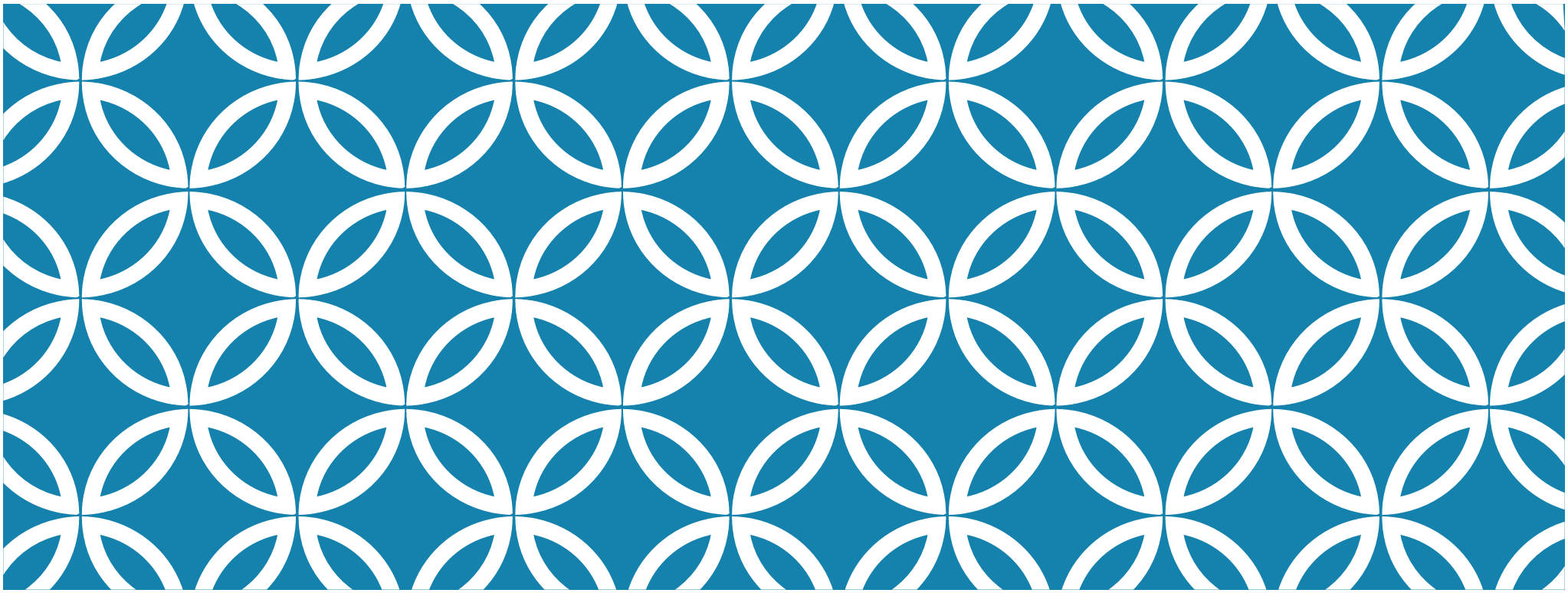


PRACTICES > REFACTORING

CS 346: Application Development

THE COST OF BEING AGILE

As Agile developers, we are always building the smallest working version of a feature.

- We aim for Minimum Viable Product (MVP) most of the time.
- The assumption is that we will iterate through feedback.
- Development is incremental.

We aren't scoping out our code in its entirety; instead we revisit and modify existing code.

However, there is a cost to doing this… **technical debt**.

WHAT IS TECHNICAL DEBT?

We all *want* to write perfect code, but for various reasons, we often end-up with less-than-perfect solutions.

- **Rushed features**: Sometimes we don't have enough time, so we cut corners.
- **Lack of tests**: We might think that the code is ready, but we haven't tested it adequately.
- **Lack of communication**: Perhaps we misunderstood requirements, or how a feature would integrate into the larger product.
- **Poor design**: Possibly our design is rigid and makes adding or modifying features difficult.

These are all choices that may cost us time later. We may need to stop and redesign a rushed feature, or we may need to fix bugs later.

We refer to this as **technical debt** — deferred costs of doing something "properly".

- Ignoring debt can lead to the quality of our code degrading over time.

