Introduction of Test Driven Development
(1st day)

for Java

株式会社テクノロジックアート
TECHNOLOGIC ARTS INCORPORATED

Purpose of the Course

What to Learn

- Meaning and effectiveness of test driven development and refactoring
- Basic steps and techniques of test driven development and refactoring

Prerequisite

- general knowledge of Agile development
- knowledge of Java programming language

Course Schedule

- 2 days

Learning Content

- You will learn why you should practice test driven development.

Agenda(1st day)

Chapter 1 Overview of test driven development

1.1. Definition and how to proceed TDD
1.2. Essence of TDD
1.3. Summary

Chapter 2 Exercise: Addition

2.1. Exercise
2.2. Summary

Chapter3 Overview of Refactoring

3.1. Two values of software
3.2. What is Refactoring
3.3. Refactoring by tool
3.4. Summary

Chapter 4 Exercise: Calculation of party price

4.1. Basics
4.2. Advanced 1
4.3. Advanced 2
4.4. Summary

Chapter 5 Summary of the Day

Chapter 1 Overview of TDD

What to learn in this chapter

- Definition and how to proceed TDD
  - Cycle of TDD
    - Write a test
    - Make it compile
    - Red
    - Green
    - Refactor

- Essence of TDD
  - It is a design technique.
  - It piles up reliability.

Chapter 1 Overview of TDD

1.1 **Definition of TDD**

1.2 Essence of TDD

1.3 Summary

What is test driven development(TDD)?

Test

Driven

Development

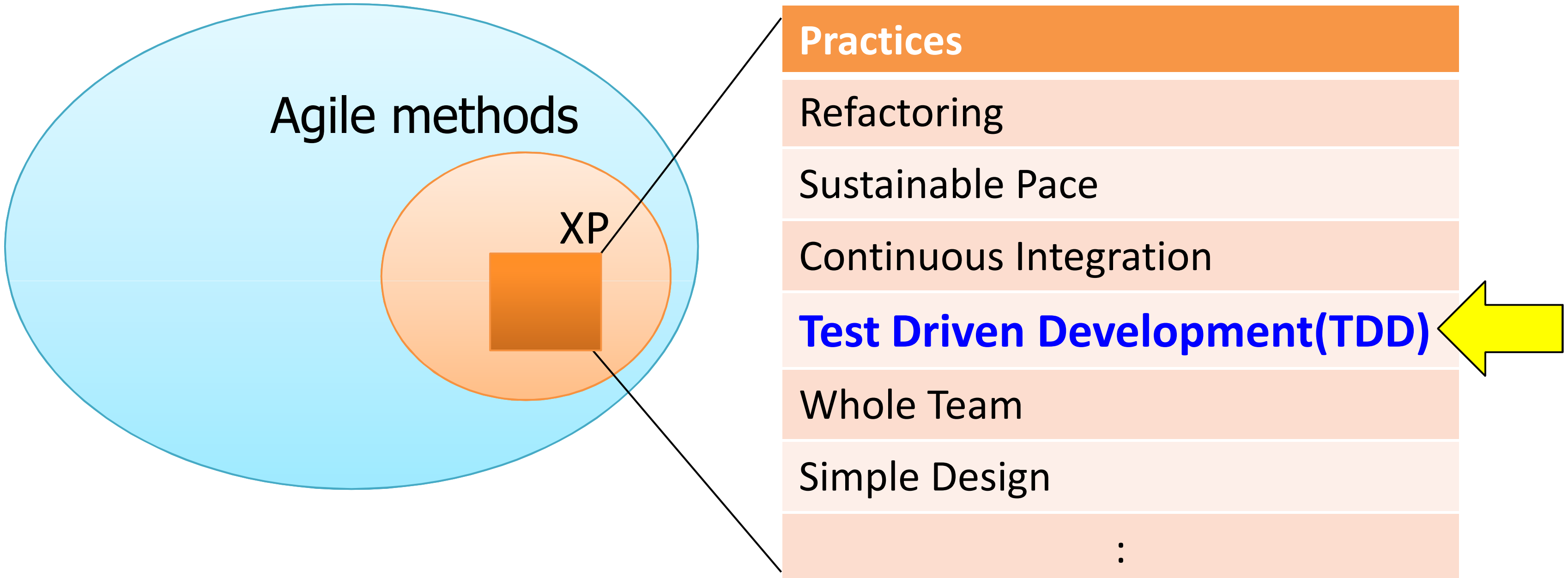
↓

This is a development method.

(It is not a testing method.)

What is TDD? A practice of XP.

- It is one of XP practice.



Practices
Refactoring
Sustainable Pace
Continuous Integration
Test Driven Development(TDD)
Whole Team
Simple Design
:

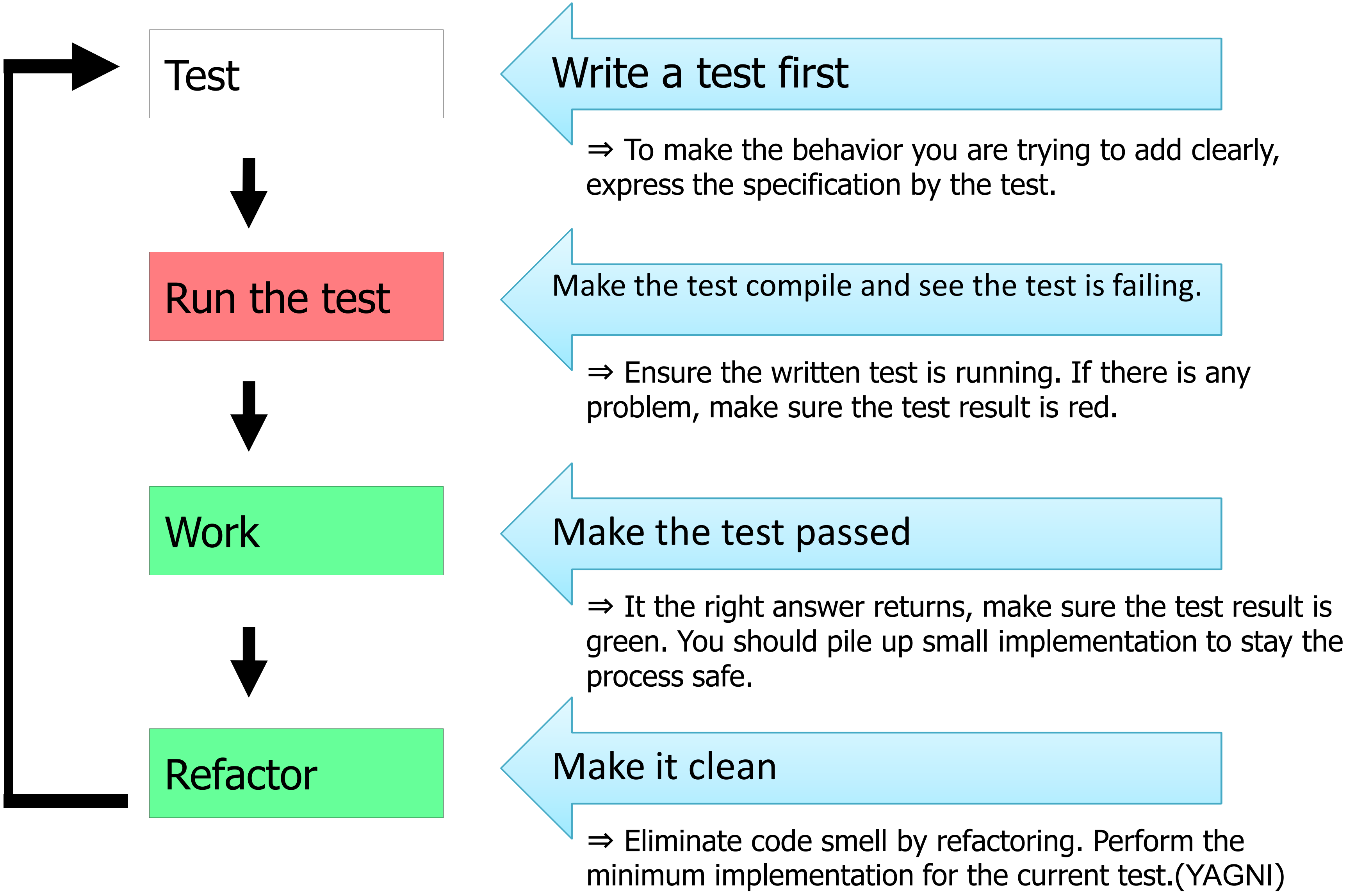
It is tightly concerned with other XP practices.

What is TDD? More concretely.

- Definition
  - Development method to design and build programs by repeating the small steps of test and implementation to produce each small function of software.

- How to proceed
  - Manage implementing functions with to do lists.
    - Write down behaviors needed and tasks to notepads as to-dos.
  - For each to-do, repeat the following
    - Firstly write a test.
    - Run the test ⇒ failed (RED)
    - Next write a production code to pass the test.
    - Run the test ⇒ passed (GREEN)
    - Refactor the code to the code as it should be.
    - Run the test ⇒ ensure it still passes (REFACTOR)

What is TDD? Cycle of TDD



Test

Run the test

Work

Refactor

Write a test first

⇒ To make the behavior you are trying to add clearly, express the specification by the test.

Make the test compile and see the test is failing.

⇒ Ensure the written test is running. If there is any problem, make sure the test result is red.

Make the test passed

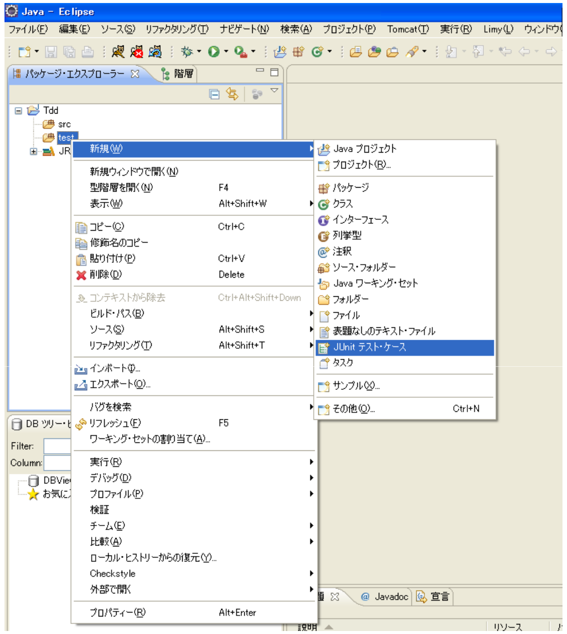
⇒ It the right answer returns, make sure the test result is green. You should pile up small implementation to stay the process safe.

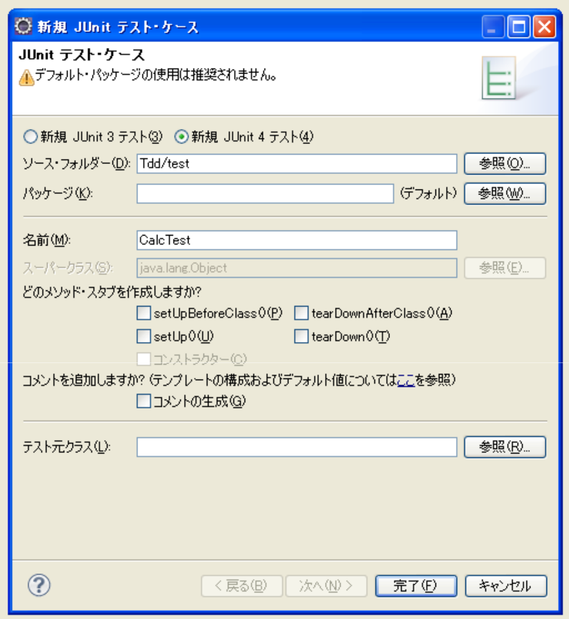
Make it clean

⇒ Eliminate code smell by refactoring. Perform the minimum implementation for the current test.(YAGNI)

JUnit

Write a Test Case class





JUnit テスト・ケース
デフォルト・パッケージの使用は推奨されません。

○ 新規 JUnit 3 テスト(3) ● 新規 JUnit 4 テスト(4)

ソース・フォルダー(D): Tdd/test

パッケージ(K): (デフォルト)

名前(M): CalcTest

スーパークラス(S): java.lang.Object

どのメソッド・スタブを作成しますか?

