Bad Smell and Refactoring Wrap Up

General philosophy

- A refactoring is just a way of rearranging code
  - Refactorings are used to solve problems
  - If there's no problem, you shouldn't refactor
- The notion of "bad smells" is a way of helping us recognize when we have a problem
  - Familiarity with bad smells also helps us avoid them in the first place
- Refactorings are mostly pretty obvious
  - Most of the value in discussing them is just to bring them into our "conscious toolbox"
  - Refactorings have names in order to crystallize the idea and help us remember it

ไม่หา bad smell และไม่ทำ code refactoring ได้ไหมครับ

ได้ครับ แต่จะเกิด "Technical Debt" ส่วน code debt

Refactoring

- Martin Fowler's book has a catalogue of:
  - 22 "bad smells"
  - 72 "refactorings"
  - We will look at some of the bad smells and what to do about them.

ถ้าต้องการทำ Refactoring ต้องมั่นใจว่า เราทำตามขั้นตอนอย่างถูกต้อง เพราะเราอาจจะไม่สามารถ undo กลับมาที่เดิม เราเปลี่ยนโครงสร้างแต่ไม่เปลี่ยนพฤติกรรม

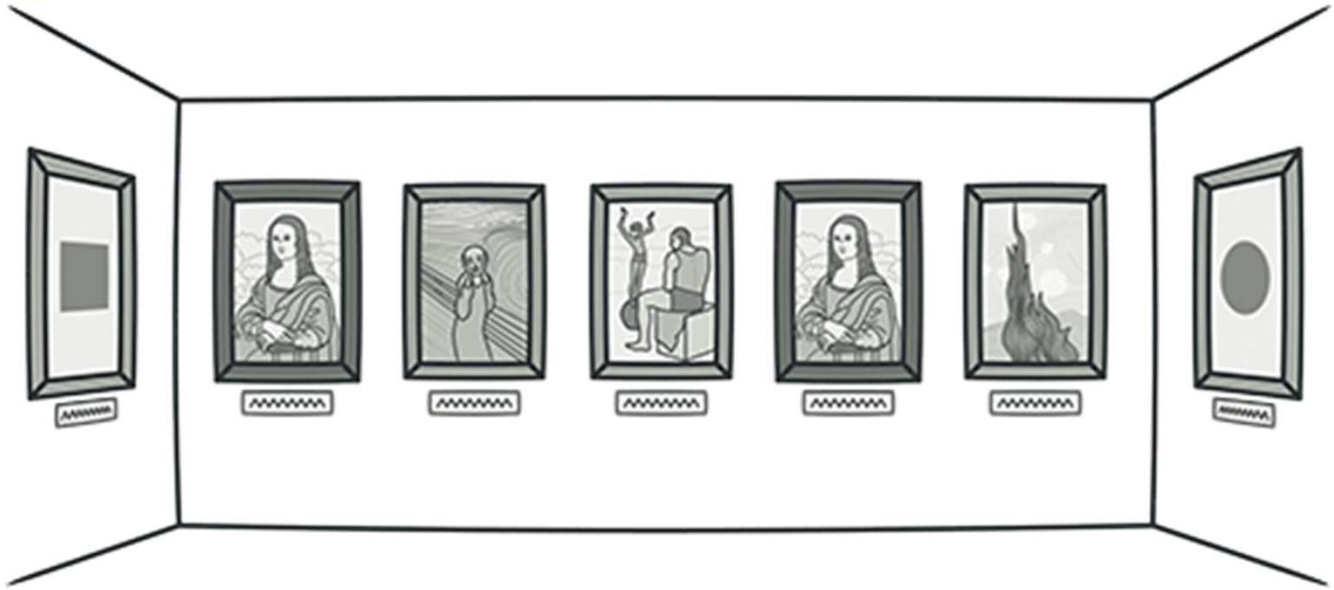
ตัวอย่าง

bad smells – duplicate code, long method, temporary variable, long parameter list, large class, feature envy, middle man, etc.

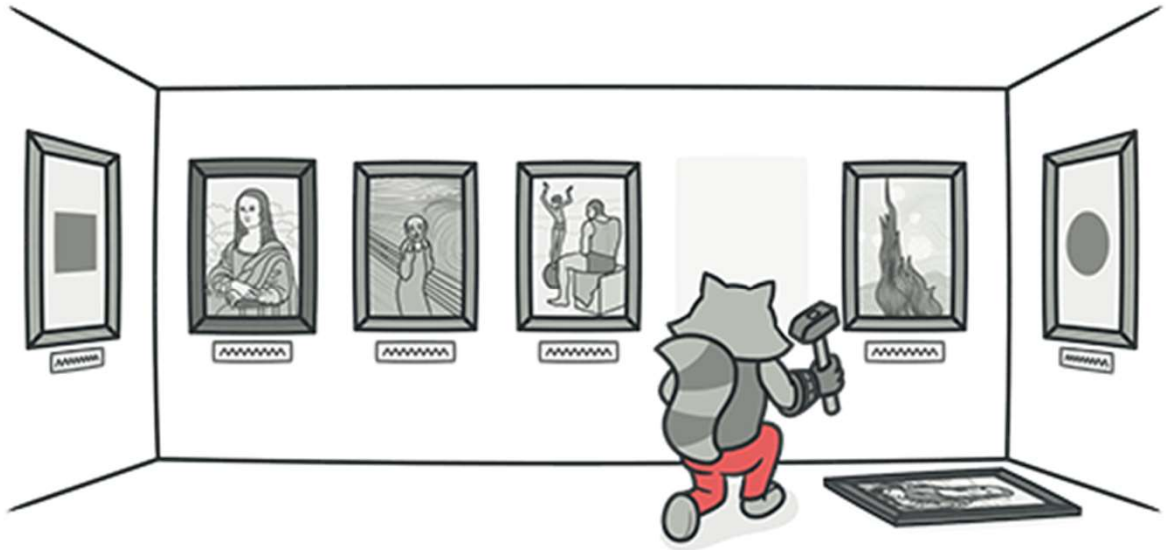
refactorings - extract method, move method, replace temp with query, extract class, … etc.