REFACTORING

"**Refactoring** is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour. Its heart is a series of small behaviour preserving transformations. Each transformation (called a "refactoring") does little, but a sequence of these transformations can produce a significant restructuring."

— Martin Fowler, Refactoring. 2000, 2018.

Refactoring suggests that code must be continually improved as we work with it. We need to be in a process of perpetual, small improvements.

The goal of refactoring is to reduce technical debt by making small continual improvements to our code.

WHAT DOES IT LOOK LIKE?

Refactoring your code means doing things like:

- Cleaning up class interfaces and relationships.
- Fixing issues with class cohesion.
- Reducing or removing unnecessary dependencies.
- Simplifying code to reduce unnecessary complexity.
- Making code more understandable and readable.
- Adding more exhaustive tests.

In other words, refactoring involves code improvement *not* related to adding functionality.

It is acceptable to refactor code ***as part of*** adding new functionality.

"CLEAN CODE"

Martin (2008) would say that refactoring produces a "clean" codebase that can be adapted over time as requirements change.

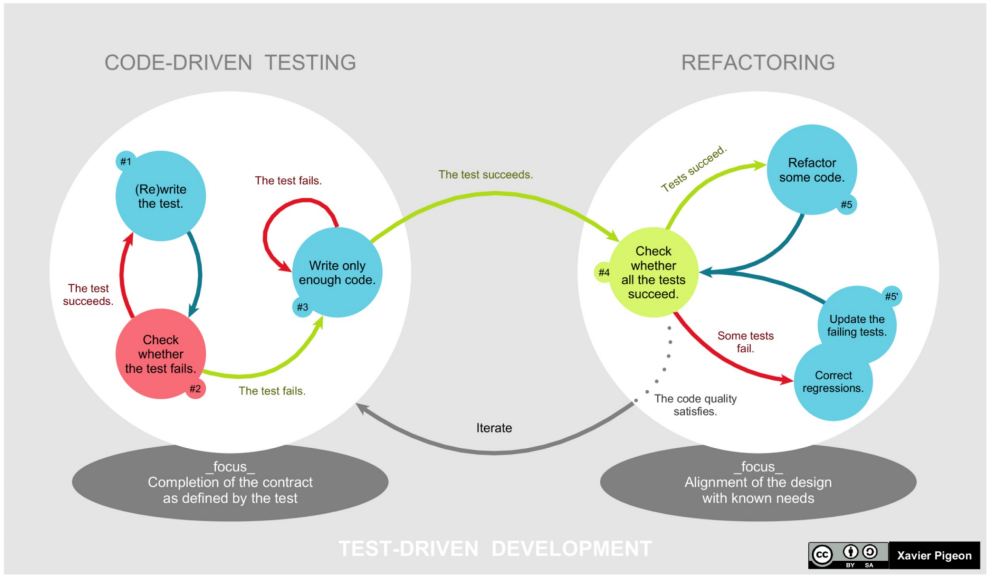
A "clean" codebase is:

- **Clear and easy to read**: variable names that make sense, no "magic number", classes that aren't bloated, well-constructed methods with no side effects.
- **Simple**: no unnecessary complexity that makes difficult to understand, or prone to errors.
- **Robust**: resilient to change, and unlikely to break when small changes are made.
- **Intentionally designed**: carefully segmented and structured with no code duplication.
- **Well-tested**: unit and integration tests demonstrate that the code is working correctly.

TDD & REFACTORING WORK TOGETHER

You should continually refactor as you expand your code and rely on the tests to guarantee that you aren't making any breaking changes to your code.

- TDD makes refactoring possible. You shouldn't refactor without significant unit tests in-place.



WHEN TO REFACTOR?

You often have to work with code for a while before you understand it. Don't be paralyzed by a less-then-perfect decision.

Rule of Three

- When you're doing something for the first time, just get it done.
- When you're doing something similar for the second time, do the same thing again.
- When you're doing something for the third time, start refactoring.

When adding a feature

Refactor existing code *before* you add a new feature, since it's much easier to make changes to clean code. You will improve it for yourself but also for those who use it after you.

When fixing a bug

If you find or suspect a bug, refactoring to simplify existing code can often reveal logic errors.

