REFACTORING AF

(AFTER FOWLER)

AARON MCCLENNEN

- I have watched multiple codebases become legacy.
- My first read of Fowler turned my life around.
- Started writing code in 1990

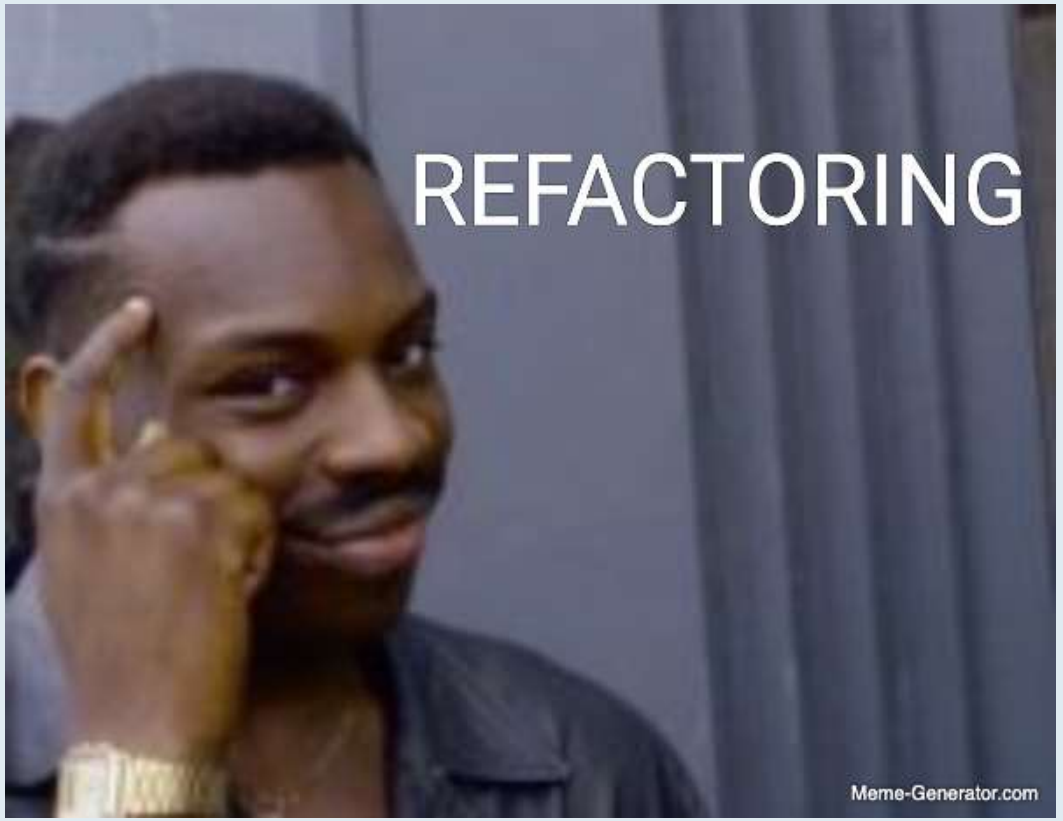
M. JEFF WILSON

- SAFe 6.0 Practice Consultant
- Old school gamer
- Coding since 1981

GOALS

REKINDLE REFACTORING EXCITEMENT

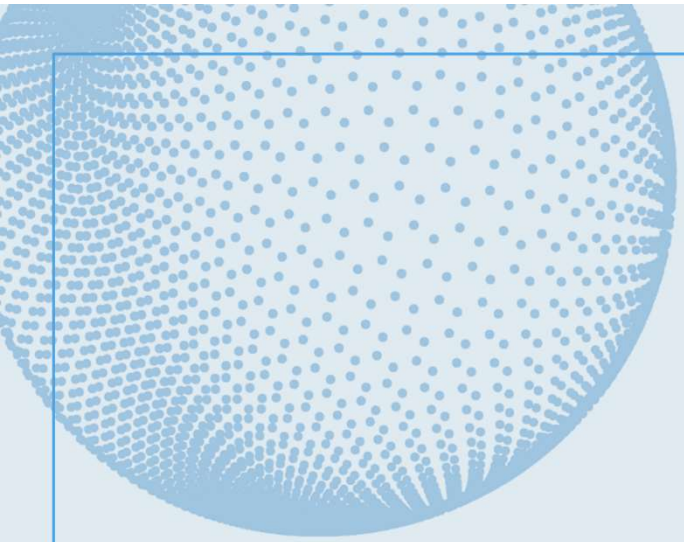
TECHNIQUES FOR HANDLING LARGE REFACTORINGS

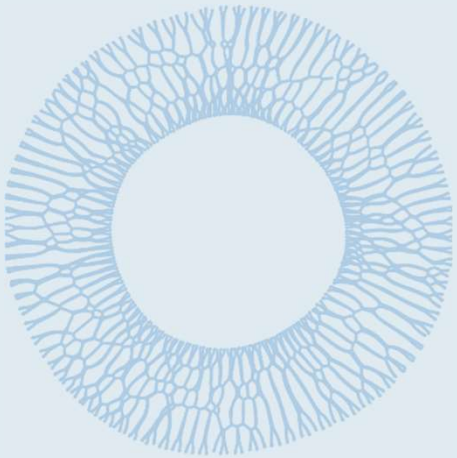


STANDING ON THE SHOULDERS OF GIANTS

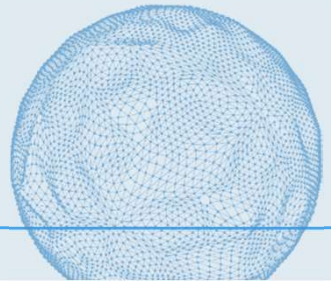
Fowler and Kerievsky both talk about large refactorings, but don't cover the details because they are too large.

