# Software Engineering
# G22.2440-001

**Session 8 – Sub-Topic 1**
**Design Patterns, Architectural Patterns**

**Dr. Jean-Claude Franchitti**

*New York University*
*Computer Science Department*
*Courant Institute of Mathematical Sciences*



# Design Patterns, Architectural Patterns

# Bibliography…

- « A System of Pattern » Bushmann et All
- « Design Patterns » Gamma et All
- « Concurrent Programming in Java » D. Lea.
- « Distributed Objects » Orfali et All
- « Applying UML and Patterns » Larman



# Patterns…

- « Patterns help you build on the collective experience of skilled software engineers. »
- « They capture existing, well-proven experience in software development and help to promote good design practice »
- « Every pattern deals with a specific, recurring problem in the design or implementation of a software system »
- « Patterns can be used to construct software architectures with specific properties… »

# Becoming a Chess Master

- First learn rules and physical requirements
  - e.g., names of pieces, legal movements, chess board geometry and orientation, etc.
- Then learn principles
  - e.g., relative value of certain pieces, strategic value of center squares, power of a threat, etc.
- However, to become a master of chess, one must study the games of other masters
  - These games contain patterns that must be understood, memorized, and applied repeatedly
- There are hundreds of these patterns



# Becoming a Software Designer Master

- First learn the rules
  - e.g., the algorithms, data structures and languages of software
- Then learn the principles
  - e.g., structured programming, modular programming, object oriented programming, generic programming, etc.
- However, to truly master software design, one must study the designs of other masters
  - These designs contain patterns must be understood, memorized, and applied repeatedly
- There are hundreds of these patterns

## Software Architecture

- A software architecture is a description of the subsystems and components of a software system and the relationships between them.
- Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system.
- The software system is an artifact. It is the result of the software design activity.



## Component

- A component is an encapsulated part of a software system. A component has an interface.
- Components serve as the building blocks for the structure of a system.
- At a programming-language level, components may be represented as modules, classes, objects or as a set of related functions.

## Subsystems

- A subsystem is a set of collaborating components performing a given task. A subsystem is considered a separate entity within a software architecture.
- It performs its designated task by interacting with other subsystems and components…



## Architectural Patterns

- An architectural Pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, their responsibilities, and includes rules and guidelines for organizing the relationships between them.

## Design patterns

- A design pattern provides a scheme for refining the subsystems or components of a software system, or the relation ships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.



## Idioms

- An Idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

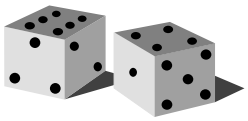
# Framework

- A framework is a partially complete software (sub-) system that is intended to be instantiated. It defines the architecture for a family of (sub-) systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made.

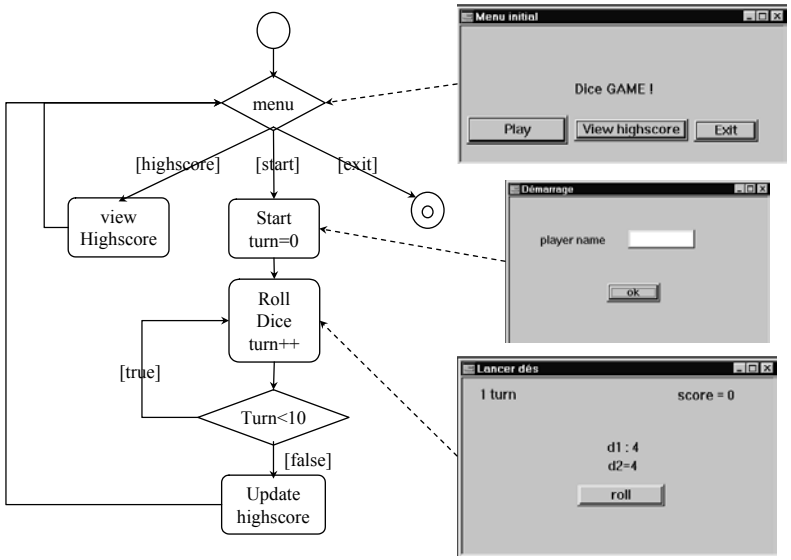


# First Example

- A Dice Game
- A Player rolls 10x 2 dices
- If result = 7, score=score + 10 points
- At the end, score of the player is registred in the highscore table.

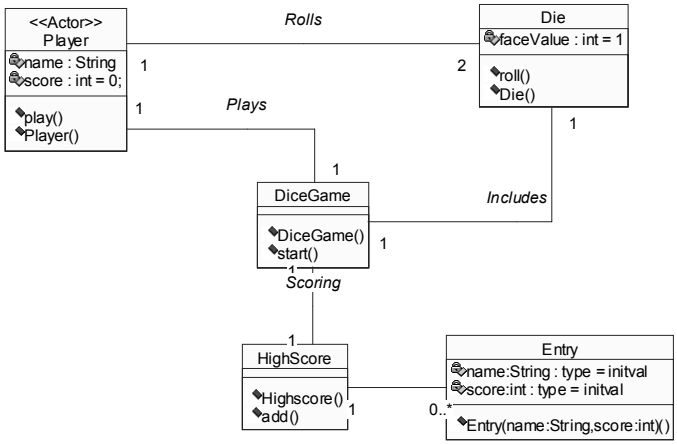


# Activity Diagram






# Analysis Diagram…



## Design Stage

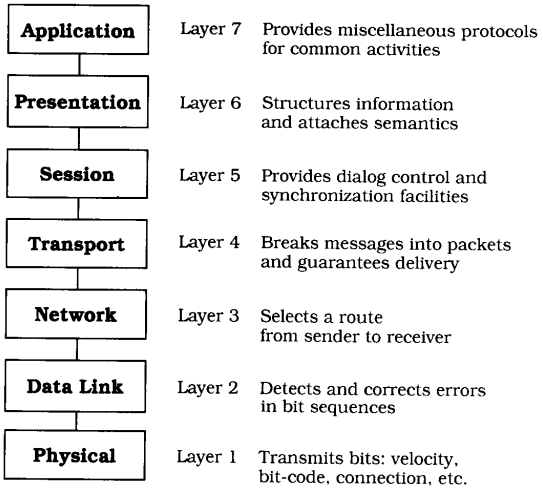
- Manage User Interface
- Manage Persistence of highscore in a file or in relational database
- Realize a layered architecture : Apply the Layer Architectural Pattern



## Layer

- Helps structure an application that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

# Layer: examples

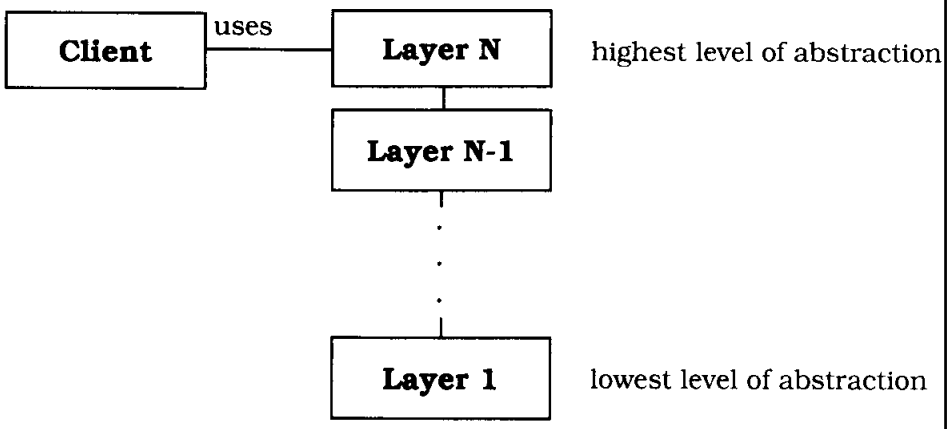





# Layer :Structure

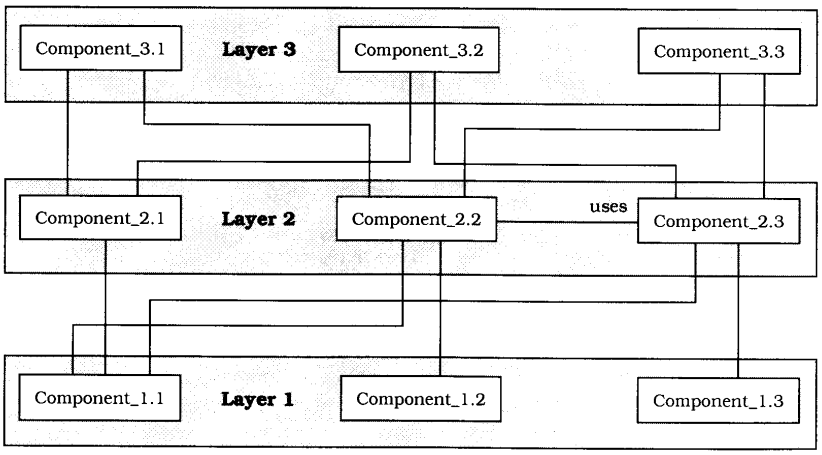
| ***Class*** Layer J | ***Collaborator*** • Layer J - 1 |
|---|---|
| ***Responsibility*** • Provides services used by Layer J+1. • Delegates subtasks to Layer J - 1. | |

# Layer: Structure






# Layer and components…



# Layers : Variants

- Relaxed Layered System:
  - A layer « j » can use service of j-1, j-2…
  - A layer can be partially opaque
    - Some service to layer j+1, others to all upper services…
- Layering through inheritance:
  - Lower layers are implemented as base classes
  - Higher level can override lower level…



# Layers : Known Uses

- Virtual machines: JVM and binary code format
- API : Layer that encapsulates lower layers
- Information System
  - Presentation, Application logic, Domain Layer, Database
- Windows NT (relaxed for: kernel and IO and hardware)
  - System services,
  - Resource management (Object manager, security monitor, process manager, I/O manager, VM manager, LPC),
  - Kernel (exception handling, interrupt, multipro synchro, threads),
  - HAL (Hardware Abstraction Level)
  - Hardware

# Layers: benefits

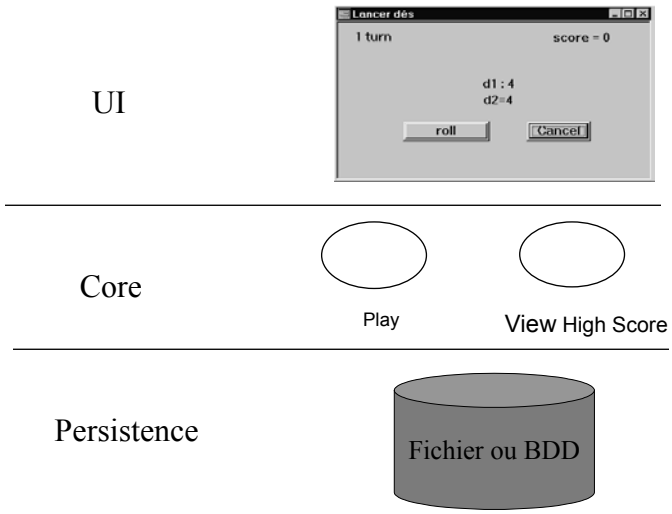
- Reuse of layers
- Support for standardization (POSIX)
- Dependencies are kept local
- Exchangeabilities :
  - Replacement of old implementation with Adapter Pattern
  - Dynamic exchange with Bridge Pattern



# Layers: Liabilities

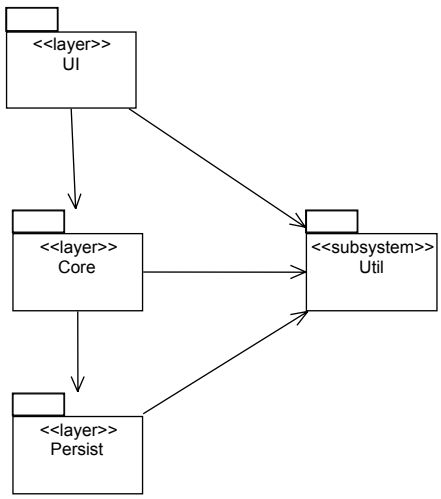
- Cascades of changing behavior
- Lower efficiency
- Unnecessary work: functions of a layer called many times for one service
- Difficulty of establishing correct granularity of layers: Too few layers -> less benefits, too many layers -> complexity and overhead…

# Applying Layer Architecture






# Package decomposition



## Layer « core »

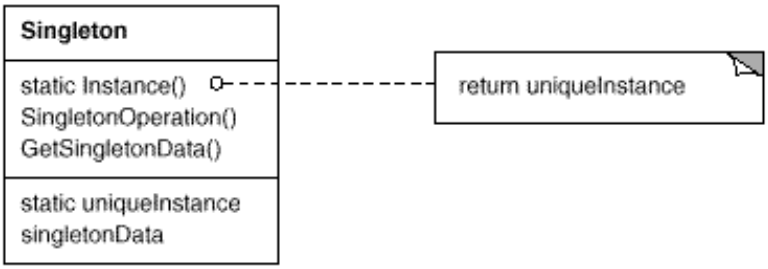
- Contain business logic classes…
- Adapt analysis classes for implementation
- Use of singleton Idiom…



## Singleton (Idiom)

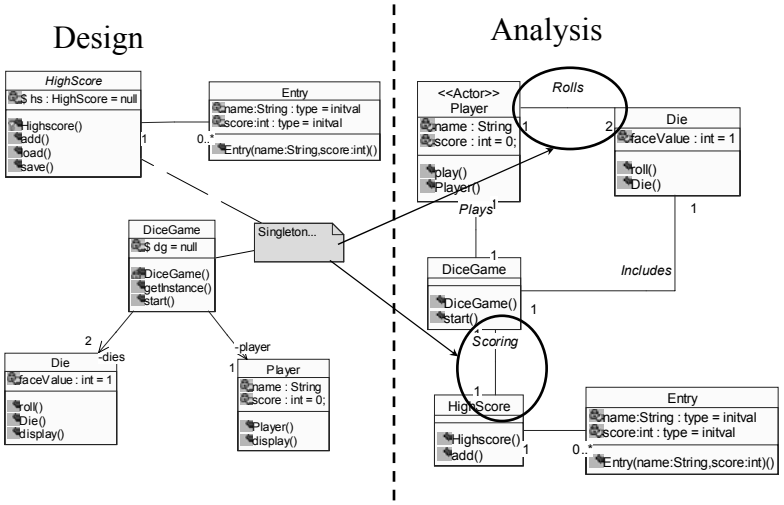
- Ensure a class only has one instance, and provide a global point of access to it.

# Singleton Structure

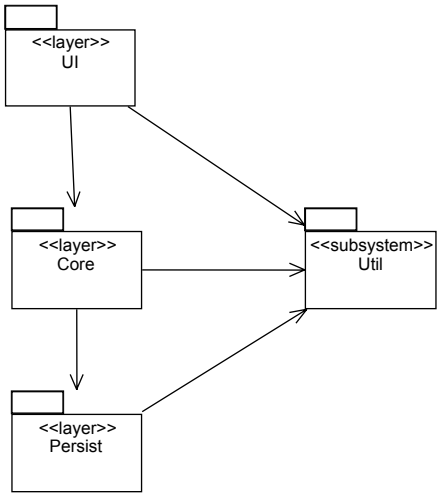





# Core « Layer »:First diagram



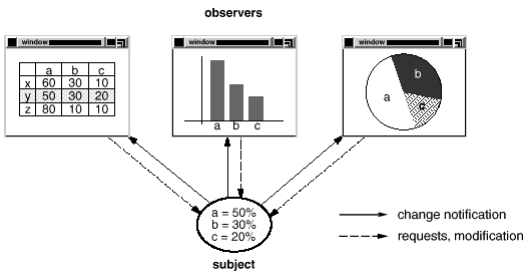
# Package decomposition






# Observer

- One-to-many dependency between objects: change of one object will automatically notify observers

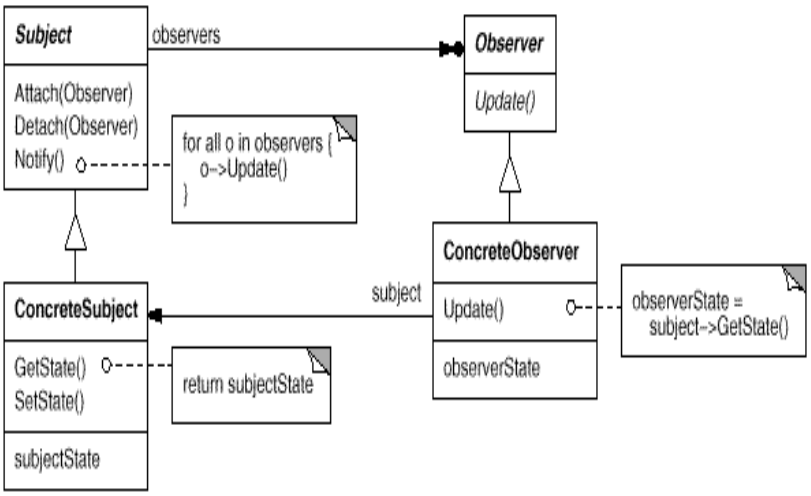


# Observer: Applicability

- A change to one object requires changing an unknown set of other objects
- Object should be able to notify other objects that may not be known at the beginning



# Observer: Structure

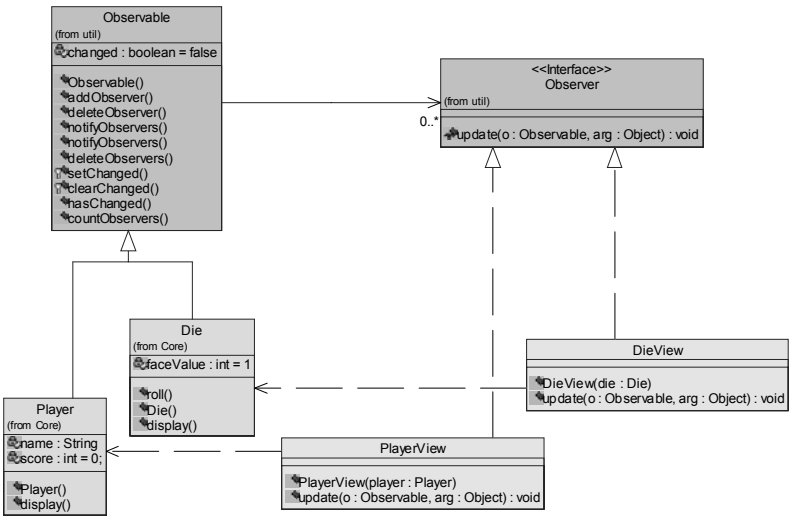


# Observer: Consequences

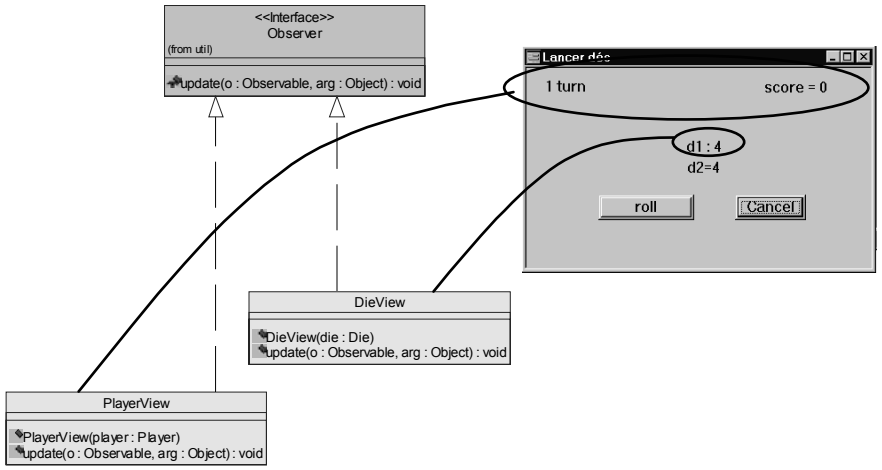
- Abstract coupling between subject and observer
- Support for broadcast communication
- Hard to maintain



