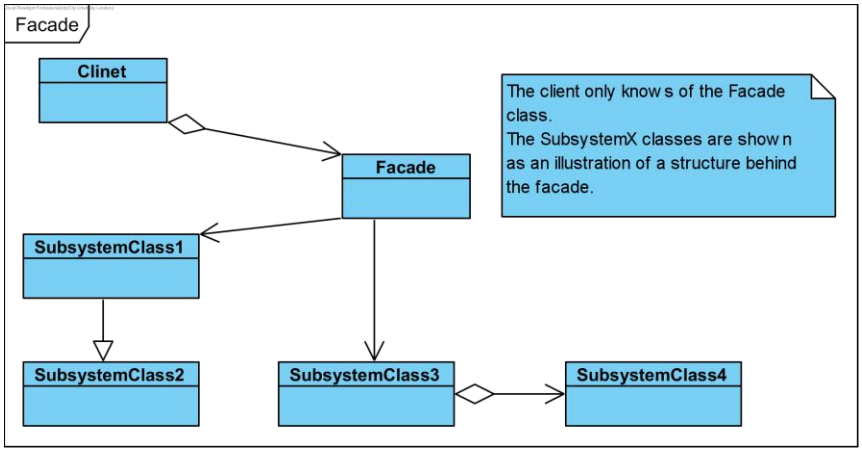
Applicability

- when you want to provide a simple interface to a complex subsystem. A Façade can provide a simple of the system that is ***good enough*** for most clients *default view*
- There are many dependencies between the clients and the implementation classes of an abstraction. Introduce Façade to *decouple the clients from the subsystem*
- Use Façade to define an entry point to each subsystem (layer your design)

33

FAÇADE (2) object structural

Structure



34

FAÇADE (3) object structural

Consequences

+ Shields clients from subsystem components
+ Promotes weak coupling between clients and subsystems
+ It does not prevent clients from using subsystem component if they want to

Implementation

- Reduce client – subsystem coupling. Further reduction of the coupling can be achieved by making the Façade an abstract class or an interface (see my example of DB connectivity in lecture 2)
- Private vs. private subsystem classes. Façade may have private and public methods.

Known Uses

- ET++ application framework

35

PROXY object structural

Intent

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

Applicability

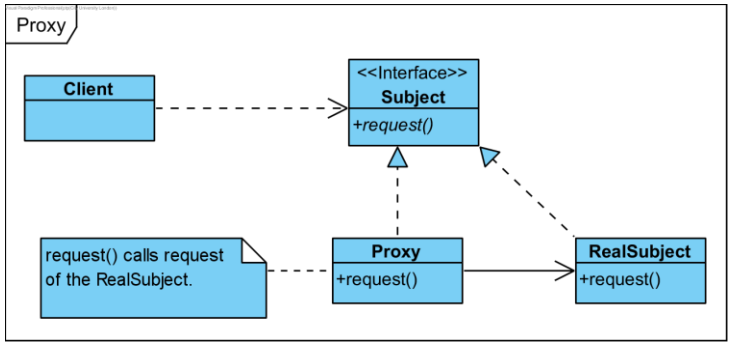
- Remote proxy
- Virtual proxy
- Protection proxy

Also known as ***Surrogate***

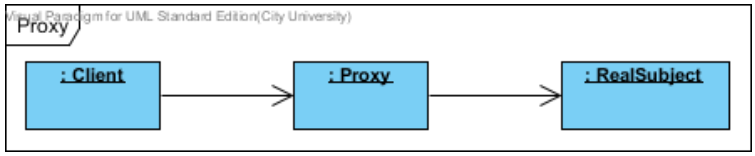
36

PROXY (2) object structural

Structure



Object Diagram



37

PROXY (3) object structural

Consequences

+ Additional indirection has many advantages (remote proxy – the object resides in a ***different address space***, virtual proxy – object is created on demand)

Implementation

- C++, Java

Known Uses

- Java RMI
- Windows COM+, DCOM, etc.

