Software Engineering - A.A. 13/14

Design Patterns
From the original [GOF] c++ version
with some regard to Java implementation

Enrico Vicario
Dipartimento di Ingegneria dell'Informazione
Laboratorio Scienza e Tecnologia del Software
Università di Firenze
enrico.vicario@unifi.it, www.dsi.unifi.it/~vicario
Design pattern: a reusable solution scheme for some recurrent problem
- A name, a problem, alternative solutions, known consequences, …
- Term first made explicit in Architecture by Christopher Alexander
- A Handbook of electronics schemes (e.g. amplifiers push-pull, totem-pole, …)
- ProducerConsumer or Monitor are also patterns in operating systems

Patterns in OO Design
- Advanced use of OO programming schemes
- Substitutability and polymorphism, inheritance, composition, …
- Override, method look-up or dynamic binding and virtual methods, …
- Differently mapped to different languages
The problem

- (an object of type) Subject has a state variable whose value is necessary also to (objects of type) Observer_A and Observer_B
- (this might occur in the transition from specification to implementation, or in refactoring for separation of main concerns)
- Much about maintaining consistency across decoupling

a naive solution

- (naive is a positive attribute, when beginning)
- Subject provides a public method that Observer invokes to get the value
- But, when shall the Observer update its copy of the state? … might be before using it … but, intermediate states could be relevant as well (e.g. Computing interest on a banking current account)
one step ahead

- assign to Subject the responsibility of notifying Observers (inversion of responsibility)
- after any change of its state, Subject invokes notify() which in turn invokes update() on each Observer

Yet, the Subject must be aware of all the types and instances of Observer
Some further steps ahead

- Move out from Subject the responsibility to register Observers: attach() and detach() to be used by the Observers, or by their creators (one more inversion of responsibility)
- Decouple the inherent business of a Subject from being observable (using inheritance to reuse the methods for observability)
- Abstract Observers under a common interface for homogenous collection

Call this Observer

- define a one to many dependency so that when an object changes state all its dependents are notified
- Also Known As: Publish & Subscribe (enterprise scale)
Dynamic registration of Observers on the Subject

- attach() and detach() invoked by the Observer or someone else
- Subject notifies to AbstractSubject (same object) through notify()
- notify() invokes update() on each registered Observer
- Each Observer may call-back on getState() to get the state (pull vs push mode)
The Observer pattern

- Amplification of interaction :-(
  - In principle might even be unbounded (call-backs)
  - Can be reduced by registering Observers on specific Topics, at the attach()
  - usual in the enterprise scale publish&subscribe

- Pull vs push mode
  - Pull: the Subject notifies that a change happened, and the Observer calls-back, if needed, according to its policy
  - Push: Notification from the Subject carries the state change, some parameter to distinguish among multiple subjects
  - A matter of: efficiency, robustness, coupling

- State consistency
  - When the update() is performed, the state must be “stable”, but, some action might trigger multiple "atomic" state changes
  - A change manager (Mediator pattern) may get responsibility of coordination
A system of 23 Design Patterns

- [GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns, elements of reusable Object-Oriented software"

In the next few slides:
emphasis on how a pattern is described, not on the pattern itself
**Intent**
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Also known as**
- Publish & subscribe, Dependents, (Listener)

**Motivation:**
- *<a situation where the problem may occur>*
- A common side-effect of partitioning is the need to maintain consistency between related objects, without making the classes tightly coupled, so as because that preserve reusability.
- A User Interface based on the Model View Controller

**Applicability**
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
Structure

- Described through a class diagram
- Close enough, C++ biased, but basically agnostic

```
Subject
  Attach(Observer)
  Detach(Observer)
  Notify()

ConcreteSubject
  GetState()
  SetState()
  subjectState

Observer
  Update()

ConcreteObserver
  Update()
  observerState

subject
  for all o in observers {
    o->Update()
  }
  observerState = subject->GetState()
```
Participants
- Subject: knows its observers; any number of Observer objects may observe a subject; provides an interface for attaching and detaching Observer objects.
- Observer: defines an updating interface for objects that should be notified of changes in a subject.
- ConcreteSubject: stores state of interest to ConcreteObserver objects; sends a notification to its observers when its state changes.
- ConcreteObserver: maintains a reference to a ConcreteSubject object; stores state that should stay consistent with the subject's; implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations
- ConcreteSubject notifies its observers whenever a change occurs
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information.
- ConcreteObserver uses this information to reconcile its state with that of the subject.

