ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/220995898

Java nano patterns: A set of reusable objects

Conference Paper · January 2010

DOI: 10.1145/1900008.1900089 · Source: DBLP

CITATIONS
12

READS
73

1 author:

Feras Batarseh
Georgetown University
23 PUBLICATIONS **78** CITATIONS

SEE PROFILE

All content following this page was uploaded by Feras Batarseh on 07 May 2019.

The user has requested enhancement of the downloaded file.

Java Nano Patterns: A Set of Reusable Objects

Feras Batarseh
School of Electrical Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd
Orlando, FL 32825
321-276-9448
Feras@mail.ucf.edu

ABSTRACT

Software patterns are used in many applications and domains. They reduce time, effort and cost as they increase reliability, reusability and testability when used for developing software systems. Although they have been criticized for their high level of abstraction, design patterns are the most commonplace patterns used. Furthermore, micro patterns are introduced in the literature. They have a lower level of abstraction and deal with Java software programs at the level of a class while design patterns describe the system design in general. In this paper, I am introducing a lower level of abstraction for Java patterns. Nano patterns are a group of reusable methods that are frequently used in Java software development. Sixteen nano patterns are defined into five groups, according to their privileges, nature and functionality. Additionally, I provide experimentation and initial results and conclude that nano patterns reduce more time, effort and cost for a Java-based software project.

General Terms
Design, Reliability, Experimentation

Keywords
Design patterns, Micro patterns, Nano patterns, Java, Reusability

1. INTRODUCTION

Design patterns became popular after the publishing of a book by the "Gang of four" (GoF), Design Patterns: Elements of Reusable Object Oriented Software [4]. Since then, the Portland Pattern Repository [1] was set to record and document design patterns. Since that time many researchers [1] [3] [6] put many efforts into developing patterns, defining support tools for them and discussing their importance, advantages and disadvantages.
Patterns are a set of reusable solutions for problems that occur over and over in many domains. Patterns are used to minimize testing, documentation and design time, effort and cost. By using patterns, the software engineer/developer prevents some problems from occurring. Furthermore, patterns help developers communicate using agreed on terms for better communication and less misunderstandings [1].
In this paper, I will discuss and give a quick review over design patterns and micro patterns. In section number 2, the main contribution will be discussed. Nano patterns will be introduced with their uses and formal definitions. Finally, I will present the experiment conducted with results, recommendations and conclusions.
First, I will start by introducing the GoF design patterns.

Design Patterns fall under three main categories; these categories are [4]: creational, structural and behavioral design patterns. For complete list of design patterns refer to [4].
As it is well agreed on [1] [3] [5] [6], design patterns cover most aspects of a software system. On the other hand, there are some critical disadvantages for design patterns:

1. They lack formal definitions: No formal definition is introduced for any design pattern.
2. Lead to ineffective solutions: In practice design patterns may result in an unnecessary redundancy of code.
3. Solve the wrong problem: Design patterns work in theory but not always when it comes to practice. They sometimes fail to meet the real problem requirements. This problem is due to the developer's choice of patterns. If the developer couldn't find a suitable pattern, he/ she will choose one that is close enough but not really close.
4. Design Patterns are not different from another patterns introduced in literature.
5. Do not lead to direct code reuse.
6. Some developers consider Design patterns complicated in nature.
7. They are validated and verified by developers' experiences and words of mouth.

Trying to solve these problems, Micro patterns were introduced in 2005 by [5]. They are based on the idea that design patterns are too abstract, and that a new level of patterns abstraction should be introduced to be closer to implementation. Micro patterns [5] capture idioms of Java programming languages. As Gil et al. stated [5]: "*Micro patterns are similar to* design patterns*, except that micro patterns stand at a lower, closer to the implementation, level of abstraction.*" Twenty seven micro patterns are defined for Java programming language. They deal with inheritance, immutability and data management.
These micro patterns and their relations are illustrated in Figure 1. For full understanding of micro patterns functionality please refer to [5].I will give only two examples here, to give a better view of the level of abstraction that micro patterns deal with.
Joiner: is an empty interface that extends another interface.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACMSE'10, April 15–17, 2010, Oxford, MS, USA.
Copyright 2010 ACM 978-1-4503-0064-3/10/04…$10.00.

