My personal notes while reading "Refactoring: Improving the Design of Existing Code (2nd Edition)" by Martin Fowler. It only contains some basic concept as my understanding. If you want to learn more, I highly recommend you should buy the book.

If you are the publisher and think this repository should not be public, please write me an email to vuminhthang [dot] cm [at] gmail [dot] com and I will make it private.

Happy reading!

Tweet

1. TABLE OF CONTENT

3. BAD SMELLS IN CODE

1. Mysterious name

Name should clearly communicate what they do and how to use them

2. Duplicated code

Same code structure in more than one place

3. Long function

the longer a function is, the more difficult it is to understand

4. Long parameter list

Difficult to understand and easily introduce bug

5. Global data

Global variable is difficult to track and debug

6. Mutable data

Changes to data can often lead to unexpected consequences and tricky bugs

7. Divergent change

One module is changed in different ways for different reasons

8. Shotgun surgery

When every time you make a change, you have to make a lot of little edits to a lot of different classes

9. Feature envy

When a function in one module spends more time communicating with functions or data inside another module than it does within its own module

10. Data clumps

Same three or four data items together in lots of places

11. Primittive Obsession

Use primitive types instead of custom fundamental types

12. Repeated switches

Same conditional switching logic pops up in different places

13. Loops

Using loops instead of first-class functions such as filter or map

14. Lazy element

A class, struct or function that isn't doing enough to pay for itself should be eliminated.

15. Speculative generality

All sorts of hooks and special cases to handle things that aren’t required

16. Temporary field

An instance variable that is set only in certain circumstances.

17. Message chains

When a client asks one object for another object, which the client then asks for yet another object...

18. Middle man

When an object delegates much of its functionality.

19. Insider trading

Modules that whisper to each other by the coffee machine need to be separated by using Move Function and Move Field to reduce the need to chat.

20. Large class

A class is trying to do too much, it often shows up as too many fields

21. Alternative Classes with Different Interfaces

Classes with methods that look to similar.

22. Data class

Classes that have fields, getting and setting methods for the fields, and nothing else

23. Refused Bequest

Subclasses doesn't make uses of parents method

24. Comment

The comments are there because the code is bad

6. MOST COMMON SET OF REFACTORING

1. Extract Function

Extract fragment of code into its own function named after its purpose.

```
function printOwing(invoice) {
  printBanner()
  let outstannding = calculateOutstanding()

  // print details
  console.log(`name: ${invoice.customer}`)
  console.log(`amount: ${outstanding}`)
}
```

to

```
function printOwing(invoice) {
  printBanner()
  let outstannding = calculateOutstanding()
  printDetails(outstanding)

  function printDetails(outstanding) {
    console.log(`name: ${invoice.customer}`)
    console.log(`amount: ${outstanding}`)
  }
}
```

Motivation: - You know what's the code doing without reading the details - Short function is easier to read - Reduce comment

2. Inline Function

Get rid of the function when the body of the code is just as clear as the name

```
function rating(aDriver) {
  return moreThanFiveLateDeliveries(aDriver) ? 2 : 1
}
function moreThanFiveLateDeliveries(aDriver) {
  return aDriver.numberOfLateDeliveries > 5
}
```

to

```
function rating(aDriver) {
  return aDriver.numberOfLateDeliveries > 5 ? 2 : 1
}
```

Motivation - When Indirection is needless (simple delegation) becomes irritating. - If group of methods are badly factored and grouping them makes it clearer

3. Extract Variable

Add a name to an expression

```
//price is base price   quantity discount + shipping
return order.quantity * order.itemPrice -
  Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
  Math.min(order.quantity * order.itemPrice * 0.1, 100)
```

to

```
const basePrice = order.quantity * order.itemPrice
const quantityDiscount = Math.max(0, order.quantity   500) * order.itemP
const shipping = Math.min(basePrice * 0.1, 100)
return basePrice  - quantityDiscount + shipping;
```

Motivation - Break down and name a part of a more complex piece of logic - Easier for debugging