<Class, Responsibility, and Collaborations cards>
collaborations often documented through a sequence diagram
Consequences

- *<Good and bad news>*
- *As main consequences you can:*
  - vary subjects and observers independently;
  - reuse subjects without reusing their observers, and vice versa;
  - add observers without modifying the subject or other observers.

- *Abstract coupling between Subject and Observer:*
  - Subject doesn't know the concrete class of any Observer
- *Support for broadcast communication.*
- *Unexpected updates:*
  - since Observers do not know of each other, unexpected updates may occur :-(
Implementation

- **Observing more than one subject:**
  an Observer may depend on many Subjects;
  in this case, extend the update() interface to let the Observer know
  which Subject is sending the notification (a parameter, or a self reference).

- **Who triggers the update?**
  Let state-setting methods of the Subject call notify():
  simple, but possibly expensive with multiple consecutive changes;
  Or else, make clients responsible for calling notify() at the right time:
  is more efficient and flexible, but puts responsibility over (unknown) clients

- **Dangling references to deleted Subjects**

- **Make sure Subject state is self-consistent before notification:**
  unexpected notification may occur
  (e.g. a Subject subclass call inherited operations);
  document which Subject methods invoke notify()

- **Update protocols:**
  push/pull notification:

- **Specifying modifications of interest explicitly:**
  specify topics/aspects of interest to limit interaction and update

---

Sample Code

- `<pieces of code, in c++>`
Known uses
- Model view controller
- *<often coming from a superior yet past world>*

Related Patterns
- Mediator: between Subjects and Observers, to encapsulate complex update
- Singleton: to ensure single instance of the Mediator
[GOF] Design Patterns refer to C++
- but, patterns are by construction somehow language agnostic: no impact on WHEN, minor impact on STRUCTURE, some impact on IMPLEMENTATION

What changes from c++ to java?
- <intentionally incomplete>
- Method lookup vs dynamic binding and virtual methods
- single inheritance, interfaces, and abstract classes,
- Protected and package-private visibility
- Garbage collection
- Nested classes
- Concrete support from standard API
- <Often an opportunity more than problem to fix>

Some references to work with
- J.W.Cooper, "The design patterns java companion"
- F.Guidi Polanco "GoF's Design Patterns in Java"
- J.Bloch, "Effective Java"
- ... GOF, "Design Patterns, elements of reusable OO software"
Patterns are often supported or applied by classes in standard APIs.

### class java.util.Observable

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>addObserver (Observer o) Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.</td>
</tr>
<tr>
<td>protected void</td>
<td>clearChanged () Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the hasChanged method will now return false.</td>
</tr>
<tr>
<td>int</td>
<td>countObservers () Returns the number of observers of this Observable object.</td>
</tr>
<tr>
<td>void</td>
<td>deleteObserver (Observer o) Deletes an observer from the set of observers of this object.</td>
</tr>
<tr>
<td>void</td>
<td>deleteObservers () Clears the observer list so that this object no longer has any observers.</td>
</tr>
<tr>
<td>boolean</td>
<td>hasChanged () Tests if this object has changed.</td>
</tr>
<tr>
<td>void</td>
<td>notifyObservers () If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.</td>
</tr>
<tr>
<td>void</td>
<td>notifyObservers (Object arg) If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.</td>
</tr>
<tr>
<td>protected void</td>
<td>setChanged () Marks this Observable object as having been changed; the hasChanged method will now return true.</td>
</tr>
</tbody>
</table>

### interface java.util.Observer

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>update (Observable o, Object arg) This method is called whenever the observed object is changed.</td>
</tr>
</tbody>
</table>
The Adapter pattern

Intent
- Convert the interface of a class into another interface that clients expect

Also Known as
- Wrapper

Motivation
- an already implemented Adaptee provides operations needed by a new client
- but the operations of Adaptee do not match the expected interface
  - mismatch on names, signature, aggregation of operations, …
Class adapter
- Based on inheritance

Object adapter
- Based on composition
Implements the interface of target by extending the adaptee
A single object instance :) 
- simple to create, and to debug

Adapter does not an override of the adaptee (method lookup) 
- … which might be not bad news, to avoid fragility due to unknown override :(  
- The override can be done anyway in the adapter
The adapter implements the target interface by composition of the adaptee (forwarding)

```c
void op1(void)
{
    adaptee->opA();
    adaptee->opB();
}
```
Different implementations of the adaptee can be composed :)  
- the adaptee can be an interface or an abstract class  
- The constructor of the adapter shall install the specific concrete adaptee  

The adapter is stateless, and can be implemented as a singleton

Two objects will be instantiated and composed :-(

In principle, the adaptee could be replaced dynamically :-)
- viceversa, this might be a feature to avoid, through an immutable field

The adaptee is not aware to be part of a composition (self problem)

In the end, this is the basic trade-off between inheritance and composition
A library
- a set of functions that you can call, these days usually organized into classes. Each call does some work and returns control to the client.

