Design Patterns • VIII

Non-GoF Design Patters, Anti-Patterns

COMP2110/2510
Software Design
Software Design for SE
September 26, 2008

Alexei Khorev
Department of Computer Science
The Australian National University

GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
- Pluggable Selector
- Active Object
- Null Object
- Generation Gap

MVC
JavaEE Patterns
Anti-Patterns
- Singletonitis Anti-Pattern

Good Pattern Practices
Patterns and OO Design

1. **GoF Catalog**
2. **Patterns Variability**
3. **Selecting and Using DPs**
4. **Non-GoF Patterns**
   - Pluggable Selector
   - Active Object
   - Null Object
   - Generation Gap
5. **Model–View–Controller**
6. **JavaEE Framework Patterns**
7. **Anti-Patterns**
   - Singletonitis Anti-Pattern
8. **Good Pattern Practices**
9. **Patterns and OO Design**

GoF Catalog

Patterns Variability

Selecting DPs

Non-GoF Patterns

Pluggable Selector

Active Object

Null Object

Generation Gap

MVC

JavaEE Patterns

Anti-Patterns

Singletonitis Anti-Pattern

Good Pattern Practices

Patterns and OO Design

Alexei Khorev

The GoF Gang

The Gang of Four in their heyday. Ralph, Erich, Richard, and John at OOPSLA 1994.

GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design

Catalog of Patterns: Three categories

The design patterns marked by red may be featured in the final exam paper. We have also discussed (albeit cursorily) the patterns marked by cyan, but they are unlikely to make into the final exam. If any pattern which we have not discussed (marked by black) are included in the final exam, you will have full right to stand up in the examination hall and vociferate **"Not fair!"**

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

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

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento

- Observer
- State
- Strategy
- Template Method
- Visitor

GoF Catalog

Patterns Variability

Selecting DPs

Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap

MVC

JavaEE Patterns

Anti-Patterns
Singletonitis Anti-Pattern

Good Pattern Practices

Patterns and OO Design

Catalog of Patterns: Variability of DPs

Design patterns form the backbone of flexible framework architecture, encapsulating the areas of variability inside classes or components. This encapsulation allows interfaces and polymorphism to work their magic, providing plug-and-play hot swapping of components that expose the same interface, while varying their individual implementation. Evolving Frameworks provides a table that relates *variability areas* with the GoF Design Patterns.

What varies	Design Pattern
Algorithms	Strategy
Operation applied to object	Visitor
Actions	Command
Implementation	Bridge
Response to Change	Observer
Interaction between objects	Mediator
Subclass of object being created	Factory Method
Families of product objects	Abstract Factory
Object being instantiated	Prototype
Structure being created	Builder
Traversal algorithm	Iterator
Object interfaces	Adapter
Object behaviour	State, Decorator

GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design

How to select and use patterns

How to Select a DP

- Consider how design patterns solve design problems — to find objects, determine object granularity, specify object interfaces etc
- Scan Intent sections — which sounds relevant to your problem; group (creational, structural and behavioural) and scope (class, object) classification are instructive
- Study how patterns interrelate — the GoF famous “Pattern Map” can help direct you to the right pattern or group of patterns
- Study patterns of like purpose — think through similarities and differences between patterns of like purpose
- Examine a cause of redesign — patterns should help you to avoid it
- Consider what should be variable in your design — what you want to be *able to change without redesign*; the table on the previous slide gives (incomplete) description of how patterns are characterized from the variability point of view

How to Use a DP

- Read the pattern once through for an overview — pay particular attention to the Applicability and Consequences
- Go back and study the Structure, Participants, and Collaborations sections
- Look at the Sample Code section to see a concrete example of the pattern in code
- Choose names for pattern participants that are meaningful in the application context
- Define the classes — declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references; identify existing classes that the pattern will affect, and modify them accordingly
- Define application-specific names for operations in the pattern — use the responsibilities and collaborations associated with each operation as a guide; be consistent in your naming conventions
- Implement the operations to carry out the Responsibilities and Collaborations

