**Chapter 24: Refactoring**

* Code is constantly evolving, don’t fear changing things

**Kinds of Software Evolution**

* Some mutations are good, some are not
* The key distinction is whether a programs quality improves or degrades under modification
* Another is the distinction between changes made during construction and those made during maintenance

Philosophy of Software Evolution

* A common weakness in programmers is that evolution happens naturally, as in its not explicitly planned for
* Big mentality shift is to plan for evolution and use it to your advantage
* “You cant always make good code better but you can always make bad code worse”

**Introduction to Refactoring**

Reasons to Refactor

* “Smells” that code needs to be refactored
  + Code is duplicated
    - Duplicated code almost always represents a failure to fully factor the design in the first place
    - This also means you need to make parallel modifications which is no good
    - **“Copy and paste is a design error”**
  + A routine is too long
  + A loop is too long or to deeply nested
  + A class has poor cohesion
    - Takes ownership of unrelated responsibilities
  + A class interface does not provide a consistent level of abstraction
  + A parameter list has too many parameters
  + Changes WITHIN a class are compartmentalized
    - Like multiple compartments within one class
    - 🡪 make 2 classes out of these compartments
  + Changes require parallel modifications to multiple classes
  + Inheritance hierarchies have to be modified in parallel
  + Case statements have to be modified in parallel
  + **BASICALLY ANY TIME ANYTHING IS MODIFIED IN PARALLEL**
  + Related data items that are used together are not organized into classes
  + A routine uses more features of another class than of its own class
  + A primitive datatype is overloaded
  + A class doesn’t do very much
  + A chain of routines pass tramp data
    - Data passed through one routine to one below it without it being used
  + A middleman object isn’t doing anything
  + One class is overly intimate with another
    - Poorly encapsulated (information hiding)
  + A routine has a poor name
  + Data members are public
  + A subclass uses only a small percentage of its parents routines
  + Comments are used to explain difficult code
  + Global variables are used
  + A routine uses setup code before a routine call, or a takedown code after a routine call
    - Means the level of abstraction is bad
  + Code looks like it was designed to be used “someday”
    - Get rid of that

Reasons Not to Refactor

* Change itself is not a virtue
* But, purposeful change combined with discipline can be a key strategy that supports steady improvement in quality

**Specific Refactorings**

Data-Level Refactorings

* Replace a magic number with a named constant
* Rename a variable with a clearer or more informative name
* Move an expression inline
* Replace an expression with a routine
* Introduce an intermediate variable
  + To summarize the purpose of the expression
* Convert a multiuse variable to multiple single use variables
* Use a local variable for local purposes rather than a parameter
* Convert a data primitive to a class
* Rather than defining stand alone constants, create a class to house them
  + Also allows for stricter type checking
* Convert a set of type codes to a class with subclasses
  + OutputType
    - Screen
    - Printer
    - File
* Change an array to an object
* Replace a traditional record with a data class

Statement-Level Refactoring

* Decompose Boolean expressions
  + Use well named intermediate variables
* Move complex Boolean expressions into well named Boolean functions
* Consolidate fragments that are duplicated within different parts of a conditional
* Use break or return instead of a loop control variable
* Return as soon as you know the answer instead of assigning a return value
* Replace conditionals with polymorphism

Routine-Level Refactorings

* Extract code from one routine and turn it into its own routine
* Move a routines code inline
* Convert a long routine into a class
* Substitute a simple algorithm for a complex algorithm
  + Or replace a complicated algorithm with a simple one
* Add a parameter
* Remove a parameter
* Separate query operations from modification operations
* Combine smaller routines by parameterizing them
  + If two routines use the same variable, combine them
* Separate routines whose behavior depends on parameters passed in
* Pass the whole object rather than specific fields
* Pass specific fields rather than a whole object

Class Implementation Refactorings

* Change value objects to reference objects
  + Have one central thing with real data than a bunch of other objects that look to the one
* Change reference objects to value objects
* Replace virtual routines with data initialization
* Change member routine or data placement
  + Pull a routine up into its superclass
  + Pull a field up into its superclass
  + Pull a constructor body up into its superclass
  + Push a routine down into its derived classes
  + Push a field down into its derived classes
  + Push a constructor body down into its derived classes
* Extract specialized code into a subclass
* Combine similar code into a superclass

Class Interface Refactorings

* Move routines to another class
* Convert one class to two
* Eliminate a class
* Hire a delegate
  + A 🡪 B 🡪 C desired
  + Dont necessarily need A 🡪 B and A 🡪 C
  + And of course the converse of this…..
* Remove a middleman
* Replace inheritance with delegation
* Replace delegation with inheritance
* Introduce a foreign routine
* Introduce an extension class
* Encapsulate an exposed member variable
* Hide routines that are not intended to be used outside the class
* Encapsulate unused routines

System-Level Refactorings

* Create a definitive reference source for data you cant control
  + Like GUI data and stuff
* Change unidirectional class association to bidirectional class association
* Change bidirectional class association to unidirectional class association
* Provide a factory method rather than a simple constructor
* Replace error codes with exceptions or vice versa

**Refactor Safely**

* Software refactoring is like brain surgery replacing a nerve
* Save code you start with
* Keep refactoring small
* Do refactorings at one time
* Make list of steps you intend to take
* Make list of other changes you encounter that need to be made, but don’t work on them till done
* Make frequent checkpoints
* Use compiler warnings
* Re-test
* Add test cases
* Review the changes
* Small changes tend to be more error prone than larger changes (peaks at 5 lines)

Bad Times to Refactor

* **Refactoring refers to changes in working code**
* Not “fixing” something that doesn’t quire work
* Don’t be afraid to completely toss code

**Refactoring Strategies**

* Spend your time on 20% of the refactorings that yield 80% of the benefit
* Refactor when you add a routine
* Refactor when you add a class
* Refactor when you fix a defect
* Target error-prone modules
* Target high-complexity modules