Implementation Patterns

Kent Beck

Preface

- “it is mostly concerned with smaller scale decisions, the kind programmers make many times a day”
- “Actually this book is built on a rather fragile premise: that good code matters. I have seen too much ugly code make too much money to believe that quality of code is either necessary or sufficient for commercial success or doesn't provide control over the future.”
- “In the end, then, this book is about responsibility. As a programmer you have been given time, talent, money, and opportunity. What will you do to make responsible use of these gifts?”

Introduction

- “The goal of this book is to help you communicate your intentions through your code.”
- “Throughout, the book is focused on programming techniques that enhance communication.”

Chapter 2 – Patterns

- “Many decisions in programming are unique. How you approach programming a web site will be quite different from how you approach building a pacemaker. However, when decisions become more and more purely technical, a sense of familiarity sets in. ”

- “The constraints or forces listed above affect how every loop in a program is written”
  - easy to read, easy to write, easy to verify, easy to modify and efficient
  - “The forces recur predictably, which is one sense in which a pattern is a pattern: it is a pattern of forces.”
- “No one set of patterns will work in all programming situations”
- “Patterns work best as aids to human decision making”
- “Each pattern bundles a common problem of programming with a discussion of the factors affecting that problem and provides concrete advice about how to quickly create a satisfactory solution”

Chapter 3 – A Theory of Programming

“These three elements – values, principles, and patterns – form a balanced expression of a style of development. The patterns describe what to do. The values provide motivation. The principles help motive of action.”

“Different values and different principles will lead to different styles”

Values

“The values are universal overarching themes of programming.”

- Communication, simplicity and flexibility color every decision I make while programming.

Communication

- “Code communicates well when a reader can understand it, modify it, or use it.”
- “There is a sound economic basis for focusing on communication while programming. The majority of the cost of software is incurred after the software has been first deployed. Thinking about my experience of modifying code, I see that I spend much more time reading the existing code tha I do writing new code. If I want to make my code cheap, therefore, I should make it easy to read”

Simplicity

- “Eliminating excess complexity enables those reading, using, and modifying programs to understand them more quickly.”
- “Just as good prose is written with an audience in mind, so good programs are written with an audience in mind.”
- “Apply simplicity at all levels. Format code so no code can be deleted without losing information. Design with no extraneous elements. Challenge requirements to find those that are essential.”
- “Communication and simplicity often work together. The less excess complexity, the easier a system is to understand. The more you focus on communication, the easier it is to see what complexity can be discarded.”
- “Sometimes, however, I find a simplification that would make a program harder to understand. I choose communication over simplicity in these cases.”

Flexibility

- “Programs should be flexible, but only in ways they change. If the constant never changes, all the complexity is cost without benefit”
- “(...) the flexibility of simplicity and extensive tests is more effective than the flexibility offered by speculative design.”

Principles

“The principles bridge between the values, (…), are clear to apply and specific”

- “Faced with ambiguity, understanding the principles allows me to “make something up” that is consistent with the rest of my practice and likely to turn out well.”

Local Consequences

- “Structure the code so changes have local consequences. If a change here can cause a problem there, then the cost of the change rises dramatically.”
- “Code with mostly local consequences communicates effectively. It can be understood gradually without first having to assemble an understanding of the whole.”

Minimize Repetition

- Contributes to keeping consequences local
- “When you have the same code in several places, if you change one copy of the code you have to decide whether or not to change all other copies.”
- “One of the ways to remove duplication is to break programs up into many small pieces – small statements, small methods, small objects, small packages.”

Logic and Data Together

- “Put logic and the data it operates on near each other, in the same method if possible or the same object, or at least the same package. To make a change, the logic and data are likely to have to change at the same time”
- “I may be writing a code in A and realize I need data from B. It's only after I have the code working that I notice that it is too far from the data. Then I need to choose what to do: move the code to the data, move the data to the code, put the code and data together in a helper object, or realize I can't at the moment think of how to bring them together in a way that communicates effectively.”