4. Inline Variable

Remove variable which doesn't really communicate more than the expression itself.

```
let basePrice = anOrder.basePrice
return (basePrice > 1000)
```

to

```
return anOrder.basePrice > 1000
```

5. Change Function Declaration

Rename a function, change list of parameters

```
function circum(radius) {...}
```

to

```
function circumference(radius) {...}
```

Motivation - Easier to understand - Easier to reuse, sometime better encapsulation

6. Encapsulate Variable

Encapsulate a reference to some data structure

```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"}
```

to

```
// defaultOwner.js
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"}
export function defaultOwner() { return defaultOwnerData }
export function setDefaultOwner(arg) { defaultOwnerData = arg }
```

Motivation - Provide a clear point to monitor changes and use of the data, like validation.

7. Rename Variable

Make shared variable's name can self-explain

```
let a = height * width
```

to

```
let area = height * width
```

8. Introduce Parameter Object

Replace groups of data items that regularly travel together with a single data structure

```
function amountInvoiced(startDate, endDate) {}
function amountReceived(startDate, endDate) {}
function amountOverdue(startDate, endDate) {}
```

to

```
function amountInvoiced(aDateRange) {}
function amountReceived(aDateRange) {}
function amountOverdue(aDateRange) {}
```

Motivation - Make explicit the relationship between the data items - Reduce the size of parameter list - Make code more consistent - Enable deeper changes to the code

9. Combine Functions Into Class

Form a class base on group of functions that operate closely on a common data

```
function base(aReading) {}
function taxableCharge(aReading) {}
function calculateBaseCharge(aReading) {}
```

to

```
class Reading() {
  base() {}
  taxableCharge() {}
  calculateBaseCharge() {}
}
```

Motivation - Simplify function call by removing many arguments - Easier to pass object to other parts of the system

10. Combine Functions Into Transform

Takes the source data as input and calculates all the derivations, putting each derived value as a field in the output data

```
function base(aReading) {}
function taxableCharge(aReading) {}
```

to

```
function enrichReading(argReading) {
  const aReading = _.cloneDeep(argReading)
  aReading.baseCharge = base(aReading)
  aReading.taxableCharge = taxableCharge(aReading)
  return aReading
}
```

Motivation - Avoid duplication of logic

11. Split Phase

Split code which do different things into separate modules

```
const orderData = orderString.split(/\s+/)
const productPrice = priceList[orderData[0].split("-")[1]]
const orderPrice = parseInt(orderData[1]) * productPrice
```

to

```
const orderRecord = parseOrder(orderString)
const orderPrice = price(orderRecord, priceList)

function parseOrder(aString) {
  const values = aString.split(/\s+/)
  return {
    productID: values[0].split("-")[1],
    quantity: parseInt(values[1])
  }
}
function price(order, priceList) {
  return order.quantity * priceList[order.productID]
}
```

Motivation - Make the different explicit, revealing the different in the code - Be able to deal with each module separately

7. ENCAPSULATION

1. Encapsulate Record

Create record (class) from object

```
organization = {name: "Acme gooseberries", country: "GB"}
```

to

```
class Organization {
  constructor(data) {
    this._name = data.name
    this._country = data.country
  }

  get name() { return this._name }
  set name(arg) { this._name = arg }
  get country() { retrun this._country }
  set country(arg) { this._country = arg }
}
```

Motivation - Hide what's stored and provide methods to get value - Easier to refactoring, for example: rename

2. Encapsulate Collection

A method returns a collection. Make it return a read-only view and provide add/remove methods

```
class Person {
  get courses() { return this._courses }
  set courses(aList) { this._courses = aList }
}
```

to

```
class Person {
  get courses() { return this._courses.slice() }
  addCourse(aCourse) {}
  removeCourse(aCourse)
}
```

Motivation - Change to the collection should go through the owning class to prevent unexpected changes. - Prevent modification of the underlying collection for example: return a copy or read-only proxy instead of collection value