# Applying Observer Pattern

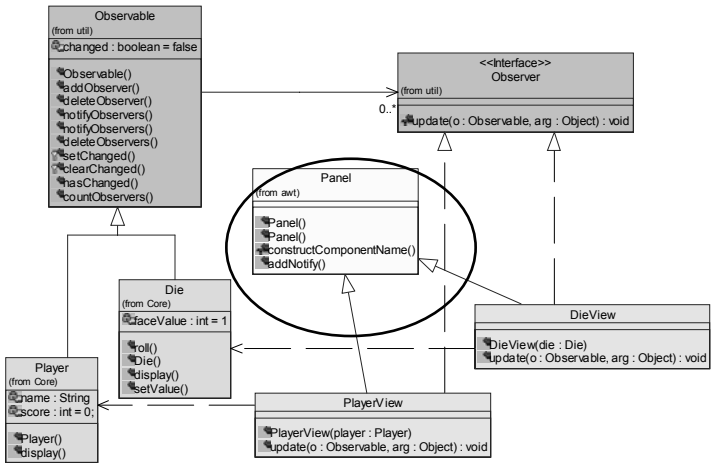


# Observer View

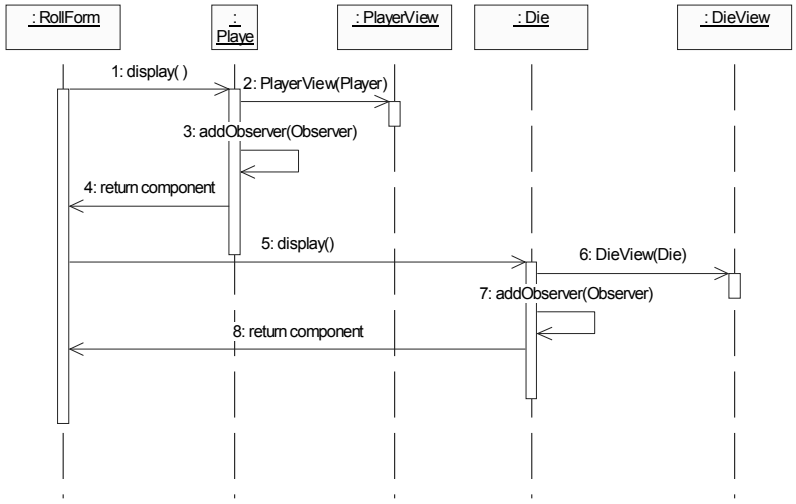





# Views are graphical objects

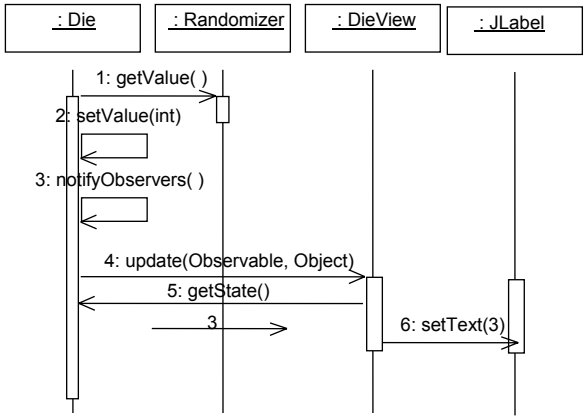


# Setting up Observer

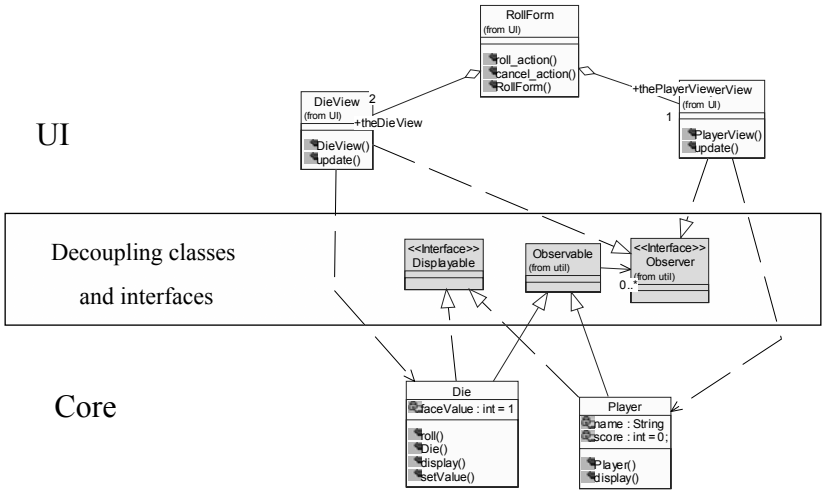





# Observer : Change Propagation

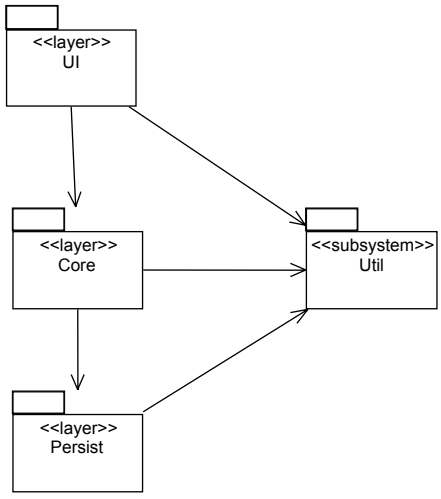


# Layered Architecture...






# Package decomposition



# Pattern Factory Method

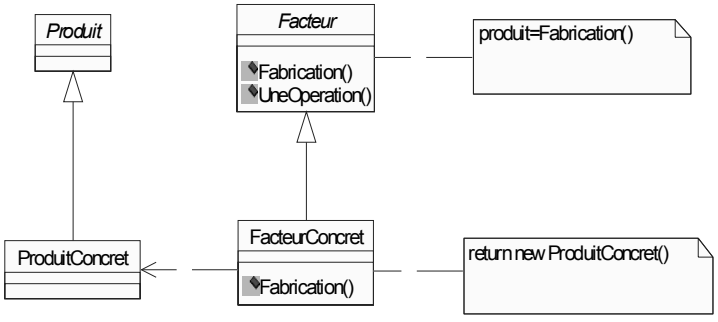
- Intent
  - Define an interface for creating an object, but let sub-classes decide which class to instantiate
  - let a class defer instantiation to subclasses
  - Also known as Virtual Constructor



# Factory Method

- Applicability : Use when
  - a class cannot anticipate the class of objects it must create
  - a class wants its subclasses to specify the objects it creates
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass to delegate.

# Structure






# Factory method

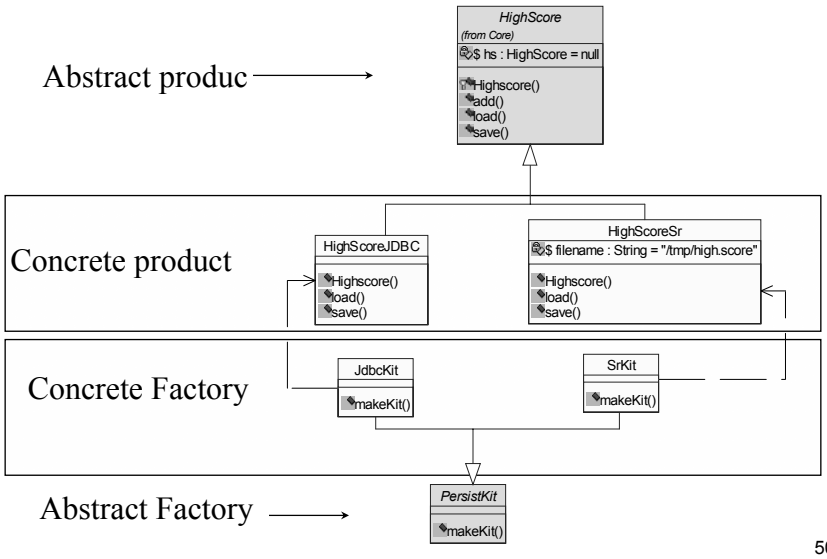
- Consequences
  - Provide hooks for subclasses
  - connects parallel class hierarchies
- Known uses
  - MacApp, ET++
  - ClassView in smalltalk80 MVC (controller creation)
  - Orbix ORB for generating PROXY object

# « Persist » Layer

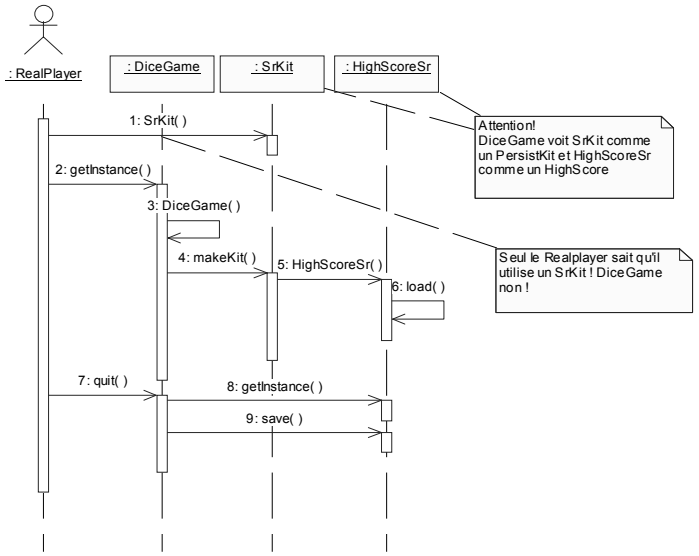
- Persistence technical classes
- Ensure independence of Core/Persist
  - Be able to switch « persistent engine »
- For example:
  - Persistence via « Serialization »
  - Persistence via a relational database (JDBC).



# Applying Factory



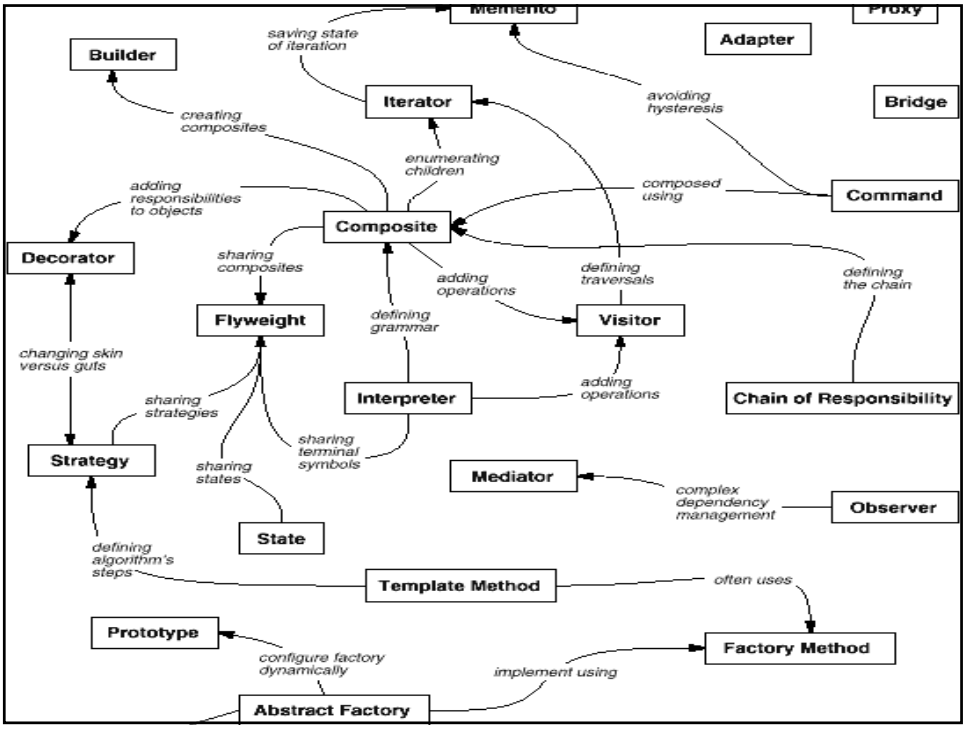
# Applying Factory






# Summary

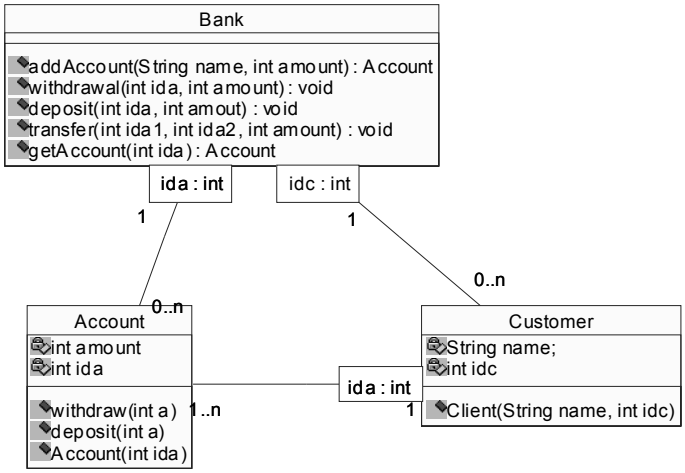
- 1 Architectural pattern : Layer
- 2 Design Patterns : Observer, Factory
- 1 Idiom : Singleton
- Pb:
  - Combining pattern to combine their forces…




# Bank example…

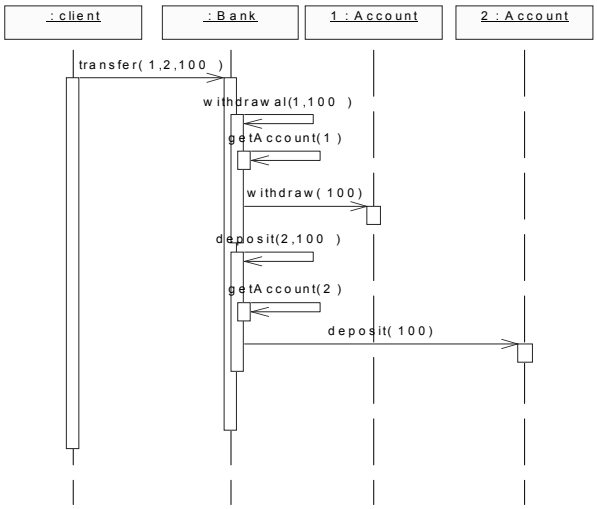
- A basic bank system:
  - 1 bank, n Account.
  - Each account belong to 1 client.
  - Each account is credited by an amount a money.
- Bank functions
  - Withdrawal on an account, Credit an account, Transfer money from one account to another…

# Naive solution






# Naive Solution

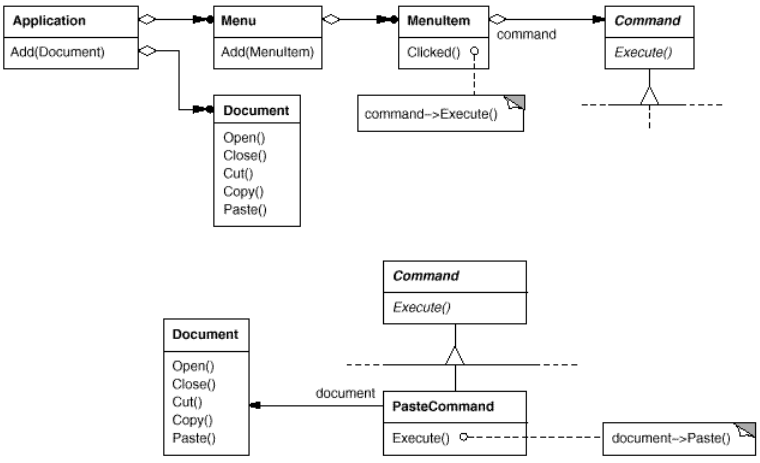


# Applying Command Pattern…

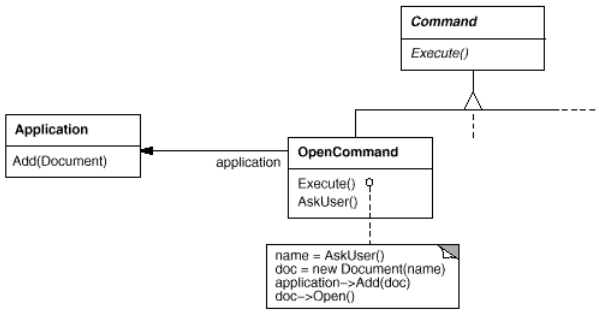
- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



# Command Example

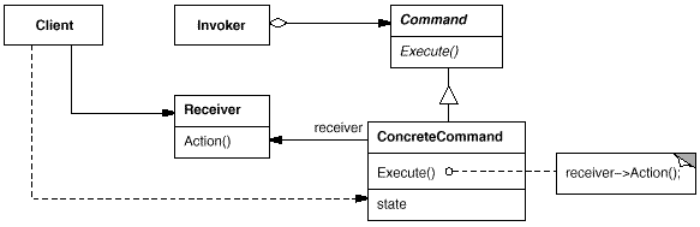


# Command Example

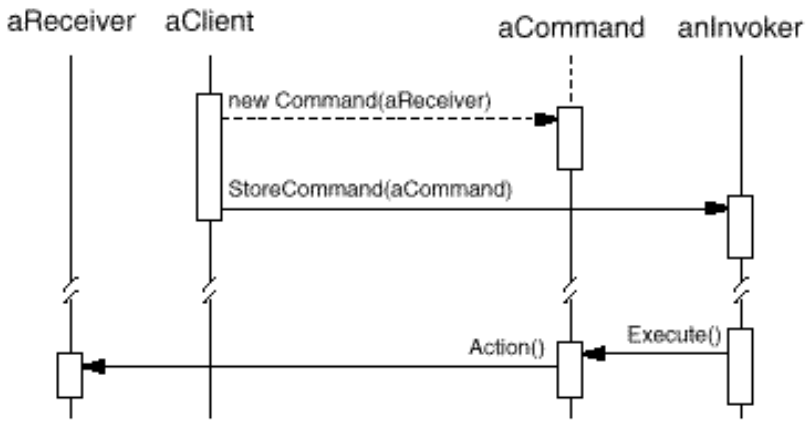





# Command Structure



# Command Structure

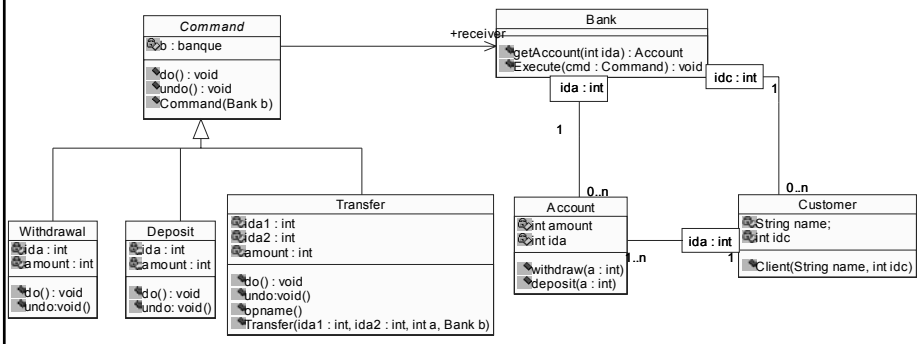





# Command Consequences

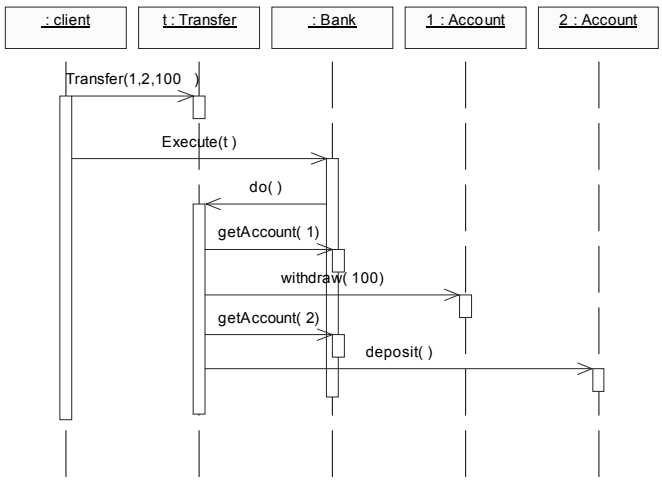
- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands are first-class objects. They can be manipulated and extended like any other object.
- It's easy to add new Commands, because you don't have to change existing classes.

# Applying Command Pattern






# Applying Command Pattern

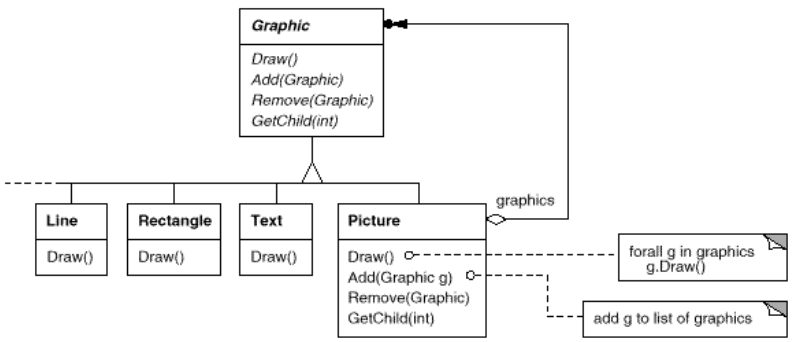


# Composite Pattern

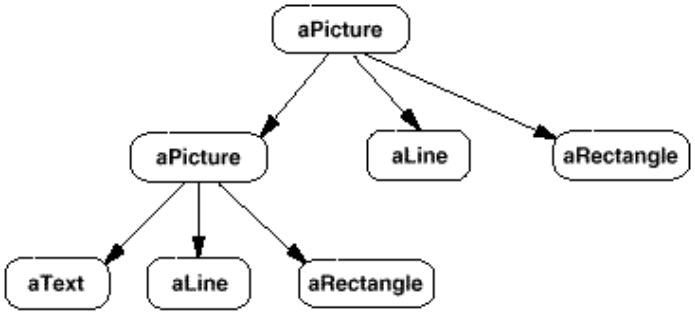
- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