Symmetry

- “An add() method is accompanied by a remove() method.”
- “Once readers understand one half of the symmetry, they can quickly understand the other half.”
- Example: process() {input(); count++; output();} to process() {input(); incrementCount(); output();} to process() {input(); tally(); output();}

Declarative Expression

- “(...) express as much of my intention as possible declaratively”

Rate of Change

- “(...) is to put logic and data that changes at the same rate together and separate logic or data that changes at different rates. These rates of change are a form of temporal symmetry”
- “The rate of change applies to data. All fields in a single object should change at roughly the same rate.”
  - “fields that are modified only during the activation of a single method should be local variables.”
  - “two fields that change together but out of sync with their neighboring fields probably belong to a helper object.”
- “Expressing the symmetry by putting them in their own object communicates their relationship to readers and is likely to set up further opportunities to reduce duplication and further localize consequences later.”

Conclusion

“The values of communication, simplicity, and flexibility provide widely ranging motivation for the patterns. The principles (…) help translate the values into action.”

Chapter 4 – Motivation

- “Maintenance is expensive because understanding existing code is time consuming and error-prone. Making changes is generally easy when you know what need changing.”
- We want to minimize the maintenance cost. For that we may do design up-front to create flexibility for future changes. However, this policy doesn't help us either once we have to spend more now and probably this changes won't be necessary.\
- “(...) the implementation patterns are focused on ways to gain immediate benefit while setting up clean code for ease of future development.”
- “The immediate benefits of clear code are fewer defects, easier sharing of code, and smoother development.”

Chapter 5 – Class

“Classes are important for communication because they describe, potentially, many specific things.”

Class: Use a Class to say, “This data goes together and this logic goes with it.”

- Kent Beck tells about the importance of the class hierarchy.
- “Reducing the number of classes of a system is an improvement, as long as the remaining classes do not become bloated.”

Simple Superclass Name: Name the roots of class hierarchies with simple names drawn from the sam metaphor

- “The right name results in a cascade of further simplifications and improvements.”
- “Once classes have been named, the names of operations follow”
- “In naming classes there is a tension between brevity and expressiveness.”
- “Look for one-word names for important classes”

Qualified Subclass Name: Name subclasses to communicate the similarities and differences with a superclass

- “The names of subclasses have two jobs. They need to communicate what class they are like and how they are different”
- “Prepend one or more modifiers to the superclass name to form a subclass name”.
- In some cases a subclass can be as important as a hierarchy root. In these cases, it should be name as a root class
- “Use the class names to tel the story of the code”

Abstract Interface: separate the interface from the implementation

- “The old adage (ditado) in software development is to code to interfaces, not implementation. This is another way of suggesting that a design decision should not be visible in more places than necessary.”
- Interface → “a set of operations without implementation”
- “Maximizing the number of interfaces doesn't minimize the cost of software. Pay for the interfaces only where you will need the flexibility they create.”

Interface: Specify an abstract interface which doesn't change often with a Java interface

- “Interfaces are a nice balance”
- “Interfaces as classes without implementations should be named as if they were classes”
- “Sometimes, naming concrete classes simply is more important to communication the hiding the use of interfaces. In this case, prefix interface names with “I”.”

Abstract Class: Specify an abstract interface which will likely change with an abstract class

- “Abstract interfaces need to support two kinds of change:”
  - “change in the implementation”
  - “change in the interface”
- “Every change to an interface requires changes to all implementations.
- “Abstract classes do not suffer this limitation. (…) new operations can be added to an abstract class without disrupting existing implementors.”