□ setUpBeforeClass()(P) □ tearDownAfterClass()(A)
□ setUp()(U) □ tearDown()(T)
□ コンストラクター(C)

コメントを追加しますか? (テンプレートの構成およびデフォルト値についてはここを参照)
□ コメントの生成(G)

テスト元クラス(L):

Select the package for test and right click.

↓

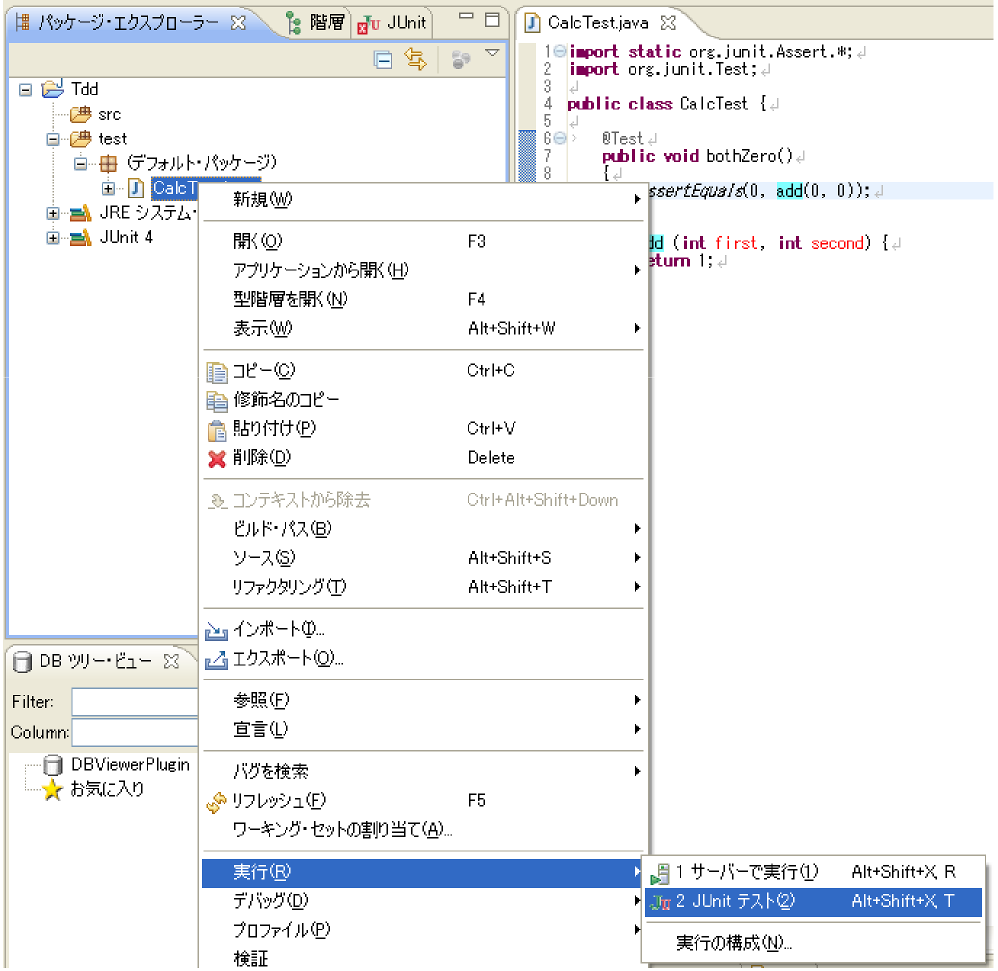
New → JUnit Test Case（JUnit4）

↓

Enter the class name and click finish.

JUnit: The case of test failure

- Run a JUnit test

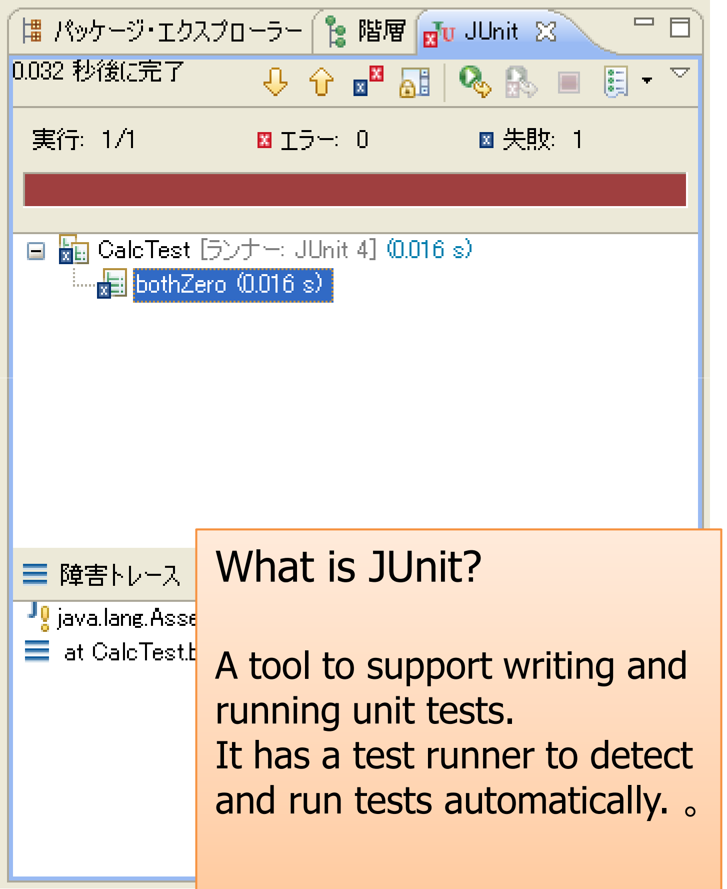


```
import static org.junit.Assert.*;
import org.junit.Test;

public class CalcTest {

    @Test
    public void bothZero()
    {
ssertEquals(0, add(0, 0));

d(int first, int second) {
eturn 1;
```



0.032 秒後に完了

実行: 1/1 ☒ エラー: 0 ☒ 失敗: 1

CalcTest [ランナー: JUnit 4] (0.016 s)
bothZero (0.016 s)

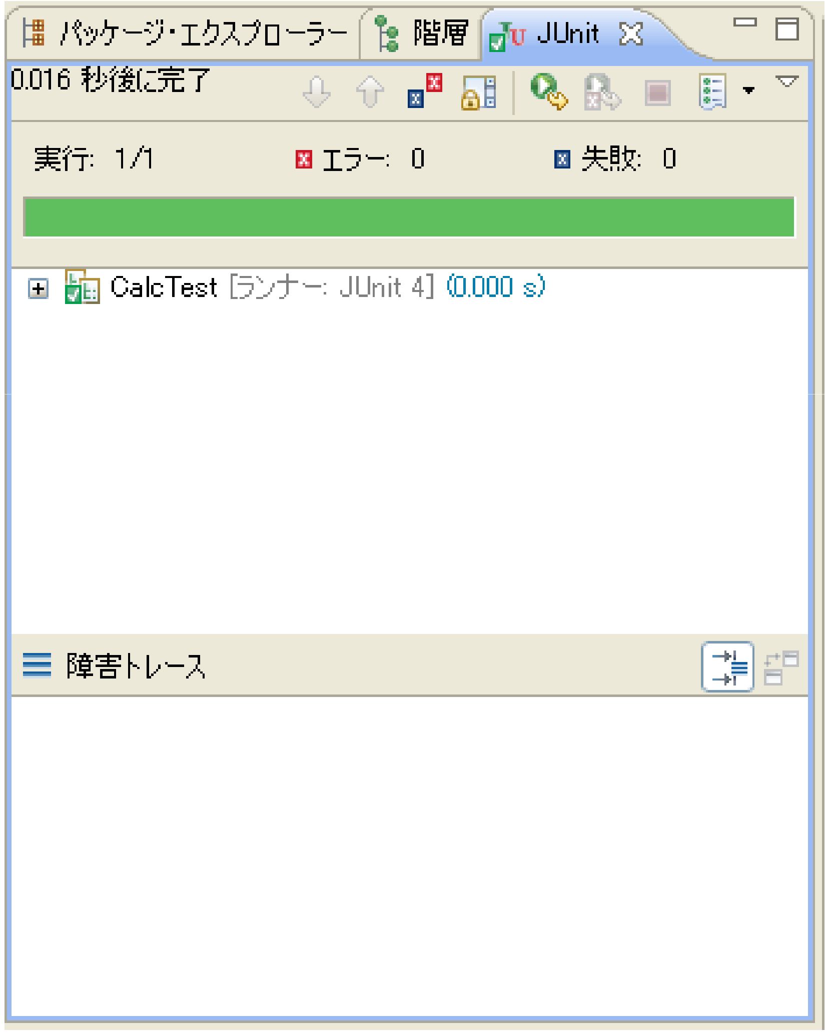
java.lang.Asse
at CalcTest.b

What is JUnit?

A tool to support writing and running unit tests.
It has a test runner to detect and run tests automatically. 。

JUnit: Test passed

- Fix the problem and run the test



0.016 秒後に完了

実行: 1/1　　☒ エラー: 0　　☒ 失敗: 0

CalcTest [ランナー: JUnit 4] (0.000 s)

Chapter 1 Overview of TDD

1.1 Definition of TDD

1.2 Essence of TDD

1.3 Summary

Essence of Test Driven Development（TDD）

Design technique

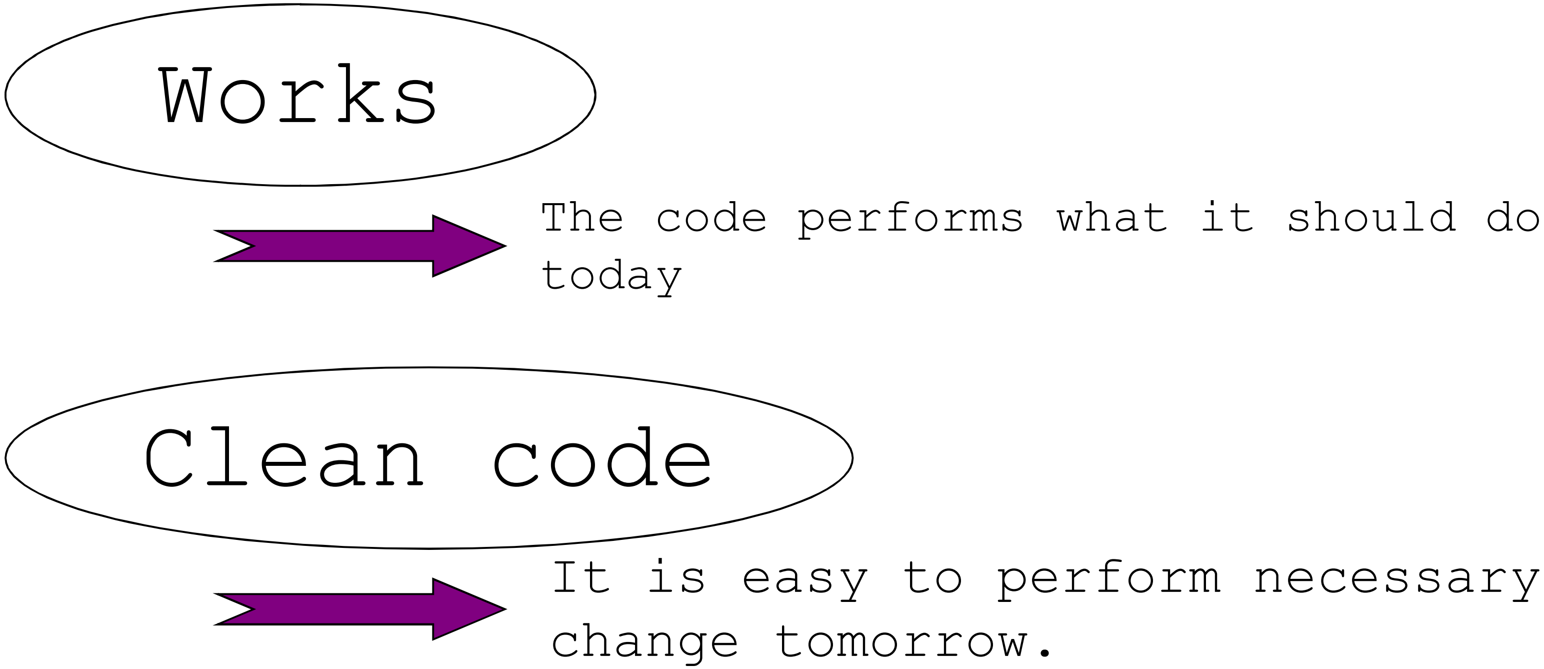
- Though writing tests, make how the production code behaviors clearer.
- It assumes evolutional design.

