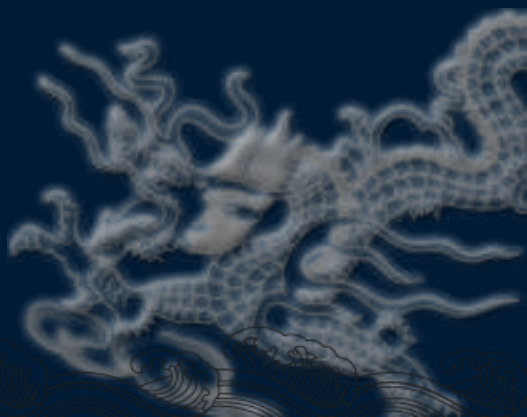
Design patterns



Why Patterns?

- What's New Here is that Nothing is New Here
- Patterns are about what *works*
- Patterns give us a way to *talk* about what works
- Patterns distill *experience*
- Patterns give us a pithy design *vocabulary*

Alexander on Patterns

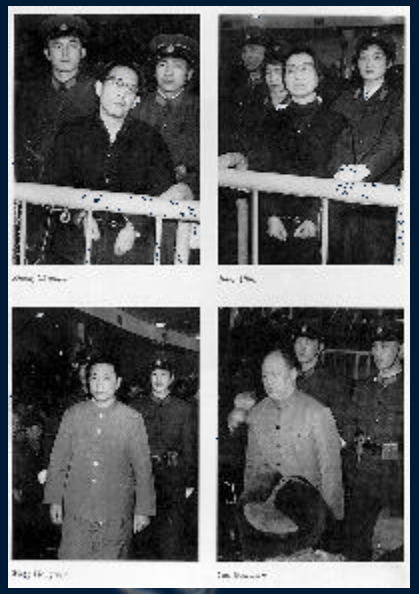
- Patterns in solutions come from patterns in problems.
- "A pattern is a solution to a problem in a context."
- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
- Christopher Alexander -- *A Pattern Language*

Design Patterns

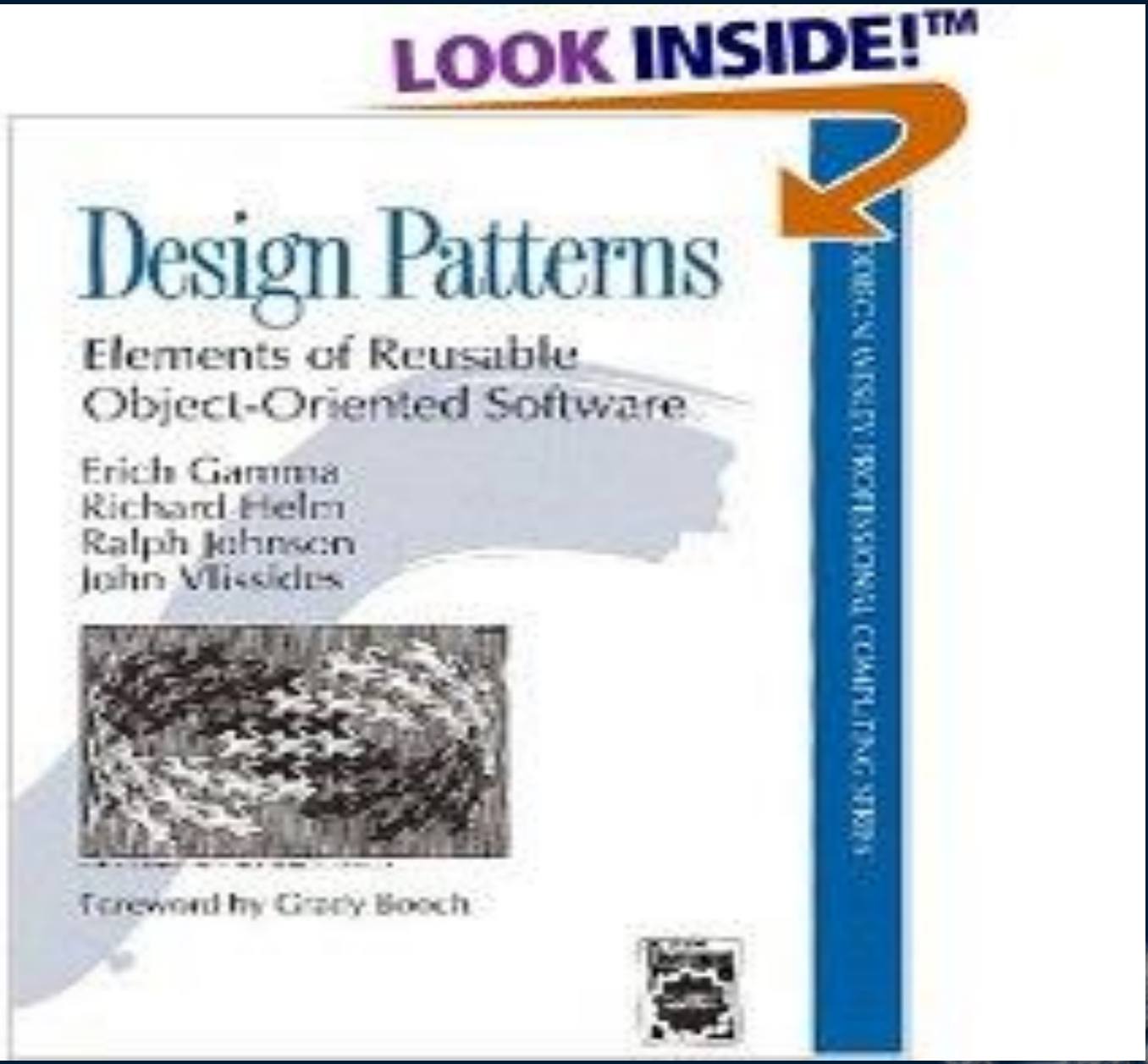
- 過去，在實做各種各類的軟體過程中，許多人累積了寶貴的物件導向分析經驗。經過蒐集整理，這些寶貴的繼承架構，class diagram，物件互動架構等等，被蒐集成所謂的 design patterns。
- design patterns are one kind of reuse, but not code reuse. Instead, it is a kind of pattern reuse.

The Gang of Four: The Essential Field Guide

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



- A landmark book that changed the way programmers think about building object-oriented programs


LOOK INSIDE!™
Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch

GoF Design Patterns

Creational patterns

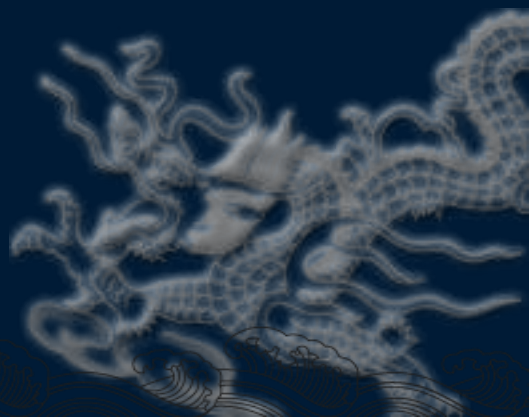
- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor



The Big Five



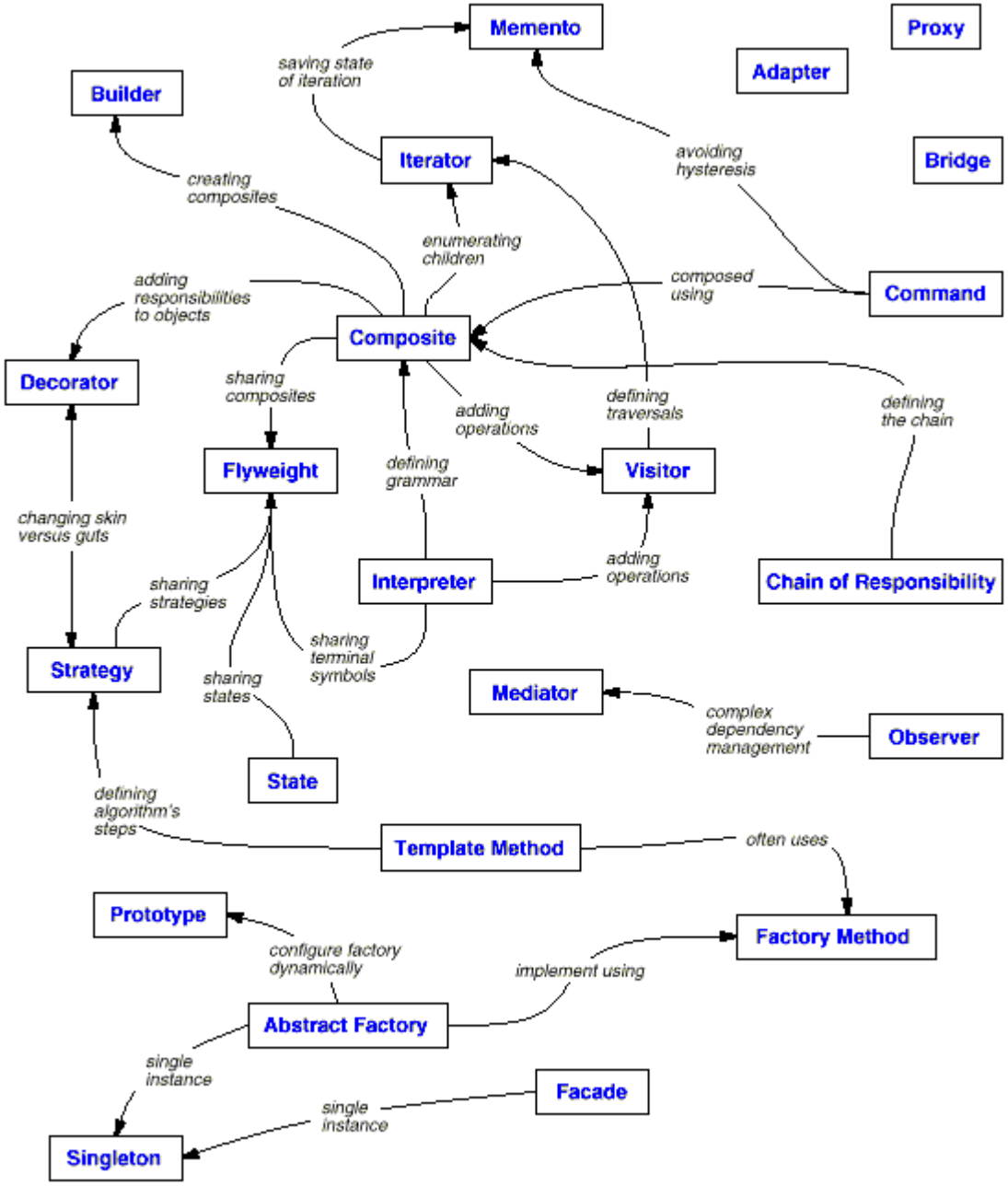


- Lion
- Leopard
- Elephant
- Buffalo
- Rhino

- Composite
- Proxy
- Strategy
- Observer
- Visitor

Design patterns essentiels

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)



Design pattern relationships

Practice of Design Pattern

- In order to achieve flexibility, design patterns usually introduce additional levels of indirection, which in some cases may complicate the resulting designs and hurt application performance.
- Often, people only understand how to apply certain software design techniques to certain problems[*citation needed*]. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that does not require specifics tied to a particular problem.