Versioned Interface: extend interfaces safely by introducing a new subinterface
- “What do you do when you need to change an interface but you can't? Typically this happens when you want to add operations.”
  - Create a subinterface: the users who want the new functionality should use the extended interfaces while existing users remain oblivious to the existence of the new interface.
  - Example: Command with run() method → ReversibleCommand with undo() method

Value Object: write an object that acts like mathematical value
- “This functional style of computing never changes any state, it only creates new values”
- “To implement value-style objects (that is, objects that act like integers rather than like holders of changing state), first draw a boundary between the world of state and the world of value. Set all state in a value-style object in the constructor, with no other fields assignments elsewhere in the object. Operations on value-style objects always return new objects. These objects must be stored by the requestor of the operation.”
  - Example: bounds.translateBy(10, 20); and bounds = bounds.translateBy(10, 20);
- “(...) I'll close by reiterating that sometimes your programs will best be expressed as a combination of state-changing objects and objects representing mathematical values.”

Specialization: clearly express the similarities and differences of related computations
- The following patterns are ways to deal with logic and data

Subclass: express one-dimensional variation with a subclass
- Some issues:
  - “you have to understand the superclass before you can understand the subclass”
  - “changes to a superclass are risky, since subclasses can rely on subtle properties of the superclasses implementation”
- “A particularly pernicious use of inheritance is creating parallel hierarchies, where for each subclass in *this* hierarchy you need a subclass in *that* hierarchy. This is a form of duplication, creating explicit coupling between the class hierarchies.”
- “Copied code introduces an ugly coupling between the two classes. You can't safely change the code in the superclass without examining and potentially changing all the places to which it has been copied.”

Implementor: override a method to express a variant of computation

Inner Class: bundle locally useful code in a private class
- “Declaring small, private classes (inner classes) gives you a low-cost way of having many of the benefits of a class without all of its costs.”
- “To get an inner class that is completely detached from its enclosing instances, declare it static”

Instance-specific Behavior: vary logic by instance
- “For the ease of code reading, try to set instance-specific behavior when an object is created and don't change it afterward.”

Conditional: vary logic by explicit conditionals

- “Conditionals as a form of expression have the advantage that the logic is still all in one class”
- “conditionals, have the disadvantage that they can't be modified except by modifying the code of the object in question.”
- “the more paths through a program the less likely the program is to be correct. The proliferation of conditionals reduces reliability.”
- Try to convert the conditional logic to messages, either with subclasses or delegation
- “In short, the strengths of conditionals – that they are simple and local – become liabilities when they are used too widely”

Delegation: vary logic by delegating to one of several types of objects

- “The common logic is held in the referring class, the variations in the delegates”
- “Delegation can be used for code sharing as well as instance-specific behavior.”
- Use double-dispatch or use the first class as a constructor parameter of the second class

Pluggable Selector: vary logic by reflectively executing a method

- “store the name of the method to be invoked in a field and invoke the method by reflection”

Anonymous Inner Class: vary logic by overriding one or two methods right in the method that is creating a new object

- “The idea is to create a class that is only used in one place, that can override one or more methods for strictly local purposes. Because it's used in one place, the class can be referred to implicitly instead by a name.”
- Some problems:
  - They are hard to test
  - They don't have names to express intention
  - They can't change

Library Class: represent a bundle of functionality that doesn't fit into any object as a set of static methods

- “Where do you put functionality that doesn't fit into any object? One solution is to create static methods on an otherwise-empty class.”
- “While library classes are fairly common, they don't scale well”
- “Try to turn library classes into objects whenever possible” → they forfeit the biggest advantage of programming with objects: a private namespace of shared data that can be used to help simplify logic.
- Static method with one parameter probably is one of the parameters instance methods

Chapter 6 – State

"The patterns in this chapter describe how to communicate your use of state. Objects are convenient packages of behavior which is presented to the outside world and state which is used to support that behavior."

