Jeff Mather
RSEG-109
Final Project
2 December 2008

Surveying Quality in Object-Oriented Design

The major goals of using object-oriented design are to facilitate the maintenance and extension of software systems by reducing the complexity of software at the class and system level[1]. Successful OO designs are resilient to change, largely because they manage interclass dependencies. In such a system, changes to one part of the system are localized and do not cause a chain of modifications through the system.

The major techniques that OO designers have at their disposal are abstraction, encapsulation, inheritance, polymorphism, and composition. But how does an engineer apply these techniques to create a good design? What are the main ways that sets of classes can be structured and interact to maximize the chance of a successful design? Over the last twenty years, numerous OO practitioners have developed a mature set of rules-of-thumb and best practices to use when constructing and evaluating OO design.

A Short History of OO Heuristics

In 1988, Ralph Johnson and Brian Foote published a paper containing one of the first collections of object-oriented design heuristics that can be used to evaluate the quality of an object-oriented system. They paid particular attention to the different trade-offs of code reuse when using inheritance (white box reuse) versus composition (black box reuse), and coined the aphorism to "favor composition over inheritance."[2] They presented roughly a dozen design maxims[3]:

- Recursion introduction
- Eliminate case analysis
- Reduce the number of arguments
- Reduce the size of methods
- Class hierarchies should be deep and narrow
- The top of the class hierarchy should be abstract
- Minimize accesses to variables
- Subclasses should be specializations
- Split large classes
- Factor implementation differences into subcomponents

---

[1] McConnell, Steve. *Code Complete*. 2nd ed. Redmond, WA: Microsoft Press. 2004. p. 78.
[2] "The Mutable Compendium of Object Oriented Wisdom: Johnson and Foote 1988." (https://wiki3.cosc.canterbury.ac.nz/index.php/Johnson_and_Foote_1988)
[3] "The Mutable Compendium of Object Oriented Wisdom: Design maxims." https://wiki3.cosc.canterbury.ac.nz/index.php/Design_maxims

- Separate methods that do not communicate
- Send messages to components instead of to self
- Reduce implicit parameter passing

Ken Auer's 1995 paper "Reusability through Self-encapsulation" continued the theme of improving OO design via a new pattern language for encapsulation. One of the most enduring results of his paper is the viewpoint that classes should interact with attributes primarily through accessor and mutator methods rather than directly through public or protected visibility or friendship[4].

A year later Arthur Riel's influential book *Object-Oriented Design Heuristics* presented more than sixty "rules of thumb" to employ when building OO systems[5]. The rules concerned many of the structural and behavioral aspects of OO modeling: classes and objects, packages of functionality, instantiation, associations between classes, single and multiple inheritance, etc.

The extremely successful book *Refactoring: Improving the Design of Existing Code* by Martin Fowler, *et al.*, turned the discussion of good OO design on its head by considering bad OO designs and showing a collection of patterns to rearchitect "smelly" code[6]. Many of the patterns that Fowler presents are reworkings of the earlier research into heuristics by Johnson and Foote, Auer, and Riel. For example, the "Replace Parameter with Method" refactoring pattern reflects Johnson and Foote's maxim to "reduce the number of arguments."

It's perhaps significant that *Refactoring* was published at the end of the first full decade of wide-spread OO adoption at a time when there was a significant amount of "technical debt" in existing software projects and when the new "Agile" methodologies – such as Extreme Programming (XP) – were being created around techniques of quick construction followed by redesign[7]. There were enough poorly designed OO systems that best practices needed to be complemented with recipes for fixing what existed.

Roughly contemporaneous with these developments, Robert Martin and his Object Mentor group published several papers on OO design, largely summarizing various

---

[4] "The Mutable Compendium of Object Oriented Wisdom: Ken Auer 1995." https://wiki3.cosc.canterbury.ac.nz/index.php/Ken_Auer_1995

[5] Riel, Arthur J. *Object-Oriented Design Heuristics*. Boston, MA: Addison-Wesley. 1996.

[6] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley. 1999.