Development approach to pile up reliability

- It is a technique to encourage development by piling up verifiable implementation little by little, which is possible only by the tests.
- Developers write the tests by themselves.
- Developers decides the range to test.
- To stay a fixed rhythm, it needs rapid build and test environment.

Essence of TDD：A Design Method①

- TDD is a designing task to make behaviors clear.
  - First, clarify visible behaviors from the outside.
    - By writing tests, consider behaviors of classes and their methods.
  - Keeping correct behaviors, evolve the inside step by step.
    - Clean code that it works



Works

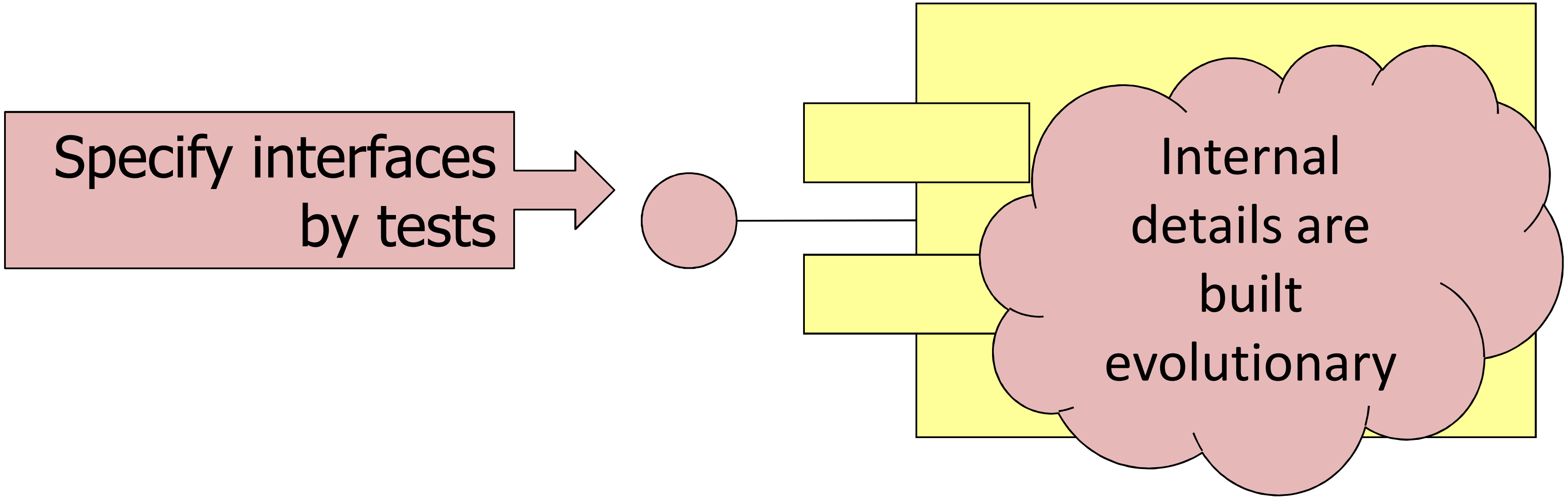
The code performs what it should do today

Clean code

It is easy to perform necessary change tomorrow.

Essence of TDD：A Design Method ②

- TDD assumes evolutionary design.
  - Not design details beforehand,
    details are designed during the implementation(Design by Code)

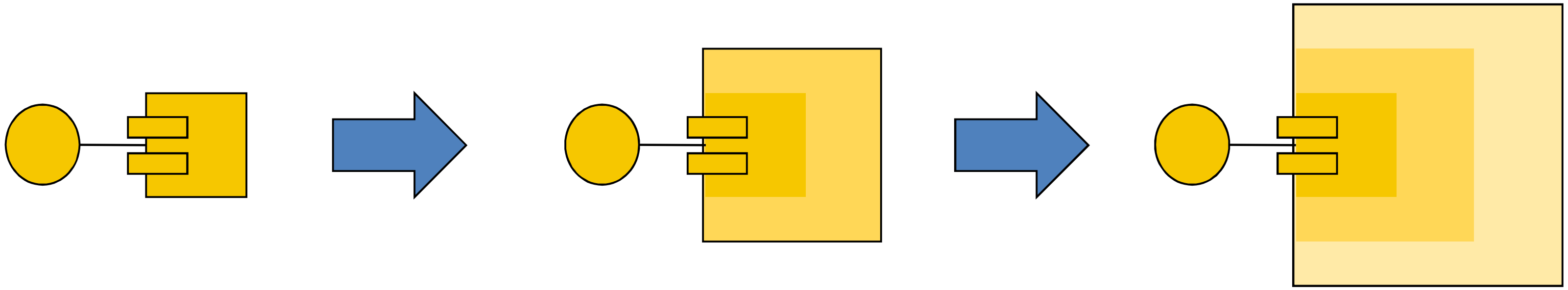


Specify interfaces by tests

Internal details are built evolutionary

Essence of TDD : Pile up reliabilities①

- Technique to encourage development
  - Characteristics
    - Traditional
      - Whole reliability is improved in the later stages(V Curve)
    - TDD
      - Pile up reliable parts(Iterative)
  - Verifiability by automated tests
    - Ensure not to degrade by making the tests executable in their own and repeating verification.



Essence of TDD : Pile up reliabilities ②

- Developers write their tests by themselves.
  - They are not the tests written by testers to assure the qualities.
  - They are the tests written by developers to make progress of the development

- Developers decides the test coverage.

Tests for quality assurance

Author:
  Testers

Target:
  Implementation code which have been written

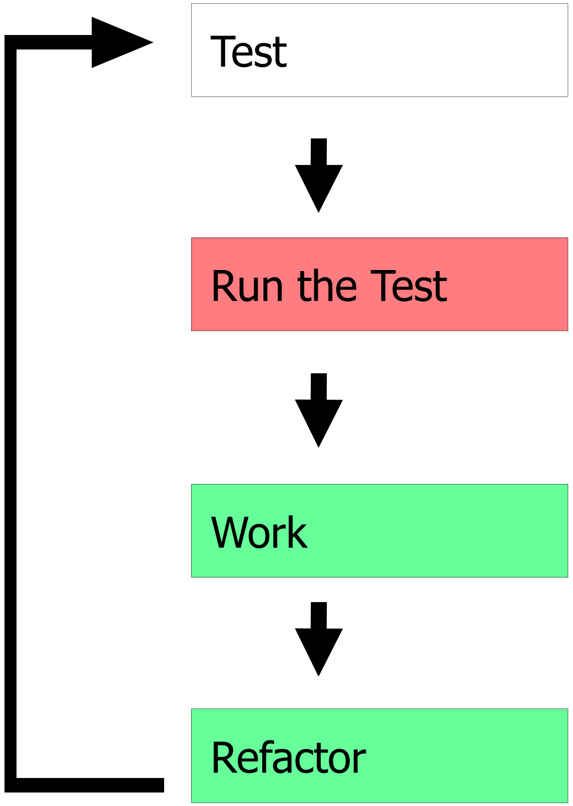
TDD tests

Author :
  Developers

Target:
  Code which does not exist yet at the beginning of the test

Essence of TDD : Pile up reliabilities ③

- it needs rapid build and test environment.
  - Writing a little by little, repeat small cycles of compiles, tests, and running.
  - Building and testing must be automated and the required time needs to be short.



Chapter 1 Overview of TDD

1.1 Definition of TDD

1.2 Essence of TDD

1.3 **Summary**

What to learn in this chapter(again)

- Definition and how to proceed TDD
  - Cycle of TDD
    - Write a test
    - Make it compile
    - Red
    - Green
    - Refactor

- Essence of TDD
  - It is a design technique.
  - It piles up reliability.

Overview of TDD : Summary

Definition

- A development approach to design and develop programs by repeating small steps of tests and implementations for every small features of the software.

Essence

- It is a design technique.
- A development method to pile up reliabilities.

How to proceed

- Repeat tests and implementations little by little, using some tools.

Chapter 2 Exercise: Addition

What to learn in this chapter

- Basic steps of TDD
  - Write a test
  - Make the test compiled
  - Red
  - Green
  - Refactor

- How to use to-do lists

Chapter 2 Exercise: Addition

2.1 **Exercise**

2.2 Summary

Getting ready: Prepare Eclipse

- Ensure JUnit is available

  In Package Explore, right click a project
  => New => JUnit Test Case

  - If it is installed, check the plug-in running
  - It not installed, install it.

- Create a new Java project

  Project name: Addition

  - Configure Build Path for JUnit

  Libraries tab in Java Build Path => Add Library => select JUnit
  => select JUnit 4

Exercise : Addition — List of test cases

- List of test cases
  - Both numbers are zero.
  - Only first number is zero.
  - Only second number is zero.
  - Neither is zero.

Once getting ready, let’s start exercise

Make it the to-do list.

Exercise : Addition — What to consider first

- First target is “Both numbers are zero.”
- There needs a method for addition.
  - The method has two arguments of number.
  - It returns the result.
- Express the above things as a test.

<u>Both numbers are zero.</u>
Only first number is zero.
Only second number is zero.
Neither is zero.

Exercise : Addition — The first test code

- Write a test code first
- *assertEquals*() raise an exception when the two arguments are not equal.
- Cannot compile yet

Both numbers are zero. Only first number is zero. Only second number is zero. Neither is zero.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class CalcTest
{
    @Test
    public void bothZero()
    {
        assertEquals(0, add(0, 0));
    }
}
```

Basics of JUnit

- Declare tests
  - Annotate test methods with @Test.

- Assertion
  - assertEquals(expected, actual)
  - assertTrue (condition)
  - assertFalse (condition)

  and others. There are many assertion methods.

For reference : Annotation

- Add meta data to classes or methods
- Describe it as “@･･･” before declaration of classes or methods.

- Example

```
@Test
public void bothZero()
{
    assertEquals(0, add(0, 0));
}
```

- Explanation
  - In this example, the annotation shows that bothZero is a test method.

Exercise：Addition — Make it compile

- Write add method

```
int add (int first, int second)
{
   return 1;
}
```

Both numbers are zero.
Only first number is zero.
Only second number is zero.
Neither is zero.

  - Compile it and run the test

🟥

First write a failing test

⇒ Ensure the written test is running. If there is any problem, make sure the test result is red.

Exercise : Addition — Make the test passed

- Make the test passed

```
int add (int first, int second)
{
    return 0;
}
```

<u>Both numbers are zero.</u>
Only first number is zero.
Only second number is zero.
Neither is zero.

- Compile it and run the test

🟩

Write a code which passes the test definitely

⇒ It the right answer returns, make sure the test result is green. You should pile up small implementation safely.

Exercise：Addition — The second test

- Add the test

```
@Test
public void firstZero()
{
    assertEquals(3, add(0, 3));
}
```

~~Both numbers are zero.~~
Only first number is zero.
Only second number is zero.
Neither is zero.

  - Compile it and run the test 🟥

- Fix the add method.

```
int add (int first, int second)
{
    return second;
}
```

  - Compile it and run the test 🟩

Exercise : Addition — The third test

~~Both numbers are zero.~~
~~Only first number is zero.~~
Only second number is zero.
Neither is zero.

- Add the test

```
@Test
public void secondZero()
{
    assertEquals(2, add(2, 0));
}
```

  - Compile it and run the test 🟥

- Fix the add method.

