Debugging and Refactoring

IDEs: Integrated Development Environments
Debugging: process of discovering defects in the program.
Bad Approach

- By inserting temporary print statements
- By manually tracing through the code

Recommended Approach: Using a debugger

Refactor: Improving a program's **internal structure** in **small steps** without modifying its external behavior

- is not rewriting [done in small steps]
- is not bug fixing [alter the external behavior]
- hidden bugs become easier to spot
- improve performance [sometimes, simpler faster than complex]
- too much refactoring: benefits no longer justify the cost: some are '**opposites**' of each other

Refactoring, even if done with the aid of an IDE, may still **result in regressions**.
Each small refactoring MUST followed by **regression testing**.
Commonly used Refactoring

- Consolidate Conditional Expression
- Decompose Conditional
- Inline Method **[inverse of Extract Function]**
- Remove Double Negative
- Replace Magic Literal
- Replace Nested Conditional with Guard Clauses
- Replace Parameter with Explicit Methods
- Reverse Conditional
- Split Loop
- Split Temporary Variable

Identify refactoring opportunities by **code smells**.
Code Smell: a surface indication that usually **corresponds to a deeper problem** in the system.

- a smell is something that's quick to spot
- smells don't always indicate a problem.

Code Smell Data Class: a class with **all data** and **no behavio**r
$\Longrightarrow$ Explore if refactoring it to move the corresponding behavior into that class is appropriate

Periodic refactoring: good way to pay off the technical debt a codebase has accumulated.
Cruft: deficiencies in internal quality that make it harder than it would ideally be to modify and extend the system further.

Documentation Principle

- **Developer-as-user:** there is a need for how such components are to be used
- **Developer-as-maintainer:** there is a need for designed, implemented and tested
- There **isn't one** thing called documentation, there are **four**: tutorials, how-to guides, explanation and technical reference.
- A writer-friendly source format is also **desirable as non-programmers**
- When writing project documents, a **top-down breadth-first explanation** is easier to understand than a bottom-up one.
  [Reader can travel down a path she is interested in until she reaches the component she is interested to learn in-depth]
- It is **not a good idea to have separate sections** for each type of artifact
  - Aim for '**just enough**' **developer doc**umentation.
  - Minimize that overhead.
- Documentation should complement the code and should provide only just enough guidance to get started.
- Anything that is **already clear** in the **code** need **not** be **described in words**
- Focus on **providing higher level information** that is not readily visible in the code or comments.
- Describe the **similarities in one place** and emphasize only the **differences in other places**.
- JavaDoc is a tool for **generating API documentation** in HTML format from comments in the source code

Error Handling

Exception: Error occurs in the execution→the code being executed creates an **exception object**→hands it off to the **runtime** system.
contains information about the error, including its type and the state of the program when the error occurred.
Throwing: Creating an **exception object** and handing it to the runtime system
↓
Call Stack: **Runtime** system attempts to find something to handle it.
↓
Searches [in reverse order] the call stack for a method that contains **exception handler** that can handle the exception.
↓
Catch: The exception handler chosen.
Assertions: define assumptions about the program state so that the **runtime can verify** them.

- **assertion failure** indicates a possible bug in the code
- can be disabled without modifying the code. (Java disables assertions **by default**)
- recommend to use liberally in the code.
  [impact on performance is low]
- Do not use to do work [assertions **can be disabled**]
- suitable for verifying assumptions about Internal Invariants, Control-Flow Invariants, Preconditions, Postconditions, and Class Invariants.

Java assert vs JUnit assertions:

- Both check for a given condition
- JUnit assertions are more powerful and customized for testing.
- JUnit **assertions** are **not disabled** by default
- Use JUnit **assertions** in **test** code
- Use Java **assert** in **functional** code.

Exceptions	Assertions
an unusual condition created by the user or the environment	programmer made a mistake in the code
• complementary ways of handling errors • different purposes	

Logging: deliberate recording of certain information during a program execution for future reference.

- **useful for troubleshooting problems**. [records some system information regularly]
- Most programming environments come with logging systems that allow sophisticated forms of logging.
- possible to log information in other ways
  e.g. into a database or a remote server.

Defensive programmer codes under the assumption "if you leave room for things to go wrong, they will go wrong".
not necessary to be 100% defensive all the time
[less prone to be misused or abused→complicated and slow]

Integration

Combining parts of a software product to form a whole.
two general approaches

1. **Late and one-time:** wait till all components are completed and integrate all finished components near the end of the project. **[not recommended]**
2. **Early and frequent:** integrate early and evolve each part in parallel, in small steps, **re-integrating frequently**.

Walking Skeleton [associate with E&f integration]

- has all the high-level components needed for the first version in their minimal form, compiles, and runs
- may not produce any useful output yet

Big-bang integration: integrating **too many changes** at the same time. [**not recommended**]
Incremental integration: integrate **a few** components at a time.

Build automation tools automate the steps of the build process, usually by means of build scripts.

