Direct Execution of Design Patterns

M

Birol Aygün
Yeditepe University, Turkey

1. INTRODUCTION

Patterns occur in almost every aspect of life and scientific discipline. Considering the biological structure of living things, the most striking aspect is the fact that there is an external shell, which is in contact with the outside through a porous skin of some type, covering a very detailed internal structure with organs specific to certain functions. This pattern occurs in essentially all living things in many different species-specific formations. In science, all theories essentially try to explain or predict certain phenomena based on patterns observed; some expressed by mathematical formulae, which themselves are patterns following mathematical rules. Another area where patterns are vital is communication, where patterns are described by message formats, syntax and semantics of the language of the messages. Finally, designers and programmers use diagrams based on specific diagram patterns, such as flowcharts, UML and many others.

The subject of our study, namely architectural patterns and design patterns, was initiated as a follow-on to the work of Christopher Alexander, a construction architect, who invented these patterns to describe buildings and created a design pattern language (Alexander, 1977). Patterns have been used very widely to describe architectural styles, pre-fabricated housing, industrial products and processes. The term "architectural pattern" refers to patterns depicting overall structures of systems. The term "design pattern" refers to design of components of the architectural pattern. We shall sometimes use the term "patterns" to refer to both kinds of patterns.

When examine the history of programming languages, we see that initial symbolic languages contained instruction formats, which are essentially patterns, following the structure of CPU hardware commands, which are converted into machine language on a one-to-one basis by translators called assemblers. These were followed by macro-assemblers capable of translating named sets of instructions into machine code. Then came the so-called higher-level languages, with detailed syntax patterns, and semantic specifications. One characteristic of the latter is that instructions can contain other nested instructions inside them, creating loop patterns, flow control patterns and other patterns.

Thus, a programmer's job is essentially using the instruction patterns available in the language he is using to create instruction patterns which will perform the required functions. These instructions are designed to be as complementary to other instructions as possible so that one instruction can process the results of another. Programming language instructions can be thought of as micro-patterns which can be combined to create progressively higher-level patterns, which are the subject of our study.

2. BACKGROUND

A major exposition of software patterns was made in "Elements of Reusable Object-oriented Software" (Gamma, Helm, Johnson & Vlissides,. 1994) containing a pattern catalog with structured explanations and examples of usage of each type. This catalog also classified design patterns into three generic groups:

1. Creational Patterns,
2. Structural Patterns,
3. Behavioural Patterns.

Another useful paper in this area is (Buschmann, Meunie, Rohnert, Sommerland & Stal, 1996).

Efforts have been made to formalize and model pattern diagrams and descriptions to make them more rigorously definable which may be conceptually useful in our approach as well (Eden, 2011).

Software design patterns area has also been approached with a view towards "componentization." Origins of this approach may be traced to the now-classic

DOI: 10.4018/978-1-4666-5888-2.ch582

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

paper "*Trusted Components for Software Industry"* (B. Meyer et al., Eiffel Software" http://archive.eiffel.com/doc/manuals/technology/bmarticles/computer/trusted/page.html, accessed on 13/11/2013), emphasizing the inadequacy of commonly suggested approaches for developing reliable software and arguing that in order to be able to write reliable software, software developers need reliable components they can use. This work was followed by Karine Arnout, a doctoral student of B. Meyer, whose PhD thesis titled "From Patterns to Components**"** (Arnout, 2004) emphasized the need to make software design patterns reliable components. In her thesis Arnout defined a software component as follows:

... software component is a reusable module with the following supplementary properties:

- *It can be used by other modules (its "clients").*
- *The supplier of a component does not need to know who its clients are.*
- *Clients can use a component on the sole basis of its official information.*

Arnaout's thesis was followed by two papers on componentization: Factory design pattern (Pattern Componentization: The Factory Example, Arnaut,K., Meyer, B. 2006) and the Visitor design pattern (Componentization: The Visitor Example" Meyer, B., Arnaut, K. 2006) giving additional details on how componentization should be performed and the evaluation criteria a component should meet.

