

Teaching Design Patterns

Asher Sterkin

NDS Technologies Israel Ltd.,

P.O. Box 23012, Har Hotzvim, Jerusalem, Israel

Abstract: This paper presents an argument in favor of the systematic teaching of object-oriented design patterns in academic institutions and provides some recommendations for organizing the teaching process. It is intended for students, teachers, software engineers, architects, and managers.

Keywords: software engineering, object-oriented design, design patterns, education

1 Acknowledgements

This paper developed from preparations for and intensive discussions at the OOPSLA's 2003 "Killer Application for Teaching Design Patterns" workshop.

Special thanks to Michael Caspersen, Richard Rasala, Stephen Wong, D. X. Nguyen, Philip R. Ventura, Henrik Bærbak Christensen, Joseph Bergin, and Carl G. Alphonse for insightful ideas generated during the pre-workshop e-mail exchange, at the workshop's face-to-face meeting, and during the follow-up discussions at the OOPSLA 2003 conference.

This paper describes my own personal position on the subject and by no means reflects the consensus of the workshop. There were some significant discrepancies and a substantial amount of teamwork would be required to reach a commonly agreed-upon version.

2 Introduction

Why should we teach design patterns in academic institutions? During the OOPSLA 2003 workshop, "Killer Application for Teaching Design Patterns," Joe Berlin suggested, "Because it's a better way to think about objects, and objects are better way to think about software." At the time this seemed to be the consensus among the workshop participants. I tend to agree with this statement, but for a variety of historical and personal reasons. I sense a deeper connection between design patterns and the education process. To rephrase the Joe's statement I would say, "We should teach design patterns in academic institutions because design patterns are a better way for introducing abstractions in software, and abstractions are the only mental aid at our disposal for building really complex software systems."

It turns out that introducing abstractions through design patterns is an integral component of *pondering* [3]-an informal part of the thinking process, which precedes and controls the formal part called *reasoning*. As such, design patterns depart from the limited realm of pure technical tricks and now deserve closer attention from the academic establishment.

This paper builds an argument in favor of such treatment of design patterns and provides some recommendations for organizing the teaching process.

3 The Challenge

Developing a modern software system that meets the tough requirements for a rich feature set, usability, dependability, and flexibility, under severe price/performance limitations, is a challenging task and requires a systematic engineering approach and appropriate tools to cope with this challenge.

The software industry has, for years, been notorious for excessively high wages and for an extremely high demand for even inexperienced software engineers. This scenario was especially typical during the late nineties of the previous century. However, the software industry has also been considered a modern form of slavery, in which people may work for 12 or more hours a day, six days a week, for months or even years without having enough time for their family, personal, and social life. Suddenly the excessive wages and high demand started vanishing with the 2000 dot.com crash, and what was left was hard work, more hard work, even more hard work, a lot of frustration, and fierce competition.

This situation is not new and was very accurately predicted by E.W. Dijkstra as early as 1968: "Hardware is rushing ahead of our programming ability and unless something drastic happens the situation will only get worse and worse. For with more and more powerful machines becoming generally available society will be more ambitious in these applications

and will be demanding more from the poor programmer who finds his tasks in the field of tension between the things to be done and the available tools. ... we have let ourselves be lured into constructing elaborate mechanisms, the actual behavior of which has grown far beyond our mental grasp or even worse: the misbehavior of which is well beyond our control. ... Closer scrutiny reveals the current source of the trouble: viz. unstructured multitude and bigness, insufficiently organized complexity with its bastards such as Chaos, Unreliability, Unadaptability and the like" [1].

Nothing has changed significantly since then over the last decades. We probably have better programming languages and more powerful off-the-shelf packages such as operating systems, database management system, Web servers, and application servers, but we also have more complex tasks to be accomplished, more sophisticated and demanding users, and much tougher business limitations. The total balance is still is not in favor of a software developer who, without proper tools and a systematic approach, cannot cope with the constantly increasing complexity.