GoF Catalog

Patterns Variability

Selecting DPs

Non-GoF Patterns

Pluggable Selector

Active Object

Null Object

Generation Gap

MVC

JavaEE Patterns

Anti-Patterns

Singletonitis Anti-Pattern

Good Pattern Practices

Patterns and OO Design

Non-GoF Patterns

- **Pluggable Selector**
  has no standard GoF structure — no diagram, “no nothing”;
  try add them yourself as an exercise ☺
- **Active Object**
  an extension of the Command Design Pattern in an
  application with multiple threads of control
- **Null Object**

  can be regarded as (any) OO language idiom, but very cute
- **Generation Gap**
  A decent DP in its own right, which did not make
  into the *Catalogue* because of paucity of uses
  The brainchild of John Vlissides
- **Model–View–Controller**
- **J2EE Patterns**
  mostly hybrids of Design and Architectural Patterns
  specially adopted for J2EE architectural framework

GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design

GoF Catalog

Patterns Variability

Selecting DPs

Non-GoF Patterns

Pluggable Selector

Active Object

Null Object

Generation Gap

MVC

JavaEE Patterns

Anti-Patterns

Singletonitis Anti-Pattern

Good Pattern Practices

Patterns and OO Design

Pluggable Selector Design Pattern

Intent: To use a single class which can be parameterized to perform different logic without requiring subclassing.

How can you parameterize the behavior of an object?

- Conventional view: objects have different state and the same behaviour
- Normally classes communicate their behaviour (even without running)
- But classes are expensive: A large family of classes with only a single method each is unlikely to be valuable
- Can the class methods be treated as fields (data) and be reset?

PLUGGABLE SELECTOR allows to specify different behaviour (logic) in different instances of the same class. In *Smalltalk*, this pattern relies on the language feature called *method selector* (a particular kind of *message selector* — a mechanism of identifying the code which the object will execute as a result of receiving the message). Java instead has to use *Reflection API* to invoke the method from a string representing the method's name. (An alternative is to use anonymous inner classes to choose the behaviour when creating an object. Both methods are employed in the JUnit framework to define test methods in the test classes which all extend the *TestCase* class.) This is how the `runTest()` method of the *TestCase* class is defined (`fName` is a *String* parameter passed to the *TestCase* constructor):

```
protected void runTest() throws Throwable {
   Method runMethod= null;
   try {
      runMethod= getClass().getMethod(fName, new Class[0]);
   } catch (NoSuchMethodException e) {...}
   try {
      runMethod.invoke(this, new Class[0]);
   } //catch InvocationTargetException and IllegalAccessException
}
```

GoF Catalog

Patterns Variability

Selecting DPs

Non-GoF Patterns

Pluggable Selector

Active Object

Null Object

Generation Gap

MVC

JavaEE Patterns

Anti-Patterns

Singletonitis Anti-Pattern

Good Pattern Practices

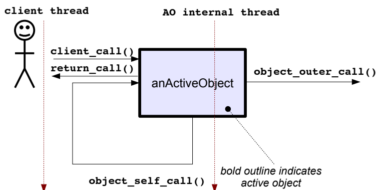
Patterns and OO Design

Active Object Design Pattern

Intent: To decouple method execution from method invocation to enhance concurrency and simplify synchronized access to an object that resides in its own thread of control.

Normally objects are passive entities in an OO program: they do something only when they receive a specific request in the form of a call to one of their methods sent from their client. Often, however, especially in concurrent (multithreaded) applications, objects which can carry out some task more or less autonomously allow significant simplification of the program logic. An *Active Object* is simply an object which controls its own thread, but its methods are just normal methods — they can be run in the object's own thread, or they can be run in the client's thread.

One often occurring problem in a multi-threaded program is how to avoid blocking a thread when it waits for an event. Instead, the thread execution is implemented as a COMMAND *active* (non-blocking) object which check for the expected event occurrence, and puts itself back into the event queue to be executed again at a later stage.



Such approach to threads design is known as *Run-To-Completion* (RTC) tasks because each *Command* instance runs to completion *before* the next *Command* instance can run. The *Command* instances do not block, which allows to reduce demand for execution environment resources since RTC threads can share the same *run-time stack* (important for real-time and embedded multithreaded systems).

Null Object Design Pattern

Intent: Provide a surrogate for another object that shares the same interface but does nothing. The NULL OBJECT encapsulates the implementation decisions of how to "do nothing" and hides those details from its collaborators.