Pool: This micro pattern is a class with no functions but only static members. Other patterns were introduced in literature to help developers in coding and development in general, but none of these Java patterns dealt with the system at the functions/methods level. Based on this idea, we are introducing our Java nano patterns.

As in [2], Beck discussed implementation patterns which are even closer to implementation-as the name implies- than any other type of patterns. The authors [2] discussed that design patterns are good on the board and when general system design is discussed. This paper aims to introduce implementation patterns for Java. Other authors introduced implementation patterns like Beck et al. [2] for SMALLTALK but none applied implementation patterns to Java. Generally, the lower abstraction is dealt with, the more patterns you have, design patterns are 23, micro patterns are 27, and Beck's implementation patterns [2] are 92.

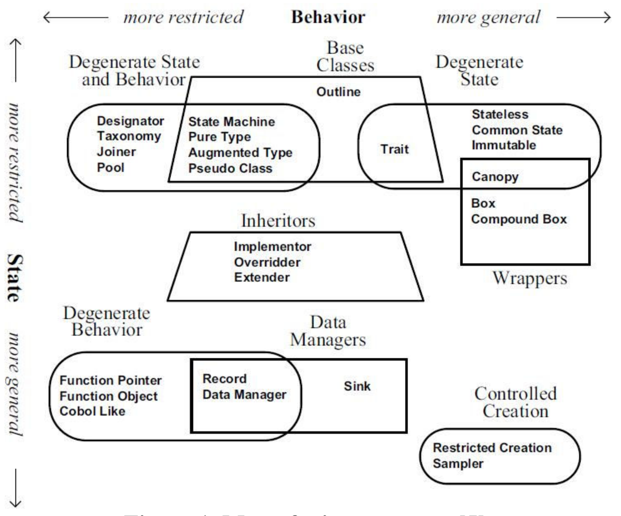

Figure 1. Map of micro patterns [5].

For Java nano patterns, there are currently 16 well defined and formalized patterns that represent methods. Work is under progress to increase their number and extract more patterns from Java. Nano patterns help the developers in reading the code, communicating with their peers and co-workers in a better manner, and most importantly less coding time.
In the next section we will introduce Java nano patterns.

2. NANO PATTERNS

2.1 Nano Patterns Catalogue:

This section starts by introducing the Java nano patterns in the following:

Interface Nano Patterns:
1. Action Triggered methods: methods containing actions after a user's action.
2. Display methods: displaying and clearing Swing and AWT objects.
3. Clear methods: initializing the Applet/ Panel or any container object in Java.
Connector Nano Patterns: (Database/Objects/Serializing objects/Files)
1. Update methods: updates data on the data source.
2. Delete methods: deletes data from the data source.
3. Select methods: selects data from the data source.
4. Insert methods: inserts data from into the data source.
5. Sort methods: sorts data from the data source.
Controller Nano Patterns:
1. Initialize methods (constructors and init method in Java): creates new data members of the class.
2. Destroy Methods: terminates an instance of a defined object, such as the class, data member (arrays).
3. Validating methods: usually returns a Boolean to make a decision accordingly.
Domain Nano Patterns:
1. Control methods: contains the main algorithm.(the core of the class)
2. Processing methods: methods to support the main goal of the class, either by calculating a minor value for use by the control methods or by setting some kind of a constraint.
Dataflow Nano Patterns:
1. Modifier methods: accepts data members as input and change them(value, type(parsers), push, pop…etc)
2. Setter methods: sets values to data members.
3. Getter methods: gets data members' values.
The introduced nano patterns have the following relations among each other:

1. One pattern may lead to another.
2. Some patterns are similar and in some cases alternative.
3. Nano patterns are extractable and documentable.
4. Nano patterns give you hint to solve a problem effectively.
5. Nano patterns give you a standard way of defining any class method members.