3. Replace Primitive with Object

Create class for data

```
orders.filter(o => "high" === o.priority || "rush" === o.priority)
```

to

```
orders.filter(o => o.priority.higherThan(new Priority("normal")))
```

Motivation - Encapsulate behaviour with data

4. Replace Temp with Query

Extract the assignment of the variable into a function

```
 const basePrice = this._quantity * this._itemPrice
if (basePrice > 1000) {
  return basePrice * 0.95
} else {
  return basePrice * 0.98
}
```

to

```
get basePrice() { return this._quantity * this._itemPrice }
//...
if (this.basePrice > 1000) {
  return this.basePrice * 0.95
} else {
  return this.basePrice * 0.98
}
```

Motivation - Avoid duplicating the calculation logic in similar functions

5. Extract Class

Extract class base on a subset of data and a subset of methods

```
class Person {
  get officeAreaCode() { return this._officeAreaCode }
  get officeNumber() { return this._officeNumber }
}
```

to

```
class Person {
  get officeAreaCode() { return this._telephoneNumber.areaCode }
  get officeNumber() { return this._telephoneNumber.number }
}
class TelephoneNumber {
  get areaCode() { return this._areaCode }
  get number() { return this._number }
}
```

Motivation - Smaller class is easier to understand - Separate class's responsibility

6. Inline Class

Merge class if class isn't doing very much. Move its feature to another class then delete it.

```
class Person {
  get officeAreaCode() { return this._telephoneNumber.areaCode }
  get officeNumber() { return this._telephoneNumber.number }
}
class TelephoneNumber {
  get areaCode() { return this._areaCode }
  get number() { return this._number }
}
```

to

```
class Person {
  get officeAreaCode() { return this._officeAreaCode }
  get officeNumber() { return this._officeNumber }
}
```

Motivation - Class is no longer pulling its weight and shouldn’t be around any more - When want to refactor pair of classes. First Inline Class -> Extract Class to make new separation

7. Hide Delegate

A client is calling a delegate class of an object, create methods on the server to hide the delegate.

```
manager = aPerson.department.manager
```

to

```
manager = aPerson.manager

class Person {
  get manager() {
    return this.department.manager
  }
}
```

Motivation - Client doesn't need to know and response to delegation's change - Better encapsulation

8. Remove Middle Man

Client call the delegate directly

```
manager = aPerson.manager

class Person {
  get manager() {
    return this.department.manager
  }
}
```

to

```
manager = aPerson.department.manager
```

Motivation - When there are too many delegating methods

9. Substitute Algorithm

Replace complicated algorithm with simpler algorithm

```
function foundPerson(people) {
  for (let i = 0; i < people.lenth; i++) {
    if (people[i] === "Don") {
      return "Don"
    }
    if (people[i] === "John") {
      return "John"
    }
    if (people[i] === "Kent") {
      return "Kent"
    }
  }
  return ""
}
```

to

```
function foundPerson(people) {
  const candidates = ["Don", "John", "Kent"]
  return people.find(p => candidates.includes(p)) || ""
}
```

Motivation - Change to algorithm which make changes easier - The clearer algorithm is, the better.

8. MOVING FEATURES

1. Move Function

Move a function when it references elements in other contexts more than the one it currently resides in

```
class Account {
  get overdraftCharge() {}
}
class AccountType {}
```

to

```
class Account {}
class AccountType {
  get overdraftCharge() {}
}
```

Motivation - Improve encapsulation, loose coupling

2. Move Field

Move field from once class to another

```
class Customer {
  get plan() { return this._plan }
  get discountRate() { return this._discountRate }
}
```

to

```
class Customer {
  get plan() { return this._plan }
  get discountRate() { return this.plan.discountRate }
}
```

Motivation - Pieces of data that are always passed to functions together are usually best put in a single record - If a change in one record causes a field in another record to change too, that's a sign of a field in the wrong place