Sometimes a class that requires a collaborator does not need the collaborator to do anything. Yet, the class wishes to treat a collaborator that does nothing the same way it treats one that actually provides behavior.

A typical situation when a method (a DB query, say) is expected to return (a reference to) an object, on which then another method will be invoked. If the original method *may* return the `null` the code which invokes the object method should add check that the returned object is not `null`, or the object fetching method should throw an exception instead of returning `null`. Either way — the code becomes ugly and error prone, the design worsens *etc.*

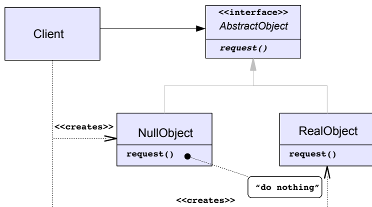
The solution — define class hierarchies consisting of real objects and *null objects*.

The use of the NULL OBJECT DP results in the code which is not ugly or convoluted, and design which does not deteriorates or becomes brittle.

```
Employee e = DB.getEmployee("Bob);
if (e.isTimeToPay(today)
   e.pay();
```

Quite often, the *Null Object* class is defined as *Singleton*. But despite the similarity, it is *not* a PROXY (intent is different).

The proper "GoF" NULL OBJECT pattern description can be found in the B. Woolf contribution "The Null Object Pattern" in *Pattern Languages of Program Design, 3*, eds. R. Martin *et al*, AW, 1998



GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design

Generation Gap

In Homage to John Vlissides

Intent: Modify or extend generated code once no matter how many times it is regenerated.

More and more code is now tool-generated (*eg*, by GUI builders in modern IDE's like Eclipse or Netbeans, parser generators and many others). The requirements for generated code include:

- to be correct — no problem
- to be efficient — can be achieved
- functionally complete — tools cannot provide full functionality, hand-coding is required (the tool's abstractions ≢ those of programming languages)
- maintainable — same reason

What happens when you hand modify the generated code to complete its functionality (like implementing event handlers etc), and *after* decide to change the UI again using the GUI builder tool? All your hand additions will be overwritten. The tool based solution — to mark the generated code against modification (*Netbeans* makes such code unmodifiable by the user) — is deficient: 1. it's messy; 2. if changes are made, the compiler cannot check it.

John Vlissides died in 2005
from brain tumor at the age of 44

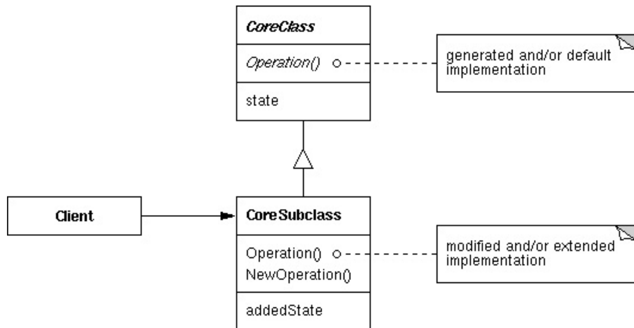
The generated and hand-added code must be separated, but this is hard to achieve if you need to modify parts which are not public. The Generation Gap pattern offers a solution to this problem through class inheritance. The generated code is encapsulated into a class which then is split in two — one is for encapsulating generated code, and another for modified code.

GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design

Generation Gap (cont.)

Applicability: the Generation Gap pattern is useful when all the following is true

- Code is generated automatically
- Generated code can be encapsulated in one or more classes
- Regenerated code usually retains the inteface and instance variables of the previous generation
- Generated classes usually aren't integrated into existing class hierarchies



GoF Catalog

Patterns Variability

Selecting DPs

Non-GoF Patterns

Pluggable Selector

Active Object

Null Object

Generation Gap

MVC

JavaEE Patterns

Anti-Patterns

Singletonitis Anti-Pattern

Good Pattern Practices

Patterns and OO Design

Generation Gap (concl.)

