Selected Design Patterns

Stéphane Ducasse
Stephane.Ducasse@univ-savoie.fr
http://www.iam.unibe.ch/~ducasse/

Goal
• What are patterns?
• Why?
• Patterns are not god on earth
• Example

Design Patterns
• Design patterns are recurrent solutions to design problems
• They are names
  • Composite, Visitor, Observer...
  • They are pros and cons

From Architecture
Christoffer Alexander

More advanced than what is used in computer science
only the simple parts got used.
... pattern languages were skipped.

Why Patterns?
Smart
Elegant solutions that a novice would not think of
Generic
Independent on specific system type, language
Well-proven
Successfully tested in several systems
Simple
Combine them for more complex solutions

There are really stupid patterns (supersuper) in some books so watch out!!!

Patterns provide...
Reusable solutions to common problems
based on experiences from real systems
Names of abstractions above class and object level
a common vocabulary for developers
Handling of functional and non-functional aspects
separating interfaces/implementation, loose coupling
between parts,...
A basis for frameworks and toolkits
basic constructs to improve reuse
Education and training support

Elements in a Pattern
Pattern name
Increase of design vocabulary
Problem description
When to apply it, in what context to use it
Solution description (generic !)
The elements that make up the design, their relationships,
responsibilities, and collaborations
Consequences
Results and trade-offs of applying the pattern

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

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

Use the Composite Pattern when:
- you want to represent part-whole hierarchies of objects
- you want clients to be able to ignore the difference between compositions of objects and individual objects.

Clients will treat all objects in the composite structure uniformly.

**Composite Pattern Participants**

**Component (Graphic)**
- declares the interface for objects in the composition
- implements default behavior for the interface common to all classes, as appropriate
- declares an interface for accessing and managing its child components

**Leaf (Rectangle, Line, Text, ...)**
- represents leaf objects in the composition. A leaf has no children
- defines behavior for primitive objects in the composition

**Composite (Picture)**
- defines behavior for components having children
- stores child components
- implements child-related operations in the Component interface

**Client**
- manipulates objects in the composition through the Component interface

**Composite Pattern Collaborations**

Clients use the Component class interface to interact with objects in the composite structure.

Leaves handle requests directly.

Composites forward requests to its child components

**Consequences**
- defines class hierarchies consisting of primitive and composite objects.
- Makes the client simple. Composite and primitive objects are treated uniformly (no cases)
- Eases the creation of new kinds of components
- Can make your design overly general

**An Alternate Structure**

Again structure is not intent!

**Queries...**

- To be able to specify different queries over a repository
  - $q_1 := \text{PropertyQuery\ property:} \#HNL \text{ with:} \#K \text{ value:} 4.$
  - $q_2 := \text{PropertyQuery\ property:} \#NOM \text{ with:} \#P \text{ value:} 10.$
  - $q_3 := \text{PatternName match:} \text{"figure"}.$
- Compose these queries and treat composite queries as one query
  - $(e_1 e_2 e_3 \ldots e_n)(q_1 \text{ and } q_2 \text{ and } q_4) \text{ or } q_3 \Rightarrow (e_2 e_5)
  - composer := AndCompositeQuery with: (Array with: q1 with: q2 with: q4)

**A Possible Solution**
In Smalltalk

- Composite not only groups leaves but can also contain composites
- In Smalltalk add:, remove: do not need to be declared into Component but only on Composite. This way we avoid to have to define dummy behavior for Leaf

Composite Variations

- Use a Component superclass to define the interface and factor code there.
- Consider implementing abstract Composite and Leaf (in case of complex hierarchy)
- Only Composite delegates to children
- Composites can be nested
- Composite sets the parent back-pointer (add/remove)

Composite Variations

- Can Composite contain any type of child? (domain issues)
- Is the Composite’s number of children limited?
- Forward
  - Simple forward. Send the message to all the children and merge the results without performing any other behavior
  - Selective forward. Conditionally forward to some children
  - Extended forward. Extra behavior
  - Override. Instead of delegating

Other Patterns

- Composite and Visitors
  - Visitors walks on structured objects
- Composite and Factories
  - Factories can create composite elements

Patterns...