A framework
- embodies some abstract design, with more behavior built in. To use it you insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points.

"One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application." (Ralph Johnson and Brian Foote)

http://martinfowler.com/bliki/InversionOfControl.html
The shift from libraries to frameworks is much about inversion of control in who-calls-whom vs who-reuses-whom:

- new code invokes reused objects of a library
- new code is invoked by the objects in a reused framework
- (inversion of control, Hollywood principle, … )
Consider a framework providing an abstract implementation for responsibilities applicable to a variety of types

- E.g. wrap a payload within an envelope, send the envelope, …envelope of what?

The framework may prepare installation of an adapter to facilitate the concrete instantiation

- Note that here the adapter is designed in advance, not as a retrofit

Two schemes

- Abstract operations
- Delegate objects
**FrameworkClass**

- Is part of the framework
- Defines a virtual narrow interface (aka primitive methods):
  a minimal set of methods that can assume all responsibilities
  that depend on the specific type of managed objects

**FrameworkAdapter**

- Invokes concrete methods in the specific domain
- ... to implement operations in the narrow interface
Pluggable adapters: delegate objects

Delegate
- specifies a narrow interface covering context dependence

FrameworkClassObj
- Delegates context dependent operations to LibClassDelegate
- Exposes a method setDelegate() enabling installation of the delegate

FrameworkAdapterObj
- Implements Delegate using SpecificClass
- Installs itself in FrameworkClassObj as delegate (e.g. in the constructor)

Client

op1()
{ delegate. op1(); }

op2()
{ ... op1(); }

FrameworkClassObj

-delegate
+op1()
+op2()
+op3()
+setDelegate()

Delegate

-delegate
+op1()

<<NarrowInterface>>

SpecificClass

+opA()
+opB()

<<Adaptee>>

FrameworkAdapterObj

+op1()
+FrameworkAdapterObj(anObject: Object)

- specificObj

<<Target>>

FrameworkAdapterObj
delegator:Object, specificObj:Object
{
    this. specificObj=specificObj;
    delegator. setDelegate(this);
}

theFramework

anInstance

op1() is a primitive method encapsulating all the dependency on classes in the instance. For some specific instance this dependency is resolved by opA() and opB().
The Factory Pattern

Intent

- define an interface for creating an object, but let subclasses decide which class to instantiate.

Motivation

- Frameworks use abstract classes to define and maintain relationships between objects.
- E.g. implement the lifecycle of a document, applicable to any type of document: the framework implements the concept that an Application manages a collection of Documents that can be opened, processed, saved, closed; a specific Application (instance) will determine the specific structure of Document, and the way how operations are concretely performed.
The Factory Pattern

Structure (and Participants and Collaborations)

- Creator is an abstract class:
  anOperation() is implemented by delegating
to a not-implemented factoryMethod() the creation of a Product
- Product is an interface
- ConcreteCreator implements the factoryMethod(),
dertermining the specific Product
**Consequences**

- **Main**: eliminate the need to bind application-specific classes into your code, and thus make your code (framework) reusable.

- **Provides hooks for subclasses**: Document could define `createFileDialog()` that creates a default file dialog object for opening an existing document; a subclass of Document defines an application-specific override; (in this case the factory method is not abstract and provides a reasonable default).
Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Also Known As

- Kit

Client delegates to the AbstractFactory the creation of Products

The choice of the concrete implementation for AbstractFactory determines the line in which Products will be instantiated.

(in Java, AbstractFactory is an interface better than an abstract class)
Consequences
- Localizes the choice of the line of Products :-) 
- rende agile la modifica della famiglia di prodotti :-) 
- Guarantees consistence of the line of Products :) 
- Hurdles the addition of a new type of Product :-(

Who chooses the concrete implementation for the Abstract Factory?
- The client itself 
- A third object that sets up the all scheme 

AbstractFactory holds no state and can be implemented as a Singleton
Construction of a GUI

- Made of buttons, Windows, Scrollbars, …
- running under Windows or Osx
Language idioms and design patterns

Idiom
- An elementary scheme, with direct mapping on some language construct
- An effective way to use some language construct
  - In C, passing the address value; in Java, using a static factory method, …
- "Effective Java" by J. Bloch is much about idioms for Java

Design pattern
- Still a scheme, of larger size, usually involving multiple classes
- More language agnostic, using idioms in the concrete implementation
  (Opposite view: a way to reproduce something that the language misses)
- "Design Patterns" by GOF: C++ oriented, but applicable on Java as well

