UML Diagram for Design Patterns

Muhazam Mustapha and Nik Ghazali Nik Daud

National Defense University of Malaysia,
Sungai Besi Camp, 57000 Kuala Lumpur, Malaysia
`{muhazam,nikghazali}@upnm.edu.my`

Abstract. UML has been used widely as a design and analysis tool for object-oriented software projects. Despite of having a good number of diagram categories, UML is still lacking of a dedicated set of diagrams for representing design patterns. While numerous works have been published on attempts to describe design patterns in terms of UML features, there is hardly any work has been done on attempts to un-clutter class diagrams at the initial development stage so that the design patterns in used can be seen clearly. With the aim to tidy-up huge networks of interconnecting class diagrams, this paper proposes a new higher level set of diagrams and layout schemes to better highlight the design patterns used in a software architecture. This proposal can go further as a proposal for a new set of diagram in UML standard.

Keywords: Design Patterns, Unified Modeling Language (UML), Architecture Description Language (ADL), Object Modeling Language (OML).

1 Introduction

1.1 UML and Design Pattern Background

One can agree that at the highest level of system design, design pattern [12] has been adopted as a way to model and communicate software ideas and architecture [17][29][34] and, more importantly, to start the design idea not-from-scratch [3][21]. Not only that pure software projects can be very conveniently presented in design pattern language like the projects given by Korhonen et al. [20] and Rößling [32], there are also many projects that are not entirely pure software, but have been successfully presented as design patterns. The examples are the parallel computation by Huang et al. [16] and database related projects by Pasala and Ram [28] and Wernhart et al. [37]

One of the most common ways to represent design pattern is through the use of unified modeling language (UML). However, unknown to many, the de facto inventor of design pattern, Gamma et al. did not represent the design patterns in their book [12] in UML format, but in object modeling technique (OMT) instead.

UML has been around since 1997, and as of now it comprises of 14 diagram sets [10][26][27]. Despite the fact that it has been used widely in program development, and despite the fact that it has so many diagrams – that may seem to be redundant – UML is still being regarded as incomplete [10]. To be more specific, in this paper, the author would like to highlight that, even though design patterns have been represented in UML, there is none of its 14 diagrams is dedicated for design pattern. The author is not the only one making this claim because Riehle, in [31] Section “3 - The Tools and UML Community,” has also made the same statement. Due to this shortcoming, there are numerous works have been done to represent design patterns in form of existing UML diagrams. For example, Dong et al. [4][5][6][7], Le Guennec et al. [14], and many other significant works [33][35][36], have performed extensive studies to achieve such visualization with UML features. The infrastructures that have been used include annotated package diagrams, collaboration diagram, roles, tags, stereotypes, metamodel profiling, as well as Venn-diagram-like dotted or shaded area, object constraint language [25] and Object-Z [19]. Similar works have also been done by France and Kim et al. [11][18] to accommodate the insufficiency of UML class diagrams to represent design pattern through the use of UML sequence and state diagrams.

Not only UML is insufficient for representing design patterns, design patterns themselves are inadequately defined. Many researchers have put a lot of effort to precisely define design pattern like Lauder and Kent [22], Le Guennec et al. [14], Mak et al. [24] and Eden et al. [8][9]. Formal definition of design patterns, not necessarily in form of UML, is vital in many research works. Software reverse engineering projects, like developing tool support for automated recognition and recovery of design patterns [1][23][30][43] and the evaluation of the design pattern itself [15], are all craving for precise definition of design patterns -otherwise the code developed for such projects won't work. The significance of such works become obvious if the code was not well documented or was not developed based on any design pattern.

To complete this background introduction, it is worthwhile to mention one noble example of non-UML scheme to visually and formally specify design pattern, i.e. the one done by Gasparis et al. [13] using LePUS.

1.2 Placing This Paper into the Background

Jakubík [17] proposed almost the same idea of representing design patterns with new kind of diagrams. Besides that, Yacoub and Ammar have used *interface diagrams* in [39][40] to illustrate a way to represent design patterns. These three articles have almost presented the very idea that is to be delivered in this paper. However, their main purpose wasn't to develop a new type of diagram that has potential to be part of UML. This gap that has been left behind by these three closest articles is the one that is to be filled in by this paper.

Besides the three papers, Zdun [41][42] and Buschmann et al. [2] have also presented quite a similar idea but more on expressive languages rather than diagrams.

Other than that, this paper is also focusing on simple, front-end sketch-type diagram that is to be used at inception phase of software development. This, on the other hand, is the gap that the background materials of sophisticated back-end development tools from various literatures in the previous subsection have left behind for this paper.

Dedicated Diagram (Possibly UML) for Design Pattern. As mentioned in previous section, UML has no dedicated diagram for design pattern [31]. As a result, there are many works have been published on attempts to extend or manipulate existing UML features to properly define design pattern. The literatures presented in the previous section show many impressive works to achieve such visual representation, but there is still no work on proposing some proper and dedicated diagrams for design pattern itself.

Filling up this very gap of lacking of dedicated diagram for design pattern is the *first* target of this paper. This diagram can be considered as an architecture description language (ADL) since at the level of design pattern, the scope is architecture. It can also be further proposed to be formally included in UML.

Uncluttered Diagram Layout. The works of Dong et al. in [4][5][7] are very good examples of the attempts to make design patterns stand up in a complicated initial software design. However, the methods used in those literatures have obviously cluttered the entire diagram. The reason is because the very features used to highlight design patterns have added untidiness to the diagram. Contrarily, a good and nicely laid out design diagrams is very vital for the initial stage of software development – inception phase.