Categories of Design Patterns

Creational Patterns
  - Instantiation and configuration of classes and objects
Structural Patterns
  - Usage of classes and objects in larger structures, separation of interfaces and implementation
Behavioral Patterns
  - Algorithms and division of responsibility
  - Concurrency
  - Distribution
  - Security

Some Creational Patterns

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

Some Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Some Behavioral Patterns

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor
Alert!!! Patterns are invading
- Design Patterns may be a real plague!
- Do not apply them when you do not need them
- Design Patterns make the software more complex
  - More classes
  - More indirections, more messages
- Try to understand when NOT applying them!

About Pattern Implementation
This is POSSIBLE implementation not a definitive one

Do not confuse structure and intent!!!
Patterns are about INTENT and TRADEOFFS

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

The Singleton Pattern
- Intent: Ensure that a class has only one instance, and provide a global point of access to it
- Problem: We want a class with a unique instance.
- Solution: We specialize the #new class method so that if one instance already exists this will be the only one. When the first instance is created, we store and return it as result of #new.

Singleton Possible Structure

Implementation Issues
- Singletons may be accessed via a global variable (ex: NotificationManager uniqueInstance notifier).
  SessionModel>>startupWindowSystem
  "Private: Perform OS window system startup"
  Notifier initializeWindowHandles.
  oldWindows := Notifier windows.
  Notifier install.
  "oldWindows = Notifier windows."

- Persistent Singleton: only one instance exists and its identity does not change (ex: NotifierManager in Visual Smalltalk)

- Transient Singleton: only one instance exists at any time, but that instance changes (ex: SessionModel in Visual Smalltalk, SourceFileManager, Screen in VisualWorks)

- Single Active Instance Singleton: a single instance is active at any point in time, but other dormant instances may also exist. Project in VisualWorks
Implementation Issues

classVariable or class instance variable

classVariable
One singleton for a complete hierarchy

Class instance variable
One singleton per class

Access?
In Smalltalk we cannot prevent a client to send a message (protected in C++). To prevent additional creation we can redefine new/new:

Object subclass: #Singleton

instanceVariableNames: 'uniqueInstance'
classVariableNames: ''
poolDictionaries: ''

Singleton class>>new

self error: 'Class ', self name, ' cannot create new instances'

Favor Instance Behavior

When a class should only have one instance, it could be tempting to define all its behavior at the class level. But this is not good:

Class behavior represents behavior of classes: “Ordinary objects are used to model the real world. MetaObjects describe these ordinary objects”

Do not mess up this separation and do not mix domain objects with metaconcerns.

What’s happens if later on an object can have multiple instances? You have to change a lot of client code!

Time and not Scope

Singleton is about time not access

time: only one instance is available at the same time
access: can’t you add an instance to refer to the object?

Singleton for access are as bad as global variables

Often we can avoid singleton by passing/refering to the object instead of favoring a global access point

It is worth to have one extra instance variable that refers to the right object

Visitor Intent

Intent: Represent an operation to be performed on the elements of an object structure in a class separate from the elements themselves. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor Possible Structure

Whenever you have a number of items on which you have to perform a number of actions, and

When you ‘decouple’ the actions from the items.

Examples:
the parse tree (ProgramNode) uses a visitor for the compilation (emitting code on CodeStream)
GraphicsContext is a visitor for VisualComponents, Geometrics, and some other ones (CharacterArray, ...)
Rendering documents

Visitor

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

Access using new: not good idea

Singleton class>>new

^self uniqueInstance

The intent (uniqueness) is not clear anymore! New is normally used to return newly created instances. The programmer does not expect this:

|screen1| screen2|

screen1 := Screen new.
screen2 := Screen uniqueInstance

When to use a Visitor

Whenever you have a number of items on which you have to perform a number of actions, and

When you ‘decouple’ the actions from the items.

Examples:
the parse tree (ProgramNode) uses a visitor for the compilation (emitting code on CodeStream)
GraphicsContext is a visitor for VisualComponents, Geometrics, and some other ones (CharacterArray, ...)
Rendering documents
Applying the Visitor

So all our problems are solved, no?
Well...
when to use a visitor
control over item traversal
choosing the granularity of visitor methods
implementation tricks

When to Use a Visitor

Use a Visitor:
when the operations on items change a lot.
Do not use a visitor:
when the items you want to visit change a lot.
Question: But how do we know what to choose up-front?

