Coding Standards and Practices

QA

Learn. To Change.

The Virtues of a Programmer

Laziness

- Hates answering the same questions over and over, so writes good documentation 🙋‍♂️ ➡️ 📃
- Hates reading documentation, so writes code clearly
- Writes tools and utilities to make the computer do all the work

The Virtues of a Programmer

Impatience

- Hates a computer that is lazy –
  an impatient programmer's code anticipates a need

Hubris

- Has pride in programs that no one will criticise

The golden rule

Follow the standards of your organisation

- Ask to see the coding standards / guidelines

- **Be sure your code always uses:**
  - Consistent **naming**
  - Effective **commenting** // comments
  - Proper and effective code **formatting**

Good practice

1. Remember ‘Rubbish in – rubbish out’. 🚮 👩‍💻 🚮

2. Choose the smallest data type for the job.
   Remember floating point issues.

3. Use constants, not literal numbers, where possible.

   ```
   const double VAT = 0.2;
   ```

4. Create variables with the shortest scope and lifetime.

Good practice

7. Make sure that objects are:
   allocated as late as possible
   and release as soon as possible.

8. Use variables for one purpose only.

```
int x = 2;  // month 2 – Feb
x = 21;     // Age = 21
```

9. Functions should only perform one task.

10. Classes must only have a single responsibility

11. Write unit tests!

Naming conventions

Naming conventions makes code easier to read:

- Easier to distinguish your items from those of a 3rd party

Follow the naming rules:

- For variables, structures, classes, data items, constants
- For code fragments - functions, methods, libraries
  - Avoid vague names like `calculate`
    be more specific `calculate_invoice_total`

Comments

- Use comments to show your code's intension

- Don't comment what is self evident

  ```
  // Calculate the VAT amount
  double vatAmount = price * VAT;
  ```

- Comment work-arounds and quick fixes (Hacks!)

- Update comments when code changes

- Remove unnecessary **comments** from **scripts** before release

Formatting

Good formatting makes code clearer!

- Format your code to allow your code to naturally flow
- Languages like Python use indentation to specify blocks
- C Based languages use braces

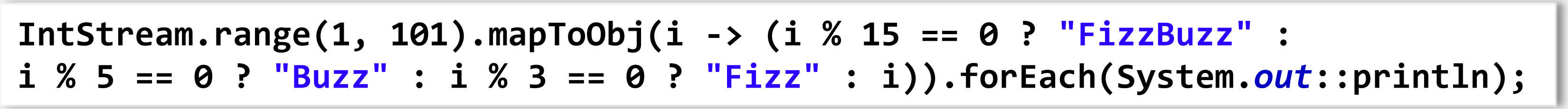
```
foreach (string name in names)
{
    if (name.length < 5)
    {
        throw new Exception("Name is too short!");
    }
}
```

Readability and style

- More time is spent on maintenance than development
  - Document what you do
  - Code that is obvious today is not obvious tomorrow

- Avoid 'clever' one-liners
  - They are rarely faster, sometimes slower
  - Often difficult to debug

KISS - Keep It Simple and Straightforward



```
IntStream.range(1, 101).mapToObj(i -> (i % 15 == 0 ? "FizzBuzz" :
i % 5 == 0 ? "Buzz" : i % 3 == 0 ? "Fizz" : i)).forEach(System.out::println);
```

Error handling

- **If anything can go wrong - it will**
  - Specifically test error conditions
- **If an opportunity exists to test for an error** - take it
- **Always report the error to an expected location**
  - Log errors to a repo that is available to dev team
  - Users don't need the full story

Programming for change

- **Defensive programming**
  - Where changes are less likely to cause problems
- **Making your program flexible**
  - Avoid artificial fixed limits
    no assumptions on variable type sizes, word length, etc.
  - Adhere to your company standards

All this makes changing a program less work

- Great for the virtue of laziness!

Help!

- **Many languages have style checkers and rules**
  - Python has PEP 008 rules
  - Perl has perlstyle documentation and PerlCritic
  - Java has Sun's code conventions
  - Google Style Guides are available for several languages

- **Code analysis tools are often based on Lint**
- **Some editors and IDEs assist with style as you type**
  - Use them to improve your productivity