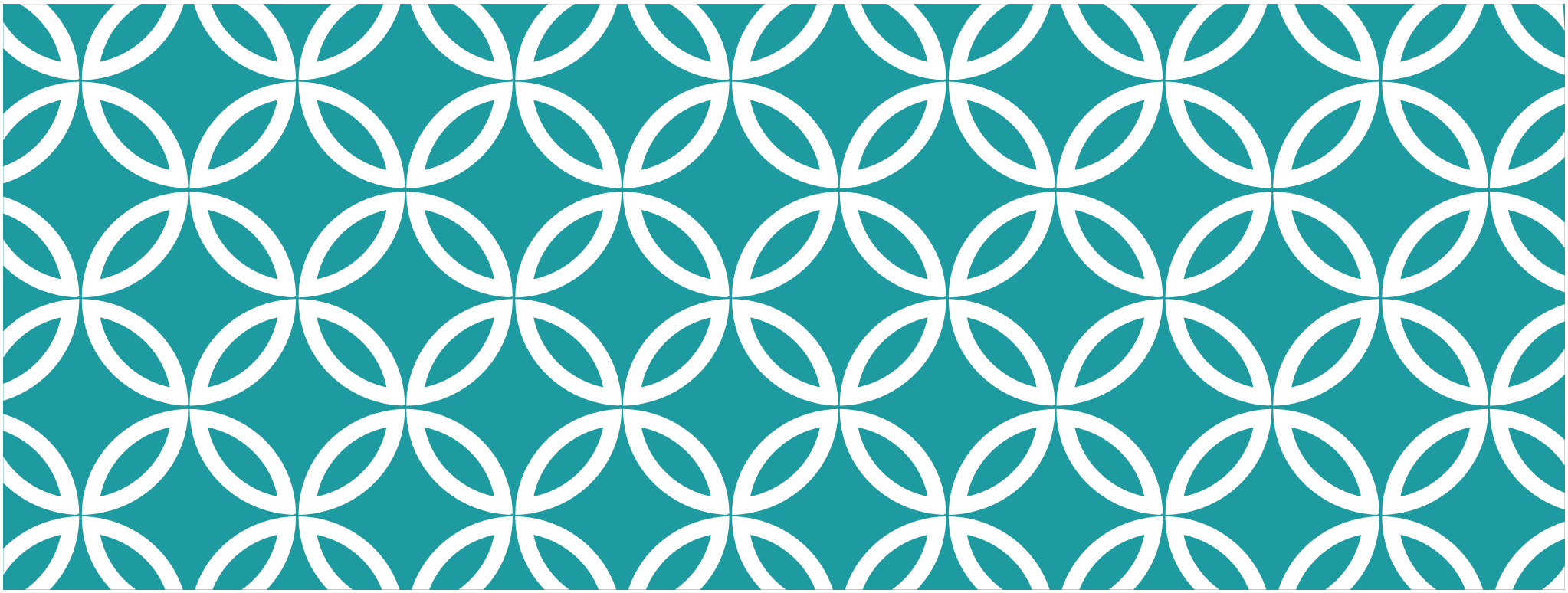
During a code review

The code review may be the last chance to tidy up the code before it becomes public.

RULES FOR REFACTORING

Make the existing code slightly better while still leaving the program in working order. Refactoring should:

- **Make the code cleaner**. Use refactoring to rename methods and variables, break apart complex methods into simpler ones, or create new data classes to isolate complexity.
- **Do not add any new functionality** during refactoring.
- **Ensure that all existing tests continue to pass**. There are two case where tests can break down:
  1. You made an error during refactoring. This one is a no-brainer: go ahead and fix the error.
  2. Your tests were too low-level. For example, you were testing private methods of classes. In this case, the tests are to blame. You can either refactor the tests themselves or write an entirely new set of higher-level tests.



REFACTORING > PATTERNS

CS 346: Application Development

REFACTORING PATTERNS

IntelliJ IDEA has built-in support for these operations!