Participants

- **CoreClass** —
  - an abstract class containing a tool-generated implementation
  - is never modified by hand
  - is overwritten by the tool on regeneration
- **CoreSubclass** —
  - a trivial subclass of *CoreClass*
  - implements extensions of or modifications to *CoreClass*. A programmer may change it to add state and/or extend, modify, or override *CoreClass* behavior
  - extensions or modifications are preserved across re-generations
- **Client** — instantiates and refers to CoreSubclass only

Collaborators

- *CoreSubclass* inherits tool-generated behavior from *CoreClass*, overriding or extending its behavior
- *CoreClass* exposes and/or delegates select functionality to *CoreClass* to allow modification or extension of its behavior

Consequences

(+) Modifications are decoupled from generated code. As a bonus, the *CoreSubclass* interface may give insight into the modification by indicating the operations that were overridden or added

(+) Modifications can have privileged access to implementation details

(+) Subsequent regeneration does not require reapplying the modifications

(–) Double the number of classes

(–) Integrating generated classes into existing class hierarchies may be difficult

A case study: explore what design (patterns, if any) is used in GUI builder module of *Netbeans*, the IDE which achieves the separation of generated and hand-added code.

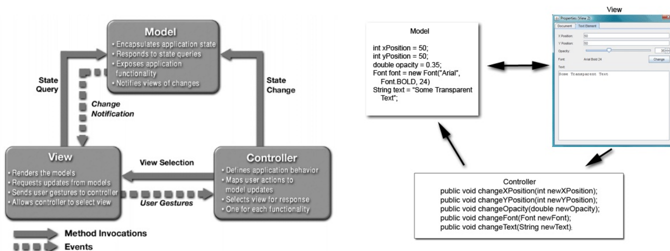
GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design

MVC (Design) Pattern

The MVC is well known to you. It is both an architectural pattern and a design pattern, depending on where it's applied. But as a design pattern, the MVC has a greater architectural underpinnings than a usual DP. The components (aka participants):

- *Model* The domain-specific representation of the information on which the application operates (like time data and time operations in the *Clock* application).
- *View* Renders the model into a form suitable for interaction, typically a user interface element. Multiple views can exist for a single model for different purposes.
- *Controller* Processes and responds to events, typically user actions, and may invoke changes on the model.

Different implementation of MVC architecture-pattern include varying degrees of (de-)coupling between the parts, and the way they interact which each other (*eg*, *Controller* can be an intermediate agent between *View* and *Model*, or the two can interact more directly).



The MVC components are not always separated. Java's *Swing* GUI framework is an example of *separable model architecture*: the *Controller* and the *View* of *Swing*'s components are joined into a single object — *UI delegate*, which among other benefits (greatly simplified logic code) allows run-time selection of UI object responsible for (pluggable) look-and-feel.

GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design

JavaEE Framework Patterns

Java (Platform) Enterprise Edition (JavaEE) is a platform for server programming in Java. In addition to standard Java distribution (JavaSE), it contains libraries and frameworks for deploying fault-tolerant (“enterprise!”), distributed, multi-tier applications, which rely on modular components (“beans”) running on Sun Java System Application Server Platform.

JavaEE consists of several frameworks built to *JavaEE specifications*. The Enterprise patterns are organised around three tier architecture:

- Presentation Tier — Intercepting Filter, Front Controller, View Helper, Composite View, Service to Worker
- Business Tier — Business Delegate, Service Locator, Session Façade, Value Object, Value Object Assembler, Composite Entity, Value List Handler
- Integration Tier — Connector, Data Access Object, Service Activator

The detailed discussion of the JavaEE architectural layers (tiers) and component based technologies (JavaBeans) is not possible here (learning JavaEE is a longer and more specialized process then learning Java programming and design). If you pursue these studies in future, the following links may be useful:

- Core J2EE Patterns — contains links to all 15 EE patterns, each given as detailed exposition as for standard GoF patterns
- The whole presentation is based on the book

  Core J2EE Patterns: Best Practices and Design Strategies

  by Deepak Alur, John Crupi and Dan Malks

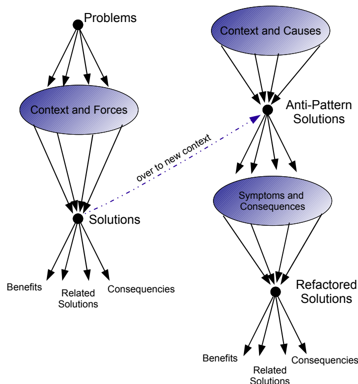
GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design

