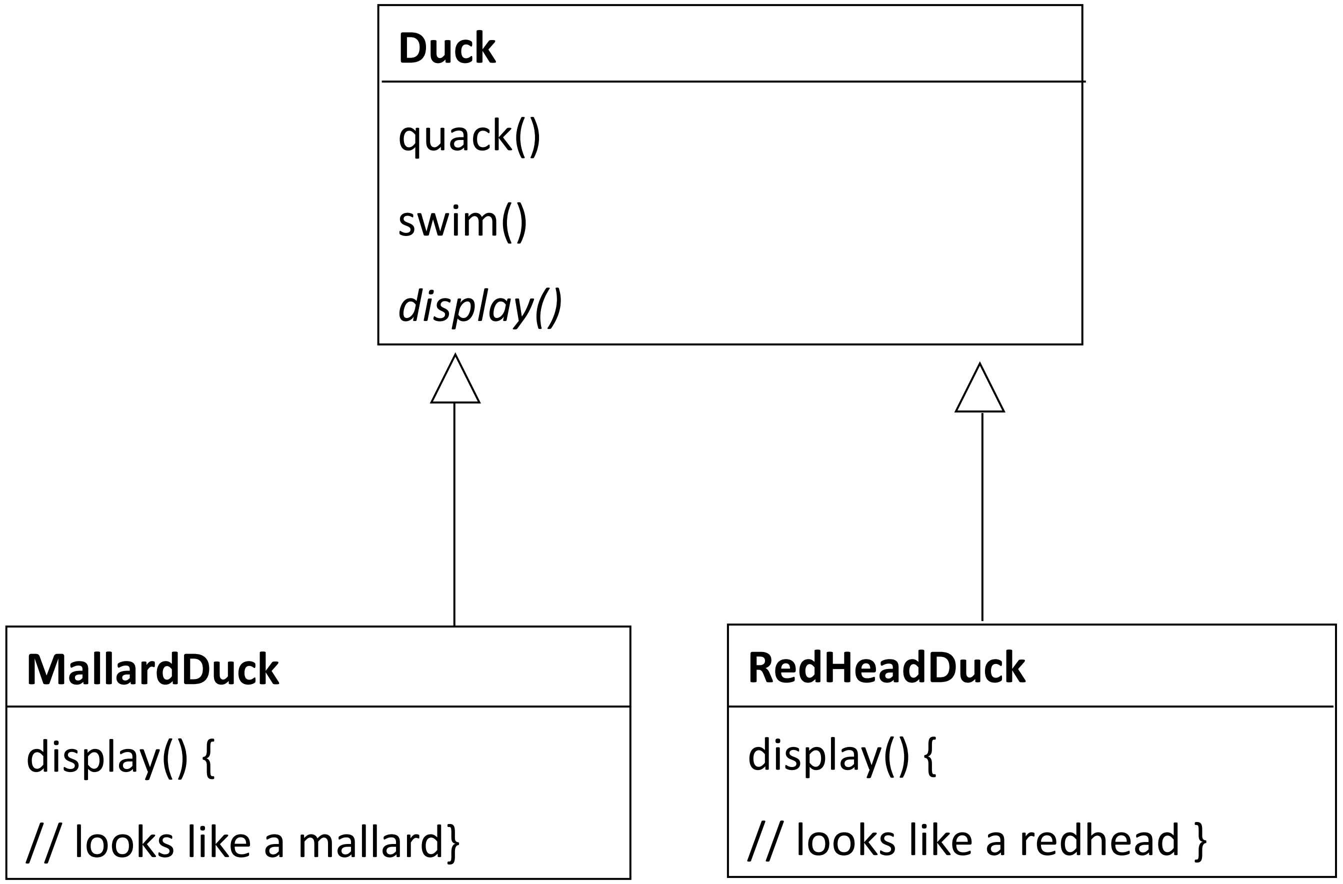
Esercizi design patterns

Angelo Di Iorio, diiorio@cs.unibo.it

Esercizio: Duck application

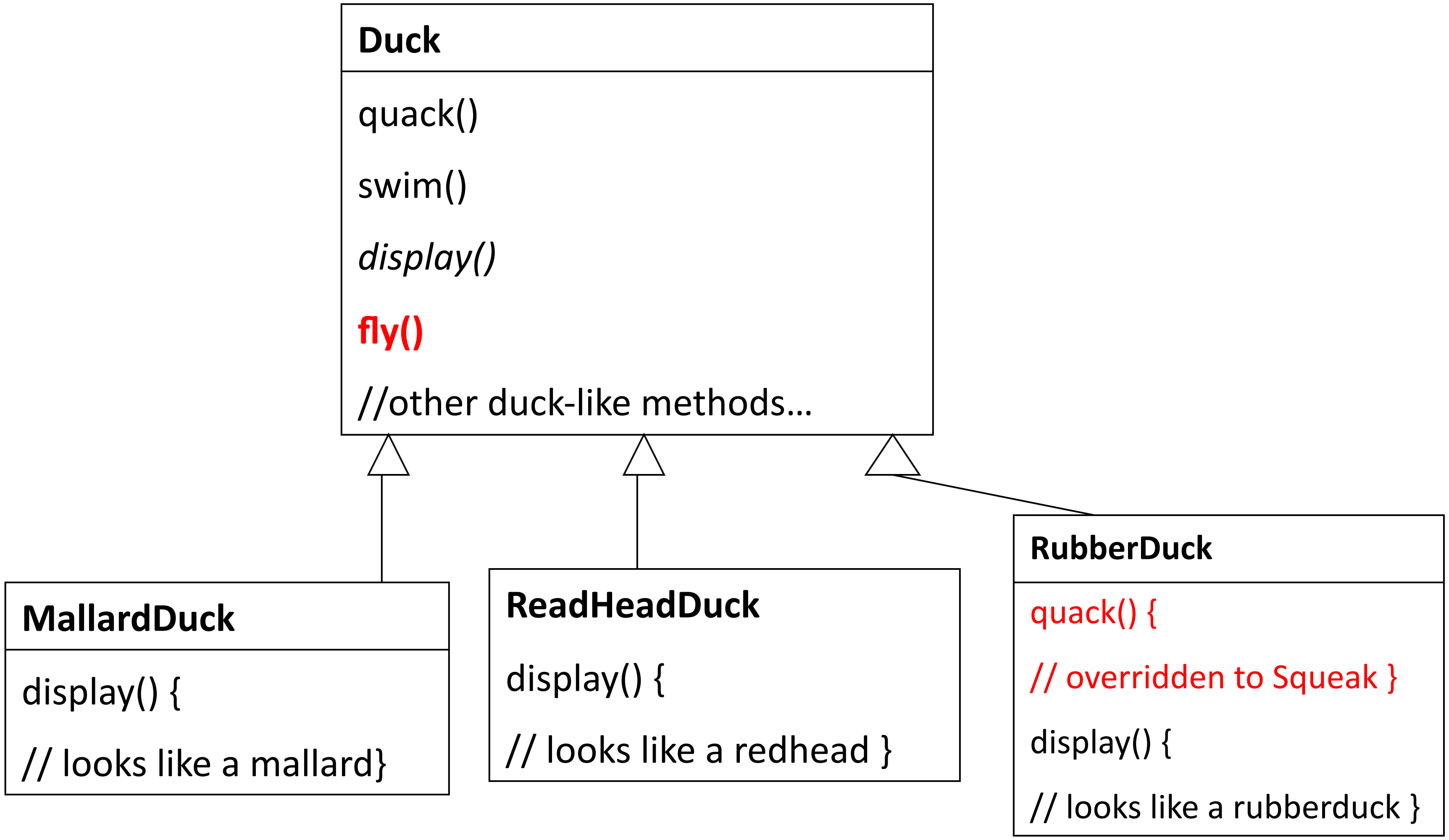


[da: 'Head First. Design Patterns.']

Estendibile?

- Come aggiungere un nuovo tipo di anatra che fa un verso diverso dalle altre?
- Come aggiungere il comportamento *fly()*?

Problemi?

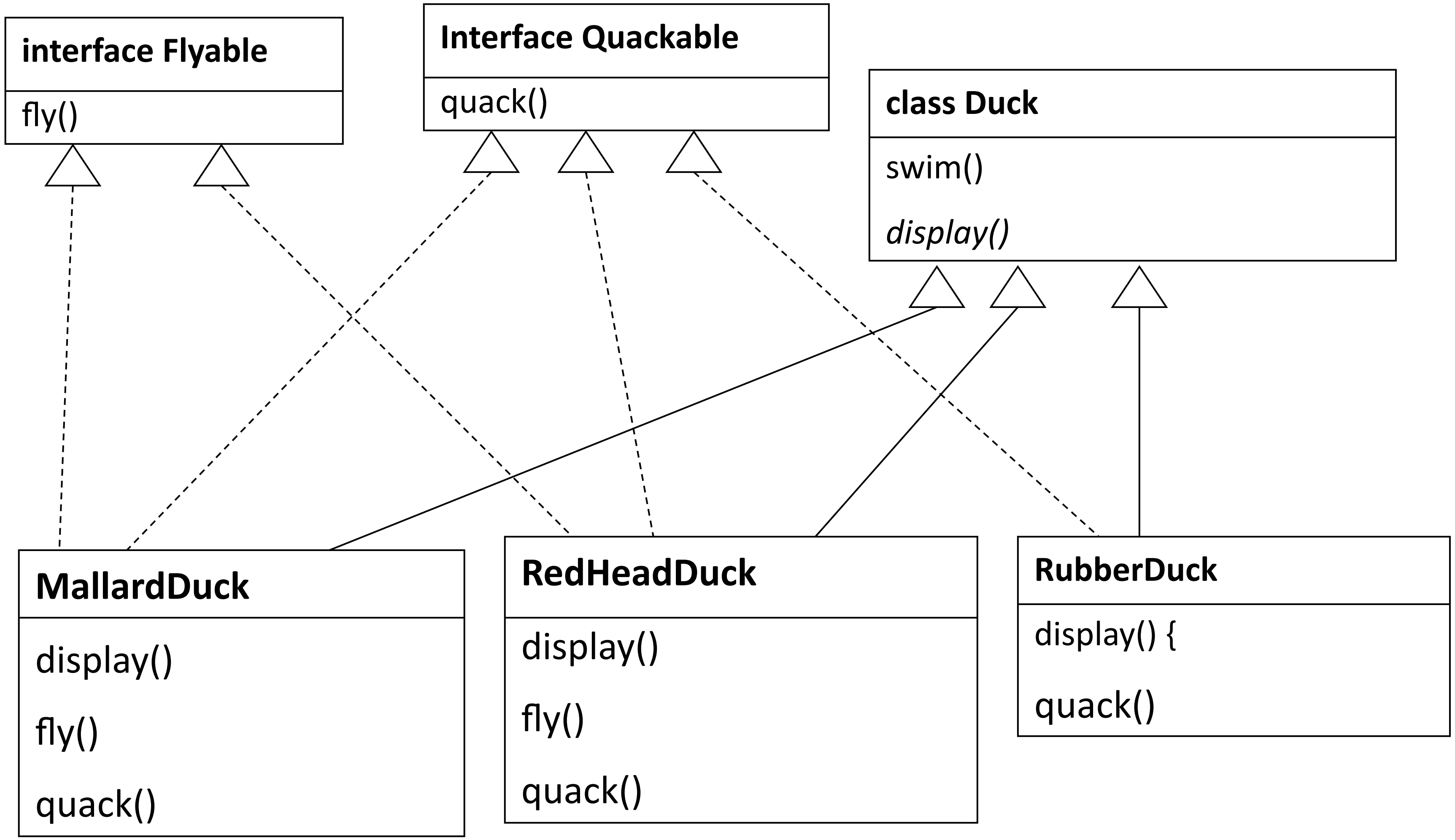


Duck
quack()
swim()
display()
fly()
//other duck-like methods…
MallardDuck
display() {
// looks like a mallard}
ReadHeadDuck
display() {
// looks like a redhead }
RubberDuck
quack() {
// overridden to Squeak }
display() {
// looks like a rubberduck }

Estendibile?