State: compute with values that change over time

- "As soon as you assume what some bit of state is, your code is at risk. You might assume incorrectly or the state might change."
- Functional vs Objects → Concurrency vs State
- "A key to managing state effectively is putting similar state together and making sure different state stays apart."
- "If two pieces of state are used together and have the same lifetime, then having them close to each other is probably a good idea"

Access: maintaing flexibility by limiting access to state

- "One dichotomy in programing languages is the distinction between accessing stored values and invoking computations."
- "Deciding what to store and what to compute affects readability, flexibility, and performance of programs."
- Discussion: We have to know where we want flexibility
- "Ease of inter-object access isn't worth the loss of independence between objects"

Direct Access: directly access by limiting access to state

- "If I store values into that variable from many parts of the program, then to make a change I will likely have to change all those parts of the program."
- "The other downside of direct access is that it is an implementation detail, below the level of most of my thoughts while programming. Setting a variable to 1 may cause a garage door to open, but code that reflects this implementation detail won't communicate well."
- There are many rules such as:
  - direct access only inside accessor methods, and maybe inside constructors too
  - direct access only inside a single class or inside a class and all its subclasses
- "There is no universal rule. Programmers need to think, communicate, and learn"

Indirect Access: access state through a method to provide a greater flexibility

- "You can hide and changes to state behind method invocations. This accessor methods provide flexibility at the cost of clarity and directness. Clients no longer assume that a certain value is store directly. Thus, you are able to change your mind about storage decisions without affecting client code."
- "My default strategy for accessing state is to allow direct access inside of a class (including inner classes) and indirect access for clients."
- "Note: if most access to an object's state are outside the object, there is a deeper design problem lurking."
- "Another strategy is to use indirect access exclusively. I find this results in a loss of clarity. Most getting and setting are trivial. They often outnumber methods that perform useful work, making the code hard to read."
- "One definite case for indirect access is where two pieces of data are coupled."

Common State: store the state common to all objects of a class as fields

- "Many calculations share the same data elements even if the values are different. When you find such a calculation, communicate it by declaring fields in a class".
- "Your reader will want to know what it takes to successfully invoke the functionality in your

object. Common state communicates this clearly and precisely."
- "Common state in an object should all have the same scope and lifetime."

Variable State: store state whose presence differs from instance to instance as a map
- "Variable state is often stored as a map whose keys are the names of the elements and whose values are the data values"
- "Variable state is much more flexible than common state. Its primary failing is that it doesn't communicate well. What data elements need to be present for an object with only variable state to function correctly?"
- "One case where variable state seems justified is where the state of one field implies the need for other fields."
  - Example: "if I have a widget whose `bordered` flag is true, then I can also have `borderWidth` and `borderColor`."
  - In the example, using common state would violate the principle where all variables in an object should have the same lifetime.
  - "The presence of several variables that share a common prefix is a clue that a helper object of some sort may be useful."
- "Use common state wherever possible. Use variable state for the fields in an object that may or may not be needed depending on usage."

Extrinsic State: store special-purpose state associated with an object in a map held by the user of that state
- "Sometimes a part of your program needs state associated with an object, but the rest of the system doesn't care"
- "Store special-purpose information associated with an object near where it will be used instead of in the object."
  - Example: information about where an object is stored on disk is useful for the persistence mechanism but not the rest of the code. Putting this information together would violate the symmetry principle. In this case, the persistence mechanism would store an `IdentityMap` whose keys are the objects stored and whose values are the information about where they are stored.

Variable: variables provide a namespace for accessing state
- "(...) to reduce coupling you should mostly use locals and fields with only an occasional static field and the private modifier."
- Discussion: we should only have to understand the context to understand what you want.
- "Work hard to ensure that the lifetime of variables is close to their scope. Additionally, make sure that sibling variables (those defined in the same scope) all have the same lifetime."
- "With scope, lifetime, and type adequately communicated in other ways, the name can be used to convey the role of the variables in the computation. By reducing the information to be conveyed to a minimum, you are free to choose simple names that read well."