- Some of these build steps such as compiling, linking and packaging, are already automated in most modern IDEs.
- Some popular build tools relevant to Java developers: Gradle, Maven, Apache Ant, GNU Make
- Some other build tools: Grunt (JavaScript), Rake (Ruby)

Some build tools (Maven, Gradle) also serve as **dependency management** tools. [Modern software projects often depend on third party libraries that evolve constantly]

Continuous integration (CI)

- extreme application of build automation.
- integration, building, and testing happens **automatically** after each code change.

Continuous Deployment (CD)

- natural extension of CI
- changes are not only integrated continuously, but also deployed to end-users at the same time.

Examples of CI/CD tools: Travis, Jenkins, Appveyor, CircleCI, GitHub Actions

Reuse

a major theme in SWE practices.

- enhance robustness of a new software system
- reduce manpower and time requirement

costs of reuse

- may be an overkill [increasing size/degrade performance]
- may not be mature/stable enough to be used in an important product
- has the risk of dying off left dependency no longer maintained.
- license restriction
- might have bugs, missing features, or security vulnerabilities
- Malicious code can sneak into your product via compromised dependencies.

Application Programming Interface (API): specifies the interface through which other programs can interact with a software component. [contract between component and its clients]

- **A class has an API:** a collection of **public methods** that you can invoke to make use of the class.
- The **GitHub API** is a collection of web request formats that the GitHub server accepts and their corresponding responses. [interacts with GitHub through that API]

When developing large systems, if you define the API of each component early, the development team can develop the components in **parallel**

Library: collection of modular code that is general and can be used by other programs.

- **Java classes you get with the JDK are library** classes that are provided in the default Java distribution.

Use a library

- **Read the doc.** to confirm its functionality fits needs
- **Check the license**
- **Download the library and make it accessible** to your project. [Alternatively, you can configure your **dependency management tool** to do it for you.]
- **Call the library API** from your code where you need to use the library's functionality.

Framework: a **reusable implementation of a software** (**or part** thereof) providing generic functionality that can be selectively customized to produce a specific application.

- Some frameworks provide a complete implementation of a default behavior which makes them immediately usable.
- Some frameworks cover **only a specific** component or an **aspect.**
- A framework facilitates the adaptation and customization of some desired functionality.
- use a technique called **inversion of control**, aka the "**Hollywood principle**"

Example: Eclipse

- is an **IDE framework**
- is a fully functional Java IDE out-of-the-box.
- Eclipse plugin system can be used to create an IDE for different programming languages

Example

- JavaFX is a framework for creating Java GUIs
- Tkinter is a GUI framework for Python
- JUnit (Java) is a framework for testing

- Frameworks for web-based applications: Drupal (PHP), Django (Python), Ruby on Rails (Ruby), Spring (Java)
- Frameworks for testing: JUnit (Java), unittest (Python), Jest (JavaScript)

Libraries	Frameworks
meant to be used 'as is'	customized/extended
code calls the library code	framework code calls your code

Platform: provides a runtime environment for applications

- an **operating system can be called a platform**
- Two well-known examples of platforms are JavaEE and .NET [both are **platform** and **framework**]

Code quality

Among various dimensions of code quality, such as run-time efficiency, security, and robustness, one of the most important is **readability** (aka understandability).

- **Avoid long methods** [the bigger the haystack, the harder it is to find a needle]
- **Avoid deep nesting** (**Arrowhead style code**) [the deeper the nesting, the harder it is for the reader to keep track of the logic]

Avoid complicated expressions

Bad

```
return ((length < MAX_LENGTH) || (previousSize != length))
        && (typeCode == URGENT);
```

Good

```
boolean isWithinSizeLimit = length < MAX_LENGTH;
boolean isSameSize = previousSize != length;
boolean isValidCode = isWithinSizeLimit || isSameSize;

boolean isUrgent = typeCode == URGENT;

return isValidCode && isUrgent;
```

Avoid any magic literals

Bad

```
return 3.14236;
...
return 9;
```

Good

```
static final double PI = 3.14236;
static final int MAX_SIZE = 10;
...
return PI;
...
return MAX_SIZE - 1;
```

Bad

```
return "Error 1432"; // A magic string!
```

- Make the code **as explicit as possible**, even if the language syntax allows them to be implicit.
- Use **enumerations** when a **certain variable** can take only a small number of finite values
- Lay out the code to **adheres to the logical structure**.

Bad

```
statement A1
statement A2
statement A3
statement B1
statement C1
statement B2
statement C2
```

Good

```
statement A1
statement A2
statement A3

statement B1
statement B2

statement C1
statement C2
```

Avoid things that would make the reader go 'huh?'

- unused parameters in the method signature
- similar things that look different
- different things that look similar
- multiple statements in the same line
- data flow anomalies such as, pre-assigning values to variables and modifying it without any use of the pre-assigned value

KISS

- Do not try to write 'clever' code. "Keep it simple, stupid"
- Choose 'clever' sol. only if additional cost of is justifiable.