Martin Fowler. 2018. **Refactoring: Improving the Design of Existing Code.** 2nd Edition. Addison-Wesley. ISBN 978-0134757599. https://refactoring.com/catalog/

6. COMPOSING METHODS
- 1. Extract Method
- 2. Inline Method
- 3. Inline Temp
- 4. Replace Temp with Query
- 5. Introduce Explaining Variable
- 6. Split Temporary Variable
- 7. Remove Assignments to Parameters
- 8. Replace Method with Method Object
- 9. Substitute Algorithm

7. Moving features between elements
- 10. Move method
- 11. Move field
- 12. Extract Class
- 13. Inline Class
- 14. Hide Delegate
- 15. Remove Middle Man
- 16. Introduce Foreign Method
- 17. Introduce Local Extension

8. ORGANIZING DATA
- 18. Self Encapsulate Field
- 19. Replace Data Value with Object
- 20. Change Value to Reference
- 21. Change Reference to Value
- 22. Replace Array with Object
- 23. Duplicate Observed Data
- 24. Change Unidirectional Association to Bidirectional
- 25. Change Bidirectional Association to Unidirectional
- 26. Replace Magic Number with Symbolic Constant
- 27. Encapsulate Field
- 28. Encapsulate Collection
- 29. Remove Record with data class
- 30. Replace Type Code with Class
- 31. Replace Type Code with Subclasses
- 32. Replace Type Code with State/Strategy
- 32. Replace Subclass with Fields

10. MAKING METHOD CALLS SIMPLER
- 41. Rename method
- 42. Add Parameter
- 43. Remove Parameter
- 44. Separate Query from Modifier
- 45. Parameterize Method
- 46. Replace Parameter with Explicit Methods
- 47. Preserve Whole Object
- 48. Replace Parameter with Method
- 49. Introduce Parameter Object
- 50. Remove Setting Method
- 51. Hide Method
- 52. Replace Constructor with Factory Method
- 53. Encapsulate Downcast
- 54. Replace Error Code with Exception
- 55. Replace Exception with Test

https://github.com/HugoMatilla/Refactoring-Summary

EXAMPLE: EXTRACT METHOD

We might extract a method from existing code.

Do this to make the original higher-level function is easier to read, or to improve the ability of a function to be called from elsewhere in the code.

```
// original
fun printOwing(name: String, amount: Double) {
  printBanner()
  //print details
  println("name: $name")
  println("amount: $amount")
 }
```

```
// refactored
fun printOwing(name: String, amount: Double) {
    printBanner();
    printDetails(name, amount);
}


fun printDetails (name: String, amount: Double) {
    println("name: $name")
    println("amount: $amount")
}
```

https://github.com/HugoMatilla/Refactoring-Summary - 1-extract-method

EXAMPLE: INLINE METHOD

We might also the opposite: remove a pointless method.

Do this when indirection is needless (simple delegation). Also do this when group of methods are badly factored and grouping them makes them clearer.

```
// original
fun getRating(): Int {
    return moreThanFiveLateDeliveries() ? 2 : 1
}


fun moreThanFiveLateDeliveries(): Boolean {
    return _numberOfLateDeliveries > 5
}
```

```
// refactored
fun getRating(): Int {
    return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

https://github.com/HugoMatilla/Refactoring-Summary#2-inline-method

EXAMPLE: MOVE METHOD

A method is using or used by more features of another class than the class on which it is defined. *Do this when classes collaborate too much and are highly coupled.*

Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation or remove it altogether.

```
// original
class Class1 {
    method()
}

class Class2 {  }
```

```
// refactored
class Class1 {  }

class Class2 {
    method()
}
```

https://github.com/HugoMatilla/Refactoring-Summary#10-move-method

EXAMPLE: EXTRACT CLASS

You have one class doing work that should be done by two. Create a new class and move the relevant fields and methods from the old class into the new class.

Do this when subsets of methods seem to belong together, or you have data that could be managed as an independent class.

```
// original
class Person {
    name,
    officeAreaCode,
    officeNumber,
    getTelephoneNumber()
}
```

```
// refactored
class Person {
    name,
    getTelephoneNumber()
}


class TelephoneNumber {
    areaCode,
    number,
    getTelephoneNumber()
}
```

https://github.com/HugoMatilla/Refactoring-Summary#12-extract-class

EXAMPLE: REMOVE MIDDLE MAN

A class is doing too much simple delegation. *Get the client to call the delegate directly.*

Do this when the "Middle man" (the server) does "too much".