Solving this cluttering problem is the *second* target of this paper. This will be achieved by equipping the dedicated design pattern diagram mentioned in the previous section with a set of layout and notation schemes whose main purpose is to reduce the cluttering.

2 Groups of GoF's Design Patterns

To achieve the target in Subsection 1.2 (dedicated design pattern diagram), this paper first has to advocate the design patterns given by GoF[1] in [12] as *elementary design patterns*. It is assumed that most of more recently discovered design patterns would be a composition of these elementary patterns. Should there be a genuinely new elementary design pattern, it should be published and would be added to the collection in Figure 8 - 9.

In order to better understand the way to convert the GoF's elementary patterns to the corresponding dedicated design pattern diagrams, those elementary design patterns would be split into seven *groups*. These groups are constructed based on structural or behavioral similarity among the patterns.

The next seven subsections are dedicated to explaining the key features that are common to the patterns in the same group, as well as the features that make each pattern in the groups unique. The reader is expected to be familiar to GoF's pattern. For that reason, the complete description of the pattern won't be presented in this paper, but rather the following seven subsections will only focus on the similarities and uniqueness of the patterns.

[1] GoF, or Gang of Four, is the commonly used short form of the four authors of *Design Patterns - Elements of Reusable Object-oriented Software* - number 12 in the reference list.

2.1 Plain Single Family (PSF)

This group consists of one abstraction hierarchy, and mainly only this hierarchy. The uniqueness of each pattern is explained as follows:

i. Template Method: The *simplest* form. Other patterns in this group can be thought of as a form of template.
ii. State: Has *aggregated* pluggable behavior.
iii. Strategy: Has *non-aggregated* pluggable behavior.
iv. Prototype: Strategy pattern with *pluggable cloning* method.
v. Adaptor: Template of *hybrid* or composed classes.

2.2 Managed Single Family (MSF)

This group consists of one abstraction hierarchy, and the client gains access to it through a *middle man*.

i. Command: The middle man is the *invoker* class.
ii. Flyweight: The middle man is the *flyweight factory* class.

2.3 Single Intra-Family Collaboration (SIFC)

This group consists of one abstraction hierarchy with some kind of collaboration exists *within* the hierarchy.

i. Composite: Collaboration in form of aggregate of *exclusive* copies of parents in children.
ii. Decorator: Collaboration in form of aggregate of parents in children (*might not* be exclusive copies).
iii. Interpreter: Collaboration in form of aggregate of parents in children, with a possible use of global *context object*.
iv. Chain of Responsibility: Collaboration between *parents*.
v. Proxy: Collaboration between *children*.

2.4 Two Family Collaboration (TFC)

This group consists of two collaborating class hierarchies.

i. Bridge: Collaboration only between *parents* of the families.
ii. Observer: Collaborations of *opposite directions* at parents and children level between families. Usually with aggregation but it is optional.
iii. Mediator: Collaborations of opposite directions at parents and children level between families. Usually *no* aggregation.

2.5 Family Optional (FO)

This group of patterns consists of non-relating classes.

i. Singleton: Make use of *static* property of class.
ii. Memento: Dynamic creation of *pooled* states.

2.6 Object Builder (OB)

This group mainly serves as object creators.

i. Builder: Involves only *one* family tree.
ii. Factory Method: Involves only *two* family trees.
iii. Abstract Factory: Involves *more than two* family trees.

2.7 Looping Strategy (LS)

This group mainly serves as aggregate processors.

i. Iterator: Looping object is *provided by the aggregate.*
ii. Visitor: Looping object is *given to the aggregate.*

3 Dedicated Design Pattern Diagram

The general construct of the proposed dedicated design pattern diagram is as follows:

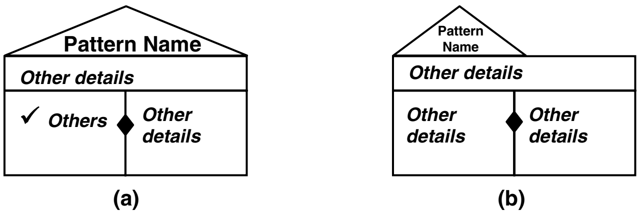

Fig. 1. Design Pattern Diagrams

Except for the triangular shape at the top, this diagram resembles class diagram. The triangle holds the pattern name with the base can be as wide as the whole box or just nice to hold the name. Unlike class diagrams, that contain the information about methods and properties of the class, design pattern diagrams contain the information about the *classes* that form the pattern. By putting only names of classes in design pattern diagrams, the cluttering caused by the details in class diagrams is greatly reduced.

There are additional infrastructure tokens to be used with the box:

i. *Check:* Denotes that the class is provided by a framework.
ii. *Black Diamond:* Denotes *sole mate class pairs* or couples. A sole mate pair is a pair of classes or children that are mainly collaborating only between the two of them, and less or none with other classes in the pattern.
iii. *Connector Line:* Denotes patterns collaboration, or more specifically, denotes which class in a certain pattern collaborates with which class in another pattern. More on this will be in Subsection 3.2.
iv. *Other relevant UML infrastructure:* Since this diagram is honoring UML, all other UML infrastructures that might fit into the diagram can be used with it. The ones that might be used mostly are the stereotype and abstract class indicator.