See also
Uncle Bob Martin's site http://cleancoder.com

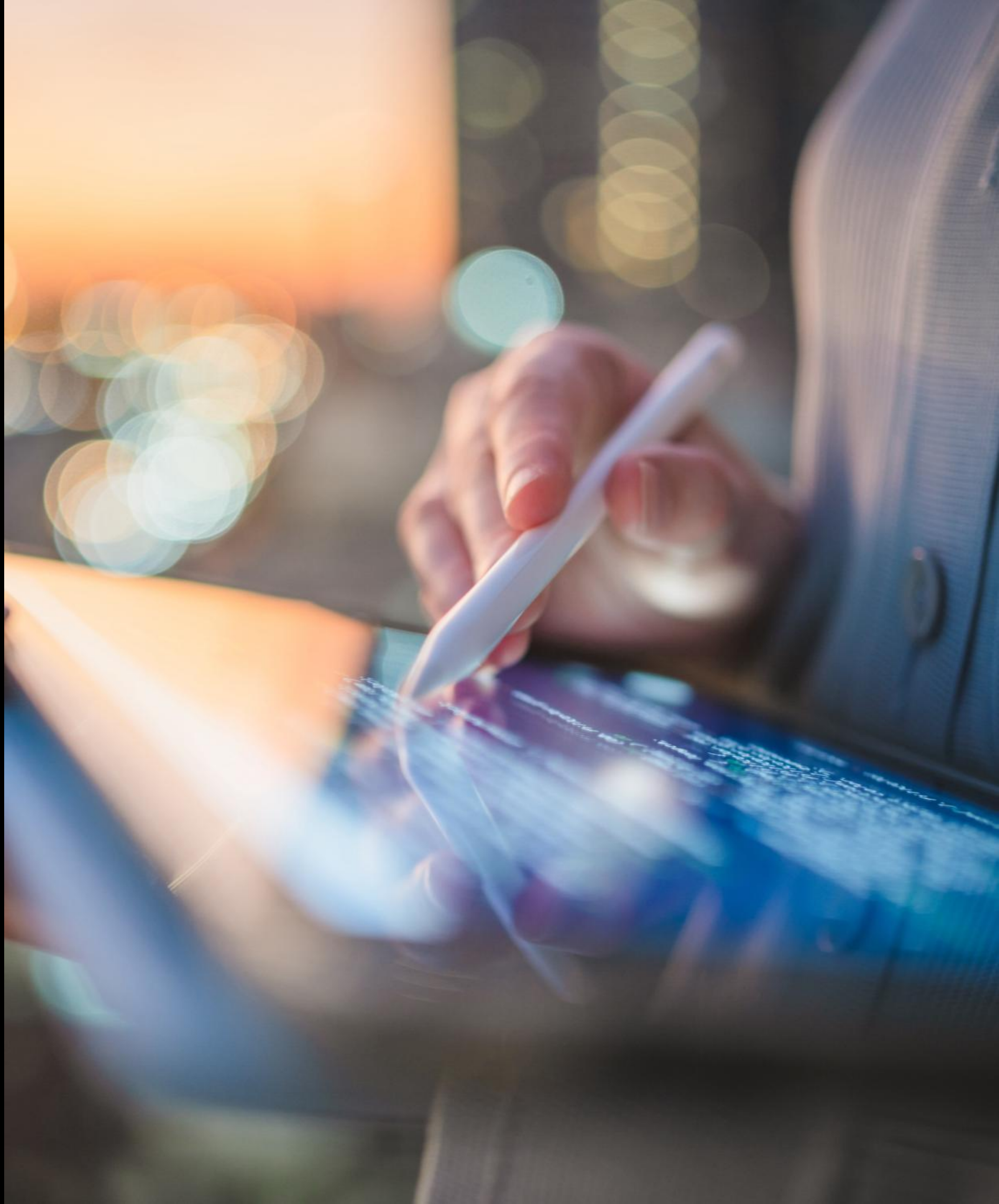
Investigate
After the course

1. Obtain the **Google Style Guide** for the language your code is written in.

2. Consider if any of the points in the Style Guide conflict with any practices commonly followed where you work.

Refactoring With Existing Tests

Learn. To Change.



What is Refactoring?

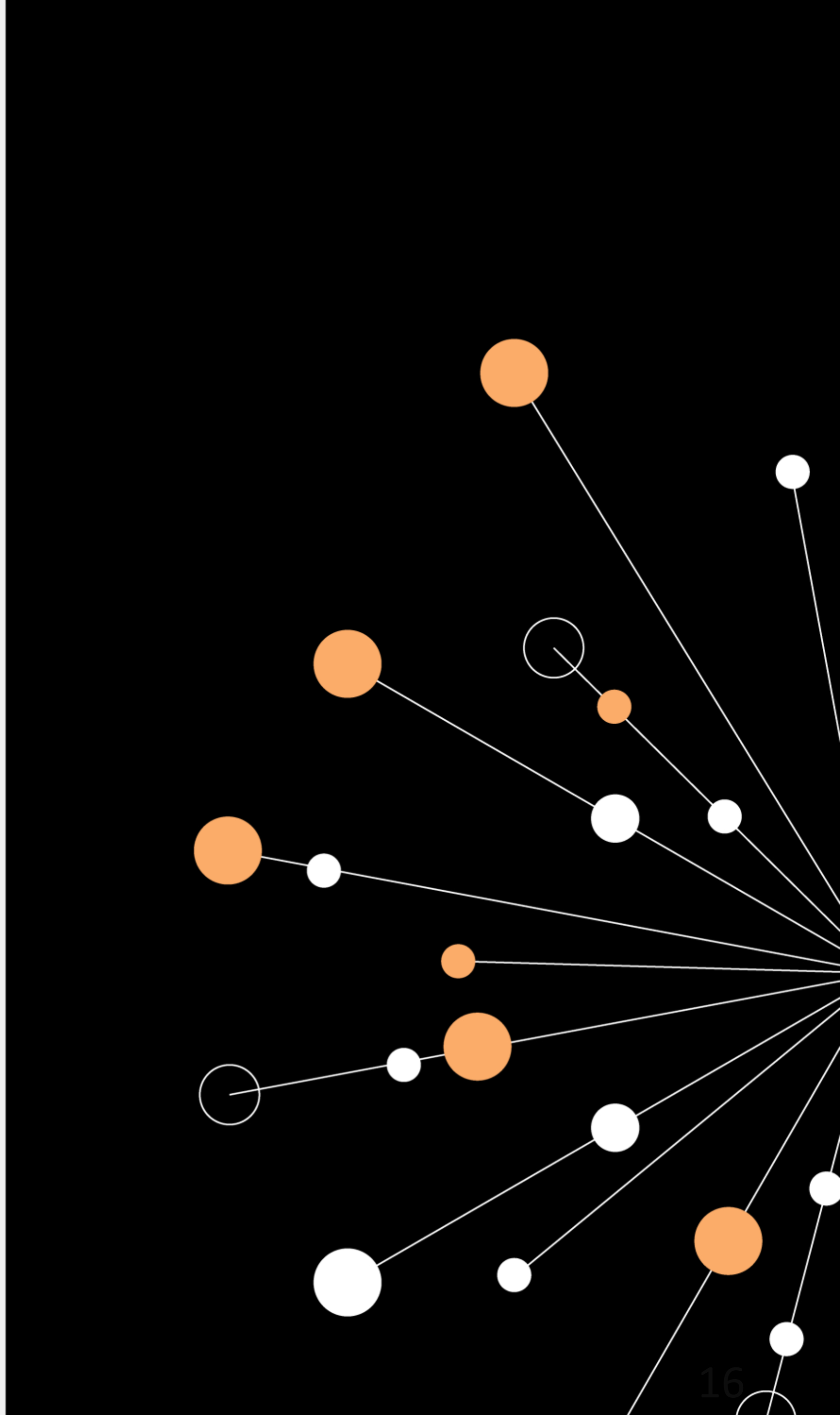
Is the process of improving the **structure**, **readability**, and **maintainability** of existing code *without changing its external behaviour*