One of the main drawbacks of patterns is that they are not formally defined. In this paper, a formal definition for every nano pattern is introduced. The variables used in these definitions and the nano patterns abbreviations will be defined first.

2.2 Formal Definitions and Specifications:

Basic definitions and abbreviations used for formal specifications:

DB: database, Dbcon: database connection, DBq: database query, F: file, Fcon: file connection path, O: an object or class outside this class, IO: interface object, M': another method outside this method in this class, S: group of statements, OM: other objects methods or classes methods, d: data, +: OR, X: AND, edit: edits/ deletes a data member, object, init: initializes a data member,, object, new: defines a data member, object, null: null value, → : call another object, method., ◄R: return, Universal set: Group of defined nano patterns, Action Triggered methods: AM, Display, methods:DisM, Clear methods:CM, Update methods:UM, Delete methods: DM, Select methods: SM, Insert methods: IM, Sort methods: SrtM, Initialize methods:InM, Destroy Methods:DesM Validating methods: VM, Control methods: CntrlM, Processing methods:PM, Modifier methods:MM, Setter methods: SetM Getter methods: GetM.
Formal specifications and descriptions for all the nano patterns are introduced next:
AM: IO → M' (Universal Set) +S
Methods being called with an interface object (Swing, AWT objects) action (ex: mouse clicked, value changed) and might call other methods too, or execute statements such as some minor calculations.

DisM: DisM → IO + S

Methods called from other objects in the class, (note: these methods are preferably private methods) their responsibility is to control the interface flow of objects (ex: Jbuttons, Jlistboxes), this method call interface objects; assign values to them like colors, size, visibility and position.

CM: CM → IO

Methods similair to display methods, but they are responsible for initial setting of the interface, Java is a visual language so in most cases methods for initializing the Japplet/ Jpanel or any Java container should be initialized, with the interface objects that it holds (ex: Jdropdownlist, Jtree, Jlabel), and their attributes. (note: these methods are preferably private methods). This method calls initialized interface objects to build up the screen.

UM: (DBq(d) + Fconn (d) + OM(d)) + S → (F + DB + O)

The only methods that communicate with the data source, these methods update the data source and they usually interact with DB connection classes or file paths. They are critical to the speed of the class execution.

The connector patterns make calls to other classes to exchange data; in these patterns issues like security should be addressed.

This method updates data outside the class, send the data in a database query form, in a file path or as other methods parameters.

DM: (DBq(d) + Fconn (d) + OM(d)) + S → (F + DB + O)

This method deletes data outside the class; send the data in a database query form, in a file path, or as other methods parameters, and is similar to the update methods.

IM: (DBq(d) + Fconn (d) + OM(d)) + S → (F + DB + O)

This method inserts data outside the class; send the data in a database query form, in a file path, or as other methods parameters, and is similar to the update and delete methods.

SrtM: (F(d) + DB(d) + O(d)) + S→ (DBq + Fconn + OM)

This method sorts the selected data that comes into the class, or send a query to return a sorted data form, regardless of the sorting criteria, this method arranges the data coming to the class to maximize the read operation by the class.

SM: (F(d) + DB(d) + O(d)) + S → (DBq + Fconn + OM)

This method gets data from outside the class; gets data into the class in forms as a query form, in a file path or as other methods parameters.

InM: new (d+O+F+IO)

Java applets have init methods as the default methods to initialize new data members, init are an example of this pattern, constructors, overloaded constructors are other examples, these methods could be public, they initialize new instances of data members (ex: arrays, integers) or new interface objects (ex: Jbuttons), create new files (ex: word documents to write on) or other sorts of user defined or built in objects.