Criticism

- **Targets the wrong problem**
- The need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. Paul Graham writes in the essay Revenge of the Nerds.
- Peter Norvig provides a similar argument. He demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan

Criticism

- **Leads to inefficient solutions**
- The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern.

Criticism

- Passionate (Patterners) force their code into this pattern and that pattern they typically miss the fact that they are more often than not INCREASING complexity rather than following Fowler's first rule of KISS (keep it simple and stupid). Other than that they seem like genuinely intelligent people. Go figure.
- Design pattern make code obscure and difficult to understand
- "The OVERUSE is worst thank NO USE it at all“
- Is important to understand and use all these things, but more important is how and where to use them

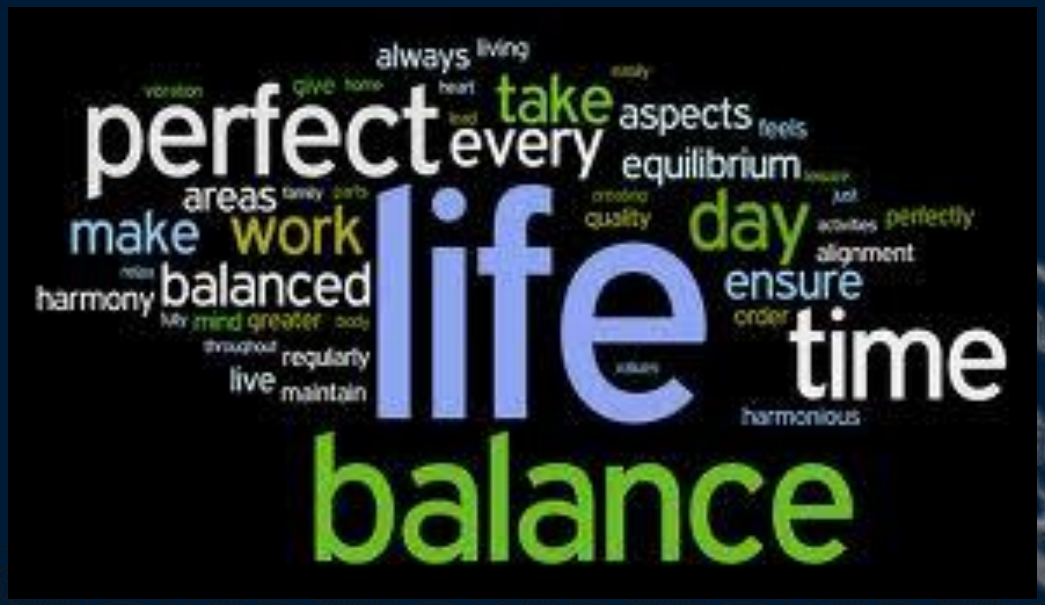
Criticism

- I would also like to add that sometimes extensive use of design patterns is overkill. Some problems, especially if the requirements and future change is very predictable and stable, can be solved very easily without employing design patterns. It's also fair to say that once a developer/designer is fluent in the use of design patterns he/she can incorporate them into a solution as easily as using any other design construct. But this doesn't mean that the next guy who comes along and is going to maintain that project or make some changes to it has the same intimacy with design patterns as the first designer therefore we have another overkill. To sum up, design patterns should be used wisely and where needed and like any other tool that we have at our disposal if you try to fix everything using design patterns you're going to end up in a big mess.

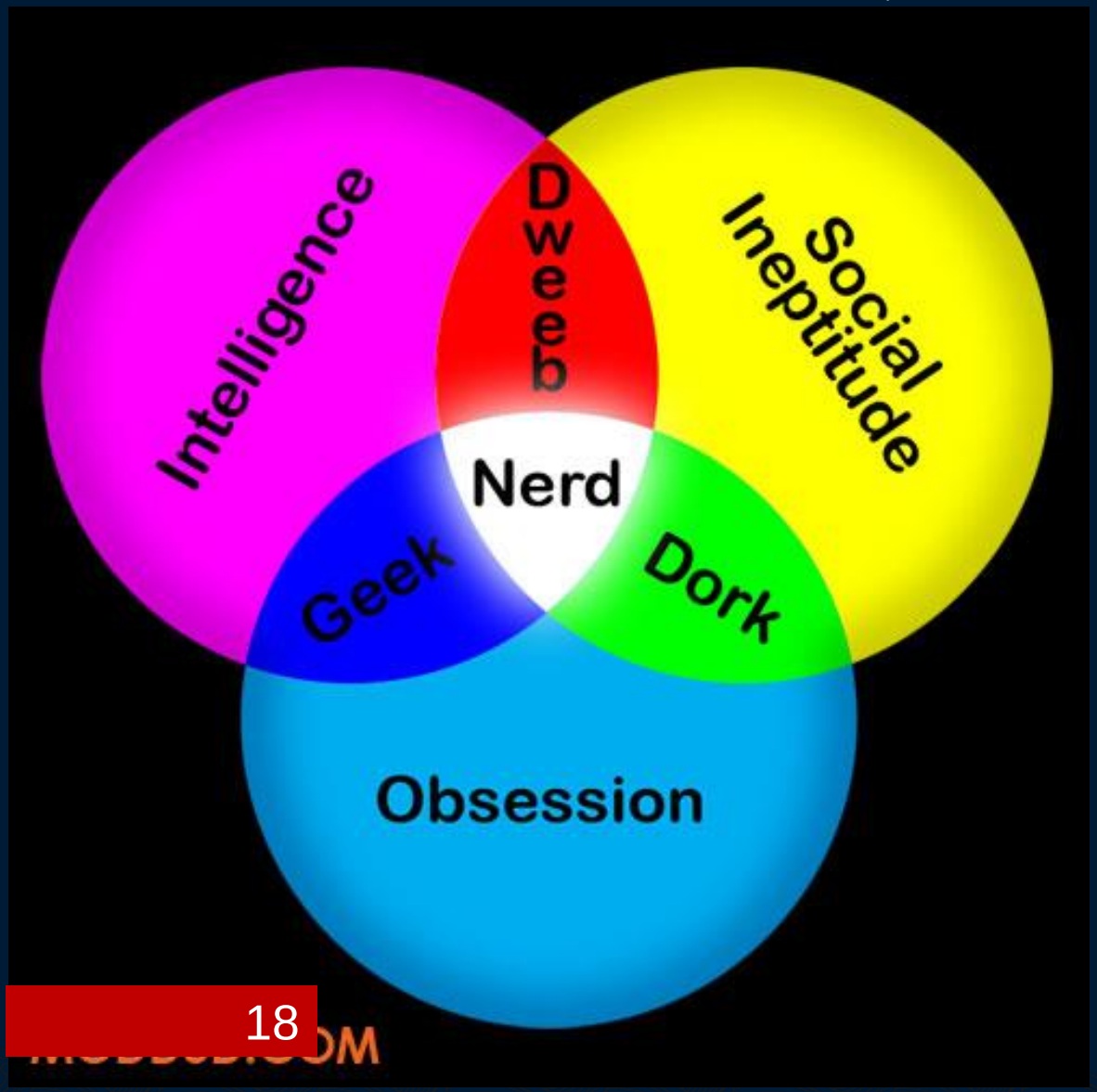
My Comments

In real world

- As a programmer
  - You have deadline
  - you have so many features to catch on
  - You have so many bugs to fix
  - acceptable, usable v.s. best, optimal

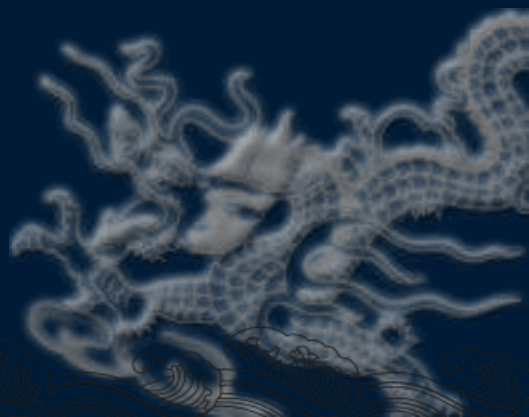


Do not turn yourself into



My Words

- SE, OO are meant to solve software development problem in practical world
- The real world is **dirty**
- **It is impractical to build a pure, perfect, optimal software in practice.**
- **In human history, software evolve and advance gradually.**
- every thing has a price (cost)



Meet the ugly truth

```
if (type == "wop") {
            assert(false);
            xp.type = T_WOP ;
            xp.wop_nodetype = this ;
            xp.val_wop = _wop;
    } else if (type == "varname") {
            xp.type = T_STRING ;
            xp.val_string = _wop->getName();
    } else if (type == "int*") {
            xp.type = TP_INT ;
            WOP_Int_Proxy * iwop = dynamic_cast<WOP_Int_Proxy*>(_wop);
            assert(iwop != NULL) ;
            xp.val_int_ptr = iwop->getIntValAddr();
            //xp.original = _wop->getVal();
    } else if (type == "int") {
            xp.type = T_INT ;
            WOP_Int_Proxy * iwop = dynamic_cast<WOP_Int_Proxy*>(_wop);
            assert(iwop != NULL) ;
```

Suggestions

- Plot a draft design and code first
- Unless it is clear that a design pattern is necessary, otherwise, focus on solving the problem itself.
- Discover the problems of your implementation and its extension and future change
- See through the nature of your problem and find a design pattern to address it.

