Test Driven Development & Code Smells

Created by

Jan Inge Nygård

(Bouvet)

Introduction

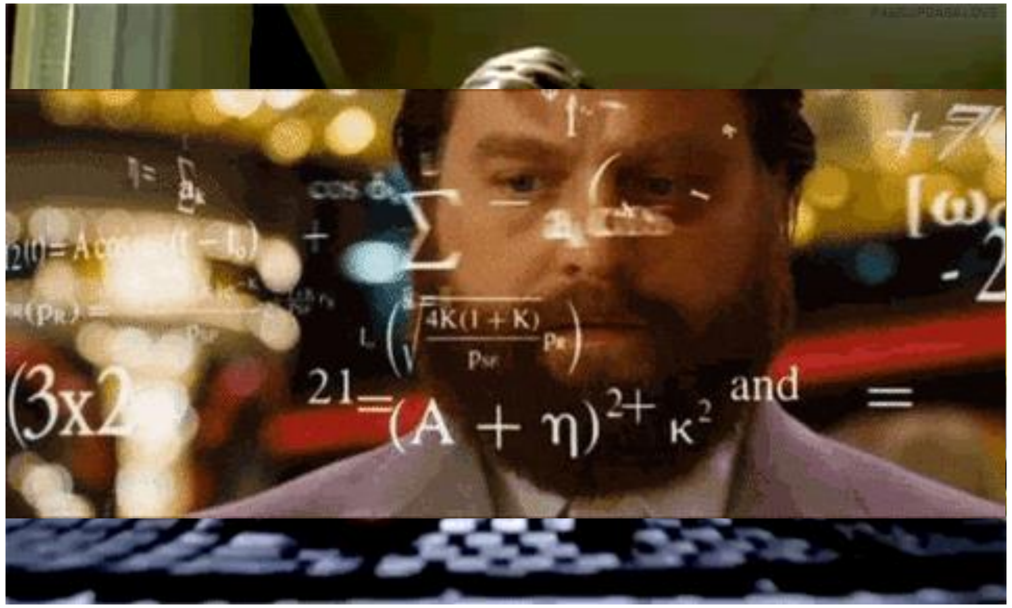
- General workflow for creating tests
- Why and how to refactor
- Code smells & examples

Workflow for creating tests

- Write **test** validation code (assert) that **checks expected** vs **computed value**
- Create **calling code** (act) that fetches the **computed value**
- Initialize necessary **input parameters** (arrange)

Why refactor?

- We **read** more code than we **write** (**90**-**10**)
- Hence, reading is the bottleneck
- Understand code better -> improve readability