As experience has proven, many software developers are tempted to accomplish their goals without thinking as hard as they work. This approach has many perceived advantages, among which high visibility and appreciation by management should be considered the greatest. The point, however, is that in the long haul, only those who work smart can survive.

4 The Problem

Unfortunately, many computer science and software engineering courses are based mostly on historical reasons. Since the "C" programming courses have been around for about 20 years, they are traditionally taught first. Design patterns are typically taught last, if at all, due to their comparatively young (only 10 years!) age. Often design patterns are part of advanced courses. Consequently, students first learn how to create a lot of ugly code using obsolete techniques and then (though not always) exposed to design techniques that could help them write less code and reach a much better, more elegant solution. The damage caused by this kind of educational process can be irreversible.

Those of CS courses fresheners, who thrive in their profession, have typically learned about a clean code structure, object thinking, design patterns and the rest of the tools required nowadays for developing good software, not through university or college courses, but from books, friends, internet and expensive training classes being provided by commercial software consulting services. And this is a shame: there is no real substitution for a proper formal education.

This paper proposes a list of topics, which, I believe, should be systematically learned and experienced at CS and SE courses. It also contains some suggestions for how this education process could be organized.

5 About Complexity

When discussing complexity with respect to software development we must distinguish three different types of complexity [2]:

- 1) Computational complexity
- 2) Programming complexity
- 3) Intellectual complexity

By *computational complexity* we mean the amount of work (computation) the computer will need to perform when it is running a particular program. By *programming complexity* we mean the complexity of the structure of the text of the program, which performs this computation. By *intellectual complexity* we mean the intellectual that effort we, as programmers, have to exert in order to create a program that performs the required computation.

First, we should mention that apart from providing some input data we have very little direct control over the computation, which is performed solely by computer. The only way to have a significant impact over computation is to provide a corresponding program.

Second, there will always be a certain trade-off between these three types of complexity. We could create a simple program, thus expending very little programming and intellectual effort, but this program would probably work very slowly or would not handle properly particular corner cases and thus would lead to a *complex* or even *incorrect* computation. On the other hand, we could create a very compact text describing a very efficient program, but this would require substantial intellectual effort.

To deal with intellectual complexity we, as human beings, may apply three mental aids [2]:

- 1) Enumeration
- 2) Mathematical induction
- 3) Abstraction

5.1 Enumeration

By *enumeration* we mean an individual, case-by-case analysis performed in a sequence typically using conditional statements such as *if-then-else* and/or *switch-case*. A fundamental fact about enumeration is that it stops working when a large number of cases must be analyzed. In other words, our brain capacity is limited and does not scale very well with the amount of enumerative analysis to be performed. For that reason state-driven software implemented using the *if-then-else* or *switch-case* statements usually fails when the number of states and/or transitions crosses a certain (not very high: 7 ± 2 [26]) threshold. Here we are not talking just about psychology, philosophy or aesthetics, but rather about hard dollars. The vast majority of inconsistent user interface behavior (especially in consumer electronics) is caused by improper implementation of the corresponding state machine, which leads to user frustration and typically prevents 90% of the product features from ever being used. The same statement is applicable to network communication software, whereby improper implementation of the state machine leaves security holes that are quickly discovered and abused by hackers and network virus producers from all around the world.

Computers, on the other hand, can perform a case-by-case analysis of almost any size, provided they are programmed correctly. The latter could be quite a challenge.

5.2 Mathematical Induction

By *mathematical induction* (a.k.a. repetition) we mean performing a repetitive task, usually implemented using *while*, *for*, *do-while* loops and recursive procedures. When performing repetitive tasks we human beings tire very quickly and thus prefer to delegate this task to a computer and to prove, in some way, that the programmed iterative activity will work correctly under certain conditions.

As with mechanical performance of enumerative tasks, computers are superb at cracking numbers in loops. They only have to be programmed correctly. The single way to "program computers correctly" is to apply appropriate abstractions.