DesM: (d+O+F+IO) → null

Java has garbage collectors but sometimes for certain purposes deleting objects is nessecary (ex: deleting a data member), destructors are one example of this nano pattern.

These methods delete/ destruct objects are within the class, and shouldnt be mixed with the DM methods described previously.

VM: OM+M'(Universal Set - GetM)→ VM◄R d

This method works as a serving method for other objects/ methods to validate some condition, or to return a boolean value according to a certain testing method (ex: return 1 if user is administrator or 0 if user is a guest).these methods shouldn't be void methods.

CntrlM: S X (CntrlM→ PM◄R d) + M' (Universal Set-AM)

This method holds the biggest chunk of domain requirements for the class/ system, this method would have domain specifications, usually holds the main logic of the class, and uses PM methods to get extra calculations.

This method shouldn't communicate with outside data sources or objects (should use the connector methods) due to its importance.

This method would be the core method that might call all other methods in the class (ex: main methods).

PM: (CntrlM→ PM◄R d) + S + PM→M' (Universal Set-AM X CntrlM)

This method is a supporting method for CntrlM and can call other methods within the class for some calculations/ data/ actions, but shouldn't be void methods.

MM: d → MM◄R d

This method should have at least one parameter as its input; it accepts data and returns values after a modification, a calculation or an action.

These methods are the ones that most affect the data in the class.

SetM: edit d

These methods are similar to MM, but are more specific to one data member and shouldn't have any operations or calculations, they just accept data (ex: from select methods), and set data to data members, they must be private, while MM could be public.

GetM: M' (Universal Set) ◄R d

These methods don't accept any parameters but should return a value (cannot be void).

They are responsible of getting the values, and they could be public, since they don't have any modification privileges.

Based on these definitions, the programmer will have a formal way of deciding what patterns to use, how to connect patterns together and where to use each pattern. In the next section, I will introduce some example Java code snippets for every pattern.

Exmaple Java code snippets are introduced next to illustrate the some of the nano patterns in more detail:

Action Triggered methods:

```
jButton1.addActionListener(new Java.awt.event.ActionListener()
{public void actionPerformed(Java.awt.event.ActionEvent e)
x=10 ;}
```

Initialize methods:

```
private CP jContentPane = new jContentPane() ;
private JPanel jPanelquizinapplet = null;
public LoginScreen loginScreen;
```

Destroy Methods:

```
Questions.removeLast();  //Questions=newLinkedList<int>()
```

Validating methods:

```
private boolean IsGreaterthanfive(int x)
{if (x>5) return true;
else return false;}
```

Control methods:

```
if (phType == Complex.ANGDEG) {
ph = (ph / Math.PI) * 180;}
x = m * Math.cos(ph);
y = m * Math.sin(ph);
```

Modifier methods:

```
public int[] getGrades() {
Iterator<OneAttempt> it= att.iterator();
int[] grades=new int[numberOfAttempts()];
int i=0;
while(it.hasNext()) {
grades[i++]=it.next().grade();
} return grades;}
```

All these code snippets are from the system built using nano patterns. This system and others will be discussed in the next section.
Another approach for usage of patterns is to extract them from an already existing code, to refine the code and restructure it. In the next section, experiments and results will be introduced.

3. EXPERIMENTS AND RESULTS

Three middle-size systems were built using design patterns, micro patterns and nano patterns. The first system built using design patterns is a trivial plane flight black box simulator. This system is called BlackBox. BlackBox saves all the actions from the pilot, like the changes of pitch degree, yaw and roll of the plane, the pilot changes these values all the time while flying the plane. The second system is a similar system. It has the same number of classes and methods. It is a car navigation system where users/ drivers request information from the system about the roads they are on. The roads are assigned to a driver randomly. This system was built using Micro patterns. The third system which was built using nano patterns is a ticketing system for airlines. In this system users can ask for customers' lists and the tickets they purchased with some details like age and flight numbers. In these experiments, I tried to assign each type of patterns a fair problem.