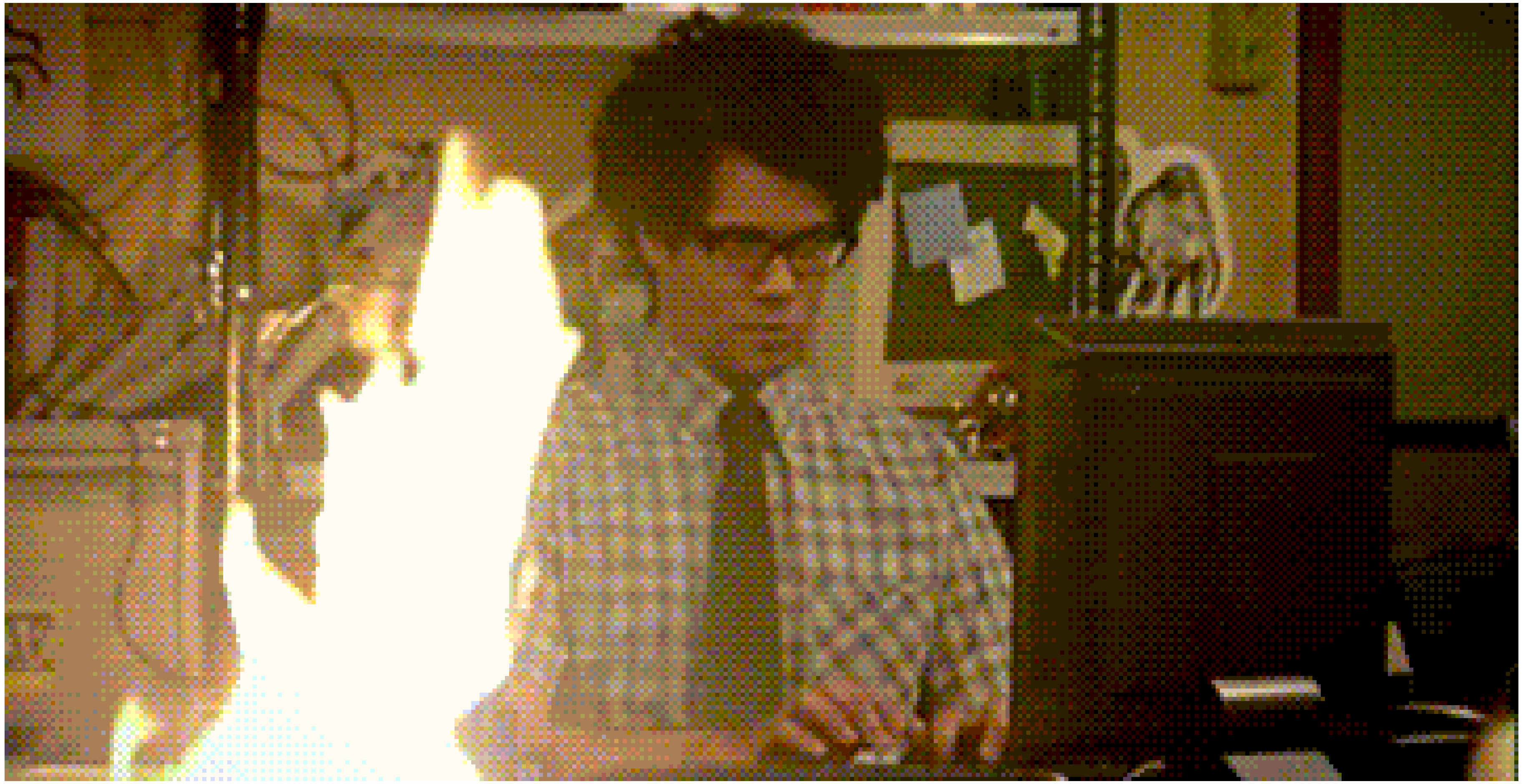


- Use 20 % effort to improve readability by 80 % (Pareto principle)
- Refactor if it makes sense business wise
- Don’t refactor code that is rarely used
- Refactor readability before design

How to refactor?

- Classes, methods, variables, etc can be **renamed**, **extracted**, **inlined** and **moved**
- This improves **readability**, sets the right level of **complexity / abstractions**, and keeps code where it is relevant to it's job / **responsibility**
- Do parallel change (expand, migrate, contract) -> keep old implementation and test new one -> then switch internal usage