Duplicate Code

Bad Smell



แก้โดย Extract Method



https://refactoring.guru/refactoring/smells

Duplicated code

Bad Smell

- Martin Fowler refers to duplicated code as "Number one in the stink parade"
  - The usual solution is to apply *Extract Method*: Create a single method from the repeated code, and use it wherever needed
  - We've discussed some of the details of this (adding a parameter list, etc.)
- This adds the overhead of method calls, thus the code gets a bit slower
  - Is this a problem?

Duplicated Code

Original method

```
extern int array_a[];
extern int array_b[];

int sum_a = 0;

for (int i = 0; i < 4; i++)
    sum_a += array_a[i];

int average_a = sum_a / 4;

int sum_b = 0;

for (int i = 0; i < 4; i++)
    sum_b += array_b[i];

int average_b = sum_b / 4;
```

```
extern int array1[];
extern int array2[];

int average1 = calc_average_of_four(array1);
int average2 = calc_average_of_four(array2);
```

New method

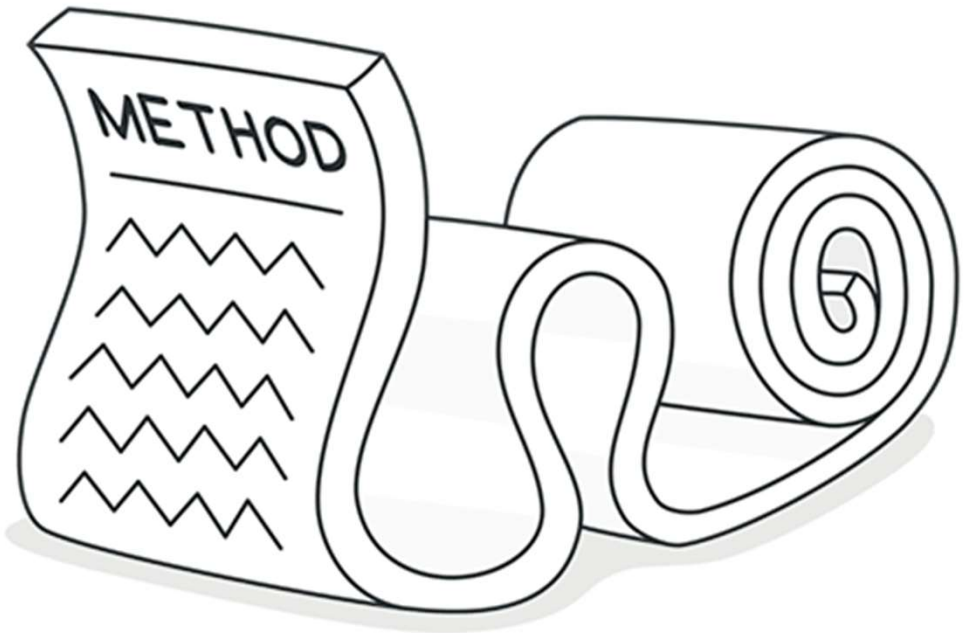
```
int calc_average_of_four(int* array) {
    int sum = 0;
    for (int i = 0; i < 4; i++)
        sum += array[i];

    return sum / 4;
}
```