Singleton

- **Rationalle and Motivation**

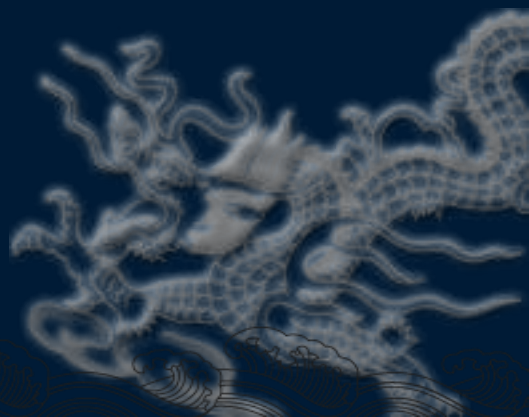
The singleton pattern applies to the many situations in which there needs to be a *single instance* of a class, a single object. It is often left up to the programmer to insure that the

An important consideration in implementing this pattern is how to make this single instance easily accessible by many other objects.

- example
  - one system can have many printers but only a spooler instance
  - a file system can have many files but there is only one

Singleton

- Timing
  - 當類別只能有一個物件個體，而且要給外界一個方便的單一窗口
  - 這唯一的物件也必須能透過繼承加以擴充



Singleton - implementation

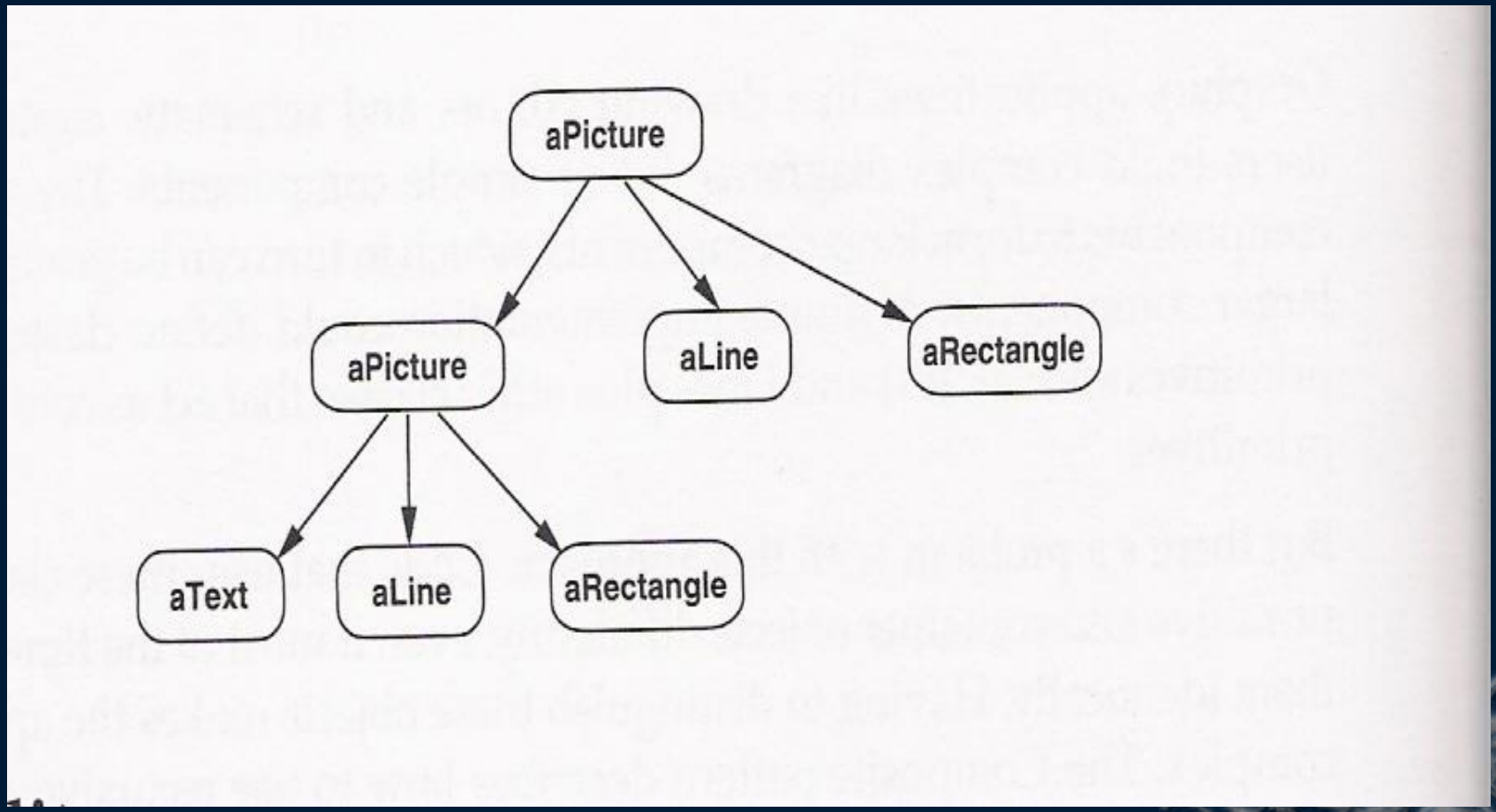
```
class Singleton {
public:
   static Singleton* Instance(); // gives back a real object!
   static proof(void); // proof that the object was made
protected:
    Singleton(); // constructor
private: static Singleton* _singleton;
};
Singleton* Singleton::_singleton = 0;
Singleton* Singleton::Instance() {
  if (_singleton == 0) { _
      singleton = new Singleton;
  } // endif
  return _singleton;
tance()
```

No class can use the constructor to new a singleton object

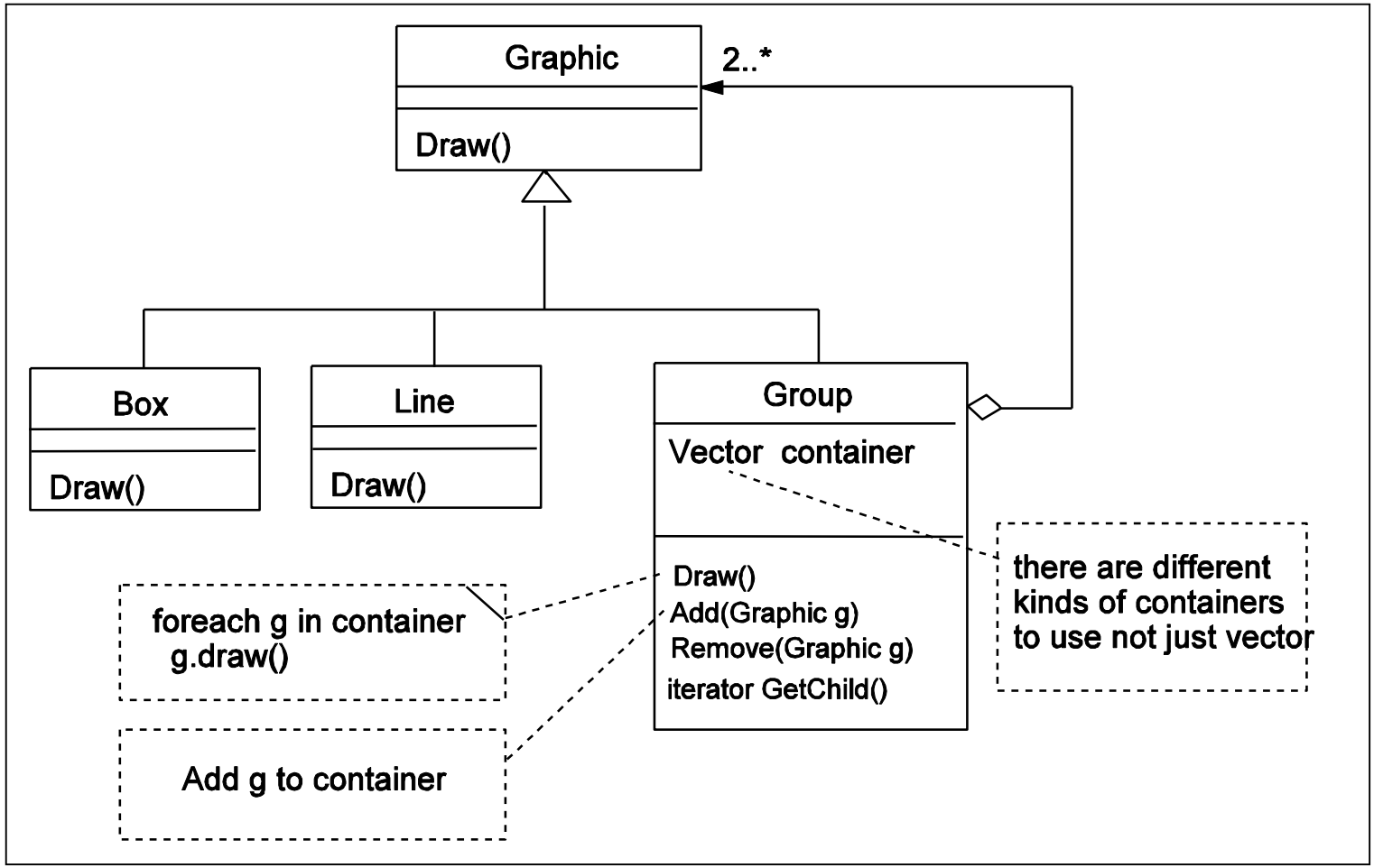
Composite

在圖形編輯器這種圖形化應用程式裡，常會用小零件拼出複雜的圖形，也會將幾個小零件結合成較大的群組物件。這個群組物件又可以當成零件來拼出更大的圖形。欲達此目的，直覺能想到的就是採用下面的 class diagram 繼承架構，其中group 是一個所謂的容器物件。

Composite object



Composite object – The first solution



Problem??