Removing duplicate code

Improving code standards

Simplifying design logic

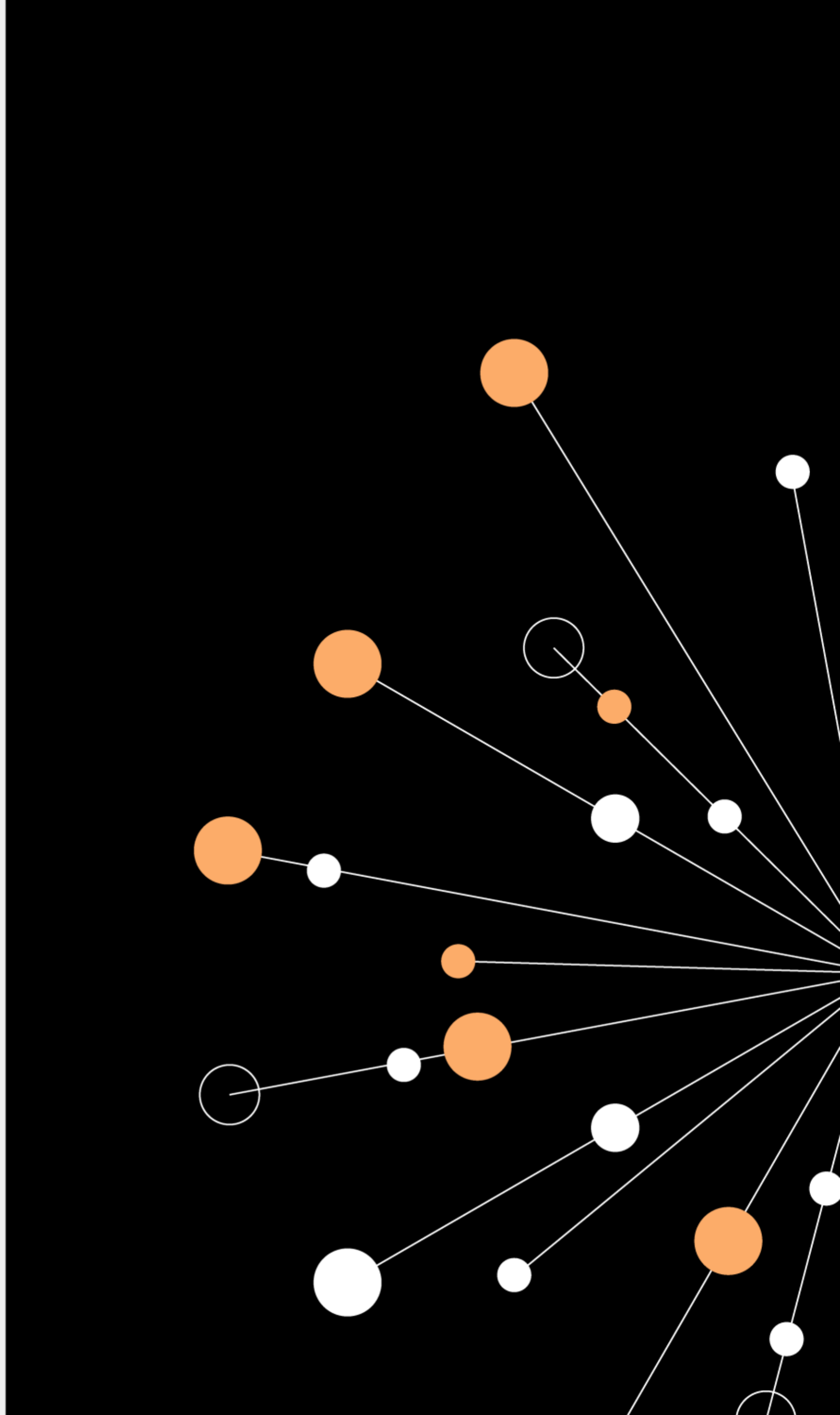
Reorganise code for efficiency



A few refactoring considerations

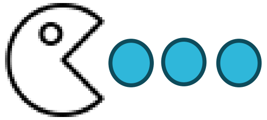
- Put repeated logic into reusable methods & classes
- Make **if** statements and **loops** clearer and readable.
- **Encapsulate Fields** – Restricting direct access
- Break Large Classes – Create new packages

Often done before adding new features or fixing bugs to make future work easier.



Principles of Refactoring

1. **Keep it small.**
   - Refactor in small doses to minimise overhead.



2. **Business catalysts.**
   - Refactor at the right time for your organisational needs, not just when the team feels like it!

3. **Team cohesion.**
   Everyone agrees on the refactoring goals
   like keeping the code readable after refactoring

4. **Transparency.**
   - Be completely open with stakeholders about the costs involved