# Composite Example

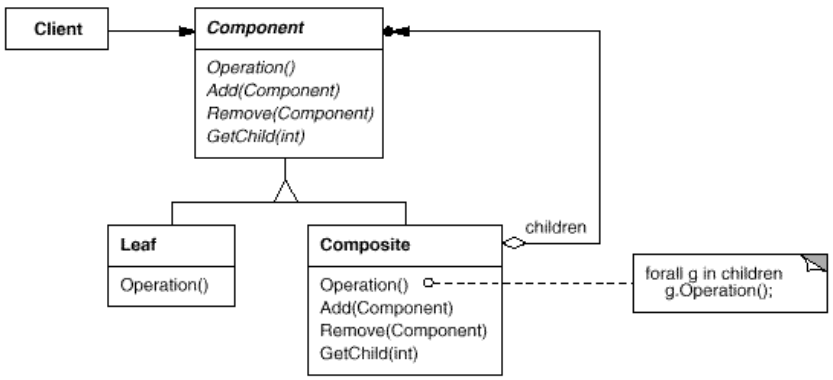


# Composite Example

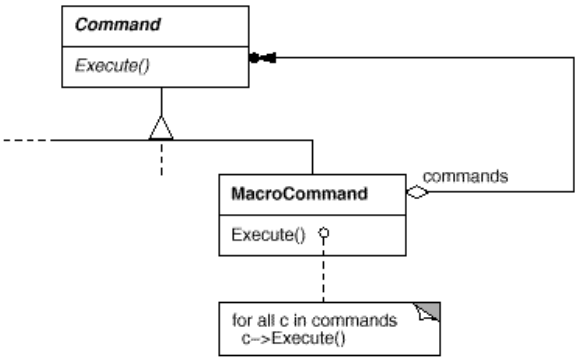





# Composite Structure

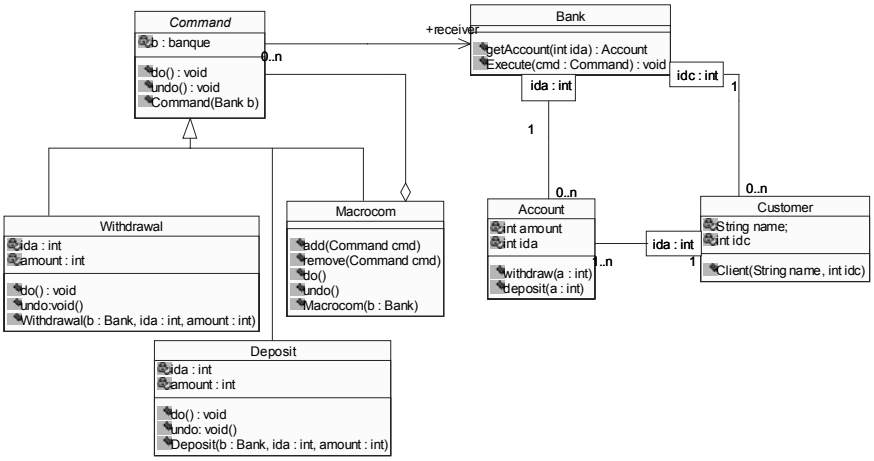


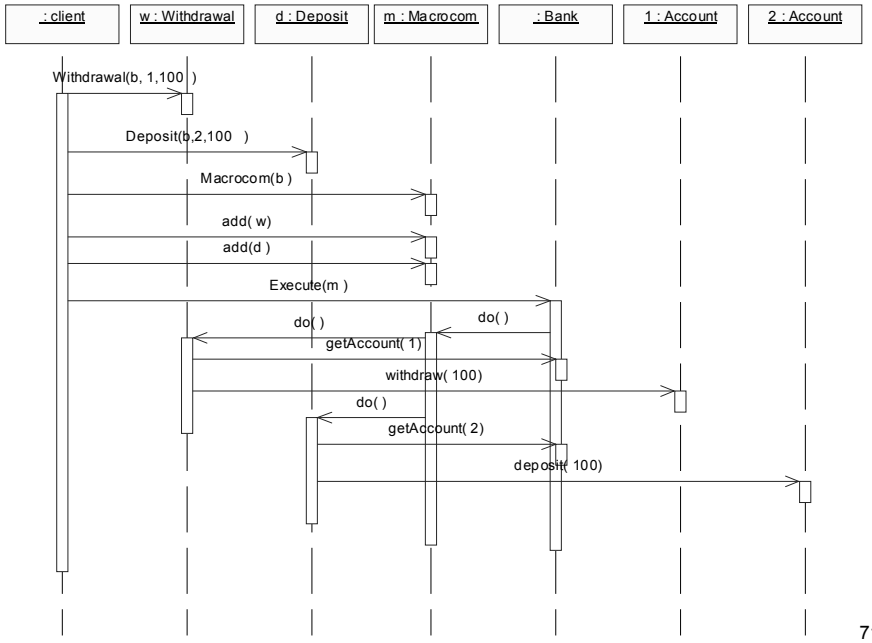
# Applying Composite on Command





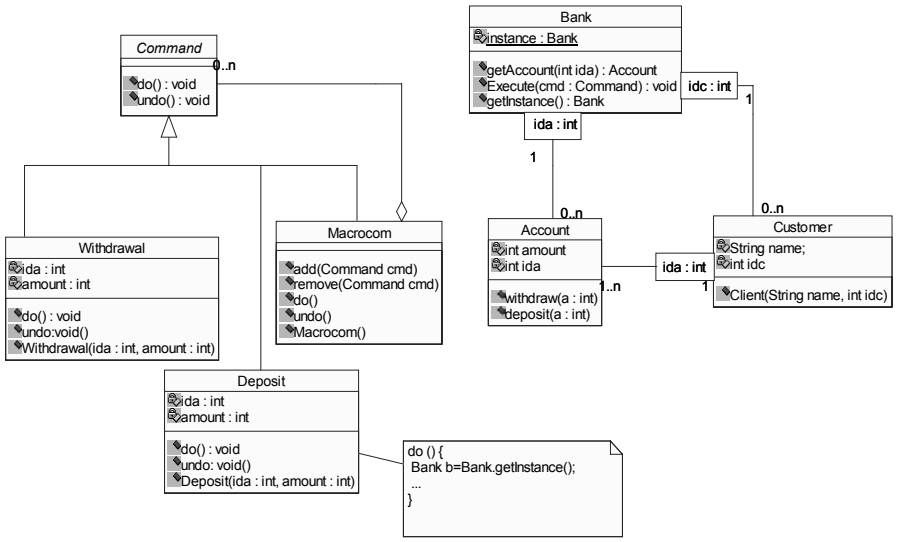

# Applying Composite








# Applying Singleton



## And So on…

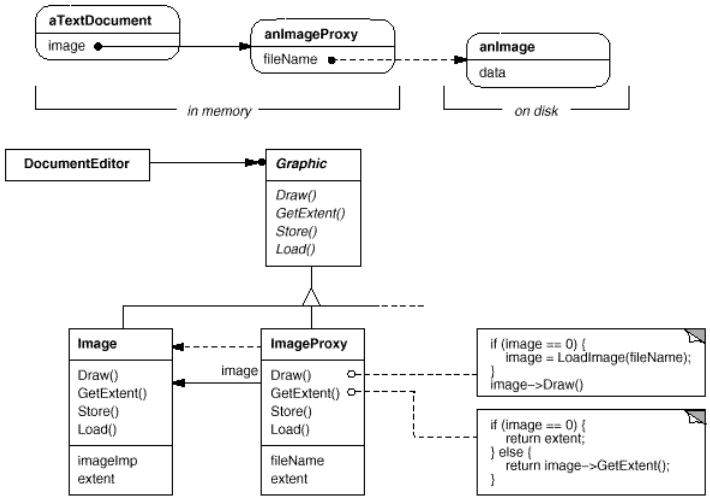
- Storing state : Memento Pattern
- Observing Account : Observer Pattern
- Visiting all object graph : Visitor Pattern
- Remote access : Proxy pattern
- …



## Proxy Pattern

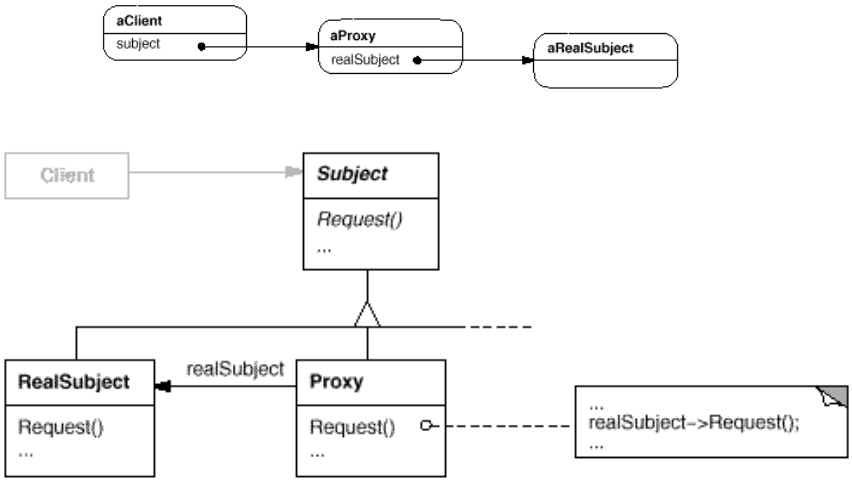
- Provide a surrogate or placeholder for another object to control access to it.

# Proxy Example





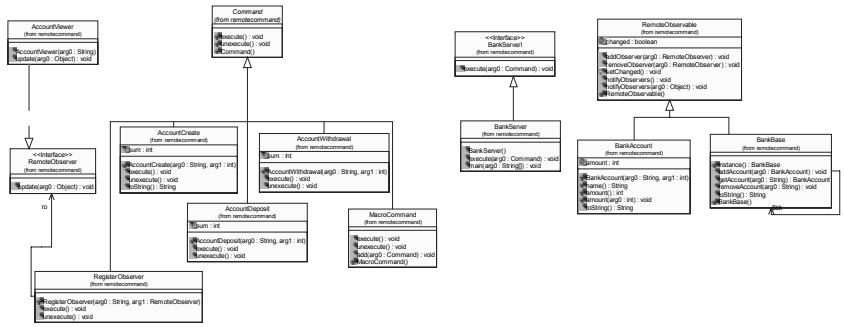

# Proxy Structure



# Proxy benefits

- remote proxy can hide the fact that an object resides in a different address space.
- A virtual proxy can perform optimizations such as creating an object on demand.
- Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.



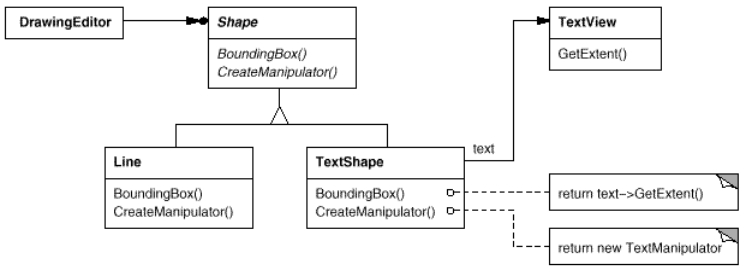


# Adapter Pattern

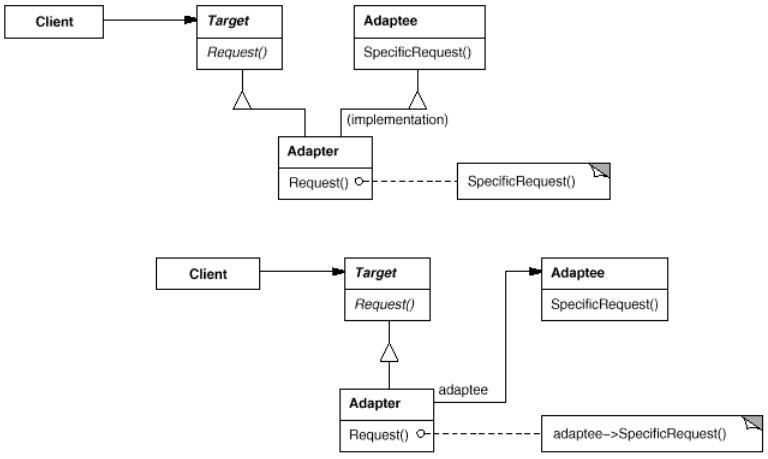
- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



# Adapter Example



# Adapter Structure

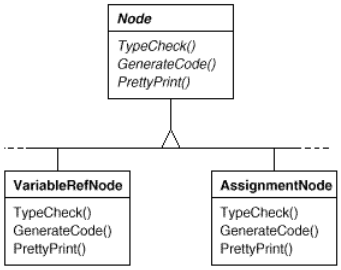





# Visitor Pattern

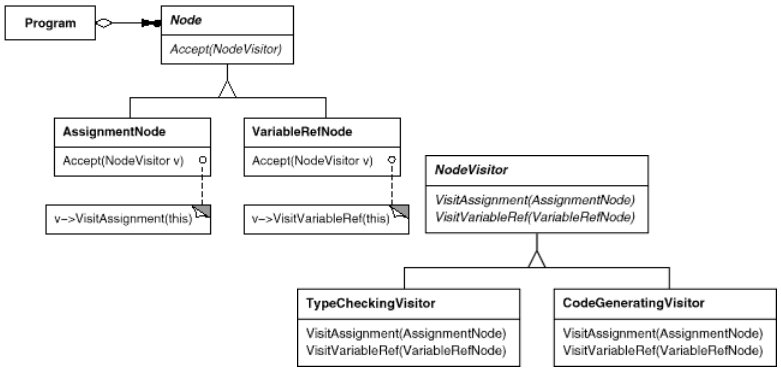
- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

# Visitor example






# Visitor example

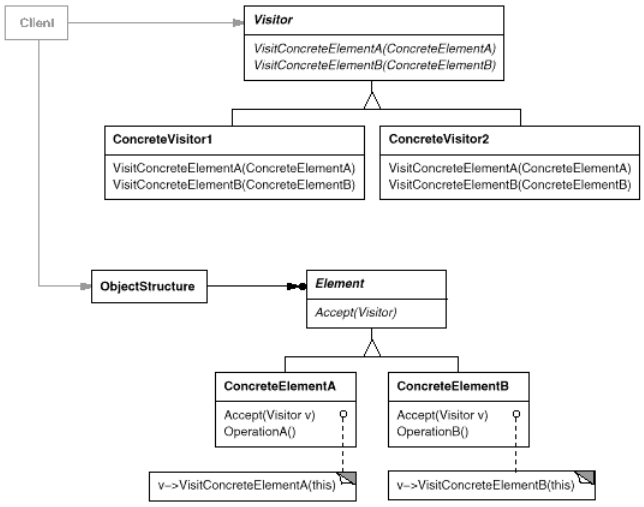


# Visitor applicability

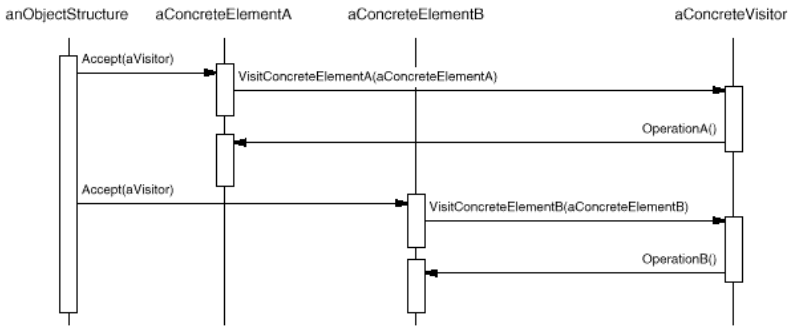
- many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations



# Visitor Structure



# Visitor Structure





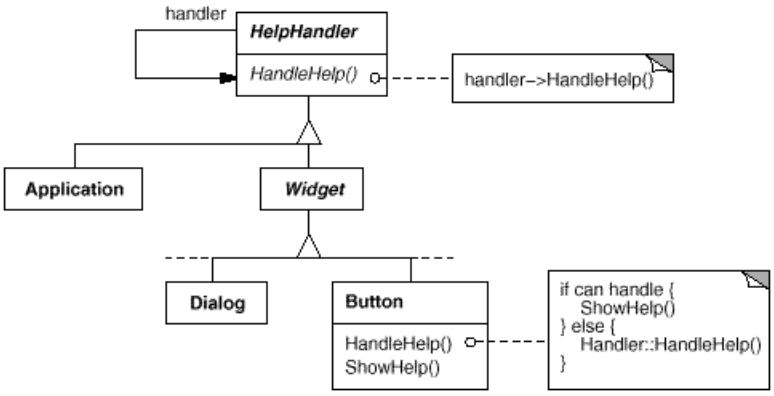

# Visitor Consequences

- *Visitor makes adding new operations easy*
- *A visitor gathers related operations and separates unrelated ones*
- *Adding new Concrete Element classes is hard*
- *Visiting across class hierarchies*
- *Accumulating state.*
- *Breaking encapsulation*

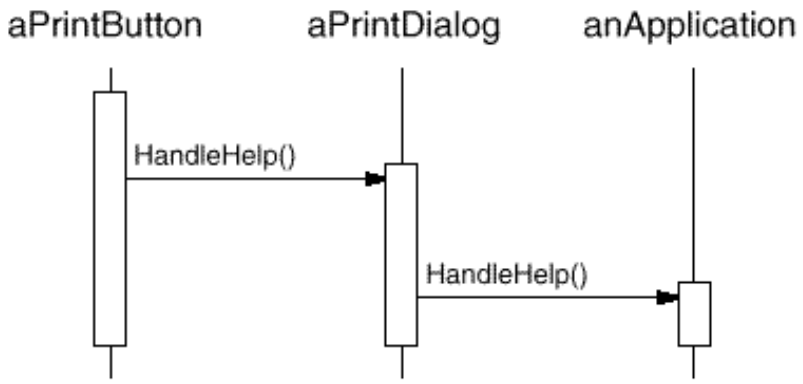
# Chain of responsibility

- 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.



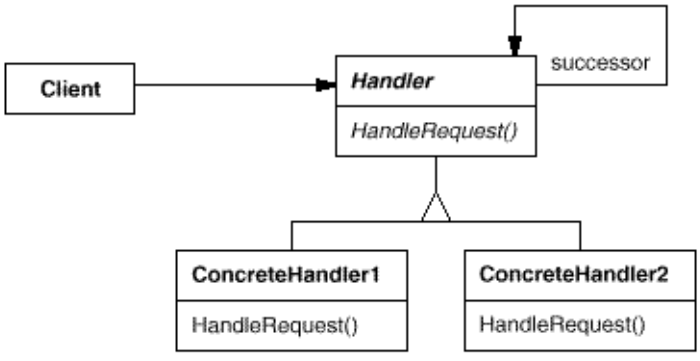


# Chain of Responsibility






# Chain of Responsibility



# Participants

- **Handler** (HelpHandler)
  - defines an interface for handling requests.
  - (optional) implements the successor link.
- **ConcreteHandler** (PrintButton, PrintDialog)
  - handles requests it is responsible for.
  - can access its successor.
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.
- **Client**
  - initiates the request to a ConcreteHandler object on the chain.



# Example…

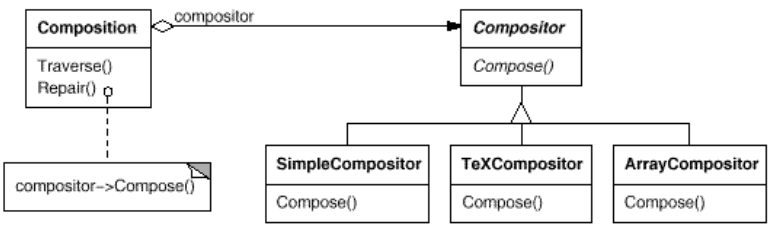
- Awt 1.0

# Strategy

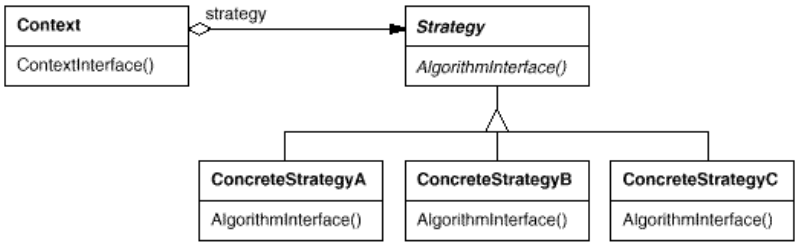
- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



# Strategy…



# Strategy

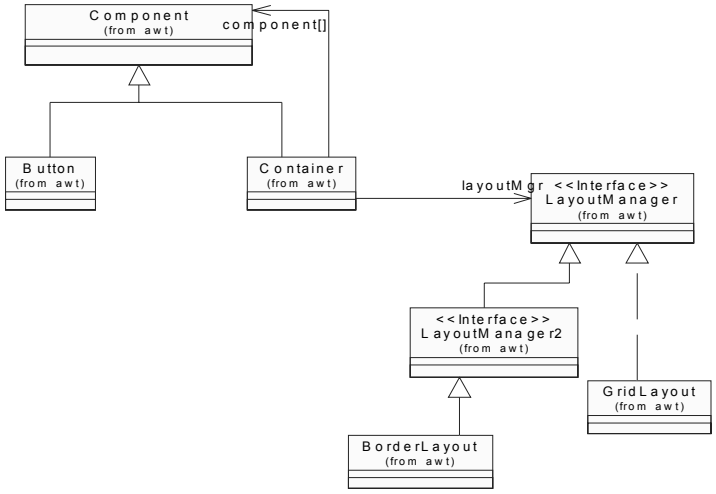





# Participants

- **Strategy** (Compositor)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor)
  - implements the algorithm using the Strategy interface.
- **Context** (Composition)
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

# Strategy…

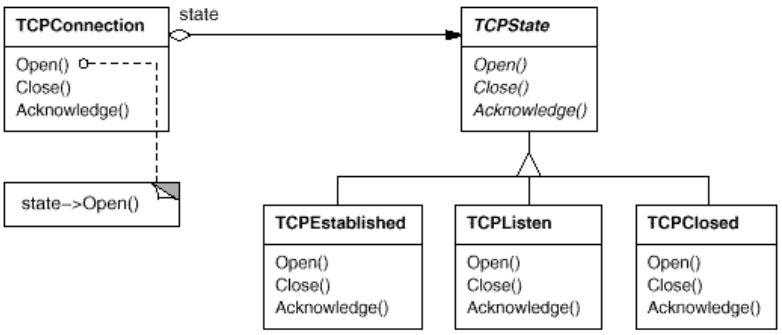





# State

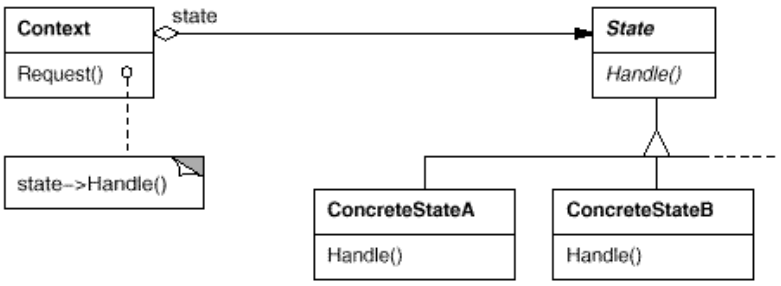
- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

# Example






# Structure



## Consequences

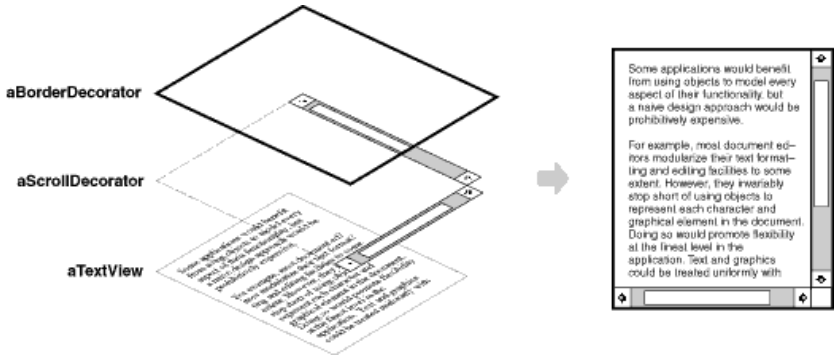
1. *It localizes state-specific behavior and partitions behavior for different states*
2. *It makes state transitions explicit*
3. *State objects can be shared*



## Decorator

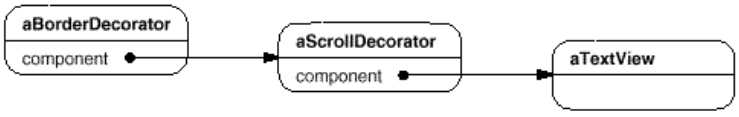
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