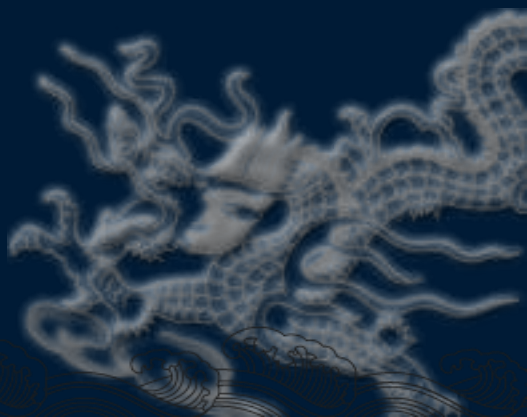
但是問題來了，譬如你們的 umleditor usecase D.2 要去ungroup 一物件。假設使用者事先點選了某一個物件，然後到Edit Menu選擇ungroup 的功能。若被點選的物件記錄在 selected 這個變數，這時候你不得不寫下下面的一段 code

```
MENU_EDITactionperformed() {
 if (selected instanceof Group)
    // 我們要把selected 的所有child 物件移除，再加入 AllObjectList
    for all child c in selected.getChild()
            selected.remove(c);
       add c to AllObjectList ;
    delete selected
 } else {
    // do nohing
}
}
```

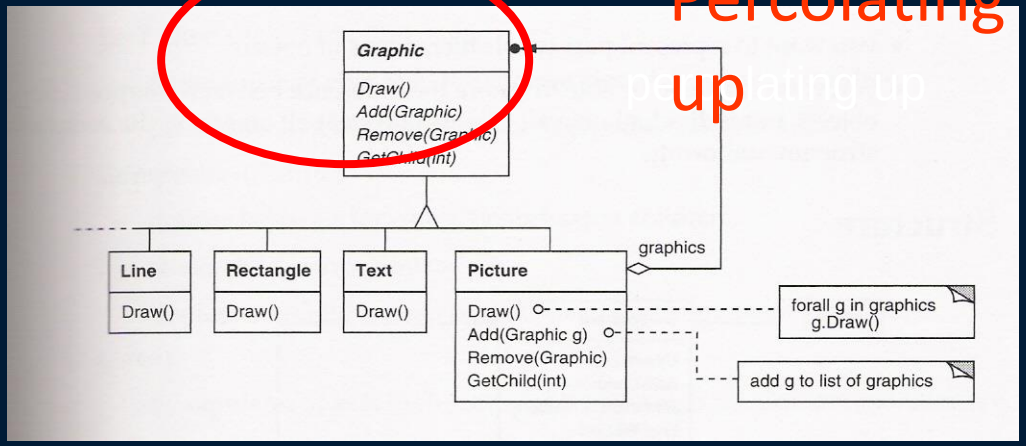
bad OO design smells?

Comments

- 我們所碰到的問題是，我們的程式碼要去判斷一個物件到底是基底物件還是group 物件
- 一般繼承的規則都告訴我們，特異化的行為應該留在sub class 就好。

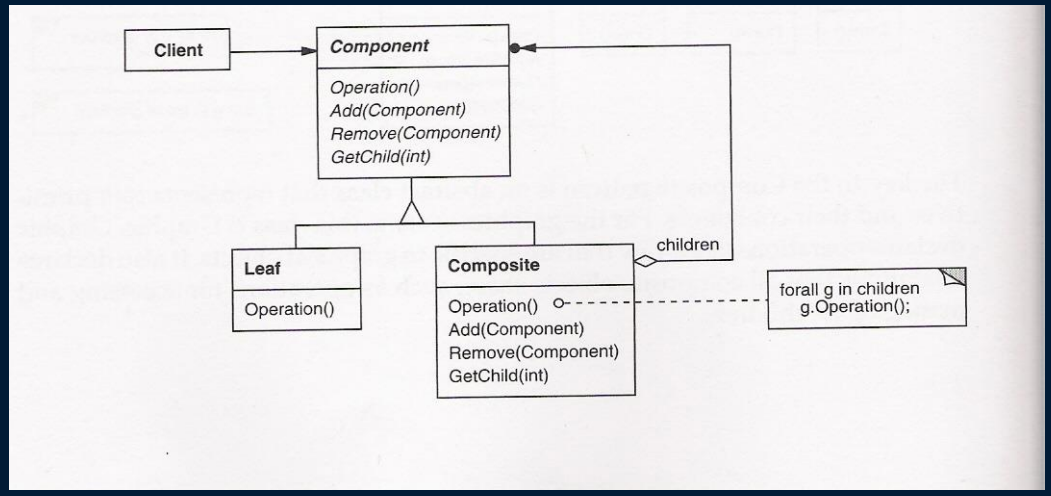


The right solution



Percolating up

Composite Design pattern



Sample code

```
class Equipment {
public:
        virtual -Equipment();
        const char* Name() { return _name; }

        virtual Watt Power();
        virtual Currency NetPrice();
        virtual Currency Discountprice();
        virtual virtual virtual
        Virtual void Add(Equipment*);
        Virtual void Remove(Equipment*);
        protected:
            Equipment(const char*);
        private:
            const char* _name;
        Iterator<Equipment*>* CreateIterator();

}
```

```
class FloppyDisk : public Equipment {

public:
FloppyDisk(const char*);
virtual -FloppyDisk();
virtual Watt Power();
virtual Currency NetPrice();
virtual Currency Discountprice();

} ;
```

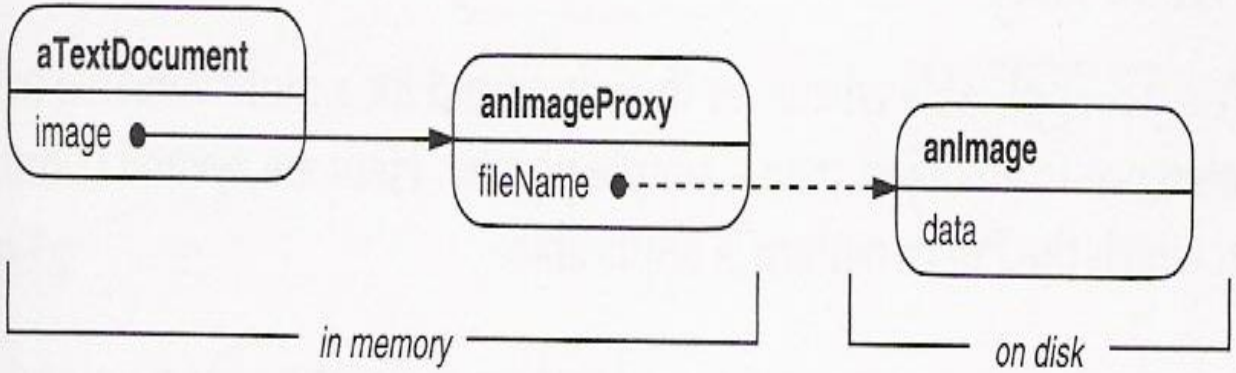
```
class CompositeEquipment : public Equipment {

public:
virtual -CompositeEquipment();
virtual Watt Power();
virtual Currency NetPrice();
virtual Currency Discountprice();
virtual void Add(Equipment*);
virtual void Remove(Equipment*);
virtual Iterator<Equipment*>* CreateIterator();

protected:
        CompositeEquipment(const char*);
private:
        List<Equipment*> _equipment;

} ;
```

```
class Chassis : public compositeEquipment {
public:
Chassis(const char*);
virtual -Chassis();
virtual Watt Power();
virtual Currency Netprice(); virtual Currency
Discountprice();

} ;
```

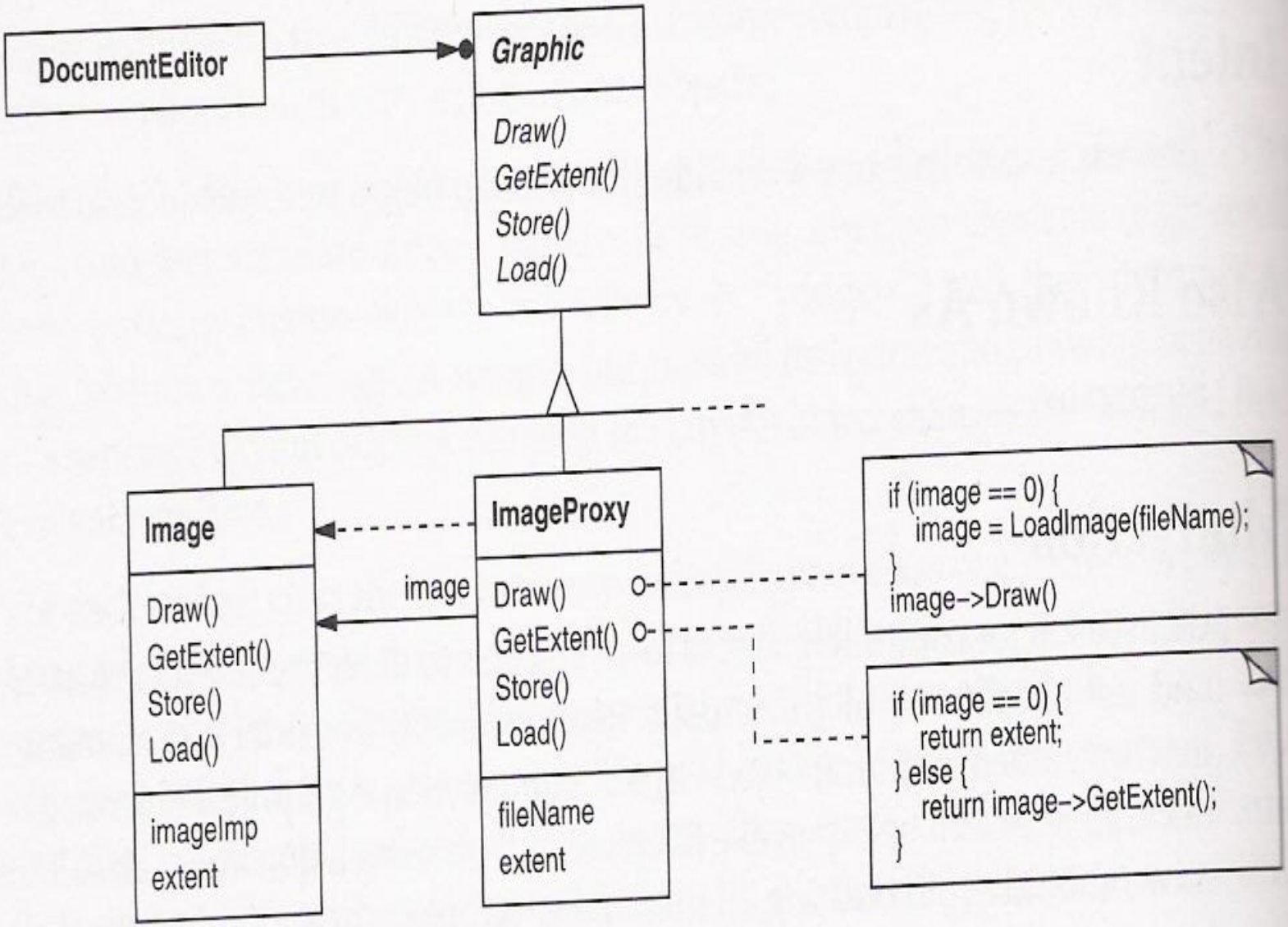
```
Cabinet * cabinet  = new Cabinet("PC Cabinet");
Chassis* chassis = new Chassis("PC Chassis");
cabinet->Add(chassis);
Bus* bus = new Bus ("MCA Bus");
bus->Add(new Card("l6Mbs Token Ring");
chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy“);
cout << "The net price is " ? chassis->NetPrice() ? end
```