```
// original
class ClientClass {
    val person = Person()
    person.doSomething()
}


class Person {
  fun doSomething() {
    val department = Department()
    department.doSomething()
  }
}
```

```
// refactored
class ClientClass {
    val person = Person()
    val department = Department()
    person.doSomething()
    department.doSomething()
}
```

https://github.com/HugoMatilla/Refactoring-Summary#15-remove-middle-man

EXAMPLE: INTRODUCE FOREIGN METHOD

A server class you are using needs an additional method, but you can't modify the source code for the original class.

```
// original
val newStart = Date(previousEnd.getYear(),previousEnd.getMonth(),previousEnd.getDate()+1)

// refactored: cannot change date class, so add “foreign method”
fun nextDay(date: Date): Date {
    return Date(date.getYear(),date.getMonth(),date.getDate()+1);
}
val newStart = nextDay(previousEnd)

// refactored: extend Date class, using Kotlin features
fun Date.nextDay(): Date {
    return Date(it.getYear(), it.getMonth(),it.getDate()+1);
}
val newStart = previousEnd.nextDay()
```

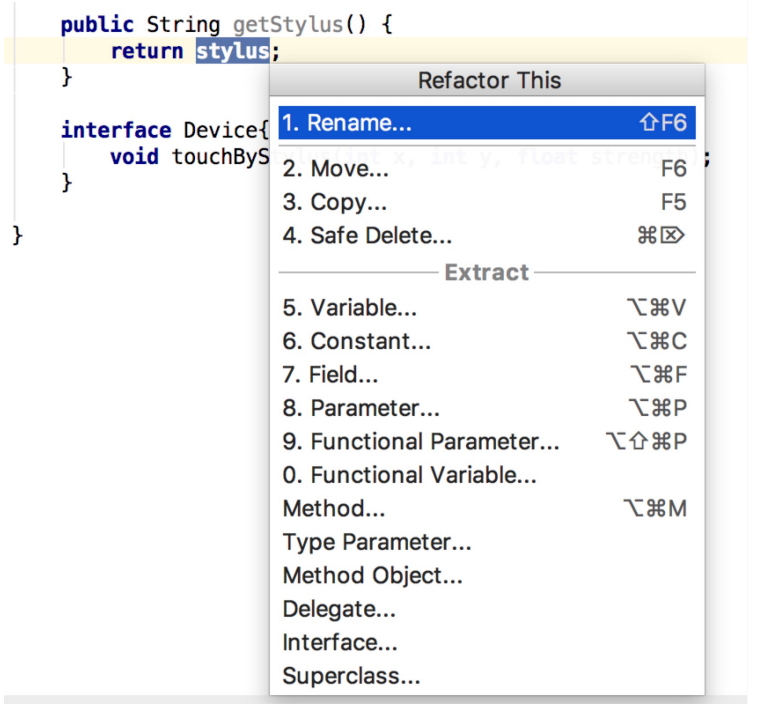
https://github.com/HugoMatilla/Refactoring-Summary#16-introduce-foreign-method

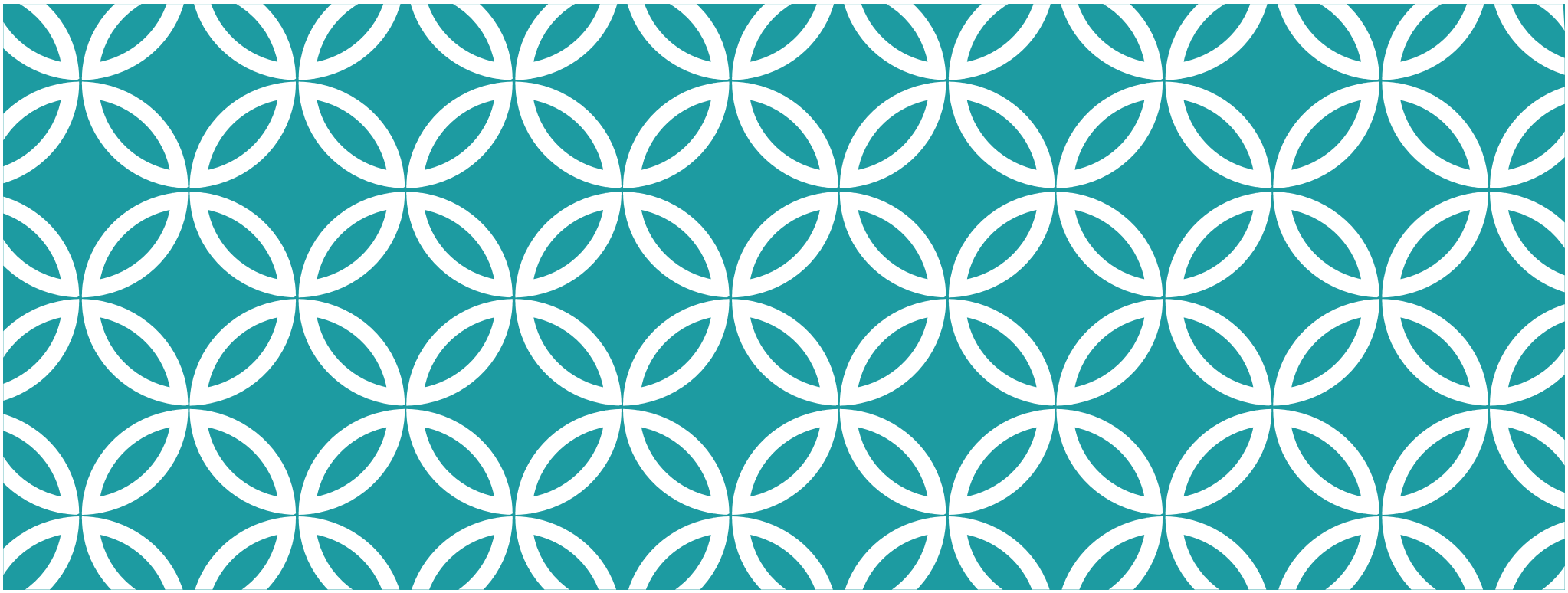
REFACTORING SAFELY

IntelliJ IDEA makes refactoring easy by providing automated ways of safely transforming your code.

To invoke the refactoring menu, select an item in your source code (e.g. variable or function name) and press `Ctrl-T` to invoke the popup menu.

You can also access it from the **Refactor** dropdown in the main menu.





APPENDIX: CODE SMELLS

CS 346: Application Development

CREDIT

Credit to refactoring.guru for this section.

CODE WHAT?

A "**code smell**" is a sign that a chunk of code is badly designed or implemented. It's a great indication that you may need to refactor the code.

Adjectives used to describe code:

- "neat", "clean", "clear", "beautiful", "elegant" <— the reactions that we want
- "messy", "disorganized", "ugly", "awkward" <— the reactions we want to avoid