5.3 Abstraction

By *abstraction* we mean ignoring certain levels of details and extracting common characteristics of a group of entities. Thus, a class is an abstraction of a group of objects with a common set of attributes, relationships, and behavior. At a more fundamental level, the variable is an abstraction of a particular set of values of a certain type, while the function is an abstraction of a certain set of computations.

There is a huge difference between human beings and computers: computers cannot introduce abstractions. In fact, abstraction, is a unique human capability [27]. For that reason one has to be warned against so-called *anthropomorphic* approach to the software developments. Human beings do not work like computers, while computers cannot think. Sometimes the best software solution is achieved when we delegate to the computer a task that will never be performed properly by a human being.

6 On Thinking

When talking about the "thinking process" we have to distinguish between two different types of activities [3], namely *reasoning* and *pondering*.

By *reasoning* we mean "all manipulations that are formalized-or could readily be so-by techniques such as arithmetic, formula manipulation or symbolic logic." Among the mental aids mentioned above, *enumeration* and *mathematical induction* are always applied through reasoning, that is, the more or less mechanical application of a certain formal technique. As a way of thinking, reasoning has two advantages and one disadvantage. It has the advantages of being teachable and learnable and actually constitutes the bulk of any formal academic course. Reasoning is also formally provable in a sense that "as soon as it has been decided in sufficient detail, *what* has to be achieved ..., there is no question any more *how* to achieve it." The biggest disadvantage of reasoning is that "we are very good at doing modest amount of reasoning. When large amounts of it are required, however, we are powerless without mechanical aids." For that reason, we are always looking for ways to delegate tasks requiring massive reasoning to computers.

By *pondering* we mean a kind of "thinking that reduces that amount of reasoning." It also includes "the 'supervision' during the reasoning, the on-going 'efficiency' control." Among the mental aids mentioned above, we always apply *abstraction* in order to *ponder* the reasoning. Pondering also has advantages and disadvantages. Its biggest advantage is that it is the only mode of thinking that distinguishes between the feasible and the unfeasible. Without proper pondering we do not know how, when, or to what extent to apply this or another type of reasoning. The biggest disadvantage of pondering is that it is difficult to teach and learn. At a certain level this kind of teaching is always done "by imitation" [4], where the teacher *shows* the pupils how the teacher would solve a particular type of problem. However, the imitative approach is not

sufficient and we would like to find some way to capture this knowledge on paper. In software engineering, *design patterns* provide a system for the codification of successful abstractions applied to a wide class of recurring problems.

7 Design Patterns

Design patterns were initially introduced by Christopher Alexander [5] for solving similar (though not identical) problems in civil engineering. Later, this approach was adopted by the object-oriented programming community [6].

The whole process of applying design patterns is based on a *deja vu* type of recognition of *conflicting forces* working in a particular *context*. These forces are *balanced* through a particular *structure*, which ensures a certain type of collaborations (*events*) between participating elements. Applying a pattern results in a modified context, which many times brings on the scene new, potentially conflicting, forces thus preparing a basis for applying additional patterns. A set of related design patterns constitutes a *pattern language*.

Although applicability of that or another design pattern could be evaluated using purely logical reasoning, at its core the pattern-driven design process appeals more to the designer's intuition, experience, and aesthetics than to the designer's reason. In this respect, design patterns shall be treated as "some verbal handles, which are no more than an aid to memory" [4], and by no means as a complete, off-the-shelf design solution. Design patterns seldom exist in isolation and should always be adjusted and modified in order to coexist together.

Due to their generic nature, most existing design patterns introduce abstractions, which help to shape properly the solution space rather than the problem space. Although some domain-specific patterns do exist [7], introducing problem space abstractions is still very often based on a common sense domain analysis and, at least at this stage, is less suitable for codification.

Sometimes we distinguish *architectural* patterns [10] that regulate the coarse-grained structure of a software system.

