9. Refactoring
Technical debt

• Debt incurred when programming goal is achieved with a shortcut, ignoring good practices
  • “Hack”
• Examples: Bad OOP, class bloat, dead code, unmaintained code, etc.
• Incurs “interest” – debt must be repaid
• Good programming gets the job done; great programming reduces future work
• Refactoring is one solution to technical debt
Quake 3 Arena Code
(calculates 1/sqrt(number))

```c
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threethirds = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y; // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 ); // what the fuck?
    y = * ( float * ) &i;
    y = y * ( threethirds - ( x2 * y * y ) ); // 1st iteration
    // y = y * ( threethirds - ( x2 * y * y ) ); // 2nd iteration, this can be removed

    return y;
}
```
## Clean Code

<table>
<thead>
<tr>
<th>Smelly Code</th>
<th>Clean Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repetitive</td>
<td>DRY</td>
</tr>
<tr>
<td>Unclear, bloated</td>
<td>Easy to understand</td>
</tr>
<tr>
<td>Fails tests</td>
<td>Successful implementation</td>
</tr>
<tr>
<td>Fails integration</td>
<td>Easy to integrate</td>
</tr>
<tr>
<td>Blocks functionality, prevents change</td>
<td>Makes it easy to add functionality and improve the software</td>
</tr>
</tbody>
</table>
What is refactoring?

• “Cleaning up” code to improve it
  • Can be done during code review
  • Can be done while fixing bugs

• Refactoring is not:
  • Commenting your code
  • Adding features
  • Renaming variables
  • Fixing bugs
  • Design
When to refactor?

• Red/green test:
  • Red: something has failed (unit testing, integration testing, bugs)
  • Green: Fix enough to stop the failure
  • Refactor afterwards

• “Make the change easy, then make the easy change”

• Software exhibits anti-patterns (“code smells”)
What makes refactoring difficult?

- Takes time – management sees no need
- Fear that code will not work after refactoring
  - How do we ensure this does not happen?
- Too much refactoring
  - Refactoring can create more code smells
Code Smells: Bloaters

• *Bloaters* are functional code that have become too hard to work with

• **Example:** Large classes/methods
  • It is almost always easier to add new functionality to existing classes/methods than to create new ones
  • Over time, they become harder and harder to understand and test
  • Compromises the Single Responsibility Principle
  • Often produces dead code without anyone noticing
def get_content_from_packet(packet):
    if packet[0] == "4":
        tcp_header = packet[int(packet[1])*4:]
        content = tcp_header[tcp_header[25]*2:]
        return content
    else:
        return None
Bloater: Magic Numbers

• Numbers with unclear meaning

• Problems:
  • Hard to read
  • Difficult to debug and maintain: logic of code is not transparent

• Solution:
  • Replace magic numbers with declared constants

```
cur_speed = -9.8 * time
```

```
GRavitATIONAL_CONST = -9.8
cur_speed = GRAVITATIONAL_CONST * time
```

• Fully implement desired code functionality
class Enemy {
    int[] stats = new int[4];
    int get_HP() {return stats[0];}
    int get_ATK() {return stats[1];}
    void on_hit(int damage) {
        stats[0] -= damage;
    }
}

Bloater: ?
class Customer {
    String address;
    String country;
    String province;
    String city;
    String zipCode;
    ...
}

Bloater: ?
Bloater: Primitive Obsession

• Over-use of primitives: strings, arrays, constants, enums, etc.
  • e.g. Large array that stores multiple unrelated variables
  • e.g. ROLE = 1 is user, ROLE = 2 is admin, ROLE = 3 is guest, etc.