3. Move Statements into Function

When statement is a part of called functions (always go togeter), move it inside the function

```
result.push(`<p>title: ${person.photo.title}</p>`)
result.concat(photoData(person.photo))

function photoData(aPhoto) {
  return [
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.data.toDateString()}</p>`
  ]
}
```

to

```
result.concat(photoData(person.photo))

function photoData(aPhoto) {
  return [
    `<p>title: ${aPhoto.title}</p>`,
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.data.toDateString()}</p>`
  ]
}
```

Motivation - Remove duplicated code

4. Move Statements To Callers

```
 emitPhotoData(outStream, person.photo)

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`)
  outStream.write(`<p>location: ${photo.location}</p>\n`)
}
```

to

```
 emitPhotoData(outStream, person.photo)
outStream.write(`<p>location: ${photo.location}</p>\n`)

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`)
}
```

Motivation - When common behavior used in several places needs to vary in some of its call

5. Replace Inline Code with Function Call

Replace the inline code with a call to the existing function

```
 let appliesToMass = false
for (const s of states) {
  if (s === "MA") appliesToMass = true
}
```

```
 appliesToMass = states.includes("MA")
```

Motivation - Remove duplication - Meaningful function name is easier to understand

6. Slide Statements

Move related code to near each other

```
 const pricingPlan = retrievePricingPlan()
const order = retreiveOrder()
let charge
const chargePerUnit = pricingPlan.unit
```

to

```
 const pricingPlan = retrievePricingPlan()
const chargePerUnit = pricingPlan.unit
const order = retreiveOrder()
let charge
```

Motivation - It makes code easier to understand and easier to extract function

7. Split Loop

Split the loop which does two different things

```
 let averageAge = 0
let totalSalary = 0
for (const p of people) {
  averageAge += p.age
  totalSalary += p.salary
}
averageAage = averageAge / people.length
```

to

```
let totalSalary = 0
for (const p of people) {
  totalSalary += p.salary
}

let averageAge = 0
for (const p of people) {
  averageAge += p.age
}
averageAge = averageAge / people.length
```

Motivation - Easier to use - Easier to understand because each loop will do only 1 thing

8. Replace Loop with Pipeline

Replace loop with collection pipeline, like `map` or `filter`

```
const names = []
for (const i of input) {
  if (i.job === "programmer") {
    names.push(i.name)
  }
}
```

to

```
const names = input.filter(i => i.job === "programmer").
                    map(i => i.name)
```

Motivation - Easier to understand the flow of data

9. Remove Dead Code

```
if (false) {
  doSomethingThatUsedToMatter()
}
```

to

```

```

Motivation - Easier and quicker for developer to understand the codebase

9. ORGANIZING DATA

1. Split Variable

Any variable with more than one responsibility should be replaced with multiple variables, one for each responsibility

```
let temp = 2 * (height + width)
console.log(temp)
temp = height * width
console.log(temp)
```

to

```
const perimeter = 2 * (height + width)
console.log(perimeter)
const area = height * width
console.log(area)
```

Motivation - Easier to understand

2. Rename Field

```
class Organization {
  get name() {}
}
```

to

```
class Organization {
  get title() {}
}
```

3. Replace Derived Variable With Query

Remove anny variables which cloud be easily calculate

```
get discountedTotal() { return this._discountedTotal }
set discount(aNumber) {
  const old = this._discount
  this._discount = aNumber
  this._discountedTotal += old - aNumber
}
```

to

```
get discountedTotal() { return this._baseTotal - this._discount }
set discount() { this._discount = aNumber }
```

Motivation - Minimize scope of mutable data - A calculate makes it clearer what the meaning of data is

4. Change Reference To Value

Treat data as value. When update, replace entire inner object with a new one

```
class Product {
  applyDiscount(arg) {
    this._price.amount -= arg
  }
}
```

to

```
class Product {
  applyDiscount(arg) {
    this._price = new Money(this._price.amount - arg, this._price.currency)
  }
}
```

Motivation - Immutable data is easier to deal with

5. Change Value To Reference

When need to share an object in different place, or have duplicated objects

```
let customer = new Customer(customerData)
```

to

```
let customer = customerRepository.get(customerData.id)
```

Motivation - Update one reference is easier and more consistent than update multiple copies