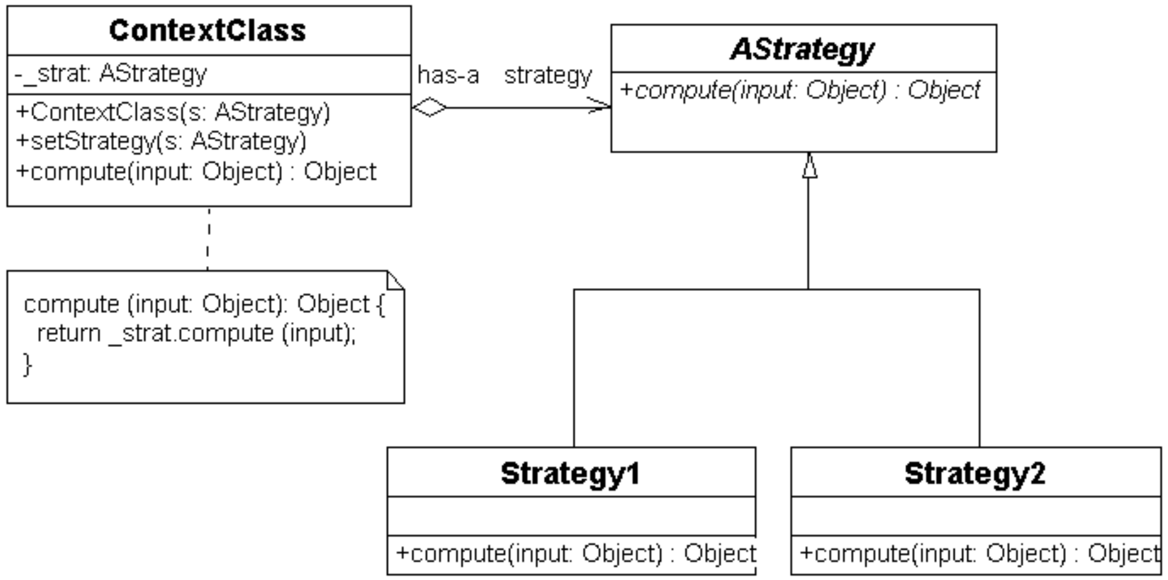
- Come aggiungere un nuovo tipo di anatra, ad esempio muta e che non vola?
- Basta fare *override* dei metodi quack() e fly()?
- E per nuovi tipi di anatra che hanno comportamenti – quack() e fly() – parzialmente sovrapposti alle altre?
- Come gestire le diverse combinazioni?

Interfacce? Problemi?

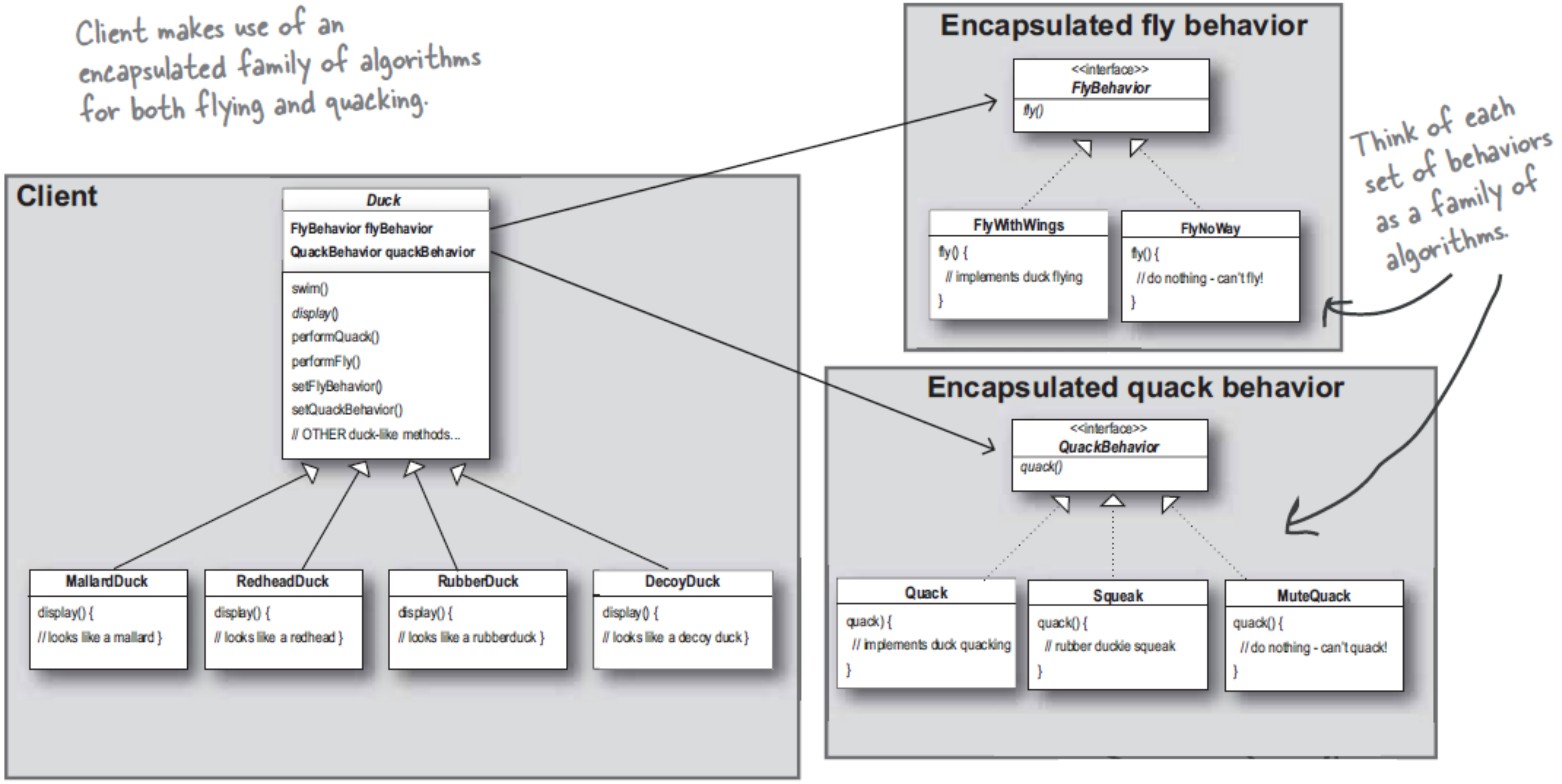


Pattern Strategy

- *Problema*: definire una famiglia di algoritmi e renderli interscambiabili
  - modificare il comportamento di una classe a run-time e disaccoppiare il comportamento (Algoritmo) dalla classe (Client) che lo usa


ContextClass
-_strat: AStrategy
+ContextClass(s: AStrategy)
+setStrategy(s: AStrategy)
+compute(input: Object) : Object
has-a
strategy
AStrategy
+compute(input: Object) : Object
compute (input: Object): Object {
return _strat.compute (input);
}
Strategy1
+compute(input: Object) : Object
Strategy2
+compute(input: Object) : Object

Strategy



Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack) {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}

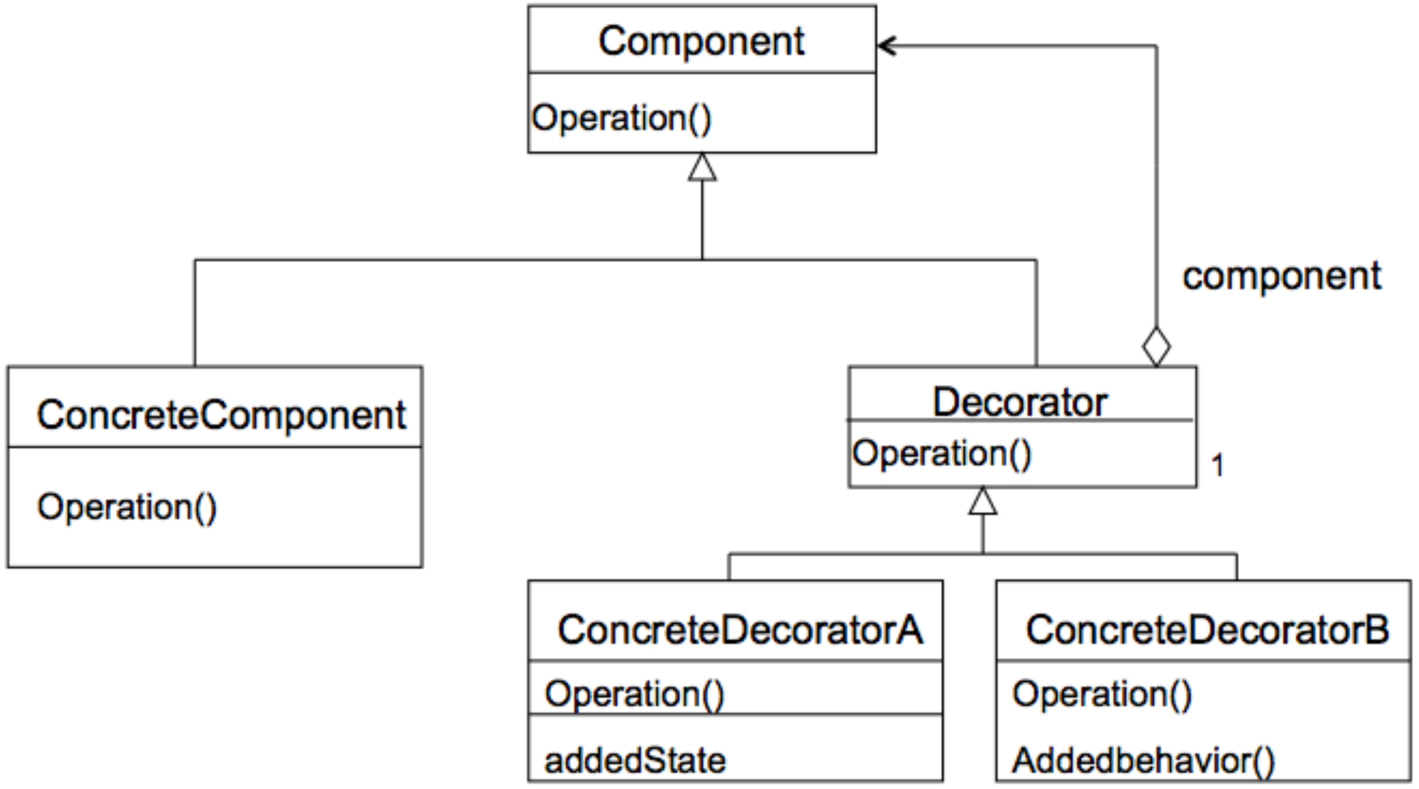
[da: 'Head First. Design Patterns.']