Dont refactor while your tests are **red**

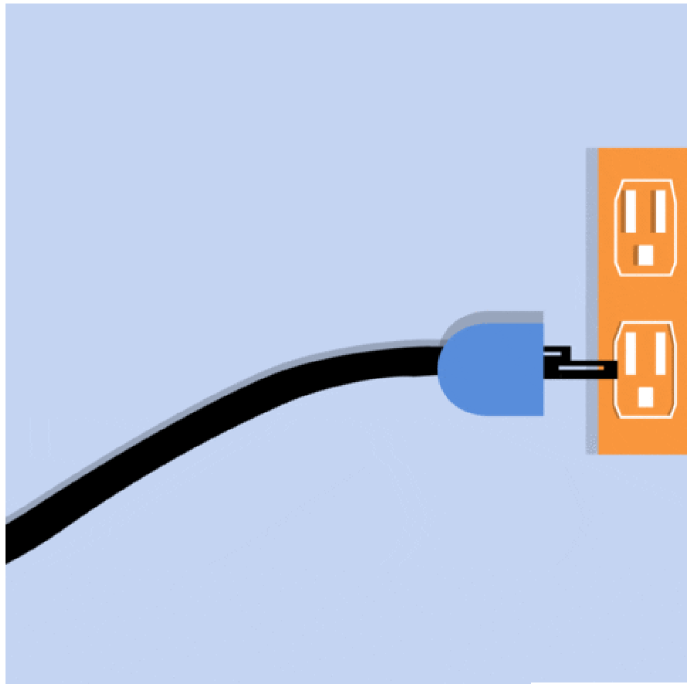


Code Smells

- Be understanding of creators of code smells -> blame the workflow instead
- Code smells indicate a bigger issue underneath
- **First** -> Readability, complexity, responsibility and duplication
- **Next** -> Introduce new abstractions (ie. new types) if needed

Avoid rigid code

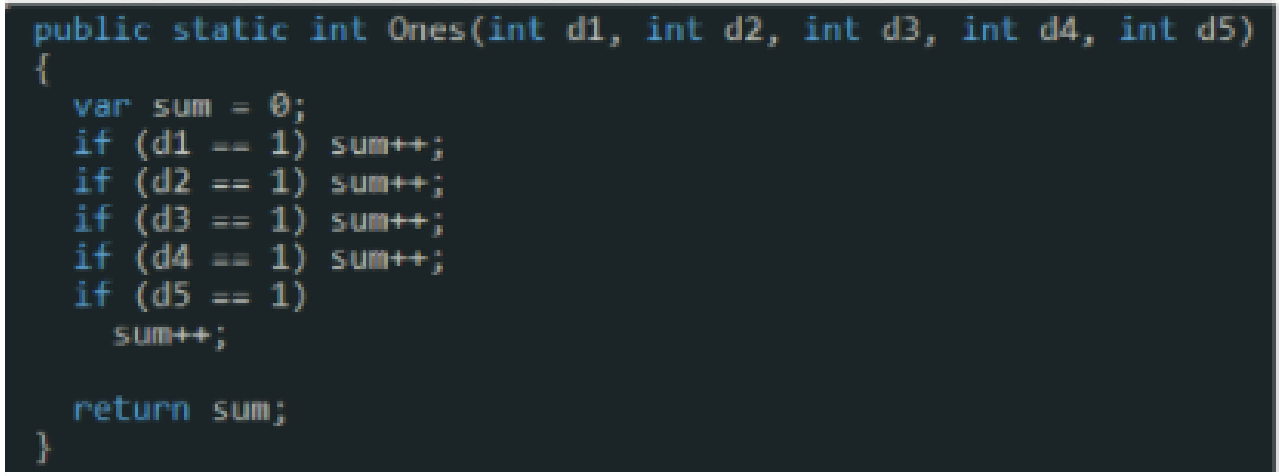
- Code should be **open-closed**, ie. flexible / modular / plug&play-able
- **New functionality** should not require changes to **old code**
- Dont cut corners -> increases **viscosity of design** -> higher technical debt -> harder to maintain
- Lower **viscosity of environment** -> Avoid manual steps with releases -> slows you down -> might forget them



Example Bloaters

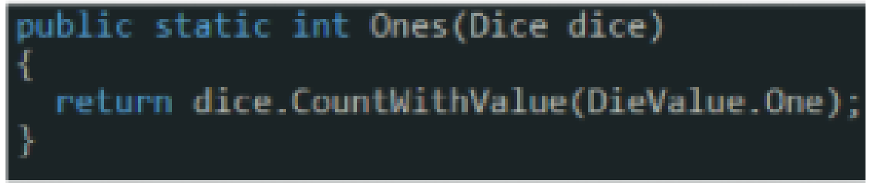
- Too many input parameters -> create a data class instead