8 "Pure" Object-Oriented vs. Generic and Generative Programming

Upon closer examination, it appears that design patterns achieve a desired effect through defining a proper abstraction for certain system *commonalities* [19], while providing enough flexibility for the system *variability points*. Pure Object-oriented programming languages such as Java and C# primarily support the implementation of system commonalities and variability points through encapsulation, inheritance, and polymorphism. As long as all variability points really exist in the run-time and/or the computer system resources are not limited with respect to the required computation this approach works just fine. If, however, the computer system resources, such as memory and CPU, are inadequate for the required computation and not all variability points really exist at the run-time, this pure OO approach turns to be wasteful. This problem is especially thorny for consumer electronics and even more so in a broadcast environment such as digital TV. The Moore's law, which "dictates a doubling computer power, or a halving its cost every eighteen months" [21], doesn't help any more due to severe price/performance limitations and consistently increasing expectations.

When variability points exist during compilation time, the generic programming approach, initially introduced with the C++ Standard Template Library (STL) and further refined with the C++ Standard Library [20] and the C++ Boost Library [22], turns out to be much more efficient. In generic programming we still use, to some extent, encapsulation and inheritance to reflect commonalities, but we use function overloading and templates to reflect variability points. To reduce multiple function calls overhead C++ in-line functions are extensively used.

Sometimes we do not mind if computer performs a certain task without applying any abstraction at all (for instance through performing the same operation in sequence several times rather than in a loop). For the sake of efficiency, we would even prefer this, but we would not want to write such a program manually. For that purpose the C++ Generative Programming [18], which utilizes unique C++ compiler features for automatic code generation, comes in very handy. Recently the most spectacular results were achieved with the C++ Boost Meta-Programming Library (MPL) [23], which encapsulates code generation primitives in a form of containers and algorithms similar to those of the C++ Standard Library.

9 Groups of Design Patterns

To overcome the complexity of a software system we are looking for a way to build our system from a *reasonable* number of *modules* of reasonable complexity communicating through reasonably *thin* interfaces. Emphasis on the word *reasonable* introduces one's subjective judgment as an integral part of the process, but nobody has found yet a way to avoid this except for probably recurring appeal to the 7 ± 2 magic number empirical rule [26].

This process of the system breakdown into modules is based on two fundamental qualities of *decoupling* and *cohesion*. By *decoupling* we understand the level of inter-module independence. By *cohesion* we understand the level of

concentration of particular system services within one module. In order to decouple the modules and keep them cohesive we apply abstraction. We could abstract a number of different things:

- 1) Behavior
- 2) Structure
- 3) Presentation

Each type of abstraction is covered by a separate group of design patterns as follows [6] (only some representatives from each group are provided):

- 1) Behavioral Patterns:
 - a) Command
 - b) Iterator
 - c) Observer
 - d) State
 - e) Strategy
- 2) Structural Patterns:
 - a) Adapter
 - b) Composite
 - c) Decorator
 - d) Facade
 - e) Proxy
- 3) Creational Patterns:
 - a) Abstract Factory
 - b) Builder
 - c) Prototype

10 User Story-Driven Top-Down Decomposition

Design patterns are not applied all at once, but rather in a sequential, one-by-one fashion. In order to master the design patterns it is very important to adopt an appropriate process, which facilitates such development.

Top-down decomposition is a method of developing software through a number of iterative steps, in order to increase developer productivity and improve software quality. This approach was introduced by E.W. Dijkstra [2] as a response to the so-called "software crisis" of the late 1960s.