Note: It is important to stress that the rules that will be presented in this paper (especially in this section) would be a set of loosely defined and implemented rules. This non-stringent-rules approach is inline with the way the rules of UML and design pattern have been implemented. While honoring preciseness as the main target in [14][22][24], this paper has to deviate from this mainstream works as it has its own target, i.e. to reduce the cluttering in design diagram at inception phase. Maintaining the way of UML and GoF's design pattern is the best option to achieve this.

3.1 Conversion Rules

This subsection describes, in writing, the rules to convert the OMT or UML based GoF's design pattern into this new dedicated design pattern diagram. Please refer to Figure 8 - 9 in the Appendix for the complete list of actual visual conversion.

i. *Orphan classes:* If the pattern contains any orphan classes (classes that are not extended from any abstract parent, like the invoker and receiver classes in command pattern), these classes have to be listed first, in any order, in the box and in single column.

   Applicability: MSF patterns as they have orphan middle men; FO patterns as all classes are orphans; and the visitor pattern since the aggregate is part of the pattern.

ii. *Parent's name:* Put parent's names before any children's names but after the orphans. If there is any orphan exists, put a horizontal double line separating them from the abstract parents. If there are only one or two families, the parent names are in one or two columns respectively. If there are more than two families, like in abstract factory pattern, everything is in one column and each family is separated by a horizontal double line.

   Applicability: All patterns except adaptor.

   Exception: Since *adaptor* pattern is using other classes, which might be orphans, as adaptees or for forming a hybrid, these classes need to be listed side by side with the respective children. This means two columns are needed for children-adaptee pairs, but the parent is only written on top of the children's column. The box on top of the adaptees is empty, *unless* the adaptees belong to another parent.

iii. *Children's name:* If there are two families, the children are written side by side in two columns. Put a diamond on the border between them if they are sole mate pairs.

   Applicability: All LS, TFC, and factory method pattern.

   If there is only one family, and if there is no distinct two groups of children, then list the children in one column.

   Applicability: Chain of responsibility, builder, and all MSF and PSF patterns except adaptor (for adaptor, see (ii) above).

   If there is only one family, but there are two distinct groups of children, then list them in two columns, and pair them side by side with a diamond if they are collaborating.

   Applicability: All SIFC patterns except chain of responsibility.

   Exception: The abstract class *decorator* in decorator pattern is considered a child since it is a child of another abstract class *component.*

Note: In many cases, parent or abstract class may be implemented as interfaces. As a matter of fact, when two patterns collaborate, the affected classes have to inherit from more than one base or parent classes. In languages that prohibit multiple inheritance, this can only be done with interface.

3.2 Layout Scheme

The Need for Connection Network. Having a software architecture expressed in form of design patterns means there must be interconnections between the patterns. Attempts to visually highlight design patterns in a software system using existing UML infrastructures or by dotted or shaded area as in [5][7] have shown an increase in the complexity of the diagram. The dotted or shaded areas of the collaborating patterns would show *overlapping* classes. This is understood as the collaboration is achieved through the sharing of these classes. This also explains why multiple inheritance or multiple interfaces are needed for the shared classes – these classes need to inherit from more than one abstract parents (or more than one interfaces) that make up the collaborating patterns.

The conversion rules described in Subsection 3.1 reduce the complexity of the software architecture diagram by removing the lines between the class diagrams. How the classes in the respective design pattern diagram are connected are as GoF described in their book [12]. However, the relationship between the design patterns still needs to be shown. Since the collaborating patterns are sharing some common classes among them, such classes will be *repeated* in the design pattern diagram. In order to *highlight* this repetition, these common classes would be connected using a line between the two design pattern diagrams.

Each and every design pattern diagram in a system must be connected to at least one other design pattern diagram. Otherwise the pattern is not used, or the pattern is used only by the client or main program which is normally not shown in the system.

Connection Rules. The simple connection rule between the repeated classes in the collaborating design patterns as explained in the previous sub-subsection has a potential to re-introduce complexity into the design diagram. To avoid such issues, a few rules are proposed so that the connections between the design pattern diagrams can be drawn without adding any complexity to the entire diagram.

i. *Connect only concrete classes:* Design patterns may seem to share the abstract and the concrete classes, but actually the sharing of the abstract one is understood from the structure. So it is necessary only to show the link between the concrete ones, unless the sharing actually involves the abstracts.
ii. *Only straight line horizontally or vertically:* As much as possible, stick to drawing the connectors horizontally or vertically as straight lines only.
iii. *Right angle bends or curves:* If it is unavoidable, bending or curving at right angle can be done. Curves should be drawn at smallest radii possible.
iv. *Linking to a class at opposite side:* If it is unavoidable, a connector line can be drawn to join a design pattern diagram with a triangle (Figure 2) to show that it is connected to the class at the opposite side rather than the one it is joining to.

v. *Passing through a diagram:* If it is unavoidable, a connector line can be drawn passing behind a design pattern diagram with a *stopper* line at both sides (Figure 3). If there are many lines involved, the same sequence should be maintained at both sides. Optionally, the lines can be labeled with matching encircled numbers at one or both ends. They are compulsory if the sequences at the sides are different (Figure 4).

Note: Just like other UML diagrams, the above connection rules are only loosely enforced. Should it cause confusions or should it cause even more cluttering, then it is advised not to follow the above rules. Stick to whichever method that really solve the cluttering problems.