A negative emotional reaction is a flag that your brain doesn't like something about the *organization* of the code - even you can't immediately identify what that is.

Conversely, a positive reaction indicates that your brain can easily perceive and following the underlying structure.

BLOATERS

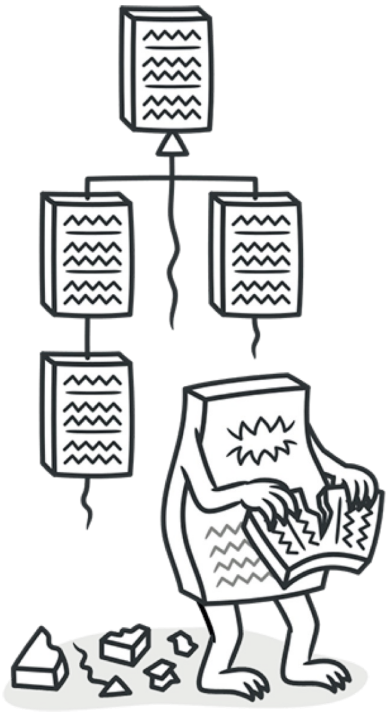
Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with.

- **Long method**: A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.
- **Large class**: A class contains many fields/methods/lines of code. This suggests that it may be doing too much. Consider breaking out a new class, or interface.
- **Primitive obsession**: Use of related primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.). Consider creating a data class, or small class instead.
- **Long parameters list**: More than three or four parameters for a method. Consider passing an object that owns all of these. If many of them are optional, either provide sensible defaults or use a builder pattern.

OBJECT-ORIENTATION ABUSERS

Incomplete or incorrect application of object-oriented principles.

- **Alternative Classes with Different Interfaces**: Two classes perform identical functions but have different method names. Consolidate methods into a single class instead, with support for both interfaces.
- **Refused bequest**: If a subclass uses only *some* of the methods and properties inherited from its parents, the hierarchy is broken. This violates the Liskov-substitution principle! Add missing behaviour or replace inheritance with delegation.
- **Switch Statement**: You have a complex `switch` operator, or sequence of `if` statements. This sometimes indicates that you are switching on type, something that should be handled by polymorphism instead. Consider whether a class structure and polymorphism makes more sense in this case.
- **Temporary Field**: Temporary fields get their values (and thus are needed by objects) only under certain circumstances. Outside of these circumstances, they're empty or have stale values. This may be a place to (carefully) introduce `nullable` types, to make it very clear what is happening.