Visitor Toy Example

Language to deal with arithmetic expressions.
It supports one kind of number, and has +, *, (, )
We want to evaluate expressions, and print them.
Example:
1 + 1
result: 1 + 1 = 2
((4 * 2) * (3 + 5)) * 3
result: (4 * 2 * (3 + 5)) * 3 = 192
...

Visitor Toy Example: ParseTree

Visitor Toy Example Solution 1

Two solutions:
add methods for evaluating, printing,... on Expression and
its subclasses
create a Visitor, add the visit methods on Expression and
its subclasses, and implement visitors for evaluation,
printing,...

Implementing the Actions

Visitor Toy Example 2

So which solution to take?
In this case you might say:

 printing is not easy
adding it directly on Expression clusters Expression (need
to add instance variables etc.)
therefore we can factor out the printing on a separate
class.
if we do this with a visitor we can then implement
evaluation there as well.

Smalltalk’s class extensions

Smalltalk has class extensions:
method addition
method replacement
So ‘Decoupling’ actions from items can be done:
e.g., put all the printing methods together.
take care: works only for methods
makes it also really easy to package a visitor!
Note: this is a static solution!
Controlling the traversal

Somewhere in the visitor, items are traversed. Different places where the traversal can be implemented:
- in the visitor
- on the items hierarchy

Granularity of Visit Methods

Sometimes you can pass context information with the visit methods. So visitors have more information for implementing their operations.

Implementation Tricks

You can implement it as we have shown before. But notice the general structure of the methods! This can be taken as advantage:
- code can be generated for a visitor.
- the method can be performed/invoked
But take care:
- only works when there is a full correspondence.
- can make the code hard to understand.

Using #perform:

Define a family of algorithms, encapsulate each in a separate class and define each class with the same interface so that they can be interchangeable.

Also known as Policy.
Strategy Intent
- Define a family of algorithms, encapsulate each in a separate class and define each class with the same interface so that they can be interchangeable.

Motivation
Many algorithms exist for breaking a stream into lines. Hardwiring them into the classes that requires them has the following problems:

- Clients get more complex
- Different algorithms can be used at different times
- Difficult to add new algorithms at run-time

Code Smells
Composition>>repair

ifTrue: [ self formatWithSimpleAlgorithm]
ifFalse: [ formatting == #Tex
            ifTrue: [self formatWithTex]
            ....]

Alternative
Composition>>repair

selector := ('formatWith, formatting) asSymbol.
self perform: selector

Still your class gets complex...

Inheritance?
May not be the solution since:
- you have to create objects of the right class
- it is difficult to change the policy at run-time
- you can get an explosion of classes bloated with the use of a functionality and the functionalities.
- no clear identification of responsibility

Strategy Solution
When
Many related classes differ only in their behavior
You have variants of an algorithm (space/time)
An algorithm uses data that the clients does not have to know

Structure
Composition>>repair

formatter format: self

Participants
Strategy (Compositor)
declares an interface common to all concrete strategies

Concrete Strategies
implement algorithm

Context
configure with concrete strategy
maintains a reference to the concrete strategy
may define an interface to let the strategy access data
Collaborations (i)
Strategy and Context interact to implement the chosen algorithm.
A context may pass all data required by the algorithm to the strategy when the algorithm is called.

GraphVisualizer>>graphIt
grapher plot: data using: graphPane pen
Grapher>>plot: data using: aPen

Context passes itself as argument
Also know as self-delegation...
GraphVisualizer>>graphIt
grapher plotFor: self
BartChartGrapher>>plotFor: aGraphVisualizer
|data|
  data := aGraphVisualizer data
  ...

BackPointer
Grapher class>>for: aGraphVisualizer
  ^ self new graphVisualizer: aGraphVisualizer
BartChartGrapher>>plot
  graphVisualizer data:
  graphVisualizer pen

Grapher (Strategy) points directly to GraphVisualizer (Context), so sharing strategy between different context may be difficult, if sharing is needed then use self-delegation

Collaboration (ii)
“A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively.” GOF
Not sure that the client has to choose...