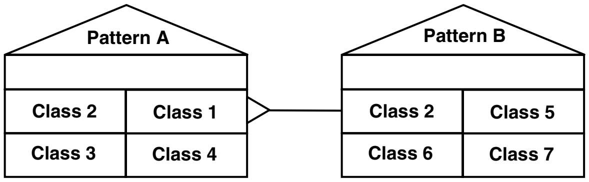

Fig. 2. Linking to opposite side - Pattern A and B share Class 2

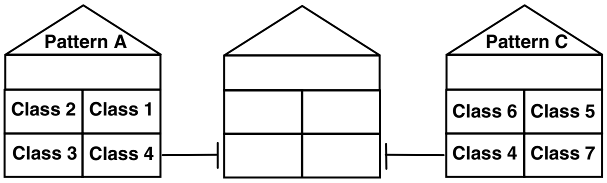

Fig. 3. Link passes behind - Pattern A and C share Class 4

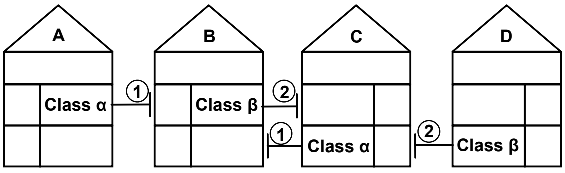

Fig. 4. Swapped links with labels - A and C share Class α; B and D share Class β

4 Examples

4.1 Iterable Composite

A popular tutorial question in a design pattern course is to ask the students to implement iteration on a composite pattern. This can be shown as the following UML diagram and the conversion:

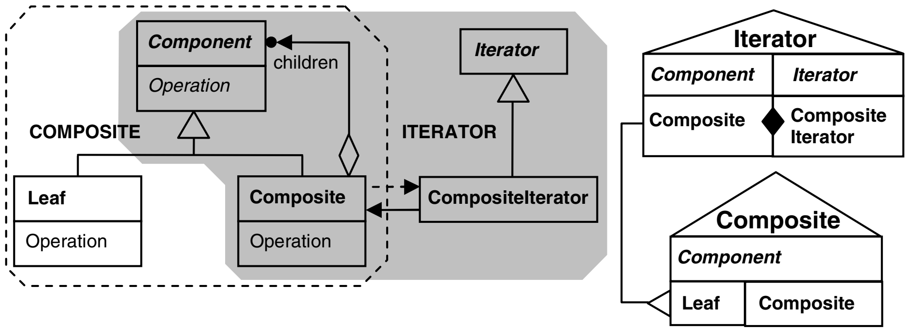

Fig. 5. Iterable Composite Pattern

4.2 Conversion of Dong's Example [5]

Figure 6 is a simplified re-drawn figure of Dong's example in [5], and the conversion of it into the proposed design pattern diagram. As opposed to Dong's example, the converted version opts to un-share the abstract Component class between the patterns Composite and Decorator – which makes the two patterns share only the class Content. This should be a better design as it increases coherence and reduces coupling. This also means the class Content needs to implement two interfaces.

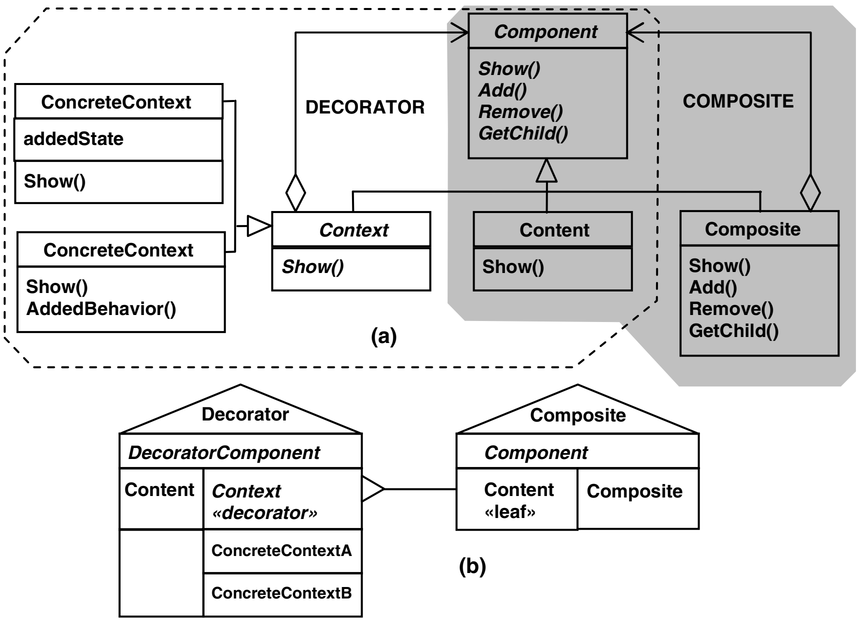

Fig. 6. Conversion of Dong's Example

4.3 Adaptor with Strategy

When two incompatible systems are connected with an adaptor pattern, it is extremely cumbersome if the adaptee system is always updated since the adaptor needs to be updated accordingly. This can be solved if the adaptee side is implemented as strategy pattern:

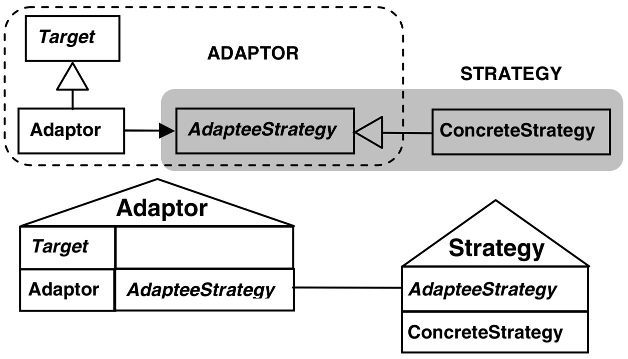

Fig. 7. Adaptor with Strategy Pattern

4.4 Singleton

Singleton is the only design pattern that is completely representable by class diagram *alone* since it is composed of only one class. See the conversion in Figure 8(d).

