

PROGRAMMING PRINCIPLES - INSTRUCTOR'S TOP LIST

Vladan Jovanovic, Georgia Southern University, vladan@georgiasouthern.edu
James Harris, Georgia Southern University, jkharris@georgiasouthern.edu
Richard Chambers, Georgia Southern University, rchamber@georgiasouthern.edu
Han Reichgelt, Georgia Southern University, han@georgiasouthern.edu
Ron MacKinnon, Georgia Southern University, rjmackin@georgiasouthern.edu
Sonny Butler, Georgia Southern University, esbutler@georgiasouthern.edu
Stevan Mrdalj, Eastern Michigan University, stevan.mrdalj@emich.edu
Daniel Shoemaker, University of Detroit Mercy, shoemadp@udmercy.edu

ABSTRACT

Experienced faculty tend to teach programming principles and practices as part of all programming courses on a purely ad hoc basis. In general, there is no agreement on a short list of generally accepted principles and practices as key points for students to learn, and such a list is also conspicuous by its absence in mainstream college textbooks used in “principles of programming” classes. In this paper we provide a starting list of programming principles that are applicable to all major programming languages. The authors used their teaching experience in programming and a broad set of textbooks in an attempt to assess current coverage and select a baseline set of principles aimed at assuring competency of our students as programmers. This paper presents results of an initial pilot survey administered with the aim of discovering an initial list of such principles.

Keywords: programming principles, teaching of programming

INTRODUCTION

Many introductory programming courses use the phrase “principles of programming” in their titles and/or course descriptions while actually focusing on a particular programming language and covering a surprisingly similar set of examples illustrating little more than basic syntax. It is almost never the case that such courses introduce students to a set of general programming principles, which might be illustrated, using the language being taught, throughout the textbook (1, 6, 9, and 12). Problem solving in these courses is typically illustrated with language specific steps rather than as a series of refinements from an abstract, language independent, solution to a concrete instantiation in a particular language with the application of specific programming principles shaping the concrete solution to improve its quality. Sample programs tend to be small programs that merely allow the student to master the basic syntax without mastering the language itself and usually ignore the issues often encountered in large scale applications. Even the more rigorous texts (e.g., 13) hardly venture beyond presenting concepts such as invariants, pre and post conditions, abstractions, information hiding and encapsulation.

In contrast, a number of professional programmer references (3, 4, 7, 10, 11, 14, 16) focus on principles and practices of programming in a broader context of program design, although there does appear to be some exclusion of specific principles due, one suspects, to the focus on a

specific programming paradigm or programming language. For example, object-oriented texts hardly ever mention the principles of cohesion and coupling.

As teachers concerned about instilling the craft of professional programming in our students, we decided to reexamine our own practices as well as the published practice of others in order to arrive at a list of generally accepted programming principles and practices that any professional programmer needed to be familiar with. We strongly believe that the current practice of introducing students to programming in the language specific manner typical of many of the introductory textbooks is likely to lead to graduates who are not sufficiently versed in the craft of programming. Introducing students to generally accepted programming principles and practices, on the other hand, is likely to lead to graduates with better programming skills.

In our view, useful principles can be categorized as: criteria for the quality of solutions (such as cohesion), general policies (such as applying the correct artifacts in visualization), and enduring rules (such as limiting access to objects). We tried to distinguish principles (rules, policies and criteria) from practices (standards, notations, methods, techniques, operational procedures, documentation forms etc.) realizing that performing a well-established practice using good principles produces a better product than one using poor principles. Principles that could easily translate into a 'test' for good quality were preferred. Some general principles and principles about principles, i.e. meta-principles, were mentioned and used but not elaborated upon.

We also had some interesting discussions regarding design patterns. The consensus arising from the discussions was that design patterns are not principles per se but are distilled experiences that provide a mechanism to transform tacit, specific architectural models into explicit knowledge that can be transferred to others. While both pattern and principle are time binding mechanisms and both are explicit knowledge artifacts, design patterns are more of a catalog of possible solutions whereas principles are concepts used to shape the source code produced regardless of the pattern used.

In order to focus on essentials, we arbitrarily decided to limit the number of key principles to ten. We judged the most important to be those principles which substantively help achieve good quality programs while still being teachable, i.e. concrete and usable principles whose utility can be demonstrated as opposed to more abstract axioms. In this paper, we describe the process by which we reached a consensus and the list of principles that we came up with. It should be stressed from the outset that it is not our intention to give a definitive list of programming principles and practices. Rather, the work reported here is merely intended as a pilot.

APPROACH

While most preparatory work was done by the lead author, the coauthors selected the top ten principles collectively. Since this study is an exploratory pilot study, empirical collection from experienced faculty members was considered appropriate.