Before



```
public static int Ones(int d1, int d2, int d3, int d4, int d5)
{
  var sum = 0;
  if (d1 == 1) sum++;
  if (d2 == 1) sum++;
  if (d3 == 1) sum++;
  if (d4 == 1) sum++;
  if (d5 == 1)
    sum++;

  return sum;
}
```

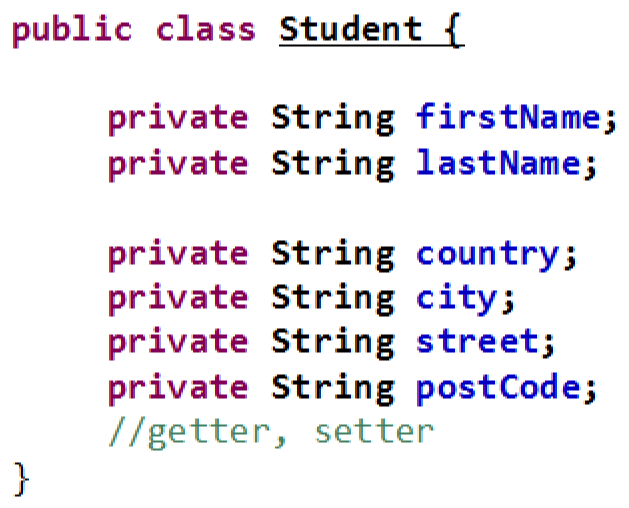
After



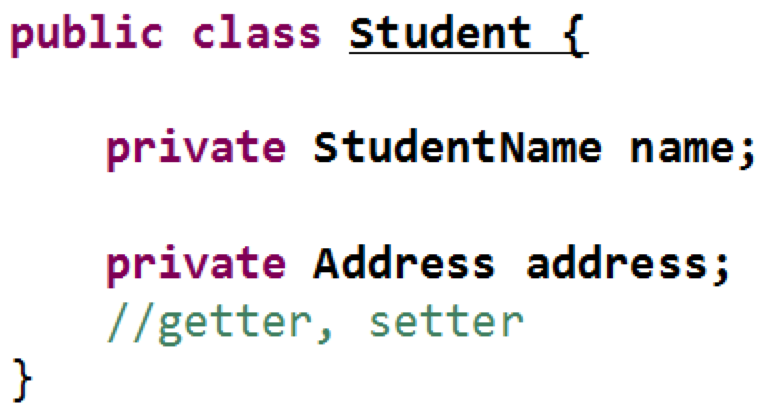
```
public static int Ones(Dice dice)
{
  return dice.CountWithValue(DieValue.One);
}
```

Example Data Clumps

- Data that fit together should stay together
- Could add behaviour to data class -> ie. accept first and last names -> create StudentName value