# Example

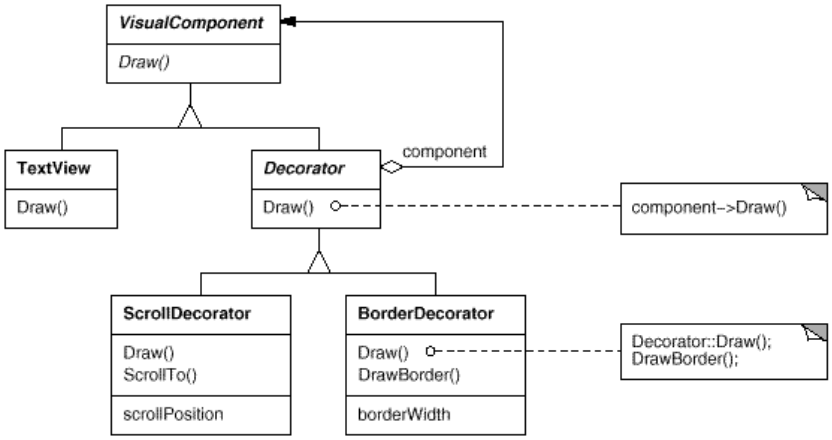





# Example

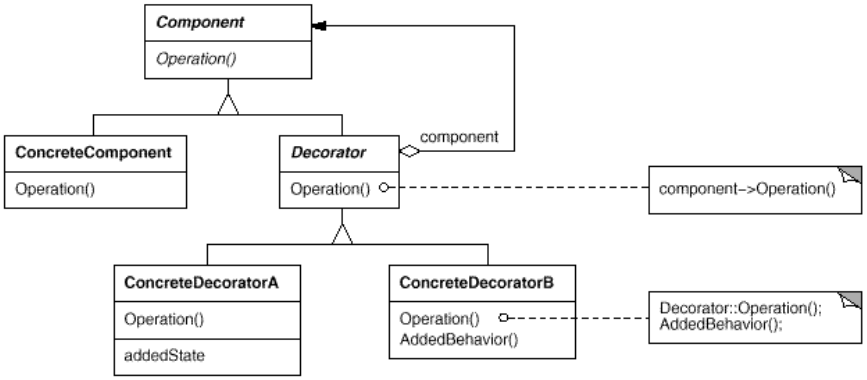


# Example






# Structure



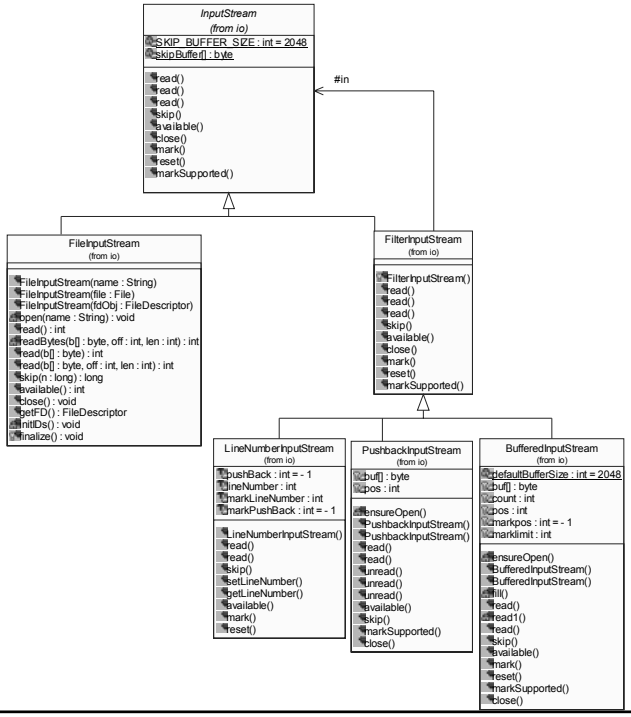
# Applicability

- to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn
- when extension by subclassing is impractical



# Consequences

1. *More flexibility than static inheritance*
2. *Avoids feature-laden classes high up in the hierarchy*
3. *A decorator and its component aren't identical*
4. *Lots of little objects*

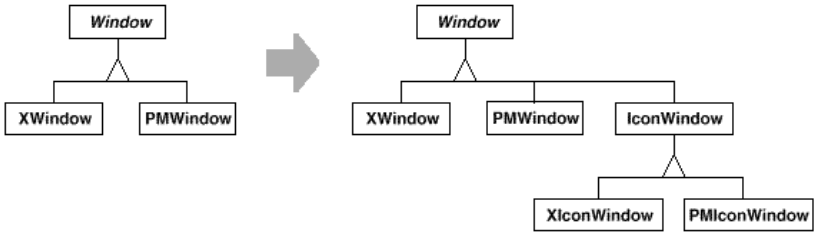





# Bridge

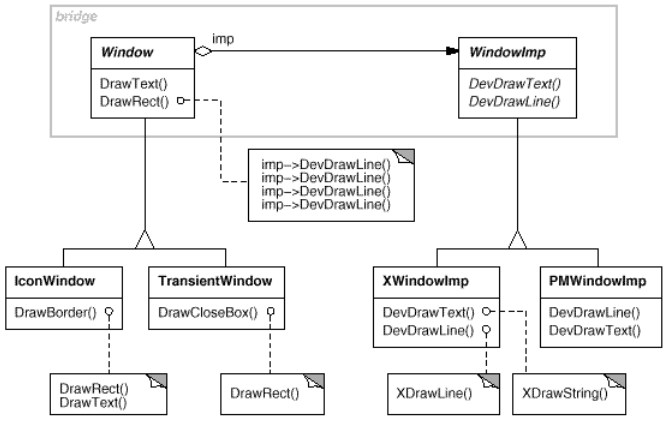
- Decouple an abstraction from its implementation so that the two can vary independently.

# Bridge

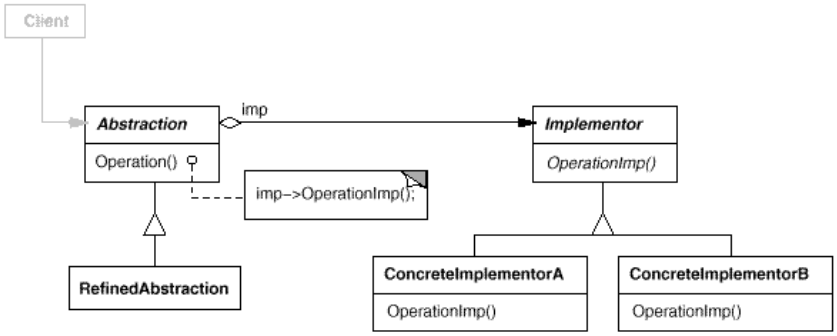





# Bridge



# Bridge Structure…

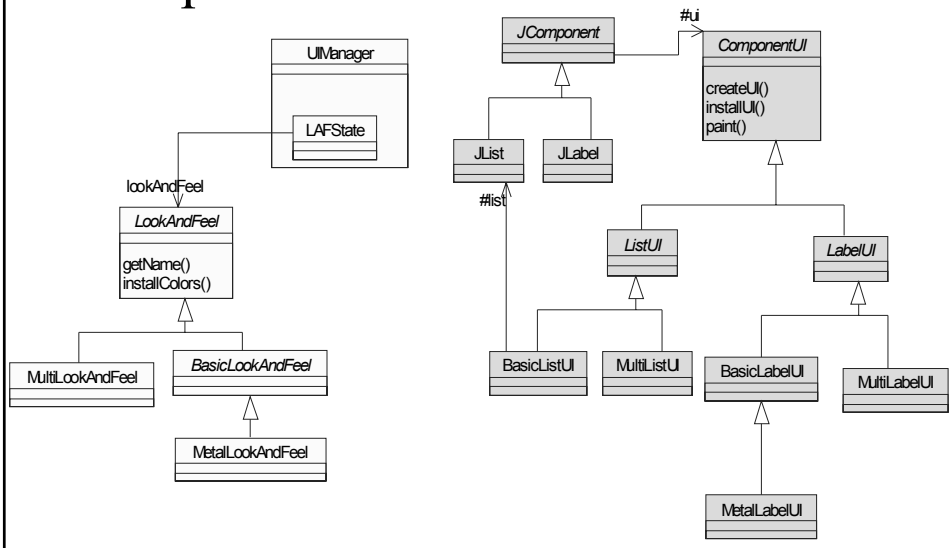




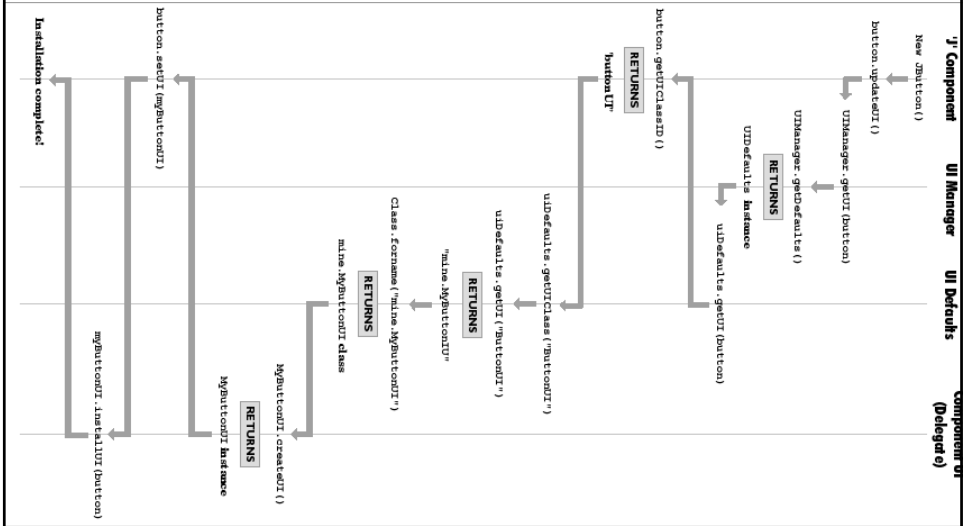

# Bridge

1. *Decoupling interface and implementation*
2. *Improved extensibility*
3. *Hiding implementation details from clients*

# Example


UIManager
LAFState
lookAndFeel
LookAndFeel
getName()
installColors()
MultiLookAndFeel
BasicLookAndFeel
MetalLookAndFeel
JComponent
#ui
ComponentUI
createUI()
installUI()
paint()
JList
JLabel
#list
ListUI
LabelUI
BasicListUI
MultiListUI
BasicLabelUI
MultiLabelUI
MetalLabelUI



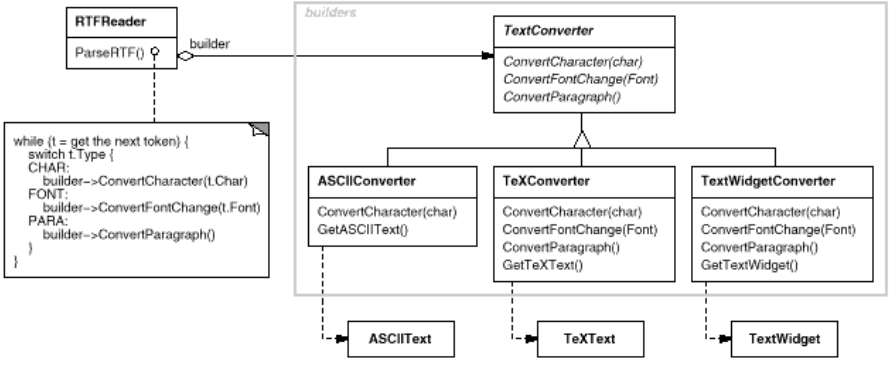

'j' Component
UI Manager
UI Defaults
ComponentUI (Delegate)
New JButton()
button.updateUI()
UIManager.getUI(button)
UIManager.getDefaults()
RETURNS
UIDefaults instance
uiDefaults.getUI(button)
button.getUIClassID()
RETURNS
"buttonUI"
uiDefaults.getUIClass("ButtonUI")
uiDefaults.getUI("ButtonUI")
RETURNS
"mine.MyButtonUI"
Class.forName("mine.MyButtonUI")
RETURNS
mine.MyButtonUI class
MyButtonUI.createUI()
RETURNS
MyButtonUI instance
myButtonUI.installUI(button)
button.setUI(myButtonUI)
Installation complete!

# Builder

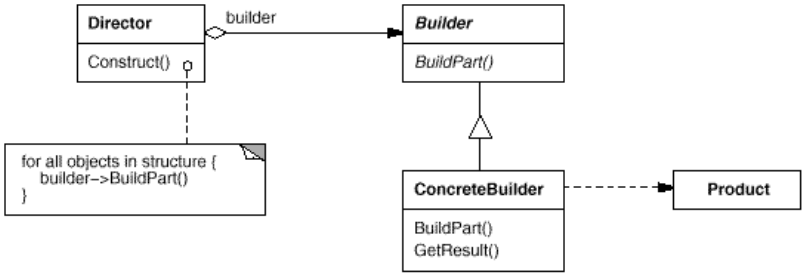
- Separate the construction of a complex object from its representation so that the same construction process can create different representations.



# Builder

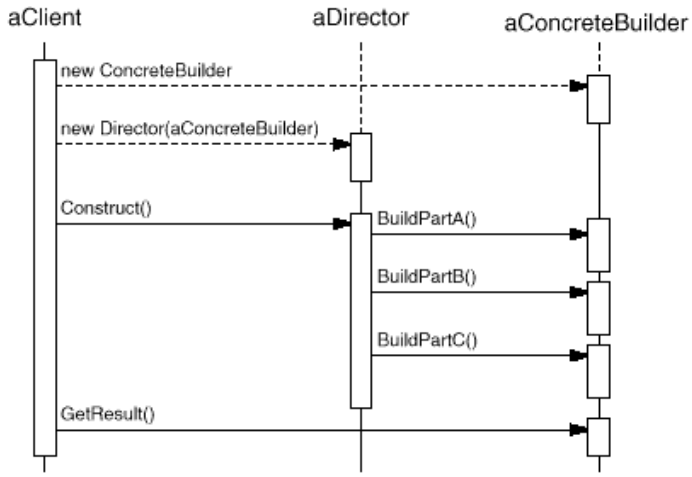


# Builder Structure…






# Builder



# Builder Consequences

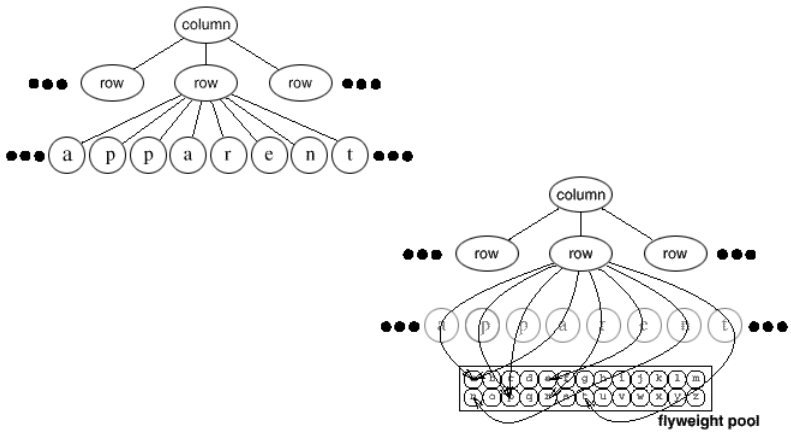
1. *It lets you vary a product's internal representation*
2. *It isolates code for construction and representation*
3. *It gives you finer control over the construction process*



# FlyWeight

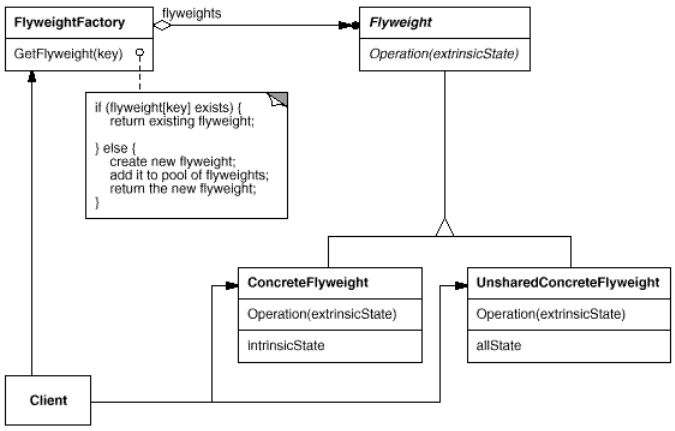
- Use sharing to support large numbers of fine-grained objects efficiently.

# FlyWeight

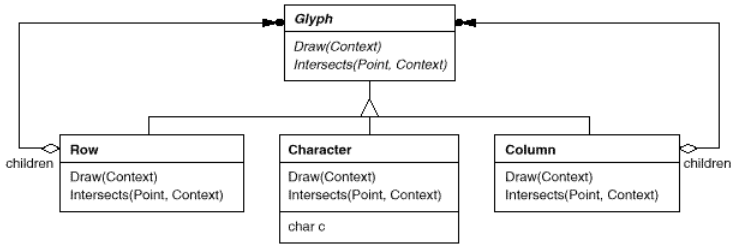





# Flyweight: Structure

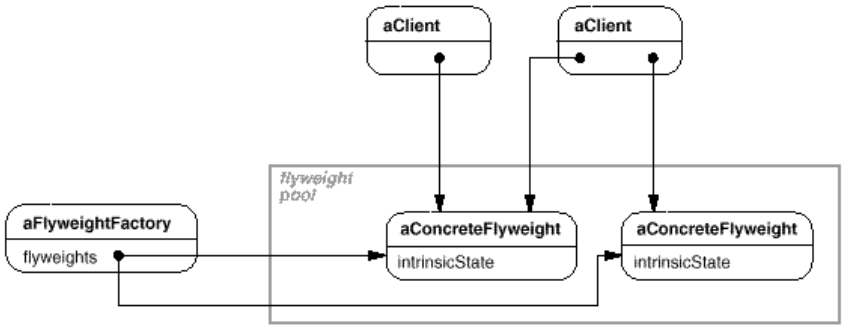


# Flyweight example






# Flyweight: Instances

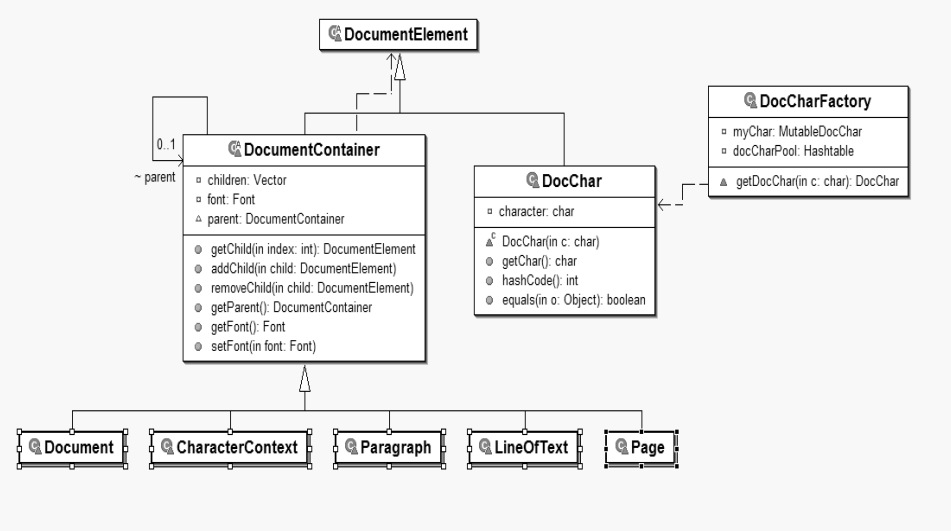


# Flyweight: Applicability

- Etat intrinsèque/extrinsèque…
- Les états extrinsèques peuvent être calculés…



# Flyweight (il a rien compris… ☺)

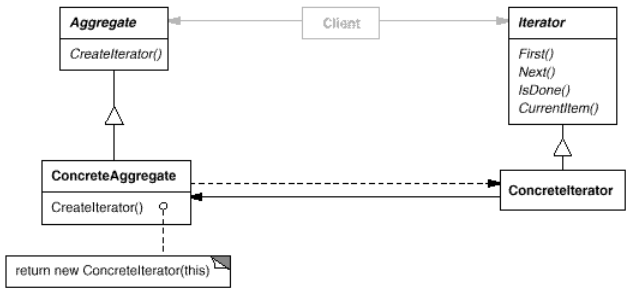


# Iterator

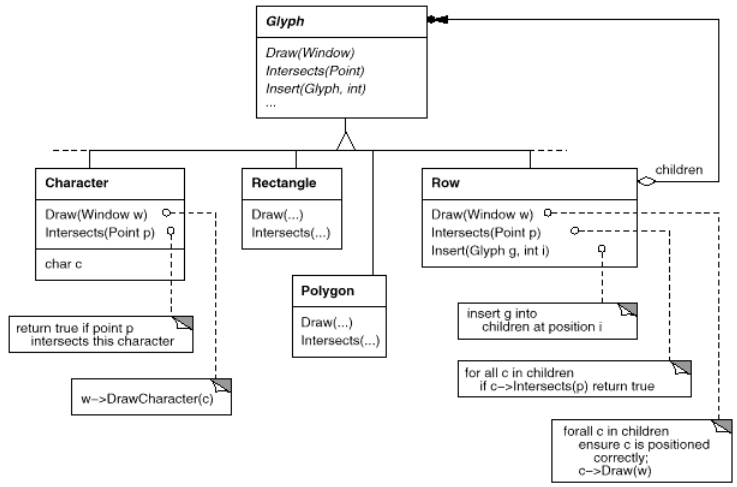
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation



# Iterator

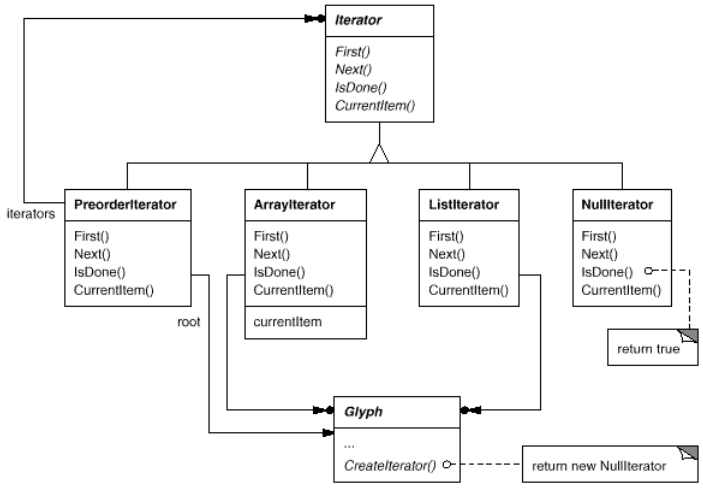


# Iterator example:






# Example

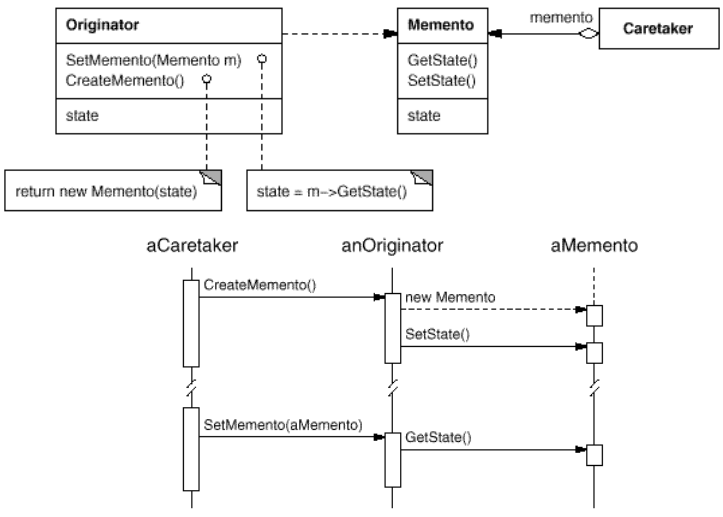


# Memento

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



# Memento Structure…

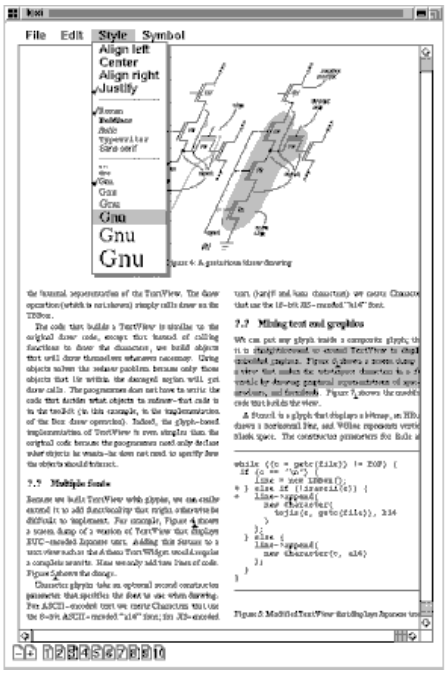


# Memento…

1. *Preserving encapsulation boundaries*
2. *It simplifies Originator*
3. *Using mementos might be expensive.*
4. *Defining narrow and wide interfaces*
5. *Hidden costs in caring for mementos*



# Case Study



# Design problems…

- *Document structure.* The choice of internal representation for the document affects nearly every aspect of Lexi's design. All editing, formatting, displaying, and textual analysis will require traversing the representation. The way we organize this information will impact the design of the rest of the application.
- *Formatting.* How does Lexi actually arrange text and graphics into lines and columns? What objects are responsible for carrying out different formatting policies? How do these policies interact with the document's internal representation?