We started our quest for general principles and practices with a survey of the commonly used college textbooks for "principles of programming" classes. Unfortunately, we were unable to

find evidence of the explicit use and exposition of principles in the literally hundreds of introductory textbooks we have reviewed and/or used during the course of their teaching careers. We also reviewed a number of course descriptions from course catalogs in Computer Science, Information Systems, Information Technology, and Software Engineering programs. While there are differences in syllabus concentration, we concluded there is no common set of programming principles explicitly treated by the courses or at least none were mentioned as a part of the course descriptions. Consequently, the lead author established an initial short list of ten common programming principles partly based on his own experience and a knowledge base captured in professional books (3, 4, 5, 9, 10, 14, 16).

In the first phase of the study, participants were asked to add any principles they regarded as essential to the initial list of ten compiled by the first author. After some sorting, this resulted in a longer list of 30; this list is presented in the appendix as Table 4.

This list of 30 principles was then re-distributed to the participants who were given the following instructions:

Rank from 1 (the most important) the first 10 principles from any of the lists shown on the back; if you want to add a principle and its rating please do so or add a comment using blank column at right. Indicate definition and possibly a reference for any principle you want added.

Participants were also asked to indicate whether they consider a particular principle to be a general principle, an object-oriented specific principle or a principle that pertained to more traditional programming languages.

RESULTS

Six of the eight faculty to which the questionnaire was sent responded in time for their responses to be included in the result. The actual responses were tabulated and a principle was listed only if two or more votes were given. The results are given in table 1, which was compiled based on the priorities received from the faculty. Note that the general category item, "Use established patterns and solutions", mentioned by three individuals consolidates a similarly worded addition which was not in the original list in Table 4.

Table 1- (Preliminary) list of principles selected by faculty

Rank	Principle	From a category	Number of votes (n = 6)
1	High Cohesion	General	6
2	Low Coupling	General	6
3	Test early, test often	General	4
4	Design with interfaces not inheritances	Object-oriented	4
5	Use established patterns and solutions	General	3
6	Dependency Inversion	General	3
7	Do not optimize code before it works	Traditional	3
8	Top Down- Stepwise refinement	Traditional	3
9	Design with composition	Object-oriented	3
10	Acyclic Dependency	Object-oriented	3

In the interest of collecting some preliminary data for further investigation we administered the same questionnaire to a group of senior undergraduate students enrolled in a capstone project. They were given the same questionnaire independently (not being aware of the voting among professors). In a population of 22 they returned 18 valid ballots and voted as in the Table-2 below.

Table -2 Results of vote by senior UG students

Rank	Principle	From a category	Number of votes (n = 18)
1	Test early test often.	General	18
2	Try to solve a general case	General	13
3	Build a little, test a little	Traditional	12
4	Open-Close.	Object-oriented	11
5	Design with Composition	Object-oriented	10
6	There is always a better way (create and evaluate different alternatives)	General	10
7	High Cohesion	General	10
8*	Use layering	General	10
9	Program structure follows output/input structure (data driven design)	Traditional	8
10	Top Down - Step wise refinement	Traditional	8

Note: Use of layering was mentioned three times by the faculty respondents, but in a quick post mortem review reconsidered and excluded. Since Dijkstra's THE paper, 'layering' is considered more of a context for programming (at the top level of architecture) than strictly a detail oriented programming principle.

It is fair to say that there is not a strong resemblance to the rankings of the faculty. There appears to be a reliance on a “quick turn over, write/test” cycle by the students whereas the faculty appear to rely more on a “write thoughtfully and then test” model.

DISCUSSION

There is nothing definitive about the data and the work we present in this paper. Still it seems plausible that a number of research questions will emerge once we develop a set of prominent principles after collecting a larger sample of responses from a more diversified group of faculty. We are also considering a modified and more refined survey instrument. This instrument would try to capture a consensus as to the actual definitions of the principles with specific examples suitable for classroom delivery illustrating their use and suggestions as to appropriate delivery methods. These results are considered preliminary and the ranking tables are presented only as an example intended to provoke response from the larger community of those teaching programming.

One useful step after preliminary analysis of the survey results, was to create a list of ‘meta’ principles (Table 3). It was helpful to reflect on meta-principles as a brainstorming exercise in separating general principles from principles that are possibly too general.

Table 3- List of Meta-Principles

1. Simplify, Generalize, Abstract, and separate concerns
2. There is no silver bullet, Essential Difficulty vs. Accidental – nature can not be fooled
3. No rules without exceptions
4. Use Modeling and appropriate visualization diagrams as a common language
5. Use working solutions, frameworks, patterns, idioms, primers and exemplars, whenever they exists
6. Do not reinvent methodological support but use proven Standards, Guidelines, Methods, Techniques, Notations, Conventions, Documentation, Forms, and other forms of best practices when available.
7. Search for a Balance (for example continuous improvement vs. disruptive technologies)
8. Experiment, do not put all eggs in one basket, evaluate alternatives and trade-offs
9. Pay attention to aesthetics.
10. Study principles and their effectiveness in accessible practice, internalize principles and heuristics and in practice use more specific principles like those from the list in Table 4.