Local Variable: local variables hold state for a single scope
- "Following the principle that information should spread as little as possible, declare local variables just before they are used and in the innermost possible scope."
- Explaining variables:
  - "(...) if you have a complicated expression, assigning bits of the expression to local variables can help readers navigate the complexity"
  - "Explaining variables are often a step towards helper methods. The expression becomes the body of the methods, and the name of the local variable suggests a name for the method."

Field: fields store state for the life of an object

- Useful fields:
  - Helper: “if an object is passed as a parameter to many methods, consider replacing the parameter with a helper field set in the complete constructor.”
  - Flag: “flag field are fine if they are only used in a few conditionals. If the code making decisions on the basis of the flag is duplicated, consider changing to a strategy field instead.”
  - Strategy: “when you want to express that there are alternative ways to perform some part of the object's computation, store in a field an object performing just the variable part of the computation.”
  - State: “are like strategy fields in that part of the behavior of the object is delegate to them. However, state fields, when triggered, set the following state themselves.”
  - Components: “these fields hold objects or data “owned” by the referring object”

Parameter: parameters communicate state during the activation of a single method

- “The coupling introduced by a parameter is weaker than the coupling introduced by a permanent reference from one object to another”
- “If many messages form one object to another require the same parameter, it may be better to permanently attach the parameter to the called object”
  - Its is better to pass the parameter to the constructor, rather than sending it for every method call

Collecting Parameter: pass a parameter to collect complicated results from multiple methods

- “When merging the results is more complicated than simple addition, it is more direct to pass a parameter that will collect the results.”
  - Example: in addTo(results), results is a collecting parameter

Parameter Object: consolidate frequently used long parameter list into an object

- “If a group of parameters is passed together to many methods, consider making an object whose fields are those parameters and passing the object instead.”
- “The introduction of the parameter object shortens the code, explains its intent, and provides a home for the algorithm for expanding and contracting a rectangle (...)”
- “While the primary motivation for introducing a parameter object is to improve readability, parameter objects can become important homes for logic. The fact that the data appears together in several parameter list is a blatant clue that they are strongly related.”
- “The best code to optimize is readable, factored, and tested; parameter objects can contribute to these goals.”

Constant: store state that doesn't vary as a constant

- “The strongest reason to use constants, though, is because you can use the names of the constants to communicate what you mean by the value”
- “An interface where all invocations of a method have literal constants as arguments can be improved by giving it separate methods for each constant”

Role-Suggesting Name: name variables after the role they play in a computation

- “I want to communicate my intent fully through my names, which often suggests long names.”
- “I'd like the names to be short to simplify code formatting.”
- “Names will be read many times for each time they are typed, so the names should be optimized for readability, not ease of typing.”
- “Both the way the data in the variable is used and the role that data plays in the computation need to be expressed”
- “What is the point of telling the complier the types of variables over and over if I have to turn

around and embed that same information again in the variable names?”
- “Sometimes I am tempted to use several words for a variable, which makes the variable too long for comfortable typing. When this happens I look at the surrounding context. Why do I need so many words to distinguish this variable's role from the role of other variables? Often this leads me to simplify the design, allowing me to again write short variables names in good conscience.”

Declared Type: declare a general type for variables
- “Since you have to declare the type, you may as well use the declared type as an opportunity to communicate”
- “It is useful to declare variables and methods with a general type where possible. Losing a little precision and generality to maintain consistency is a reasonable trade-off.”
- “The best thing about generalizing declared types is that it opens up options for changing the concrete class during later modifications.”
- “In general, the further a decision propagates, the less flexibility you have for future change.”

Initialization: initialize variables declaratively as much as possible

Eager Initialization: initialize fields at instance creation time
- “Initialize variables in the declaration if possible. This puts the declared and actual types close together for readers.”
- “Initialize fields in the constructor if they can't be initialized in the declaration”

Lazy Initialization: initialize fields whose values are expensive to calculate just before they are first used
- “create a getter method and initialize the field when the getter is first called”

Chapter 7 - Behavior

Control Flow: express computations as a sequence of steps

Main Flow: clearly express the main flow of control

- "It's not that exceptional conditions are unimportant, just that focusing on expressing the main flow of the computation clearly is more valuable"
- "Therefore, clearly express the main flow of your program. Use exceptions and guard clauses to express unusual or error conditions."