Both are about reusing solutions
- Characterizing recurrent problems, alternative choices, consequences, …

Incrementally involved in the incremental transition
- from a specification model to an implementation (design patterns)
- … and then to the concrete implementation (idioms)
  (Tutorial view: Java language -> Java Idioms -> Design Patterns (in Java))
creational patterns:
- E.g. the **Abstract Factory** Provides an interface for creating families of related or dependent objects without specifying their concrete classes
- Abstract about the process of object creation: encapsulate knowledge about which concrete classes the system uses, and hide how instances of these classes are created and put together; to provide flexibility in what gets created, by whom, how and when.
- *<no counterpart for managing destruction? A good reason for Java>*

structural patterns:
- E.g. the **Adapter**: converts the interface of a class into another interface that some client expects.
- about how classes and objects are composed to form larger structures: *class* patterns use inheritance to compose interfaces or implementations; *object* patterns compose objects to realize new functionality.

behavioral patterns:
- E.g. the **Observer**: defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified
- about abstracting on the control-flow of operations in different objects, to shift the focus from control-flow to the way objects are interconnected
Intent
- Creates a surrogate or placeholder that controls the access to another object

E.g. lazyness principle in the construction of a medical record
- The textual part is light; and locally available
- Images are stored elsewhere

... also in displaying a resultset or a complex page
Make the client unaware of proxy intermediation
Make the proxy unaware of the specific type of the real object
- Localize the choice at the time of installation
Proxies can serve for various purposes

- Virtual proxy: virtualizes operations on some object (e.g. lazy load)
- Remote proxy: a local placeholder for some remote object
- Protection proxy: e.g. for authentication, authorization, access control
- Smart reference: Replaces a reference and adds behavior on the invocation (e.g. count or limit the number of accesses)

... but it is not possible chaining of multiple proxies layering specialized responsibilities

- a Proxy can refer to a Subject, but not to another Proxy
Evolving the Proxy structure to enable chaining

1. Initial structure: The proxy is associated to a Subject
2. The proxy can also be associated to another proxy
3. Proxies can have different types

Chaining is now allowed
Call this a Decorator

- Evolution of a Proxy towards repetitive composition

What can this serve for?

- Dynamically add responsibilities so as to extend its responsibilities

- Patterns are discovered rather than invented, they emerge through evolution
  - (on overdesign and refactoring)
Il tipo cittadino è caratterizzato da attributi diversi in diversi contesti

- Anagrafe, sanità, lavoro, ...

In ciascun contesto è implementata, in modo diverso, l’operazione di costruzione del descrittore
Un solo oggetto :-)  
Di facile installazione :-)  

La scelta delle responsabilità è statica ;-(  
Le combinazioni esplodono :-(
Cittadino_base è il "modello base"

Un decorator è associato a un oggetto decorato
  - Esegue un suo comportamento e (prima o dopo) forwarda all’oggetto decorato

I decorators sono differenziati
Add an AbstractDecorator
- maintains a reference and performs a forward
  (implemented but abstract as not functional)
- often omitted in the practice
Il numero delle classi scala in modo lineare :-)

N+1 oggetti :-(
  - Difficoltà nel debug

Necessaria l’installazione iniziale :-|
  - A carico del costruttore del decoratore (riceve il riferimento all’oggetto decorato)

Efficienza :-(
  - Forwarding
  - Le classi alte devono essere leggere

Configurazione modificabile dinamicamente :-)
  - Addition and withdraw
le decorazioni modificano il comportamento ma non l’interfaccia
- spesso la decorazione implica operazioni specifiche
- e.g. il cittadino nel profilo dell’assistenza sanitaria ha delle operazioni (scelta del medico, set del numero di codice, …) che non sono rilevanti nel profilo anagrafico, e viceversa

ciascuna funzione rilevante in qualche decorazione deve essere dichiarata nella classe base
- la classe base raccoglie responsabilità non coesive
- ciascun decoratore deve implementare il forwarding su operazioni che non sono per lui rilevanti
- gli oggetti si appesantiscono in termini di occupazione di memoria

Questo limita molto il campo di applicazione dello schema

Dire che due oggetti hanno la stessa interfaccia significa che hanno le stesse operazioni, a livello concettuale
non basta che ereditino da una classe base comune
Decorator
- aggiunge comportamento senza modificare l’interfaccia
- E per questo può essere applicato ripetitivamente
- Adapter aggiusta l’interfaccia realizzando lo stesso comportamento