Esercizio

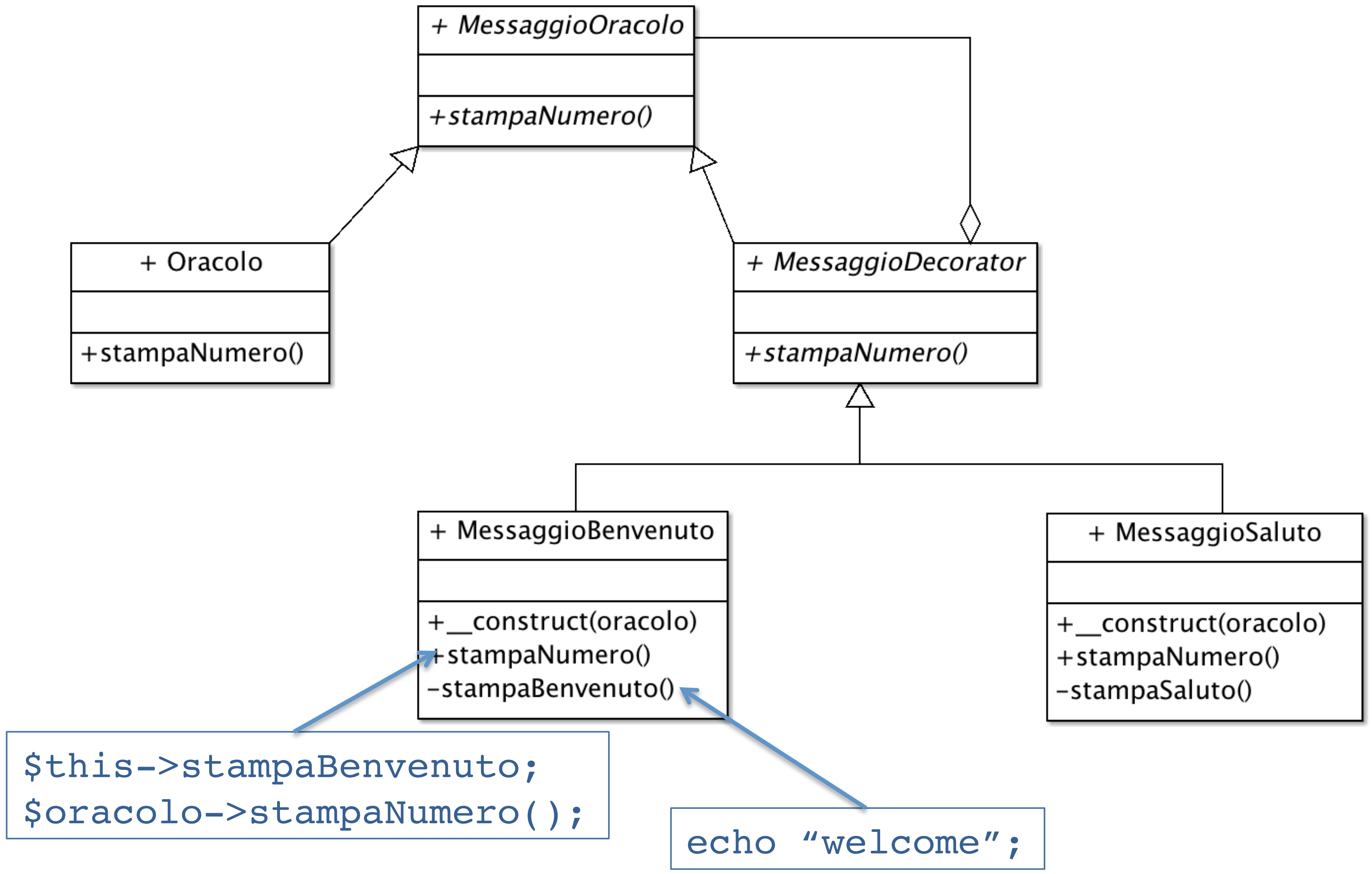
- Una classe *Oracolo* esporta un metodo per restituire una numero casuale (*stampaNumero*).
- Estendere la classe per permettere di:
  - stampare un messaggio di benvenuto prima di cercare il numero
  - stampare un messaggio di saluto alla fine
  - stampare entrambi i messaggi precedenti, anche in ordine diverso

Decorator Pattern

- *Problema*: aggiungere un comportamento ad un oggetto dinamicamente (a run-time)



Oracolo



```
$this->stampaBenvenuto;
$oracolo->stampaNumero();
```

```
echo “welcome”;
```

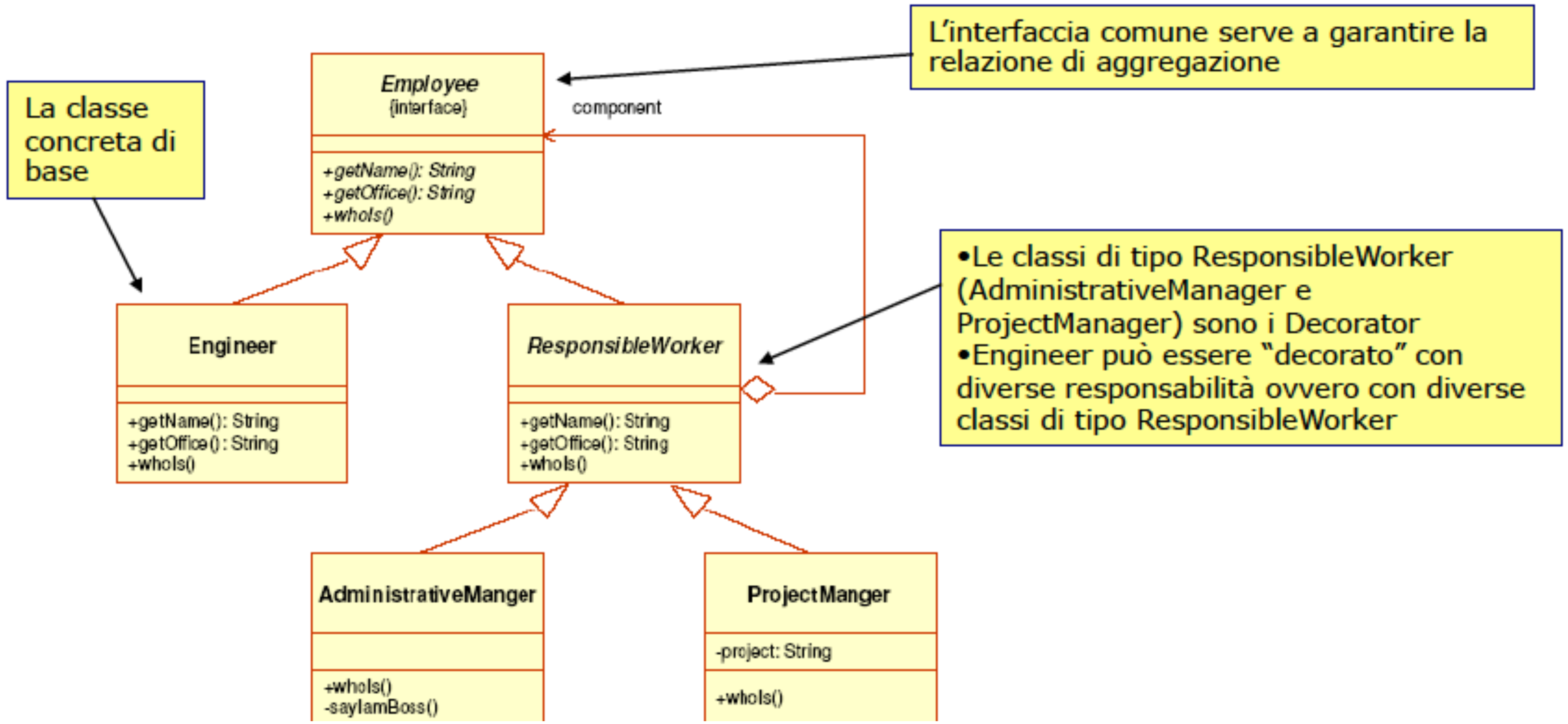
Run-time

```
oracolo = new Oracolo();
oracolo.stampaNumero();         // stampa 327189

welcome = new MessaggioBenvenuto(oracolo);
bye = new MessaggioSaluto(oracolo);

welcome.stampaNumero();
   // stampa “welcome 790789”

bye.stampaNumero();
   // stampa “33909 bye”

all = new MessaggioSaluto(welcome);
all.stampaNumero();
   // stampa “welcome 4446 bye”

crazy = new MessaggioBenvenuto(new MessaggioSaluto(oracolo))
```

Esercizio

- Disegnare un diagramma UML che modella il seguente dominio:
  - una azienda è costituita da Employee che afferiscono a diversi uffici
  - un Engineer è un tipo di Employee
  - un Engineer può assumere l’incarico di capoufficio (AdministrativeManager) o di capoprogetto (ProjectManager)
  - un Engineer può essere capo ufficio ed anche capo progetto di più progetti
- Quale design pattern?

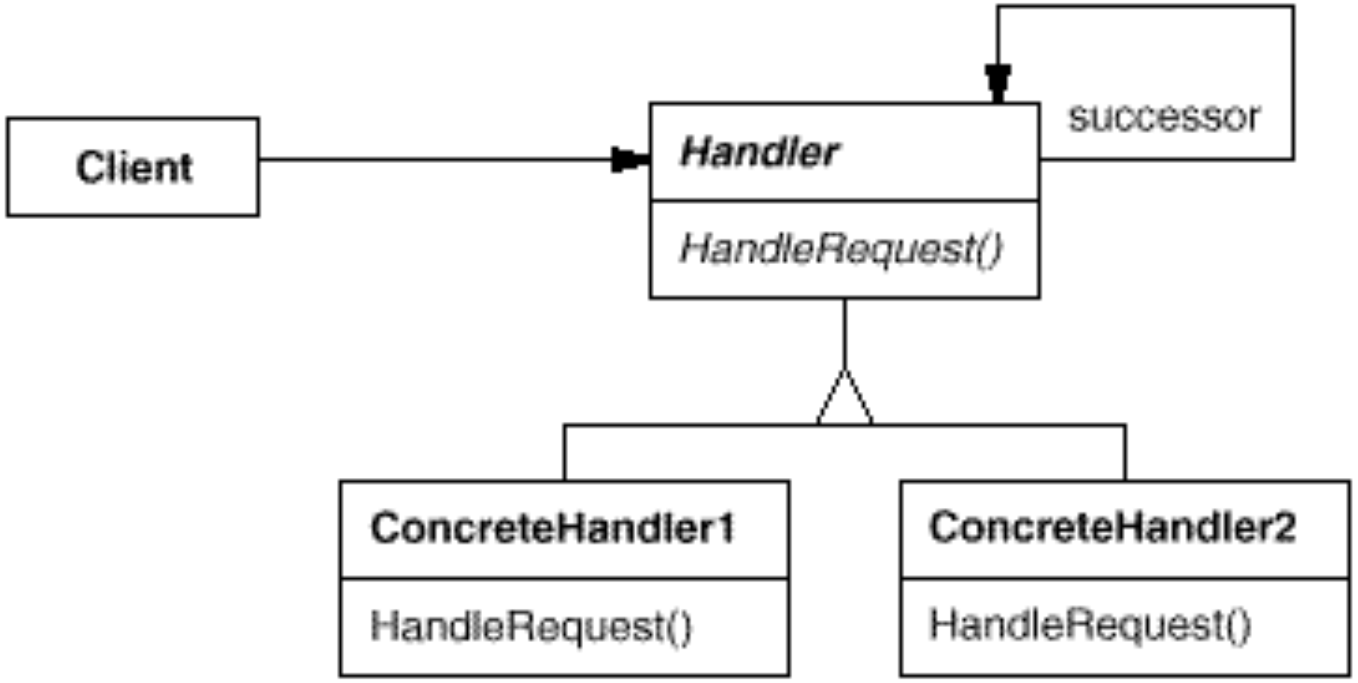
Decorator


La classe concreta di base
Employee
{interface}
+getName(): String
+getOffice(): String
+whoIs()
component
L'interfaccia comune serve a garantire la relazione di aggregazione
Engineer
+getName(): String
+getOffice(): String
+whoIs()
ResponsibleWorker
+getName(): String
+getOffice(): String
+whoIs()
•Le classi di tipo ResponsibleWorker (AdministrativeManager e ProjectManager) sono i Decorator
•Engineer può essere "decorato" con diverse responsabilità ovvero con diverse classi di tipo ResponsibleWorker
AdministrativeManger
+whoIs()
-sayIamBoss()
ProjectManger
-project: String
+whoIs()