Proxy pattern

- 我們想要延緩一個物件的建立與初始化，原因是物件的建立與初始化需要速度與記憶體等等代價
- 我們希望當物件真正需要的時候才進行建立與初始化（on demand)
  - 例如一份文件中的許多影像物件
- 但是這個延緩的過程應該物件自行處理掉，例如讀檔的程式碼不需要處理這些細節

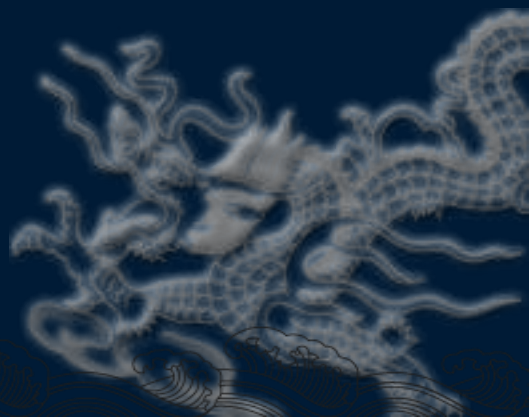


The image proxy creates the real image

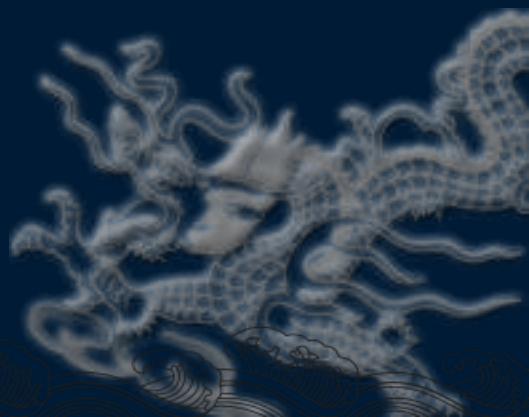

DocumentEditor
Graphic
Draw()
GetExtent()
Store()
Load()
Image
Draw()
GetExtent()
Store()
Load()
imageImp
extent
image
ImageProxy
Draw()
GetExtent()
Store()
Load()
fileName
extent
if (image == 0) {
image = LoadImage(fileName);
}
image->Draw()
if (image == 0) {
return extent;
} else {
return image->GetExtent();
}

Sample code

```
class Graphic {
public:
        virtual -Graphic();

        virtual void Draw(const Point& at) = 0;
        virtual void HandleMouse(Event& event) = 0;

        virtual const point& GetExtent() = 0;

        virtual void Load(istream& from) = 0;
        virtual void Save(ostream& to) = 0;
        protected:
        Graphic () ;

} ;
```



```
class Image : public Graphic {

public:
Image(const char* file); // loads image from a file
virtual -Image();

virtual void Draw(const point& at);
virtual void HandleMouse(Event& event);

virtual const point& GetExtent();
virtual virtual private: I I ...

void Load(istream& from);
void Save(ostream& to);

};
```



Imageproxy has the same interface as Image:

```
class Imageproxy : public Graphic (
public:
        ImageProxy(const char* imageFile);
        virtual -ImageProxy();

        virtual void Draw(const point& at);
        virtual void HandleMouse(Event& event);

        virtual const point& GetExtent();

        virtual void Load(istream& from);
        virtual void Save(ostream& to);
protected:
        Image* GetImage();
private:
         Image* _image;
         Point _extent;
         char* _fileName;
} ;
```

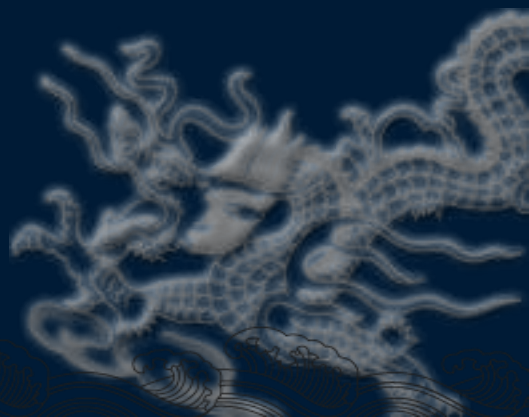
```
ImageProxy::Imageproxy (const char* fileName) {

_fileName = strdup(fileName);
_extent = Point::Zero; //  don't know extent yet _image = 0;

}
Image* Imageproxy: :GetImage() {
      if (_image == 0)  {
                  _image = new Image(_fileName);

       }
       return _image;
}
void ImageProxy::Draw (const point& at) {
    GetImage()->Draw(at);
}
```

Finally, suppose we have a class TextDocument that can contain Graphic objects:

```
class TextDocument {
public:
    TextDocument();

void Insert(Graphic*);
/ / ...

} ;
```

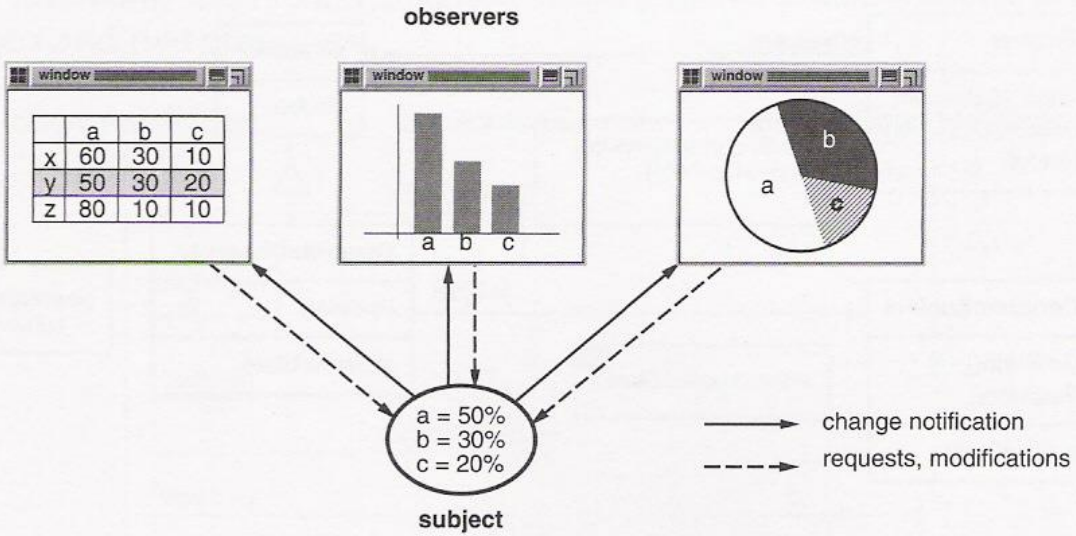
We can insert an ImageProxy into a text document like this:

```
TextDocument* text = new TextDocument;
/ / ...
text->Insert(new ImageProxy("anImageFileName“);
```