Message: express control flow by sending a message

- "Using this flexibility [implementations may vary] wisely, making clear and direct expressions of logic where possible and deferring details appropriately, is an important skill if you want to write programs that communicate effectively."

Choosing Message: vary the implementors of a message to express choices

- Use polymorphism to vary the implementations
- "One of the costs of choosing messages is that a reader may have to look at several classes before understanding the details of a particular path through the computation. As a writer you can help the reader navigate by giving the methods intention-revealing names."
- "Also, be aware of when a choosing message is overkill. If there is no possible variation in a computation, don't introduce a method just to provide the possibility of variation."

Double Dispatch: vary the implementors of messages along two axes to express cascading choices

- "Double dispatch introduces some duplication with a corresponding loss of flexibility. The type names of the receivers of the first choosing message get scattered over the methods in the receiver of the second choosing message. (…), I would have to add methods to all the `Brushes`."

Decomposing (Sequencing) Message: break complicated calculations into cohesive chunks

- "Decomposing messages need to be descriptively named. (…) Only those readers interested in implementation details should have to read the code invoked by the decomposing message"
- "Difficulty naming a decomposing message is a tip-off that this ins't the right patter to use. Another tip-off is long parameter lists."
- "If I see these symptoms, I inline the method invoked by the decomposing message and apply a different pattern, like Method Object, to help me communicate the structure of the program"

Reversing Message: make control flows symmetric by sending a sequence of messages to the same receiver

- Example: compute() { input(); helper.process(this); output()} →
  compute() {input(); process(helper); output()} and process() { helper.process(this); }
- "Sometimes the helper method invoked by a reversing message becomes important on its own. (…) it would probably be better structured by moving the whole compute() method to the `Helper` class"
- "Sometimes I feel silly introducing methods "just" to satisfy "aesthetic" urge like symmetry. Aesthetics go deeper than that. Aesthetics engage more of your brain than strictly linear logic thought. Once you have cultivated your sense of aesthetics of code, the aesthetics impressions you receive of your code is valuable feedback about the quality of the code."

Inviting Message: invite future variation by sending a message that can be implemented in different ways

- "If there is a default implementation of the logic, make it the implementation of the message. If

not, declare the method abstract to make the invitation explicit”

Explaining Message: send a message to explain the purpose of a clump of logic

- “You can use messages to make this distinction by sending a message named after the problem you are solving which in turn sends a message named after how the problem is to be solved”
- Example: highlight(Rectangle area) { reverse(area); }
  - In the context of a client, highlight() can be much more expressive than the actual problem solution that is reverse()
- “Consider introducing an explaining message when you are tempted to comment a single line of code”

Exceptional Flow: express the unusual flows of control as clearly as possible without interfering with the expression of the main flow

- “Express the main flow clearly, and these exceptional paths as clearly as possible without obscuring the main flow. Guard clauses and exceptions are two ways of expressing exceptional flow”

Guard Clause: express local exceptional flows by early return

- initialized() { if (!isInitialized()) { … } } → initialized() { if (isInitialized()) { return; } …}
- “Nested conditions breed defects”
- Guard Clauses minimize the nested structures

Exception: express non-local exceptional flows with exceptions

- “Throwing an exceptional at the point of discovery and catching at the point where it can be handled is much better than cluttering all the intervening code with explicit checks for all the possible exceptional conditions, none of which can be handled”
- “Exceptions cost. They are a form of design leakage.”
- “In short, express control flows with sequence, messages, iteration, and conditionals (in that order) wherever possible. Use exceptions when not doing so would confuse the simply communicated main flow.”

Checked Exception: ensure that exceptions are caught by declaring them explicitly

- Checked exception is an interesting idea to prevent program termination, but they add length to method declaration, add another thing to read and understand and makes it more difficult to change the code.