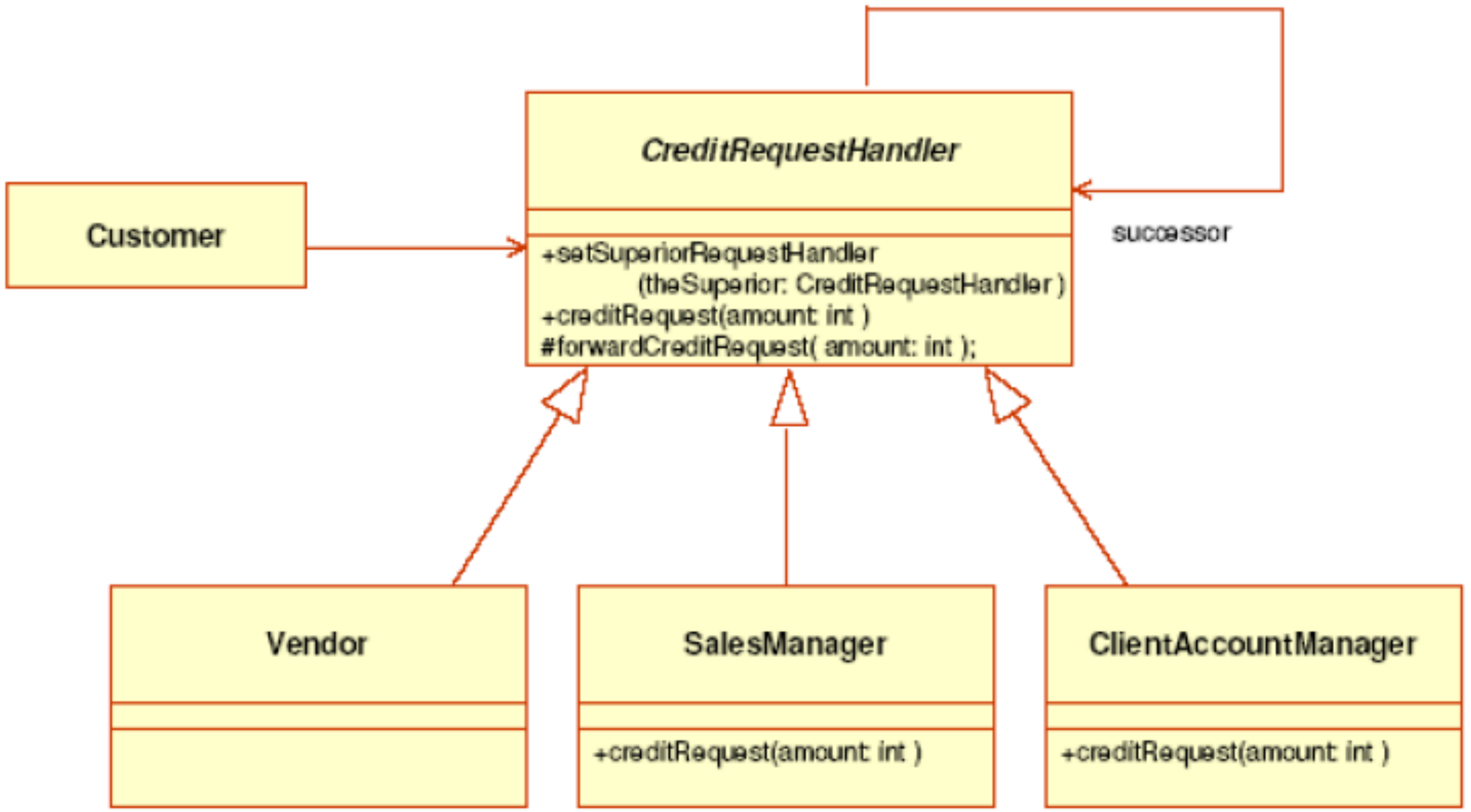
Esercizio

- Disegnare un diagramma UML che modella il seguente dominio:
  - Un’azienda deve gestire le richieste di credito dei clienti (customers).
  - Internamente l’azienda si organizza in diversi livelli:
    - Il livello più basso (vendor) può approvare le richieste fino a un dato importo.
    - Le richieste che superano questo importo vanno gestite da un livello superiore (sales manager), il quale ha un altro importo massimo da gestire.
    - Oltre tale importo le richieste sono gestiste da un ‘client account manager’

Chain of Responsibility



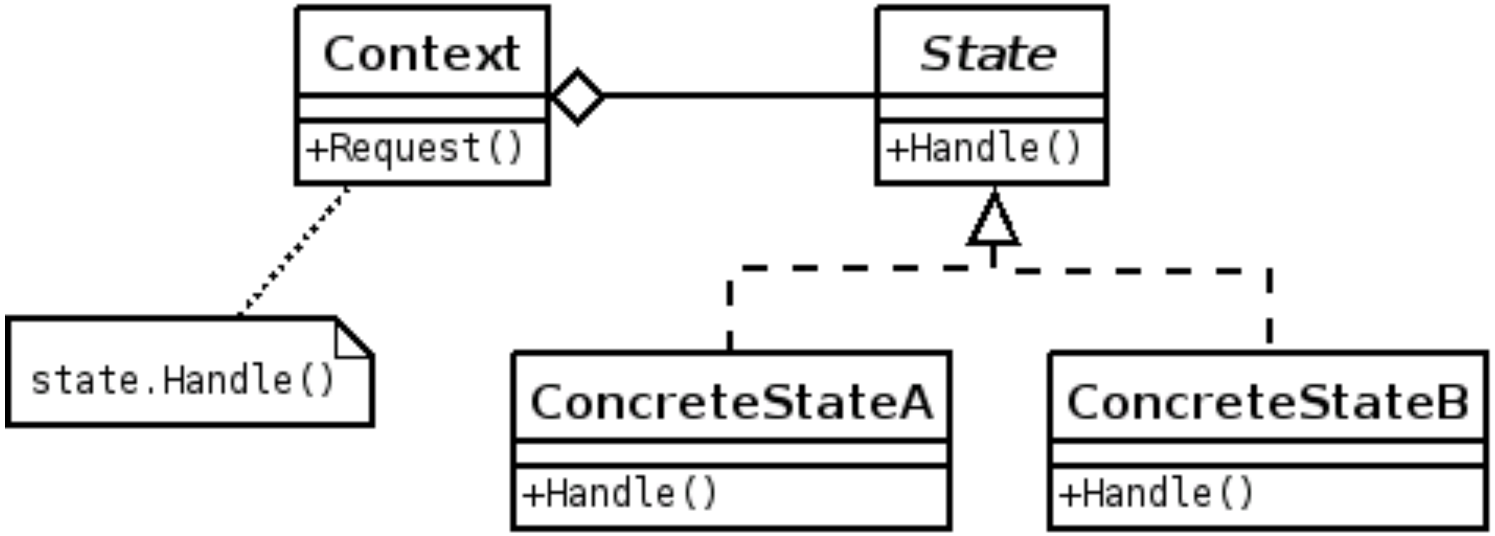
Chain of Responsibility


CreditRequestHandler
+setSuperiorRequestHandler
(theSuperior: CreditRequestHandler)
+creditRequest(amount: int)
#forwardCreditRequest(amount: int);
successor
Customer
Vendor
SalesManager
+creditRequest(amount: int)
ClientAccountManager
+creditRequest(amount: int)

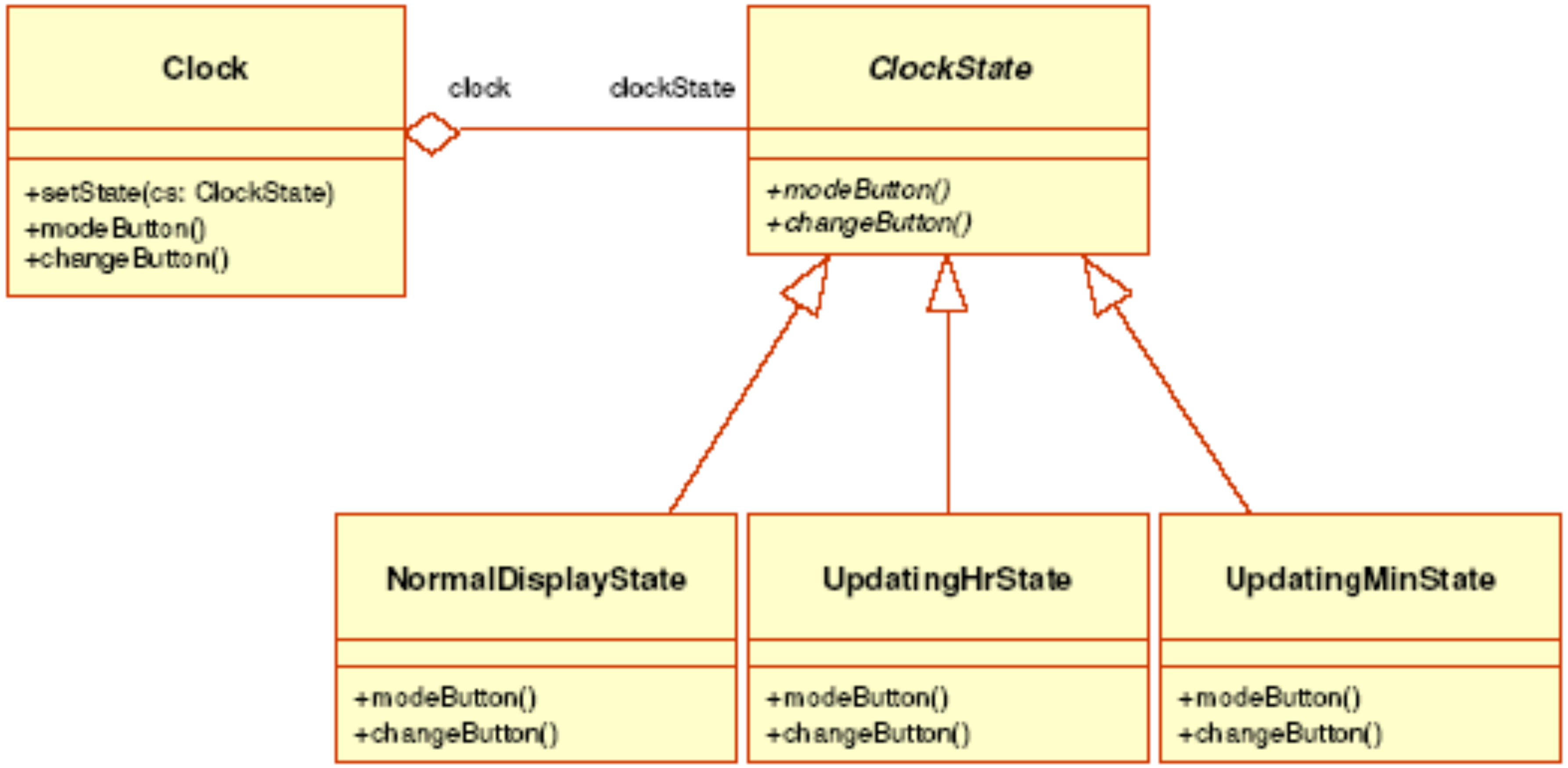
Esercizio

- Disegnare un diagramma UML che modella il seguente dominio:
  - Un orologio ha due pulsanti: MODE e CHANGE*.*
  - MODE permette di scegliere la modalità: “visualizzazione normale”, “modifica delle ore” o “modifica dei minuti”.
  - CHANGE esegue operazioni diverse in base alla modalità:
    - accendere la luce del display, se è in modalità di visualizzazione normale,
    - incrementare in una unità le ore o i minuti, se è in modalità di modifica di ore o di minuti.

State


Context
+Request()
State
+Handle()
state.Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()

State


Clock
+setState(cs: ClockState)
+modeButton()
+changeButton()
clock
clockState
ClockState
+modeButton()
+changeButton()
NormalDisplayState
+modeButton()
+changeButton()
UpdatingHrState
+modeButton()
+changeButton()
UpdatingMinState
+modeButton()
+changeButton()

Riferimenti

- E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.*
- Larman, *Applying UML and patterns, Pearson 2005.*
- Head First Design Patterns By Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates, first edition 2004.

Questions on patterns

(dal materiale del Prof. Ciancarini)

On design patterns

Which GoF pattern is based on a recursive structure?

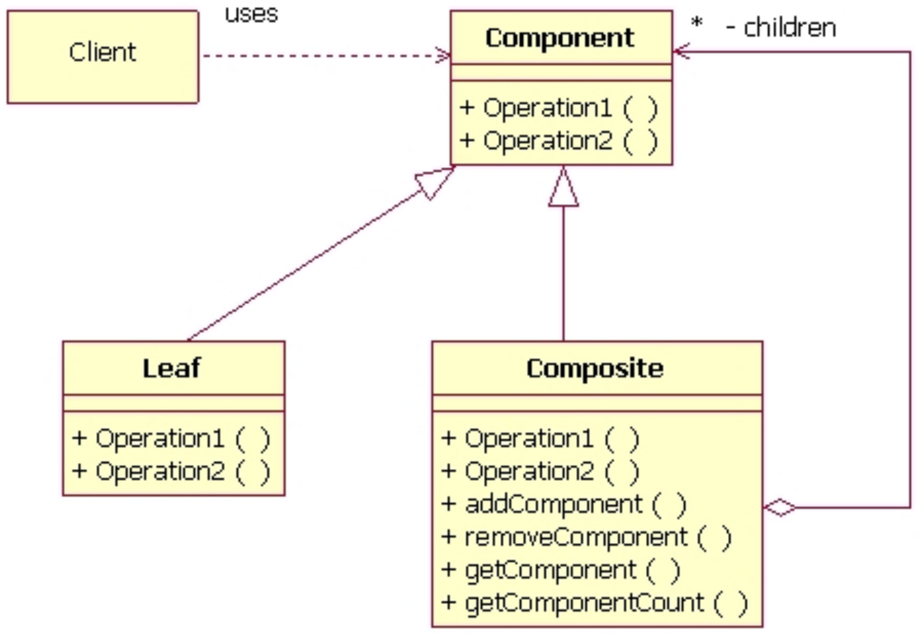
a) Bridge

b) Composite

c) Abstract factory

d) Strategy

e) Singleton



On design patterns

Which GoF pattern inspires this diagram?