38

CHAIN OF RESPONSIBILITY object behavioral

Intent

Avoid coupling the sender of a request to its receiver by giving 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

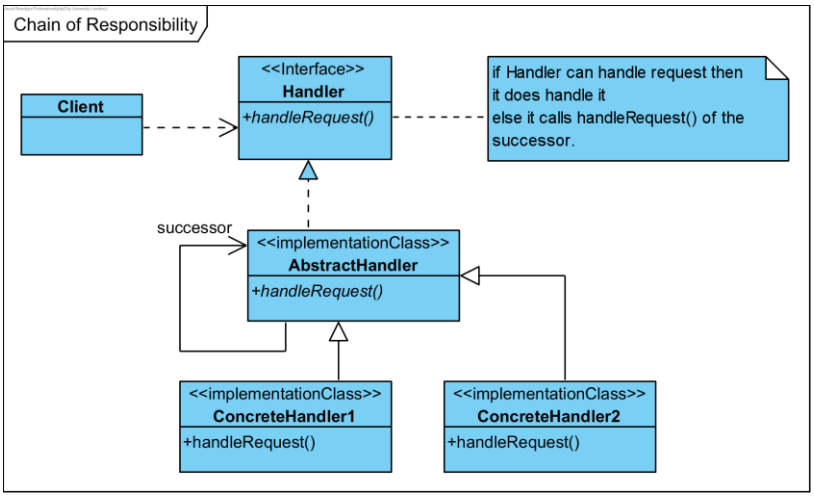
Applicability

- when more than one object can handle a request and the handler is ***not known a priori***
- A request is issued to one of several objects without specifying the receiver explicitly
- The set of objects that can handle a request should be specified dynamically (i.e. at run time)

39

CHAIN OF RESPONSIBILITY(2) object behavioral

Structure



40

CHAIN OF RESPONSIBILITY (3) object behavioral

Consequences

+ Reduced coupling – freeing an object from having to know which object will handle the request
+ Added flexibility in assigning responsibilities to objects
– ***Receipt is not guaranteed*** since the request has no explicit receiver

Implementation

- Implementation of the successor chain: either define new links or use existing links
- Connecting successors – if no particular order exist, this will have to be implemented somehow.
- Representing requests. In the simplest form this is a mere method invocation, but may take other forms, too.

41

MEDIATOR object behavioral

Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes ***loose coupling*** by keeping the objects from referring to each other explicitly.

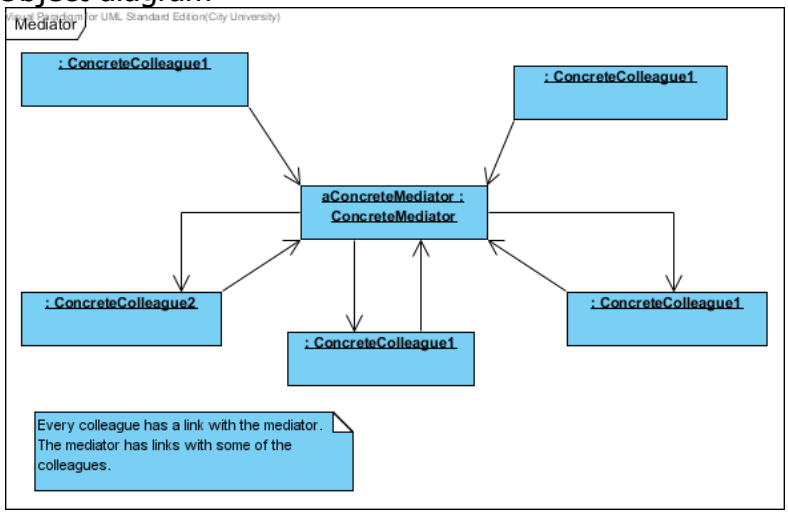
Applicability

- When a set of objects communicate in well defined but complex way
- Reusing an object is difficult because it refers and communicates with many other objects
- A behaviour that is distributed between several classes should be customisable without a lot of sub-classing.