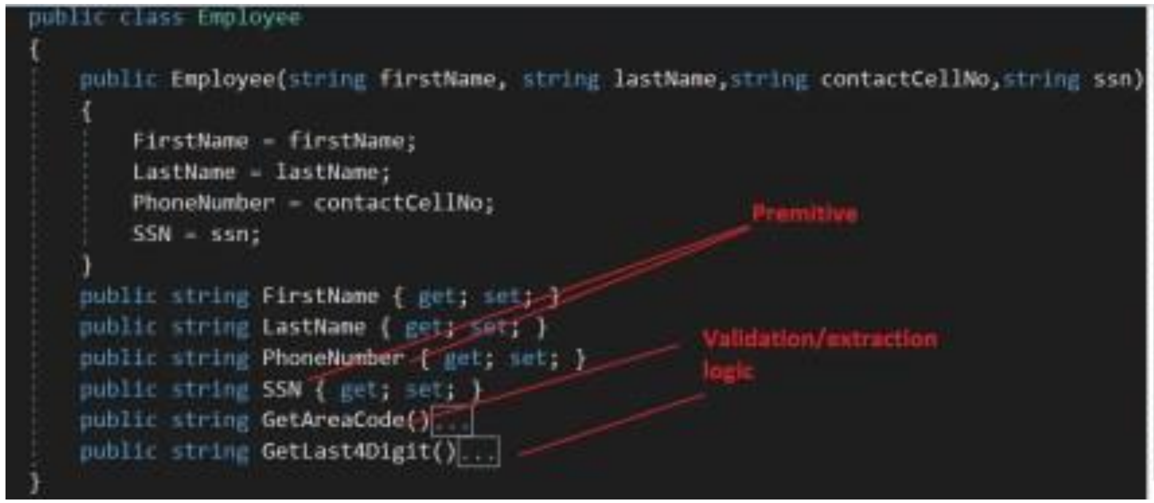
```java
public class Student {

    private String firstName;
    private String lastName;

    private String country;
    private String city;
    private String street;
    private String postCode;
    //getter, setter
}
```



```java
public class Student {

    private StudentName name;

    private Address address;
    //getter, setter
}
```

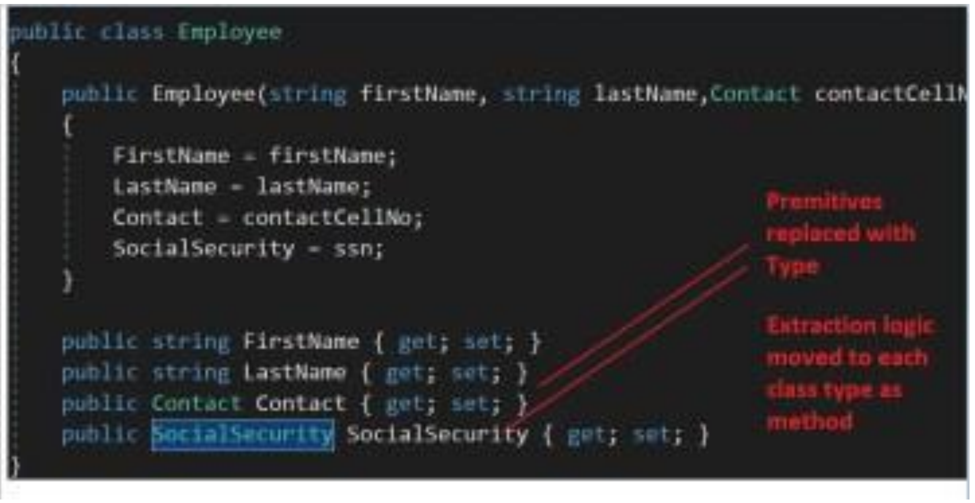
Example Primitive Obsession

- Avoid describing complex concepts with basic types
- Solution -> create a new type to represent that complex object



```
public class Employee
{
    public Employee(string firstName, string lastName,string contactCellNo,string ssn)
    {
        FirstName = firstName;
        LastName = lastName;
        PhoneNumber = contactCellNo;
        SSN = ssn;
    }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
    public string SSN { get; set; }
    public string GetAreaCode()...
    public string GetLast4Digit()...
}
```