a) Bridge

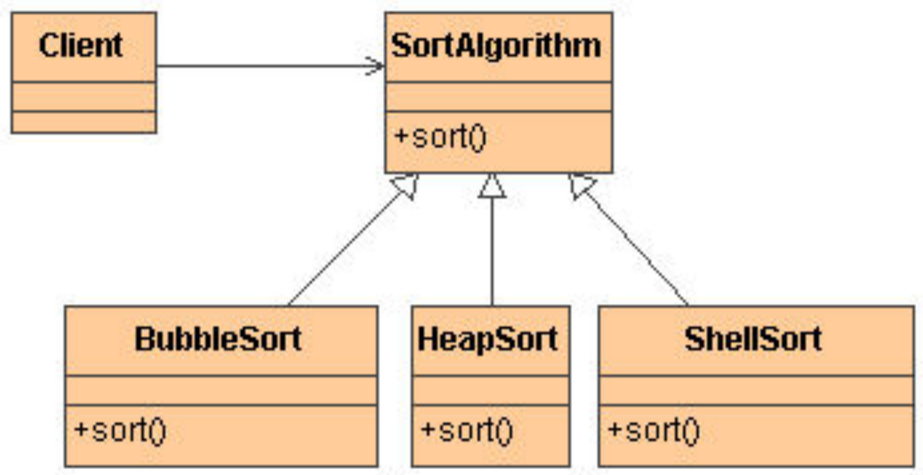
b) Composite

c) Abstract factory

d) Strategy ✓

e) Singleton

f) Facade



On design patterns

You are creating an application that simulates a technical support service provider. All requests are initially handled by front office support and are forwarded to higher levels as and when required

a)Strategy

[✓ b)Chain of responsibility]

c)Builder

d)State

On design patterns

You are enhancing an existing application in a pizza shop. The price of the pizza depends on the options selected by the user. Each option carries a different additional price. There are a large number of options available (ex: extra cheese, type of crust, toppings and so on)

a)Abstract factory

b)Strategy

c)Composite

[✓ d)Decorator]

On design patterns

You are creating an application that needs functionality for logging. You need to implement a logger and log information into a file

a) [✓ Singleton]
b) Observer
c) Chain of responsibility
d) Abstract factory

On design patterns

Which design pattern would resolve incompatible interfaces or provide a stable interface to similar components with different interfaces?

a)Controller

b)Mediator

c)Visitor

[✓ d)Adapter]

On design patterns

Which of the following statements are true for the Singleton pattern?

a)Exactly one instance of a class is allowed, it is a singleton

b)Objects need a global and single point of access

c)Define a protected method, "getInstance()", of the class that returns the singleton

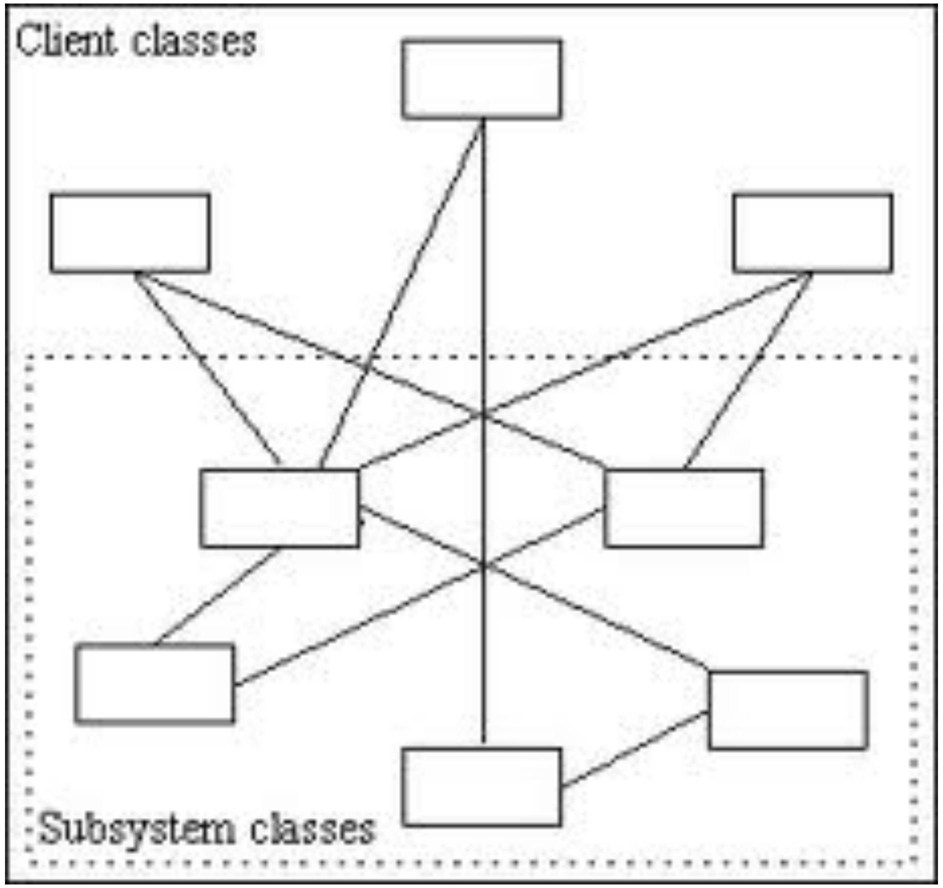
d)All of the above

e)None of the above

On design patterns

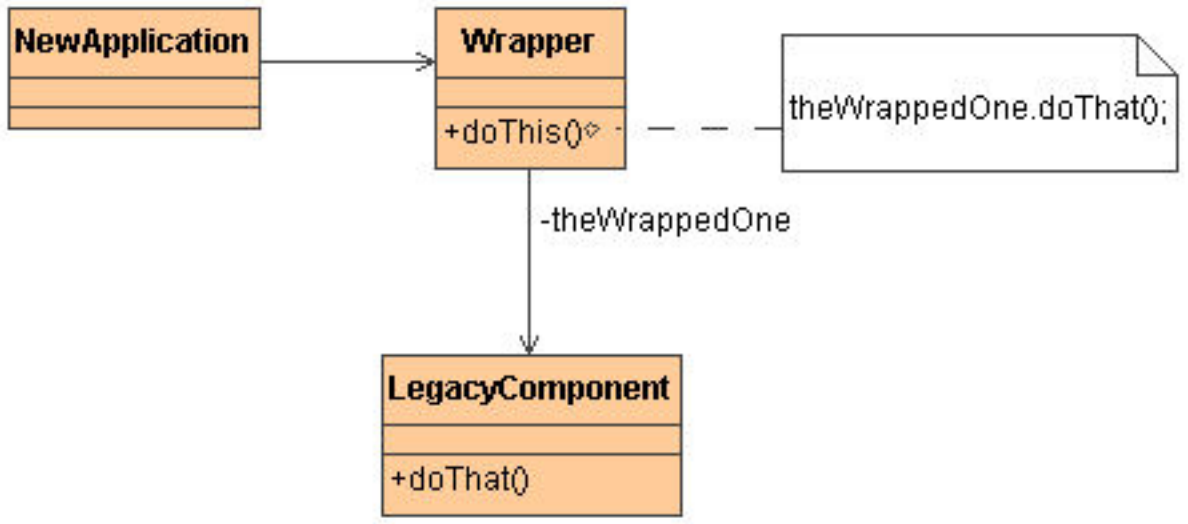
This is a problematic situation: client's classes have too many relationships with subsystem's classes.

Which pattern solves this problem?



Which pattern?

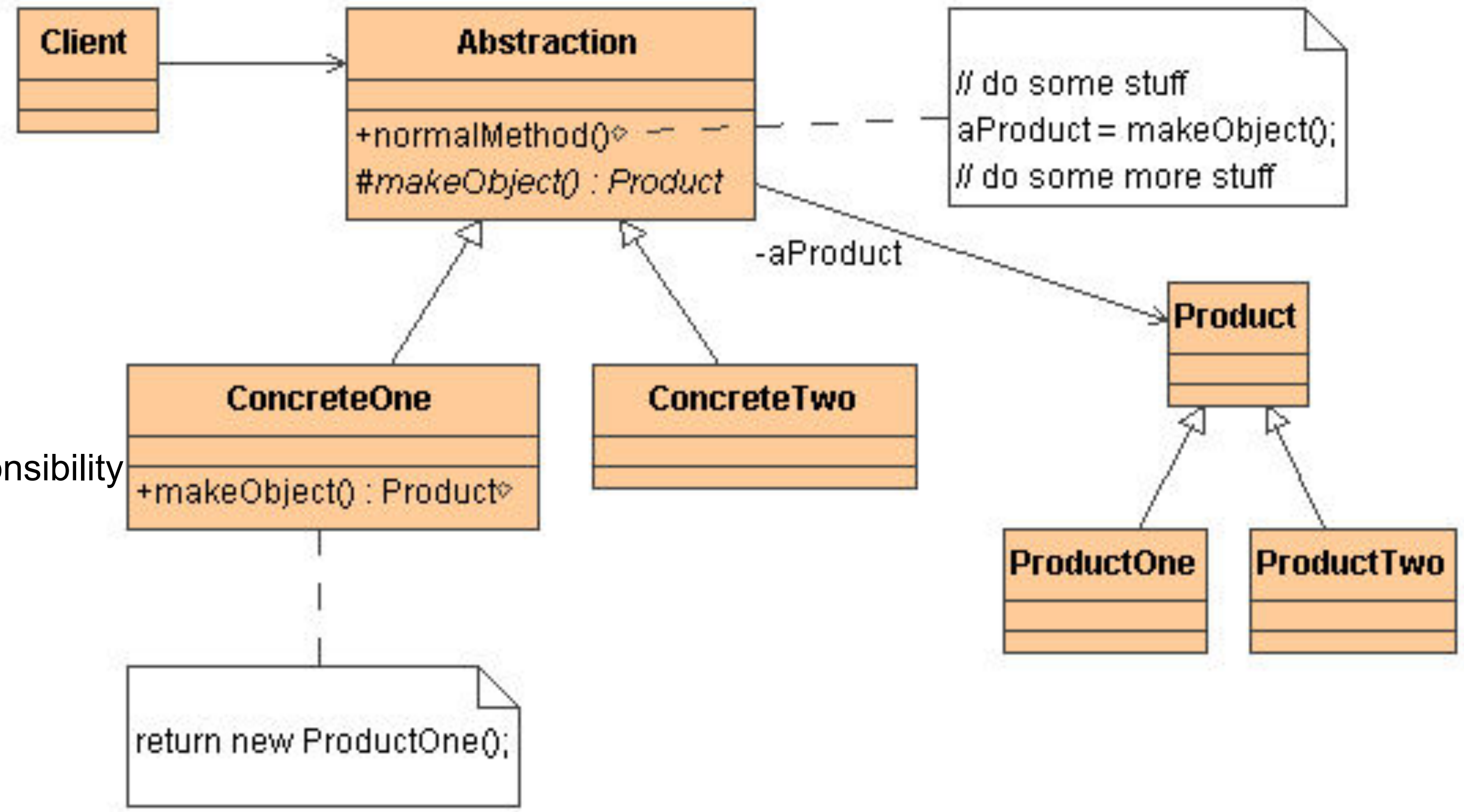
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients

Which pattern?