• Problem:
  • Not extensible
  • Breaks single responsibility

• Solution:
  • Replace primitives by implementing objects
    • State pattern can be used
class Customer {
    String address;
    String country;
    String province;
    String city;
    String zipCode;
    ...
}

class Customer {
    Address address;
    ...
}

class Address {
    String get_country() {...}
}
Bloater: Long Parameter List

• You write some simple code to read list of customers

```java
void read_customerlist(String filename)
```

• Later, you’re asked to expand it to read lists of orders too and convert prices from CAD to USD or not

```java
void read_customerlist(String filename, Boolean isOrder, Boolean convertPrices)
```

• Later yet, because there are too many customers, you also want to be able to limit reading a certain range or certain number of customers

```java
void read_customerlist(String filename, Boolean isOrder, Boolean convertPrices, int numCustomers, Date startDate, Date endDate)
```
Bloater: Long Parameter List

• Problems: Hard to read, hard to call, even harder to test

• Some ways to fix this:
  • Settings should belong to the class; isOrder, convertPrices and numCustomers can belong to the CustomerList class, and set by the caller
  • Group parameters together: use a DateRange object instead of startDate and endDate
  • Separate concerns: Write a different method to convert prices or to read order lists
Code smells: OOP Misuse

• This category of code smells covers over-use or mis-use of object-oriented programming principles

```java
public Boolean allow_action(action, person) {
    switch (person.role) {
    case roles.ADMIN:
        //allow editing all documents
        break;
    case roles.USER:
        //allow editing own documents
        break;
    case roles.GUEST:
        //allows viewing all documents
        break;
    }
}
```
OOP Misuse: Switch statements

• Switch statements should generally be replaced with *polymorphism*
  • Admin, User, and Guest should be different subclasses of User
  • Each class handles its own (inherited) `allow_action()`

• Alternative: Use *State* design pattern
  • Person’s role becomes its own object and is composed by `allow_action()`’s class
  • `allow_action()` ask’s the role object whether or not to allow the action
OOP Misuse: Incomplete Inheritance

• If a child object is using only a small part of the methods of the parent class, then OOP is not being used correctly

• Caused by need for code reuse

• Issues: The methods are still there, and may cause errors
  • e.g. You have a Furniture (tables, chairs, closets, etc.) class that has methods: move(), getMaterial(), paintColor(),
  • Later, you decide Doors are Furniture, but calling move() on them would cause an error
  • Later yet, you decide Beds are also Furniture, but paintColor() and getMaterial() both return unexpected results
OOP Misuse: Incomplete Inheritance

- Solution: Either abandon inheritance or improve it
- Abandoning inheritance: Use object composition instead
  - Put an object of the superclass inside of the target class
  - Call methods of the superclass whenever necessary
  - Now it is not possible to call Door.move();

```java
class Door {
    Furniture DoorFurniture;
    Material getMaterial() {
        return DoorFurniture.getMaterial();
    }
}
```
OOP Misuse: Incomplete Inheritance

• Improving Inheritance: Rethink the inheritance structure
• Bed example:
  • Put Furniture under Moveable, put the Furniture.move() function under Moveable class, then inherit Bed from Moveable; or
  • Create PaintableFurniture subclass and put tables, chairs under it; put paintColor() in it only
• Good inheritance makes code extensible
class QuadSolver {
    static double determinant;
    static double[] solve(double a, double b, double c) {
        get_determinant(a, b, c);
        if (determinant > 0) {
            return new double[]{(-b + Math.sqrt(determinant)) / 2*a,
                                (-b - Math.sqrt(determinant)) / 2*a};
        } else if (determinant == 0) {
            return new double[]{-b / 2*a};
        } else return null;
    }
    static void get_determinant(double a, double b, double c) {
        determinant = b*b - 4*a*c;
    }
}
OOP Misuse: Temporary Fields

• Fields in a class that are only used to store a temporary value to support methods

• Problems:
  • Makes code harder to read since fields are only related to a few methods
  • Correct and possible ranges of values are not clear, harder to test
  • Possible misuse of field value

• Solution:
  • Usually simple: set the temporary field to be a local variable within the useful methods
Code Smell: Change Preventers

• Change preventers increase the cost of making changes/adding features to the code

• Two main cases:
  • Modification requires many different changes to a class
  • Modification requires making the same change to many classes

• Both due to poor class structure/programming
Change Preventer: Divergent Change