- Fowler "Refactoring Improving the Design of Existing Code" 1999, 2018
- Kerievsky "Refactoring To Patterns" 2005
- Gamma et al. (Gang of Four) "Design Patterns: Elements of Reusable Object-Oriented Software" 1994
- Feathers "Working Effectively with Legacy Code" 2005





REFACTORING OVERVIEW



WHAT IS REFACTORING?

A disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

— Martin Fowler

- Refactoring lowers the cost of enhancements.
- Refactoring is a part of day-to-day programming.
- Automation is helpful but not essential.

REFACTORING WORKFLOW

Make it work

Make it right

Make it fast

WHY REFACTOR?

To make the next change to the code easier – which means *cheaper* and *faster*.

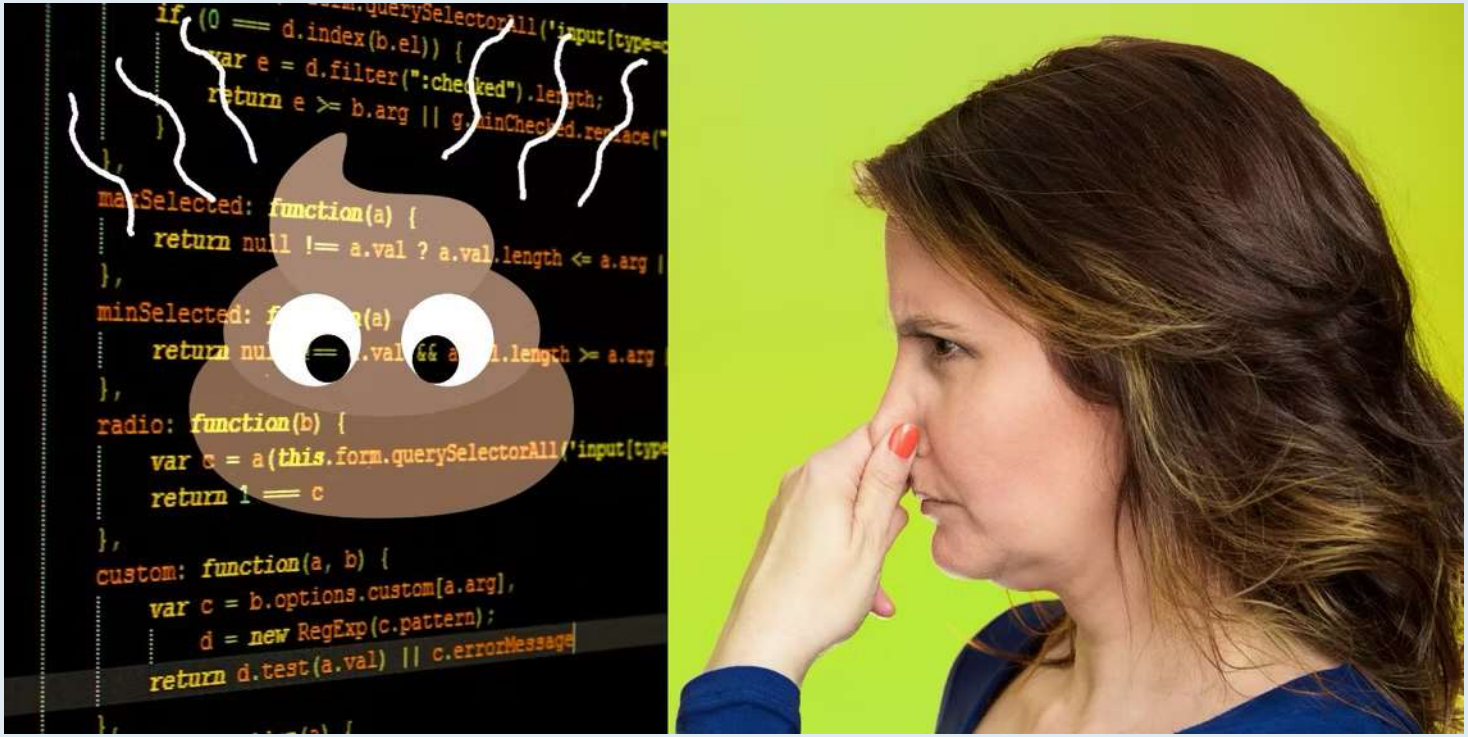
Refactoring reduces:

…Technical debt, whose interest is paid in time.

…Time spent understanding code.

…Time to restructure the code to support new functionality.

WHY REFACTOR?



SOME CODE SMELLS

Long Method
Large Class
Primitive Obsession
Long Parameter List
Data Clumps
Comments
Duplicate Code
Dead Code

Speculative Generality
Switch Statements
Feature Envy
Divergent Change
Shotgun Surgery
Inconsistent Names
Magic Numbers
Temporary Field

Lazy Class
Combinatorial Explosion
Conditional Complexity
Middle Man
Message Chains
Solution Sprawl
Type Embedded in Name
Indecent Exposure