Adapter
- Adatta una implementazione disponibile a una interfaccia attesa

Proxy
- Non cambia né l’interfaccia né il comportamento
- Ha l’intento di placeholder

Bridge
- ha l’ intento di separare a priori (up-front) l’implementazione dall’astrazione per favorire l’evoluzione disaccoppiata delle due
- Adapter risponde piuttosto ad un approccio a posteriori

Decorator and Aspect Oriented Programming
### Intent
- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

### Motivation
- A configuration is often made by compositions of components, that, in turn, can be either elementary or made by composition.
- E.g. the elements of a graphical interface, the structure of a Computer SW Configuration Item (CSCI), a system made of subsystems in mechanics or electronics, ...
- Another perspective: a structure of classes that produces a tree of objects.

![Composite Pattern Diagram]

- The key to the Composite pattern is an abstract class that represents both primitives and their containers, so that a client is not requested to distinguish leaves from intermediate nodes.
Structure (of classes, and objects)

- **Component**: declares the interface for objects in the composition; implements default behavior for child-related operations.
- **Leaf**: defines behavior for primitive objects in the composition.
- **Composite**: defines behavior for child-related operations; stores reference to child components.
- **Client**: manipulates objects in the composition through the Component interface, without distinguishing Leafs from Composites.

- Default behavior in the Component is an example of the evil that occurs when a base class is charged of operations not shared by all derived classes.

![Composite Diagram]

**Participants**

- Component: declares the interface for objects in the composition; implements default behavior for child-related operations.
- Leaf: defines behavior for primitive objects in the composition.
- Composite: defines behavior for child-related operations; stores reference to child components.
- Client: manipulates objects in the composition through the Component interface, without distinguishing Leafs from Composites.
Consequences

- defines class hierarchies consisting of primitive objects and composite objects.
- makes the client simple
  by allowing uniform treatment of composite structures and individual objects
- makes easy to add new kinds of components:
  newly defined Composite or Leaf subclasses work automatically,
  and Clients don't have to be changed.
Maximizing the Component interface

- the first aim is transparency: make client unaware of Leafs and Composites.
- To this end, Component declares all operations of Leaf and Composite.
- syntactic question: how does the interface become concrete?
- behavioral question: what if add() is invoked on a Leaf object? (aim of safety)

Component is an abstract class

- provides a "void" implementation for child-related operations overridden by Composite, and used by Leaf.
- ... possibly rising an exception, which should not happen: the pattern is not for the case of a Leaf be promoted to Composite; neither for for providing robustness in the creation; much more for making traversal easy when the structure is up.

The solution is standard but violates a principle of class hierarchy design

- a class should define only operations that are meaningful to all subclasses.
explicit parent references
- a child can maintain a reference to its (unique) parent to facilitate traversal
- usually charged to Component
- In this case, special behavior occurs on the root

Sharing components
- when parent reference is not explicit, the same (equal) Component can appear in multiple Composites
- to reduce storage, but also to guarantee consistency
When you program you see classes, but behavior, execution, debugging, and testing are much about objects

- the relation is often not direct
- the line of evolution proxy->decorator->composite is a good example

- by the way, note that they are all classified as structural (convert the interface of a class into another interface), but they also much support the purpose of behavioral patterns: (shift the focus from control flow among objects to composition of classes)
On topologies of classes and objects - 2/2

### Proxy

- **Client**
- **Subject**
  - `Request()`
- **RealSubject**
  - `Request()`
- **Proxy**
  - `realSubject`
  - `Request()`

### Decorator

- **VisualComponent**
  - `Draw()`
- **TextView**
  - `Draw()`
- **Decorator**
  - `component` ➔ `Draw()`
- **ScrollDecorator**
  - `Draw()`, `scrollTo()`, `scrollPosition`
- **BorderDecorator**
  - `Draw()`, `drawBorder()`, `borderWidth`

### Composite

- **Component**
  - `Client`
  - `Operation()`, `AddComponent()`, `RemoveComponent()`, `GetChildren()`
- **Leaf**
  - `Operation()`
- **Composite**
  - `Operation()`, `AddComponent()`, `RemoveComponent()`, `GetChildren()`

```
      aClient
           ▼
        /   \
Proxy  aProxy aRealSubject
           ▲     ▲
         /   \
RealSubject   Subject   \            
             ▲                  \         
            /                    \       
           Proxy               aProxy
                          ▲       
                        /         
                      Reqest()  ▲
                       ▼          aRealSubject
Component
          ▲
        /  \
    aComposite aComposite aLeaf
           ▲     ▲     ▲
         /   \
      aLeaf aLeaf aLeaf
```