5 Conclusion

The most immediate and significant use of this diagram is during the inception phase [3][21]. At this stage, the developers normally would like to see the big picture of the project which makes the top-down design mode more convenient.

The patterns would first be laid out on the design diagram, and then the interconnections between them would be decided. Two patterns are interconnected if they share some common classes. These shared classes play extra roles as they are playing one role on one end of the connection (pattern) and another role on the other end. These interconnections would, in a very natural manner, give us the solution model for the problem domain by suggesting what extra roles that the classes in the respective pattern should play with respect to the pattern that it is connected to. The operation or method contracts would also be easy to be anticipated just by looking at the type of the pattern used. Once the pattern roles and those extra roles played by the classes are finalized, it is easy to start with the proper OOD mode of development, i.e. bottom-up. This is a great help in designing the system, for example, using CRC session approach [38].

Other than just a clearer layout of software architecture, the proposed design pattern offers some more advantages as follows:

i. *Less documentation:* The diagram is self-documenting. The structure of the individual pattern is known and the relation among them is clearly pictured.
ii. *Clear highlight of update hotspot:* Since the diagram consists of GoF patterns, the places where the update would normally be done are already documented by them.
iii. *Clearer class hierarchy documentation:* The information about which class extends which class is obvious from the individual diagrams, and any multiple inheritance or interfaces can be seen from the connection of the classes.
iv. *Ensures the best granularity:* Example 4.2 shows how easy it is to split (refactor) an abstract class based on which pattern it belongs to. The sharing nature shown in the diagram can effectively suggest the best granularity level for the system.

5.1 MOF Definition

There is clearly a need to define the diagram proposed in this article using MOF (meta-object facility). At this stage of the research however, it is decided that it would be good to expose to the world about the idea of having such diagram first before proceeding to the formal definition of the diagram. The move to MOF definition would definitely be in the next future step of this work.

5.2 What This Work Is NOT

Due to the numerous strong mainstream works on design patterns that are not in the same direction as this article that might mislead our readers, it is good to make some clear proclamation statements about the direction of our works. These statements are about what our works are NOT meant for, and the purpose is to avoid any *unfair* expectations from our work that are not in our research's problem statements. The author also hopes that, by stating these clear statements of disclaimer, the mind can be made more open to new directions in this field instead of following the mainstream works that might already be close to saturation.

- *This work is NOT meant to expressively define design patterns.* Due to this, the proposed diagram set won't have any feature for such purposes either in the patterns' structure, behavior, constraint or instantiation. This article honors all cited works related to patterns definition but our problem statement is different, i.e. to compactly visualize design pattern at architecture level to hide the class details, instead of visualizing at pattern level that reveals the class details.
- *This paper is NOT meant to catalogue all existing design patterns.* Even though the author is advocating patterns of GoF's, it doesn't mean the existence of other patterns is unnoticed. At this level GoF's patterns are only accepted as the most elementary patterns – due to the fact that they are among the earliest documented – and the author expects to be able to define

most of later patterns in term of GoF's patterns. The proof of this theory or otherwise would be part of the next step in this work.

- *This work is NOT, in any way, forcing the programming community to memorize all GoF's patterns.* The popularities of GoF's patterns are not all the same. An experienced programmer can tell the structure of the most widely used GoF's pattern by heart and conveniently draw the diagram of it.

References

1. Antoniol, G., Casazza, G., Di Penta, M., Fiutem, R.: Object-oriented design patterns recovery. Journal of Systems and Software 59(2), 181–196 (2001)
2. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture: On Patterns and Pattern Languages. John Wiley & Sons, Inc., Hoboken (2007)
3. Cantor, M.R.: Object-oriented Project Management with UML. John Wiley & Sons, Inc., New York (1998)
4. Dong, J.: UML Extensions for Design Pattern Compositions. Journal of Object Technology 1(5), 149–161 (2002)
5. Dong, J.: Representing the Applications and Compositions of Design Patterns in UML. In: Proceedings of The 2003 ACM Symposium on Applied Computing, Melbourne, Florida, USA (March 2003)
6. Dong, J., Yang, S.: Visualizing Design Patterns with A UML Profile. In: Proceedings of IEEE Symposium on Human Centric Computing Languages and Environments, Auckland, New Zealand (October 2003)
7. Dong, J., Yang, S., Zhang, K.: Visualizing Design Patterns in Their Applications and Compositions. IEEE Transactions on Software Engineering 33(7), 433–453 (2007)
8. Eden, A.H., Yehudai, A.: Patterns of the Agenda. In: Dannenberg, R.B., Mitchell, S. (eds.) ECOOP 1997 Workshops. LNCS, vol. 1357, pp. 100–104. Springer, Heidelberg (1998)
9. Eden, A.H., Yehudai, A., Gil, J.: Precise Specification and Automatic Application of Design Patterns. In: Proceedings of 12th IEEE International Conference on Automated Software Engineering, Incline Village, Nevada, USA (November 1997)
10. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edn. Pearson Education, Inc., Boston (2004)
11. France, R.B., Kim, D.K., Ghosh, S., Song, E.: A UML-Based Pattern Specification Technique. IEEE Transactions on Software Engineering 30(3), 193–206 (2004)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-oriented Software. Addison-Wesley, Indianapolis (1995)
13. Gasparis, E., Nicholson, J., Eden, A.H.: LePUS3: An Object-Oriented Design Description Language. In: Stapleton, G., Howse, J., Lee, J. (eds.) Diagrams 2008. LNCS (LNAI), vol. 5223, pp. 364–367. Springer, Heidelberg (2008)
14. Le Guennec, A., Sunyé, G., Jézéquel, J.-M.: Precise Modeling of Design Patterns. In: Evans, A., Caskurlu, B., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 482–496. Springer, Heidelberg (2000)
15. Hsueh, N.L., Chu, P.H., Chu, W.: A quantitative approach for evaluating the quality of design patterns. The Journal of Systems and Software 81(8), 1430–1439 (2008)