Table 1. Experimental results: comparison between three types of patterns.

Pattern	Time for Analysis Phase	Time for Design Phase	Time for Coding Phase	Time for Testing Phase	Total Time
Design	2	1	20	5	28
Micro	2	2	22	4	30
Nano	2	4	12	2	20

The three systems have a lot in common, same domain, same size and number of classes. Results were recorded and a comparison between the three approaches is introduced here based on the time spent in development for each life-cycle phase. The results are illustrated in table 1 and Figure 2. Design patterns that were used in the first system are: Singleton, Bridge and Façade. Micro patterns used are: Joiner, DataManager and Sink. Nano patterns used in the third system are: ActionTriggered, Display, Initialize, Getter, Setter, Select, Delete, Update and Processing. As illustrated in figure 2, the analysis time was the same for all the patterns due to the similarity of the three systems. At design time, design patterns were the best. They provided high-level templates for design, but this didn't help in reducing coding and implementation time. When implementation started, nano patterns were the best due to the "drag and drop" fashion. I was able to select methods/ nano patterns and had the classes ready. I just needed to set the classes' parameters. Implementation for micro and design patterns took longer time due to the abstract representations that came out from the design phase. Testing and code readability was best for nano patterns. Micro patterns and design patterns saved some time at early development phases but nano patterns superiority appeared at the total time consumed to build the three projects.

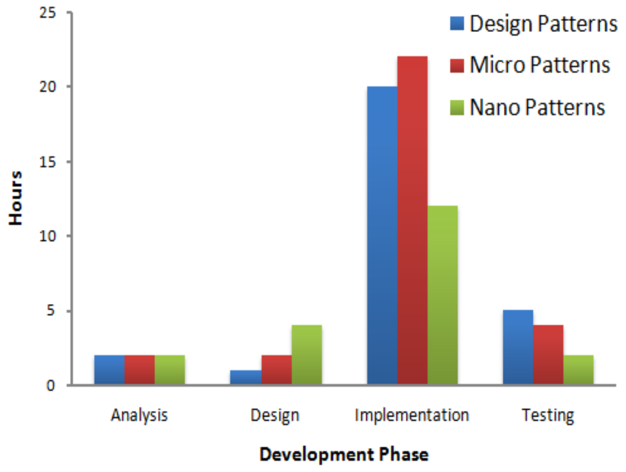

Figure 2. Experimental results.

Nano patterns used fewer resources than both design and micro patterns.

4. CONCLUSIONS AND FUTURE WORK

This paper introduced nano patterns, a set of reusable Java objects. The paper discussed nano patterns against design patterns [4] and micro patterns [5]. An experiment was conducted to illustrate comparison between the three types of patterns. Nano patterns saved the most time. The systems under experiments were midsize systems so it can be concluded that nano patterns are best for small and midsize systems. Experiments for large systems are undergoing. This research will continue with planning to add more nano patterns to the catalogue, building the software tool that encapsulates all the nano patterns and experimenting nano patterns on more systems and against more types of patterns.

5. REFERENCES

[1] Agerbo, E., and Cornils, A. 1998. How to Preserve the Benefits of Patterns. In Proceedings of the 13th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98). Pages 134-143

[2] Beck, K. 1997. SMALLTAK: best practice patterns. Published by Prentice Hall, 1st edition.

[3] Dong, J., Yang, S., and Zhang, K. 2007. Visualizing Design Patterns in Their Application and Compositions. IEEE Transactions on Software Engineering, Vol. 33, No. 7, July 2007

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.1995. Design Patterns: Elements of Reusable Object-Oriented Software. Published by Addison Wesley.

[5] Gil, J. and Maman, I. 2003. Micro Patterns in Java Code. In Proceedings the 20th OOPSLA conference 2005, pp 97-116.

[6] Lauder, A. and Ken, S. 1998. Precise visual specification of design patterns. In Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP) Brussels, Belgium 1998.

View publication stats