- Enrico Vicario - AA 13/14 SW Engineering
Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - "<<Relevance of the latter line could be questioned?>>"

Motivation

- Separate the responsibility of knowing the sequence of construction from the ability to perform single steps also with different implementations
- E.g. assemble a Composite with different types of Leaf
- E.g. assemble a memory structure based on the parsing of some XML file

Participants

- Builder: specifies an abstract interface for creating parts of a Product object.
- ConcreteBuilder: implements the Builder interface; maintains the state of the Product under construction, provides a method to retrieve the Product
- Director: determines the sequence of construction of parts
Client
- creates a Director and gives it a ConcreteBuilder (stored as a Builder)

Director
- on invocation of construct(), implements the construction through properly sequenced commands to the Builder

Client
- finally retrieves the Product from the ConcreteBuilder through getResult()
Client view of the process:
- creates a PizzaBuilder, choosing the concrete version for a specific type of Pizza
- passes it to the constructor of a Waiter (Director);
- invokes construct on the Waiter;
- get the Pizza (Product) from the PizzaBuilder
Client
- depends on the particular type of Pizza (Product) only at the creation of the Builder

Waiter (Director)
- Sequences the steps of construction (first the pasta and then the dressing), but is not aware of how the steps are performed

Builder
- Performs the steps (add pasta, add condimento), maintains a representation of the Product and returns it through the method getPizza()
- is not aware of the sequence of steps

Positive consequences follow
- Change the sequence without affecting creation of parts, and viceversa
- Support multiple versions of parts (really relevant?)
**Builder vs Abstract Factory**
- Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (simple or complex).
- Builder returns the product as a final step, with an Abstract Factory, the Product gets returned immediately.

**Builder is to creation as Strategy is to algorithm**
- In the sense that it allows different implementations for steps

**Builder often builds a Composite.**
- overcoming the safety weakness of Composite due to the possible invocation of child-related operations on Leaf nodes

**Builder is often used in testing**
- to build a scaffold around the tested set of classes

**And in the construction of a memory structure from the parsing of a file**
- The parser act as Director
- Full exploitation with different implementations for the same component
Problem

- Assistito is responsible for the operation getStorico(), which retrieves the history of health services provided to some Cittadino.
- getStorico() must have different implementations depending on whether the history is retrieved for clinical purposes by the Cittadino or by his/her present/past Physician, or if it is retrieved for administrative purposes.
Basic solution: Assistito exposes different operations

- The responsibility of choosing the right implementation is left to the Client
- An intermediate proxy could add protection
- Yet, the choice is repeated by some object at each call

(Out of the expected rhetoric: what about overloading the operation name and passing the client self-reference to drive the binding?)
Localize selection of the specific implementation at creation
  - Possibly abstracted through delegation to some Factory

Leads to a stiff scheme,
  - E.g. not open to evolve for a refinement of types
  - Usual pros and cons of extension by subclassing
In principle the ConcreteStrategy can be changed dynamically
- E.g. based on some run-time performance metrics
- If not useful, might be a limit, adding space for unexpected mutability

Strategy can receive a reference to some Context
- or provide a setDelegate() operation allowing an external object to select
Strategy

- Changes behavior, by selecting one out of N implementations

Decorator

- Changes behavior through layered additions, with different possible combinations and orders
Java idioms for the Strategy pattern

- using anonymous nested classes
  specialized behavior can be determined even object by object
Un tipo nel modello di specifica ha diverse specializzazioni e diverse implementazioni

- Gli studenti della laurea triennale e della laurea di specializzazione hanno una diversa rappresentazione del curriculum e.g. un diverso calcolo della media
- Facoltà diverse usano diverse tecnologie di database e.g. diversa rappresentazione degli esami sostenuti

E’ un caso di nested generalization [Rum91]

- Una direzione attiene al tipo, l’altra all’implementazione
Ha le solite caratteristiche di uno schema di subclassing

- Un solo oggetto, facile da installare :-)
- Ma il numero delle classi esplode :-(
- E la scelta della classe è statica :-|
  (non un problema nello specifico esempio)
Studente astratto
- Implementa una narrow interface (protected) che incapsula la dipendenza dalla tecnologia del database (get_esami())
- Delega ad un archivio astratto

Archivio concreto
- Implementa l’archivio astratto su uno specifico database

Studente concreto
- Implementa le operazioni che dipendono dal tipo di studente
Ci sono 2 oggetti :-)  
L’associazione deve esser installata :-)  
- responsabilità a carico del costruttore dello studente concreto  
In principio l’associazione può essere modificata dinamicamente :-|  
- Ma non è l’intento, semmai è il caso di strategy  