```
int add (int first, int second)
{
    if (second == 0) return first;
    return second;
}
```

  - Compile it and run the test 🟩

Exercise : Addition　—　The final test

~~Both numbers are zero.~~
~~Only first number is zero.~~
~~Only second number is zero.~~
Neither is zero.

- Add the test

```
@Test
public void bothNotZero()
{
    assertEquals(5, add(2, 3));
}
```

  - Compile it and run the test

🟥

- Fix the add method.

```
int add (int first, int second)
{
    if (first != 0 && second != 0) return first + second;
    if (second == 0) return first;
    return second;
}
```

  - Compile it and run the test

🟩

Exercise : Addition — Refactor

- Fix the add method.

```
int add (int first, int second) {
    if (first != 0 && second != 0) return first + second;
    if (second == 0) return first;
    return second;
}
```

If second == 0, first equals to first + second

```
int add (int first, int second) {
    if (first != 0 && second != 0) return first + second;
    if (second == 0) return first + second;
    return second;
}
```

🟩

If first == 0, second equals to first + second

```
int add (int first, int second) {
    return first + second;
}
```

🟩

Chapter 2 Exercise: Addition

2.1 Exercise

2.2 **Summary**

What to learn in this chapter(again)

- Basic steps of TDD
  - Write a test
  - Make the test compiled
  - Red
  - Green
  - Refactor

- How to use to-do lists

Exercise : Addition　—　Summary

Basic steps of TDD

- Write a to-do list by examining test cases
- Consider a necessary method for a to-do item.
- Write a test for it.
- Make it compile (Red)
- Make it pass (Green)
- Refactor

● In this exercise we implement by small steps
● It does not always need to divide to such small steps.
● If you are confident , it is no problem to jump to a lager implementation.
● But when make a mistake, start over by smaller steps.

⬇

Size of steps can be changed according to the situation.

Essence of Test Driven Development（TDD） （again）

Design technique

- Though writing tests, make how the production code behaviors clearer.
- It assumes evolutionary design.

Development approach to pile up reliability

- It is a technique to encourage development by piling up verifiable implementation little by little, which is possible only by the tests.
- Developers write the tests by themselves.
- Developers decides the range to test.
- To stay a fixed rhythm, it needs rapid build and test environment.

Chapter3 Overview of Refactoring

What to learn in this chapter

1. Two values of software
   - What can it do now?
   - What will it be able to do in the future?

2. What is refactoring?
   - Definition of refactoring
   - Motivation of refactoring
     - Code smell
   - Techniques for refactoring
     - Refactoring catalog
   - Notices

3. Refactoring by tools
   - Refactoring support in Eclipse

Chapter3 Overview of Refactoring

3.1 **Two values of software**

3.2 What is refactoring

3.3 Refactoring by tools

3.4 Summary

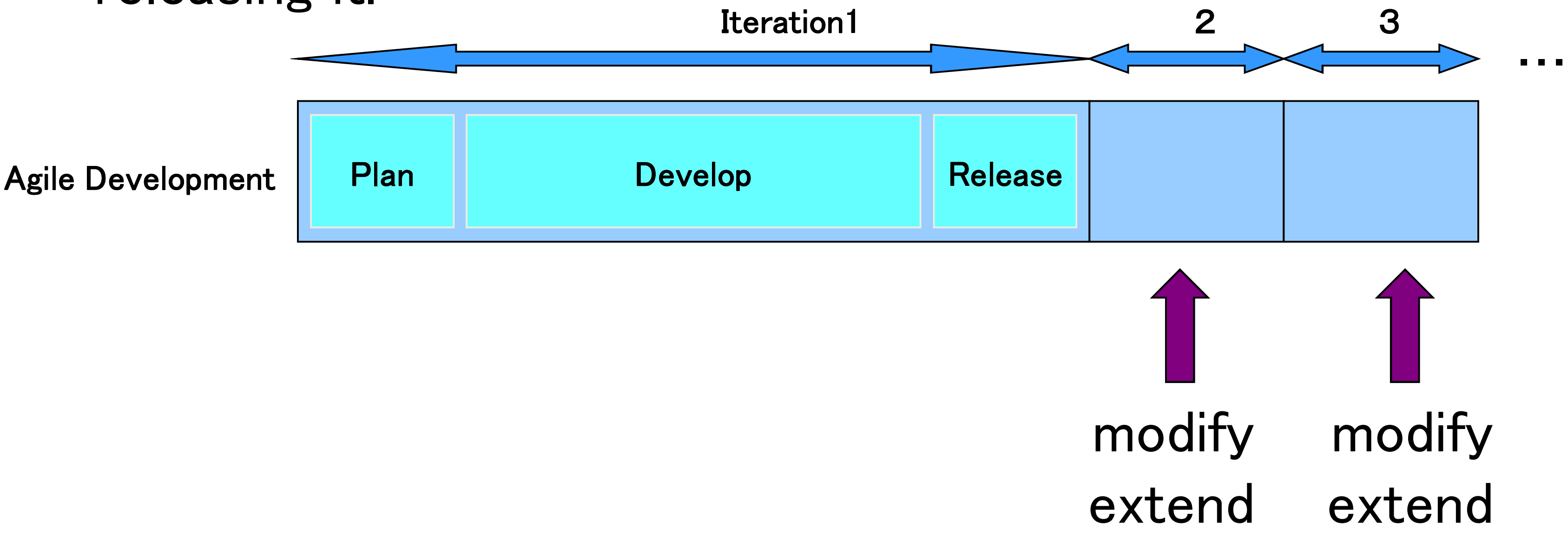
Background

- Problems which often happens in projects.
  - A lot of time which a modification needs.
  - There are parts anyone cannot understand because the author went to another job.
  - Bugs occur in other places where the modification has been maden.

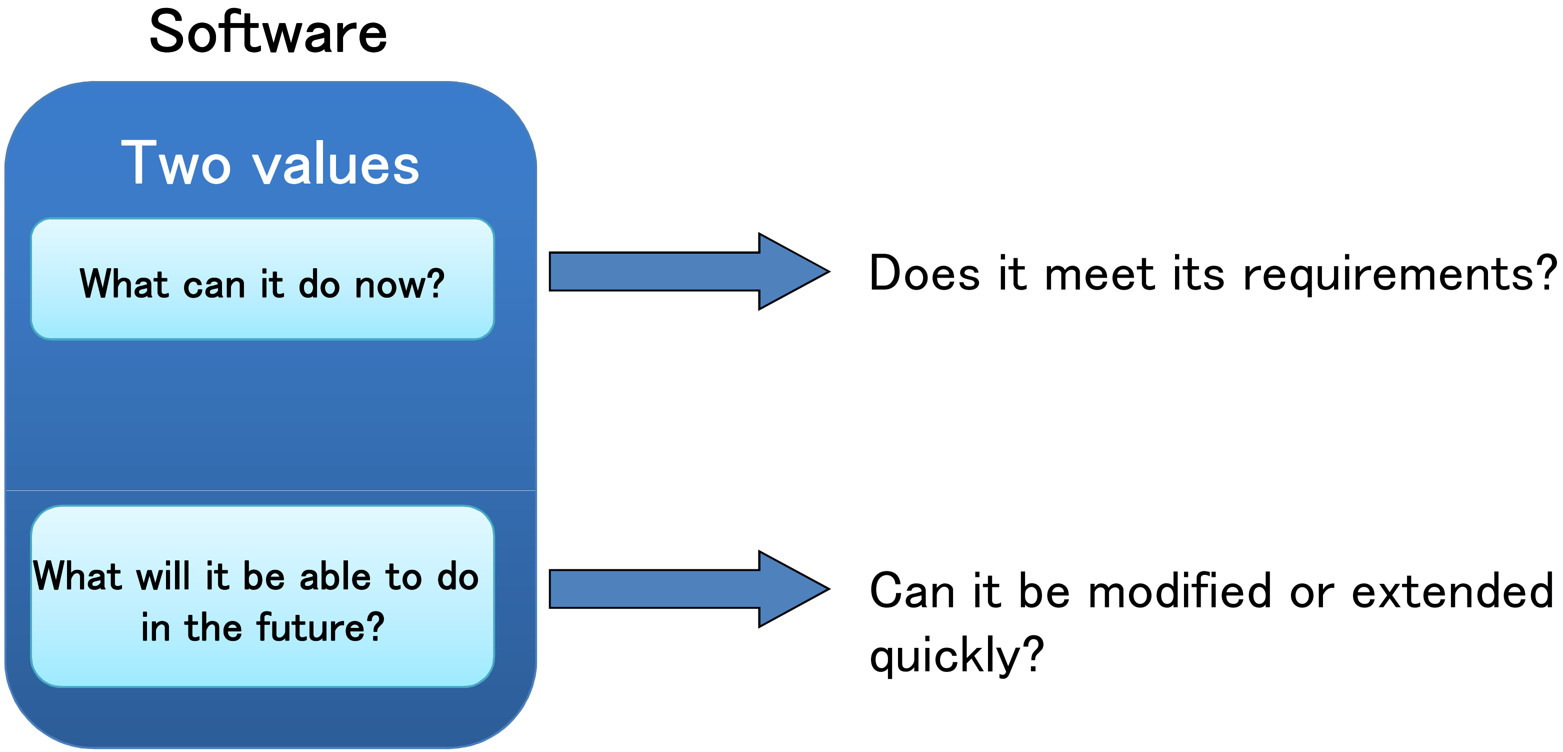
Want to solve those problems

Life cycle of software development

- Software is often modified and extended.
- There is rare cases that software keeps untouched after releasing it.



Two values of software


Software
Two values
What can it do now?
What will it be able to do in the future?
Does it meet its requirements?
Can it be modified or extended quickly?

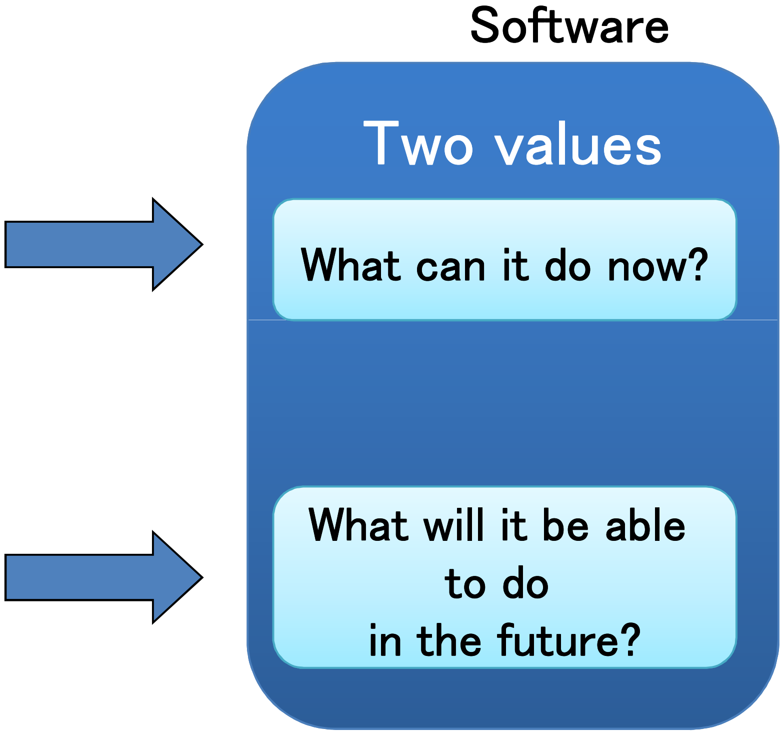
To produce more valuable software, how do TDD and Refactoring deal with it?

Two values of software

Clean code that works

Working code and tests makes "What it can do now" clear.

Keeping clean code ensures "What it will be able to do in the future".



Chapter3 Overview of Refactoring

3.1 Two values of software

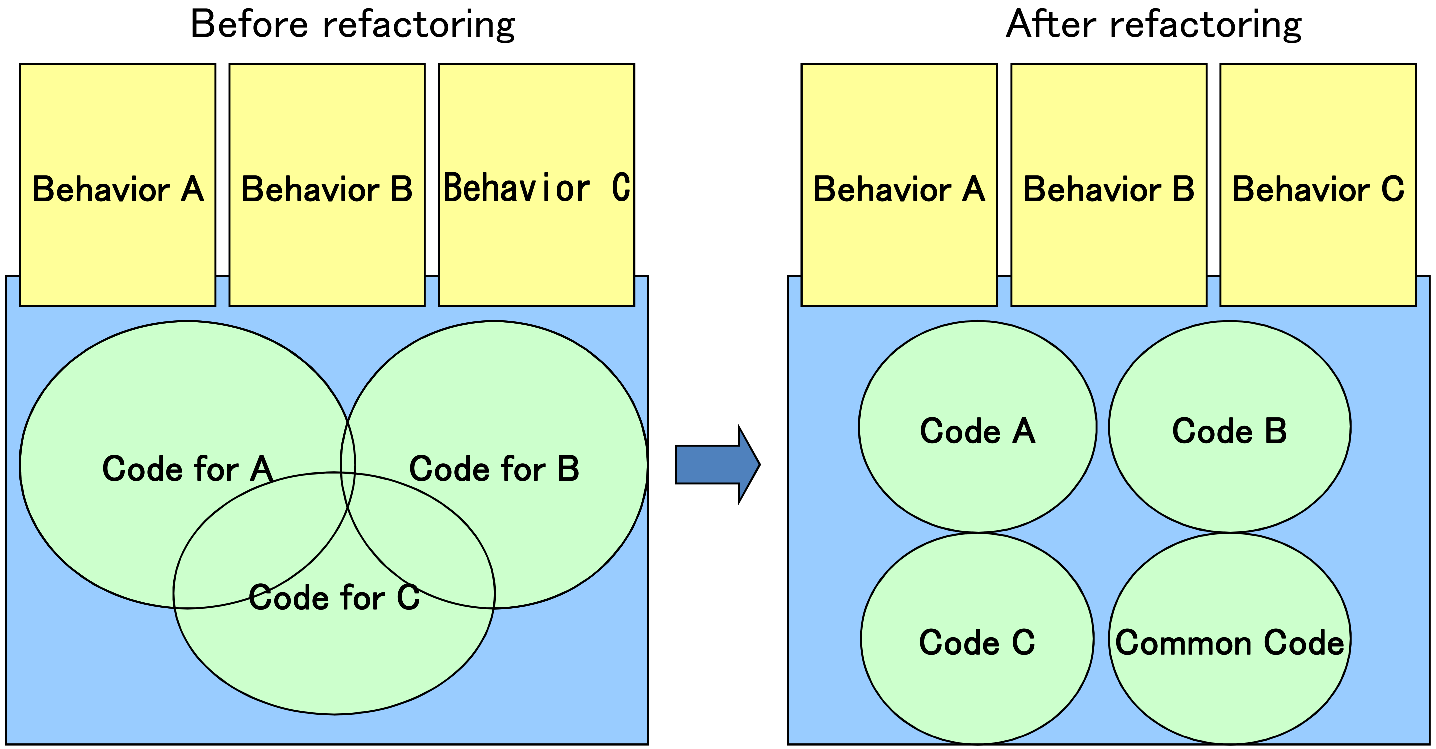
3.2 **What is refactoring**

3.3 Refactoring by tools

3.4 Summary

What is Refactoring

- restructuring an existing body of code, altering its internal structure without changing its external behavior (http://martinfowler.com/refactoring/)



Picture of refactoring

Elements of refactoring

- Where should we refactor?
  - Where code smell exists

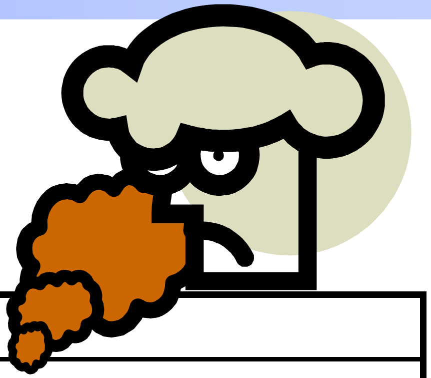
- How should we refactor?
- How far should we refactor?
  - Refactoring catalog
  - Design patterns

- Is it time consuming?
- Does changing code cause a bug?
  - Tests
  - Tool support
    - Refactoring tools
    - Other tools

- Notice
  - Do not add a feature

Code smell

- a hint that something might be wrong



Examples of Code smells

Smell	Symptoms
Duplicate Code	identical or very similar code exists in more than one location
Long method	a method, function, or procedure that has grown too large.
Large class	a class that has grown too large.
Long Parameter List	a long list of parameters in a procedure or function
Divergent Change	a single class that needs to be modified by many different types of changes
Shotgun Surgery	we need to modify many classes when making a single change
Feature Envy	one method is too interested in other classes
Data Clumps	there exists a set of primitives that always appear together
Primitive Obsession	there are no small classes for small entities
Switch Statements	The situation where switch statements or type codes are overused.
Parallel Inheritance Hierarchies	every time you make a subclass of one class, you also have to make a subclass of another

Code smell

- Lazy class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class
- Refused Bequest
- Comments
- Conditional Complexity
- Indecent Exposure
- Solution Sprawl
- Combinatorial Explosion
- Oddball Solution

From “Refactoring: Improving the Design of Existing Code” and “Refactoring to Patterns”

Code smell

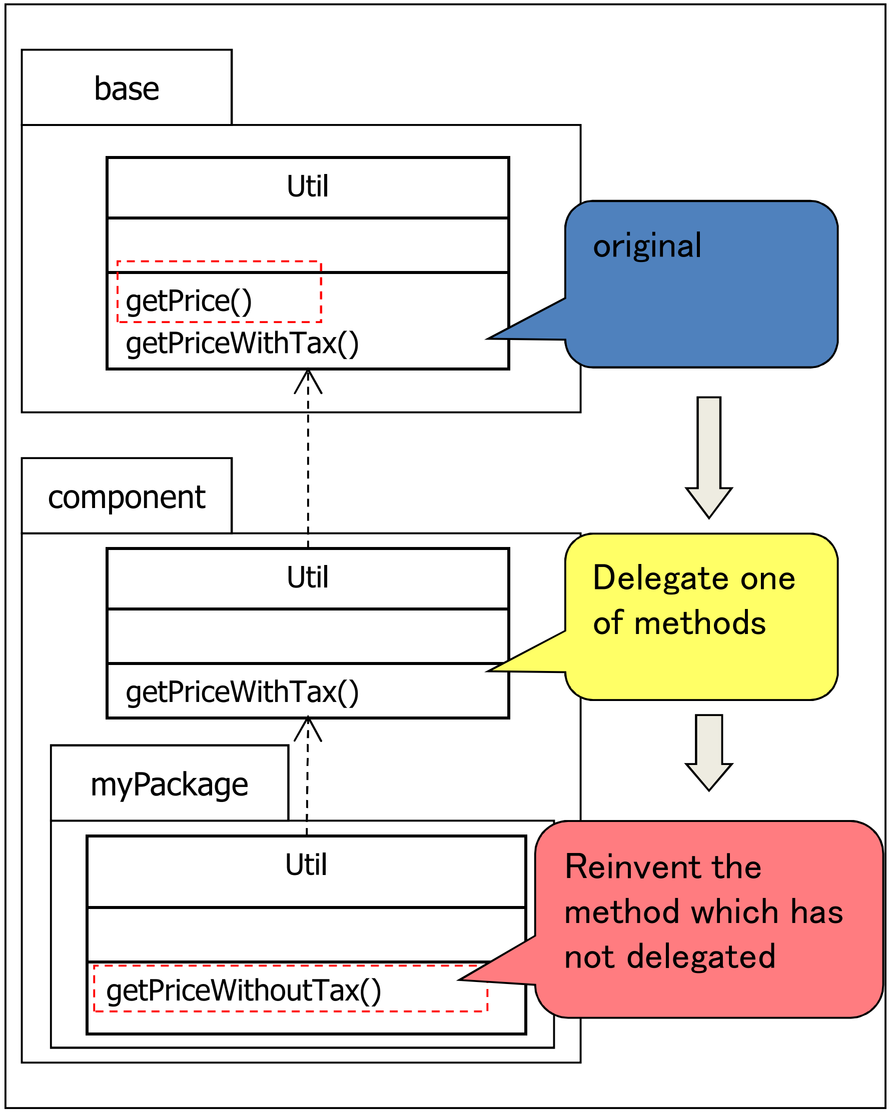
- Duplicate Code
  - Copy and paste
  - Re-implement

Example 1 copy and paste