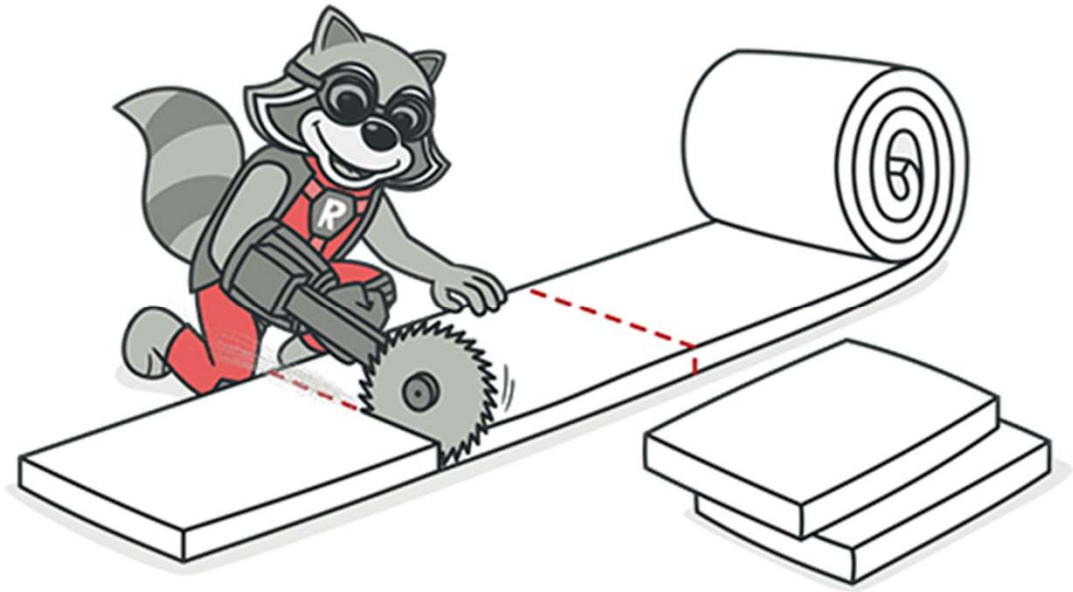
Extract Method

Long Method

Bad Smell



แก้โดย Extract Method



อาจจะมีปัญหาอื่นๆ ตามมาเช่นกัน
ก็ทำ Refactor อื่นๆ กันเพิ่ม

Long methods

Bad Smell

- Another "bad smell" is the overly long method
- Almost always, you can fix long methods by applying *Extract Method*
  - Find parts of the method that seem to perform a single task, and make them into a new method
- Potential problem: You may end up with lots of parameters and temporary variables
  - Temporaries: Consider *Replace Temp With Query*
  - Parameters: Try *Introduce Parameter Object* and *Preserve Whole Object*
  - If all else fails, use *Replace Method With Method Object*

Side effects → introduce the new bad smells

Long Method

Method ที่ใช้สร้างรายงานมักจะยาว

```
defmodule Report do
  def print(user, purchase_order) do
    # Company data
    print_company_data()

    # User data
    IO.puts("Name: #{user.first_name} #{user.last_name}")

    # Order data (with filters and calculations)
    purchase_order.items
    |> Enum.filter(&(&1.status == 3))
    |> Enum.map(fn item ->
      IO.puts("Item: #{item.name}")
      IO.puts("Price: #{item.price}")
      IO.puts("Amount: #{item.amount}")
      total = item.price * item.amount
      IO.puts("Total: #{total}")
    end)
  end
end
```

⋮

Temporary variables

Bad Smell

- According to Fowler, temporary variables "tend to encourage longer methods, because that's the only way you can get at the temp."
- Solution: Use the *Replace Temp With Query* refactoring

```
double basePrice = quantity * itemPrice;
  if (basePrice > 1000) return basePrice * 0.95;
  else return basePrice * 0.98;
```

Replace Temp With Query

Refactor

- Create a method to compute or access the temporary variable
- Example (from Fowler):
- Replace:

Temp

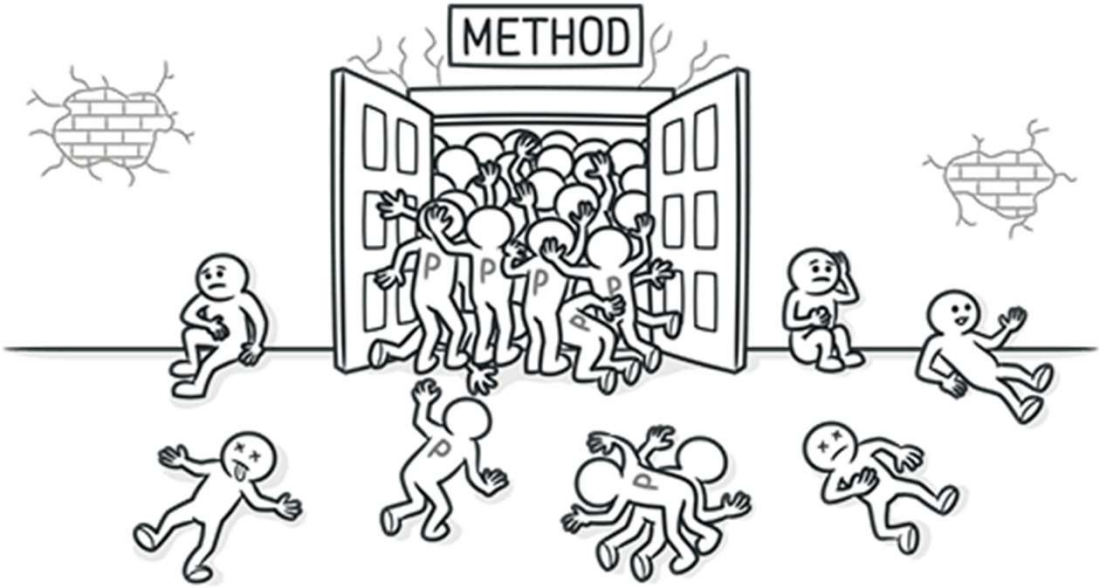
```
double basePrice = quantity * itemPrice;
if (basePrice > 1000) return basePrice * 0.95;
else return basePrice * 0.98;
```

with:

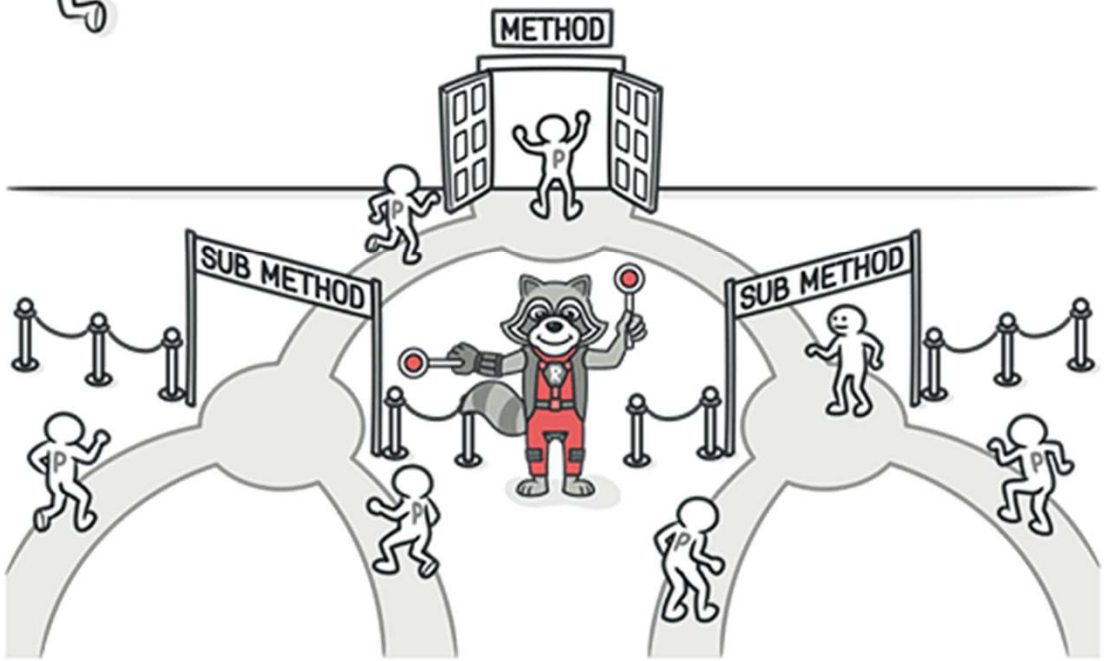
```
if (basePrice() > 1000) return basePrice() * 0.95;
else return basePrice() * 0.98;
...
double basePrice() { return quantity * itemPrice; }
```

Long Parameter List

Bad Smell



แก้โดย Replace Parameter With Method



Long parameter list

Bad Smell

- Long parameter lists are difficult to understand, difficult to remember, and make it harder to format your code nicely
- We've already discussed some solutions, such as *Introduce Parameter Object*
- Another solution, which we won't go into in detail, is called *Replace Parameter With Method*
  - The idea is that you shouldn't pass a parameter into a method if the method has enough information to compute the parameter for itself

Long Parameter List (Sample)

```
defmodule Library do
  def loan(user_name, email, password, user_alias, book_name, book_ed, active) do
    # ... loan/7
    # ... too many parameters that can be grouped in structs!
  end
end
```

Replace Parameter with Method

Calling a query method and passing its results as the parameters of another method, while that method could call the query directly.

Before:

```
int price = quantity * itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice(price, discountLevel);
```

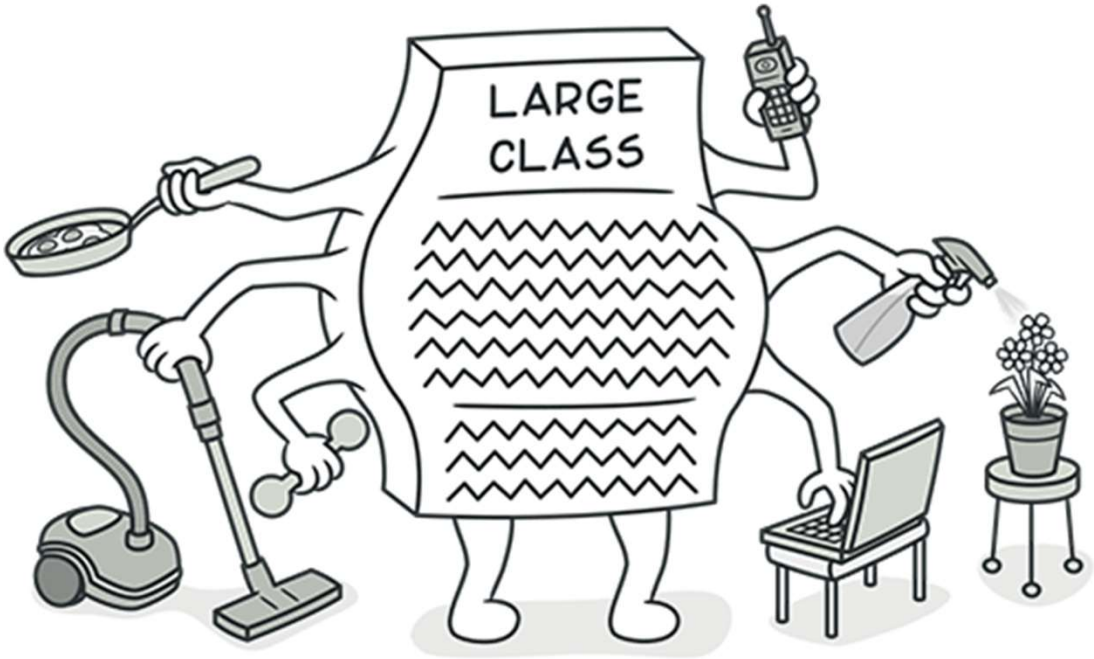
After:

```
int basePrice = quantity * itemPrice;
double finalPrice = discountedPrice(basePrice);
```