Benefits of Refactoring

Makes code easier to understand

Improves code maintainability

Increases quality and robustness

Makes code more reusable

Improves the design

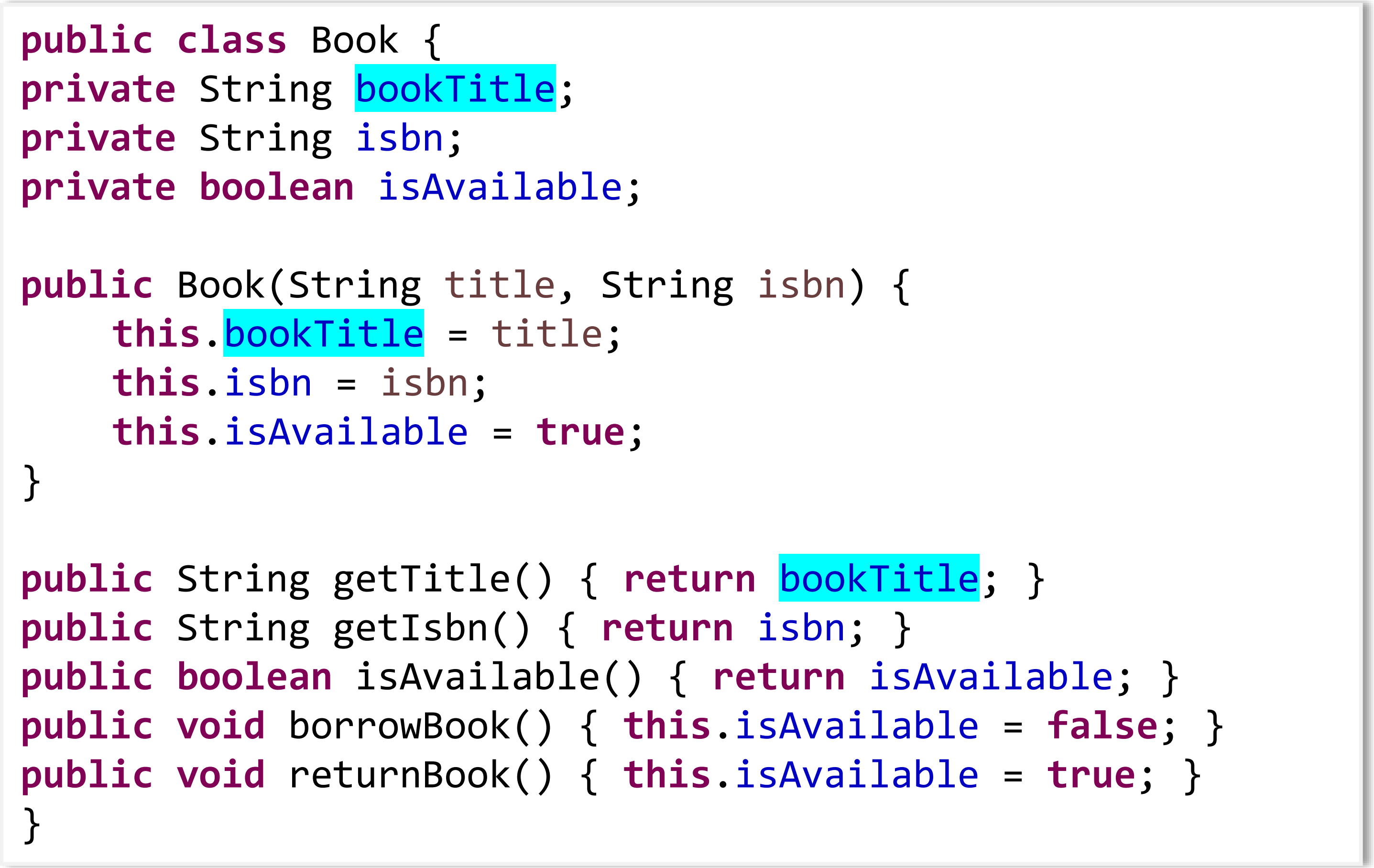
Easier to find bugs as code is cleaner

Refactoring ≠ Rewriting
Goal: Improve *internal structure*, readability, maintainability.

Common Refactorings:

Renaming

Repeat **<ALT> <SHIFT> R** (Eclipse) until you're satisfied you have an identifier that best reflects what the item represents.



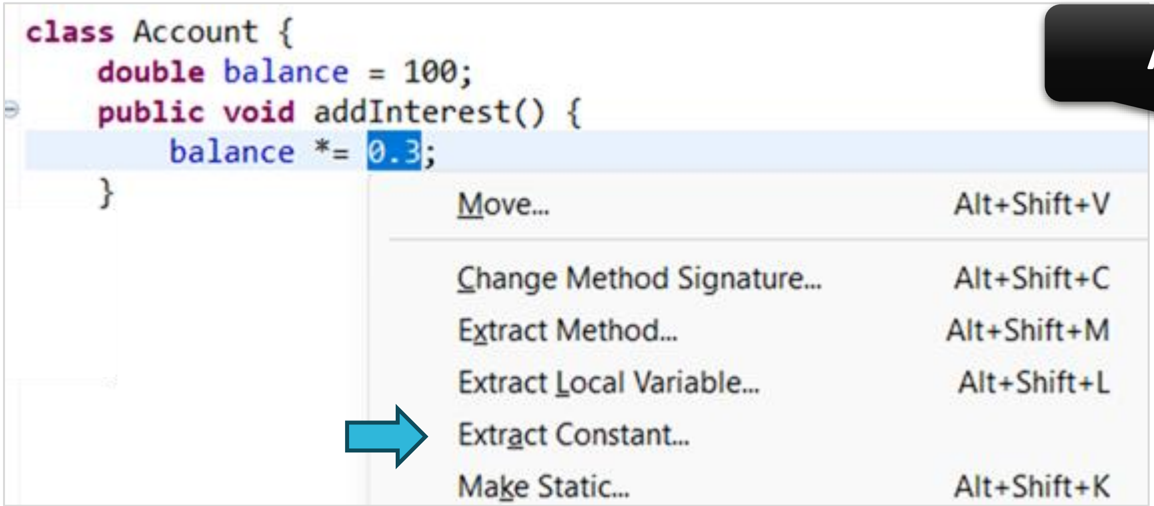
```java
public class Book {
private String bookTitle;
private String isbn;
private boolean isAvailable;

public Book(String title, String isbn) {
    this.bookTitle = title;
    this.isbn = isbn;
    this.isAvailable = true;
}

public String getTitle() { return bookTitle; }
public String getIsbn() { return isbn; }
public boolean isAvailable() { return isAvailable; }
public void borrowBook() { this.isAvailable = false; }
public void returnBook() { this.isAvailable = true; }
}
```

Common Refactoring:

Extract Constant



```
class Account {
    double balance = 100;
    public void addInterest() {
        balance *= 0.3;
    }
```

Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Extract Method... Alt+Shift+M
Extract Local Variable... Alt+Shift+L
Extract Constant...
Make Static... Alt+Shift+K