Consequences
Define a family of pluggable algorithms
Eliminates conditional statements
Clients can choose between several implementations
Clients must be aware of the different strategies
Increase the number of objects
Communication overhead between client and strategies
Weaken encapsulation of the client

Domain-Specific Objects as Strategies
Strategies do not have to be limited to one single algorithm
They may represent domain specific knowledge
Mortgage
  FixedRateMortgage
  OneYear...

Known Uses
ImageRenderer in VW: “a technique to render an image using a limited palette”
ImageRenderer
  NearestPaint
  OrderedDither
  ErrorDiffusion

View-Controller
  a view instance uses a controller object to handle and respond to user input via mouse or keyboard.
  Controllers can be changed at run-time

Abstract Factory
Provide an interface for creating families of related or dependent objects without specifying their concrete classes
Also known as: Kit

Abstract Factory Intent
• Provide an interface for creating families of related or dependent objects without specifying their concrete classes
Abstract Factory Motivation

You have an application with different looks and feels. How to avoid to hardcode all the specific widget classes into the code so that you can change from Motifs to MacOsX?

Abstract Factory Applicability

a system should be independent of how its products are created, composed, and represented
a system should be configured with one of multiple families of products
a family of related product objects is designed to be used together, and you need to enforce this constraint you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Abstract Factory Structure

Abstract Factory Motivation

Abstract factory introduce an interface for creating each basic kind of widget

Abstract Factory Participants

AbstractFactory (WidgetFactory) declares an interface for operations that create abstract product objects
ConcreteFactory (MotifWidgetFactory, PMWidgetFactory) implements the operations to create concrete product objects

Abstract Product (Window, ScrollBar) defines a product object to be created by the corresponding concrete factory
Client uses only interfaces declared by AbstractFactory and AbstractProduct classes

Implementation: Specifying the Factory

MazeGame class>>createMazeFactory
^ (MazeFactory new
  addPart:Wall named: #wall;
  addPart:Room named: #room;
  addPart:Door named: #door;
  yourself)EnchantedMazeGame class>>createMazeFactory
^ (MazeFactory new
  addPart:Wall named: #wall;
  addPart:Room named: #room;
  addPart:Door named: #door;
  yourself)Implementation: Using the Factory

MazeFactory>>createMaze: aFactory
| room1 room2 aDoor |
  room1 := (aFactory make: #room) number: 1.
  room2 := (aFactory make: #room) number: 2.
  aDoor := (Factory make: #Door) from: room1 to: room2.
  room1 atSide: #north put: (Factory make: #wall).
  room1 atSide: #east put: aDoor.
  room2 atSide: #south put: (Factory make: #wall).
  room2 atSide: #west put: aDoor.
^ Maze new addRoom: room1; addRoom: room2; yourself
MazeFactory>>make: partName
^ (partCatalog at: partName) new

Implementation: Using the Factory

MazeFactory>>createMaze: aFactory
| room1 room2 aDoor |
  room1 := (aFactory make: #room) number: 1.
  room2 := (aFactory make: #room) number: 2.
  aDoor := (Factory make: #Door) from: room1 to: room2.
  room1 atSide: #north put: (Factory make: #wall).
  room1 atSide: #east put: aDoor.
  room2 atSide: #south put: (Factory make: #wall).
  room2 atSide: #west put: aDoor.
^ Maze new addRoom: room1; addRoom: room2; yourself
MazeFactory>>make: partName
^ (partCatalog at: partName) new

Implementation: Using the Factory

MazeFactory>>createMaze: aFactory
| room1 room2 aDoor |
  room1 := (aFactory make: #room) number: 1.
  room2 := (aFactory make: #room) number: 2.
  aDoor := (Factory make: #Door) from: room1 to: room2.
  room1 atSide: #north put: (Factory make: #wall).
  room1 atSide: #east put: aDoor.
  room2 atSide: #south put: (Factory make: #wall).
  room2 atSide: #west put: aDoor.
^ Maze new addRoom: room1; addRoom: room2; yourself
MazeFactory>>make: partName
^ (partCatalog at: partName) new

Implementation: Using the Factory

MazeFactory>>createMaze: aFactory
| room1 room2 aDoor |
  room1 := (aFactory make: #room) number: 1.
  room2 := (aFactory make: #room) number: 2.
  aDoor := (Factory make: #Door) from: room1 to: room2.
  room1 atSide: #north put: (Factory make: #wall).
  room1 atSide: #east put: aDoor.
  room2 atSide: #south put: (Factory make: #wall).
  room2 atSide: #west put: aDoor.
^ Maze new addRoom: room1; addRoom: room2; yourself
MazeFactory>>make: partName
^ (partCatalog at: partName) new
Abstract Factory Collaborations