10. SIMPLIFYING CONDITIONAL LOGIC

1. Decompose Conditional

Decomposing condition and replacing each chunk of code with a function call

```
if (!aDate.isBefore(plan.summerStart) && !aData.isAfter(plan.summerEnd)) {
  charge = quantity * plan.summerRate
} else {
  charge = quantity * plan.regularRate + plan.regularServiceCharge
}
```

to

```
if (summer()) {
  charge = summerCharge()
} else {
  charge = regularCharge()
}
```

Motivation - Clearer intention of what we're branching on

2. Consolidate Conditional Expression

Consolidate different condition check which the result action is same to a single condition check with single result

```
if (anEmployee.seniority < 2) return 0
if (anEmployee.monthsDisabled > 12) return 0;
if (anEmployee.isPartTime) return 0
```

to

```
if (isNotEligableForDisability()) return 0

function isNotEligableForDisability() {
   return ((anEmployee.seniority < 2)
    || (anEmployee.monthsDisabled > 12)
    || (anEmployee.isPartTime))
}
```

Motivation - Often lead to Extract Function, which reveal intead of the code by function name - If conditions are not related, don't consolidate them

3. Replace Nested Conditional with Guard Clauses

If condition is unusual condition, early return (Guard Clauses) and exist the function

```
function getPayAmount() {
  let result
  if (isDead) {
    result = deadAmount()
  } else {
    if (isSeparated) {
      result = separatedAmount()
    } else {
      if (isRetired) {
        result = retiredAmount()
      } else {
        result = normalPayAmount()
      }
    }
  }
  return result
}
```

to

```
function getPayAmount() {
  if (isDead) return deadAmount()
  if (isSeparated) return separatedAmount()
  if (isRetired) return retiredAmount()
  return normalPayAmount()
}
```

Motivation - It shows conditional branch are normal or unusual

4. Replace Conditional with Polymorphism

Using object oriented class instead of complex condition

```
switch (bird.type) {
  case 'EuropeanSwallow':
    return 'average'
  case 'AfricanSwallow':
    return bird.numberOfCoconuts > 2 ? 'tired' : 'average'
  case 'NorwegianBlueParrot':
    return bird.voltage > 100 ? 'scorched' : 'beautiful'
  default:
    return 'unknow'
}
```

to

```
class EuropeanSwallow {
  get plumage() {
    return 'average'
  }
}
class AfricanSwallow {
  get plumage() {
    return this.numberOfCoconuts > 2 ? 'tired' : 'average'
  }
}
class NorwegianBlueParrot {
  get plumage() {
    return this.voltage > 100 ? 'scorched' : 'beautiful'
  }
}
```

Motivation - Make the separation more explicit

5. Introduce Special Case

Bring special check case to a single place

```
if (aCustomer === "unknown") {
  customerName = "occupant"
}
```

to

```
class UnknownCustomer {
  get name() {
    return "occupant"
  }
}
```

Motivation - Remove duplicate code - Easy to add additional behavior to special object

6. Introduce Assertion

Make the assumption explicit by writing an assertion

```
if (this.discountRate) {
  base = base - (this.discountRate * base)
}
```

to

```
assert(this.discountRate >= 0)
if (this.discountRate) {
  base = base - (this.discountRate * base)
}
```

Motivation - Reader can understand the assumption easily - Help in debugging

11. REFACTORING APIS

1. Separate Query from Modifier

Separate function that returns a value (query only) and function with side effects (example: modify data)

```
function alertForMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms()
      return "Don"
    }
    if (p === "John") {
      setOffAlarms()
      return "John"
    }
  }
  return ""
}
```

to

```
function findMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      return "Don"
    }
    if (p === "John") {
      return "John"
    }
  }
  return ""
}
function alertForMiscreant (people) {
  if (findMiscreant(people) !== "") setOffAlarms();
}
```