Avoid Premature Optimizations

- You may not know which parts are the real performance bottlenecks.
- Optimizing can complicate the code
- Hand-optimized code can be harder for the compiler to optimize
- there are also cases in which optimizing takes priority over other things

Single Level of Abstraction Principle (SLAP)

- Avoid having multiple levels of abstraction within a code fragment.

Bad (`readData();` and `salary = basic * rise + 1000;` are at different levels of abstraction)

```
readData();
salary = basic * rise + 1000;
tax = (taxable ? salary * 0.07 : 0);
displayResult();
```

Good (all statements are at the same level of abstraction)

```
readData();
processData();
displayResult();
```

- Ensure that the code is written at the **highest level of abstraction** possible.

Bad (all statements are at a low levels of abstraction)

```
low-level statement A1
low-level statement A2
low-level statement A3
low-level statement B1
low-level statement B2
if condition X :
    low-level statement C1
    low-level statement C2
```

Good (all statements are at the same high level of abstraction)

```
high-level step A
high-level step B
if condition X:
  high-level step C
```

- Sometimes **possible to pack two levels of abstraction** into the code

Happy path: execution path taken when everything goes well

- should be clear and prominent in your code
- **deals with unusual conditions as soon as they are detected** so that the reader doesn't have to remember them for long.
- keeps the main path un-indented.

Bad

```
if (!isUnusualCase) {  //detecting
    if (!isErrorCase) {
        start();    //main path
        process();
        cleanup();
        exit();
    } else {
        handleError();
    }
} else {
    handleUnusualCase(); //handling
}
```

Good

```
if (isUnusualCase) { //Guard Clause
    handleUnusualCase();
    return;
}

if (isErrorCase) { //Guard Clause
    handleError();
    return;
}

start();
process();
cleanup();
exit();
```

Bad

```
for (condition1)
    if (condition2)
        statement A
        statement B
        statement C
        statement D
statement E
```

Good

```
for (condition1)
    if (not condition2)
        continue
    statement A
    statement B
    statement C
    statement D
statement E
```

Naming Well

- nouns for classes/variables
- verbs for methods/functions.

Name for a	Bad	Good
Class	`CheckLimit`	`LimitChecker`
Method	`result()`	`calculate()`

- Distinguish clearly between **single-valued** and **multi-valued** variables.

Good

```
Person student;
ArrayList<Person> students;
```

- **Avoid foreign language words, slang, and names** that are only meaningful within specific contexts/times
- multiple words should be in a **sensible order**.

Bad	Good
`bySizeOrder()`	`orderBySize()`

- **Don't use numbers or case to distinguish names.**

Bad	Bad	Good
`value1` , `value2`	`value` , `Value`	`originalValue` , `finalValue`

- Avoid Misleading Name

Bad	Good	Reason
`phase0`	`phaseZero`	Is that zero or letter O?
`rwrLgtDirn`	`rowerLegitDirection`	Hard to pronounce
`right` `left` `wrong`	`rightDirection` `leftDirection` `wrongResponse`	`right` is for 'correct' or 'opposite of 'left'?
`redBooks` `readBooks`	`redColorBooks` `booksRead`	`red` and `read` (past tense) sounds the same
`FiletMignon`	`egg`	If the requirement is just a name of a food, `egg` is a much easier to type/say choice than `FiletMignon`

Avoid Unsafe ShortCuts

- Always **include a default branch** in case statements. [all possible outcomes have been considered at the branching point]
- use the **default branch for the intended default action** and not just to execute the last option
- no default action, you can use the **default branch to detect errors**

Bad

```
if (red) print "red";
else print "blue";
```

Good

```
if (red) print "red";
else if (blue) print "blue";
else error("incorrect input");
```

Variables

- Use **one variable for one purpose**
- Do **not reuse formal parameters** as local variables

Bad

```
double computeRectangleArea(double length, double
    length = length * width;  // parameter reused
    return length;
}
```

Good

```
double computeRectangleArea(double length, double
    double area;
    area = length * width;
    return area;
}
```

- **Avoid empty catch statements: at least give a comment** to explain why the catch block is left empty.

Code Duplication and Scope

- Get rid of unused code the moment it becomes redundant.
- **Code duplication**, especially when you copy-paste-modify code, often indicates a **poor quality**
  [guideline is closely related to the **DRY Principle**]
- **Minimize global variables**.
- Define variables in the **least possible scope.**

DRY (Don't Repeat Yourself) Principle: Every piece of knowledge must have a **single, unambiguous, authoritative** representation within a system

Comment

- Do not repeat in comments information that is already obvious from the code.
- Comments should **explain the WHAT and WHY** aspects of the code, rather than the HOW aspect.

WHAT: The specification of what the code is supposed to do.
WHY: The rationale for the current implementation.
HOW: The explanation for how the code works.