[7] Fowler credits Ward Cunningham with creating the term "technical debt." "In this metaphor, doing things the quick and dirty way sets us up with a technical debt, which is similar to a financial debt. Like a financial debt, the technical debt incurs interest payments, which come in the form of the extra effort that we have to do in future development because of the quick and dirty design choice." – (http://martinfowler.com/bliki/TechnicalDebt.html)

aspects of other people's research. Key among these was his 2000 paper "Design Principles and Design Patterns," which I will explore in more detail later.

Steve McConnell discussed many general OO principles in the 2004 edition of his book *Code Complete*, which surveys best practices for software design and construction. Several chapters of this large book chock treat OO design principles. In particular, McConnell focuses on the desirable characteristics of a design; and he devotes several sections to encapsulation and abstraction, summarizing much of Riel's heuristics.

McConnell's Best Practices

In *Code Complete* McConnell presents an extensive catalogue of desirable traits in OO software design, many of which are object-oriented restatements of more general characteristics of software quality.[8] Designs should be simple – eschewing excessive cleverness – and easy to maintain. Systems should have loosely coupled parts, which are built for extensibility and reusability. Classes should have a large number of classes that use a particular class ("high fan-in") but should rely on very few classes themselves ("low fan-out"). The system should build portability into it and have no extra parts. When organizing the design, classes should be partitioned into different strata or tiers, to make it easier to examine the design from multiple levels. Systems should also be implemented using "standard techniques" instead of "exotic pieces" that might be hard for maintainers to understand.

McConnell devotes several pages to fundamental OO concepts. The system should have low-, mid- and high-level classes, and the abstractions should be consistent within the various strata of a system. Classes should not expose their implementations, either by making data members public or by providing methods that suggest how the class solves construction problems. A system or layer should not excessively distribute information between classes, since that forces a greater degree of coupling between classes. And classes should be strongly cohesive; the routines and data in a class should fulfill a central purpose.

McConnell suggests some ways of determining whether encapsulation is done correctly:

- Are areas that are likely to change separated and isolated in their own classes?
- Do classes minimize (or, better yet, prohibit) access to their internal structure?
- Are implementation details hidden from the interface specification?
- If the code is written in C++ does it avoid "friendship" between classes?

McConnell also suggests some additional guidelines:

- Design for testability.
- Use design patterns explicitly when the ideas appear in the design.
- Favor composition over inheritance, since inheritance is less flexible.
- Be suspicious of classes with only one instance or only one subclass.

[8] Most of the discussion in this section comes from chapters five and six in McConnell.

- Avoid deep inheritance trees.
- Prefer polymorphism to explicit type comparisons in code.
- Make all data private, not protected.
- Minimize the number of class routines, the number of other classes and routines a class uses, and the amount two classes collaborate.

He circles back around to the main point of OO design by advocating that if classes don't reduce, isolate or hide complexity, they shouldn't exist.

Riel's Heuristics

As previously mentioned, McConnell employs and extends Riel's sixty-one design heuristics. Riel acknowledges that these aren't black-and-white rules, since we was trying to avoid "religious wars" like that caused by Dykstra's article "The Use of goto Considered Harmful."[9] In general, though, these heuristics present good choices and can be considered rules.

Several of the heuristics appeared above in the discussion of *Code Complete*, but many more appear in *Object-Oriented Design Heuristics*.[10] Here is a small sampling of the some of the important ones that have not yet been discussed, along with Riel's own numbering:

- Classes should embody exactly one key abstraction. (2.8)
- Keep related data and behavior in one place. (2.9)
- Don't confuse classes with the roles they play. (2.11)
- Look for partitioning of behavior and/or data within a class. The presence of non-communicating member indicates that a class is likely too big. (3.4)
- Most of the methods of a class should be using most of he class's data members. (4.6)
- Classes must never know (or care to know) who contains or uses them. (4.13)
- Inheritance is only used to model specialization. (5.1)
- Do not model the dynamic semantics of a class through the use of inheritance. (5.14) Use composition instead. (5.18)
- When you have a choice between containment and association, choose containment. (7.1)