Another more recent and useful work on use of design patterns in software components is that of Petr Stepan (Stepan, P. 2011) emphasizing use of design patterns as explicit, rather than implicit, artifacts in software development.

An approach to increase actual code re-use through software components was proposed in "(Behavioural) Design Patterns as Composition Operators, Kung-Kiu Lau et al, 2010" suggesting the inclusion of software design pattern code in software repositories along with other software components. A published industrial implementation of a specific design pattern is Matilda, in which code was written using the Pipes and Filters architectural pattern (Wada et al., 2010).

Another source of work on usage of design patterns is Java Design Pattern Framework developers (http://java-source.net/open-source/j2ee-frameworks/dinamica-framework, accessed on 3/12/2013). It provides some examples of design patterns implemented in Java, such as the Dinamica framework which permits usage of the MVC pattern by application.

An important consideration in the software development life cycle is making sure that software meets requirements. Traditionally, requirements have been divided into functional requirements and non-functional requirements. Further, certain aspects of these requirements have been designated as "concerns," and the term "separation of concerns" has been used to highlight the need to address the concerns as independently as possible (Dijkstra, E.W. 1982). Work has been done to structure "connectors" between requirement statements so that they can be mapped to architectural and design patterns (Gross, D. & Yu, E. 2001).

3. PROPOSAL FOR A NEW APPROACH

If a programmer can find a resemblance between a given problem and a problem pattern he is familiar with, he tries to re-structure the problem to fit that pattern, which we can call the "deductive step." If he feels that there is a good fit, he tries to break the problem into subproblems and repeats the above process until he feels he can start to implement some of the resulting parts, which we can call the "inductive step." If the deductive step is not useful, he tries to find another pattern he is familiar with. If successful he repeats the above steps for those subproblems.

If he cannot find a good match between the problem and the problem patterns he is familiar with, he tries (i) a higher level solution pattern, such as divide-and-conquer, and repeatedly applying the above approach to each of the resulting parts recursively, or (ii) he picks one of the patterns he found so far hoping that he can inductively create a solution using his existing bag of solutions. If a suitable pattern cannot be found, a new solution may be developed and stored as a new pattern.

A new type of "virtual machine" called a Pattern Machine (PM) which can help users develop their programs using existing patterns more effectively is proposed. The PM will include not only patterns but a rich framework, outlined below, to provide as much advantage as possible to the user based on a detailed knowledge of patterns stored in the PM. We can think

8 more pages are available in the full version of this document, which may be purchased using the "Add to Cart" button on the publisher's webpage:
www.igi-global.com/chapter/direct-execution-of-design-patterns/113046

Related Content

NLS: A Reflection Support System for Increased Inter-Regional Security
V. Asproth, K. Ekker, S. C. Holmbergand A. Håkansson (2014). *International Journal of Information Technologies and Systems Approach (pp. 61-82).*
www.irma-international.org/article/nls/117868

Elitist-Mutated Multi-Objective Particle Swarm Optimization for Engineering Design
M. Janga Reddyand D. Nagesh Kumar (2015). *Encyclopedia of Information Science and Technology, Third Edition (pp. 3534-3545).*
www.irma-international.org/chapter/elitist-mutated-multi-objective-particle-swarm-optimization-for-engineering-design/112785

Metadata Standards in Digital Audio
Kimmy Szeto (2018). *Encyclopedia of Information Science and Technology, Fourth Edition (pp. 6447-6463).*
www.irma-international.org/chapter/metadata-standards-in-digital-audio/184341

Performance Appraisal
Chandra Sekhar Patro (2018). *Encyclopedia of Information Science and Technology, Fourth Edition (pp. 4337-4346).*
www.irma-international.org/chapter/performance-appraisal/184140

An Open and Service-Oriented Architecture to Support the Automation of Learning Scenarios
Ângels Rius, Francesc Santanach, Jordi Conesa, Magí Almiralland Elena García-Barriocanal (2011). *International Journal of Information Technologies and Systems Approach (pp. 38-52).*
www.irma-international.org/article/open-service-oriented-architecture-support/51367