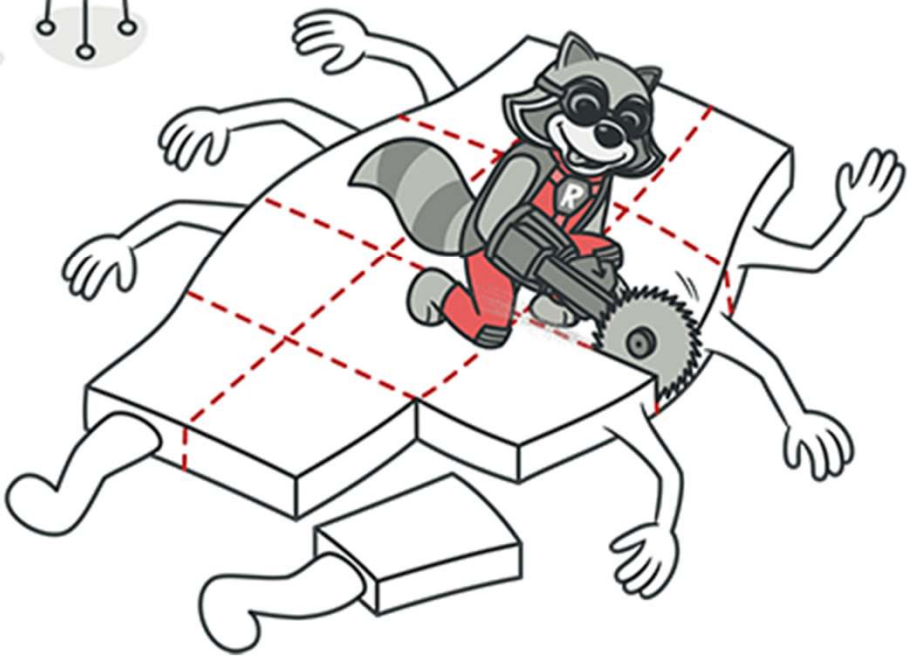
```
Double discountedPrice( int price)
{
    …
    discountLevel = getDiscountLevel();
    …
}
```

Large Class

Bad Smell



แก้โดย Extract Class



Large class

Bad Smell

- Classes can get overly large
  - Too many instance variables
  - More than a couple dozen methods
  - Seemingly unrelated methods in the same class
- Possible refactorings are *Extract Class* and *Extract Subclass*
- A related refactoring, *Extract Interface,* can be helpful in determining how to break up a large class

Large Class

- Lack of cohesion
- Unrelated methods

```
defmodule ShoppingCart do
  # Rule 1
  def calculate_total(items, subscription) do
    # ...
  end

  # Rule 2
  def calculate_shipping(zip_code, %{id: 3}), do: 0.0
  def calculate_shipping(zip_code, %{id: 4}), do: 0.0
  def calculate_shipping(zip_code, _), do
    10.0 * Location.calculate(zip_code)
  end

  # Rule 3
  def apply_discount(total, %{id: 3}), do: total * 0.9
  def apply_discount(total, %{id: 4}), do: total * 0.9
  def apply_discount(total, _), do: total

  # Rule 4
  def send_message_subscription(%{id: 3}, _), do: nil
  def send_message_subscription(%{id: 4}, _), do: nil
  def send_message_subscription(subscription, user),
    do: Subscription.send_email_upgrade(subscription, user)

  # Rule 5
  def print(user, order) do
    # ...
  end
end
```