DISPENSIBLES

A dispensable is something pointless or unneeded whose absence would make the code cleaner and easier to understand.

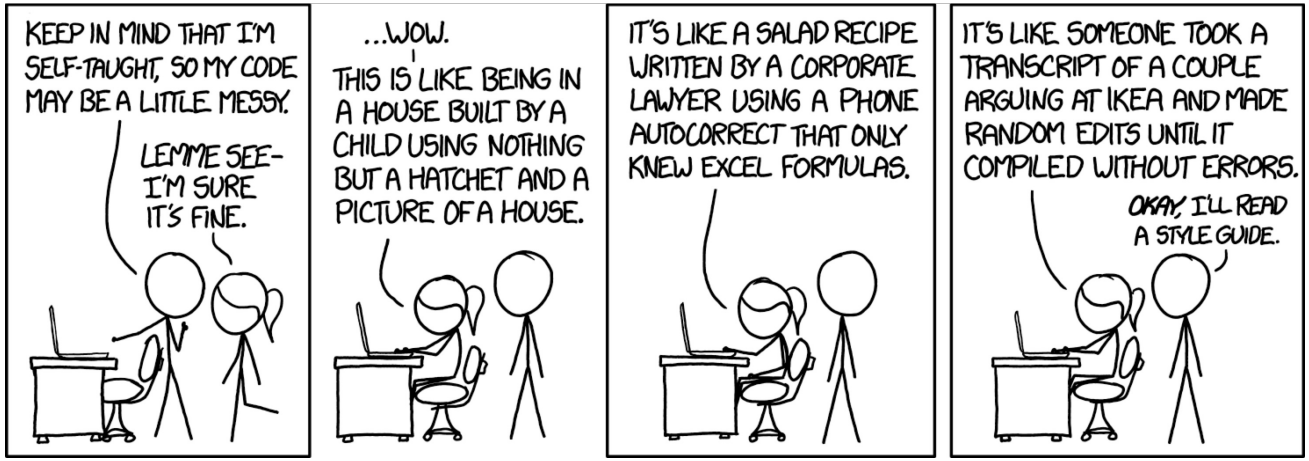
- **Comments**: A method is filled with explanatory comments. These are usually well-intentioned, but they're not a substitute for well-structured code. Replace or remove excessive comments.
- **Duplicate Code**: Two code fragments look almost identical. Typically, done accidentally by different programmers. Extract the methods into a single common method that is used instead. If the methods solve the same problem in different ways, pick and keep the most efficient one.
- **Dead Code**: A variable, parameter, field, method or class is no longer used (usually because it's obsolete). Delete unused code and unneeded files. You can always find it in Git history.
- **Lazy Class**: Understanding and maintaining classes always costs time and money. If a class doesn't do enough to earn your attention, it should be deleted. This is tricky: sometimes a small data class is clearer than using primitives (e.g. a Point class, vs using x and y stored as doubles).

COUPLERS

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

- **Feature envy**: A method accesses the data of another object more than its own data. This smell may occur after fields are moved to a data class. If this is the case, you may want to move the operations on data to this class as well.
- **Inappropriate intimacy**: One class uses the internal fields and methods of another class. Either move those fields and methods to the second class, or extract a separate class that can handle that functionality.
- **Middle man**: If a class performs only one action, delegating work to another class, why does it exist at all? It can be the result of the useful work of a class being gradually moved to other classes. The class remains as an empty shell that doesn't do anything other than delegate. Remove it.

FINAL WORD



https://xkcd.com/1513