Motivation - Immutable function (query only) is easy to test and reuse

2. Parameterize Function

Combine function with similar logic and different literal value

```
function tenPercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.1)
}
function fivePercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.05)
}
```

to

```
function raise(aPerson, factor) {
  aPerson.salary = aPerson.salary.multiply(1 + factor)
}
```

Motivation - Increase usefulness of the function

3. Remove Flag Argument

Remove *literal* flag argument by clear name functions

```
function setDimension(name, value) {
  if (name === 'height') {
    this._height = value
    return
  }
  if (name === 'width') {
    this._width = value
    return
  }
}
```

to

```
function setHeight(value) { this._height = value }
function setWidth(value) { this._width = value }
```

Motivation - Easy to read and understand code - Be careful if flag argument appears more than 1 time in the function, or is passed to further function

4. Preserve Whole Object

Passing whole object instead of multiple parameters

```
const low = aRoom.daysTempRange.low
const high = aRoom.daysTempRange.high
if (aPlan.withinRange(low, high)) {}
```

to

```
if (aPlan.withInRange(aRoom.daysTempRange)) {}
```

Motivation - Shorter parameter list - Don't need to add additional parameter if function needs more data in the future - Be careful if function and object are in different modules, which make tight coupling if we apply this refactor

5. Replace Parameter with Query

```
availableVacation(anEmployee, anEmployee.grade)

function availableVacation(anEmployee, grade) {
  // calculate vacation...
}
```

to

```
availableVacation(anEmployee)

function availableVacation(anEmployee) {
  const grade = anEmployee.grade
  // calculate vacation...
}
```

Motivation - Shorter list of parameters - Simpler work for the caller (because fewer parameters) - Be careful because it can increase function's dependency

6. Replace Query with Parameter

Replace internal reference with a parameter

```
targetTemperature(aPlan)

function targetTemperature(aPlan) {
  currentTemperature = thermostat.currentTemperature
  // rest of function...
}
```

to

```
targetTemperature(aPlan, thermostat.currentTemperature)

function targetTemperature(aPlan, currentTemperature) {
  // rest of function...
}
```

Motivation - Reduce function's dependency - Create more pure functions

7. Remove Setting Method

Make a field immutable by removing setting method

```
class Person {
  get name() {...}
  set name(aString) {...}
}
```

to

```
class Person {
  get name() {...}
}
```

8. Replace Constructor with Factory Function

```
leadEngineer = new Employee(document.leadEngineer, 'E')
```

to

```
leadEngineer = createEngineer(document.leadEngineer)
```

Motivation - You want to do more than simple construction when you create an object.

9. Replace Function with Command

Encapsulate function into its own object

```
function score(candidate, medicalExam, scoringGuide) {
  let result = 0
  let healthLevel = 0
  // long body code
}
```

to

```
class Scorer {
  constructor(candidate, medicalExam, scoringGuide) {
    this._candidate = candidate
    this._medicalExam = medicalExam
    this._scoringGuide = scoringGuide
  }

  execute() {
    this._result = 0
    this._healthLevel = 0
    // long body code
  }
}
```

Motivation - You want to add complimentary operation, such as undo - Can have richer lifecycle - Can build customization such as inheritance and hooks - Easily break down a complex function to simpler steps

10. Replace Command with Function

```
class ChargeCalculator {
  constructor (customer, usage){
    this._customer = customer
    this._usage = usage
  }
  execute() {
    return this._customer.rate * this._usage
  }
}
```

to

```
function charge(customer, usage) {
  return customer.rate * usage
}
```

Motivation - Function call is simpler than command object

12. DEALING WITH INHERITANCE

1. Pull Up Method

Move similar methods in subclass to superclass

```
class Employee {...}

class Salesman extends Employee {
  get name() {...}
}

class Engineer extends Employee {
  get name() {...}
}
```

to

```
class Employee {
  get name() {...}
}

class Salesman extends Employee {...}
class Engineer extends Employee {...}
```

Motivation - Eliminate duplicate code in subclass - If two methods has similar workflow, consider using Template Method Pattern