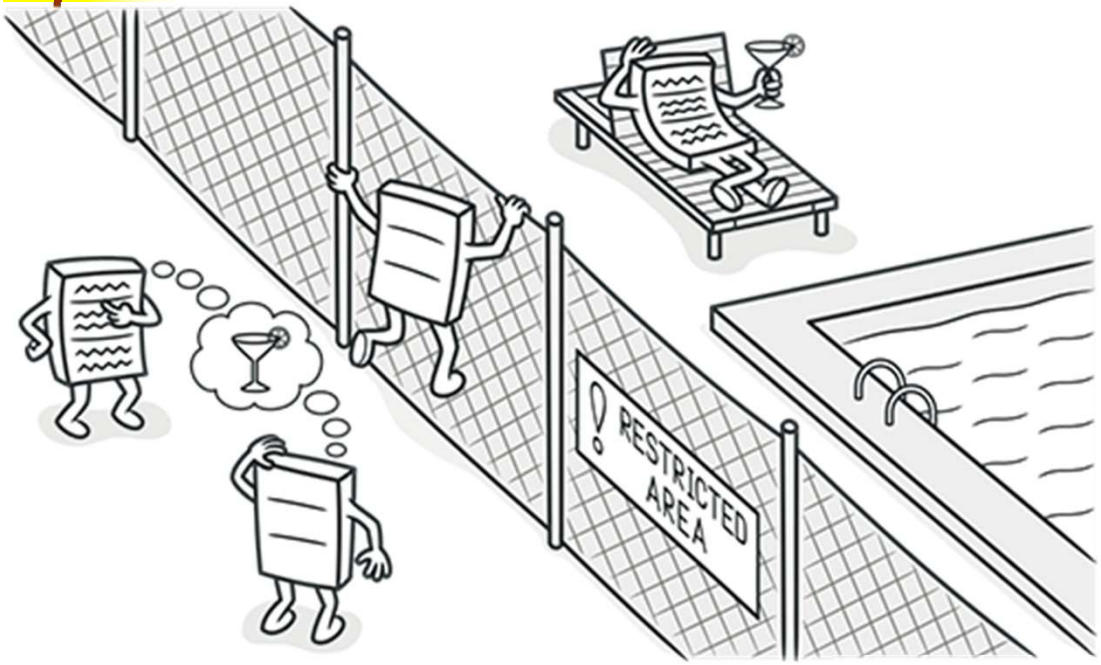
Extract Class

Refactor

- *Extract Class* is used when you decide to break one class into two classes
  - Classes tend to grow, and get more and more data
  - The actual refactoring technique involves creating a new (empty) class, and repeatedly moving fields and methods into it, compiling and testing after each move

Feature Envy

Bad Smell



แก้โดย Move Method

Feature envy

Bad Smell

- "Feature envy" is when a method makes heavy use of data and methods from another class
  - Use *Move Method* to put it in the more desired class
- Sometimes only part of the method makes heavy use of the features of another class
  - Use *Extract Method* to extract those parts that belong in the other class

อยู่ใน Basket แต่ทำงานเกี่ยวกับ Item

Method นี้อยู่ผิดที่

```
// Before refactoring:
class Item { .. }
class Basket {
    // ..
    float getTotalPrice(Item i) {
        float price = i.getPrice() + i.getTax();
        if (i.isOnSale())
            price = price - i.getSaleDiscount() * price;
        return price;
    }
}
```

Move Method

```
// After refactoring:
class Item {
    // ..
    float getTotalPrice() {
        float price = getPrice() + getTax();
        if (isOnSale())
            price = price - getSaleDiscount() * price;
        return price;
    }
}
```

Feature Envy (Sample)

```
defmodule Order do
  def calculate_total_item(id) do
    item = OrderItem.find_item(id)
    total = (item.price + item.taxes) * item.amount
    discount = OrderItem.find_discount(item)

    unless is_nil(discount) do    # <-- all data comes from OrderItem!
      total - total * discount
    else
      total
    end
  end
end
```