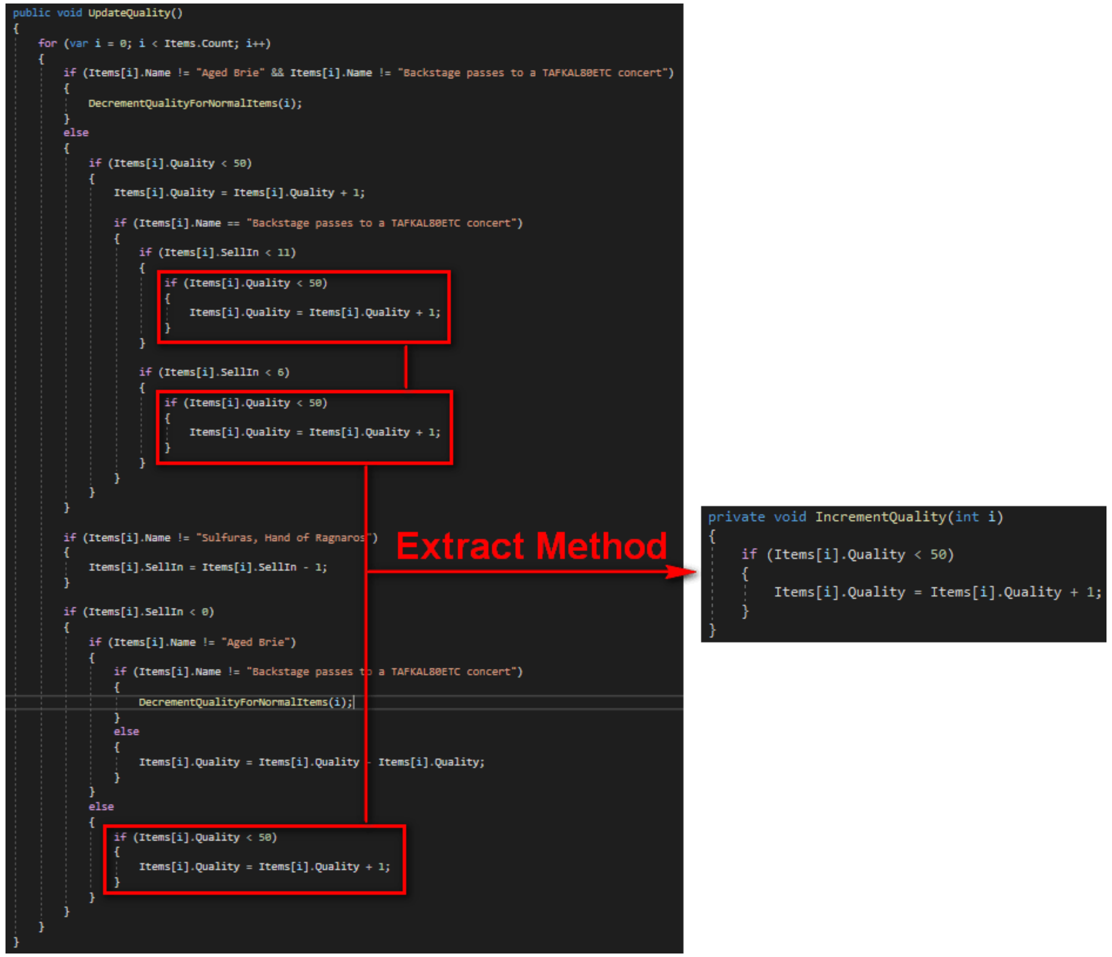
Premitive
Validation/extraction logic



```
public class Employee
{
    public Employee(string firstName, string lastName,Contact contactCellN
    {
        FirstName = firstName;
        LastName = lastName;
        Contact = contactCellNo;
        SocialSecurity = ssn;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Contact Contact { get; set; }
    public SocialSecurity SocialSecurity { get; set; }
}
```

Premitives replaced with Type
Extraction logic moved to each class type as method