• Case 1: Type change
• If we need to change HP from int to float...

```java
class Player {
    private int maxHP;
    private int HP;
    private int DEF;
    void on_attacked(Enemy enemy) {
        this.HP -= (enemy.ATK - this.DEF);
    }
    int getHP() {
        return HP;
    }
    Boolean isFullHP() {
        return (maxHP == HP);
    }
}
```
Change Preventer: Divergent Change

• Case 1: Type change
  • Types should not be changed (including interfaces)
  • Lazy way out: Add another variable and switch to that one
    • This can create dead code (another code smell)
  • Re-examine design; types and rationale should be defined in design
    • Why did we want HP to be an int in the first place?
    • If there’s a good reason, perhaps the change should not be made
Change Preventer: Divergent Change

• Case 2: Adding functionality
• If we need to add a new type of product...

```java
class Order {
    Product product;
    float getPrice() {
        if (product.name == "Apple") { return 4.5 }
        if (product.name == "Orange") { return 4.1 }
        ...
    }
    float getDiscount() {
        if (product.type == "Fruit") { return 0.95 }
        else
            ...
    }
}
```
Change Preventer: Divergent Change

- Adding a new product requires changing every method that hardcodes conditionals based on the product

- Not the right way to code
  1. Export prices/discounts into a database file
  2. Read the prices/discounts into each Product
  3. Each Product has a getPrice() and a getDiscount() to retrieve them

- Divergent Change may also be a result of Bloater classes: solution is to extract different methods into different classes
Change Preventer: Shotgun Surgery

• You have many Enemies and Objects that each have an onAttacked() method
• The onAttacked() method checks range between Enemy and Player, and applies an effect
  • e.g. a Trap is dismantled, an Enemy is hit, a Button is pressed
• Later, you find that there is a bug: the Player can hit things through a wall!
• Now, it is time to add a check for walls...
  • Every onAttacked() needs to be fixed!
Change Preventer: Shotgun Surgery

- The responsibility for handling attacks was given to *many* classes
- Shallow fix: add a method to Player that handles attacks, onAttack()
  - Change the logic for resolving attacks to first pass through onAttack(), then go through the target’s onAttacked()
- Deep fix: add a class that handles attacks
  - Rewrite code so that this class handles attack results
Change Preventer: Shotgun Surgery

• Other examples:
  1. Many functions are logging by calling the same function
  2. Each function on a customer account is checking the customer’s balance
  3. Each function on a user account is checking the user’s permissions
Code Smells: Dispensables

• Dispensables are not helpful for the code, but are generally indicative of a larger issue

• Example: Excessive explanatory comments
  • Commenting is *good*, but it is indicative of a deeper issue
  • The best comment is a method’s name and API
  • Some possibilities are:
    1. Code solves a problem in a “hacky” manner
    2. Bloated method that handles too much
    3. Complicated expression that should be expressed with variables
if (((Order.cost >= 100 || Order.cost <= 200) && Customer.MemberStatus == 1) ||
    Customer.MemberStatus == 2) { //Explain this...
    discount = 0.9;
}

float MIN_DISCOUNT_COST = 100;
float MAX_DISCOUNT_COST = 200;
if (Customer.isSilverMember) {
    if (Order.cost >= MIN_DISCOUNT_COST && Order.Cost <= MAX_DISCOUNT_COST) {
        discount = 0.9;
    }
}
if (Customer.isGoldMember) {
    discount = 0.9;
}
Dispensables: Comments

• Some other refactoring solutions:
  • Extract complicated code into its own method, and use the method name/API to help explain it
  • Give the method a better name
  • Use assertions with clear definitions:

```java
int getSelectedFont() {
    // Either there is selected text or there is a selected box.
    return (selectedText.length() > 0) ?
        selectedText.font :
        selectedObject.text.font;
}
```

```java
int getSelectedFont() {
    Assert.isTrue(selectedText.length() > 0 ||
        selectedObject != null);
    return (selectedText.length() > 0) ?
        selectedText.font :
        selectedObject.text.font;
}
```
Dispensables: Duplicate Code

- Exact code duplicates because of copy-paste programming
- Near-duplicates because two or more programmers wrote the same code separately
  - e.g. duplicate file input/output operations
  - e.g. duplicate access/correctness checks
- Duplicate code is harder to maintain
  - Easily becomes dead code
  - It may also indicate bad program structure
Dispensables: Duplicate Code