16. Huang, K.C., Wang, F.J., Tsai, J.H.: Two design patterns for data-parallel computation based on master-slave model. Information Processing Letters 70(4), 197–204 (1999)
17. Jakubík, J.: Modeling Systems Using Design Patterns. In: Informatics and Information Technologies Student Research Conference, April 2005, pp. 151–158. Slovak University of Technology, Bratislava, Slovakia (2005)
18. Kim, D.K., France, R., Ghosh, S., Song, E.: A UML-Based Metamodeling Language to Specify Design Patterns. In: Proceedings of Workshop on Software Model Engineering (WiSME) with Unified Modeling Language Conference, San Francisco, California, USA (2003)
19. Kim, S.K., Carrington, D.: Using Integrated Metamodeling to Define OO Design Patterns with Object-Z and UML. In: 11th Asia-Pacific Software Engineering Conference, Busan, Korea (November 2004)
20. Korhonen, A., Malmi, L., Saikkonen, R.: Design Pattern for Algorithm Animation and Simulation. In: Proceedings of the First Program Visualization Workshop, Joensuu, Finland (July 2000)
21. Larman, C.: Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development, 3rd edn. Pearson Education, Inc., Upper Saddle River (2005)
22. Lauder, A., Kent, S.: Precise Visual Specification of Design Patterns. In: 12th European Conference on Object-Oriented Programming (ECOOP 1998), Brussels, Belgium (July 1998)
23. De Lucia, A., Deufemia, V., Gravino, C., Risi, M.: Design pattern recovery through visual language parsing and source code analysis. Journal of Systems and Software 82(7), 1177–1193 (2009)
24. Mak, J.K.H., Choy, C.S.T., Lun, D.P.K.: Precise Modeling of Design Patterns in UML. In: Proceedings of 26th International Conference on Software Engineering (ICSE 2004), Scotland, UK (May 2004)
25. Object Constraint Language Version 2.0 (2006), Object Management Group, `http://www.omg.org/cgi-bin/doc?formal/06-05-01.pdf`
26. OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.2 (2009), Object Management Group, `http://www.omg.org/spec/UML/2.2/Infrastructure`
27. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2 (2007), Object Management Group, `http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF`
28. Pasala, A., Ram, D.J.: FlexiFrag: A design pattern for flexible file sharing in distributed collaborative applications. Journal of Systems Architecture 44(12), 937–954 (1998)
29. Pauwels, S.L., Hübscher, C., Bargas-Avila, J.A., Opwis, K.: Building an interaction design pattern language: A case study. Computers in Human Behavior 26(3), 452–463 (2010)
30. Rasool, G., Philippow, I., Mäder, P.: Design pattern recovery based on annotations. Advances in Engineering Software 41(4), 519–526 (2010)
31. Riehle, D.: The Perfection of Informality: Tools, Templates, and Patterns. Cutter IT Journal 16(9), 22–26 (2003)
32. Rößling, G.: A First Set of Design Patterns for Algorithm Animation. Electronic Notes in Theoretical Computer Science 224, 67–76 (2009)

33. Schleicher, A., Westfechtel, B.: Beyond Stereotyping: Metamodeling Approaches for the UML. In: Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS), Maui, Hawaii, USA (January 2001)
34. Smith, J.M., Stotts, D.: Elemental Design Patterns: A Link Between Architecture and Object Semantics. Technical Report. Department of Computer Science, University of North Carolina at Chapel Hill, North Carolina, USA (2002)
35. D'Souza, D., Auletta, V., Birchenough, A.: First-Class Extensibility for UML - Packaging of Profiles, Stereotypes, Patterns. In: France, R.B. (ed.) UML 1999. LNCS, vol. 1723, pp. 265–277. Springer, Heidelberg (1999)
36. Sunyé, G., Le Guennec, A., Jézéquel, J.-M.: Design patterns application in UML. In: Hwang, J. (ed.) ECOOP 2000. LNCS, vol. 1850, p. 44. Springer, Heidelberg (2000)
37. Wernhart, H., Kühn, E., Trausmuth, G.: The replicator coordination design pattern. Future Generation Computer Systems 16(6), 693–703 (2000)
38. Wilkinson, N.M.: Using CRC Cards: An Informal Approach to Object-oriented Development, SIGS Books, New York (1995)
39. Yacoub, S.M., Ammar, H.H.: Pattern-Oriented Analysis and Design. Addison-Wesley, Boston (2004)
40. Yacoub, S.M., Ammar, H.H.: Pattern-Oriented Analysis and Design (POAD): A Structural Composition Approach to Glue Design Patterns. In: Proceedings of 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS), Santa Barbara, California, USA (July 2000)
41. Zdun, U.: Some patterns of component and language integration. In: Proceedings of EuroPLoP 2004, Irsee, Germany (July 2004)
42. Zdun, U., Avgeriou, P.: Modeling Architectural Patterns Using Architectural Primitives. In: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, San Diego, California, USA (October 2005)
43. Zhu, H., Bayley, I., Shan, L., Amphlett, R.: Tool Support for Design Pattern Recognition at Model Level. In: 2009 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009), Seattle, Washington, USA (July 2009)