ภายใน module Order
มี method ที่อ้างถึงแต่ OrderItem เท่านั้น
ย้ายไปได้

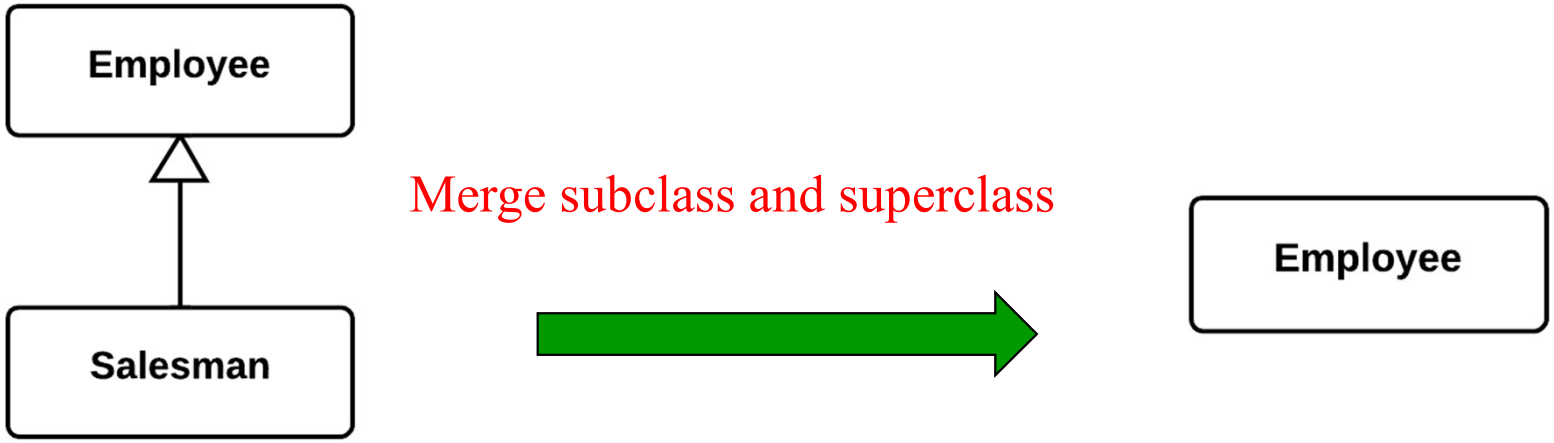
Lazy Class

แก้โดย Collapse Hierachy

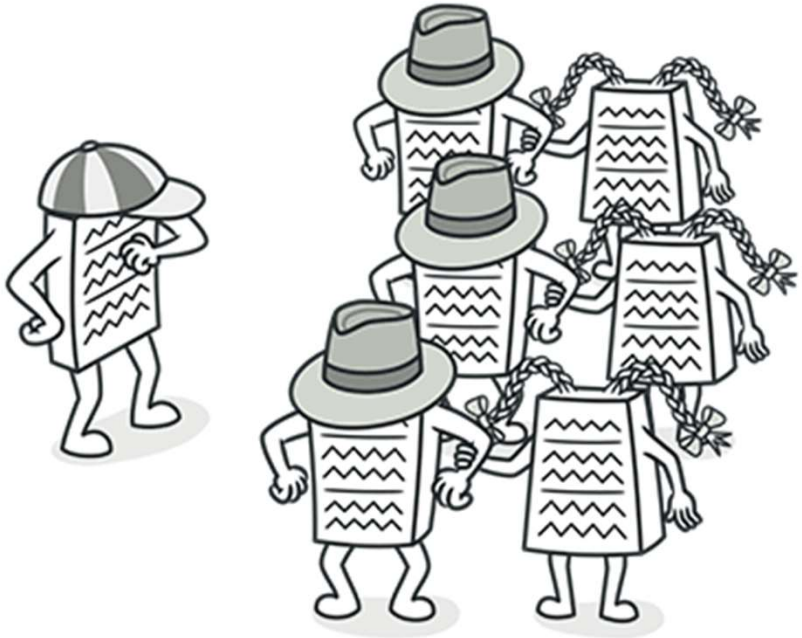
Lazy Class

Bad Smell

- Classes that doesn't do much that's different from other classes.
- Use Collapse Hierarchy



Middle Man



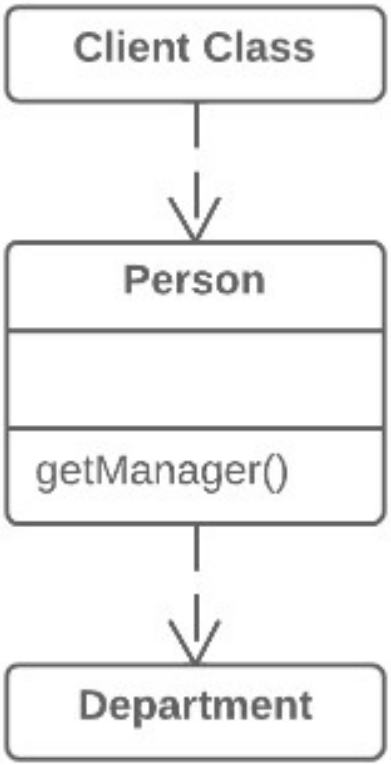
แก้โดย Remove Middle Man

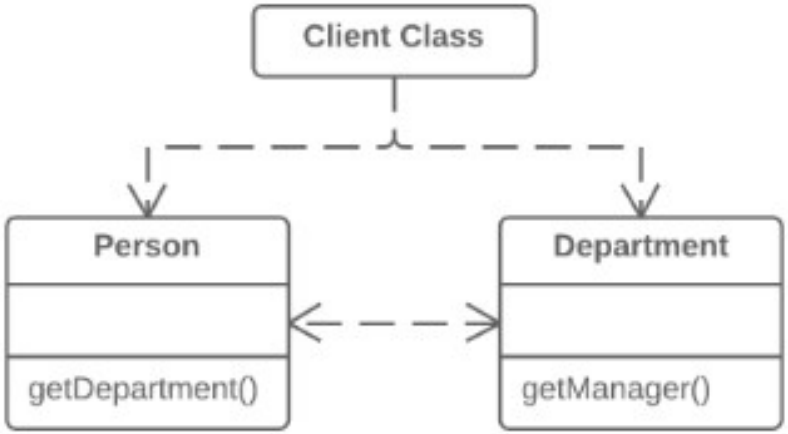
Middle Man

Bad Smell

- *"All hard problems in software engineering can be solved by an extra level of indirection."*
- If you notice that many of a class's methods just turn around and beg services of delegate sub objects, the basic abstraction is probably poorly thought out.
- An object should be more than the some of its parts in terms of behaviours!
  - *(Remove middle man, replace delegation with inheritance)*

Remove Middle Man





Get the client to call the delegate directly

Bad smells in code

Bad Smell

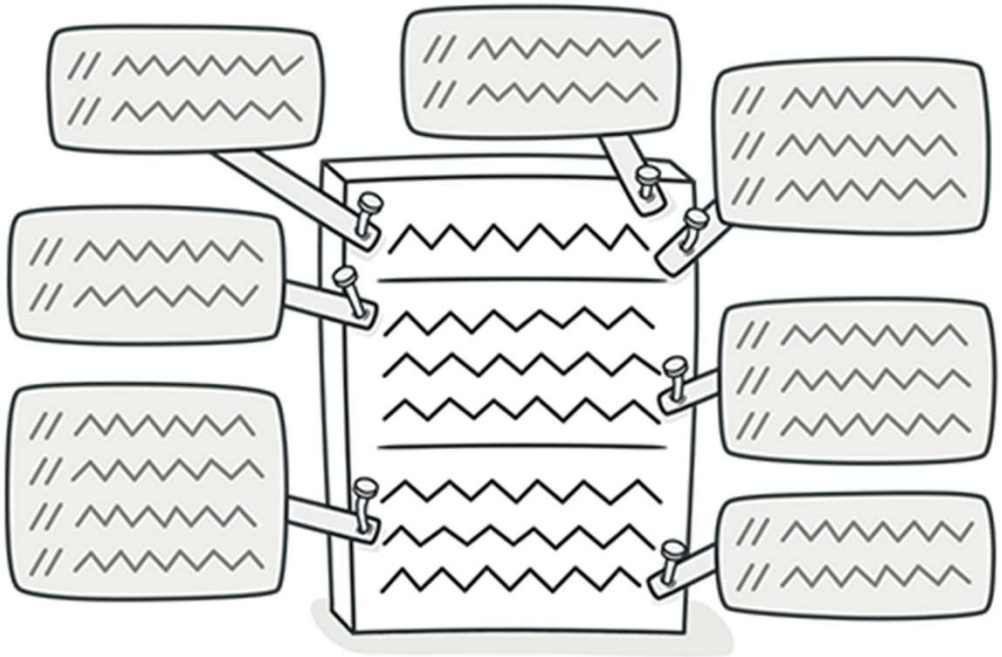
Data class

getter/setter

- Class consists of (simple) data fields and simple accessor/mutator methods only.
- *"Data classes are like children. They are OK as a starting point, but to participate as a grownup object, they need to take on some responsibility."*