# Design problems…

- *Embellishing the user interface.* Lexi's user interface includes scroll bars, borders, and drop shadows that embellish the WYSIWYG document interface. Such embellishments are likely to change as Lexi's user interface evolves. Hence it's important to be able to add and remove embellishments easily without affecting the rest of the application.
- *Supporting multiple look-and-feel standards.* Lexi should adapt easily to different look-and-feel standards such as Motif and Presentation Manager (PM) without major modification.

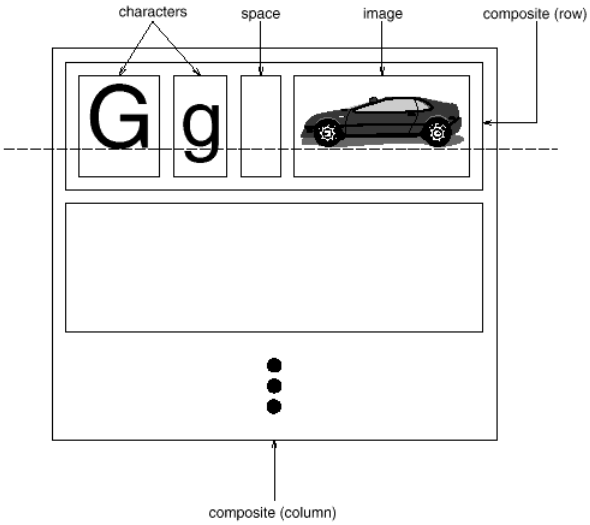
# Design problems…

- *Embellishing the user interface.* Lexi's user interface includes scroll bars, borders, and drop shadows that embellish the WYSIWYG document interface. Such embellishments are likely to change as Lexi's user interface evolves. Hence it's important to be able to add and remove embellishments easily without affecting the rest of the application.
- *Supporting multiple look-and-feel standards.* Lexi should adapt easily to different look-and-feel standards such as Motif and Presentation Manager (PM) without major modification.

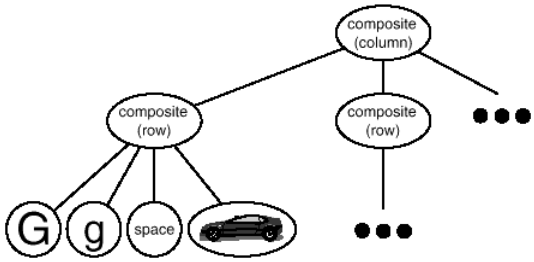


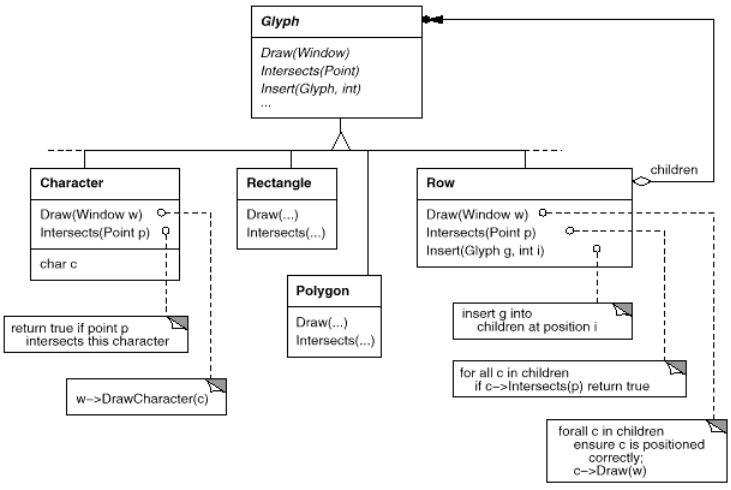
# Design problems…

- *Spelling checking and hyphenation.* How does Lexi support analytical operations such as checking for misspelled words and determining hyphenation points? How can we minimize the number of classes we have to modify to add a new analytical operation?

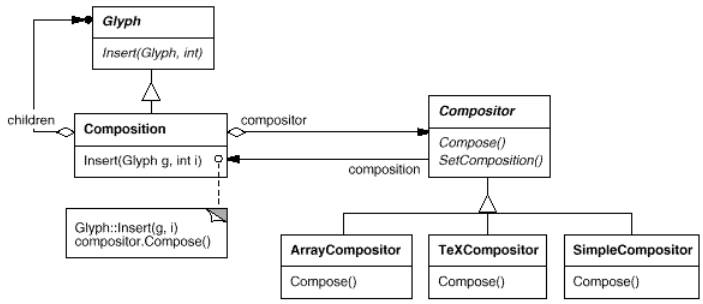

characters
space
image
composite (row)
G
g
composite (column)

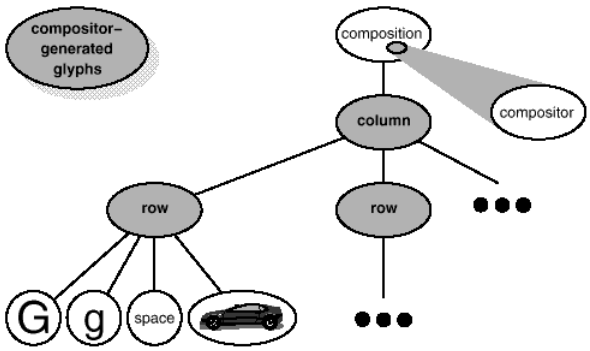



composite (column)
composite (row)
composite (row)
G
g
space

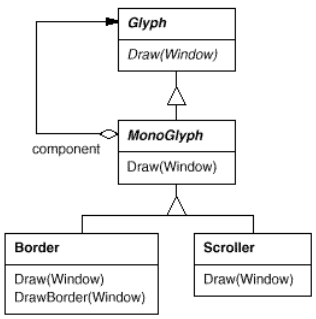

Glyph
Draw(Window)
Intersects(Point)
Insert(Glyph, int)
...
Character
Draw(Window w)
Intersects(Point p)
char c
Rectangle
Draw(...)
Intersects(...)
Polygon
Draw(...)
Intersects(...)
Row
Draw(Window w)
Intersects(Point p)
Insert(Glyph g, int i)
children
return true if point p
intersects this character
w->DrawCharacter(c)
insert g into
children at position i
for all c in children
if c->Intersects(p) return true
forall c in children
ensure c is positioned
correctly;
c->Draw(w)

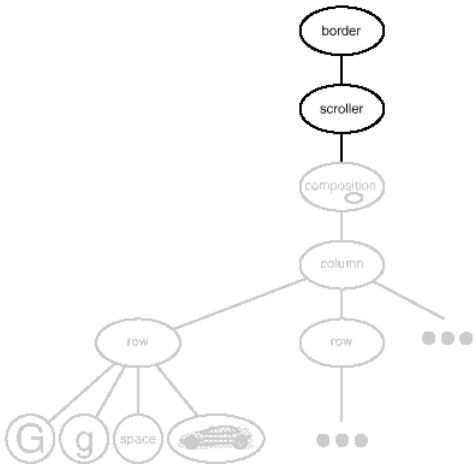



Glyph
Insert(Glyph, int)
children
Composition
Insert(Glyph g, int i)
compositor
composition
Compositor
Compose()
SetComposition()
Glyph::Insert(g, i)
compositor.Compose()
ArrayCompositor
Compose()
TeXCompositor
Compose()
SimpleCompositor
Compose()

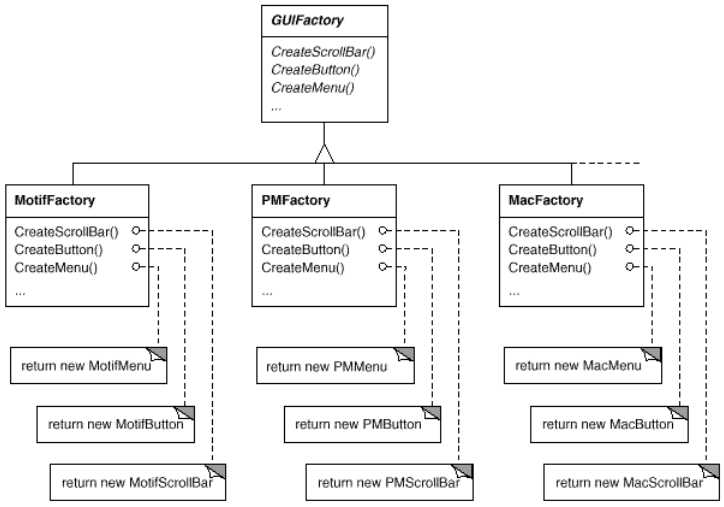

compositor–
generated
glyphs
composition
compositor
column
row
row
G
g
space

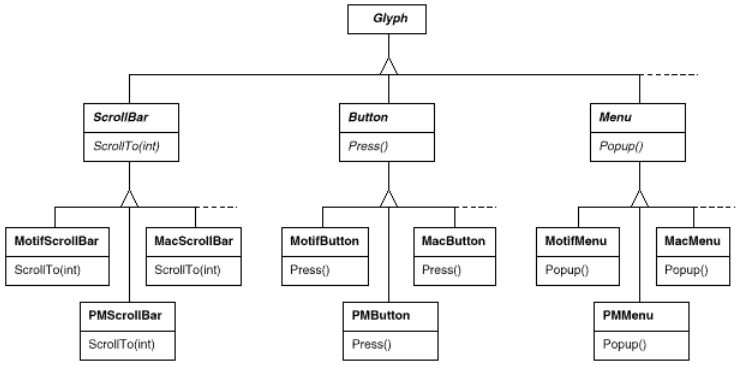



Glyph
Draw(Window)
component
MonoGlyph
Draw(Window)
Border
Draw(Window)
DrawBorder(Window)
Scroller
Draw(Window)

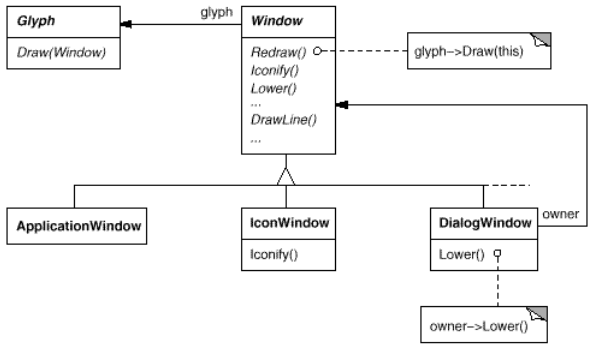

border
scroller
composition
column
row
row
G
g
space

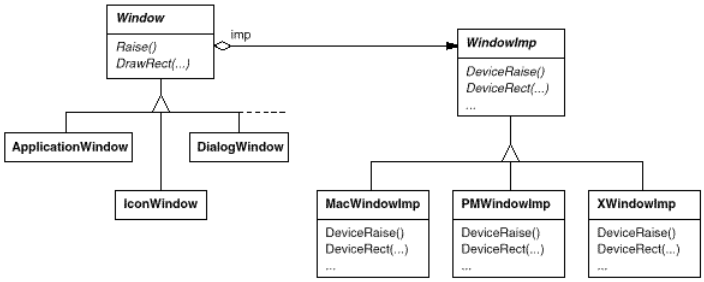



GUIFactory
CreateScrollBar()
CreateButton()
CreateMenu()
...
MotifFactory
CreateScrollBar()
CreateButton()
CreateMenu()
...
PMFactory
CreateScrollBar()
CreateButton()
CreateMenu()
...
MacFactory
CreateScrollBar()
CreateButton()
CreateMenu()
...
return new MotifMenu
return new MotifButton
return new MotifScrollBar
return new PMMenu
return new PMButton
return new PMScrollBar
return new MacMenu
return new MacButton
return new MacScrollBar

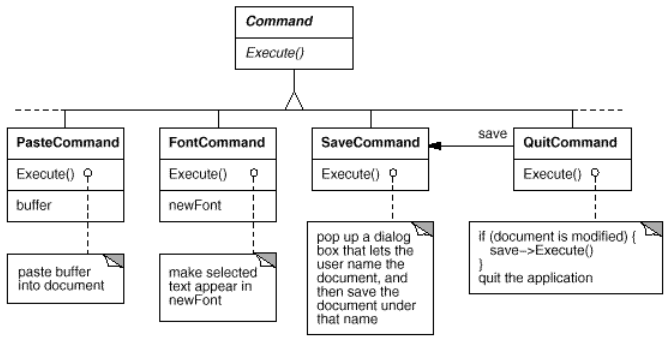

Glyph
ScrollBar
ScrollTo(int)
Button
Press()
Menu
Popup()
MotifScrollBar
ScrollTo(int)
MacScrollBar
ScrollTo(int)
PMScrollBar
ScrollTo(int)
MotifButton
Press()
MacButton
Press()
PMButton
Press()
MotifMenu
Popup()
MacMenu
Popup()
PMMenu
Popup()

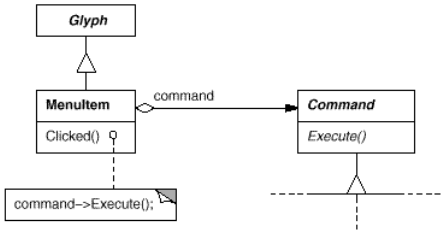



Glyph
Draw(Window)
glyph
Window
Redraw()
Iconify()
Lower()
...
DrawLine()
...
glyph->Draw(this)
ApplicationWindow
IconWindow
Iconify()
DialogWindow
Lower()
owner
owner->Lower()

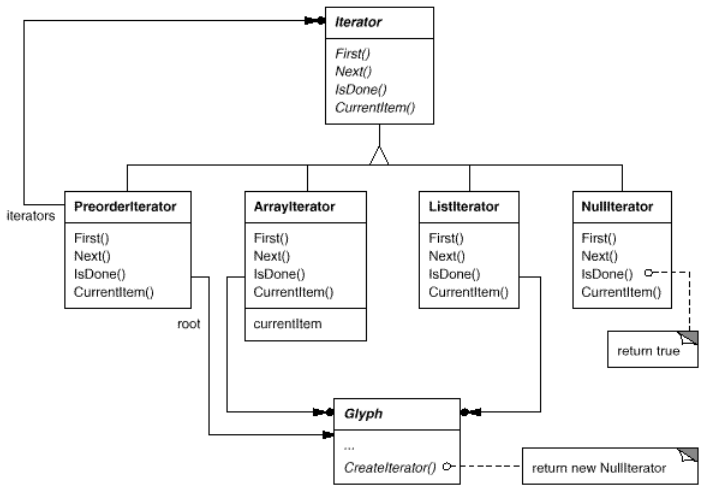

Window
Raise()
DrawRect(...)
imp
WindowImp
DeviceRaise()
DeviceRect(...)
...
ApplicationWindow
DialogWindow
IconWindow
MacWindowImp
DeviceRaise()
DeviceRect(...)
...
PMWindowImp
DeviceRaise()
DeviceRect(...)
...
XWindowImp
DeviceRaise()
DeviceRect(...)
...

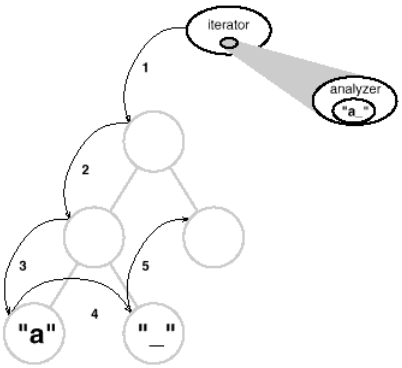



Command
Execute()
PasteCommand
Execute()
buffer
FontCommand
Execute()
newFont
SaveCommand
Execute()
save
QuitCommand
Execute()
paste buffer into document
make selected text appear in newFont
pop up a dialog box that lets the user name the document, and then save the document under that name
if (document is modified) {
save->Execute()
}
quit the application

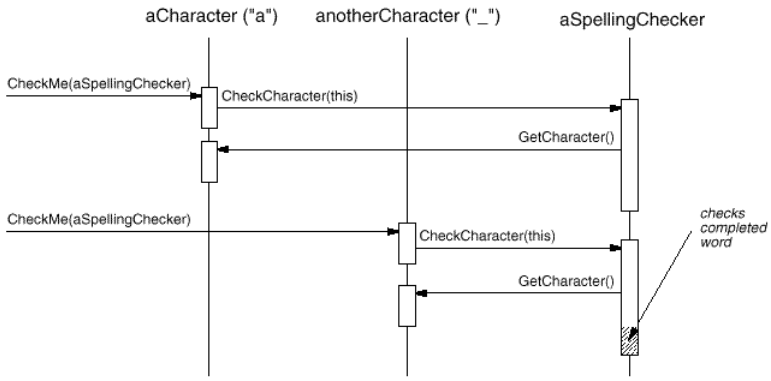

Glyph
MenuItem
Clicked()
command
Command
Execute()
command->Execute();




Iterator
First()
Next()
IsDone()
CurrentItem()
iterators
PreorderIterator
First()
Next()
IsDone()
CurrentItem()
root
ArrayIterator
First()
Next()
IsDone()
CurrentItem()
currentItem
ListIterator
First()
Next()
IsDone()
CurrentItem()
NullIterator
First()
Next()
IsDone()
CurrentItem()
return true
Glyph
...
CreateIterator()
return new NullIterator


iterator
analyzer
"a"
1
2
3
4
5
"a"
"_"




aCharacter ("a")
anotherCharacter ("_")
aSpellingChecker
CheckMe(aSpellingChecker)
CheckCharacter(this)
GetCharacter()
CheckMe(aSpellingChecker)
CheckCharacter(this)
GetCharacter()
checks completed word

# Summary (C. Alexander)

- It is possible to create building architectures by stringing together patterns in a rather loose way. A building made like this, is an assembly of patterns. It is not dense. It is not profound. But it is also possible to put patterns together in such way that many patterns overlap in the same physical space: the building is very dense; it has many meanings captured in a small space; and through this density, it becomes profound.



# Architectural Patterns…

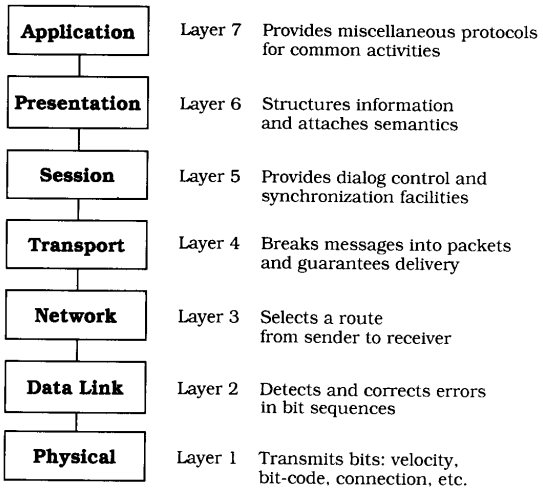
- From MUD to Structure…
  - Layers, Pipe and Filters, Blackboard
- Distributed Systems…
  - Broker, Pipe and Filters, Microkernel
- Interactive Systems…
  - MVC, PAC
- Adaptable Systems…
  - Microkernel, Reflection…

# Layer

- helps structure application that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.