```
double tax = 1.05;

public int getPrice(Item item, int amount){
    int price;
    if ("Apple".equals(item.getName()))
        price = 100 * amount;
    else
        price = 50 * amount;
    return price;
}
public double getPriceWithTax(Item item,
         int amount){
    double price;
    if ("Apple".equals(item.getName()))
        price = 100 * amount * tax;
    else
        price = 50 * amount * tax;
    return price;
}
```

Example 2 re-implement



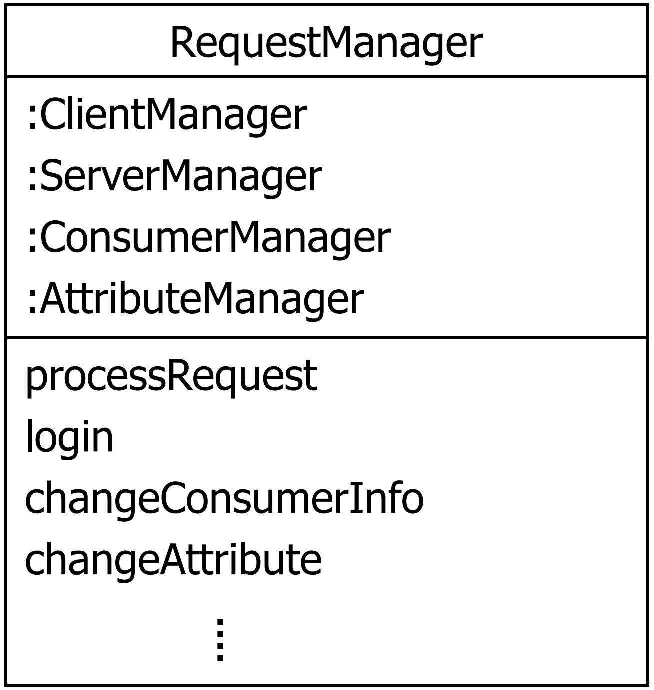
Code smell

- Long method
  - One method performs multiple operations.
  - There is a comment augmenting the complex procedure.