Martin's Design Principles

---

[9] From the Amazon.com product description: http://www.amazon.com/Object-Oriented-Design-Heuristics-Arthur-Riel/dp/020163385X/ (Retrieved 26 November 2008)

[10] For summaries of all 61 heuristics see https://wiki3.cosc.canterbury.ac.nz/index.php/Riel's_heuristics, http://lcm.csa.iisc.ernet.in/soft_arch/OO_Design_Heuristic.htm, or http://www.cs.grinnell.edu/~rebelsky/Courses/CS223/2004F/Handouts/riel-heuristics.html

If McConnell gave the general rules of what you should look for (or avoid) in good design, Martin demonstrates the "why" behind the "what." For Martin, the cardinal rule of good object-oriented design is the minimization and management of dependencies. "Changes that introduce new and unplanned-for dependencies" cause designs to rot.[11] Martin presents eleven principles drawn from academic literature to insulate classes and packages from rigidity, fragility, immobility, and "viscosity" – all of which represent dependencies that might make a system's hard to maintain or change. Five of these concern object-oriented class design. (The remaining six principles concern package design.)

- The Single Responsibility Principle (SRP)[12]
- The Open Closed Principle (OCP)
- The Liskov Substitution Principle (LSP)
- The Dependency Inversion Principle (DIP)
- The Interface Segregation Principle (ISP)

The Single Responsibility principle states, "there should never be more than one reason for a class to change." If a class has multiple "axes of change" – Martin gives the example of a modem class that's responsible for managing connections **and** transmitting data – then the class should be refactored into separate classes, each of which is responsible for one or the other (but not both). This is essentially a restatement of the design goal of maximizing cohesion within a class.

The Open Closed Principle – "a module should be open for extension and closed for modification" – forms the basis of modern, patterns-oriented design; and Martin considers it to be the most important OO design principle. The OCP holds that a system design should contain abstract base classes to which additional behavior can be added by creating new concrete subclasses, which the various consumers of the functionality use through polymorphism. In this way, the client's code does not have to change very much (if at all) when new functionality is provided. The OCP dampens the ripples of change that can break a system. Such changes are usually necessary when there are hard-coded if/elseif/else or switch/case constructs to examine type or when one class associates directly with concrete subclasses.

In order to move away from those code constructs, the Gang of Four design patterns book is replete with applications of the OCP. Almost every pattern (it seems) employs the OCP as a fundamental part of its construction. The Strategy, State, Factory Method, Builder, Prototype, and Adapter patterns – just to name a few – use abstract base classes with concrete subclasses to perform clients' requests polymorphically.[13]

---

[11] Martin, Robert. "Design Principles and Design Patterns." 2000. p. 4. (http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf)

[12] The Single Responsibility Principle was published in a separate paper from 2002: "SRP: The Single Responsibility Principle." (http://objectmentor.com/resources/articles/srp.pdf).

[13] Gamma, Erich, *et al*. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley. 1994.

In order for the OCP to succeed fully, subclasses should be completely substitutable for base classes, according to the Liskov Substitution Principle. If a subclass violates this strong version of the "IS-A" rule of inheritance, it's possible for a class to be given a derived object but be unable to perform one of the operations in the base class, or (probably worse) the subclass may succeed in calling the base class method but be left in an inconsistent or incorrect state.

One consequence of the LSP is that inheritance can be restated as a principle of contracts. The preconditions of the methods of the subclass must not be stronger than the preconditions of the base class, and the post-conditions must not be weaker. Basically, a subclass should "expect no more and provide no less" than its base class.[14]

Martin presents the Dependency Inversion Principle as the way to realize the benefits of the OCP. Designers and programmers should depend "upon interfaces or abstract functions and classes, rather than upon concrete functions."[15] In general, every concrete thing in a design represents something that is difficult to change without a introducing a cascade of other changes. By adding an extra layer of abstraction between the client of some functionality and the actual functionality itself, the engineer can insert new functionality at some later stage, extending an interface without modifying it.