# Layer: examples

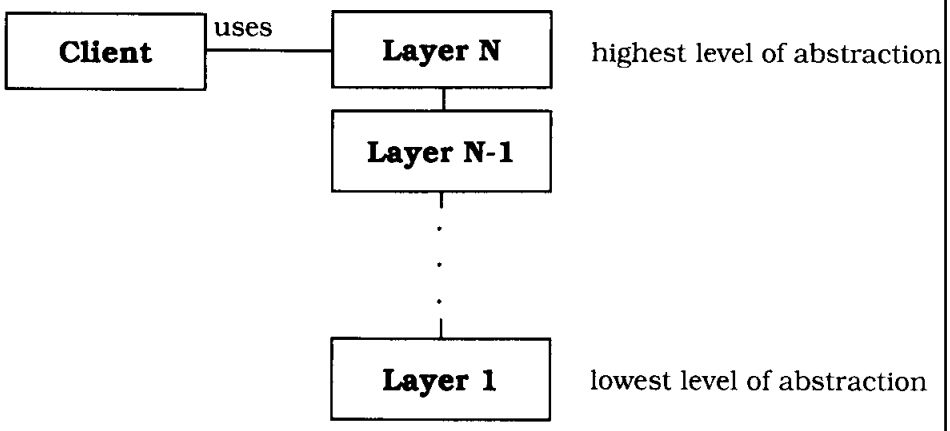


# Layer :Structure

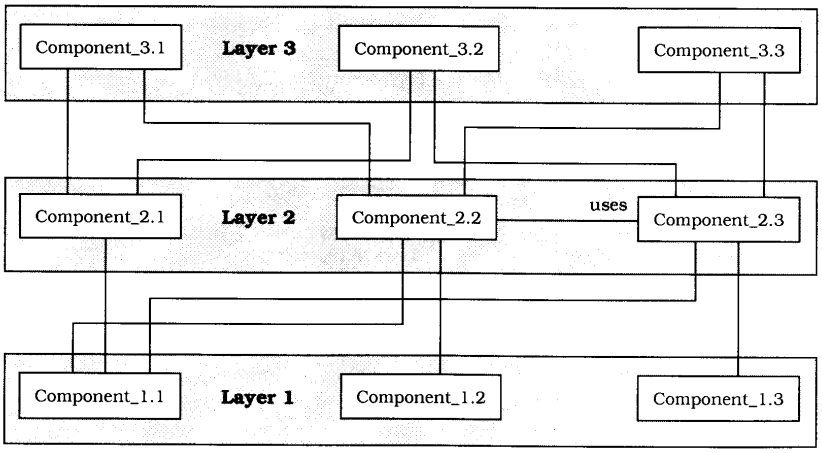
| ***Class*** Layer J | ***Collaborator*** • Layer J-1 |
| --- | --- |
| ***Responsibility*** • Provides services used by Layer J+1. • Delegates subtasks to Layer J-1. | |



# Layer: Structure

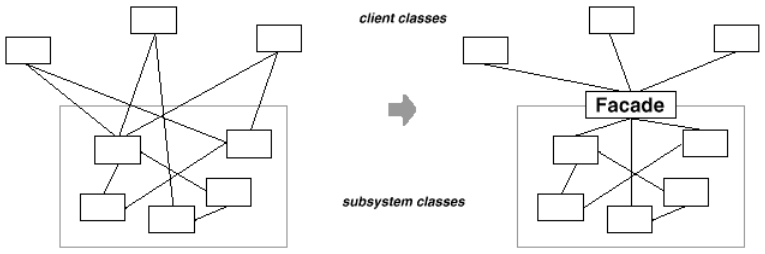


# Layer and components…

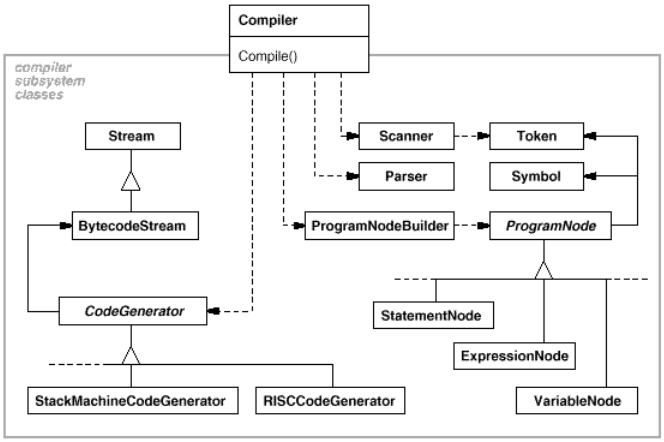





# Layer and Facade DP…



# Layer and Facade DP






# Layers : Variants

- Relaxed Layered System:
  - A layer « j » can use service of j-1, j-2…
  - A layer can be partially opaque
    - Some service to layer j+1, others to all upper services…
- Layering through inheritance:
  - Lower layers are implemented as base classes
  - Higher level can override lower level…

# Layers : Known Uses

- Virtual machines: JVM and binary code format
- API : Layer that encapsulates lower layers
- Information System
  - Presentation, Application logic, Domain Layer, Database
- Windows NT (relaxed for: kernel and IO and hardware)
  - System services,
  - Resource management (Object manager, security monitor, process manager, I/O manager, VM manager, LPC),
  - Kernel (exception handling, interrupt, multipro synchro, threads),
  - HAL (Hardware Abstraction Level)
  - Hardware



# Layers: benefits

- Reuse of layers
- Support for standardization (POSIX)
- Dependencies are kept local
- Exchangeabilities :
  - Replacement of old implementation with Adapter Pattern
  - Dynamic exchange with Bridge Pattern

## Layers: Liabilities

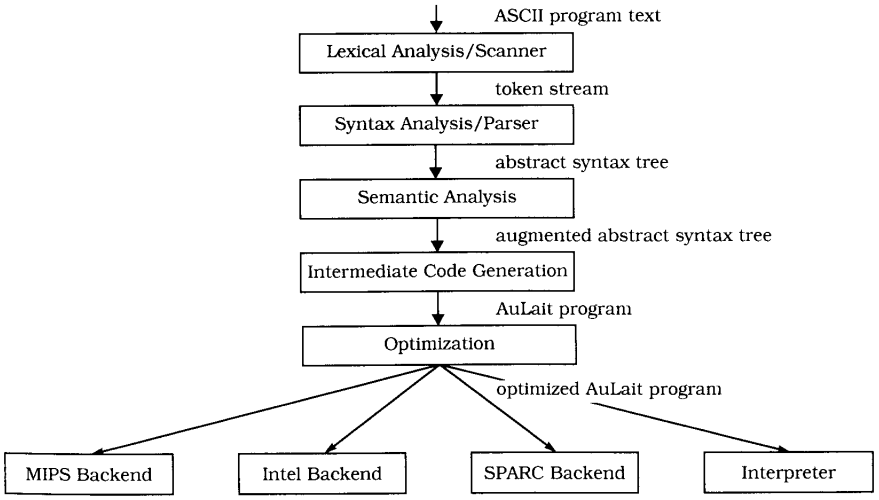
- Cascades of changing behavior
- Lower efficiency
- Unnecessary work: functions of a layer called many times for one service
- Difficulty of establishing correct granularity of layers: Too few layers -> less benefits, too many layers -> complexity and overhead…



## Pipes and Filters

- Provides a structure for systems that process a stream of Data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters.
- Recombining filters allows the building of families of related systems.

# Pipes and Filters: Example






# Pipes and Filters: Structure

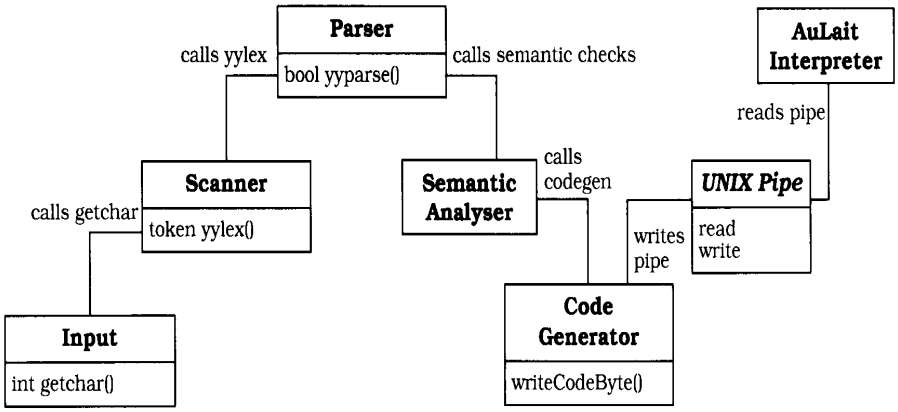
| ***Class*** Filter | ***Collaborators*** |
|---|---|
| ***Responsibility*** • Gets input data. • Performs a function on its input data. • Supplies output data. | • Pipe |

| ***Class*** Pipe | ***Collaborators*** |
|---|---|
| ***Responsibility*** • Transfers data. • Buffers data. • Synchronizes active neighbors. | • Data Source • Data Sink • Filter |

| ***Class*** Data Source | ***Collaborators*** |
|---|---|
| ***Responsibility*** • Delivers input to processing pipeline. | • Pipe |

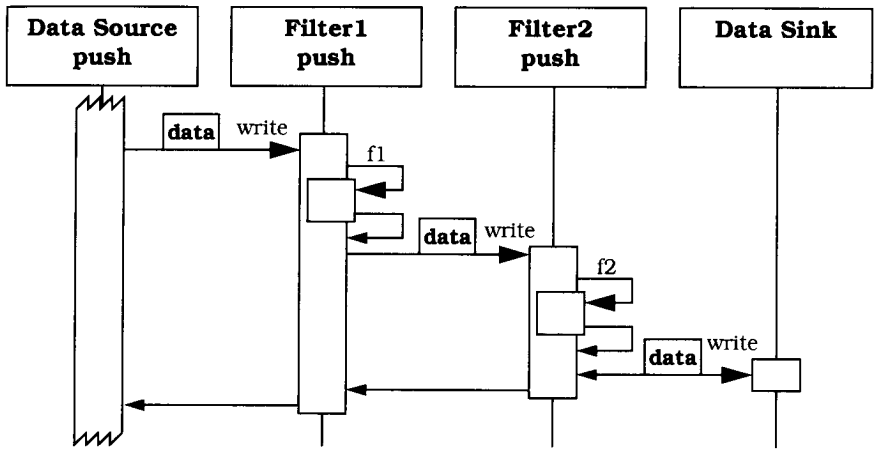
| ***Class*** Data Sink | ***Collaborators*** |
|---|---|
| ***Responsibility*** • Consumes output. | • Pipe |

# Pipes and Filters

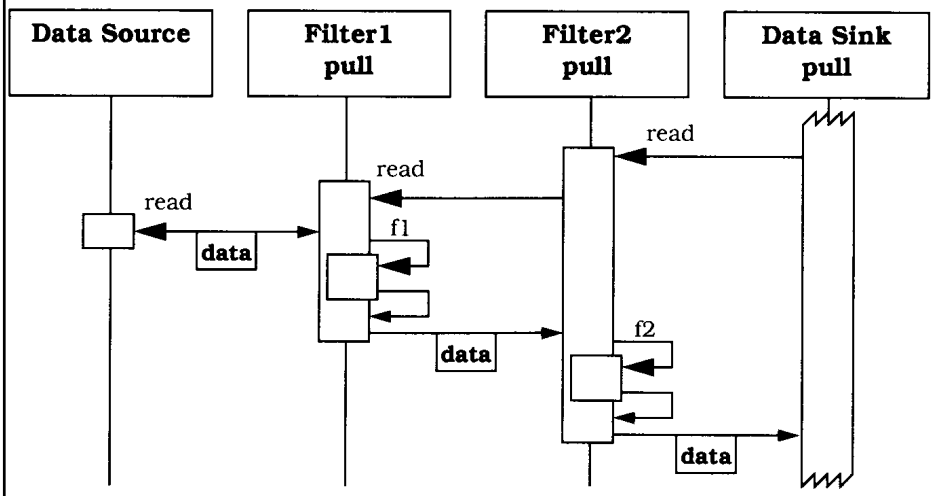





# Pipes and Filters: push pipeline

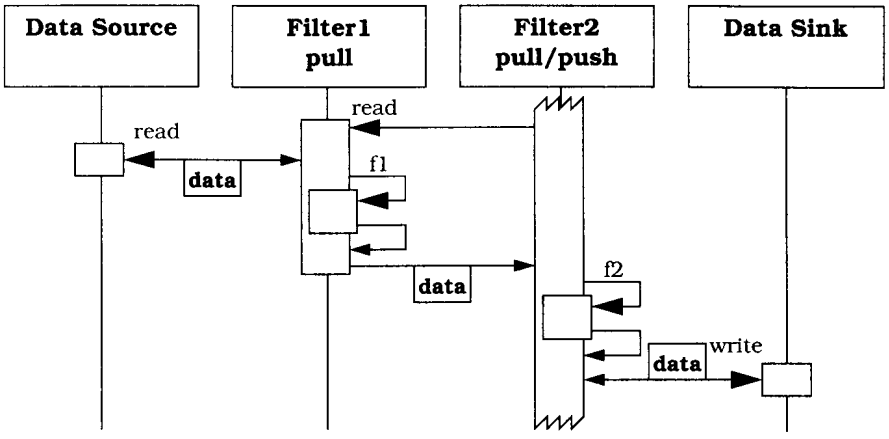


# Pipes and Filters: pull pipeline

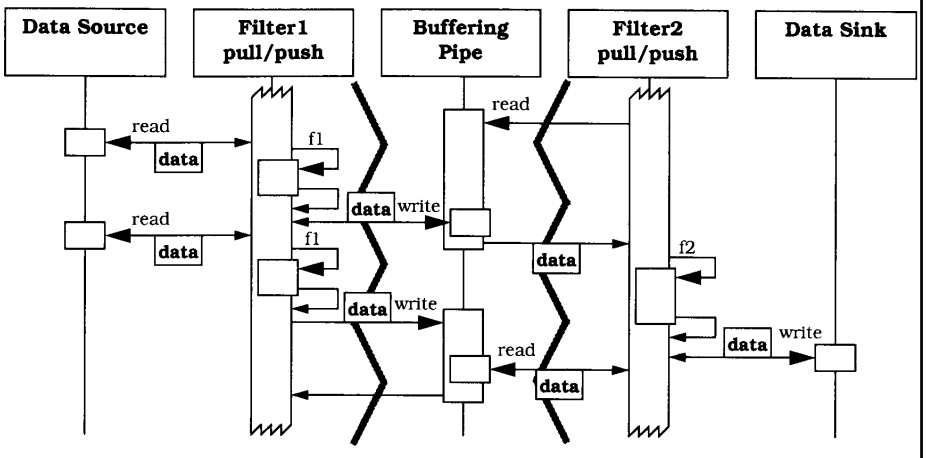





# Pipes and Filters: push-pull pipeline



# Pipes and Filters : Threaded Filters






# Pipes and Filters: Known Uses

- Unix
- CMS Pipelines (extension IBM mainframes)
- LASSPTools (Numerical Analysis)
  - Graphical input devices (knobs or sliders)
  - Filters for numerical analysis and data extraction
  - Data sinks to produce animation from numerical data streams…
- Khoros : Image recognition…
- WEB !! Servlet !!

## Pipes and Filters Benefits

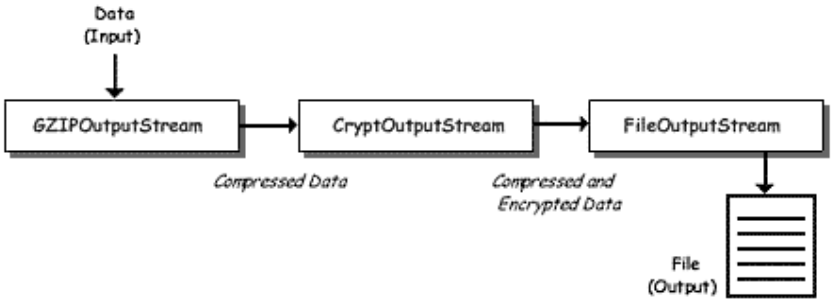
- No intermediate file necessary (but possible)
- Flexibility by filter exchange
- Flexibility by recombination
- Reuse of filter components
- Rapid prototyping of pipeline
- Efficiency by parallel processing



## Pipes and Filters Liabilities

- Sharing state information is expensive or inflexible
- Efficiency gain by parallel processing is often an illusion
  - Cost of data transfer, filters that consume all data before one output, context switch on one computer, synchronization of filters via pipes
- Data transformation overhead
- Error Handling

[Sun Developpers]

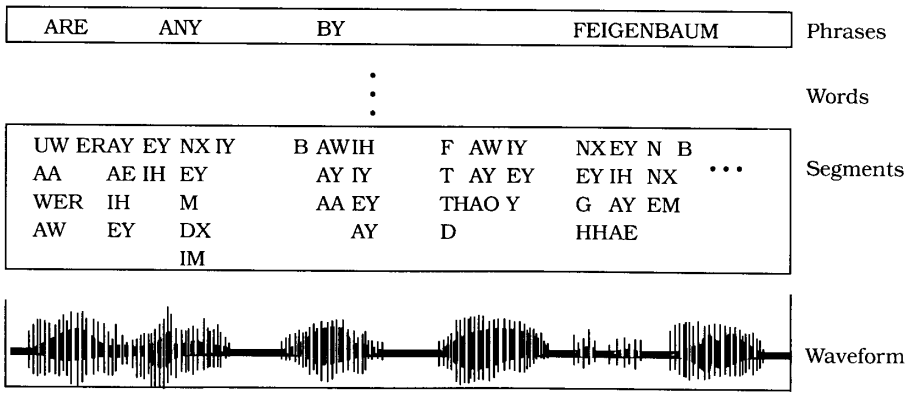


# Blackboard

*The Blackboard architectural pattern is useful for problems for which no deterministic solution strategies are known. Several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.*



# Blackboard Example



# Blackboard Structure

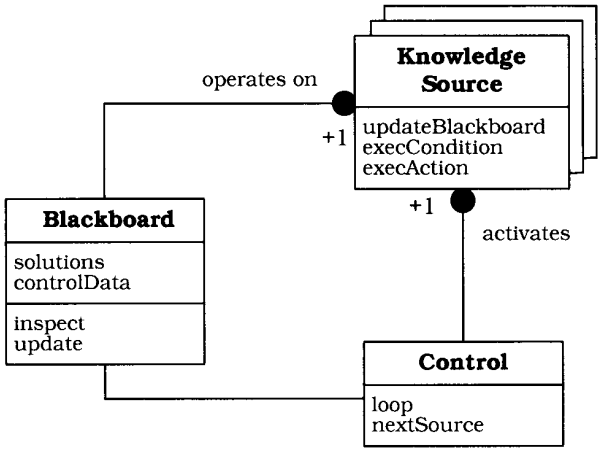
| Class | Collaborators |
|---|---|
| Blackboard | - |
| **Responsibility** | |
| • Manages central data | |

| Class | Collaborator |
|---|---|
| Knowledge Source | • Blackboard |
| **Responsibility** | |
| • Evaluates its own applicability<br>• Computes a result<br>• Updates Blackboard | |

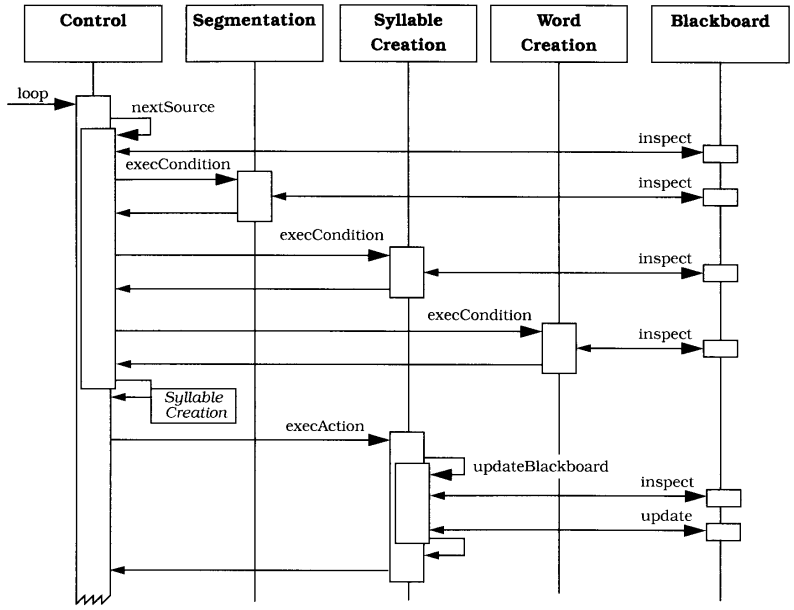
| Class | Collaborators |
|---|---|
| Control | • Blackboard<br>• Knowledge Source |
| **Responsibility** | |
| • Monitors Blackboard<br>• Schedules Knowledge Source activations | |



# Blackboard Structure



# Blackboard Structure






# Blackboard Variants

- Production System (OPS Language)
  - Blackboard : working memory
  - Knowledge source: Condition-action rules
  - Control: conflict resolution module.
- Repository:
  - blackboard: Data,
  - Application program: knowledge source.
  - Control: user input, external program

# Blackboard known uses

- HEARSAY-II: Speech recognition
- HASP/SIAP: detect enemy submarine
- Crysalis: infer three-dimensional structure of protein molecule from X-Ray diffraction Data.
- Tricero: Aircraft activities. Extend blackboard to distributed computing



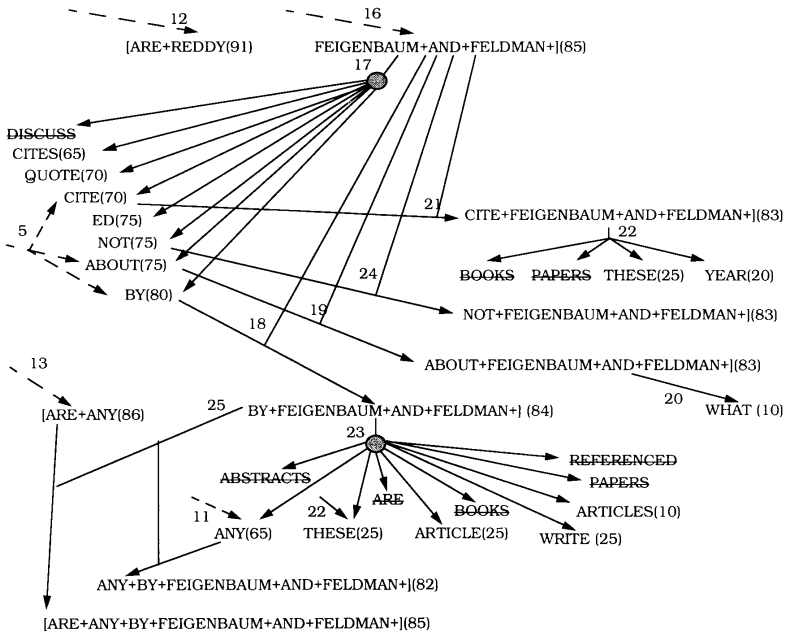
# Blackboard benefits

- Experimentation: different algo, different control heuristics
- Changeability and maintainability: separation data/control.
- Reusable knowledge source
- Support for Fault tolerance and robustness: Tolerance of noisy data…

# Blackboard Liabilities

- Difficulty of testing: no deterministic algo
- No good solution is guaranteed.
- Difficulty of establishing a good control strategy
- Low efficiency: (rejecting wrong hypothesis)
- High development effort : trial-and-error programming
- No support for parallelism



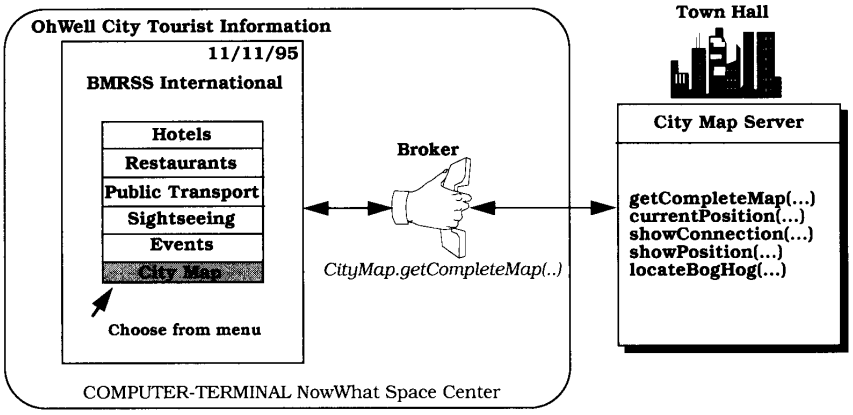


# Broker

- Used to structure distributed software systems with decoupled components that interact by remote service invocation.
- A broker component is responsible for coordinating communication, such as forwarding request, as well as for transmitting result and exception.