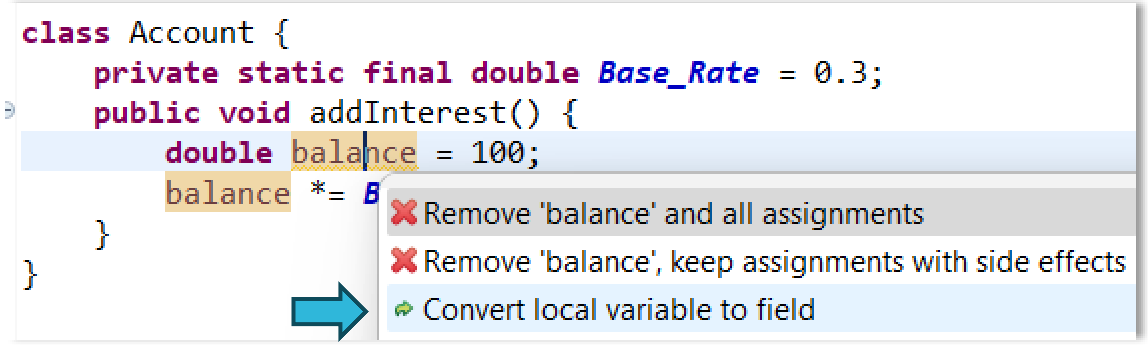
Alt-Shift-T

```
class Account {
        private static final double Base_Rate = 0.3;
        double balance = 100;
        public void addInterest() {
                balance *= Base_Rate;
        }
```

Common Refactorings:

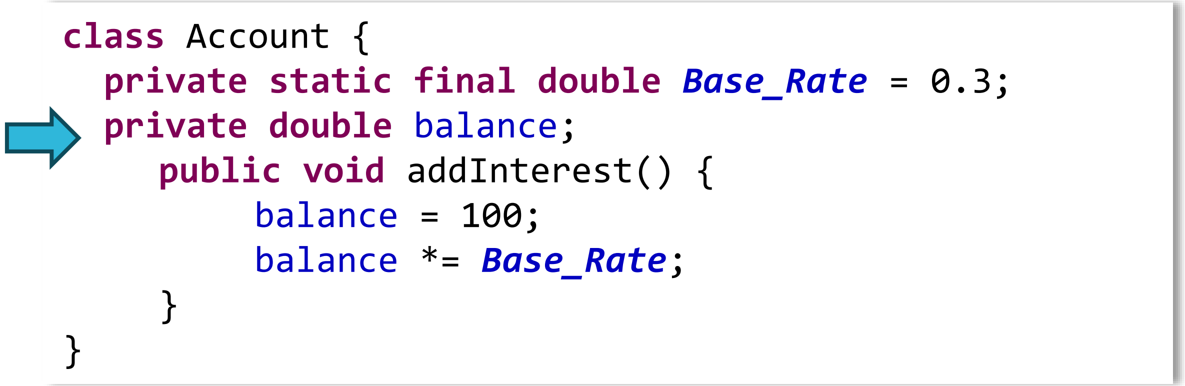
Extract to field

- Highlight literal (e.g. int or String), then **Ctrl-1**



```
class Account {
    private static final double Base_Rate = 0.3;
    public void addInterest() {
        double balance = 100;
        balance *= B
    }
}
```

Remove 'balance' and all assignments
Remove 'balance', keep assignments with side effects
Convert local variable to field

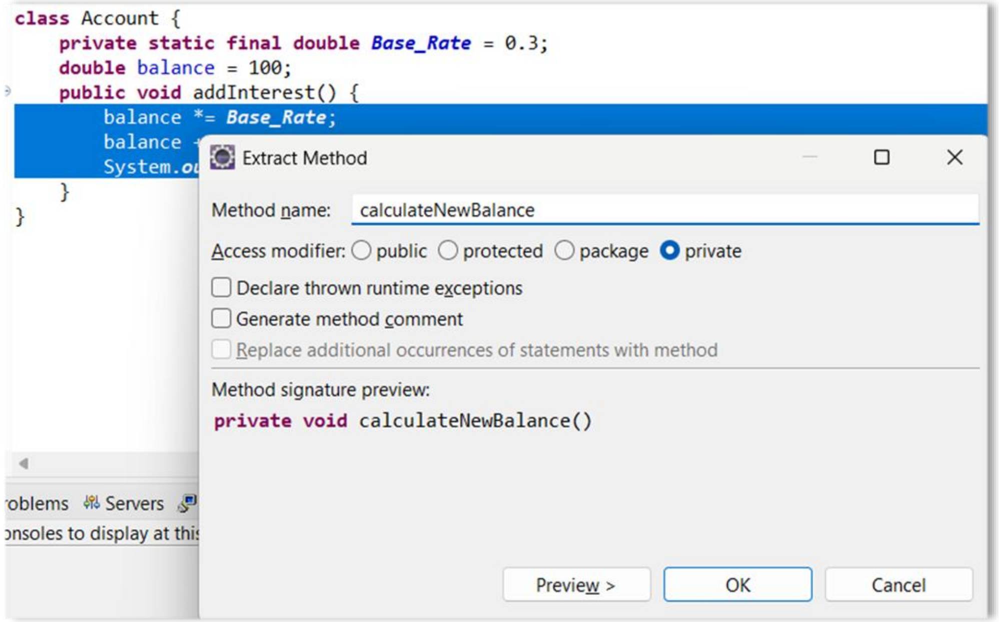


```
class Account {
  private static final double Base_Rate = 0.3;
  private double balance;
     public void addInterest() {
          balance = 100;
          balance *= Base_Rate;
     }
}
```

Common Refactoring:

Extract Method

- Eclipse **`<ALT> <SHIFT> M`** `(extract to method)`
- Remove code duplication
- **Break up** long methods
- **Clarity**: move lines to a method which expresses the intent
  - Code becomes **self-documenting**; much better than comments



```
class Account {
    private static final double Base_Rate = 0.3;
    double balance = 100;
    public void addInterest() {
        balance *= Base_Rate;
        balance +
        System.ou
    }
}
```