42

MEDIATOR (2) object behavioral

Object diagram



43

MEDIATOR (3) object behavioral

Consequences

+ It limits sub-classing as typically only the mediator need to be sub-classed.
+ Decouples colleagues: it ***promotes loose coupling*** between them
+ Simplifies object protocols by replacing many to many interactions with one to many
− Centralises control. Trades complexity of interaction for complexity of mediator.

Implementation

- There is no need to define an abstract mediator class (interface) when colleagues work with only one mediator.
- Colleague – Mediator communication must be implemented. One option is to implement the Mediator as an Observer

44

VISITOR object behavioral

Intent

Represent an operation to be performed on the elements of an object structure. Visitors allows one to define a new operation without changing the classes of the elements on which it operates

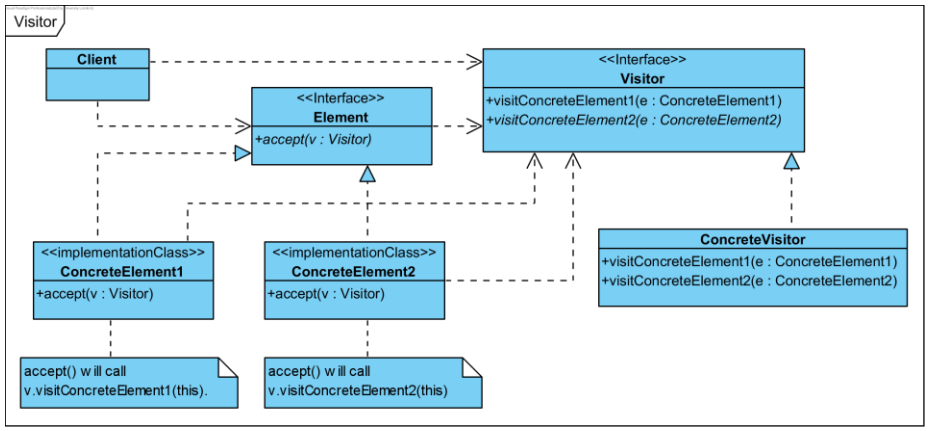
Applicability

- when classes define many unrelated operations
- class relationships of objects in the structure rarely change, but the operations on them change often
- algorithms keep state that's updated during traversal

45

VISITOR (2) object behavioral

Structure



46

VISITOR (3) object behavioral

Consequences

- \+ flexibility: visitor & object structure are independent
- \+ localized functionality
- – circular dependency between Visitor & Element interfaces
- – Visitor is brittle to new ConcreteElement classes

Implementation

- double dispatch: the operations is defined by the Element visited and by the Visitor (hence double dispatch)
- general interface to elements of object structure

Known Uses

- ProgramNodeEnumerator in Smalltalk-80 compiler
- IRIS Inventor (3-D graphics) scene rendering
- TAO IDL compiler to handle different backends

47

Interactive systems

The next group of patterns are related to Interactive Systems

Consider the following aspects:

Goals:

- support execution of user operations
- support ***unlimited-level undo/redo***

Constraints/forces:

- scattered operation implementations
- must store ***undo state***
- not all operations are undoable

48

Solution: Encapsulate Each Request

A **Command** encapsulates

- an operation (`execute()`)
- an inverse operation (`unexecute()`)
- a operation for testing reversibility (`boolean reversible()`)
- state for (un)doing the operation

Command may

- implement the operations itself, *or*
- delegate them to other object(s)

49

COMMAND object behavioral

Intent

encapsulate the request for a service as an object, thus allowing for clients to be parameterised with different requests, queue or log requests and support undoable operations. The pattern involves 4 terms:

- Command:
- Receiver
- Invoker
- Client

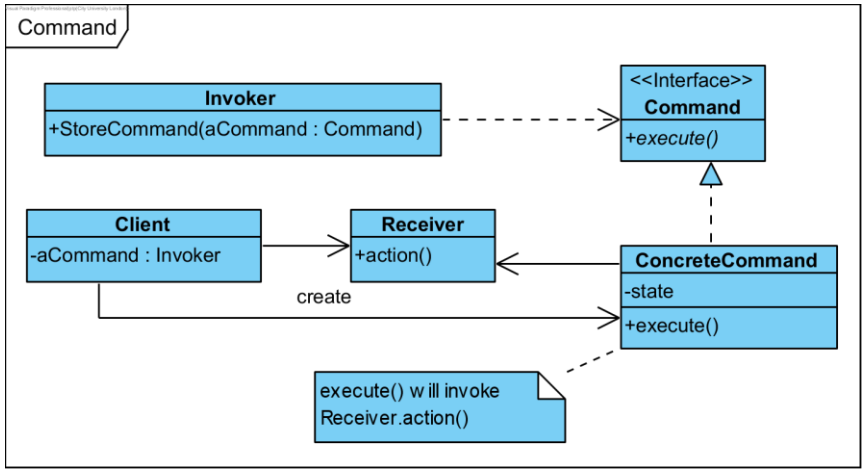
Applicability

- to parameterise objects with an action to perform
- to specify, queue, & execute requests at different times
- for multilevel undo/redo

50

COMMAND (2) object behavioral

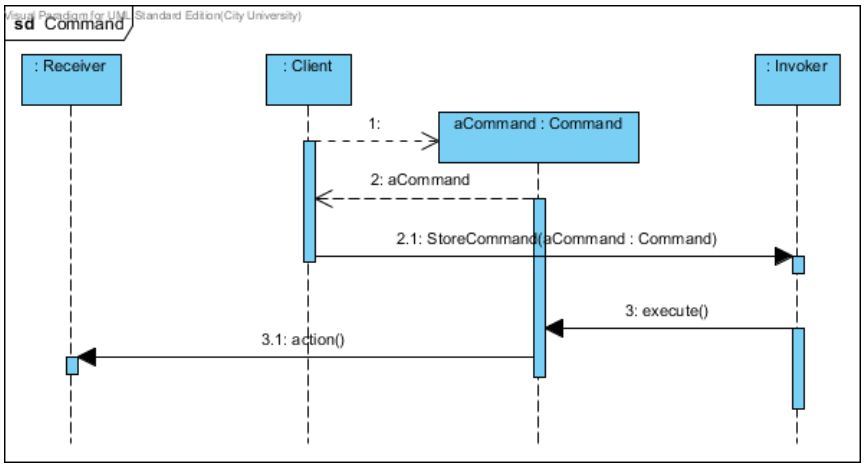
Structure



51

COMMAND (3) object behavioral

Interaction



52

COMMAND (4) object behavioral

Consequences

- \+ Decoupling between the object that invokes the operation from the one that knows how to perform it
- \+ supports arbitrary-level undo-redo
- \+ composition of commands into 'macro-commands'
- – might result in lots of trivial command subclasses

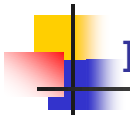
Implementation

- copying a command before putting it on a history list
- Supporting undo/redo. This may require additional state to be stored ('checkpointing')

Known Uses

- InterViews Actions
- MacApp, Unidraw Commands
- JDK's UndoableEdit, AccessibleAction
- Emacs

53



ITERATOR object behavioral

Intent

access elements of an aggregate object without exposing its underlying representation.