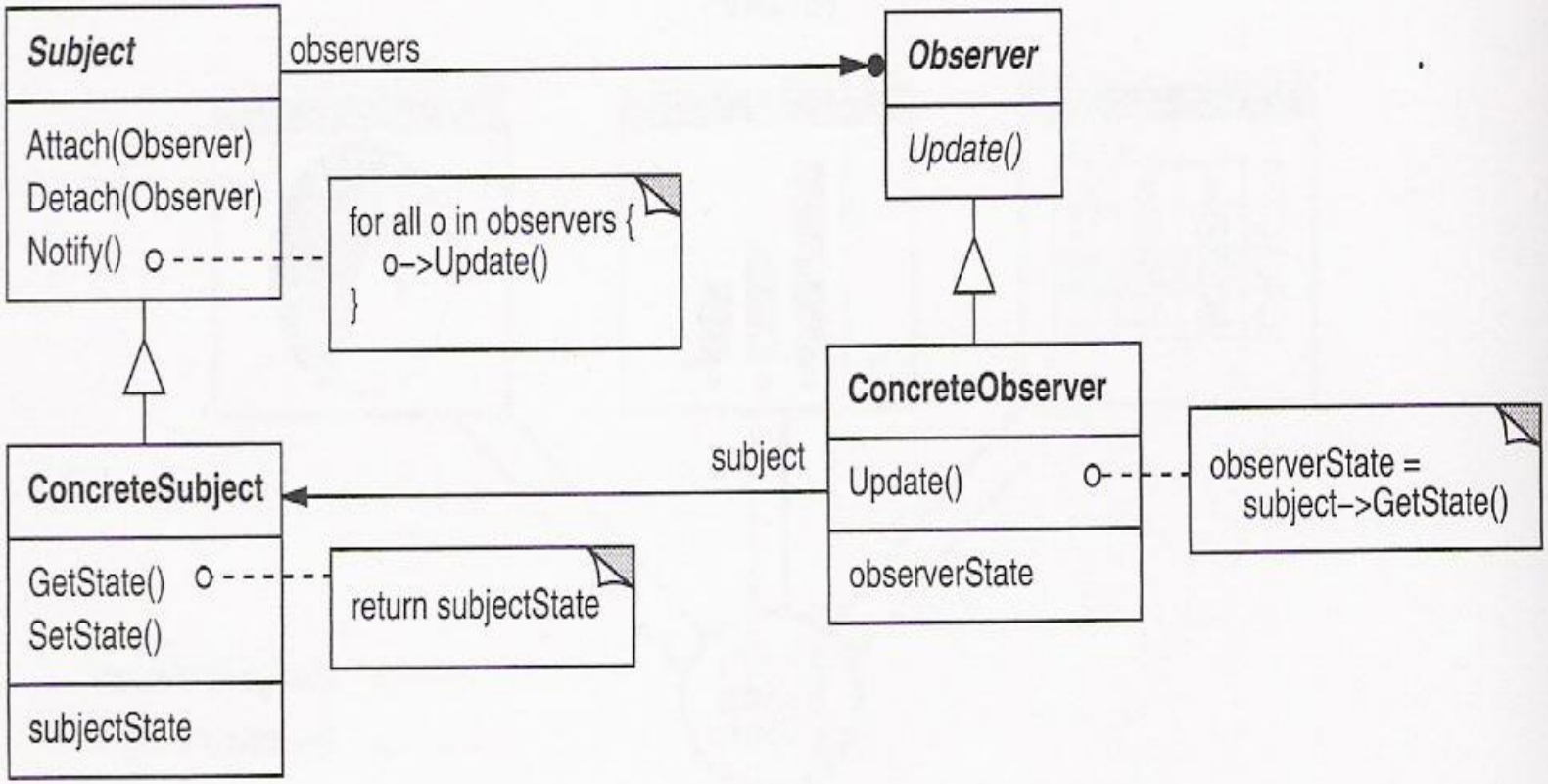


Observer Pattern

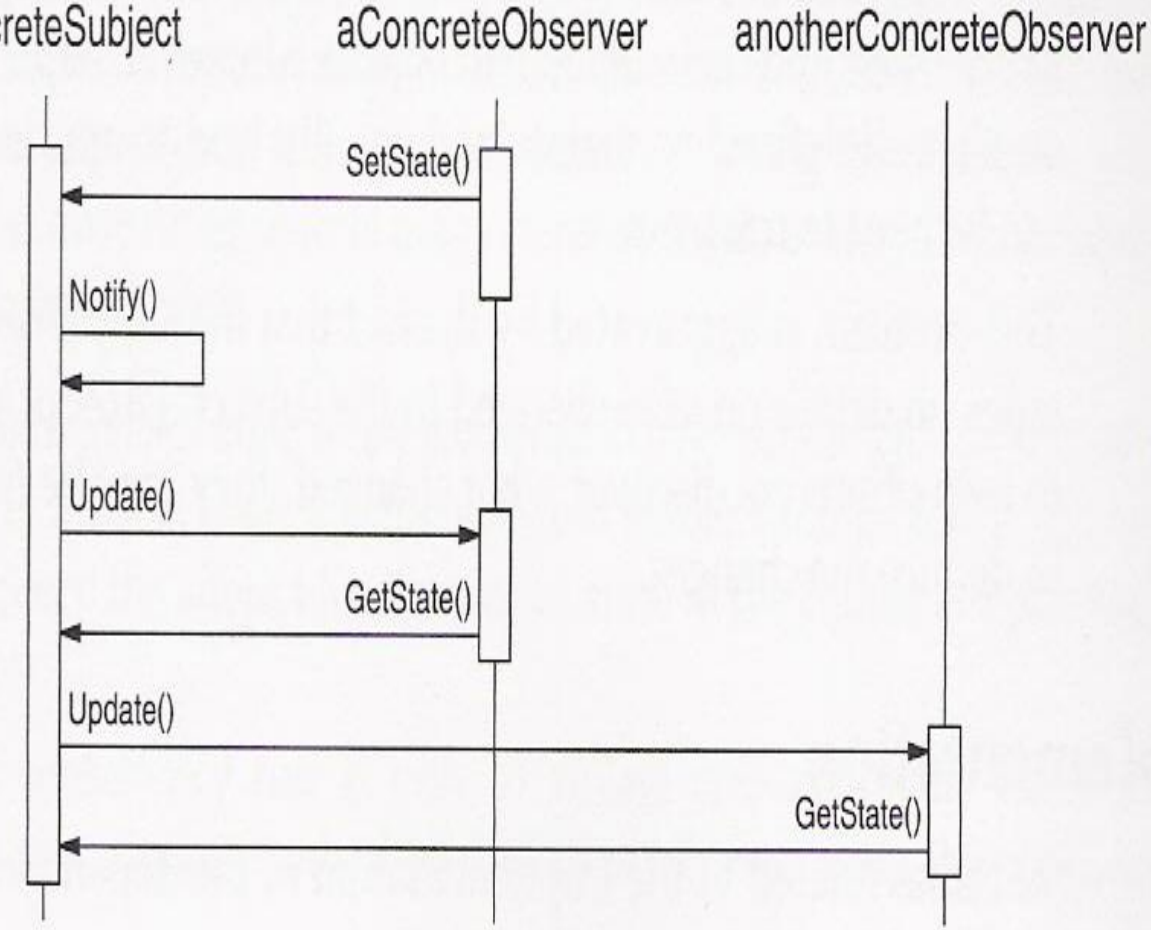
- Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Structure

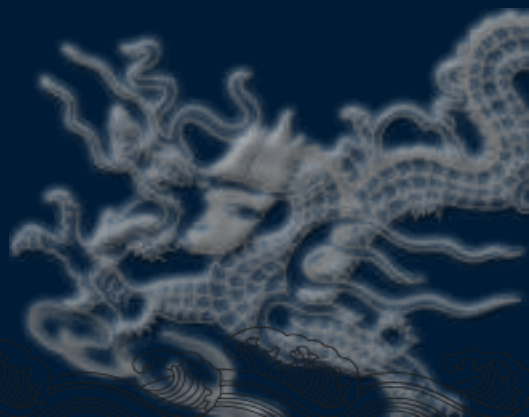

Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
Update()
for all o in observers {
o->Update()
}
ConcreteObserver
subject
ConcreteSubject
Update()
observerState =
subject->GetState()
GetState()
SetState()
return subjectState
observerState
subjectState

subject and two observers:



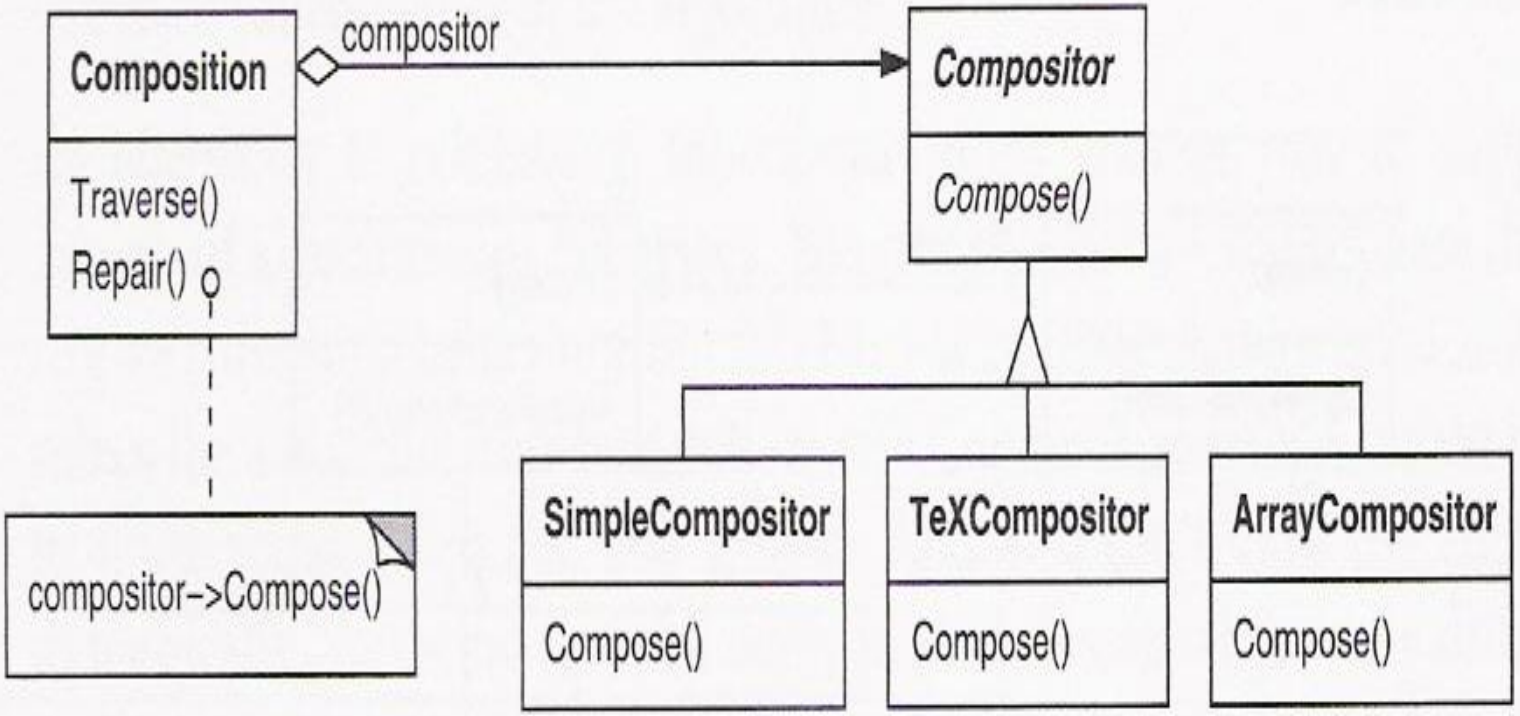
Strategy Pattern

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



Motivation

- Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:
- Clients that need linebreaking get more complex if they include the linebreaking code. That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.
- Different algorithms will be appropriate at different times. We don't want to support multiple linebreaking algorithms if we don't use them all.
- . It's difficult to add new algorithms and vary existing ones when linebreaking is an integral part of a client.
- We can avoid these problems by defining classes that encapsulate different linebreaking algorithms. An algorithm that's ep.capsulated in this way is called a strategy.