CODE SMELLS I RUN INTO A LOT

If it Stinks, Change it

- Grandma Beck

CODE SMELLS

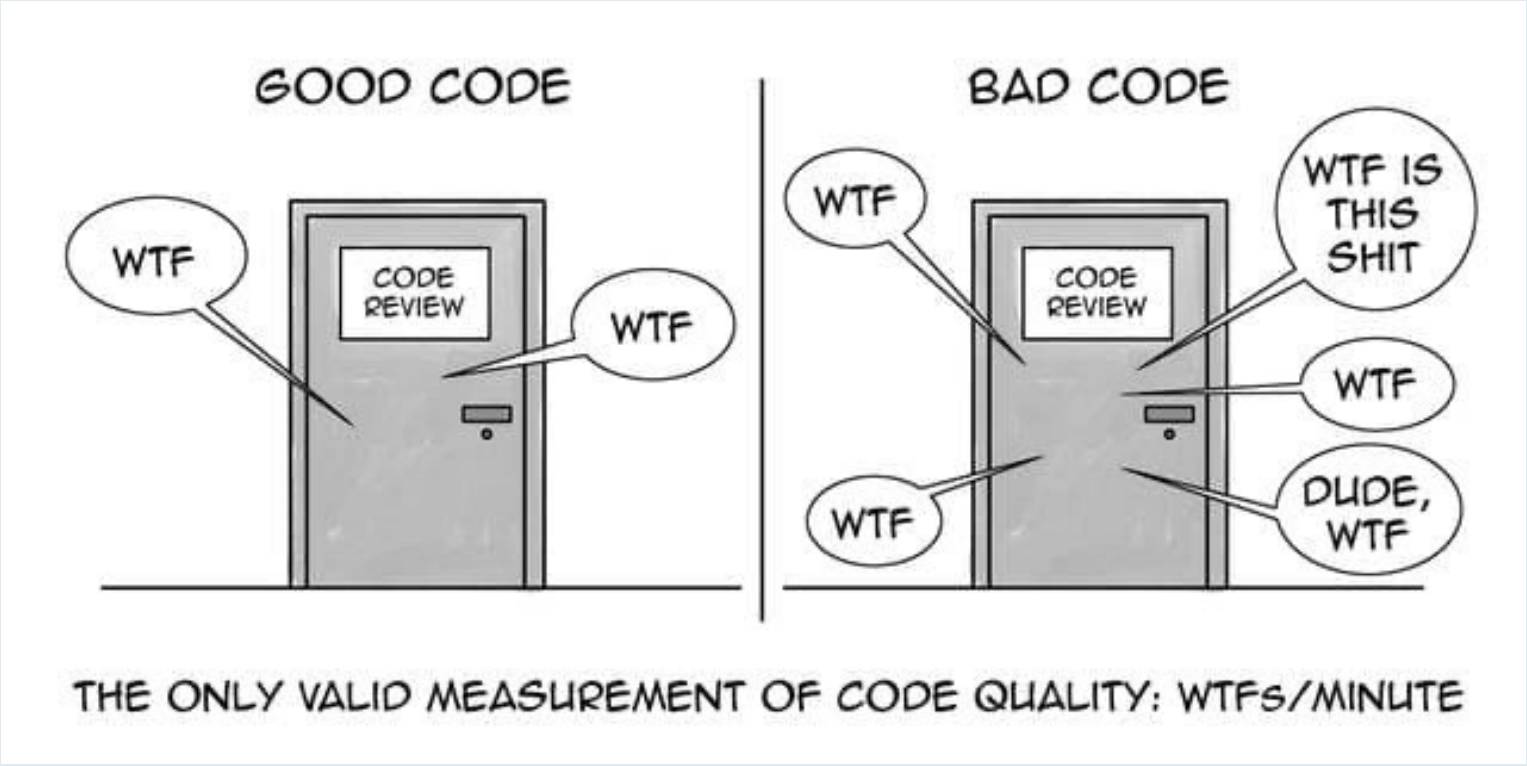
Long Methods

Large Classes

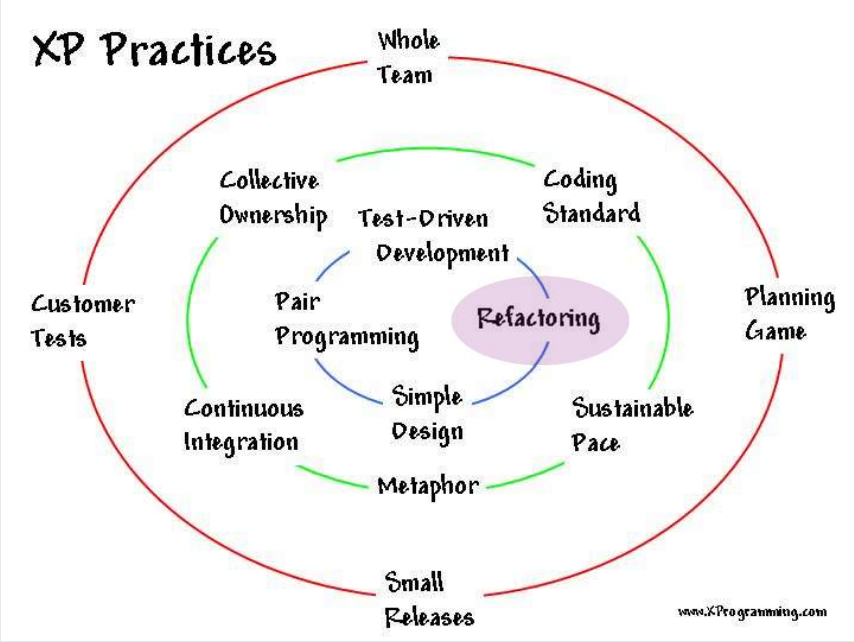
CODE SMELLS

Duplicate Code

WHY REFACTOR?