Collaborations
Normally a single instance of ConcreteFactory is created at run-time
AbstractFactory defers creation of product objects to its
ConcreteFactory subclass

Consequences
It isolates concrete classes
It makes exchanging product families easy
It promotes consistency among products
Supporting new kinds of products is difficult (set of
products is somehow fixed)
The class factory "controls" what is created

Using Prototypes
The concrete factory stores the prototypes to be cloned in a
dictionary called partCatalog.

make: partName
^ (partCatalog at: partName) copy

The concrete factory has a method for adding parts to the
catalog:
addPart: partTemplate named: partName
partCatalog at: partName put: partTemplate

Prototypes are added to the factory by identifying them with a
symbol:

In Relations
Builder and Abstract Factory are closely related
But Builder is in charge of assembling parts
AbstractFactory is responsible of producing parts that
work together

Known Uses
VisualWorks UILookPolicy

Chain of Responsibility
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.

Chain the receiving objects and pass the request along the chain until an object handles it.

Motivation
The problem here is that the object that ultimately provides the help isn't known explicitly to the object
(e.g., the button) that initiates the help request.

How to decouple senders and receivers?
By giving multiple objects a chance to handle a request. The request gets passed along a chain of objects until
one of them handles it.

Chain of Resp. Possible Structure

Client
Sender
Recipient
Recipient
Recipient
Recipient
Participants

Handler
  defines an interface for handling requests
  may implement the successor link
ConcreteHandler
  handles requests it is responsible for
    can access its successor
Client
  initiates the request to a concreteHandler

Dynamic

The first object in the chain receives the request and
  either handles it or forwards it to the next candidate
  on the chain, which does likewise.
  The object that made the request has no explicit
  knowledge of who will handle it

Chain

Can be a linked list
  But also a tree (cf. Composite Pattern)

Usually order can represent
  specific to more general
  priority: more important (security... in SmallWiki)

Consequences (i)

Reduced coupling. The pattern frees an object from
  knowing which other object handles a request.
  An object only has to know that a request will be handled
    "appropriately."
  Both the receiver and the sender have no explicit
  knowledge of each other, and an object in the chain
  doesn’t have to know about the chain’s structure.

Simplify object interconnections. Instead of objects
  maintaining references to all candidate receivers, they
  keep a single reference to their successor

Consequences (II)

Added flexibility in assigning responsibilities to
  objects.
  flexibility in distributing responsibilities among objects.
  can add or change responsibilities for handling a request
  by adding to or otherwise changing the chain at run-time.

Receipt isn’t guaranteed.
  no guarantee it’ll be handled; the request can fall off the
  end of the chain without ever being handled.
  A request can also go unhandled when the chain is not
  configured properly.

Differences with Decorator

A Decorator usually wraps the decorated object: clients
  point to the decorator and not the object

A Decorator does not have to forward the same
  message

A decorated object does not have to know that it is
  wrapped

With a chain of responsibility, the client asks the first
  chain objects explicitly.

Variations

Do the work or pass? or both?
  the DP says that the handler either does the work or
  passes it to its successor but it can also do part of the
  job (see OO recursion)

OO Recursion: Hash, = and copy

Person>> = aPerson
  ^ self name = aPerson name

PersonName>> = aPersonName
  ^ (self firstName = aPersonName firstName)
  and: [(self lastName = aPersonName lastName)]

String>> = aString
  ...

OO Recursion: Hash, = and copy

Person>>= hash
  ^ self name hash

PersonName>>= hash
  ^ self firstName hash bitXor: self lastName hash
**OO Recursion**

With Chain of Responsibility you may recur from leave to root, from more specific to more general.
Default in root, specific and recursion in leaves

With OO recursion, from composite (person) to components (leaves)
Default in leaf (String =)

**Smalltalk Specific**

Automatic Forwarding with doesNotUnderstand:
can work
but can be dangerous

**Wrap-up**

Patterns are names
Patterns are about tradeoffs
Know when not to apply them