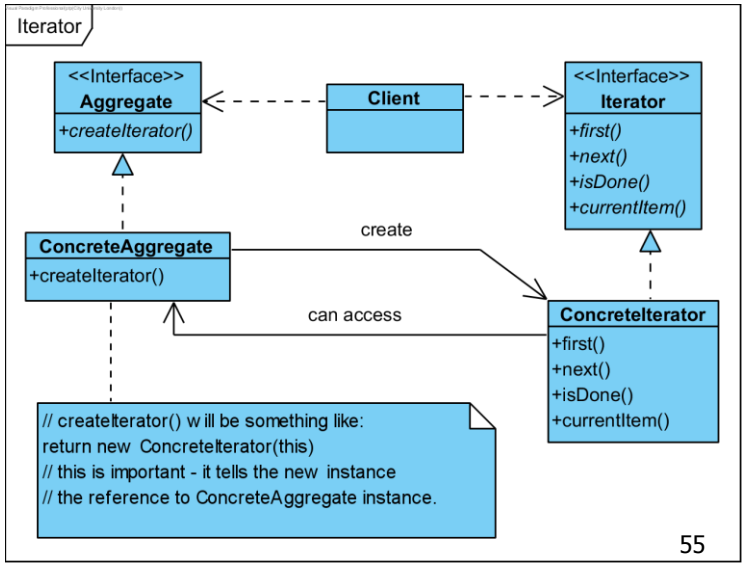
Applicability

- to access and aggregate object's contents without exposing its internal representation.
- To support multiple traversals of aggregate objects
- To provide a uniform interface for traversing different aggregate structures (that is to support polymorphic iterations)

54

ITERATOR (2) object behavioral

Structure



55

ITERATOR (3) object behavioral

Consequences

+ flexibility: aggregate & traversal are independent
+ multiple iterators & multiple traversal algorithms
– additional communication overhead between iterator & aggregate

Implementation

- internal (iterator controls) versus external iterators (client controls)
- violating the object structure's encapsulation
- robust iterators (insertions /deletions do not interfere with traversal)
- synchronization overhead in multi-threaded programs
- batching in distributed & concurrent programs

Known Uses

- C++ STL iterators
- JDK Enumeration, Iterator
- Unidraw Iterator

56

How to recognise Patterns

- Difficulties
  - Descriptions of patterns look superficially alike. E.g. COMPOSITION and DECORATOR appear almost identical.
  - As you encounter additional patterns you find it increasingly difficult to tell them apart
- Solution 1
  - Focus on ***Intent.*** Intent of COMPOSITION is to group the components into a whole, the intent of DECORATOR is to decorate a component.
- Solution 2
  - Remember where the ***pattern is put to use***. Many remember STRATEGY as the pattern of layout managers, DECORATOR - as the patterns for scroll bars.

57

Observations

Patterns are applicable in all stages of the OO lifecycle
- analysis, design, & reviews
- realization & documentation
- reuse & refactoring

Patterns permit design at a more abstract level
- treat many class/object interactions as a unit
- often beneficial *after* initial design
- targets for refactoring classes

Variation-oriented design
- consider what design aspects are ***variable***
- identify applicable pattern(s)
- vary patterns to evaluate tradeoffs
  - Tradeoffs is what makes design difficult, but also interesting!
  - Get the work done with as little code as possible vs. get the code run as fast as possible (even if wasteful) vs …
- repeat

58

But…

Don't apply them blindly

Added indirection can yield increased complexity, cost (e.g. delays)

Resist branding everything a pattern

Articulate specific benefits

Demonstrate wide applicability

Find at least *three* existing examples from code other than your own!

Pattern design even harder than the OO design!

- UML modelling tools (including Visual Paradigm) support the use of design patterns
  - Visual Paradigm offers a number of tutorials (and videos!) about how to apply design patterns in practice. Worth having a look.

59

Summary

- We looked at Design Patterns
  - How they are described
    - Name
    - Problem they solve
    - Solution
    - Consequences & trade-offs of application
  - Their classification
    - Purpose
    - Scope
- We looked at some examples of Design patterns
- We also discussed how to recognised the need for design patterns in software development projects.
- Very extensive literature exists on Design Patterns – books and on-line resources.
  - Worth looking at some of them

60