STANDARD AGILE PRACTICE



Scaled Agile Framework (SAFe)

11.4 Refactor to support new behavior of the code

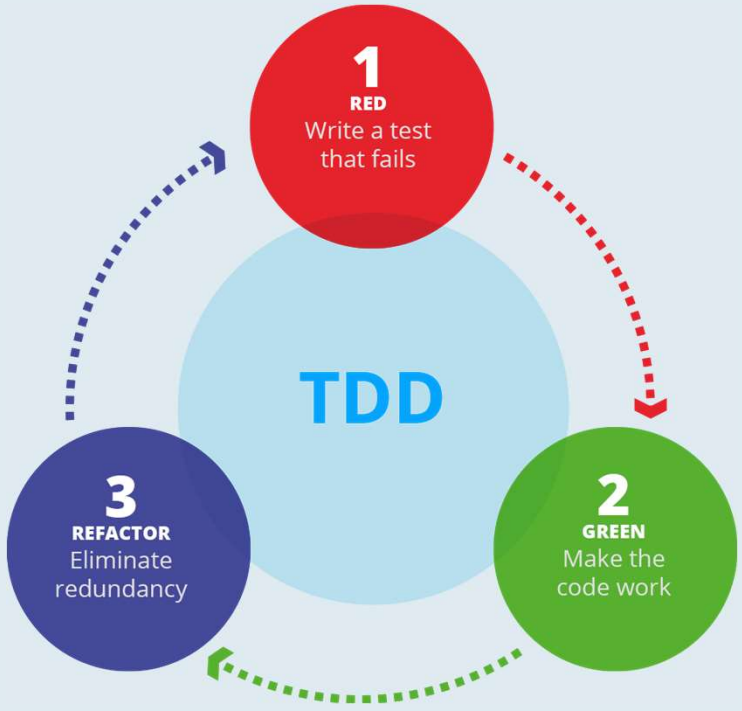
WHEN SHOULD WE REFACTOR?

As a part of day-to-day programming…

- You need to change the code for new requirements.
- A bug is hard to fix because the code is tangled.
- Scrum "chores," Technical tasks/enablers.

You need to reduce technical debt…

- Need to get code under test.
- SAFe IP Iteration or XP/Scrum hardening sprints.



HOW DO WE REFACTOR SAFELY?

Test coverage

- **Red-Green-Refactor**
  - Get the code under test
  - Run tests before and after refactoring

HOW DO WE REFACTOR SAFELY?

Test coverage

Start small and proceed incrementally

- **Red-Green-Refactor**
- **Make every change small**
  - Ensure no bugs before moving to next step

HOW DO WE REFACTOR SAFELY?

Test coverage

Start small and proceed incrementally

Prioritize

- **Red-Green-Refactor**
- **Make every change small**
- **Focus on the highest priority code**
  - Where new functionality is to be introduced.
  - Code that's hard to read or maintain.
  - Code that changes a lot.

HOW DO WE REFACTOR SAFELY?

Test coverage

Start small and proceed incrementally

Prioritize

Track Technical Debt

- **Red-Green-Refactor**
- **Make every change small**
- **Focus on the highest priority code**
- **Use a tool to identify and manage technical debt**
  - SonarQube
  - Veracode
  - Linters (Checkstyle, PMD, UCDetector)…

HOW DO WE REFACTOR SAFELY?

Test coverage

Start small and proceed incrementally

Prioritize

Track Technical Debt

Focus on Progress, not Perfection

- **Red-Green-Refactor**
- **Make every change small**
- **Focus on the highest priority code**
- **Use a tool to identify and manage technical debt**
- **“Always leave the campground cleaner than you found it.”**
  - If you touch it, refactor it.

A SMALL EXAMPLE

Shotgun Surgery:

- One change cascades through multiple classes and/or source files.

Adding new code would be easier if only…

EXAMPLE: BEFORE (A)

```
class Call {
  public boolean isCreditCheckEnabled() {
    return agentSessions.stream()
      .anyMatch(as -> as.isCreditCheckEnabled());
  }
}

class AgentSession {
  public boolean isCreditCheckEnabled() {
    return agentLocation
           .isCreditCheckEnabled(agentId);
  }
}

class AgentLocation {
  private Data localdata = …;
  public boolean isCreditCheckEnabled(String agentId) {
    return Handwavium.check(agentId, localdata, “CREDITCHECK”);
  }
}
```

EXAMPLE: A'

```
class Call {
  public boolean isCreditCheckEnabled() {
    return agentSessions.stream()
      .anyMatch(as -> as.isCreditCheckEnabled());
  }
}

class AgentSession {
  public boolean isCreditCheckEnabled() {
    return agentLocation
           .isCreditCheckEnabled(agentId);
  }
}

class AgentLocation {
  private Data localdata = …;
  public boolean isCreditCheckEnabled
                       (String agentId) {
    return Handwavium.check(agentId, localdata,
                            "CREDITCHECK");
  }
}
```

```
public enum FeatureFlag {
 CreditCheck;
}

class Call {
  public boolean isFeatureEnabled(FeatureFlag flag) {
    return agentSessions.stream()
      .anyMatch(as -> as.isFeatureEnabled(flag));
  }
}

class AgentSession {
  public boolean isFeatureEnabled(FeatureFlag flag) {
    return agentLocation
             .isFeatureEnabled(agentId, flag);
}

class AgentLocation {
  private Data localdata = …;
  public boolean isFeatureEnabled
                  (String agentId, FeatureFlag flag) {
    return Handwavium.check(agentId, localdata,
                            flag);
  }
}
```

EXAMPLE: B

Adding a new flag is now trivial

```
public enum FeatureFlag {
 CreditCheck, Payment;
}

class Call {
  public boolean isFeatureEnabled(FeatureFlag flag) {
    return agentSessions.stream()
      .anyMatch(as -> as.isFeatureEnabled(flag));
  }
}

class AgentSession {
  public boolean isFeatureEnabled(FeatureFlag flag) {
    return agentLocation
             .isFeatureEnabled(flag, agentId);
}

class AgentLocation {
  private Data localdata = …;
  public boolean isFeatureEnabled
                 (String agentId, FeatureFlag flag) {
    return Handwavium.check(agentId, localdata,
                            flag);
  }
}
```

MY PROBLEMS ARE BIGGER THAN THAT...