Anti-Pattrens

The term *Anti-Patterns* was introduced by Andrew Koenig in 1995 to define *bad practices*, some kind of antipodes to the DPs. Two key elements distinguish an anti-pattern form a bad habit, bad practice or simply “bad idea”:

- Some repeated pattern of action, process or structure that initially appears to be beneficial, but ultimately produces more bad consequences than beneficial results, **and**
- A refactored solution that is clearly documented, proven in actual practice and repeatable

(from *AntiPatterns* book by W. H. Brown *etal*).



Also called *pitfalls* or *dark patterns*, the Anti-Patterns are “classes of commonly reinvented bad solutions to common (recurring) problems”.

Patterns and AntiPatterns are related, but if a pattern is “a problem and its solution in a context”, an anti-pattern is two solutions: “the first solution is *problematic* — it is a commonly occurring solution that generates overwhelmingly negative consequences; the second solution is the *refactored solution*, in which the anti-pattern is resolved and transformed into a more beneficial form.

A pattern can evolve into an anti-pattern when used in an inappropriate context.

GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design

Types of Anti-Patterns

“Since the first step toward recovery is admitting that you have a problem, the anti-pattern problem helps readers to clarify the problem in dramatic terms. They can then assess the applicability of the problem’s symptoms and consequences to their own situation. Many people also find anti-patterns entertaining. To err is human. We all laugh at our mistakes and the mistakes of others when no insult is intended.” (*AntiPatterns*, W. H. Brown *etal*)

Anti-Patterns often have funny, catchy name (unlike Patterns, where technical rigour in pattern template description leaves little room for jokes).

Just like the Software Patterns, the Anti-Patterns occur in all aspects of software development. There are

- Organizational anti-patterns
- Analysis anti-patterns
- Software design anti-patterns (OO design anti-patterns is a sub-type)
- Programming anti-patterns (“anti-idioms”, “persistent idiocies”)
- Methodological anti-patterns
- Testing anti-patterns
- Configuration management anti-patterns

GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design

(OO) Software Design Anti-Patterns

- **Abstraction inversion**: Not exposing implemented functionality required by users, so that they re-implement it using higher level functions
- **Ambiguous viewpoint**: Presenting a model (usually OOAD) without specifying its viewpoint
- **Lava flow**: Dead code and forgotten design information is frozen in an ever changing design
- **Big ball of mud**: A system with no recognizable structure
- **Blob**: Generalization of God object from object-oriented design
- **God object**: Concentrating too many functions in a single part of the design (class)
- **Object cesspool**: Reusing objects whose state does not conform to the (possibly implicit) contract for re-use
- **Object orgy**: Failing to properly encapsulate objects permitting unrestricted access to their internals
- **Poltergeists**: Objects whose sole purpose is to pass information to another object
- **Singletonitis**: The overuse of the singleton pattern
- **Yet Another Useless Layer**: Adding unnecessary layers to a program, library or framework
- **Yo-yo problem**: A structure (*eg*, of inheritance) that is hard to understand due to excessive fragmentation
- **Spaghetti Code**: Ad hoc software structure makes it difficult to extend and optimize code

Plus many-many more: find the The AntiPattern Book and read it, check Wiki's and do your own analysis. It may help you to improve your assignment 3 design ☺.

GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design

Singletonitis Anti-Pattern

This discussion is based on Antonio Vieiro's Sept. 2006 blog entry Singletonitis

The Singletonitis anti-pattern has been identified by Antonio Vieiro.

Diagnosis: Copious use of the *Singleton* objects when you don't need them in your code.

Detrimental Effects:

- Depending on the number of class loaders (objects responsible for loading classes into the JVM) there can be multiple singletons (one per class loader).
- In multithreaded applications, if a singleton instance is shared between threads it has to be synchronized. Thus, the *Singleton* class definition must include the double-checked locking idiom (which is complex, error-prone and depends on the presize memory model for correct implementation).
- Singleton may result into coupling between parts of an application which were meant to be separate.
- The unit testing objective of *testing in isolation* may be compromised if a singleton is involved — due to coupling between the class under testing and the singleton; for details see this entry on IBM Developerworks
- *Singleton* object is not polymorphic (its children are not singletons), so when run-time behaviour selection is needed you have a problem.