Appendix: The Complete Catalogue of GoF's Design Pattern Conversion

This appendix (Figure 8 and 9) presents the complete list of GoF design patterns with their conversion into the new design pattern diagrams. This list can be used as reference in converting a bigger system as long as it is based on GoF's patterns.

Should there be a discovery of a genuinely new pattern, for the convenience of everyone, it is suggested that the discoverer himself should document the corresponding conversion into this new diagram after considering all criteria proposed in this paper. The author of this paper is not in any way tries to force a consensus in the conversion of any new pattern – the discoverer's decision should be honored.

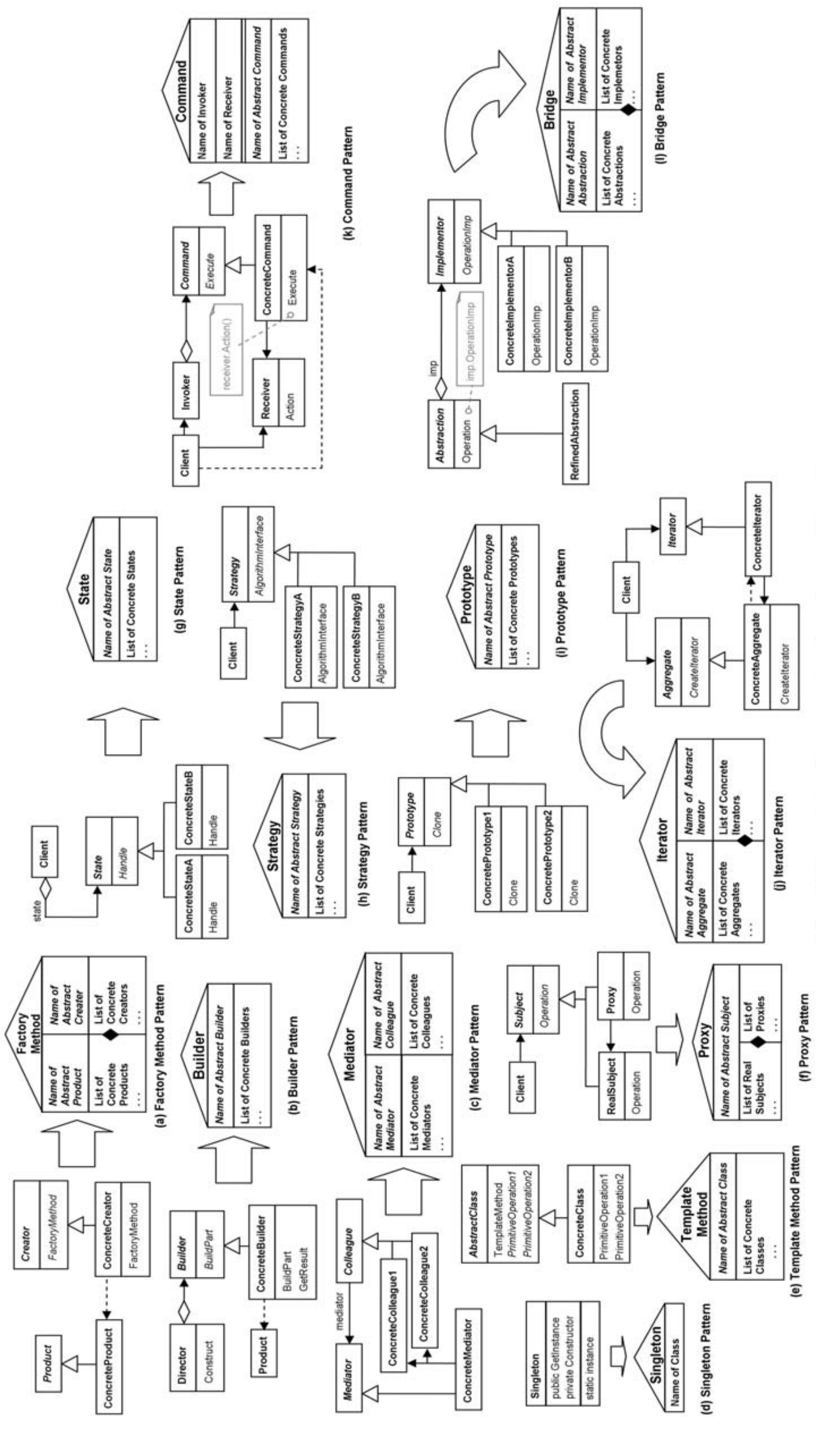

Fig. 8. Complete Conversion Catalogue (Part I)