HOW TO MOVE FROM THE SIMPLE TO THE MORE COMPLEX

HOW TO TACKLE AN ACTUAL CODEBASE

Moving from Fowler to your own code is a giant step.

A PLAN TO MAKE A PLAN

Think of refactoring as a trip

1. Decide the Trip is Necessary
2. Understand the Landscape
3. Select a Destination
4. Chart Your Course
5. Make It So

DECIDE THE TRIP IS NECESSARY

OVERCOMING (STATIONARY) INERTIA

The Journey of a thousand miles begins with a single step.

- Laozi

A challenge not a chore

Refactor in anger.

UNDERSTAND THE LANDSCAPE

Scratch refactoring from Michael Feathers "Working with Legacy Code" 2005

- The goal is to get familiar with the code, not to change it.
- "The only rule is to revert your changes when you're done."
- After figuring out what you want to do, move on to next step.

SELECT A DESTINATION

Knowing where to head is the first half of the battle.

CHART YOUR COURSE

Once you have decided what the change should become, write out **a refactoring plan**

A Series of lower level refactorings or other small changes.

MAKE IT SO

We will get to this in the Example

EXAMPLE REFACTORING

REPLACE METHOD AND INTERFACE WITH TEMPLATE PATTERN

THE PROBLEM

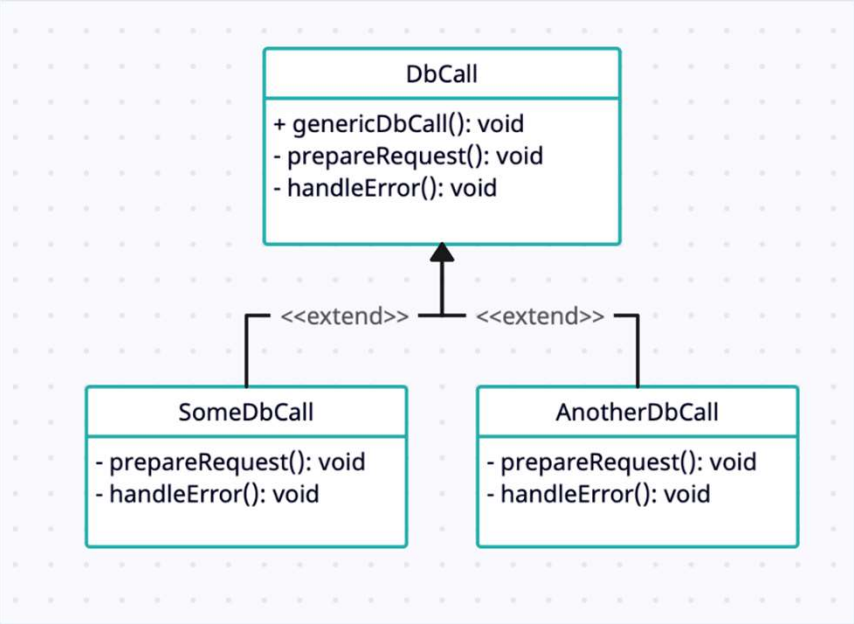
My evil twin wrote some code

https://stackoverflow.com/questions/28079920/how-to-refactor-this-template-method-implemetation

```
public static void genericDbCall(
            RecordDto record,
            DatabaseInterface di){
  try {//begin database transaction
    //some logic
    di.prepareRequest(record);
    //more computation
    //database commit();
    di.handleError(record.getId());
  }
  finally {
    // rollback();
  }
}
```

TEMPLATE METHOD

```
public static void genericDbCall(
            RecordDto record,
            DatabaseInterface di){
  try {//begin database transaction
    //some logic
    di.prepareRequest(record);
    //more computation
    //database commit();
    di.handleError(record.getId());
  }
  finally {
    // rollback();
  }
}
```



REFACTORING PLAN: HIGH LEVEL

My revelation:

I already have the subclasses from the existing implementations.

A. Make a base class
B. Turn the implementations into subclasses of the new base class
C. Remove the interface
D. Stop parameter passing

REFINING THE ROUTE

We need a base class for the template method.

- **Method object is a way to make the static method into a class.**

1. Method Object
2. Inline Method

REFINING THE ROUTE

We need a base class for the template method.

Template method patterns have subclasses.

- **Add the implementations as subclasses following the pattern**

3. Make Implementations a Child of MO
4. Remove use of Interface
5. Repeat 3 & 4 for other implementations
6. Make Base Class Abstract

REFINING THE ROUTE

We need a base class for the template method.

Template methods have subclasses.

The methods of the interface have to move before it can be deleted

- **I chose to copy paste the abstract methods rather than have the base implement the interface**

7. Use Self Instead of Interface
8. Drop the Interface
9. Stop Implementing Interface
10. Delete Interface

REFINING THE ROUTE

We need a base class for the template method.

Template methods have subclasses.