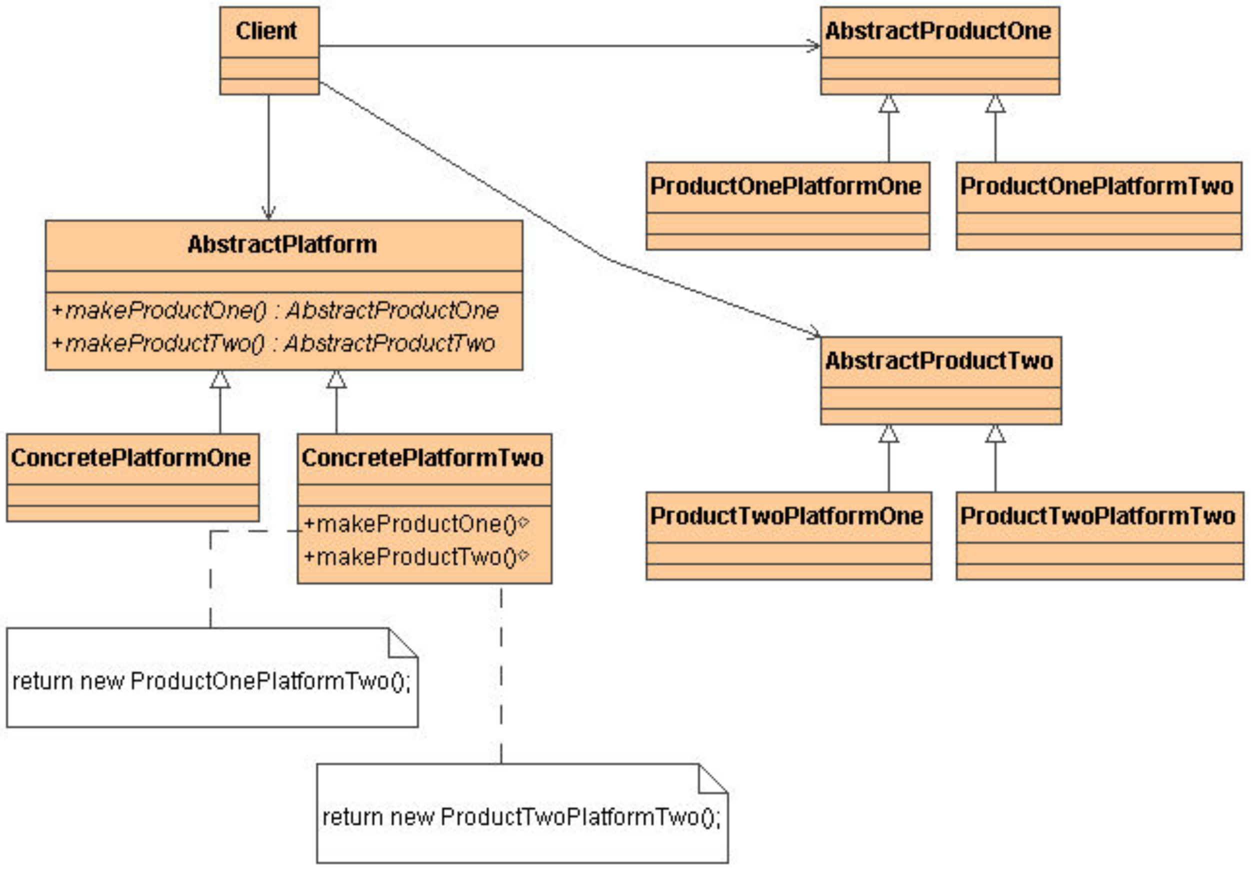
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



defines an interface for creating objects, but lets subclasses decide which classes to instantiate

Which pattern?

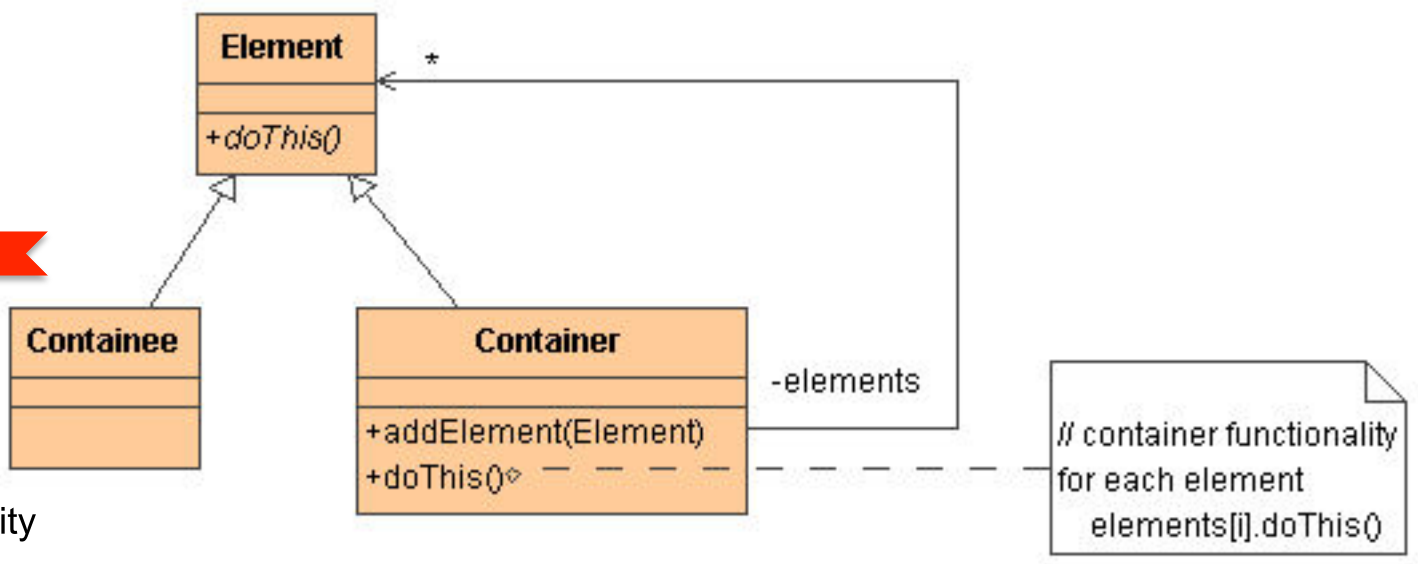
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



Provide an interface for creating families of related objects, without specifying concrete classes

Which pattern?

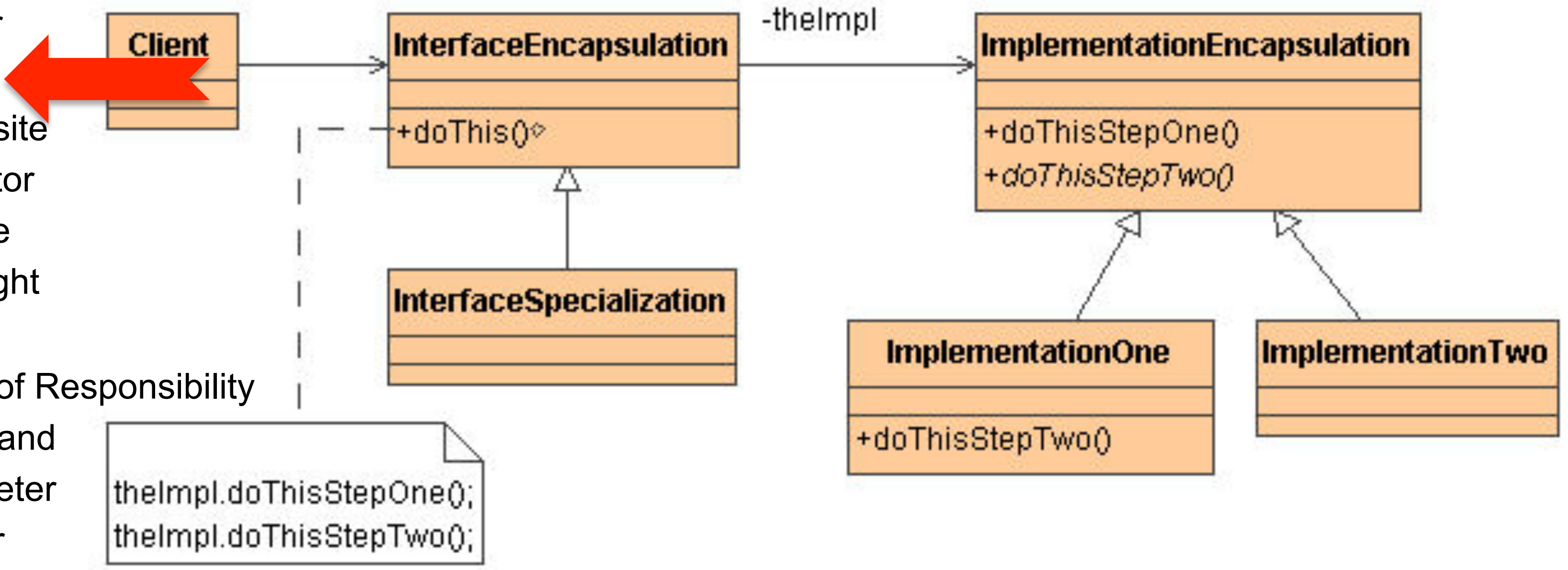
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



composes objects into tree structures and lets clients treat individual objects and compositions uniformly

Which pattern?

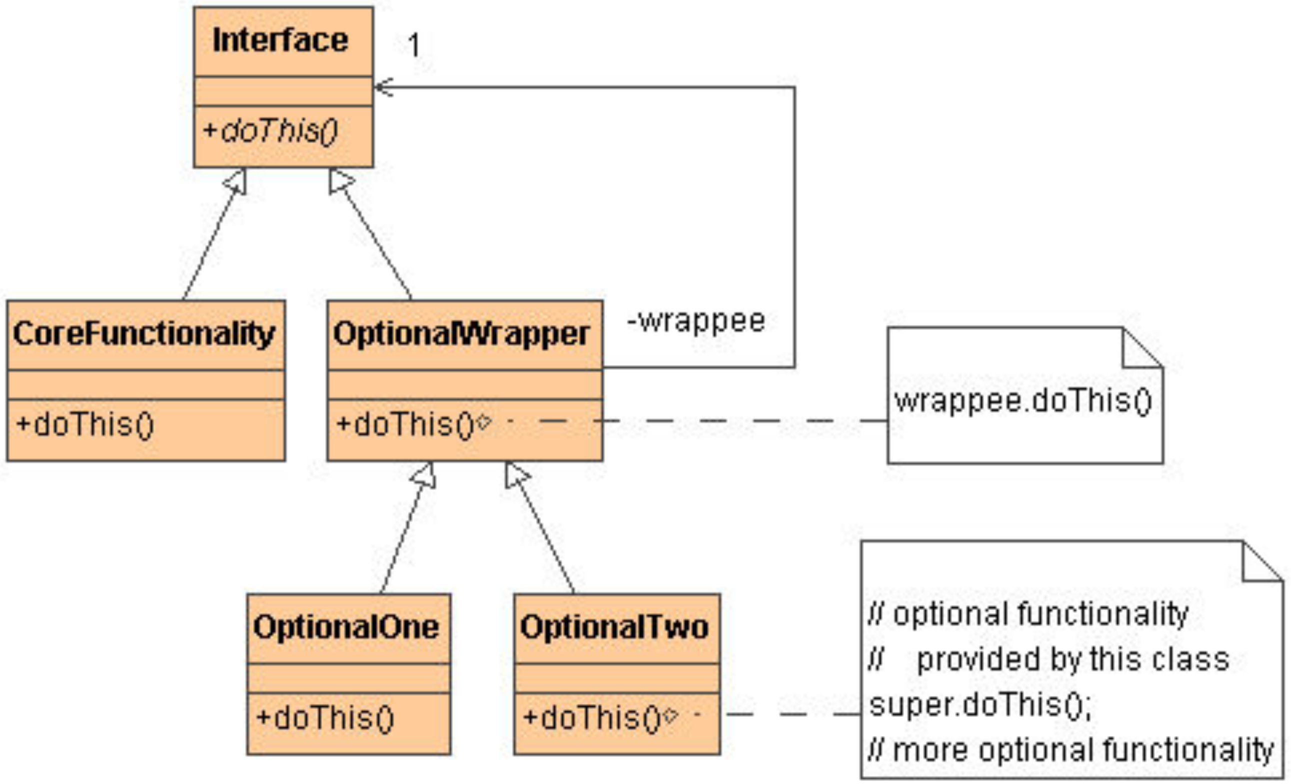
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



decouples an abstraction from its implementation, so that the two can vary independently

Which pattern?