CONCLUSION

The initial objective of this paper and work was to empirically identify a useful set of programming principles to be taught in a “principles of programming” class. The main focus of work now shifts to the study of commonly recognized principles while further refining the detailed lists, minimizing redundancy among principles, and obtaining statistically significant samples of opinions from a larger pool of interested practitioners.

More refinement is also needed to define principles and provide good sources for those readers who want to explore them in detail. Furthermore, some guidance is yet to emerge for those readers willing to apply selected principles in their teaching of programming. We are, for instance, interested in other alternatives to the standard writing of small or toy programs for

principles classes and are wondering if specific maintenance actions on the source code of an existing, larger application may be more preferable.

We hope to create a body of knowledge of language independent principles. This body of knowledge will relate the principles to each other and demonstrate the relationship of principles with design patterns and other programming solutions less dependent on language syntax than programming idioms.

Key ideas like Information Hiding, Layering, etc. can be treated as principles but we considered them to be of insufficient help to novices needing fairly concrete examples. These key ideas also used to explain goodness in other principles. Some of us disagreed with the exclusions as well as with naming for some of the principles. Still we remain open to continue to work together and present a more refined list at some later point. We would appreciate feedback on any aspect of using principles to teach programming or our list so that we may improve it. Please direct your comments to the first author's e-mail address.

REFERENCES

1. Adams, J. (1998). An Introduction to Computing with C++, Prentice Hall.
2. Astels, D. (2003). Test-driven development, Pearson Publishing.
3. Coad, P., Mayfield, M. (1999). Java Design 2ed., Yourdon Press.
4. Davies, A. (1995). 201 Principles of Software Development, McGraw Hill.
5. Enders, A., Rombach, D. (2003). A Handbook of Software and Systems Engineering Pearson.
6. Friedman, F., Kofman, E. (2004). Problem Solving, Abstraction, and Design Using C++ Addison Wesley.
7. Horstmann, C. (2004). Object-Oriented Design & Patterns, J. Wiley.
8. Hunt, A., Thomas, D. (2000). The Pragmatic Programmer, Addison Wesley.
9. Jia, X. (2002). Object-Oriented Software Development Using Java, Addison Wesley.
10. Kernighan, B., Pike, R. (1999). The Practice of Programming, Addison Wesley.
11. Larman, G. (2002). Applying UML and Patterns 2ed., Prentice Hall.
12. Liang, D. (2004). Java Programming with JBuilder 3rd edition, Prentice Hall.
13. Liskov, B., Guttag, J. (2001). Program Development in Java, Addison Wesley.
14. Martin, R. (2002). Agile Software Development, Prentice Hall.
15. Muldner, T. (2002). C++ Programming with Design Patterns Revealed, Addison Wesley.
16. Page-Jones, M. (1988). The Practical Guide to Structured Systems Design 2ed., Prentice Hall.
17. Wampler, B. (2002). The Essence of Object-Oriented Programming with Java and UML Addison Wesley.

APPENDIX

Table 4. Initial List of Principles

General Principles:

1. High Cohesion
2. Low Coupling
3. Try to solve a general case (if you can first).
4. Use layering
5. Test early test often.
6. In order to learn, prepare simple programs (and vary them) as experiments to learn syntax, but prepare simple tools and use them to make real programs.
7. People and time are not interchangeable (adding people late prolongs projects).
8. Design test cases as requirements before code design
9. There is always a better way (create and evaluate different alternatives).
10. Dependency Inversion
Abstractions should not depend on details- details should depend on abstractions.

Traditional (procedural) Principles

1. Limited Scope of control (also known as 7 ± 2)
2. Program structure follows output/input structure (data driven design)
3. Top Down - Stepwise refinement
4. Build a little, test a little
(Write 5-10 lines, one method/procedure at a time) inspect, compile, test-run).
5. Data locality (data used together travel together as records, physical pages...)
6. Unit test before integration
7. Separate interface (signature) from implementation
8. Parameterize interfaces and use virtual drivers
9. Localize responsibility for error processing
10. Do not optimize code before it works.

O-O Principles

1. Open-Close. Classes should be open for extension but closed for modification.
2. Design with Composition not implementation inheritance (also known as Delegation)
3. Design with interfaces not multiple inheritances
4. Liskov's Substitution. Subclasses should be really substitutable for their base classes.
5. Interface Segregation (also known as minimal interfaces)
Clients should not depend upon methods they do not use, several small interfaces are better than a single general one .
6. Law of Demeter
For an operation O on a class C, only operations on the following objects should be called: itself, its parameters, object it creates, or its contained instance objects.
7. Acyclic Dependency. Dependencies between packages must form no cycles.
8. Release- Reuse. The granule of reuse is the granule of release.
9. Single Responsibility. Class should have only one reason to change
10. Stable Dependencies. Depend in direction of stability, as stable items are more resilient and more difficult to change.