Extract Method

Method name: calculateNewBalance

Access modifier: ○ public ○ protected ○ package ● private

□ Declare thrown runtime exceptions

□ Generate method comment

□ Replace additional occurrences of statements with method

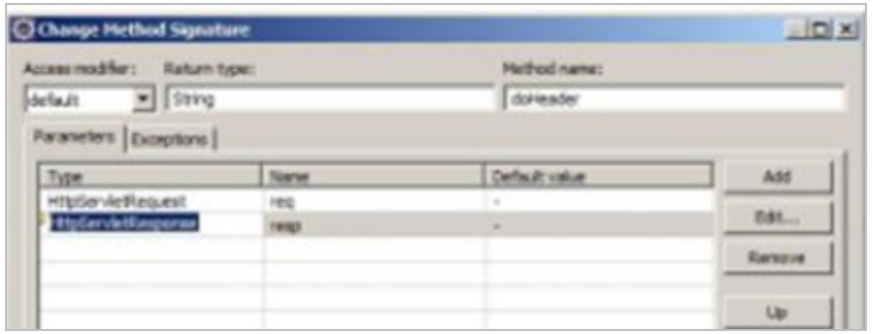
Method signature preview:

```
private void calculateNewBalance()
```

Common Refactoring:

Extract Method for Testability

1. Enter new method name
2. Check accessibility.
3. If necessary:
   - Get method code to compute return value (not void methods)
   - Use the return value and adjust the code calling the method
4. Change method signature (name and parameters)
5. Make the new method is cohesive (does one clear, focused task)



Access modifier: default

Return type: String

Method name: doHeader

Type	Name	Default value
HttpServletRequest	req	-
HttpServletResponse	resp	-

Common Refactorings:

Extract Class

Break a large class into smaller classes based on:

- Cohesive behaviour
- Related functionality

Common Refactorings:

Replace Inheritance by Delegation

Favour Composition over Inheritance

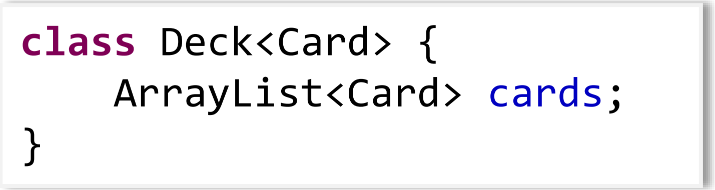
Suppose: `class Deck<Card> extends ArrayList<Card>`

Reasoning: a Deck is a list of Cards

Wrong: relationship is **`has-a`** not **`is-a`**

Doesn't expose **unnecessary methods** of **`ArrayList`**

Expose only methods a **`Deck`** needs, and delegate their implementation to the contained **`ArrayList`**



```
class Deck<Card> {
    ArrayList<Card> cards;
}
```

Common Refactorings:

Remove Duplication

DRY: Don’t Repeat Yourself

E.g. two blocks of code which are almost identical:

- Extract value(s) where they differ to variable(s)

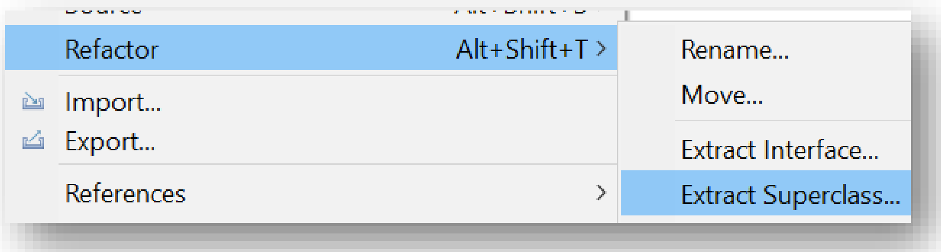
```
@Test
public void gameWith0PinsKnockedDownScores0() {
    roll20(0);
    assertThat(game.score(), 0);
}
```

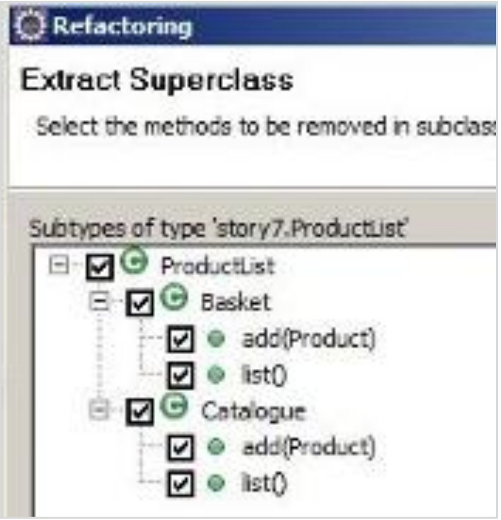
```
@Test
public void gameWith1PinsKnockedDownScores20() {
    roll20(1);
    assertThat(game.score(), 20);
}
```

Make a common method with parameters

Common Refactoring: Extract Superclass

1. Suppose Basket and Catalog have commonality.
   - Both have a List of Products, methods to **`add()`** and **`list()`**
2. Choose one, e.g. Basket -> Extract Superclass. Select methods, fields to be extracted.
3. Basket, etc. will extend new superclass.