When combined with popular Agile Software development practices [24] such as user stories [28], unit tests [29], and mock objects [30], the top-down decomposition could be briefly outlined as follows:

For each user story:

- 1) Specify an acceptance test; make it to fail
- 2) If required, address the project organization and version control issues
- 3) At each step:
 - a) Identify a smallest possible piece of the program, which would allow making a progress by either adding detail to the current level of abstraction or introducing a new one.
 - b) Specify a unit test using mocks for all other units regardless of whether they are already implemented or not; make unit test to fail
 - c) Provide a simplest possible solution that allows passing the unit test
 - d) Re-factor the code in order to eliminate duplications, fix bugs, improve code readability or/and improve performance
 - e) Repeat until the whole story is implemented
- 4) Integrate and test
- 5) If required, perform additional re-factoring to eliminate "code smells"
- 6) Integrate and test

The very purpose of introducing new abstractions at every step is to eliminate the program and/or computational complexity through restricting enumerative analysis to the absolute minimum and keeping the iterative tasks under the full control. As it appears, abstraction turns to be the only mental tool at our disposal to fight the complexity through keeping enumerative analysis and iterative processes under control.

To avoid possible confusion we have to treat “layers of abstraction” in a sense of function call tree rather than in a sense of “Layers” architectural pattern [10]. Relationship between the “Layers” architectural pattern and top-down decomposition is a complex topic I plan to cover in more details elsewhere in a separate article [30].

The user story driven top-down decomposition approach described above provides the following advantages:

- 1) High speed of development
- 2) Small number of errors
- 3) Early integration and testing
- 4) Early demonstration/deployment
- 5) Support for parallel development
- 6) 'Just enough improvements'
- 7) Strong feeling of 'flow' and accomplishment

At the end of every step we will get some working code even if it doesn't do very much. At first the program will indeed do almost nothing, but it will still be useful since it will provide us with a reasonable framework to make and implement decisions about coding convention, development environment, version control, and so on. In addition this approach from the very beginning delivers a (perhaps partially) working code, which is good in its own right as an interim milestone, but may have other uses as well (e.g., early integrations and demonstrations).

We should also note that top-down decomposition is primarily abstraction-driven rather than feature-driven. As long as we do not need particular features for the further refinement of the software structure we tend to ignore them in order to keep the software structure flexible. We always tend to build a "family of related programs" [2] rather than one particular program.

At some stage of the process, introducing an initial set of architectural and design patterns makes room for many smaller design patterns and it is quite easy to start addressing too many questions concurrently and to lose focus. If this happens, the development process will get out of control and will turn into the typical patch-on-patch chaotic movement that most software developers are so unfortunately familiar with. To proceed further requires discipline and concentration. The guiding question should always [2]: which decision can we delay no longer without endangering meaningful progress in the design of the system, while at the same time committing ourselves to as little as possible order to preserve flexibility of the overall structure. User stories and their priorities provide an excellent input for this decision making process.

Following the minimum commitment approach of top-down decomposition means that we should have enough courage to make partial design decisions and to be fully confident that we'll be able to cut off our losses quickly and to make appropriate corrections if something goes wrong.

The need for changes should not be surprising and/or frustrating since only “dead” software does not change. As long as the number of changes is under control and the change follows a well-defined procedure, changes are kindly welcome. Since the "one change always in one place" expectation is sometimes too naïve for complex software structures, the real goal should be to prevent chaotic, uncontrolled cascades of modifications caused by of ANY new change.

11 Teaching Design Patterns

Learning design patterns in isolation is similar to studying a foreign language with only a dictionary. Sometimes it is worthwhile when one wants to understand the meaning of a word more deeply or to become familiar with a new word. As those who speak one or more foreign languages know it very well, learning through sentences in real-life situations is a much more efficient method. This method facilitates building and maintaining a picture of the language wholeness and harmony from the very beginning.

Building a software system according to the top-down decomposition approach is similar: we apply patterns one by one in order to further refine the system structure. However we do not deal with patterns in isolation, but rather consider them as a whole. In other words we speak in this pattern language.

The goal of teaching the design pattern process should be to teach *pattern thinking* rather than a particular subset of the design pattern catalog. The design pattern catalog may vary over time [25] and it will take some time to stabilize this catalog. At any development, in each new area, a new set of design patterns must be discovered [11-16, 9].