Disaccoppia una astrazione dalla sua implementazione per lasciare che le due ammettano gerarchie di specializzazione indipendenti :-)}
Lo studente astratto rende disponibile ai suoi eredi il riferimento all’archivio
Lo studente concreto invoca direttamente l’archivio

Risparmia un forwarding (efficienza) :-)
Però accoppia lo studente concreto con l’archivio :-(

Code:
```java
get_archivio()
{
    return impl;
}

get_media()
{
    archivio=get_archivio();
    esami=archivio->get_esami;
    retu...
}
```
Lo schema della delega può servire comunque a disaccoppiare astrazione e implementazione
oppure a disaccoppiare parti dei requisiti con diversa stabilità
Ovviamente fino a che c'è una unica implementazione è inutile la implementazione astratta (!)
Bridge

vs. Adapter

- Il bridge divide un componente in due parti ortogonali, l’adapter adatta due componenti concettualmente equivalenti.
- Il bridge è a priori (up-front): separa l’implementazione dall’astrazione per favorirne una probabile evoluzione disaccoppiata.
- L’adapter a posteriori.

vs. Strategy

- La struttura di strategy è un subset di bridge.
- Ma l’intento è diverso.
rappresenta una operazione eseguita sugli elementi di una struttura di oggetti
separa una gerarchia di dati e una gerarchia di operazioni
object structure
- punta una collezione di elementi con implementazione concreta diversa

concrete element
- ciascun elemento accetta chiamate caratterizzate dall’indirizzo di un visitor
- sull’interfaccia esiste una unica funzione di accept
- implementa l’interfaccia astratta element

concrete visitor
- implementa una operazione nelle diverse versioni che trattano elementi di tipo concreto diverso
- implementa l’interfaccia astratta visitor
call back - pull mode
- sull’accept l’oggetto risponde chiamando il visitor concreto passato nella chiamata e passandogli il proprio indirizzo
- il visitor opera sull’oggetto
Double dispatch
- Il client del metodo `accept` seleziona il doppio contesto formato dall’elemento trattato e dal visitor che esegue il trattamento.

La selezione del metodo di visita è specificata nel nome invocato in `accept`.
- Possibile usare una selezione per binding dinamico.
  - I metodi nel visitor sono overloaded sullo stesso nome.
  - Ma sono distinti dal tipo del parametro.
  - Non cambia molto.
abilita l’evoluzione delle operazioni applicate agli elementi :-)  
  - una struttura dati stabile su cui non si prevedono a priori tutte le possibili elaborazioni (e.g. elaborazioni su un set di dati come nel caso Physis)  
  - è una chiusura strategica  

mantiene localizzate operazioni logicamente coesive che operano su tipi diversi :-)  
  - offre anche un contesto locale per l’accumulazione nel corso della visita

la classe visitor assomiglia un po’ a un functoide :-(
  - c’è una operazione ma non uno stato  
  - il contesto su cui lavorare è ricevuto nella chiamata  
  - il nome della classe (visitor) è quello di una funzione  
  - separa dati e funzioni  

e’ costoso aggiungere un nuovo tipo di elemento concreto :-(  
  - l’operazione è moltiplicata per il numero di visitors concreti  

element deve esporre una interfaccia pubblica per tutto quello che è usato dal visitor :-(
  - problema comune a schemi in modo pull
offre un accesso sequenziale agli elementi di un oggetto aggregato senza esporne la rappresentazione

oggetto aggregato = (multi)insieme di elementi omogenei
- sull’oggetto sono possibili attraversamenti diversi
- e.g. lista, albero, grafo, ...
- e.g. visita in pre- o post-ordine, visita breadth- o depth-first, filtraggio, ...

lo schema iterator sottrae all’aggregato la responsabilità della politica di attraversamento
- mantiene e manipola un cursore che punta un elemento corrente
- operazioni all’interfaccia: first, next, currentItem, isDone
il client vede l’aggregato attraverso una interfaccia astratta che nasconde la rappresentazione concreta
- e.g.: una lista può essere implementata con (quasi) la stessa interfaccia in forma sequenziale o collegata, con puntatori o indici, in ordine o meno, ...

l’implementazione concreta dell’iteratore dipende dalla rappresentazione concreta dell’aggregato
- la creazione dell’iteratore concreto è a carico della rappresentazione concreta dell’aggregato (operazione `CreateIterator()` )
Iterator
- definisce una interfaccia astratta capace di tenere traccia di un elemento corrente e determinare un elemento successivo
- diverse forme concrete implementano diverse politiche di attraversamento