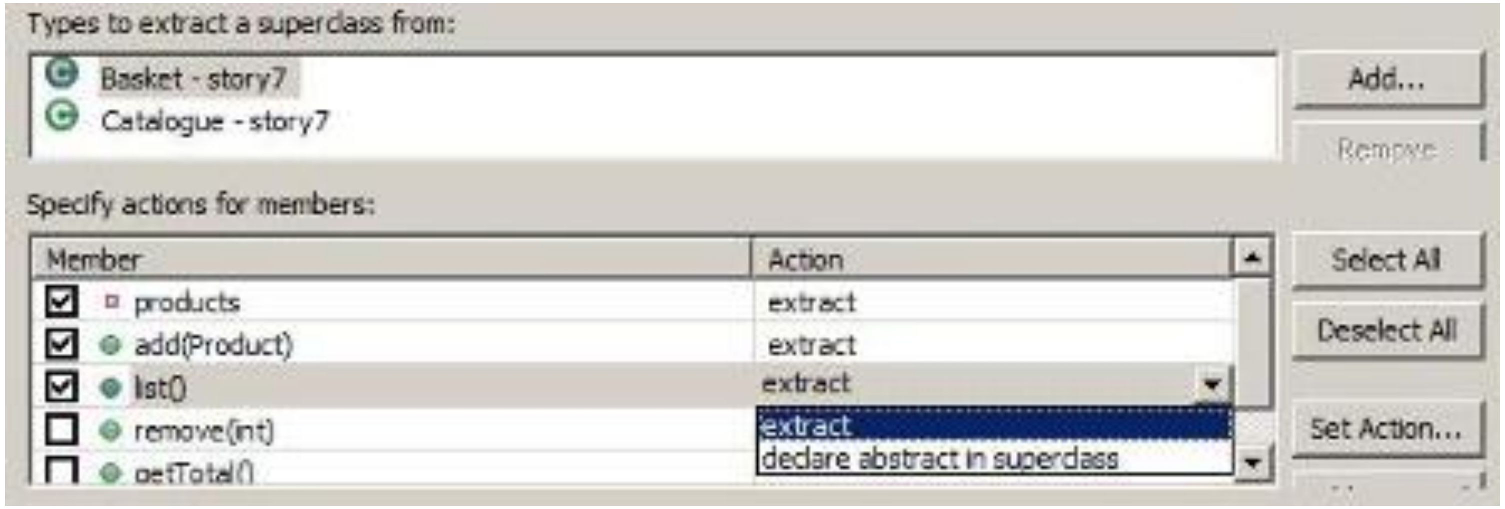


Extract Superclass

Select the methods to be removed in subclas

Subtypes of type 'story7.ProductList'

- ☑ ProductList
  - ☑ Basket
    - ☑ add(Product)
    - ☑ list()
  - ☑ Catalogue
    - ☑ add(Product)
    - ☑ list()



Types to extract a superclass from:

- Basket - story7
- Catalogue - story7

Specify actions for members:

Member	Action
☑ products	extract
☑ add(Product)	extract
☑ list()	extract
☐ remove(int)	
☐ getTotal()	

extract
declare abstract in superclass

Common Refactorings:

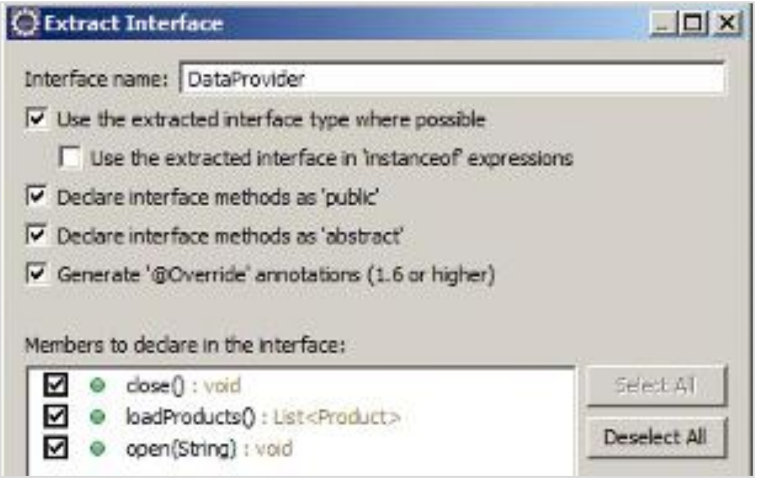
Coding to Interfaces

Extract Interface

Suppose a class needs a Repository / DAO:

```
CSVFileProvider provider = new CSVFileProvider();
```

- Candidate for decoupling interface & implementation:
- Plug in different implementations without affecting rest of code.



Extract Interface

Interface name: DataProvider

☑ Use the extracted interface type where possible

☐ Use the extracted interface in 'instanceof' expressions

☑ Declare interface methods as 'public'

☑ Declare interface methods as 'abstract'

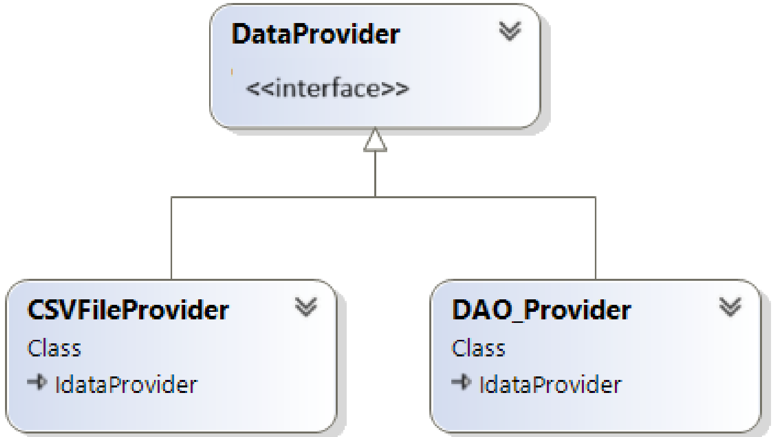
☑ Generate '@Override' annotations (1.6 or higher)

Members to declare in the interface:

☑ ● close() : void

☑ ● loadProducts() : List<Product>

☑ ● open(String) : void



Seams

Seams allow for substitution of classes and functions

A seam is a **joint** where you can insert or replace something

Object Seams - Dependency Substitution
based on either inheritance or interface implementation.

Example - This example is based on the substitution principle:

```
public void ProcessAccount(AccountProcessor proc, Account acc) {
}
```

Inject a mock or stub object

```
class TestAccountProcessor : AccountProcessor{
    // Substitute implementation
}
```

Seams

For legacy code:

- Don't change, substitute when possible
- Change the smallest amount of code possible