uppose a Composition class is responsible for maintaining and updating the

Sample code

Suppose a composition classa
is responsible for maintaining
and updating the linebreaks of text displayed in a text viewer

```
class Composition {

public:
Composition(Compositor*);
void Repair();

private:
Compositor* _compositor;
Component* _components;
int _componentCount;
int _lineWidth;
int* _lineBreaks;

int _lineCount;

} ;
```

Strategy 1

```
class Compositor {
public:
virtual int Compose( Coord
natural[], Coord stretch[],
Coord shrink[], int component
Count , int lineWidth, int
breaks[]
) = 0;
protected:
Compositor();

} ;
```

```
class SimpleCompositor : public compositor {

public:
SimpleCompositor();

virtual int Compose ( Coord natural[], Coord
stretch[], Coord shrink[], int componentCount,
int lineWidth, int breaks[]

) ;
II

} ;
```

Strategy 2

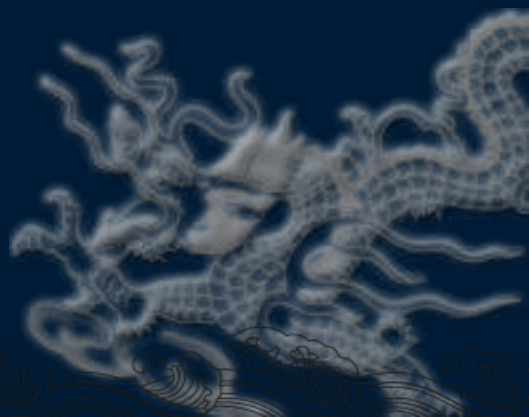
```
class TeXCompositor : public Compositor {
public:
TeXCompositor();

virtual int Compose ( Coord natural[], Coord stretch[], Coord
shrink[], int component Count , int lineWidth, int breaks[]) ;

} ;
```

Strategy 3

```
class ArrayCompositor : public Compositor public:
ArrayCompositor(int interval);

virtual int Compose ( Coord natural[], Coord stretch[], Coord
shrink[], int componentCount, int lineWidth, int breaks[]

) ;

} ;
```

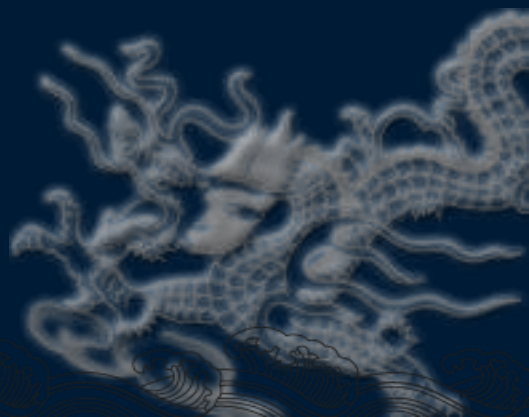
To instantiate Composition, you pass it the compositor you want to use:

```
Composition* quick = new Composition(new SimpleCompositor);
Composition* slick = new Composition (new TeXCompositor);
Composition* iconic = new Composition(new ArrayCompositor);
```



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.

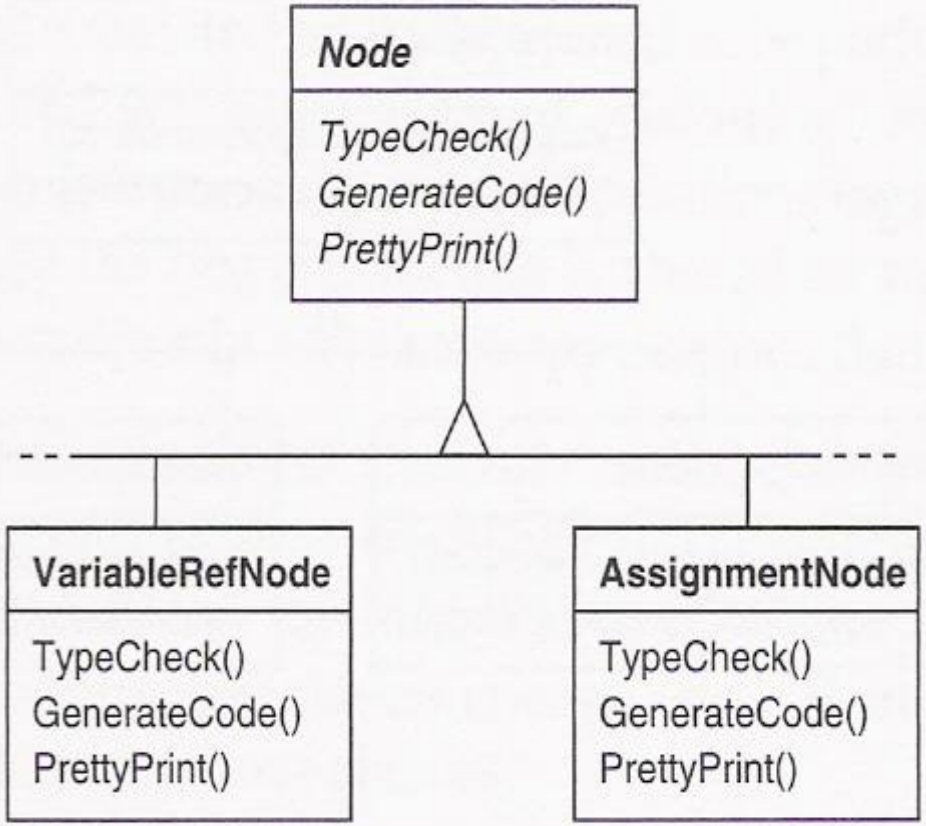


Motivation

Consider a compiler that represents programs as abstract syntax trees. It will need to perform operations on abstract syntax trees for "static semantic" analyses like checking that all variables are defined. It will also need to generate code. So it might define operations for type-checking, code optimization, flow analysis, checking for variables being assigned values before they're used, and so on. Moreover, we could use the abstract syntax trees for pretty-printing, program restructuring, code instrumentation, and computing various metrics of a program.

Most of these operations will need to treat nodes that represent assignment statements differently from nodes that represent variables or arithmetic expressions.

Hence there will be one class for assignment statements, another for variable accesses, another for arithmetic expr

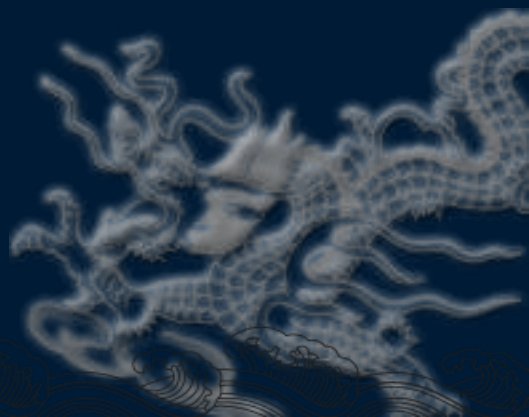

Node
TypeCheck()
GenerateCode()
PrettyPrint()
VariableRefNode
TypeCheck()
GenerateCode()
PrettyPrint()
AssignmentNode
TypeCheck()
GenerateCode()
PrettyPrint()

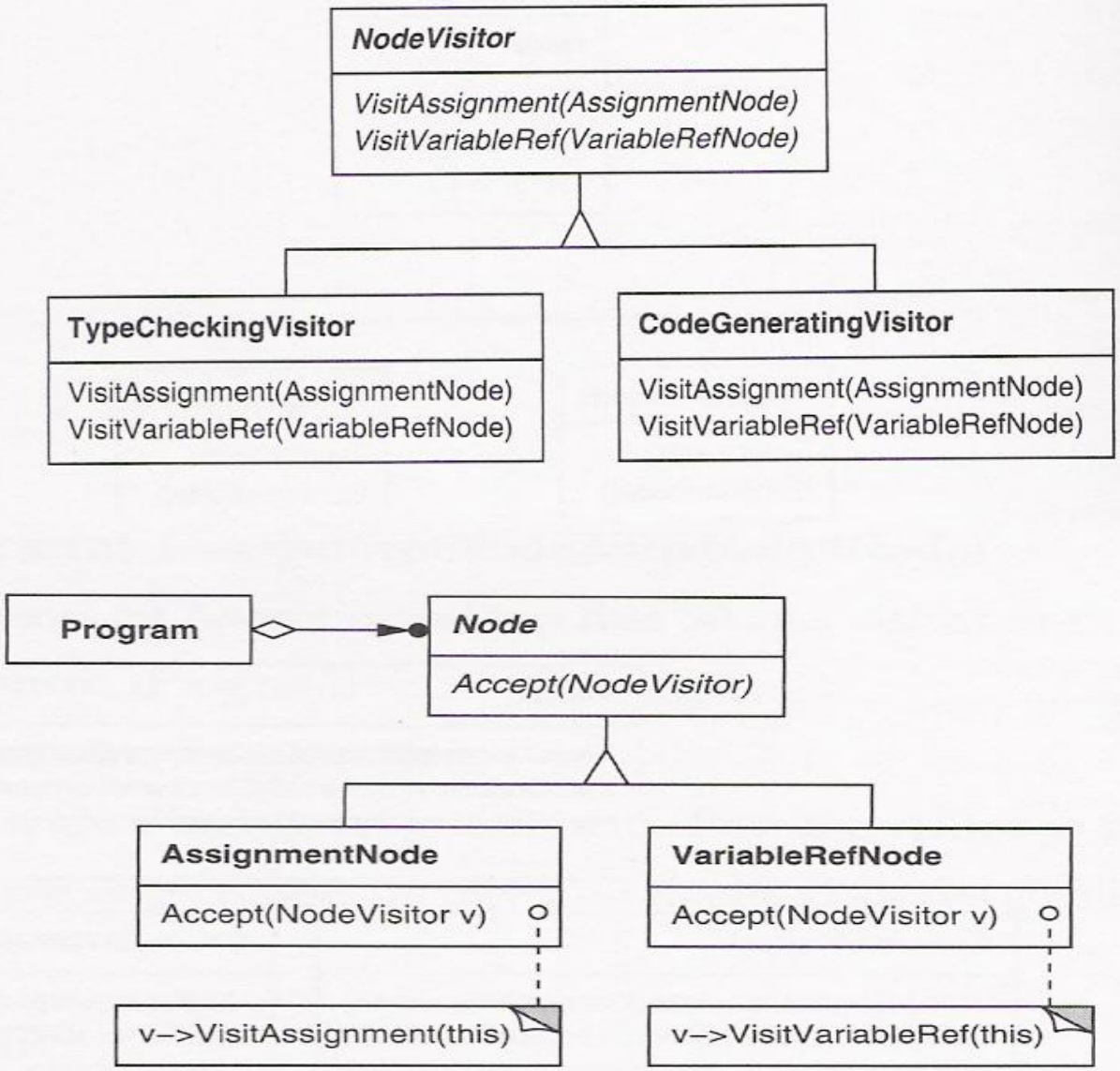
Problem??