• Refactoring solution:
  • If duplicate methods in the same class, remove one
  • If duplicate methods in two subclasses, pull method up to parent class
  • If duplicate methods in two classes, consider creating superclass or creating a new class
  • Rethink program structure – why did several programmers create the same method?
Lazy Class

• Classes that do almost nothing should be removed
  • The more classes there are, the harder it is to understand and maintain a program

• May result from moving features of a class to another class

• Example: You created eight classes for monopoly spaces: ColorProperty, Railroad, Utility, Jail, Card, Go, Parking, GoToJail
  • Later, you find out that ColorProperty, Railroad and Utility are very similar, so you made 1 Property superclass for them and moved all coinciding methods into Property
  • Now ColorProperty, Railroad, and Utility are nearly empty
  • Parking is also a Lazy Class
Lazy Class: Data Class

- Data classes contain only data and getters/setters
- Either move more responsibilities into this class, or remove it

```java
class Player {
    PlayerStats playerStats;
}

class PlayerStats {
    int HP;
    int ATK;
    int DEF;
    int get_HP() { //... }
}
```
Code smell: Couplers

• Principle: Maximize cohesion, minimize coupling
• Coupling is excessive dependency between two different classes
• Law of Demeter: Each class should “only talk to its friends”
  • A method can call its arguments’ methods, but no further from that
• Feature Envy: If a class excessively calls another class to provide functionality
class Customer {
    CustomerOrderList orderList;
    float getOrderTotal() {
        float orderTotal;
        for (CustomerOrder order : orderList.getList()) {
            orderTotal += order.getPrice();
        }
        return orderTotal;
    }
    Boolean isGoldMember() {
        return orderList.isGoldMember();
    }
    void addOrder(CustomerOrder order) {
        orderList.getList().add(order);
    }
}
Couplers: Feature Envy

• Each method has an issue
  1. `getOrderTotal()` should’ve been the responsibility of `CustomerOrderList`; the method should be moved there
  2. `isGoldMember()` should be the responsibility of `Customer`; the Boolean should be moved to `Customer`
  3. No one should call `addOrder()` through `Customer`; callers should be made to call `CustomerOrderList` directly

• Feature Envy is a sign of problematic separation of concerns
• Sometimes it is best to move the envied class entirely into the envying class
Couplers: Message Chains

• A method call that looks like this:

  ```java
  player.getStats().getLocation().getX()
  ```

  ```java
  customer.getOrder().generateReceipt().print()
  ```

• Reduces “number of lines”, but is harder to read

• High coupling between calling code and player/customer class

• Solution: Hide the Delegate
  
  • Create getLocationX() for Player, or getLocation() for Player and then extract the X in the calling code
  
  • Create printReceipt() for Customer
Refactoring overview: Composing methods

• Writing good methods that handle the right amount of responsibility

• Good methods should have:
  • Few or no explanatory comments
  • Few parameters
  • No excessive calls to other classes
  • Good use of local variables
Refactoring overview: Moving features

- Many code problems can be solved by simply moving methods/classes to the right place
- Move methods into the most appropriate class for single responsibility
- Move methods out or extract a new class if a class has too many responsibilities
- Similar classes should often be subclasses of a parent class
- However, if a subclass is not using its parent class, move it out
Refactoring overview: Design patterns

• Refactoring into design patterns is a common solution
• Constructors can be replaced with Factory Methods
  • Especially useful when subclasses are created for the constructed object
• Types can be replaced with States
• Duplicate calls to implementation class can be replaced with Command
• Excessive subclasses can be replaced with Decorator
Refactoring overview: Simplification

• Good engineering is simplicity
• Remove unnecessary parameters
  • If several parameters of an object are used, pass the whole object
  • Some parameters can be acquired by code body using a method call
• Merge similar methods using parametrization
• Avoid complicated conditionals
  • Use several conditionals/methods
  • Avoid “control flag” variables
• Use exceptions instead of error codes (“return -1”)