Linker Seams

- Different Builds can be defined by varying the classpath

gcc -o app_test app.c mock_logger.c

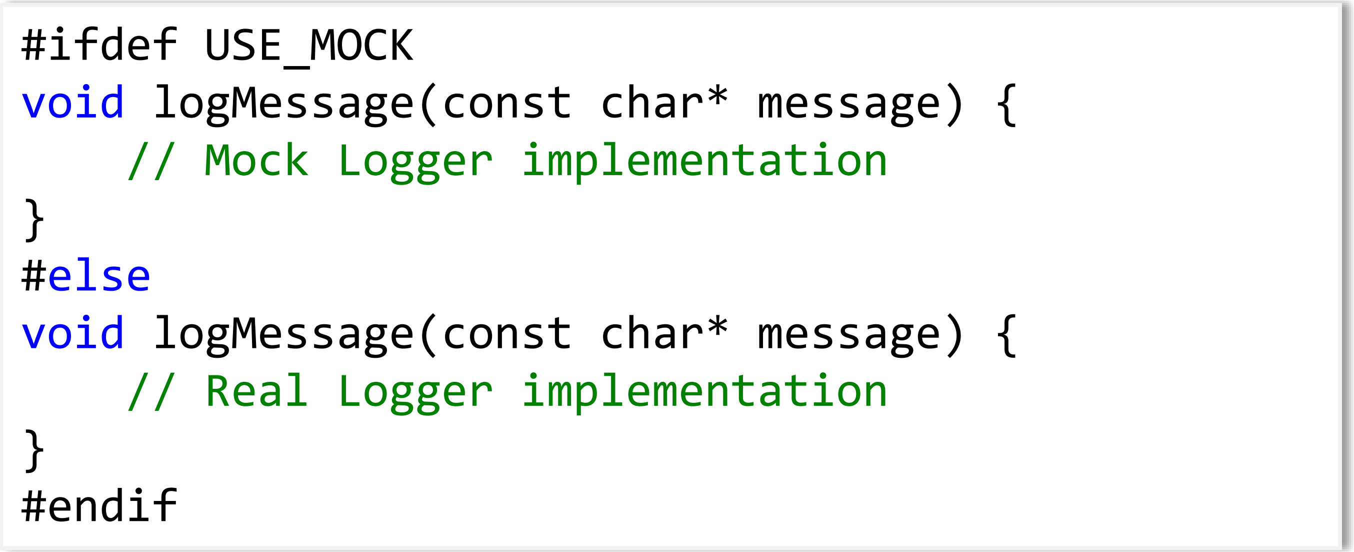
C and C++

Java and C# use runtime binding like dependency injection to swap dependencies **without modifying core logic**

Seams

Pre-processor Seams

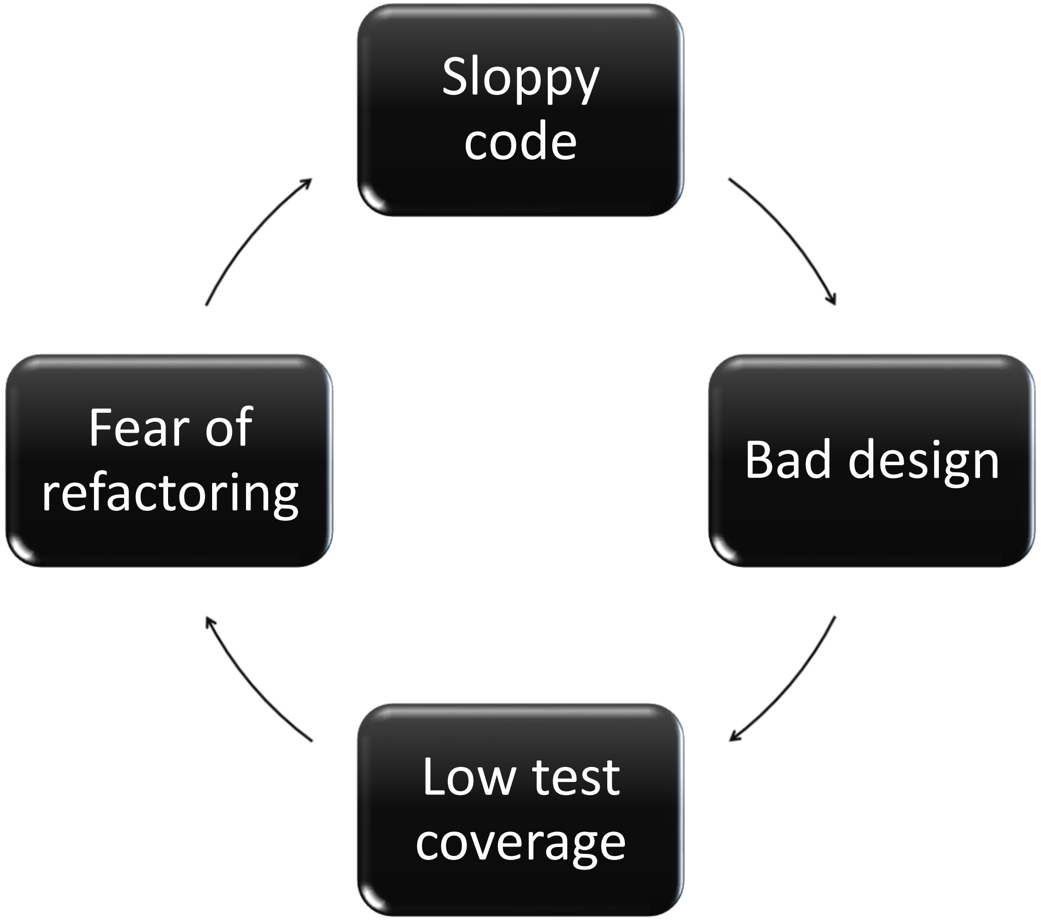
- Based on pre-processor directors managing substitution. Requires a pre-processor



```
#ifdef USE_MOCK
void logMessage(const char* message) {
    // Mock Logger implementation
}
#else
void logMessage(const char* message) {
    // Real Logger implementation
}
#endif
```

Refactoring with little or no test coverage

- Code that has little or no test coverage is usually badly designed
- Makes it hard to know if your code changes with break other parts of the application
- This makes it harder to write tests



Refactoring without tests

- It's a good to have unit tests **before** you start to refactor code
- **However…**
  - It is sometimes necessary to refactor **without** having unit test
  - The old code might be huge, and tangled, with **no existing tests**.
  - Or it might be so old that it is untestable by modern techniques.

- Don't forget, small step refactoring is easier to implement and creates fewer bugs.

"Refactoring with existing tests" Lab

LAB

Learn. To Change.