# Broker example



# Broker structure

| Class | Collaborators |
|---|---|
| Broker | • Client<br>• Server<br>• Client-side Proxy<br>• Server-side Proxy<br>• Bridge |
| **Responsibility**<br>• (Un-)registers servers.<br>• Offers APIs.<br>• Transfers messages.<br>• Error recovery.<br>• Interoperates with other brokers through bridges.<br>• Locates servers. | |

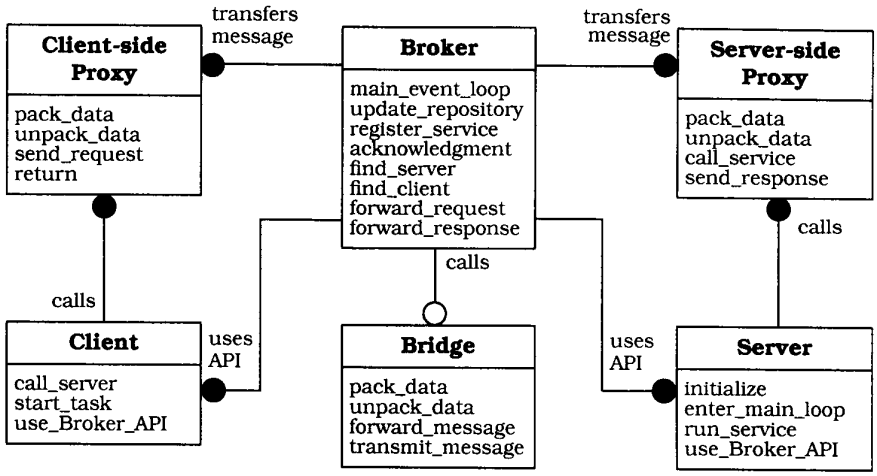
| Class | Collaborators |
|---|---|
| Client-side Proxy | • Client<br>• Broker |
| **Responsibility**<br>• Encapsulates system-specific functionality.<br>• Mediates between the client and the broker. | |

| Class | Collaborators |
|---|---|
| Server-side Proxy | • Server<br>• Broker |
| **Responsibility**<br>• Calls services within the server.<br>• Encapsulates system-specific functionality.<br>• Mediates between the server and the broker. | |

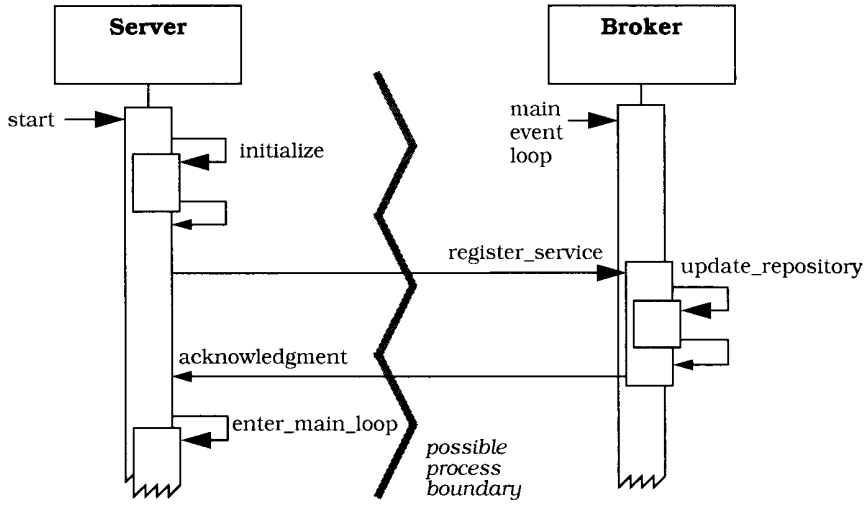
| Class | Collaborators |
|---|---|
| Bridge | • Broker<br>• Bridge |
| **Responsibility**<br>• Encapsulates network-specific functionality.<br>• Mediates between the local broker and the bridge of a remote broker. | |



# Broker Structure

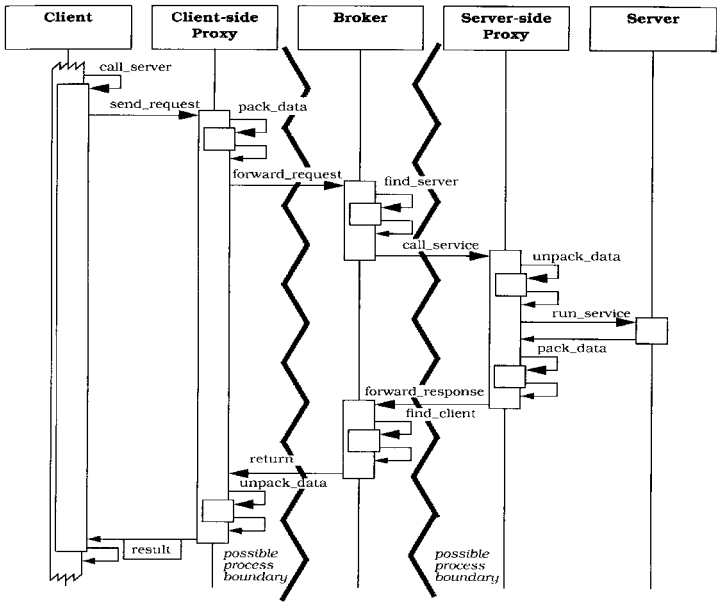


# Broker Structure

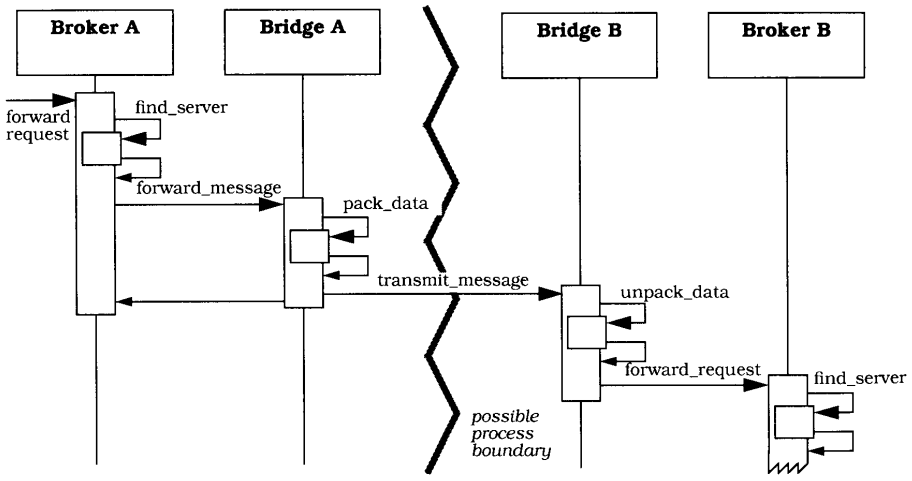





# Broker Structure



# Broker Structure






# Broker Variants

- Direct Communication Broker System:
  - Direct link to server
- Message Passing Broker System
  - Focus on transmission of data. Type of the message determine the behavior of the broker…
- Trader System :
  - service identifiers are used to access server functionality. Request can be forwarded to more than one server…
- Callback broker system: event driven…

## Known Uses

- CORBA
- IBM SOM/DSOM
- Microsoft Ole 2.x
- WWW
- ATM-P: Message passing broker. Telecommunication switching system based on ATM.



## Broker benefits

- Location transparency
- Changeability and extensibility of components
- Portability of a broker system (Layered)
- Interoperability between brokers (bridge)
- Reusability (of services)

## Broker Liabilities

- Restricted efficiency (indirection layer)
- Lower Fault tolerance: fault a broker or a server… replication of components…
- Testability:
  - Of components (benefits)
  - Of application (liabilities)



## Model-View-Contoler (MVC)

- The model contains the core functionality and data?
- Views display information to the user.
- Controllers handle user input.
- A change propagation mechanism ensure consistency between user interface and the model.

# MVC

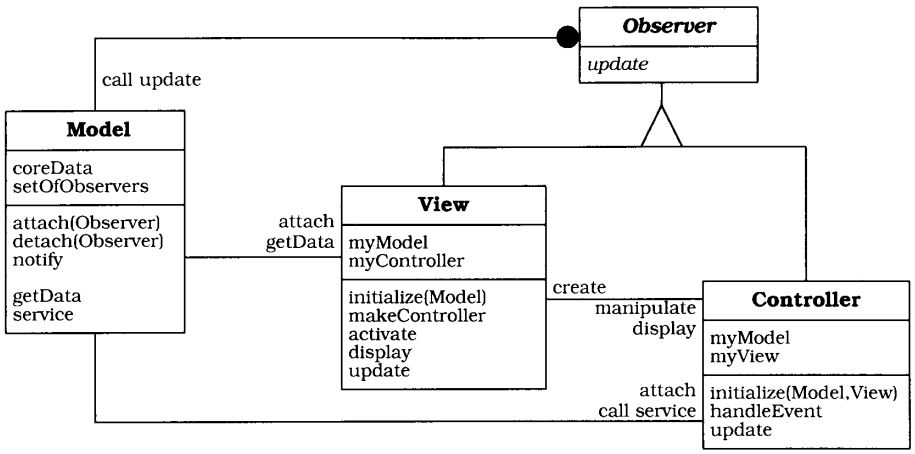
| *Class* Model | *Collaborators* |
|---|---|
| ***Responsibility*** • Provides functional core of the application. • Registers dependent views and controllers. • Notifies dependent components about data changes. | • View • Controller |

| *Class* View | *Collaborators* |
|---|---|
| ***Responsibility*** • Creates and initializes its associated controller. • Displays information to the user. • Implements the update procedure. • Retrieves data from the model. | • Controller • Model |

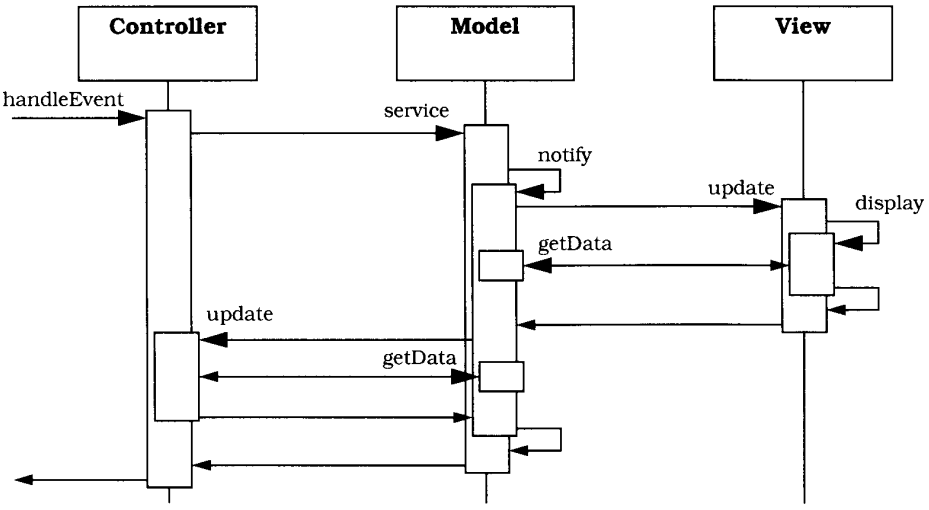
| *Class* Controller | *Collaborators* |
|---|---|
| ***Responsibility*** • Accepts user input as events. • Translates events to service requests for the model or display requests for the view. • Implements the update procedure, if required. | • View • Model |



# MVC Structure

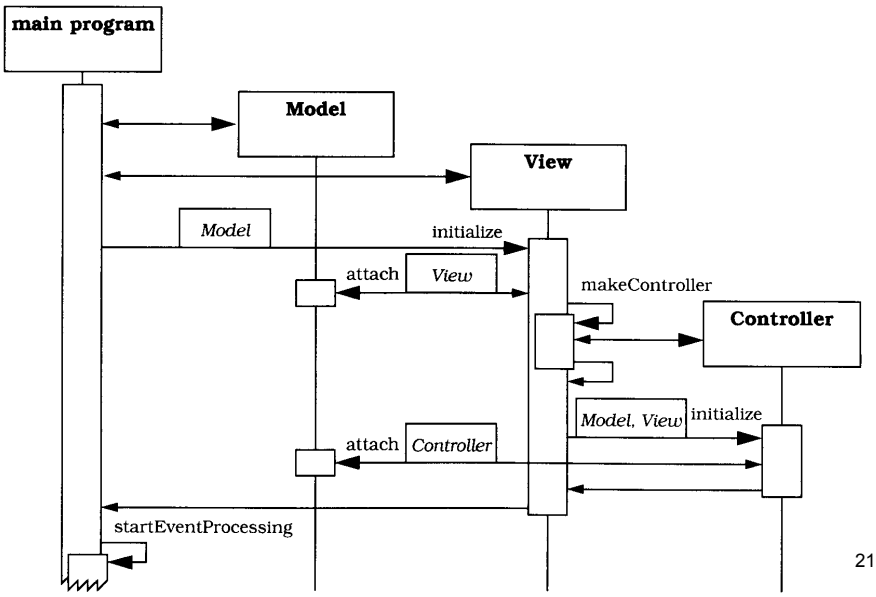


# MVC Structure






# MVC Structure



## MVC Known Uses

- Smalltalk
- MFC
- ET++: application Framework
- Java/Swing



## MVC benefits

- Multiple views of the same model
- Synchronized views: change propagation
- Pluggable views and controllers
- Exchangeability of ‘look and feel’
- Framework potential

# MVC Liabilities

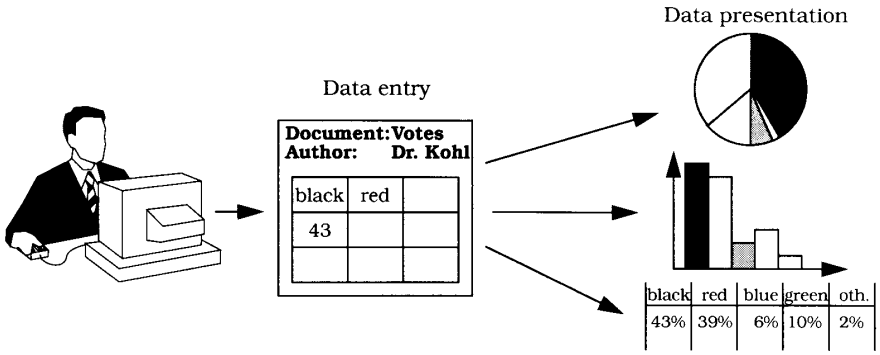
- Increased complexity
- Potential for excessive number of updates
- Intimate connection between view and controller
- Close coupling of views and controllers to a model
- Inefficiency of data access in view
- Inevitability of change to view and controller when porting
- Difficulty of using MVC with modern user-interface tools



# Presentation-Abstraction-Control

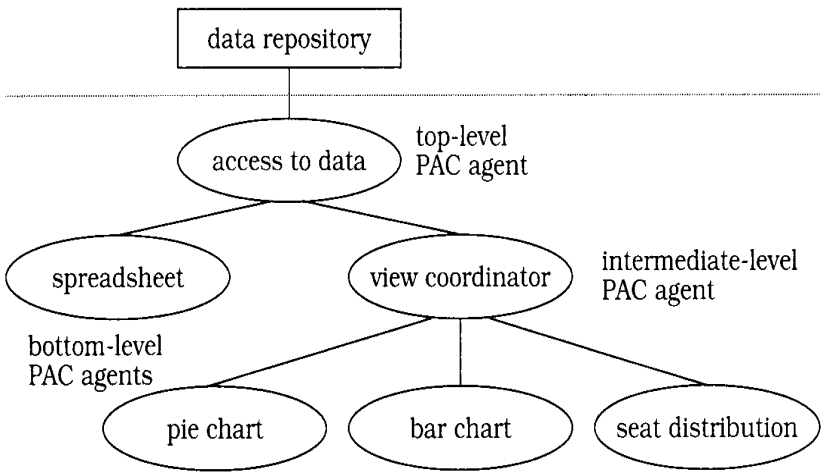
- PAC define a hierarchy of cooperating agents.
- Each agent consists of three components: presentation, abstraction, control.
- Separates human computer interaction from its functional core and its communication with other agents…

# PAC Example






# PAC Example



# PAC Structure

| Class | Collaborators |
|---|---|
| Top-level Agent | • Intermediate-level Agent<br>• Bottom-level Agent |
| **Responsibility** | |
| • Provides the functional core of the system.<br>• Controls the PAC hierarchy. | |

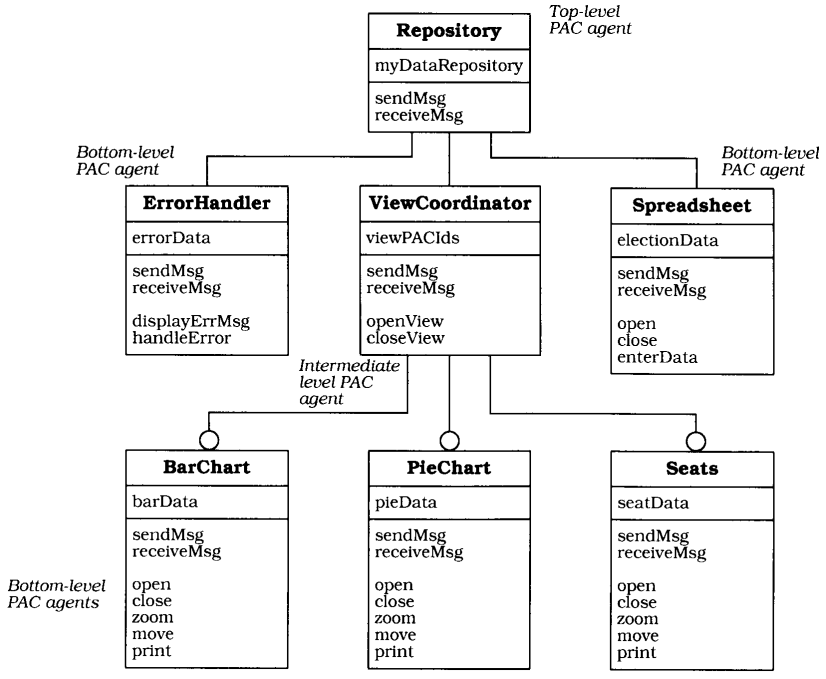
| Class | Collaborators |
|---|---|
| Interm. -level Agent | • Top-level Agent<br>• Intermediate-level Agent<br>• Bottom-level Agent |
| **Responsibility** | |
| • Coordinates lower-level PAC agents.<br>• Composes lower-level PAC agents to a single unit of higher abstraction. | |

| Class | Collaborators |
|---|---|
| Bottom-level Agent | • Top-level Agent<br>• Intermediate-level Agent |
| **Responsibility** | |
| • Provides a specific view of the software or a system service, including its associated human-computer interaction. | |



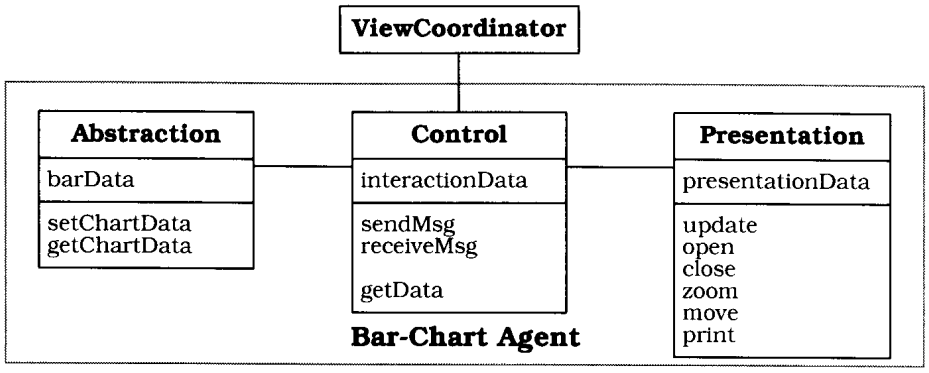
# Top Level PAC

- Abstraction : Global Data model
- Presentation : Some Graphical elements
- Control:
  - Allow sub-agent to access abstraction
  - Manage hierarchy of PAC component
  - Manage info about interaction (log, check applicability of triggered application…