```
public int getPrice(Item item, int amount) {
    int price;
    // As for Apple, there is discount by its amount.
    if ("Apple".equals(item.getName())) {
        switch (amount) {
        case 1:
            price = 100 * amount;
            break;
        case 2:
            price = 95 * amount;
            break;
        case 3:
            price = 90 * amount;
            break;
        default:
            price = 85 * amount;
            break;
        }
    } else
        price = 50 * amount;
    return price;
}
```

Code smell

- Large class



RequestManager

:ClientManager
:ServerManager
:ConsumerManager
:AttributeManager

processRequest
login
changeConsumerInfo
changeAttribute
⋮

- Long Parameter List

```
public boolean isAvailable(
        String name,
        Date currentDate,
        Date limitDate,
        boolean activeFlag);

public void setAccount(String number,
        String name,
        Date startDate,
        int amount,
        int accountType);
```

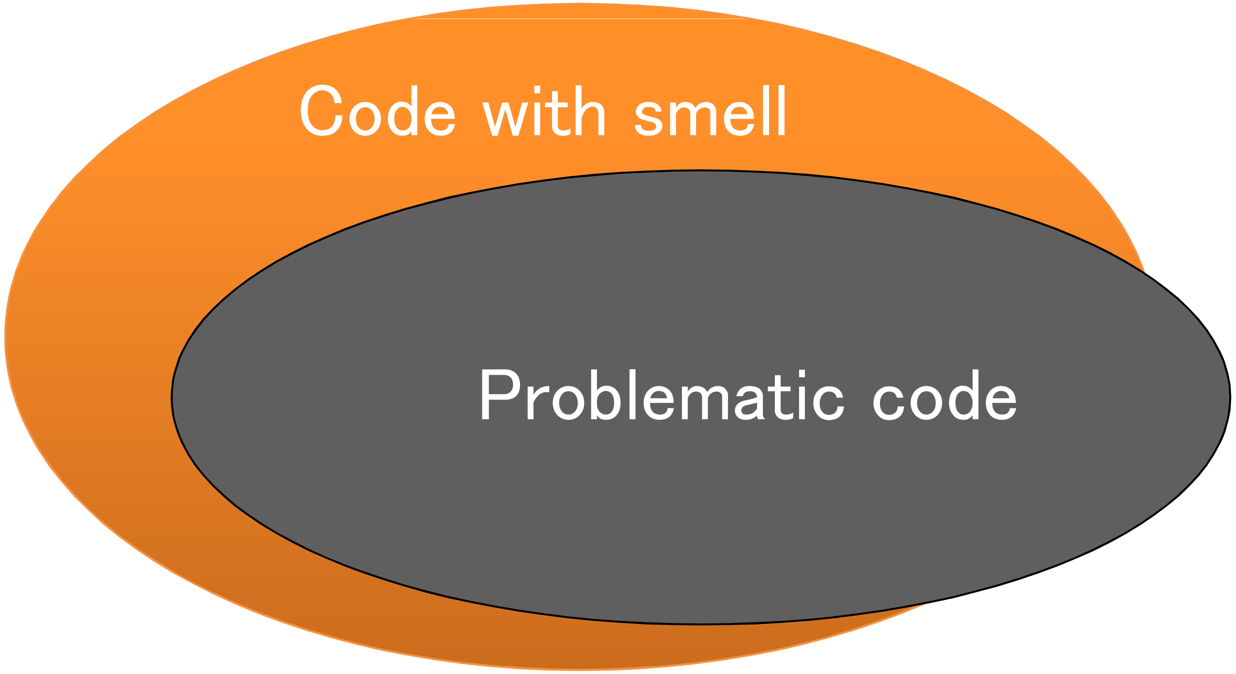
Code smell

- All code with smell are not problematic.
  - Easy to find a problem
  - Where there is a problem,

  it often smells.

→ Smell

  - It is also effective to list up original smells by your team.



Refactoring Catalog

- Various refactoring approaches are provided.
  - Method composition
  - Move features of objects
  - Reorganize data
  - Simplify conditionals
  - Simplify method invocation
  - How to inherit
  - Big refactoring

Smell and refactoring catalog

Smell	Refactoring
Duplicate Code	・Extract Method ・Extract Class ・Pull Up・Form Template Method
Long method	Extract Method ・Inline・Introduce Parameter Object・Decompose Conditional
Large class	Extract Class ・Extract Subclass ・Extract Interface ・Replace Data Value with Object
Long Parameter List	Replace Parameter with Method ・Introduce Parameter Object
Divergent Change	Extract Class
Shotgun Surgery	Move Method・Move Field ・Inline Class
Feature Envy	Move Method・Move Field・Extract Method
Data Clumps	Extract Class ・Introduce Parameter Object・Introduce Parameter Object

From "Refactoring: Improving the Design of Existing Code"

Refactoring by hands

```
void printOwing (double amount) {
    printBanner();

    // print details
    System.out.println("name:" + _name);
    System.out.println("amount:" + amount);
}
```

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails(double amount) {
    System.out.println("name:" + _name);
    System.out.println("amount:" + amount);
}
```

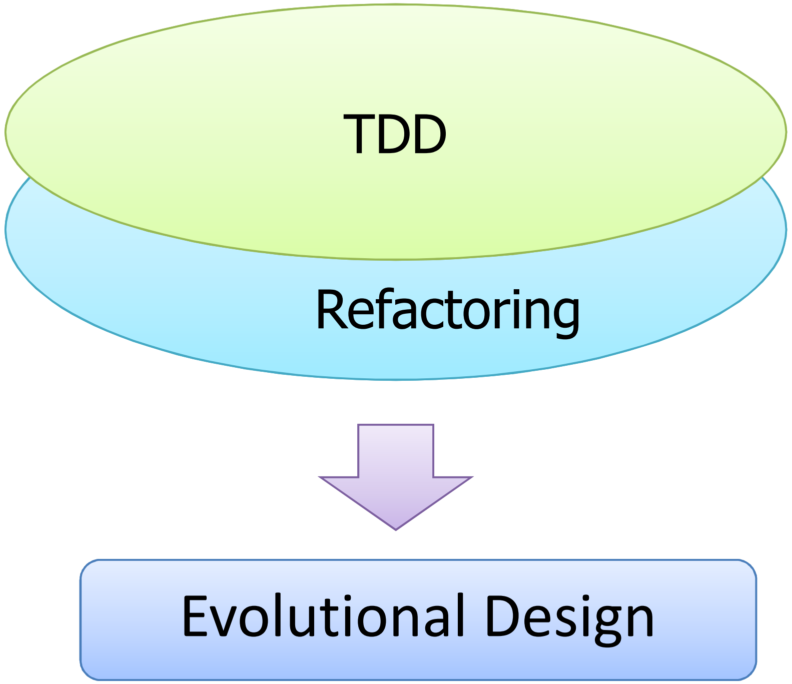
･same granularities
･understandable

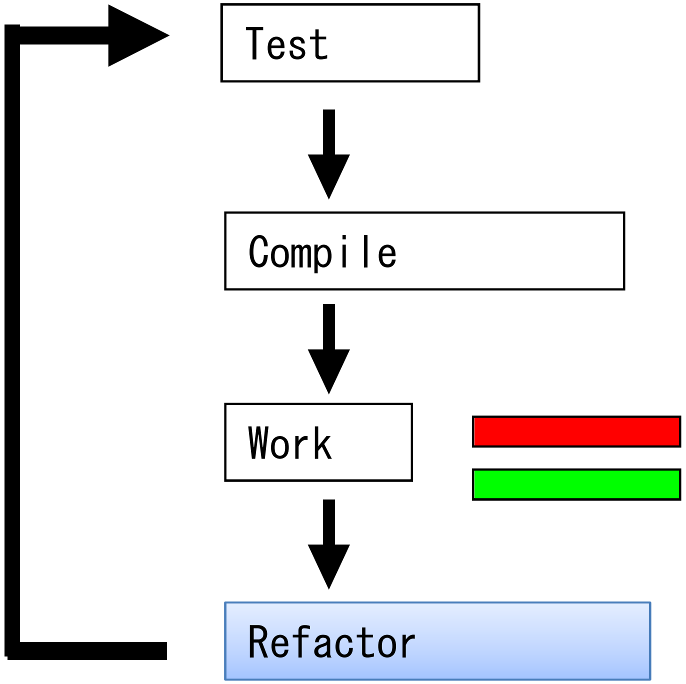
- "Extract Method"
- Steps
  1. Make a new method, and name it by the intent
  2. Copy the extracted code from the original method to the new method.
  3. Find local scope variables(local variables and arguments) in the extracted code.
  4. Check if the temporary variables are used only in the extracted code. In that case, declare them as temporary variables of the new method.
  5. Check if the extracted code make changes the local scope variables.
  6. If there is only one variable to be changed, return the value from the new method and assign the value to the suitable variable of the original method. If there are more variables, try other refactoring.
  7. Pass the local scope variables which are read from the new method as its arguments.
  8. Compile
  9. In the original method, replace the extracted code with a call to the new method.
  10. Compile and test

From "Refactoring: Improving the Design of Existing Code"

Refactoring as a part of TDD

- Refactoring is embedded in a cycle of TDD.





Guideline for Refactoring

- What is clean code = good design?

  - Two values of software are both high.
    - Meets requirements (current value)
    - Easy to modify and extend (future value)

  - More concretely
    - Object structure = design with high cohesion and low coupling

  - Guideline
    - Object oriented design principles
    - Design patterns

It is important to share the acknowledgement in the team

Notice : Do not add a feature

- When refactoring, do not add a feature
  - Changing structures and functions simultaneously loses the reliability of tests and makes it difficult to find problems.

It is important not to change behaviors.

Notice：Overdesign, excessive flexibility, and so on

- Flexible as needed
  - Excessive flexibility causes higher maintenance cost and more bugs.
  - YAGNI (You Aren't Going to Need It.)

- Performance tuning
  - Refactoring is not for performance
  - However refactoring localizes the parts which need to be tuned

- Sometimes modification of tests is needed.
  - Such modification should be as little as possible.

Chapter3 Overview of Refactoring

3.1 Two values of software

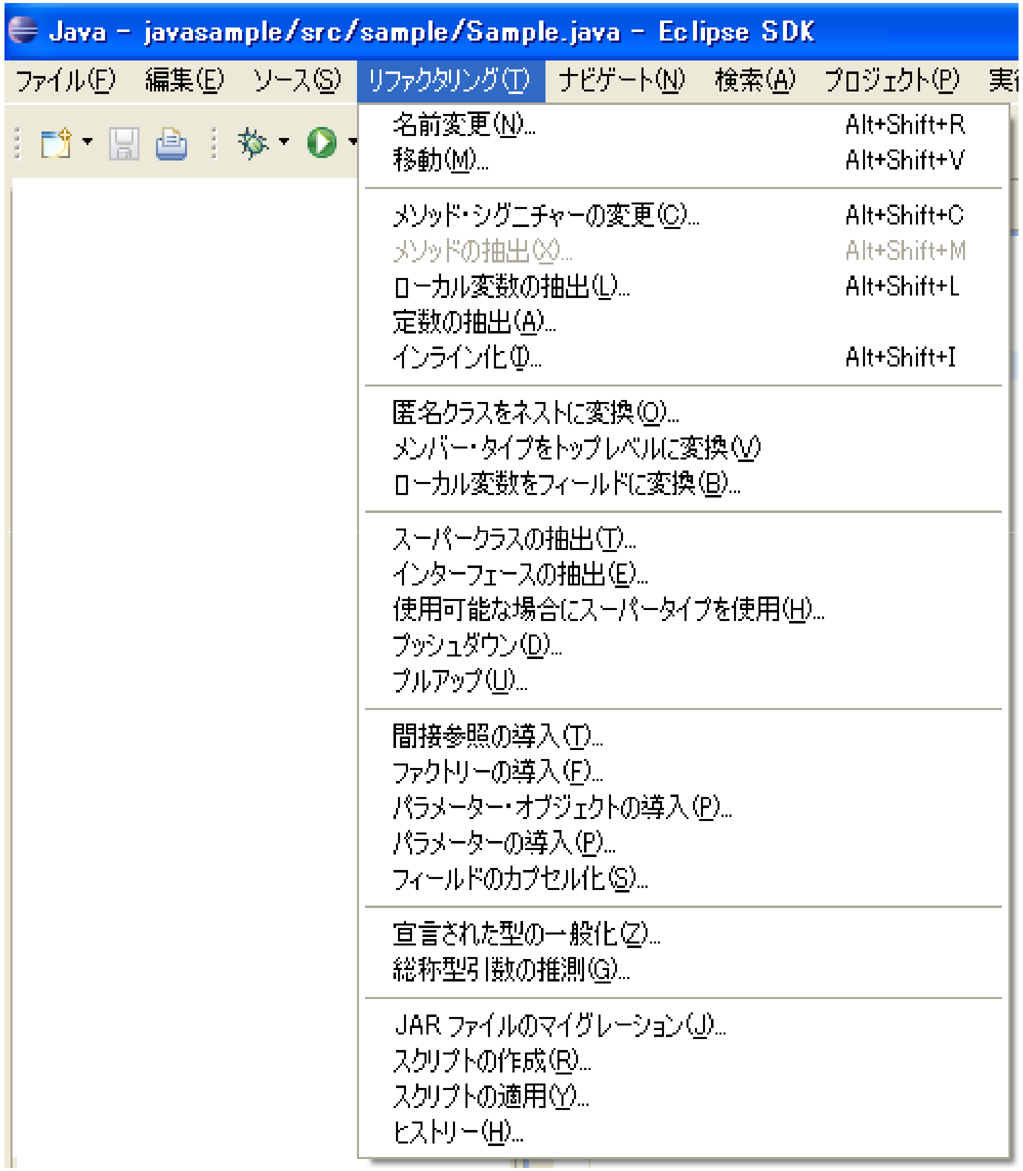
3.2 What is refactoring

3.3 **Refactoring by tools**

3.4 Summary

Tool Support

- Eclipse
  - One of its top menus is for refactoring
    - Eclipse takes refactoring very important


Java - javasample/src/sample/Sample.java - Eclipse SDK
ファイル(F) 編集(E) ソース(S) リファクタリング(T) ナビゲート(N) 検索(A) プロジェクト(P) 実
名前変更(N)... Alt+Shift+R
移動(M)... Alt+Shift+V
メソッド・シグニチャーの変更(C)... Alt+Shift+C
メソッドの抽出(X)... Alt+Shift+M
ローカル変数の抽出(L)... Alt+Shift+L
定数の抽出(A)...
インライン化(I)... Alt+Shift+I
匿名クラスをネストに変換(O)...
メンバー・タイプをトップレベルに変換(V)
ローカル変数をフィールドに変換(B)...
スーパークラスの抽出(T)...
インターフェースの抽出(E)...
使用可能な場合にスーパータイプを使用(H)...
プッシュダウン(D)...
プルアップ(U)...
間接参照の導入(T)...
ファクトリーの導入(F)...
パラメーター・オブジェクトの導入(P)...
パラメーターの導入(P)...
フィールドのカプセル化(S)...
宣言された型の一般化(Z)...
総称型引数の推測(G)...
JAR ファイルのマイグレーション(J)...
スクリプトの作成(R)...
スクリプトの適用(Y)...
ヒストリー(H)...