  (Extract/move method)

Comment



แก้โดย Extract Method

Comments

Bad Smell

- In the context of refactoring, Fowler claims that long comments are often a sign of opaque, complicated, inscrutable code.
  - They aren't against comments so much as in favour of self-evident coding practices.
  - Use Extract Methods, Extract Variable

Conclusion

- นี่แค่เป็นตัวอย่าง
- ยังไม่ได้ลงรายละเอียด Refactoring Techniques
- เราควรศึกษาเพิ่ม และจำชื่อเป็นคำศัพท์ที่ใช้ในการทำงานได้จริง

Example: Smelly version

```
if (user != null) {
  if (user.isAdmin()) {
    if (user.isActive()) {
      performAdminAction(user);
    } else {
      handleInactiveAdmin(user);
    }
  } else {
    performNonAdminAction(user);
  }
} else {
  handleNullUser();
}
```

What do you think about this code ?

How would you refactor this ?

Or keep it as is ?

Example: Fixed Version

```
if (user == null) {
  handleNullUser();
  return;
}
if (!user.isAdmin()) {
  performNonAdminAction(user);
  return;
}
if (!user.isActive()) {
  handleInactiveAdmin(user);
  return;
}
performAdminAction(user);
```

This type of bad smell is called **Nested Conditionals** where you nest condition clauses, resulting in low readability code.

Example: Smelly version

Have you done this ?

```go
func FormatUser(name string, email string, age int, isActive bool, isAdmin bool, profilePicURL string) (string, error) {
  // ... complex logic to format user data based on parameters ...
  return formattedUser, nil
}

// Usage
formattedString, err := FormatUser("Alice", "alice@example.com", 30, true, false, "https://...")
```

Do you find this smelly ?

How would you improve this ?

Example: Fixed version

It's what you've just learned, Long Parameter List!

```go
type UserData struct {
  Name       string
  Email      string
  Age        int
  IsActive   bool
  IsAdmin    bool
  ProfilePicURL string
}

func FormatUser(user UserData) (string, error) {
  // ... access and format user data from the struct ...
  return formattedUser, nil
}

// Usage (cleaner!)
user := UserData{
  Name:       "Alice",
  Email:      "alice@example.com",
  Age:        30,
  IsActive:   true,
  IsAdmin:    false,
  ProfilePicURL: "https://...",
}

formattedString, err := FormatUser(user)
```

Define another object for the parameter.

Some people won't refactor this on 2 or 3 variables...

how many variables would it take for you to consider refactoring ?

Example: Smelly version

```python
class ProductSearcher:
    def __init__(self, database):
        self.database = database

    def search(self, query):
        # Parse the user query (extract keywords, filter options)
        search_terms = parse_query(query)

        # Build the database query based on parsed terms
        db_query = build_query(search_terms)

        # Execute the query against the database
        results = self.database.execute(db_query)

        # Apply any additional filter criteria (e.g., price range)
        filtered_results = apply_filters(results, search_terms)

        # Sort the results based on user preferences (e.g., relevance, price)
        sorted_results = sort_results(filtered_results, search_terms)

        # Format and present the search results for the user interface
        formatted_results = format_results(sorted_results)

        return formatted_results
```

Example: Fixed version

```python
# Separate classes for each responsibility:

class QueryParser:
    def parse_query(self, query):
        # ... extract keywords and filter options ...

class DatabaseQueryBuilder:
    def build_query(self, search_terms):
        # ... construct database query based on terms ...

class FilterManager:
    def apply_filters(self, results, search_terms):
        # ... apply additional filter criteria ...

class Sorter:
    def sort_results(self, results, search_terms):
        # ... sort results based on user preferences ...

class ResultFormatter:
    def format_results(self, results):
        # ... format results for presentation ...

# Usage in a ProductSearchService class:

class ProductSearchService:
    def __init__(self, database):
        self.parser = QueryParser()
        self.builder = DatabaseQueryBuilder()
        self.filter_manager = FilterManager()
        self.sorter = Sorter()
        self.formatter = ResultFormatter()
        self.database = database

    def search(self, query):
        search_terms = self.parser.parse_query(query)
        db_query = self.builder.build_query(search_terms)
        results = self.database.execute(db_query)
        filtered_results = self.filter_manager.apply_filters(results, search_terms)
        sorted_results = self.sorter.sort_results(filtered_results, search_terms)
        formatted_results = self.formatter.format_results(sorted_results)
        return formatted_results
```

God class !

Break the functionalities into smaller parts and inject them.

However, this "solution" introduces many smaller classes to initialize and inject, so how would you decide to keep the god class or to extract it?