Example Long Method



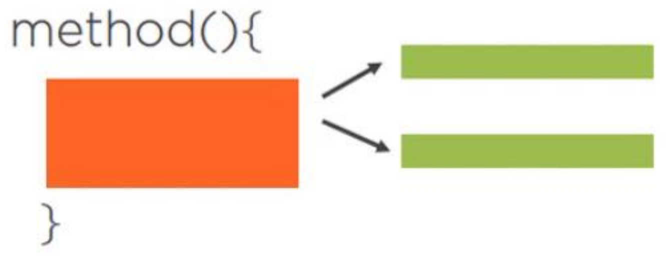
```
public void UpdateQuality()
{
    for (var i = 0; i < Items.Count; i++)
    {
        if (Items[i].Name != "Aged Brie" && Items[i].Name != "Backstage passes to a TAFKAL80ETC concert")
        {
            DecrementQualityForNormalItems(i);
        }
        else
        {
            if (Items[i].Quality < 50)
            {
                Items[i].Quality = Items[i].Quality + 1;

                if (Items[i].Name == "Backstage passes to a TAFKAL80ETC concert")
                {
                    if (Items[i].SellIn < 11)
                    {
                        if (Items[i].Quality < 50)
                        {
                            Items[i].Quality = Items[i].Quality + 1;
                        }
                    }

                    if (Items[i].SellIn < 6)
                    {
                        if (Items[i].Quality < 50)
                        {
                            Items[i].Quality = Items[i].Quality + 1;
                        }
                    }
                }
            }
        }

        if (Items[i].Name != "Sulfuras, Hand of Ragnaros")
        {
            Items[i].SellIn = Items[i].SellIn - 1;
        }

        if (Items[i].SellIn < 0)
        {
            if (Items[i].Name != "Aged Brie")
            {
                if (Items[i].Name != "Backstage passes to a TAFKAL80ETC concert")
                {
                    DecrementQualityForNormalItems(i);
                }
                else
                {
                    Items[i].Quality = Items[i].Quality - Items[i].Quality;
                }
            }
            else
            {
                if (Items[i].Quality < 50)
                {
                    Items[i].Quality = Items[i].Quality + 1;
                }
            }
        }
    }
}
```

```
private void IncrementQuality(int i)
{
    if (Items[i].Quality < 50)
    {
        Items[i].Quality = Items[i].Quality + 1;
    }
}
```

- Too many lines -> hard to read
- Doing too much
- Should only do one thing
- Solution -> split up & extract

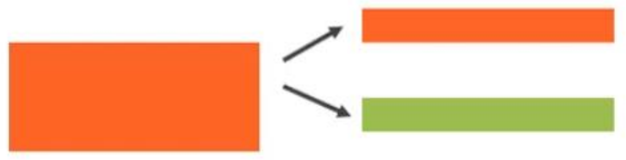
Example Divergent Class

- God class -> “One class to rule them all”
- Violates the 1st responsibility principle
- New change -> have to update multiple code blocks within the god class
- Solution -> Extract & decouple until class only does one thing



Extract Method

(split a method into several smaller methods)