The methods of the interface have to move before it can be deleted

Clean up parameters and references.

- **Remove parameter from Fowler comes in handy**

11. Visibility of domain from base class
12. Use DTO from base class
13. Delete redundant parameter

REFACTORING PLAN: DETAIL

1. Method Object
2. Inline Method

---

3. Make Implementations a Child of MO
4. Remove use of Interface
5. Repeat 3 & 4 for other implementations
6. Make Base Class Abstract

---

7. Use Self Instead of Interface
8. Drop the Interface
9. Stop Implementing Interface
10. Delete Interface

---

11. Visibility of domain from base class
12. Use DTO from base class
13. Delete redundant parameter

METHOD OBJECT: BEFORE

```
public class UtilClass {
  public static void genericDbCall(
      RecordDto record,
      DatabaseInterface di) {

    try {
      di.prepareRequest(record);
      di.handleError(record.getId());
    }
    finally {…}
  }
}
```

METHOD OBJECT

```
public class UtilClass {
  public static void genericDbCall(
      RecordDto record,
      DatabaseInterface di) {

    new GenericDbCall(record).process(di);
  }
}
```

```
public class GenericDbCall {
  private final RecordDto record;

  public GenericDbCall(RecordDto record){
    this.record = record;
  }

  public void process(
                DatabaseInterface di){
    try{
      di.prepareRequest(record);
      di.handleError(record.getId());
    }
    finally{…}
  }
}
```

INLINE METHOD: BEFORE

```
public class Caller {
  public static void main(String[] args)
  {
    RecordDto record =
      new RecordDto();
    DatabaseInterface di =
      new InsertImplementation();

    UtilClass.genericDbCall(record, di);
  }
}
```

INLINE METHOD

```
public class Caller {
  public static void main(String[] args)
  {
    RecordDto record =
      new RecordDto();
    DatabaseInterface di =
      new InsertImplementation();

    UtilClass.genericDbCall(record, di);
  }
}
```

```
public class Caller {
  public static void main(String[] args)
  {
    RecordDto record =
      new RecordDto();
    DatabaseInterface di =
      new InsertImplementation();

    new GenericDbCall(record).process(di);
  }
}
```

MAKE IMPLEMENTATION A CHILD OF MO

```
public class InsertImplementation
      implements DatabaseInterface {
…
}
```

```
public class Caller {
  public static void main(String[] args) {
    RecordDto record = new RecordDto();
    DatabaseInterface di =
      new InsertImplementation();
…
  }
}
```

```
public class InsertImplementation
      extends GenericDbCall
      implements DatabaseInterface {
  public InsertImplementation(RecordDto record) {
    super(record);
  }
…
}
```

```
public class Caller {
  public static void main(String[] args) {
    RecordDto record = new RecordDto();
    DatabaseInterface di =
      new InsertImplementation(record);
…
  }
}
```

REMOVE USE OF INTERFACE: BEFORE

```
public class Caller {
  public static void main(String[] args) {
    RecordDto record = new RecordDto();
    DatabaseInterface di =
      new InsertImplementation(record);

    new GenericDbCall(record).process(di);
  }
}
```

REMOVE USE OF INTERFACE

```
public class Caller {
  public static void main(String[] args) {
    RecordDto record = new RecordDto();
    DatabaseInterface di =
      new InsertImplementation(record);

    new GenericDbCall(record).process(di);
  }
}
```

```
public class Caller {
  public static void main(String[] args) {
    RecordDto record = new RecordDto();
    InsertImplementation di =
      new InsertImplementation(record);

    di.process(di);
  }
}
```

REPEAT PRIOR 2 STEPS

Do the same thing for each implementation of the interface.

MAKE BASE CLASS ABSTRACT

```
public interface DatabaseInterface {
  void prepareRequest(RecordDto record);
  void handleError(String id);
}

public class GenericDbCall {
…
}
```

```
public abstract class GenericDbCall {
  abstract void prepareRequest(RecordDto
record);
  abstract void handleError(String id);
…
}
```

USE SELF INSTEAD OF INTERFACE: BEFORE

```
public abstract class GenericDbCall {
 abstract void prepareRequest(RecordDto record);
 abstract void handleError(String id);
…
  public void process(DatabaseInterface di){
    try{
      di.prepareRequest(record);
      di.handleError(record.getId());
    }
    finally{}
  }
}
```

USE SELF INSTEAD OF INTERFACE

```
public abstract class GenericDbCall {
 abstract void prepareRequest(RecordDto record);
 abstract void handleError(String id);
…
  public void process(DatabaseInterface di){
    try{
      di.prepareRequest(record);
      di.handleError(record.getId());
    }
    finally{}
  }
}
```

```
public abstract class GenericDbCall {
 abstract void prepareRequest(RecordDto record
 abstract void handleError(String id);
…
  public void process(DatabaseInterface di){
    try{
      prepareRequest(record);
      handleError(record.getId());
    }
    finally{}
  }
}
```

DROP THE INTERFACE: BEFORE

```
public abstract class GenericDbCall {
…
  public void process(DatabaseInterface di){
    try{
      prepareRequest(record);
      handleError(record.getId());
    }
    finally{}
  }
}
```

DROP THE INTERFACE

```
public abstract class GenericDbCall {
…
  public void process(DatabaseInterface di){
    try{
      prepareRequest(record);
      handleError(record.getId());
    }
    finally{}
  }
}
```

```
public abstract class GenericDbCall {
…
  public void process(){
    try{
      prepareRequest(record);
      handleError(record.getId());
    }
    finally{}
  }
}
```