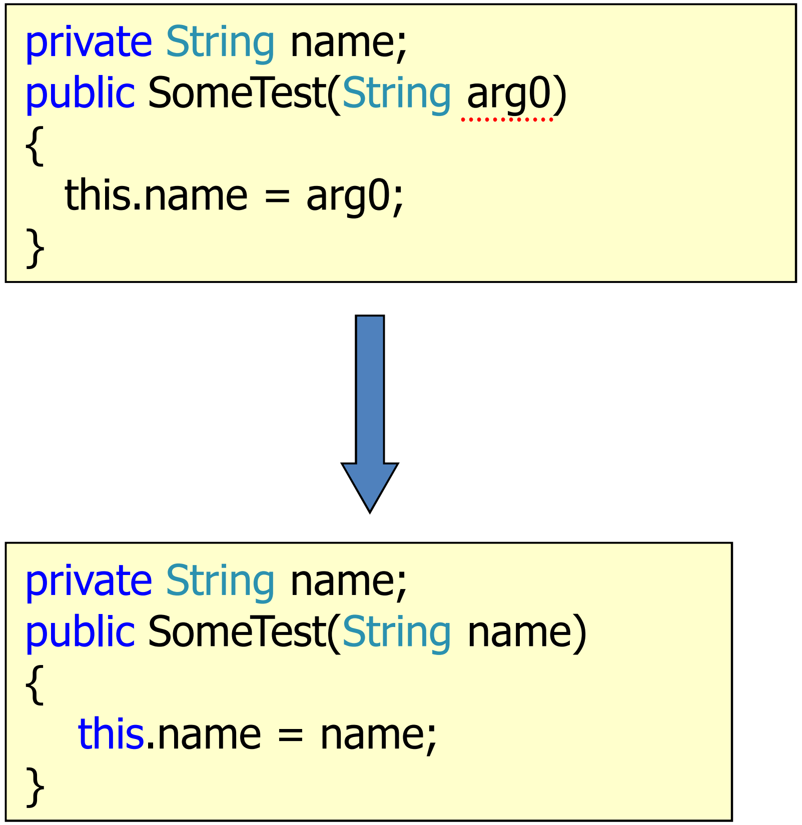
Rename

Motivation

- Names of classes, methods, or variables are ambiguous or meaningless

Operation

- Rename it and make the code self descriptive



```
private String name;
public SomeTest(String arg0)
{
  this.name = arg0;
}
```

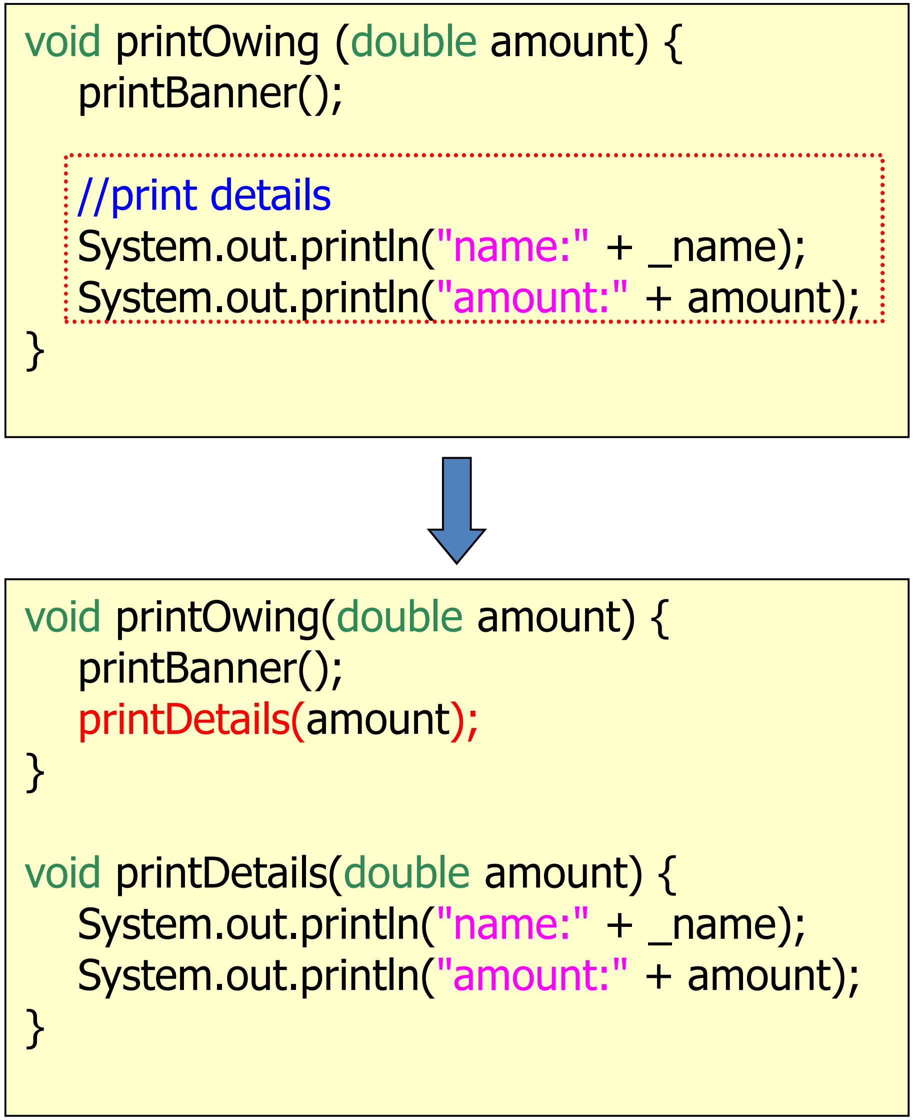
```
private String name;
public SomeTest(String name)
{
   this.name = name;
}
```

Extract Method

- **Motivation**
  - There is a code snippet to be grouped as a unit.
    - Long method
    - Cannot understand its intent without a comment.

- **Important：Method name should represent its body.**

- **Operation**
  - Make a code snippet a method and name it descriptively.



```
void printOwing (double amount) {
    printBanner();

    //print details
    System.out.println("name:" + _name);
    System.out.println("amount:" + amount);
}
```

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails(double amount) {
    System.out.println("name:" + _name);
    System.out.println("amount:" + amount);
}
```

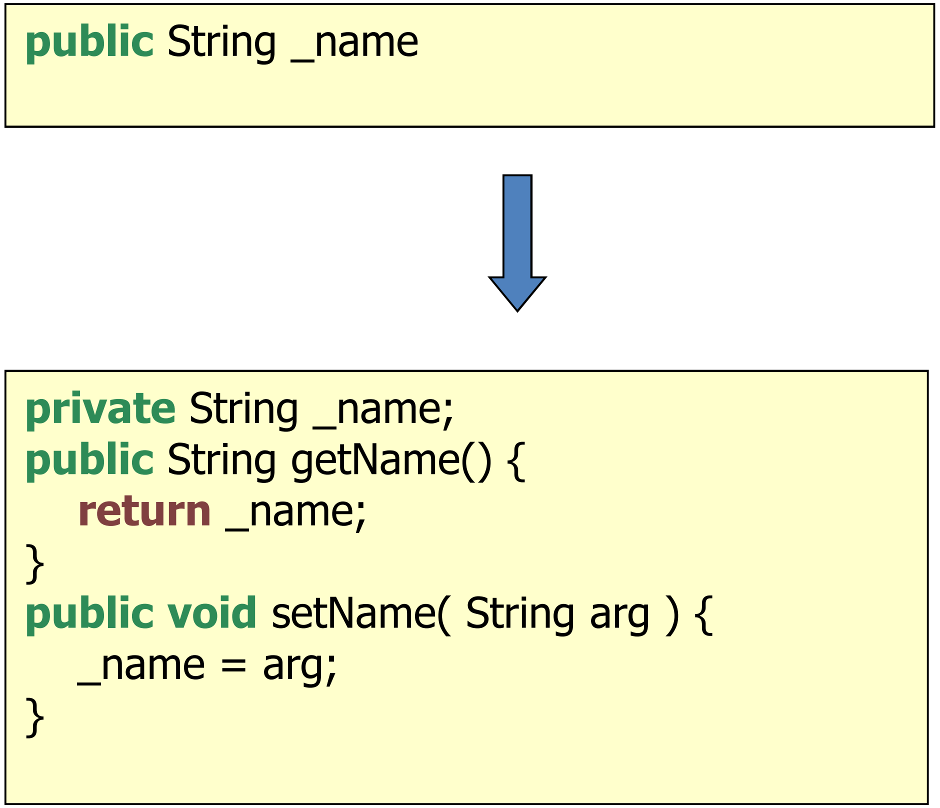
Encapsulate Field

Motivation

- There is a public field.

Operation

- Replace the field with a property.

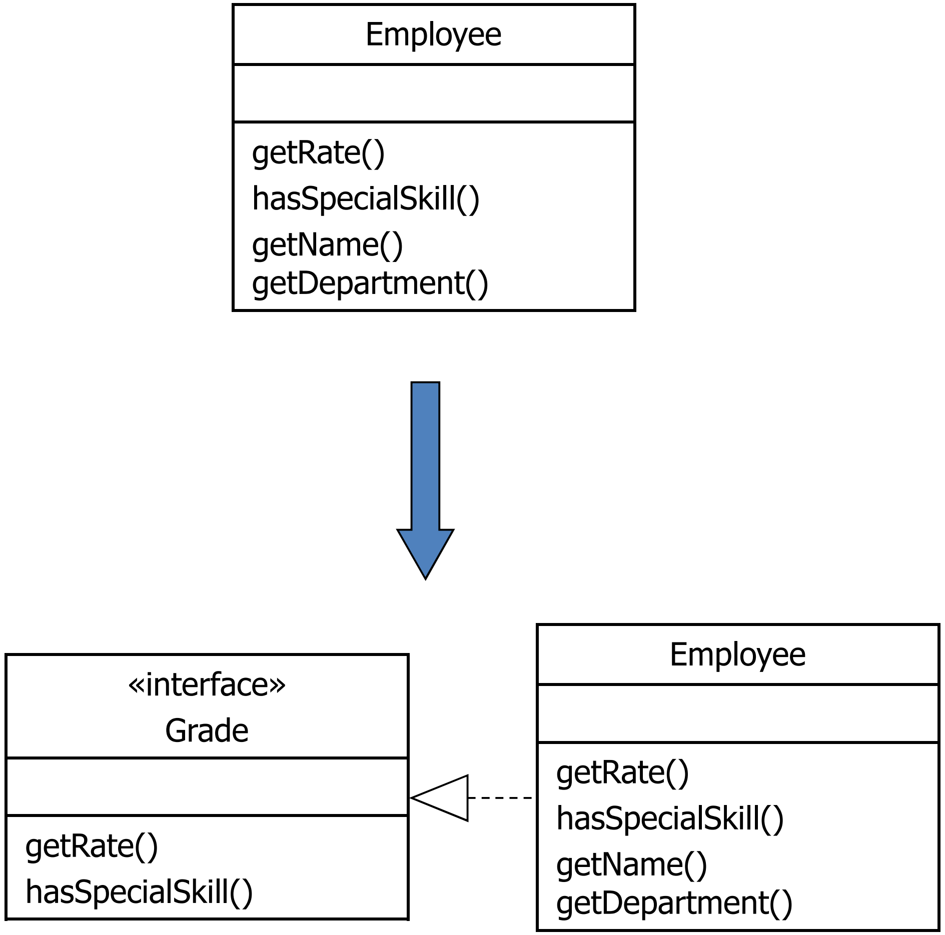


```
public String _name
```

```
private String _name;
public String getName() {
    return _name;
}
public void setName( String arg ) {
    _name = arg;
}
```

Extract Interface

- Motivation
  - Multiple clients use a clump of interfaces in one class.
- Operation
  - Extract the clump as an interface.



Promote Local Variable to Parameter

Motivation

- A method requires more information from the caller.

Operation

- Add a parameter to pass the necessary information.