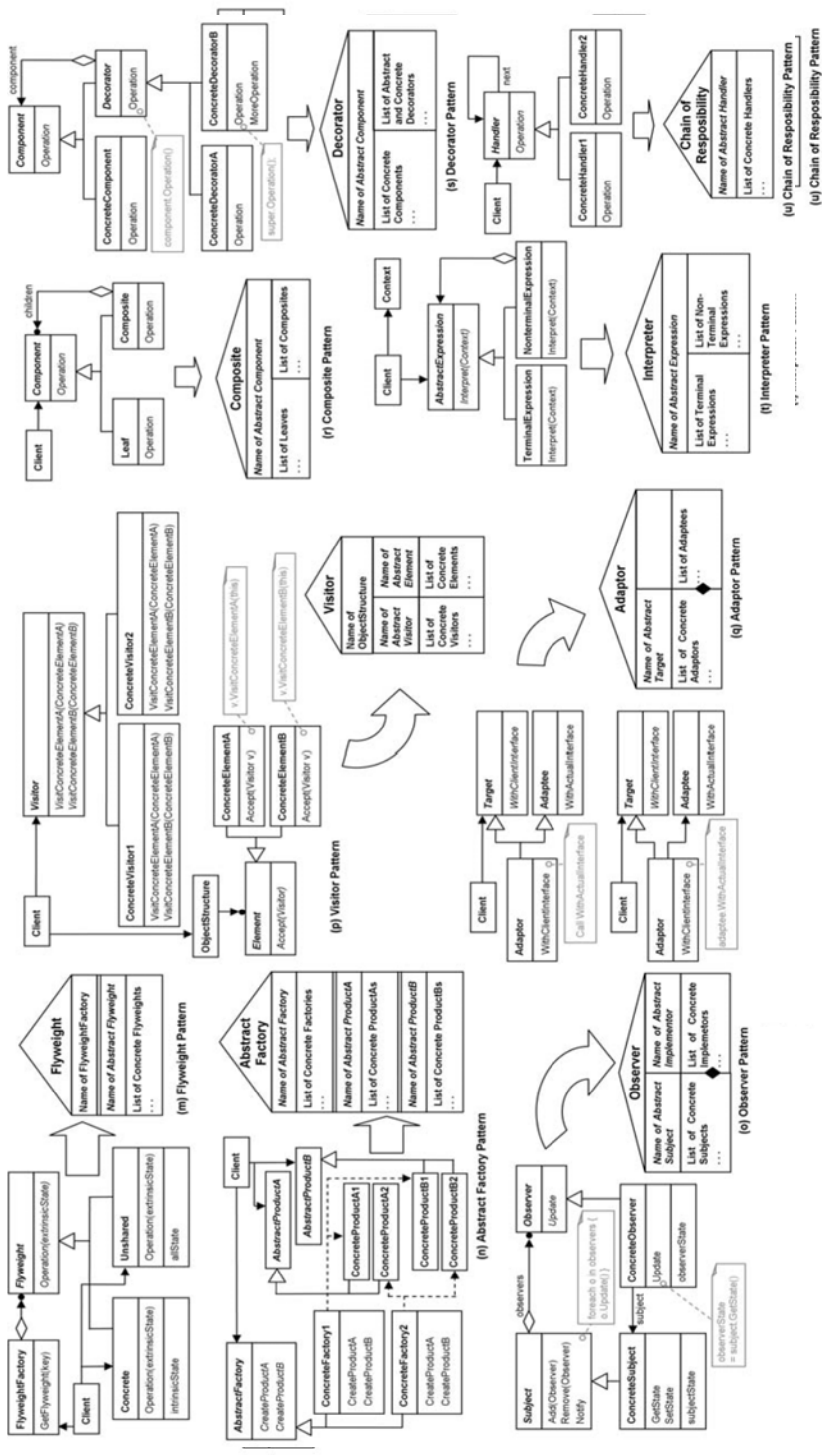
(m) Flyweight Pattern
Flyweight
Name of FlyweightFactory
Name of Abstract Flyweight
List of Concrete Flyweights
(n) Abstract Factory Pattern
Abstract Factory
Name of Abstract Factory
List of Concrete Factories
Name of Abstract ProductA
List of Concrete ProductAs
Name of Abstract ProductB
List of Concrete ProductBs
(o) Observer Pattern
Observer
Name of Abstract Subject
List of Concrete Subjects
Name of Abstract Implementor
List of Concrete Implementors
(p) Visitor Pattern
Visitor
Name of ObjectStructure
Name of Abstract Visitor
List of Concrete Visitors
Name of Abstract Element
List of Concrete Elements
(q) Adaptor Pattern
Adaptor
Name of Abstract Target
List of Concrete Adaptors
List of Adaptees
(r) Composite Pattern
Composite
Name of Abstract Component
List of Leaves
List of Composites
(t) Interpreter Pattern
Interpreter
Name of Abstract Expression
List of Terminal Expressions
List of Non-Terminal Expressions
(s) Decorator Pattern
Decorator
Name of Abstract Component
List of Concrete Components
List of Abstract and Concrete Decorators
(u) Chain of Resposibility Pattern
Chain of Resposibility
Name of Abstract Handler
List of Concrete Handlers

Fig. 9. Complete Conversion Catalogue (Part II)