STOP IMPLEMENTING INTERFACE: BEFORE

```
public class InsertImplementation
      extends GenericDbCall
      implements DatabaseInterface {
…
}
```

STOP IMPLEMENTING INTERFACE

```
public class InsertImplementation
      extends GenericDbCall
      implements DatabaseInterface {
…
}
```

```
public class InsertImplementation
      extends GenericDbCall {
…
}
```

9.5 Repeat for All Implementations

DELETE INTERFACE

```
public interface DatabaseInterface
{
   void prepareRequest(RecordDto record);
   void handleError(String id);
}
```

VISIBILITY OF DOMAIN FROM BASE CLASS

```
public abstract class GenericDbCall {
  private final RecordDto record;
…
}
```

```
public abstract class GenericDbCall {
  protected final RecordDto record;
…
}
```

USE DTO FROM BASE CLASS: BEFORE

```
public class InsertImplementation extends
GenericDbCall{
…
  @Override
  public void prepareRequest(RecordDto param) {
    param.getId();
    param.getSomeOtherField();
  }
  @Override
  public void handleError(String id) {
    System.out.print(id);
  }
}
```

USE DTO FROM BASE CLASS

```
public class InsertImplementation extends
GenericDbCall{
…
  @Override
  public void prepareRequest(RecordDto param) {
    param.getId();
    param.getSomeOtherField();
  }
  @Override
  public void handleError(String id) {
    System.out.print(id);
  }
}
```

```
public abstract class GenericDbCall {
  protected final RecordDto record;
…
}
```

```
public class InsertImplementation extends
GenericDbCall{
…
  @Override
  public void prepareRequest(RecordDto param){
    record.getId();
    record.getSomeOtherField();
  }
  @Override
  public void handleError(String id) {
    System.out.print(record.getId());
  }
}
```

DELETE REDUNDANT PARAMETER: BEFORE

```
public class InsertImplementation extends
GenericDbCall{
…
  @Override
  public void prepareRequest(RecordDto param){
    record.getId();
    record.getSomeOtherField();
  }
  @Override
  public void handleError(String id) {
    System.out.print(record.getId());
  }
}
```

DELETE REDUNDANT PARAMETER

```
public class InsertImplementation extends
GenericDbCall{
…
  @Override
  public void prepareRequest(RecordDto param){
    record.getId();
    record.getSomeOtherField();
  }
  @Override
  public void handleError(String id) {
    System.out.print(record.getId());
  }
}
```

```
public class InsertImplementation extends
GenericDbCall{
…
  @Override
  public void prepareRequest(){
    record.getId();
    record.getSomeOtherField();
  }
  @Override
  public void handleError() {
    System.out.print(record.getId());
  }
}
```

FINISHED CODE: USE

```
public class Caller {
  public static void main(String[] args) {
    RecordDto record = new RecordDto();

    InsertImplementation di = new InsertImplementation(record);
    di.process();
  }
}
```

FINISHED CODE: BASE CLASS

```
public abstract class GenericDbCall {
  protected final RecordDto record;

  public GenericDbCall(RecordDto record) {
    this.record = record;
  }

  abstract void prepareRequest();
  abstract void handleError();

  public void process(){
    try {
      prepareRequest();
      …
      handleError();
    }
    finally {}
  }
}
```

FINISHED CODE: IMPLEMENTATION

```
public class InsertImplementation extends GenericDbCall{

  public InsertImplementation(RecordDto record) {
    super(record);
  }

  @Override
  public void prepareRequest() {
    record.getId();
    record.getId();
  }

  @Override
  public void handleError() {
    System.out.print(record.getId());
  }
}
```

HAL JUST WANTS TO HELP

AUTOMATED REFACTORING

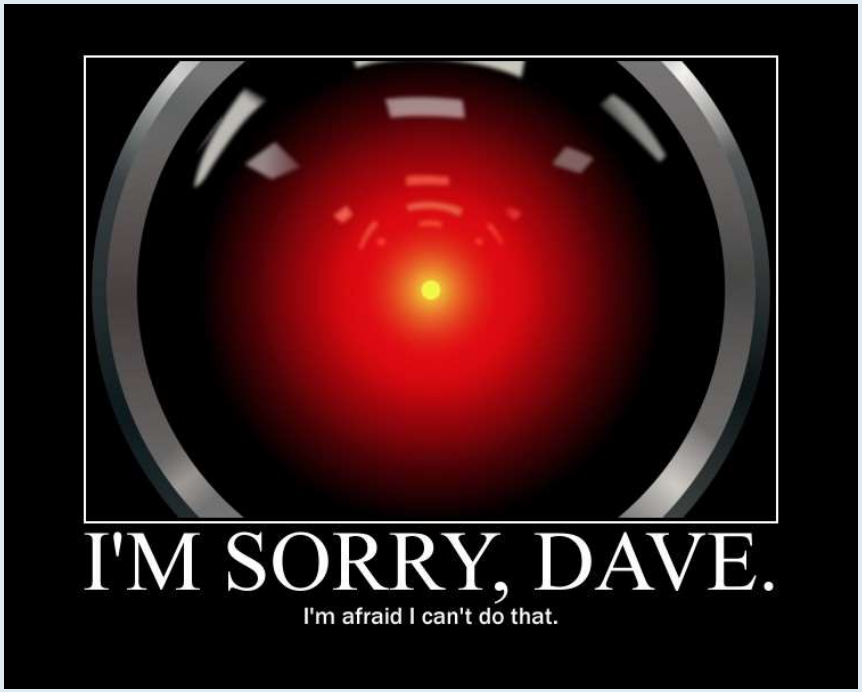
Use them, they are safe. Mostly.