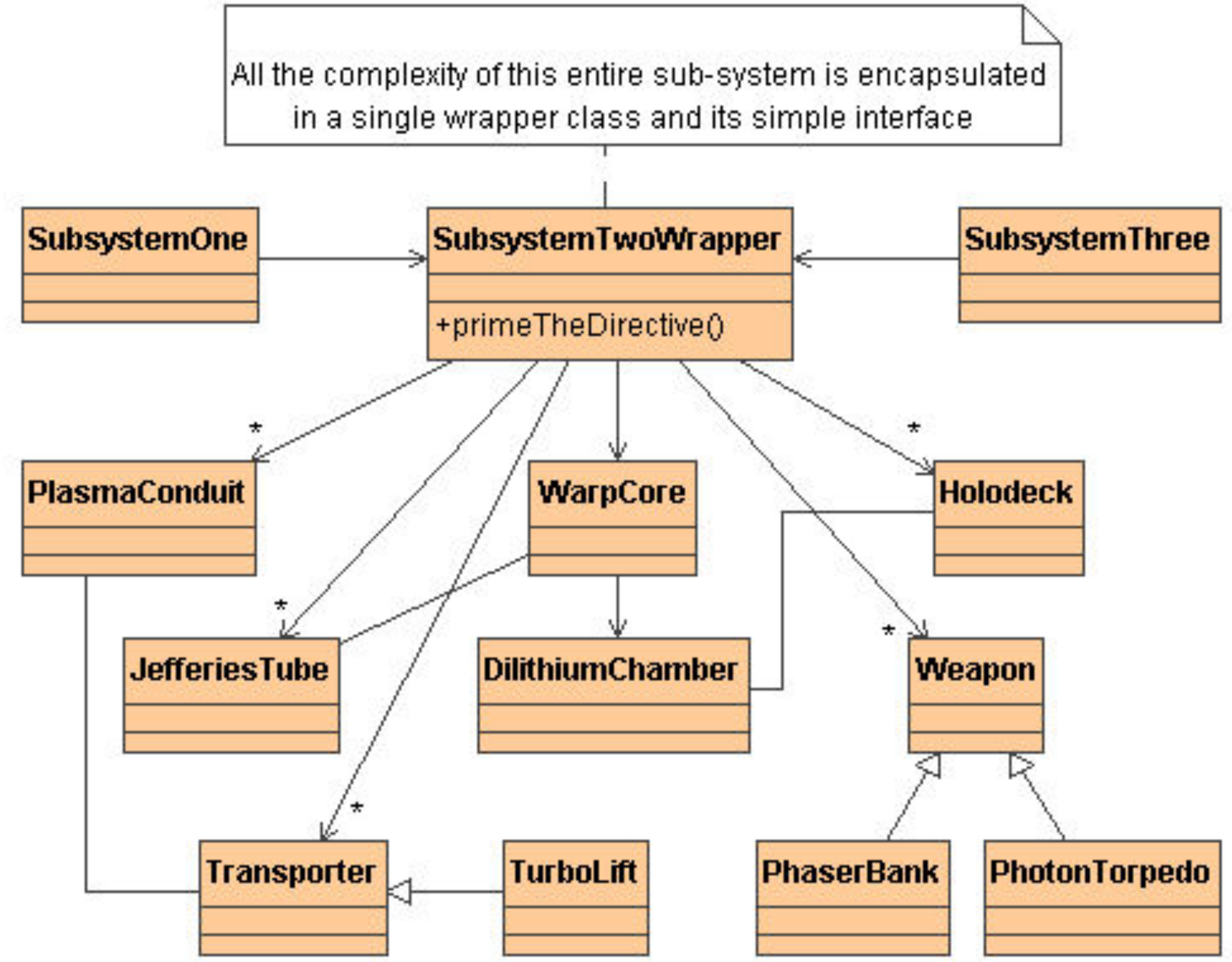
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



Attach additional responsibilities to an object dynamically. Provide a flexible alternative to subclassing for extending functionality. Recursive composition;
- 1-to-1 "has a" up the "is a" hierarchy
- a single core object wrapped by possibly many optional objects
- user configuration of optional features to an existing class

Which pattern?

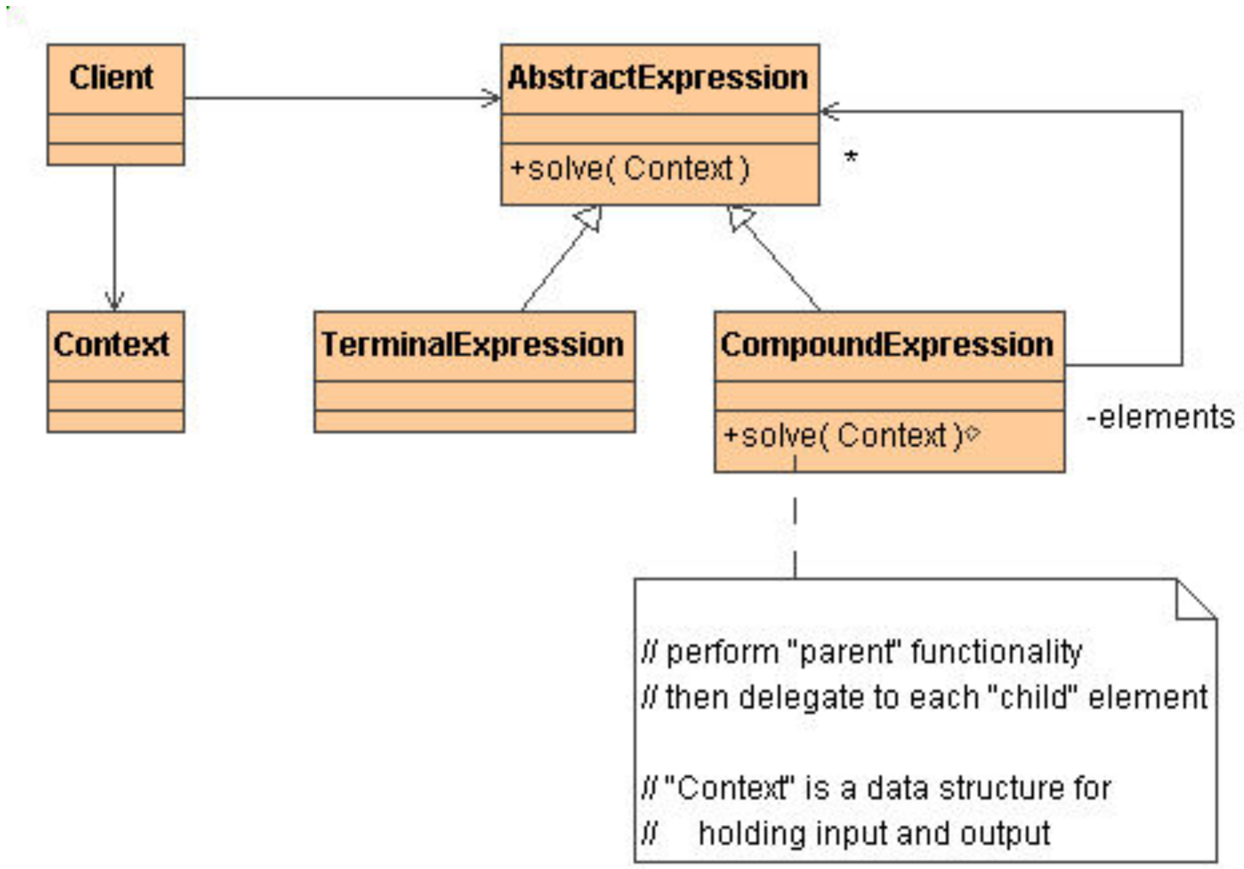
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



defines a unified, higher level interface to a subsystem that makes it easier to use

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor

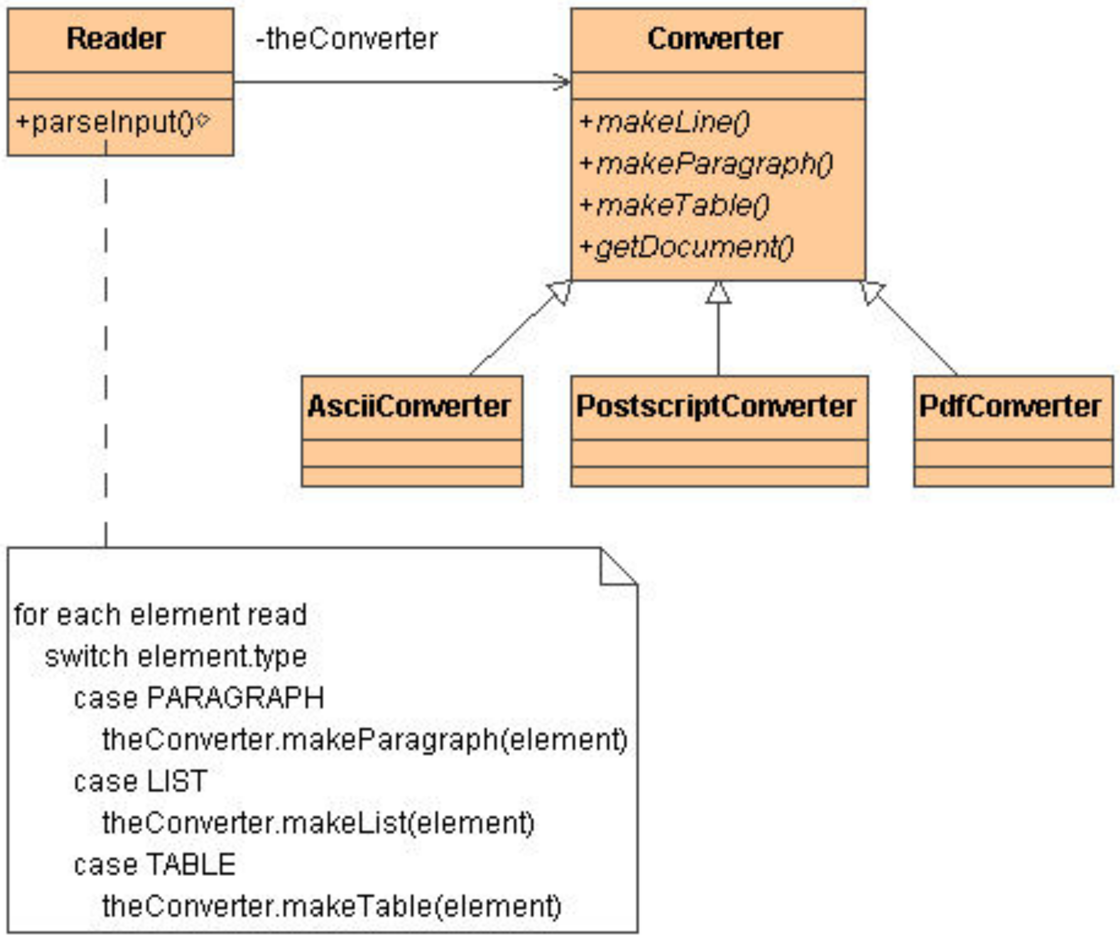
Which pattern?


Client
AbstractExpression
+solve(Context)
*
Context
TerminalExpression
CompoundExpression
+solve(Context)
-elements
// perform "parent" functionality
// then delegate to each "child" element
// "Context" is a data structure for
// holding input and output

Given a language, define a representation for its grammar along with a processor that uses the representation to parse sentences in the language

Which pattern?

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor

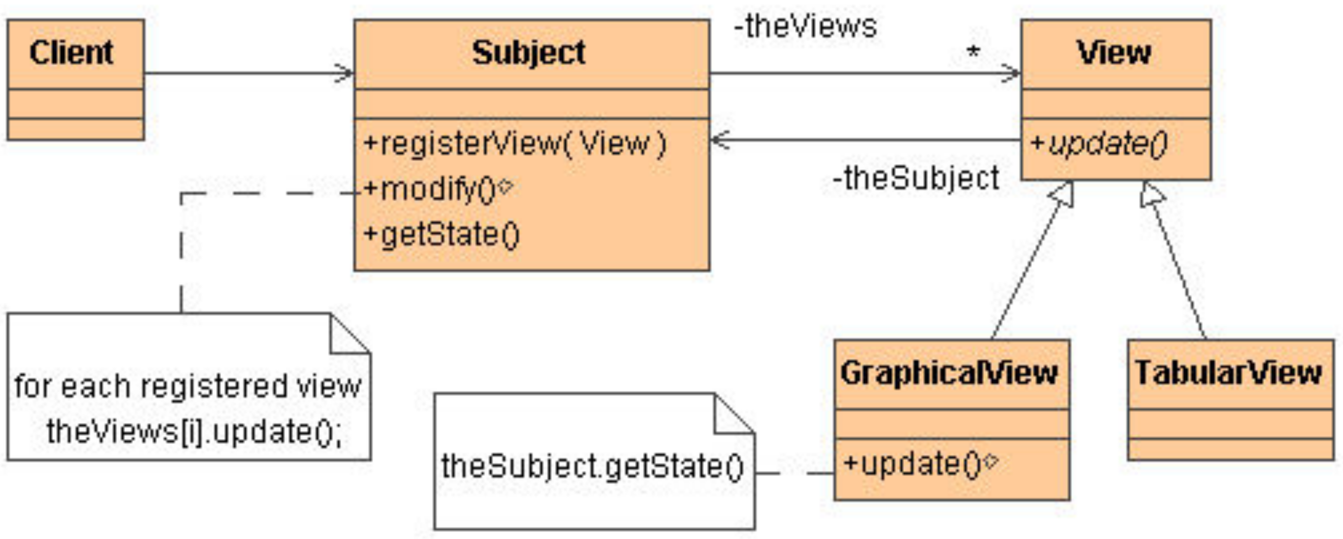


```
for each element read
  switch element.type
    case PARAGRAPH
      theConverter.makeParagraph(element)
    case LIST
      theConverter.makeList(element)
    case TABLE
      theConverter.makeTable(element)
```

Separate the construction of a complex object from its representation so that the same construction process can create different representations. One common input, many possible outputs

Which pattern?