2. Pull Up Field

Pull up similar field to superclass to remove duplication

```
class Employee {...}

class Salesman extends Employee {
  private String name;
}

class Engineer extends Employee {
  private String name;
}
```

to

```
class Employee {
  protected String name;
}

class Salesman extends Employee {...}
class Engineer extends Employee {...}
```

3. Pull Up Constructor Body

You have constructors on subclasses with mostly identical bodies. Create a superclass constructor; call this from the subclass methods

```
class Party {...}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super()
    this._id = id
    this._name = name
    this._monthlyCost = monthlyCost
  }
}
```

to

```
class Party {
  constructor(name){
    this._name = name
  }
}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super(name)
    this._id = id
    this._monthlyCost = monthlyCost
  }
}
```

4. Push Down Method

If method is only relevant to one subclass, moving it from superclass to subclass

```
class Employee {
  get quota {...}
}

class Engineer extends Employee {...}
class Salesman extends Employee {...}
```

to

```
class Employee {...}
class Engineer extends Employee {...}
class Salesman extends Employee {
  get quota {...}
}
```

5. Push Down Field

If field is only used in one subclass, move it to those subclasses

```
class Employee {
  private String quota;
}

class Engineer extends Employee {...}
class Salesman extends Employee {...}
```

to

```
class Employee {...}
class Engineer extends Employee {...}

class Salesman extends Employee {
  protected String quota;
}
```

6. Replace Type Code with Subclasses

```
function createEmployee(name, type) {
  return new Employee(name, type)
}
```

to

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name)
    case "salesman": return new Salesman(name)
    case "manager":  return new Manager (name)
  }
}
```

Motivation - Easily to apply Replace Conditional with Polymorphism later - Execute different code depending on the value of a type

7. Remove Subclass

You have subclasses do to little. Replace the subclass with a field in superclass.

```
class Person {
  get genderCode() {return "X"}
}
class Male extends Person {
  get genderCode() {return "M"}
}
class Female extends Person {
  get genderCode() {return "F"}
}
```

to

```
class Person {
  get genderCode() {return this._genderCode}
}
```

8. Extract Superclass

If 2 classes have similar behaviors, create superclass and move these behaviors to superclass

```
class Department {
  get totalAnnualCost() {...}
  get name() {...}
  get headCount() {...}
}

class Employee {
  get annualCost() {...}
  get name() {...}
  get id() {...}
}
```

to

```
class Party {
  get name() {...}
  get annualCost() {...}
}

class Department extends Party {
  get annualCost() {...}
  get headCount() {...}
}

class Employee extends Party {
  get annualCost() {...}
  get id() {...}
}
```

Motivation - Remove duplication - Prepare for Replace Superclass with Delegate refactor

9. Collapse Hierarchy

Merge superclass and subclass when there are no longer different enough to keep them separate

```
class Employee {...}
class Salesman extends Employee {...}
```

to

```
class Employee {...}
```

10. Replace Subclass with Delegate

"Favor object composition over class inheritance" (where composition is effectively the same as delegation

```
class Order {
  get daysToShip() {
    return this._warehouse.daysToShip
  }
}

class PriorityOrder extends Order {
  get daysToShip() {
    return this._priorityPlan.daysToShip
  }
}
```

to

```
class Order {
  get daysToShip() {
    return (this._priorityDelegate)
      ? this._priorityDelegate.daysToShip
      : this._warehouse.daysToShip
  }
}

class PriorityOrderDelegate {
  get daysToShip() {
    return this._priorityPlan.daysToShip
  }
}
```

Motivation - If there are more than 1 reason to vary something, inheritance is not enough - Inheritance introduce very close relationship

11. Replace Superclass with Delegate

If functions of the superclass don't make sense on the subclass, replace with with delegate

```
class List {...}
class Stack extends List {...}
```

to

```
class Stack {
  constructor() {
    this._storage = new List();
  }
}
class List {...}
```

Motivation - Easier to maintain code