- The problem here is that distributing all these operations across the various node classes leads to a system that is hard to understand, maintain, and change. It would be confusing to have type-checking code mixed with pretty-printing code or flow analysis code
- It would be better if each new operation could be added separately and the node classes were independent of the operations that apply to them

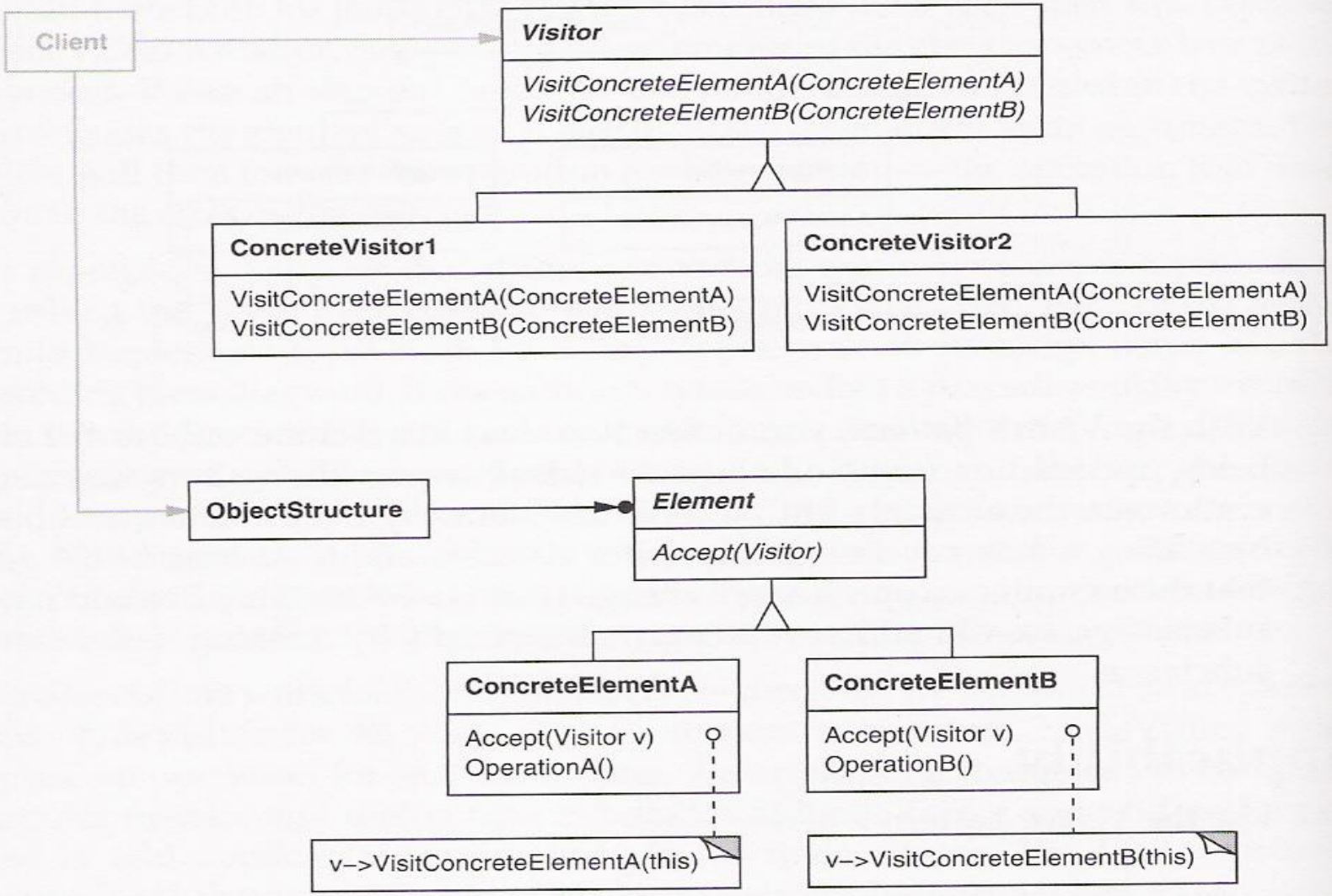
More comments

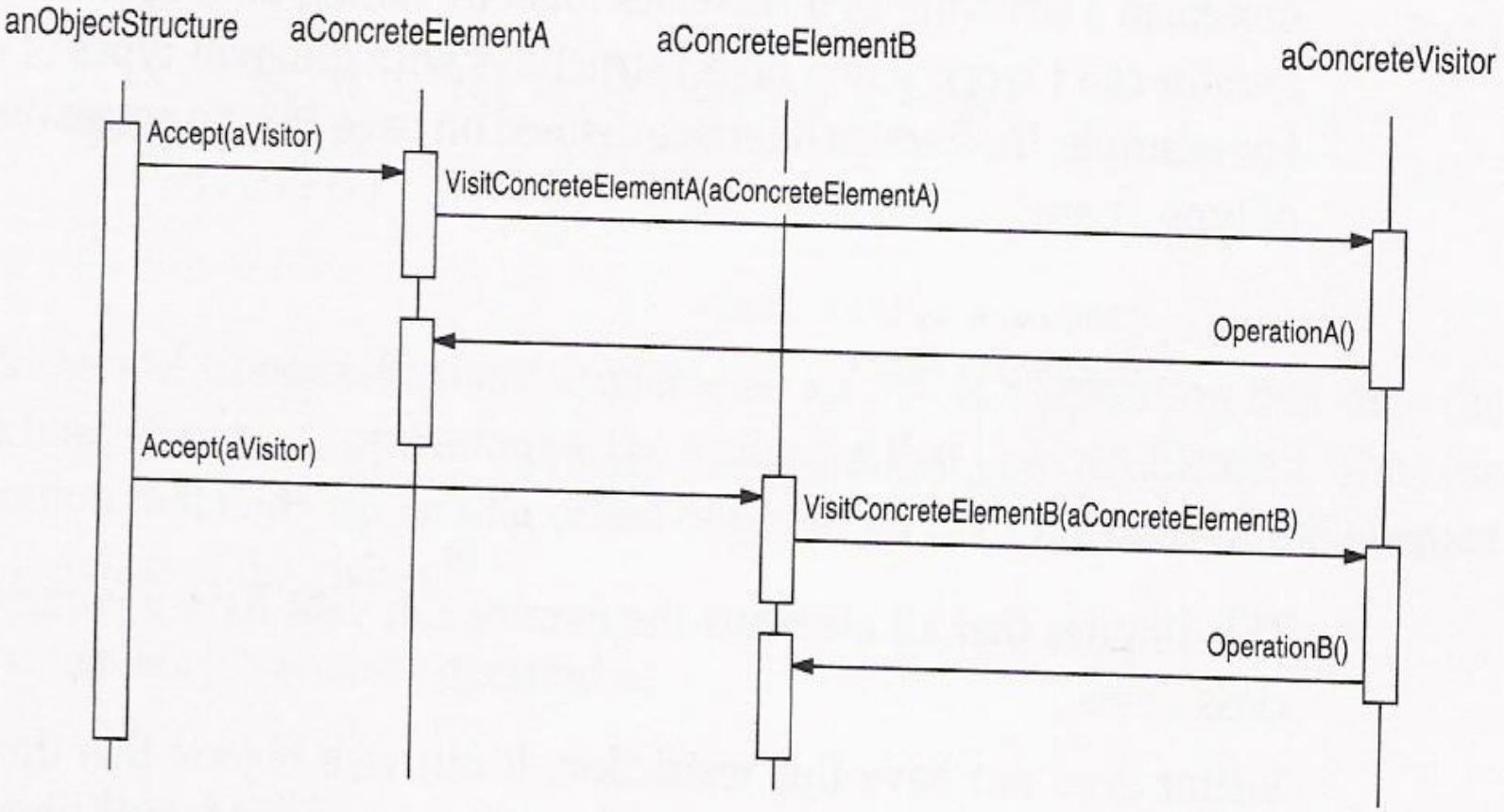
- 實際上 typecheck, generatecode 這些visitor 都是分別執行的
- 我們希望這些程式碼不要跟 node的本身資料結構扯在一起
- 我們希望 visitor 到底是哪一個在拜訪node時再告知node就好了





attern, you define two class hierarchies: one for the elem


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)


anObjectStructure
aConcreteElementA
aConcreteElementB
aConcreteVisitor
Accept(aVisitor)
VisitConcreteElementA(aConcreteElementA)
OperationA()
Accept(aVisitor)
VisitConcreteElementB(aConcreteElementB)
OperationB()