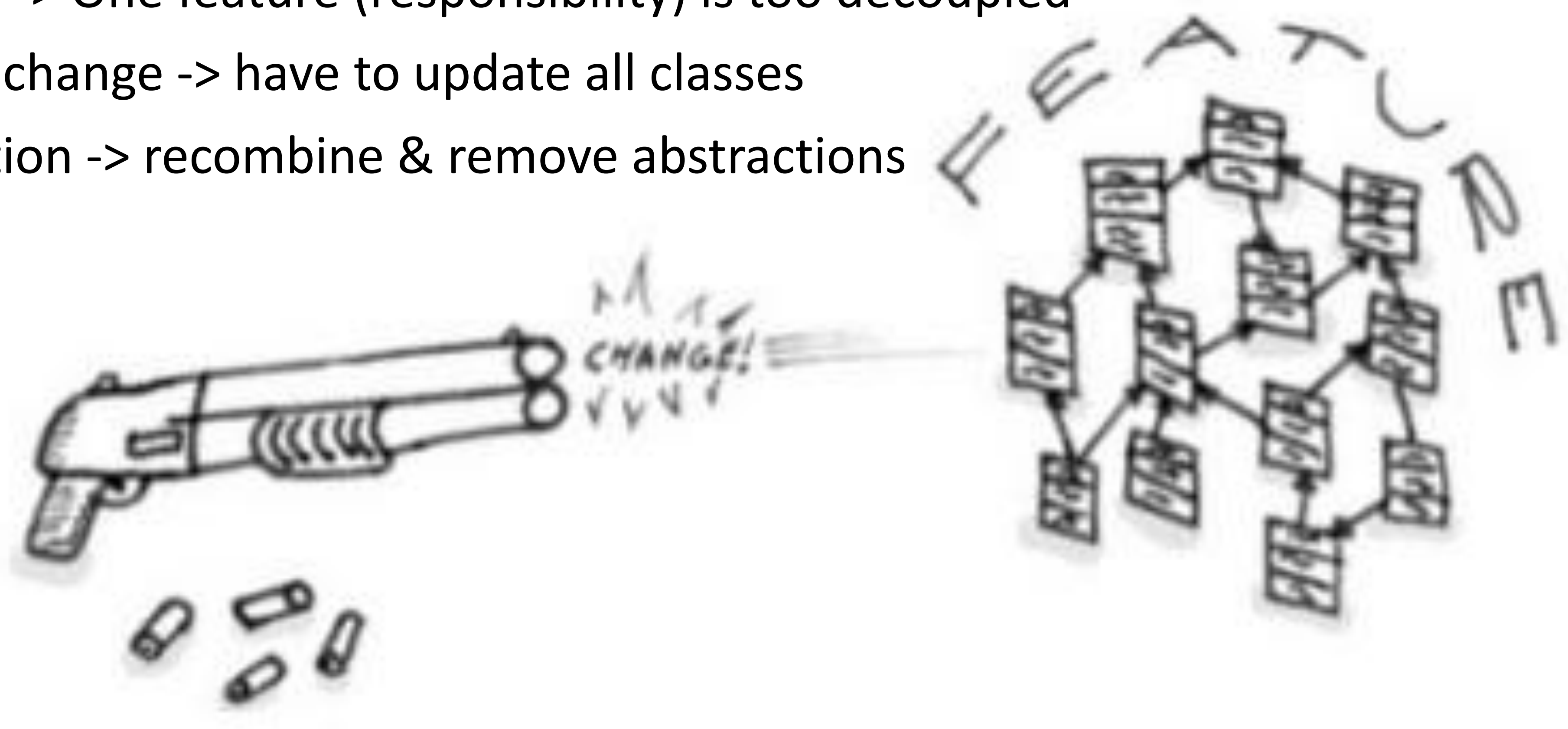


Extract Class

(Split into several smaller classes)

Example Shotgun Surgery

- Opposite of divergent change -> too many extractions
- Issue -> One feature (responsibility) is too decoupled
- New change -> have to update all classes
- Solution -> recombine & remove abstractions



Conclusions

- Refactor old code mainly to fix readability
- Design new code to avoid need for future refactoring
- Hence, refactoring fixes the past, design improvements fixes the future
- Solution to bloaters, data clumps and primitive obsession seem related, ie. new type / data class + some behaviour inside
- Divergent change and shotgun surgery are opposite extremes that we want to avoid

Sources

- https://makolyte.com/wp-content/uploads/2020/05/primitive-obsession-before-and-after.png
- https://lilitao.github.io/assets/images/code-smell-data-clumps.png
- https://i0.wp.com/thecodebuzz.com/wp-content/uploads/2019/03/Premitive-Obsession-resolution-2.png?fit=785%2C184&ssl=1
- https://makolyte.com/wp-content/uploads/2020/04/image-10.png
- https://image.slidesharecdn.com/presentation1-170116182047/95/code-smells-and-its-type-with-example-20-638.jpg?cb=1484590941
- https://ducmanhphan.github.io/img/refactoring/change-preventers/solutions-divergent-change.png
- https://giphy.com/

Any questions?

Thanks for me!

You can reach me at

- Phone: +47 41221040
- Email: jan.nygard@bouvet.no