In a principle that prefigures his 2002 Single Responsibility Pattern, Martin presents the Interface Segregation Principle. This principle states, "if you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each client and multiply inherit them into the class." As with the SRP, this principle aims to improve the cohesion of classes by trimming the size of interfaces, so that classes represent a narrow window of functionality. "The ISP acknowledges that there are objects that require non-cohesive interfaces; however it suggests that clients should not know about them as a single class. Instead, clients should know about abstract base classes that have cohesive interfaces."[16] Designers are advised to add new interfaces to existing objects rather than changing existing interfaces. (The Decorator design pattern seems to be an example of adding interface to an object at runtime in a similar, albeit different, way that incorporates the spirit of the ISP.)

The Edge of the Map

The three authors explored in this paper aim to bring high-quality OO design out of academic journals, professional conferences and the classroom in order to put them into widespread use by engineers who actually design, construct and maintain software. McConnell's *Code Complete* presented a wide-ranging survey of many high-quality software best practices, only some of which explicitly dealt with OO programming. Riel proposed easy-to-digest heuristics for evaluating OO designs, covering much the same

---

[14] Martin, *Principles*, p. 12.
[15] Martin, *Principles*, p. 13.
[16] Martin, Robert. "The Interface Segregation Principle." 1996. (http://objectmentor.com/resources/articles/isp.pdf).

OO ground that McConnell would later tread but with significantly more detail. And Martin examines a very deep, narrow slice of OO design, going into quite a bit of theoretical and practical detail on a comparatively small set of topics.

Each of these authors has contributed to the popularization of techniques that should be employed in creating and evaluating OO designs. They are not the only practitioners who have ventured into this arena, of course. In particular, the design patterns community aims to create general-purpose solutions that reuse proven, high-quality designs. Several of these patterns (such as the twenty-three patterns in the Gang of Four's seminal work) concern OO system design at the class or package level. But many pattern languages simply exist within the context of OO systems as they solve higher-level problems. (Enterprise design patterns are object-oriented, for example, but they are more concerned with the interaction of objects and classes in different tiers of a distributed business application.)

While patterns wrap up designs that solve familiar problems using high-quality, object-oriented techniques, the elements of OO style are not completely fixed and probably never will be. The number and scope of those techniques keeps growing at the edges of practice. Some of these new techniques are language-dependent: In languages with exception-handling (like Java and C++), classes should handle their own errors and then throw exceptions to communicate problems. Others are not: Functions should return an interface to something, rather than the thing itself.[17]

Finally, the academic community continues to contribute to the growing number of best practices. Consider the "Law of Demeter," which McConnell discusses very briefly (like many of the principles he presents). In a nutshell, this "law" says that programmers should not reach through objects, even if they're using public methods of another object. For example, one should not write code like this:

```
objectA.aMethodReturningObjectX().aMethodOfX();
```

Doing so creates a semantic dependency between two classes, which is (almost certainly) not apparent to the system maintainer and (occasionally) not even visible to the compiler.

Larman presents the rule in the following way.[18] Objects should only send messages to these objects:

1. The "this" object.
2. A parameter of the method.

---

[17] Both of these examples are from Nathan A. Good's "Build seven good object-oriented habits in PHP." 2008. (http://www.ibm.com/developerworks/opensource/library/os-php-7oohabits/index.html).

[18] Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3rd ed. Upper Saddle River, NJ: Prentice Hall PTR. 2005. p. 430. For a very brief discussion, see McConnell *Code Complete*, p. 150.

3. An attribute of "this."
4. An element of a collection, which is an attribute of "this."
5. An object created within the method.

Hopefully, the kinship of the Law of Demeter to the some of the principles presented earlier – such as minimizing coupling between classes – is obvious.

In conclusion, there are numerous rules and suggestions for improving object-oriented designs. Some have gone so far as attempting to quantify the quality of OO systems based on these heuristics and maxims. Until such metrics are completely automatic, they're probably overkill. Fortunately, learning and applying the principles surveyed in this paper should produce immediate results.