Aggregate
- interfaccia astratta sull’aggregato (addElement(), CountElements(), …)
- include anche un metodo CreateIterator()

ConcreteAggregate
- implementa la rappresentazione concreta di Aggregate
- crea l’iterator concreto e ci installa il proprio indirizzo
Client

- delega ad aggregate la creazione di un iterator che poi usa per accedere gli elementi di aggregate stesso
- può controllare direttamente la visita operando su iterator (external iterator) oppure può delegare a iterator l’intera visita (internal iterator)
  - trade-off tra flessibilità e semplicità di uso
Iterator permette di variare la politica di attraversamento di un aggregato
- e.g. visita in pre-ordine o post-ordine, filtraggio sugli elementi visitati, …
- per farlo è sufficiente cambiare la forma concreta istanziata dell’iteratore

presenta una interfaccia astratta sull’aggregato a cui possono corrispondere più rappresentazioni

semplifica l’interfaccia nell’aggregato
- sottrae l’interfaccia delegata all’iterator

permette di avere più visite contemporaneamente pendentì
- ciascun iteratore tiene memoria dello stato della sua visita

ha un forte accoppiamento tra iteratore concreto e aggregato concreto
- l’aggregato concreto deve rendere pubblica l’interfaccia usata dall’iteratore
java.util.Iterator interface
- public boolean hasNext()
- public <E> next()
- public void remove()
Provides a unified interface for interfaces of a set of classes

Decoupling, Simplicity, Protection

But, exposes something that shall then be preserved,
- the facade itself may absorb the evoluzione.
- Better through a combined use of interfaces and abstract classes (reduces the impact on the side of classes used by the facade, not on the side of clients of the facade)
La facciata può essere configurata all’installazione

In principio la configurazione può essere variata dinamicamente
La specializzazione della facade può essere realizzata per ereditarietà

La scelta è statica

Non c’è complessità di installazione
Facade vs mediator
- La facciata sta sopra il mediator in mezzo
- Il mediator assume responsabilità che non sono di singoli oggetti ma di insiemi di oggetti e.g. consistenza
- Gli oggetti associati al mediator ne sono aware, gli oggetti associati alla facciata no

Facade and abstract factory
- Astrazione e consistenza nella configurazione della facciata

Facade and singleton
- Unicità e visibilità della facciata
Mediator (behavioral)

- Definisce un oggetto che incapsula il modo con cui interagiscono un insieme di oggetti
  - disaccoppia gli oggetti evitando che essi si referenzino direttamente
- i colleghi sono di tipo concreto diverso

- tutti i colleghi vedono solo il mediator
  - sono alleviati dal conoscersi a vicenda
  - stanno sotto una classe comune da cui ereditano il riferimento al mediator

- Il mediator vede la diversità dei colleghi
  - assume la responsabilità di formare la configurazione
  - i colleghi stanno sotto la stessa classe ma hanno operazioni diverse
  - il mediator astratto può essere omesso se i colleghi partecipano ad un unico gruppo

- Il mediator opera come front-end (interfaccia) verso i clienti
Mediator - conseguenze

- risponde alla complessità che nasce con l’uso diffuso della composizione e del suo possibile adattamento dinamico
  - crea astrazione sul modo con cui oggetti sono composti e interagiscono
  - per cambiare la composizione è sufficiente subclassare il mediator riusando gli stessi colleghi

- Il mediator centralizza il controllo dell’interazione
  - scompone relazioni n->n in relazioni n-a-1 e 1-a-n :-)
  - questo può rendere il mediator molto complesso e monolitico :-(

- Possibile organizzare il controllo secondo lo schema observer
  - il mediator opera come observer e i colleghi come subjects
  - quando un collega cambia stato lo notifica al mediator che poi propaga l’interazione
  - il notify del subject/colleague non ha iterazione perché esiste un unico observer/mediator
  - l’update sull’observer/mediator chiama indietro piu’ subjects concreti secondo lo schema di controllo del gruppo
    - schema observer in modo push o pull?
Per agevolare l'intervento sulle singole parti, i patterns tendono a rendere complesso l'impianto di insieme
- Aumentano il rischio di over-design

- riducono l'accoppiamento tra classi e oggetti
- congelano nel modello statico e nelle interfacce di un sistema meccanismi di creazione, strutturazione e comportamento degli oggetti che realizzano requisiti del sistema stesso indipendentemente dai dettagli della programmazione
- creano un linguaggio di discussione che eleva il livello di astrazione e sintesi nella descrizione di una struttura del codice
- aumenta il peso della progettazione rispetto alla implementazione


http://sourcemaking.com/design_patterns