COPILOT

For GenAI refactors, keep a human in the loop to guard against hallucinations.

Your milage may vary.

- √ Give it tests and ask it for code that passes the tests.
- ? Generate tests for existing code.
- X Ask it to refactor existing code.



SUMMARY

REFACTORING JOURNEY

1. **Decide that the trip is necessary**

- Hard to change code
- WTFs/minute
- Code smells

REFACTORING JOURNEY

1. Decide that a trip is necessary
2. **Understand the landscape**

- Talk to the original developer
- Develop your own code reading skills
- Scratch refactoring

REFACTORING JOURNEY

1. Decide that the trip is necessary
2. Understand the landscape
3. **Select a destination**

- What design would make the next change easier while also being easy to understand?

REFACTORING JOURNEY

1. Decide that the trip is necessary
2. Understand the landscape
3. Select a destination
4. **Chart your course**

- Each step is a simple, standalone, refactor that brings you closer to the code you want
- Write it down

REFACTORING JOURNEY

1. Decide that the trip is necessary
2. Understand the landscape
3. Select a destination
4. Chart your course
5. **Make it so**

- Get the current code under test
- Run the tests and verify they pass
- Make a small change/refactor
- Verify the tests still pass
- Commit
- Make the next small change/refactor
- Repeat until done

HOW DO YOU GET TO CARNEGIE HALL?

Practice, Practice, Practice

HOW DO YOU GET TO CARNEGIE HALL?

Practice, Practice, Practice

...or just take the tour.

BUILDING YOUR KNOWLEDGE

Learn from others

- Books (see bibliography)
- Online resources
  - https://refactoring.guru/
  - Refactoring | Baeldung on Computer Science
  - Stackoverflow

BUILDING YOUR EXPERIENCE

Learn by doing

- Practice on your own code; start simple
  - Rename method/variable
  - Extract method/variable
  - Hide method
- Practice with katas
  - https://kata-log.rocks/refactoring
  - https://github.com/emilybache

BIBLIOGRAPHY

Refactoring: Improving the Design of Existing Code, Martin Fowler, Kent Beck (1999, 2018)

Refactoring to Patterns, Joshua Kerievsky (2004)

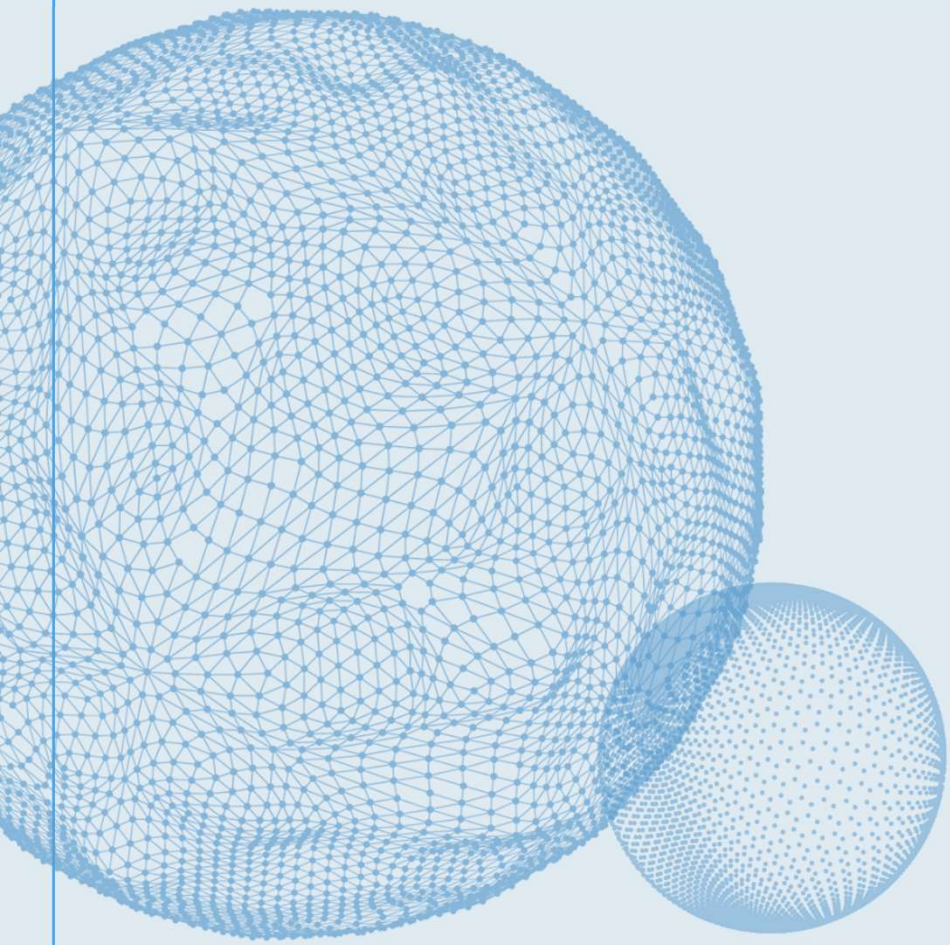
Working Effectively with Legacy Code, Michael Feathers, (2004)

Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, et al (1994)

Clean Code: A Handbook of Software Agile Craftsmanship, Robert Martin, (2008)

Slides:

https://tinyurl.com/2wwmjxj4



THANK YOU

AARON MCCLENNEN,

AARON_MCCLENNEN@YAHOO.COM

M. JEFF WILSON, @MJEFFW