12 Summary

In this short article I've summarized some fundamental concepts of the software engineering I would like to be in possession of every CS or SE course graduate. The list is far from to be complete and some other important elements as design principles [17] are yet to be included. Still, it hopefully does reflect certain level of formal education the software

industry desperately needs. There is a big gap between what has the software industry accumulated in a form of the “best practices” and what is actually studied in the academic institutions. We need to close this gap as soon as possible in order to make our software development business sustainable.

13 References

- [1] Dijkstra, E.W.: "On Useful Structuring," <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD227.PDF>, February 1968
- [2] Dijkstra, E.W.: "Notes on Structured Programming," <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>, April 1970
- [3] Dijkstra, E.W.: "On the Teaching Programming, i.e., on the Teaching of Thinking," <http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD473.PDF>, February 1975
- [4] Dijkstra, E.W.: "Craftsman or Scientist," <http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD480.PDF>, March 1975
- [5] Christopher Alexander: "The Timeless Way of Building," Oxford University Press, 1979
- [6] Gamma, E., et al.: "Design Patterns. Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995
- [7] Fowler M.: "Analysis Patterns: Reusable Object Models," Addison-Wesley, 1996
- [8] Fowler M.: "Patterns of Enterprise Application Architecture," Addison-Wesley, 2003
- [9] Hohpe G., Woolf B.: "Enterprise Integration Patterns," Addison-Wesley, 2003
- [10] Buschman, F., et al.: "Pattern-Oriented Software Architecture: A System of Patterns," John Wiley & Sons, 1996
- [11] Schmidt D., et al.: "Pattern-Oriented Software Architecture vol.2: Patterns for Concurrent and Networked Objects," John Wiley & Sons, 1999
- [12] Volter M., et al.: "Server Component Patterns," John Wiley & Sons, 2002
- [13] Borchers J.: "A Pattern Approach to Interaction Design," John Wiley & Sons, 1997
- [14] Noble J., Weir C.: "Small Memory Software: Patterns for Systems with Limited Memory," Addison-Wesley, 2001
- [15] Douglass, B.P.: "Real-Time Design Patterns," Addison-Wesley, 2003
- [16] Alexandrescu A.: "Modern C++ Design: Generic Programming and Patterns Applied," Addison-Wesley, 2001
- [17] Martin, Robert C.: "Agile Software Development, Principles, Patterns, and Practices," Prentice Hall, 2002
- [18] Czarnecki K., Eisenecker U.W.: "Generative Programming: Methods, Tools and Applications," Addison-Wesley, 2000
- [19] Cooplén, J.: "Multi-Paradigm Design for C++," Addison-Wesley, 1999
- [20] Dinkum C++ Library Reference, <http://www.dinkumware.com/manuals>
- [21] Gilder G.: "Telecosm. The World after Bandwidth Abundance," Touchstone, 2000
- [22] C++ Boost Library: <http://www.boost.org>
- [23] C++ Boost MPL Library: <http://www.boost.org/libs/mpl/doc/index.htm>
- [24] Kent B.: "Extreme Programming eXplained: Embrace Change," Addison-Wesley, 2000
- [25] Kevlin H., Buschmann F.: "Beyond the Gang of Four," OOPSLA 2003, Tutorial #23, ACM, 2003
- [26] Miller, G.A.: "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", The Psychological Review, 1956, vol. 63, pp. 81-97, available at <http://www.well.com/~smalin/miller.html>
- [27] Korzybsky A.: "Science and Sanity: An Introduction to Non-aristotelian Systems and General Semantics", Institute of General Semantics; 5th edition (April 1, 1995)
- [28] Cohn, M.: "User Stories Applied", Addison-Wesley, 2004
- [29] Kent Beck: "Test-Driven Development: By Example", Addison-Wesley, 2002
- [30] Asher Sterkin: "Layered Architecture Revised", to be presented at ESA'06 conference