Cure:

- Do not use Singleton at all
- Do not use Singleton — use "Simpleton" instead
- Consider using the Monostate pattern (see Lecture 17)
- Benefits of resisting to use Singleton even when temptation is high are presented on Scott Violet' s blog about the Application class

Arguments against the Singleton are also to be found on the WIKI page Singletons are evil

GoF Catalog

Patterns Variability

Selecting DPs

Non-GoF Patterns

Pluggable Selector

Active Object

Null Object

Generation Gap

MVC

JavaEE Patterns

Anti-Patterns

Singletonitis Anti-Pattern

Good Pattern Practices

Patterns and OO Design

How to discover patterns

"Seven Habits" of Effective Pattern Writers

(as professed by John Vlissides in the *Pattern Hatching* book)

- *Take Time to Reflect* — document every step of your pattern discovery process (successful or not), look at as many system as possible (GoF used the rule of having at least two usages for a solution to qualify to become a pattern).
- *Adhere to a Structure* — organize collected raw material into a structure; it should not necessarily be the GoF structure (use more prosey style like Alexandre's, or more rigorous), but *structure nonetheless*: "A pattern is a *structured exposition* of a solution to a problem in a context.
- *Be Concrete Early and Often* — "Motivation" section is necessary in a pattern description, and it should be included at the beginning; "tell the whole truth", *ie*, discuss problems and drawbacks as well as benefits.
- *Keep Patterns Distinct and Complementary* — when patterns overlap, the communication gets more difficult. "How pattern X differs from pattern Y?" Given variability of implementation, this question is often unavoidable.
- *Present Efficiently* — the form ("typesetting") and substance ("writing style") are both important in delivering the presentation quality. Diagrams are probably essential, though not all of them have to be well defined, even a "Dessin dÉnfant" (a child's drawing) conveys more information than many lines of prose.
- *Iterate Tirelessly* — just as you do get your code absolutely right the first (ten) time(s), so is the pattern writing.
- *Collect and Incorporate Feedback* — no pattern can be trusted until it is used by *someone other than its author*.

Personal thought (ABK): pattern writers must be few, pattern users must be plenty. Pattern writing is like poetry or music composition — there *cannot* be many good poets or composers, but many performer (actors and musicians) are needed to keep the art alive (even if they read poems and play music of those who are already dead).

GoF Catalog

Patterns Variability

Selecting DPs

Non-GoF Patterns

Pluggable Selector

Active Object

Null Object

Generation Gap

MVC

JavaEE Patterns

Anti-Patterns

Singletonitis Anti-Pattern

Good Pattern Practices

Patterns and OO Design

Final Words

"Design patterns are wonderful things. They can help you with many design problems. But the fact that they exist does not mean that they should always be used." (R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*). The cost of pattern should not exceed the benefit of its use.

Design Patterns realize the fundamental principles of OO design:

- Program to interface, not implementation
- Prefer Composition to Inheritance

Knowing the Design Patterns helps you to be a better designer, but...

"do not turn off the brain; your creativity is still required. It isn't always clear when to apply a design pattern. In addition you also need to know which variation of a pattern to apply, and how to tweak the pattern. You always need to adapt a pattern to your particular problem" (Erich Gamma in Patterns and Practice)

The *Design Patterns* book tells how to apply patterns, but *removing a pattern* is equally important (and books don't teach you that) — "this can simplify a system and a simple solution should almost always win."

"...you learn patterns by programming. And, not just toy examples; real life examples. But no, you cannot learn patterns just from reading a book. With just a book you might not initially understand them fully. Once you start applying a pattern to one of your own programming problems, you start to understand it a lot better and are ready for the next learning iteration."

GoF Catalog
Patterns Variability
Selecting DPs
Non-GoF Patterns
Pluggable Selector
Active Object
Null Object
Generation Gap
MVC
JavaEE Patterns
Anti-Patterns
Singletonitis Anti-Pattern
Good Pattern Practices
Patterns and OO Design