Exception Propagation: propagate exceptions, transforming them as necessary so the information they contain is appropriate to the catcher

- “Wrap the low level exceptions in the higher-level exception so that when the exception is printed, on a log for example, enough information is written to help find the defect”

Chapter 8 – Methods

“The common issues in dividing a program into methods are the size, purpose, and naming of the methods. If you make too many too small methods, readers will have a hard time following your fragmented expression of ideas. Too few methods leads to duplication and the attendant loss of flexibility.”

Composed Methods: compose methods out of calls to other methods

- “Compose methods out of calls to other methods, each of which is at roughly the same level of abstraction”
- “How long should a method be? Some people recommend numerical limits, like less than a page or 5-15 lines. While it may be true that most readable code satisfies such a limit, limits beg the question “why?””
- Nice arguments on the use of smaller methods at page 78, 3rd and 4th paragraphs.
  - “When reading for overall structure, seeing lots of code at once is valuable.”
  - “To understand details, I want closely related details gathered together and segregated from irrelevant details”
- “Compose methods based on facts, not speculation. Get your code working, then decide how it should be structured.”

Intention-revealing Name: name methods after what they are intended to do

- “Methods should be named for the purpose a potential invoker might have in mind for using the method”
- “Unless the implementation strategy is relevant to users, leave it out of the name”

Method Visibility: make methods as private as possible

- “The more methods that are revealed, the harder it is to change the interface to an object (...)”
- “A too-narrow interface leaves all clients performing more work than necessary to use your object.”
- “My general strategy for visibility is to restrict it as much as possible”
- “One good use of static methods is as replacement for constructors.”

Method Object: turn complex methods into their own objects

- “This is one of my favorite patterns, probably because I use it so infrequently bu the results are spectacular when I do”
- Idea: create an object in which its fields are the parameters of the complex method and create a calculate() method that does all the work.
- “The code in the new class is easy to refactor. You can extract methods and never have to pass any parameter because all the data used by the method is stored in fields.”

Overridden Method: override methods to express specialization

Overloaded Method: provide alternative interfaces to the same computation

- “Different return types for different overload methods make reading the code too difficult. Better to find a new name for the new intention. Give different computations different names.”

Method Return Type: declare the most general possible return type

- “(...) you'd like your methods to be as broadly applicable as possible, so pick the most abstract return type that expresses your intention.”

Method Comment: comment methods to communicate information not easily read form the code

- “Express as much information as possible through the names and structure of the code. Add

comments to express information that is not obvious from the code. Where they are expected, add javadoc comments to explain the purpose of methods and classes."
- "Automated tests can communicate information that doesn't fit naturally in method comments"
- "If a method comment is the best possible medium for communication, write a good comment."

Helper Method: create small, private methods to express the main computation succinctly
- "Eliminate helpers (at least temporarily) when the logic of a method becomes unclear. Inline all the helper methods, take a fresh look at the logic, and re-extract methods that make sense."

Debug Print Method: use `toString()` to print useful debugging information

Conversion: express the conversion of one type of object to another cleanly
- "Another issue to consider is the dependencies between classes. It's not worth introducing a new dependency just to have a convenient expression of conversion."

  Conversion Method:
  - For simple, limited conversions, provide a method on the source object that returns the converted object

  Conversion Constructor:
  - For most conversions, provide a method on the converted objects's class that takes the source object as parameter

Creation: express object creation clearly

Complete Constructor: writer constructors that return fully formed objects
- "Even if you have a factory method [for more abstraction], provide a complete constructor beneath it so curious readers can easily"
- "When implementing a complete constructor, funnel all the constructors to a single master constructor that does all the initialization."