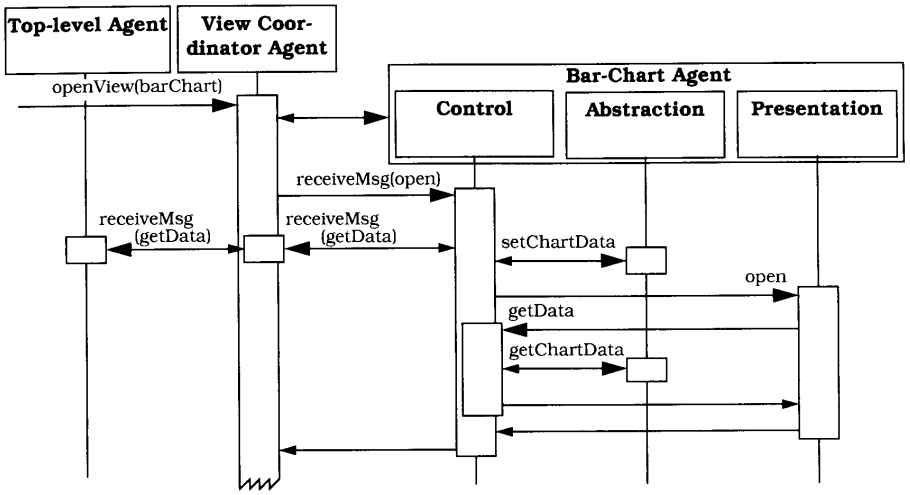





# PAC Structure

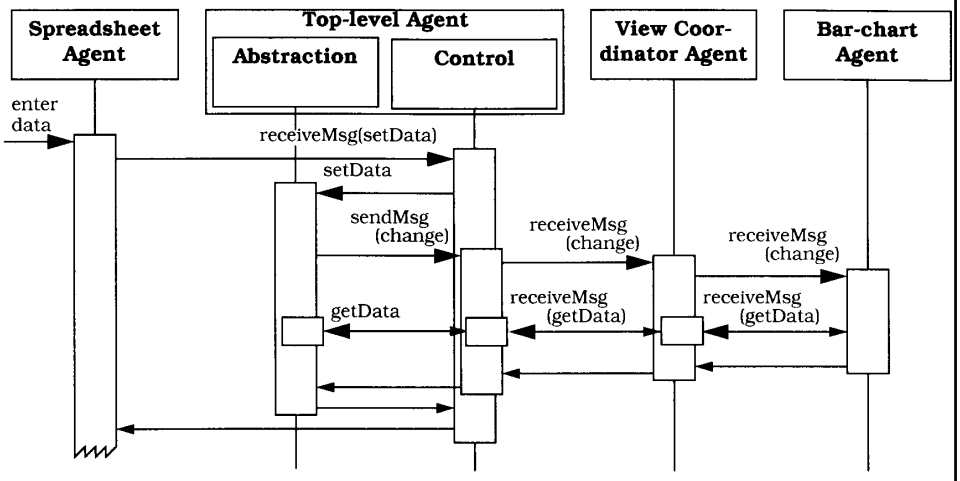


# PAC Structure

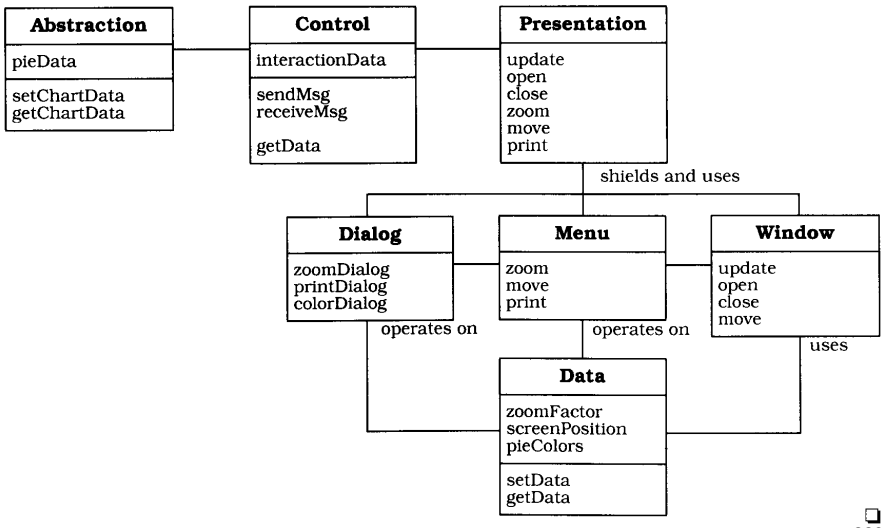





# PAC Structure



# PAC Structure






# PAC Known Uses

- Network Trafic Management (TS93)
  - Gathering traffic data
  - Threshold checking and generation exceptions
  - Logging and routing of network exception
  - Vizualisation of traffic flow and network exceptions
  - Displaying various user-configurable views of the whole network
  - Statistical evaluation of traffic data
  - Access to historic traffic data
  - System administration and configuration

## PAC Benefits

- Separation of concerns: Agent and inside an agent
- Support for change and extension
- Support for multi-tasking: each PAC agent can run its own thread on a different computer…



## PAC Liabilities

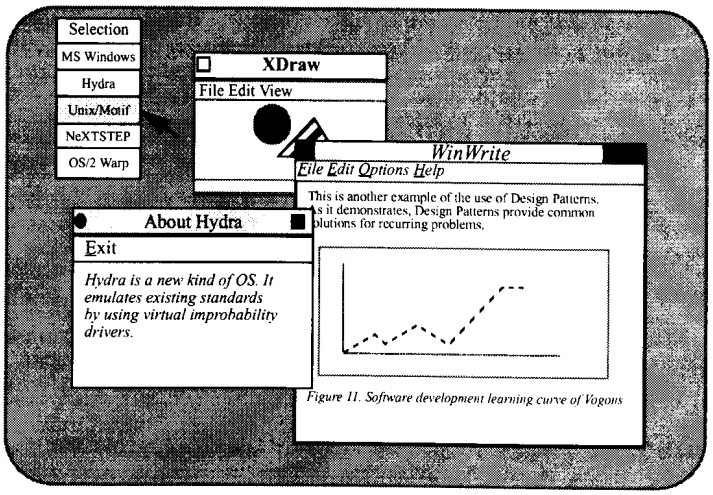
- Increased system complexity: Coordination of agents…
- Complex control component: coordonate action inside agent and with other agents…
- Efficiency : data are propagated throught the tree…
- Applicability : Not a graphic editor where each object is a PAC agent…

# Microkernel

- Applies to software systems that be able to adapt to changing system requirements.
- It separates a minimal functional core from extended functionality and customer specific parts.
- The Microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.



# Microkernel



# Microkernel Architecture

| Class: Microkernel | Collaborators |
|---|---|
| **Responsibility**<br>• Provides core mechanisms.<br>• Offers communication facilities.<br>• Encapsulates system dependencies.<br>• Manages and controls resources. | • Internal Server |

| Class: Internal Server | Collaborators |
|---|---|
| **Responsibility**<br>• Implements additional services.<br>• Encapsulates some system specifics. | • Microkernel |

| Class: External Server | Collaborators |
|---|---|
| **Responsibility**<br>• Provides programming interfaces for its clients. | • Microkernel |

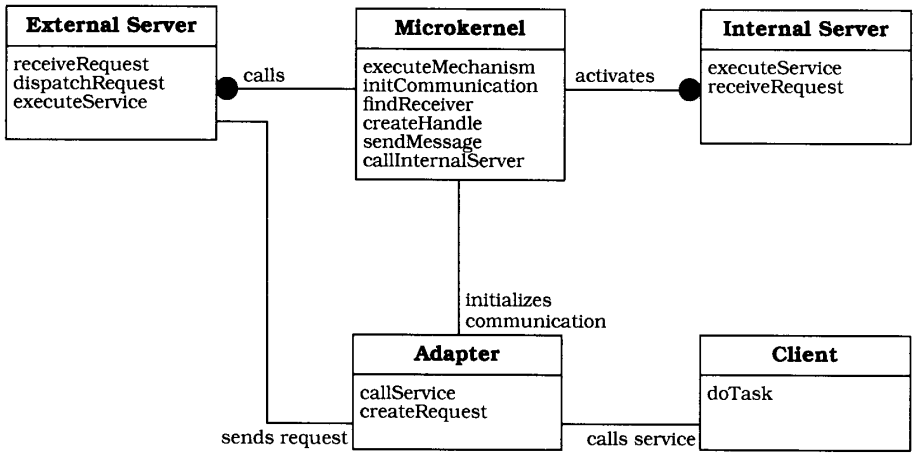


# Microkernel Architecture

| Class: Client | Collaborators |
|---|---|
| **Responsibility**<br>• Represents an application. | • Adapter |

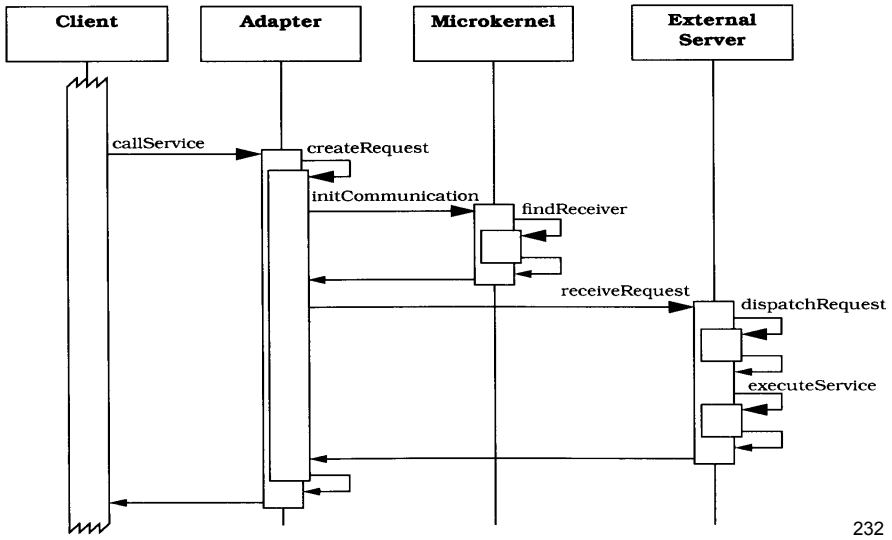
| Class: Adapter | Collaborators |
|---|---|
| **Responsibility**<br>• Hides system dependencies such as communication facilities from the client.<br>• Invokes methods of external servers on behalf of clients. | • External Server<br>• Microkernel |

# Microkernel Architecture

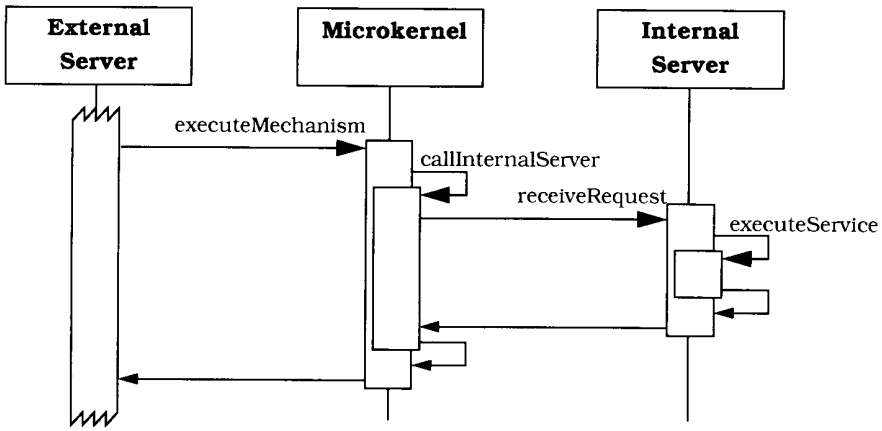





# Microkernel Structure



# Microkernel Structure






# Microkernel variants

- Microkernel system with indirect Client-Server connections. MK establish channel of communication between client and external servers.

# Microkernel known Uses

- Mach (92): Emulate other operating system (NeXTSTEP)
- Amoeba (92):
  - Kernel: process, threads system memory, communication, IO
  - Services not in the kernel are internal servers..



# Known uses

- Chorus
- WINDOWS NT:
  - External servers: OS/2.1.X, posix server and win32 server
- MKDE: Microkernel Databank Engine
  - External server : Data model of SQL database

# Microkernel Benefits

- Portability : no need to port external servers…
- Flexibility and extensibility
- Separation of policy and mechanism:
  - Mechanism in kernel, policy in external servers
- Scalability
- Reliability: Distributed Microkernel… :-/
- Transparency : Microkernel ~ broker…



# Microkernel Liabilities

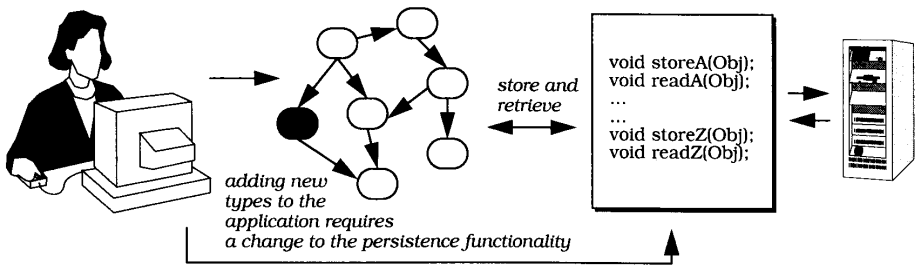
- Performance
- Complexity of design and implementation.
  - Basic functionalities of the micro-kernel ??
  - Separation mechanism/policy => deep knowledge of domain.

# Reflection

- Provides a mechanism for changing structure and behavior of software dynamically.
- Support modification of fundamental aspects: type structures and function call mechanism
- Meta-level makes the software self-aware
- Base-level includes application logic. Its implementation builds on the meta-level.



# Reflection example



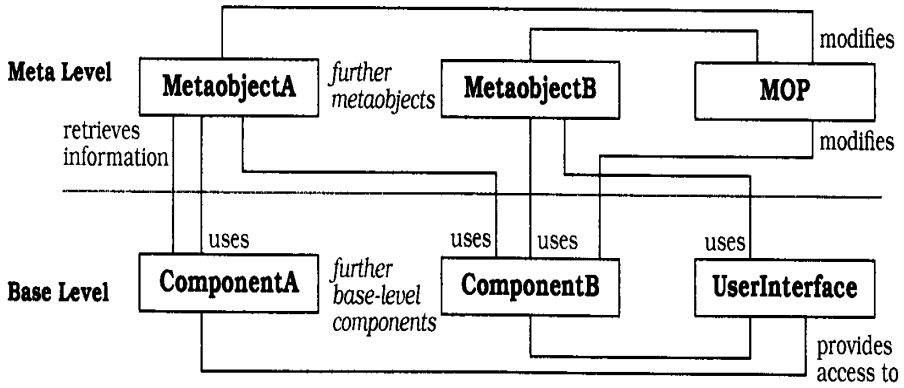
| Class<br>Base Level | Collaborators<br>• Meta Level |
|---|---|
| **Responsibility**<br>• Implements the application logic.<br>• Uses information provided by the meta level. | |

| Class<br>Meta Level | Collaborators<br>• Base Level |
|---|---|
| **Responsibility**<br>• Encapsulates system internals that may change.<br>• Provides an interface to facilitate modifications to the meta-level. | |

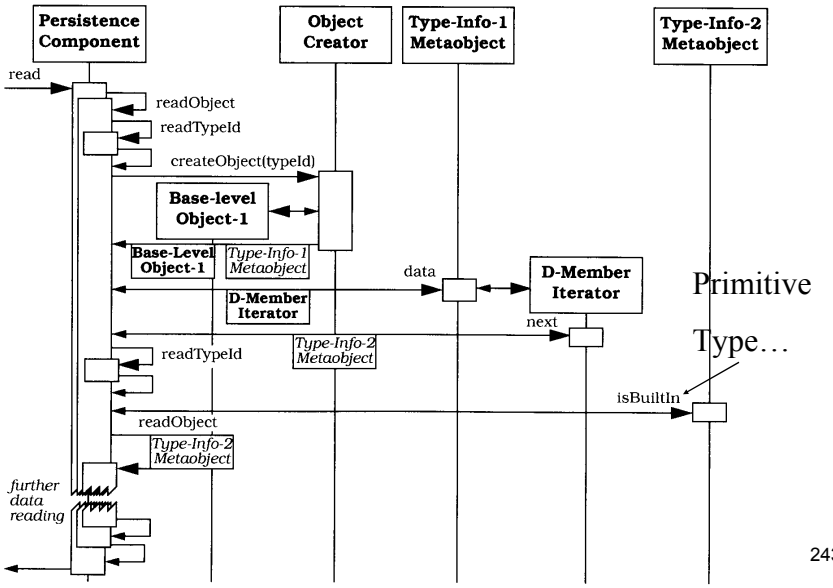
| Class<br>Metaobject Protocol | Collaborators<br>• Meta Level<br>• Base Level |
|---|---|
| **Responsibility**<br>• Offers an interface for specifying changes to the meta level.<br>• Performs specified changes | |



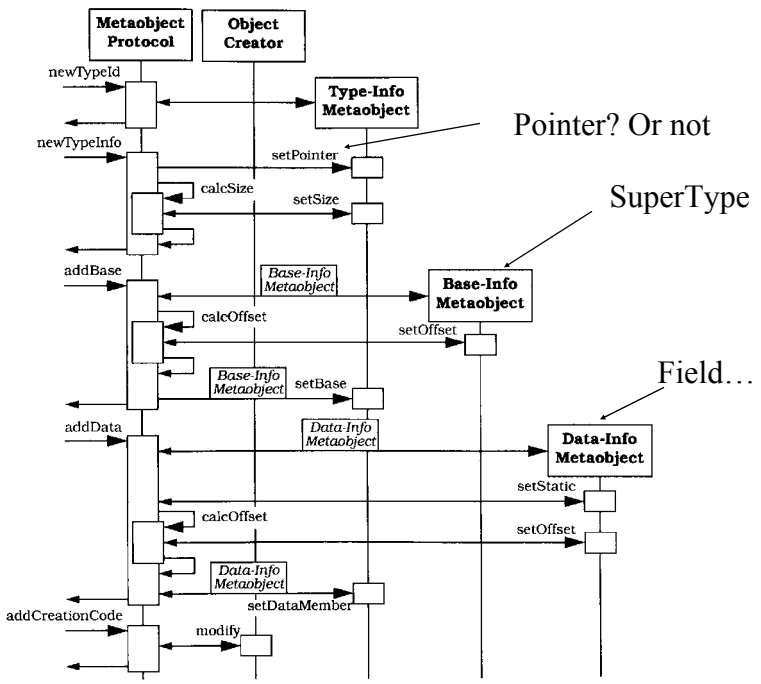
# Reflection structure



# Reflection example





# Reflection known Uses

- CLOS : generic function and generic function invocation
- MIP: run-time type information system for C++
- Pgen: persistence component for C++ based on MIP
- Ole2.0, CORBA (dynamic invocation)…



# Reflection benefits

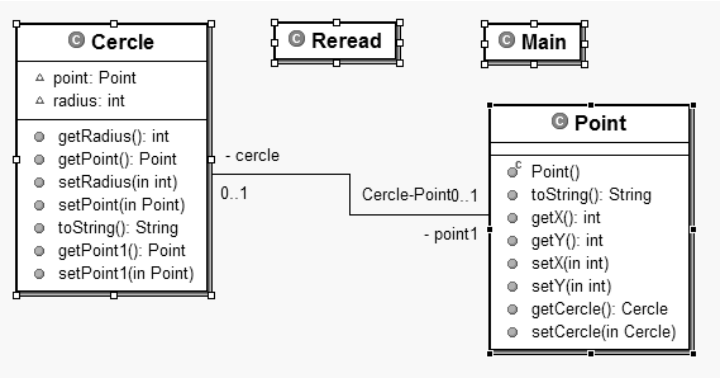
- No explicit modification of source code
- Changing a software is easy: no need for visitors, factories and strategies patterns
- Support for many kind of change

# Reflection Liabilities

- Modification at the meta-level can cause damage.
- Increased number of component
- Lower efficiency
- Not all potential changes supported (only those supported by the MOP)
- Not all languages support reflection



# Reflection example



# Reflection example

```
public class Main {
public static void main(String args[]) throws Exception {
Point p = new Point();
p.setX(3);
p.setY(4);
Cercle c = new Cercle();
c.setPoint(p);
c.setRadius(6);
XMLEncoder e = new XMLEncoder(new BufferedOutputStream(new
FileOutputStream(args[0])));
e.writeObject(c);
e.close();
System.out.println(c);
}
}
```



# Reflection example

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_03" class="java.beans.XMLDecoder">
 <object class="Cercle">
  <void property="point">
   <object class="Point">
    <void property="x">
     <int>3</int>
    </void>
    <void property="y">
     <int>4</int>
    </void>
   </object>
  </void>
  <void property="radius">
   <int>6</int>
  </void>
 </object>
</java>
```

# Reflection example

```
public class Reread {
public static void main(String args[])
throws Exception {
XMLDecoder d = new XMLDecoder(new
          BufferedInputStream(new
          FileInputStream(args[0])));
Cercle c = (Cercle)d.readObject();
d.close();

System.out.println(c);
}
}
```



# Summary (C. Alexander)

- It is possible to build an architecture by stringing together patterns, in a rather loose way. A building made like this, is an assembly of patterns. It is not dense. It is not profound. But it is also possible to put patterns together in such way that many patterns overlap in the same physical space: the building is very dense; it has many meanings captured in a small space; and through this density, it becomes profound.

# Drawbacks of Patterns

- Patterns do not lead to direct code reuse.
- Individual Patterns are deceptively simple.
- Composition of different patterns can be very complex.
- Teams may suffer from pattern overload.
- Patterns are validated by experience and discussion rather than by automated testing.
- Integrating patterns into a software development process is a human-intensive activity.