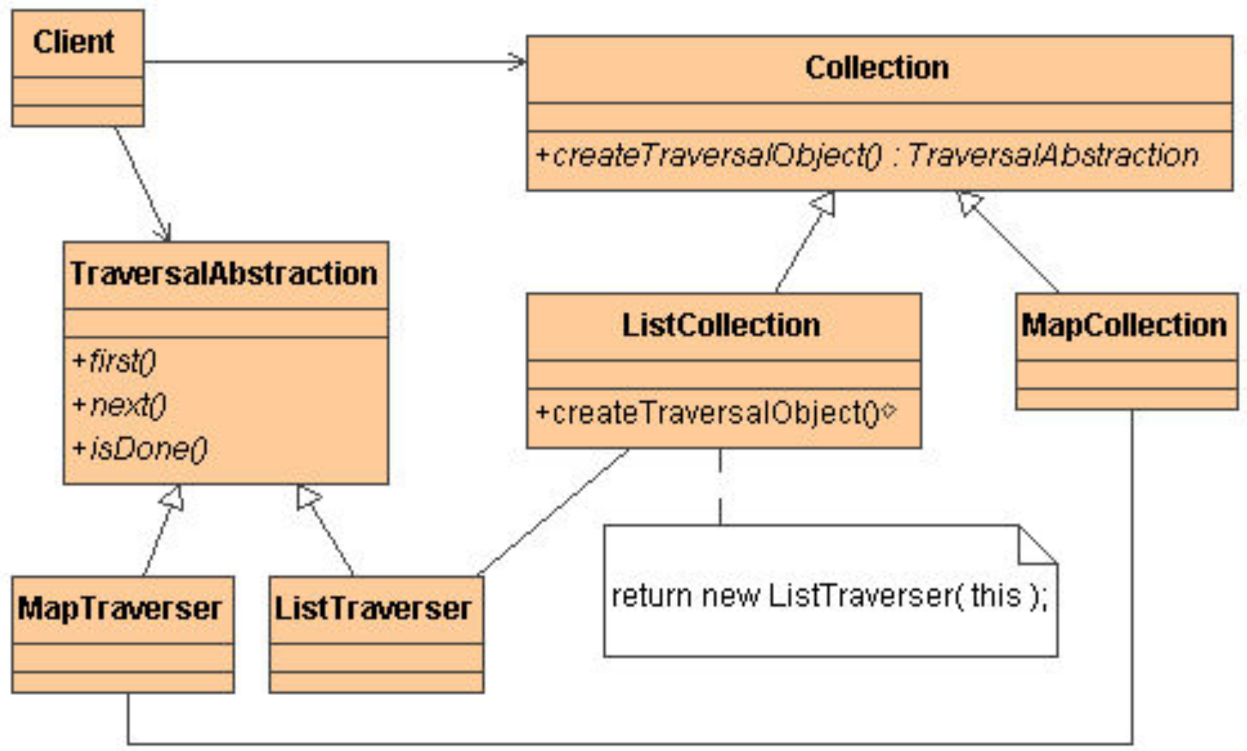
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



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

Which pattern?

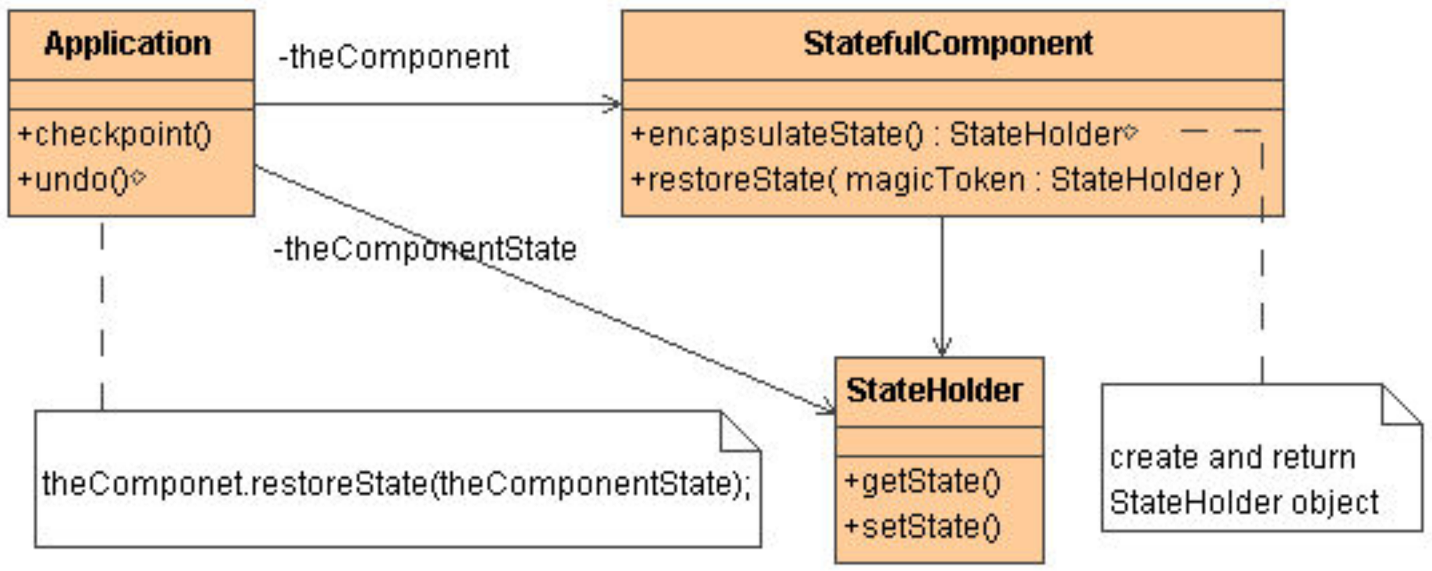
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



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

Which pattern?

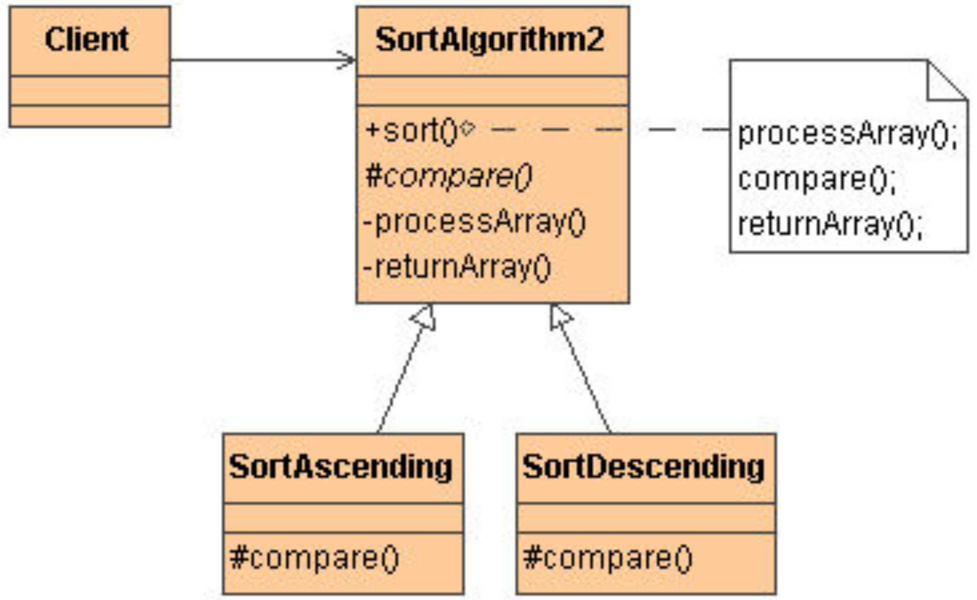
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
Undo, rollback: a magic cookie that encapsulates a "check point" capability

Which pattern?

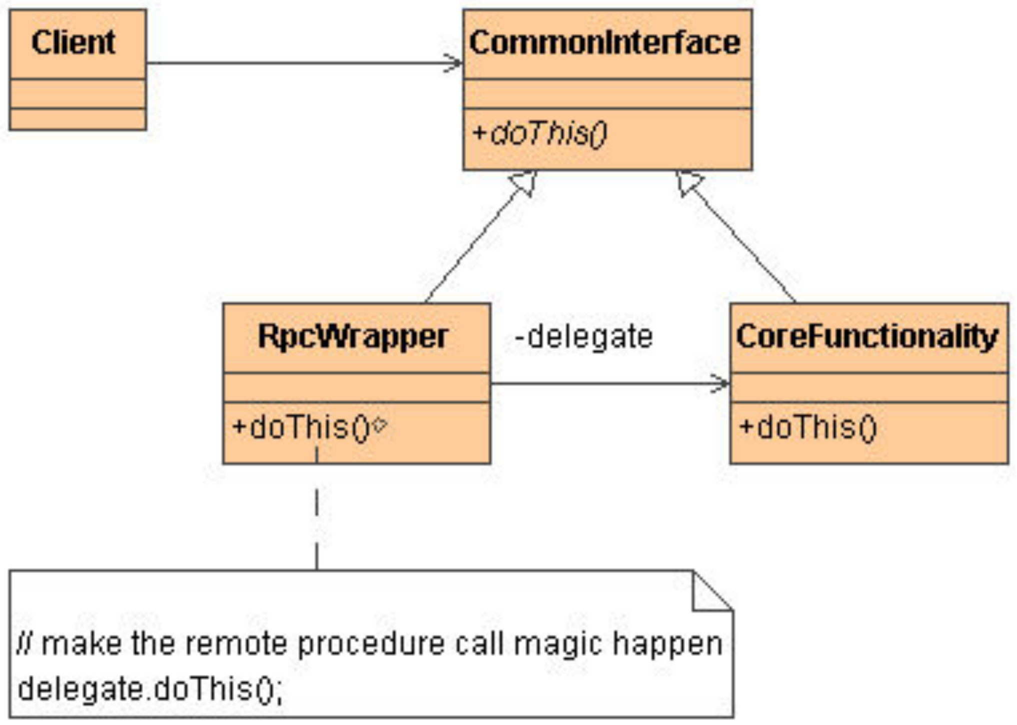
1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Which pattern?

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor



provides a surrogate or place holder to provide access to an object

Quale pattern?

- Accede sequenzialmente agli elementi di una collezione
  - Chain of responsibility
  - Iterator
  - Decorator
  - Builder

Quale pattern?

- Definisce una dipendenza tra oggetti per cui se un oggetto cambia gli altri sono aggiornati automaticamente
  – Proxy
  – Observer
  – Chain of Responsibility
  – Bridge

Quale pattern?

- Serve a rappresentare un oggetto complesso in una struttura con relazione gerarchica
  - Chain of Responsibility
  - Proxy
  - Composite
  - Abstract Factory

Quale pattern?

- Definisce un’interfaccia per creare un oggetto ma delega alla sottoclassi la creazione delle istanze
  - Factory Method
  - Builder
  - Singleton
  - Bridge

Quale pattern?

- Permette di fare wrapping di un componente legacy e ne adatta le interfacce
  - Mediator
  - Proxy
  - Decorator
  - Adapter

Quale pattern?

- Crea un’istanza di parecchie famiglie di classi
  - Abstract Factory
  - Builder
  - Factory method
  - Decorator

Quale pattern?

- Una sola classe rappresenta l’interfaccia di un intero sottosistema
  - Singleton
  - Mediator
  - Façade
  - Abstract factory

References

www.pearsonvue.com/omg/

www.vincehuston.org/dp/patterns_quiz.html

www.objectsbydesign.com/projects/umltest/bparanj-answers-1.html

dn.codegear.com/article/31863

Design Patterns Quick Reference:

http://www.mcdonaldland.info/2007/11/28/40/