Factory Method: express more complex creation as a static method that may need explanation or later refinement
- "These methods have a couple of advantages over constructors: they can return a more abstract type (a subclass or an implementation of an interface) and they can be named after their intention, not just the class."
- "However, factory methods add complexity, so they should be used when their advantages are valuable, not just as a matter of a course."
- "I don't want to waste my readers' time, so if all that's happening is vanilla object creation, I express it as a constructor. If something else is happening at the same time, I introduce a factory method to tip curious readers to this fact."

Internal Factory: encapsulate in a helper method object creation that may need explanation or later refinement
- "What do you do when the creation of a helper object is private but complex or subject to change by subclass? Make a method which creates and returns the new object."
- "Internal factories are common in lazy initialization."

Collection Accessor Method: provide methods that allow limited access to collections
- "Suppose you have an object that contains a collection. How do you provide access to that collection? The simplest solution is a getting method for the collection"
- "This gives clients maximum flexibility but creates a variety of problems. Internal state that

depends on the contents of the collection can be invalidated behind your back if you return the whole collection."
- "(...), offer methods that provide limited, meaningful access to the information in the collection."
- "If clients need to iterate over the elements of the collection, provide a method that returns an iterator"
- "If you find yourself duplicating most of the collection protocols, it's likely you have a design problem. If your object did more work for its clients, if wouldn't have to offer so much access to its innards."

Boolean Accessor Method: if it helps communication, provide two methods to set boolean values, on for each state (eg. true and false)

Query Method: return boolean values with methods names asXXX

Equality Method: define `equals()` and `hashCode()` together

Getting Method: occasionally provide access to fields with a method returning that field
- "Following the principle of putting logic and data together, the need for public – or package – visible getting methods is a clue that logic should be elsewhere. Rather than writing the getting method, try moving the logic that uses the data instead"
- "There are a couple of exceptions to my aversion to visible getting methods"
  - "One is when I have a set of algorithms located in their own objects. Algorithms need access to data and need a getting method to receive it."
  - "(...), getting methods that will be invoked by tools will often have to be public"

Setting Method: even less frequently provide the ability to set fields with a method
- "Using a setting method as part of the interface lets the implementation leak out (…) Naming the interface after the purpose of the method helps the code speak [even if the implementation is just a setter]"

Safe Copy: avoid aliasing errors by copying objects passed in or out of accessor methods

Chapter 9 – Collections

- The author discusses a little on the performance of each collection and when to choose between them

Chapter 10 – Evolving Frameworks

- ”The preceding implementation patterns assume that changing code is cheap compared to understanding and communicating the intent of code. This has been true of most of my development experience.”
- “However, framework development, in which client code can't be changed by the framework developers, violates this assumption.”
- “This chapter sketches out how implementation patterns change when developing frameworks.”

Changing Frameworks without Changing Applications

- “The perfect framework upgrade adds new functionality without changing any existing functionality. (…) Maintaining backward compatibility often adds complexity to the framework.”
- “(...) in framework development it is often more cost-effective to add complexity in order to enhance the framework developer's ability to improve the framework without breaking client code.”
- “I tend to make fields protected in most of my code, but while developing frameworks I make them private. This makes my superclasses harder for clients to use, but allows me to change my framework data representations without affecting clients' applications.”

Incompatible Upgrades

- “Staging the upgrade is small steps gives clients warning of what is coming an lets them decide when to make the investment in changing their code.”
- “Packages can provide a way to offer clients incremental access to upgrades. By introducing new classes in a new package, you can give them the same name as the old classes. (…) Changing imports is less risky and intrusive than changing code”
- “Another incremental strategy is to change either the API or the implementation but not both in the same release.”
- “You can reduce the cost of changing code if clients can switch to your upgraded functionality with a simple find/replace operation.”

Encouraging Compatible Change

Library Class

- “If you can represent all your functionality as procedures calls with simple parameters then clients are well insulated from future changes.”
- “The big problem with representing an API as a library class is the limited number of concepts and variants that can be easily expressed.”

Objects

- Style of Use, Abstraction, Creation and Methods

…