```
private void aMethod()
{
    String aString = "";
    aString = ...
}
```

```
private void aMethod(String aString)
{
    aString = ...
}
```

Other tools：Automated tests

- What are automated tests?
  - Intent
    - Ensure modification of code has not changed behaviors
      - Make it possible to check effectiveness before deployment
  - Tool
    - JUnit etc

Other tools：Version Control System

- What is version control system?
  - Intent
    - Manage source code and other data.
    - Make it possible to go back to the previous good state.
  - Tool
    - CVS (Concurrent Versions System)
    - Subversion
    - Mercurial
    - etc
- When it takes longer than expected, you sometimes need to give up the refactoring and revert changes.

Chapter3 Overview of Refactoring

3.1 Two values of software

3.2 What is refactoring

3.3 Refactoring by tools

3.4 **Summary**

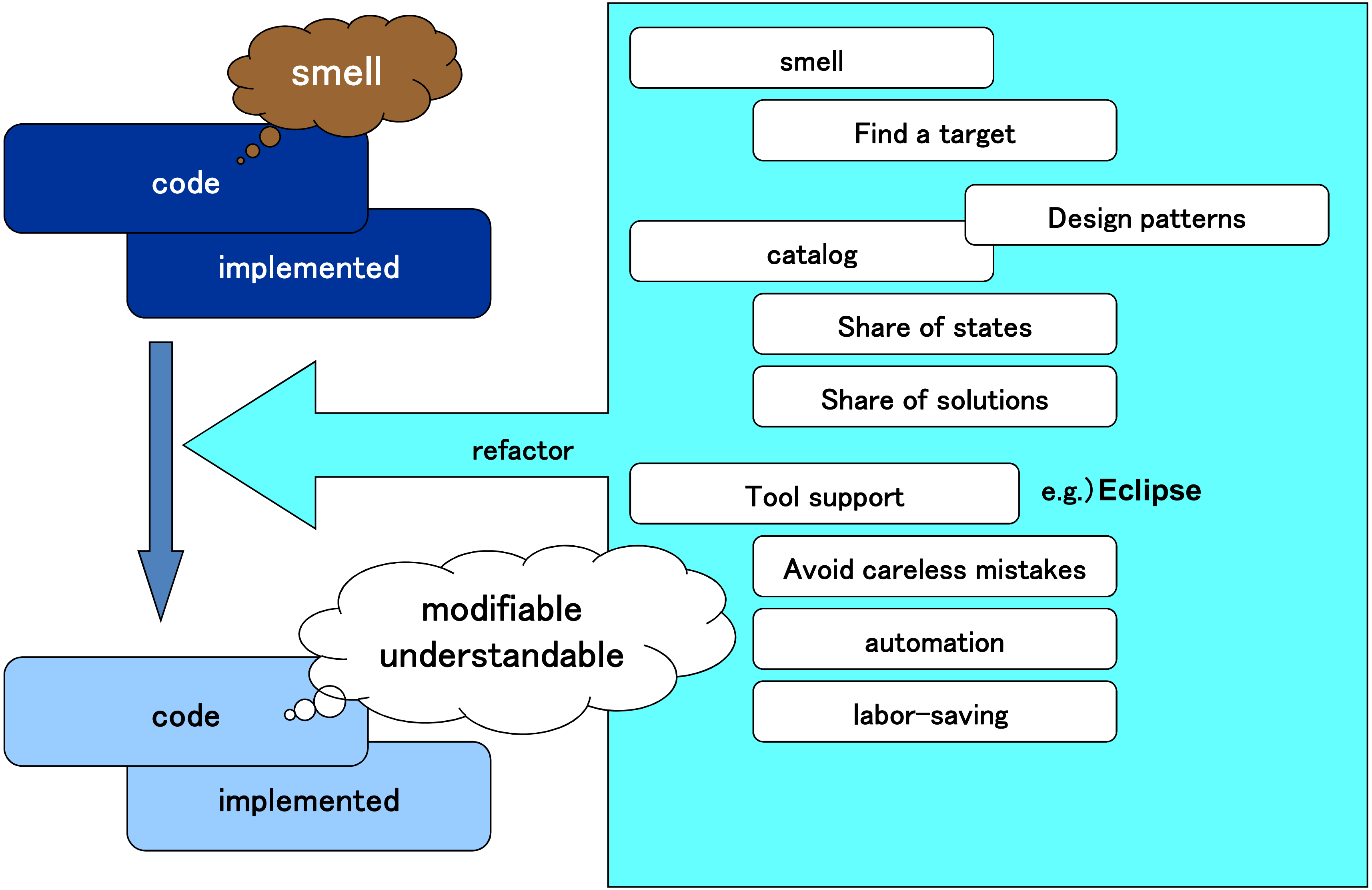
What to learn in this chapter (again)

1. Two values of software
   - What can it do now?
   - What will it be able to do in the future?

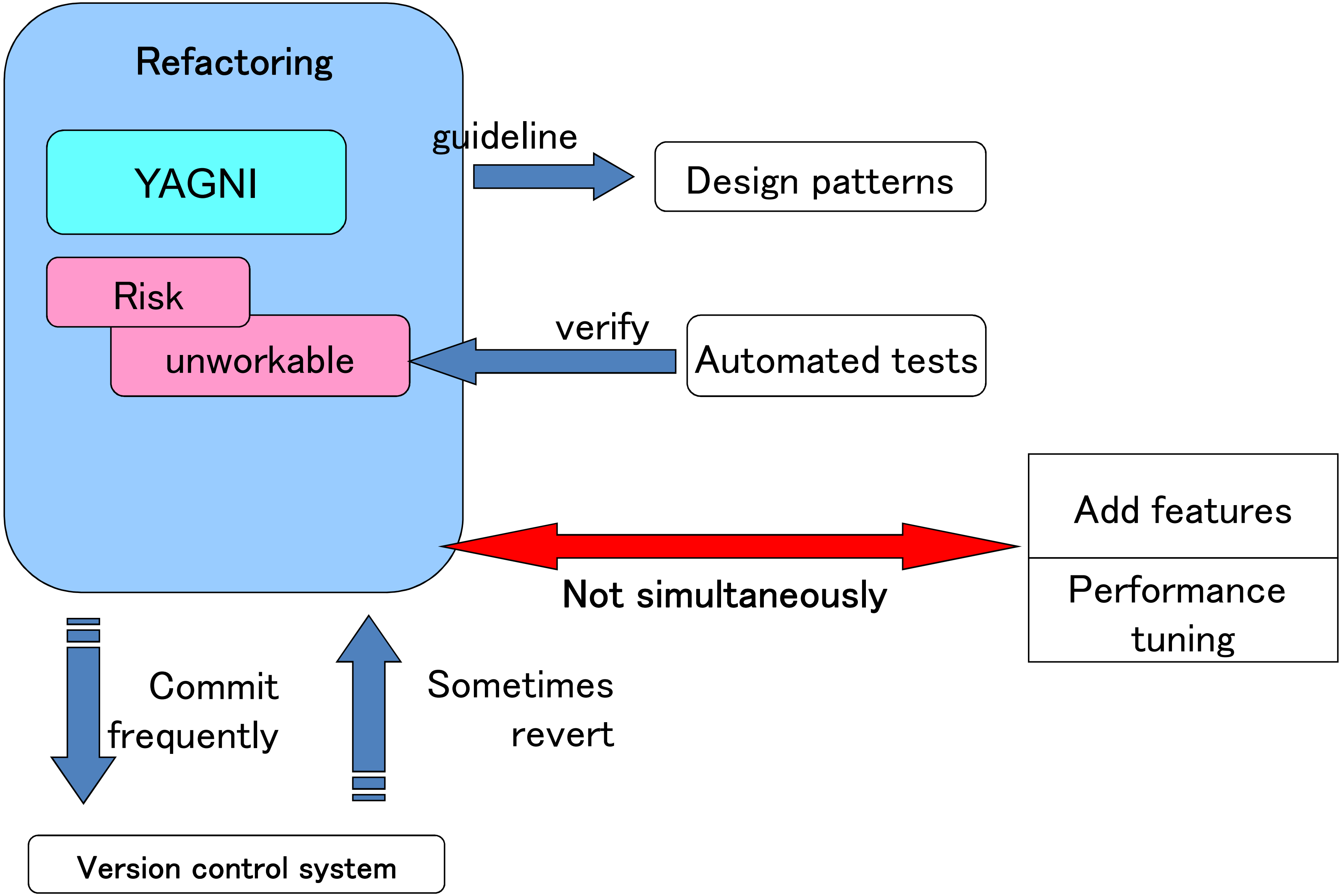
2. What is refactoring?
   - Definition of refactoring
   - Motivation of refactoring
     - Code smell
   - Techniques for refactoring
     - Refactoring catalog
   - Notices

3. Refactoring by tools
   - Refactoring support in Eclipse

Significance of refactoring



Position of refactoring



Refactoring

YAGNI

Risk

unworkable

guideline

Design patterns

verify

Automated tests

Not simultaneously

Add features

Performance tuning

Commit frequently

Sometimes revert

Version control system

Evolutional design and refactoring

Evolutional design

- Changes of specification and design cause a lot of modifications of code
  - It needs to lower costs and risks
    - TDD
    - Refactoring
  - Learning and applying those techniques, achieve evolutional design (agile development) with controlling costs and risks.

Chapter 4 Exercise: Calculation of prices for party

What to learn in this chapter

- Basics
  - Basic steps of TDD.
  - Three strategies achieving green
  - Tests for Exceptions
- Advanced 1
  - Evolutional design
  - Fake it in action
  - Express details by tests
- Advanced 2
  - Responsibility change with metaphor change

Chapter 4 Exercise: Calculation of party price

4.1 **Basics**

4.2 Advanced 1

4.3 Advanced 2

4.4 Summary

Three strategies for internal implementation

- Fake it
  - When complex implementation is to be needed, make it green by fake values and eliminate duplications later.
  - Eliminate duplications between test and production code by refactoring.
  - Making it green has the highest priority.
- Obvious implementation
  - When there seems to be an obvious solution, try it
  - When the solution turns out to be bad, go back and start over with fake it.
- Triangulation
  - It does not make sense when you have an implementation in mind.(by Kent Beck)
  - When exploring a design, it can help.

Exercise: Calculation of party price ー　Basic scenario

- You make the component to calculate price for the party when entered number of participant and the course.
- There three courses Matsu(7,000yen), Take(5,000yen), and Ume(3,000yen).
- Course price times number of participants becomes the price of the party.
- There is a coupon to discount 10,000yen from the price of party.
- Three coupons can be used at the same time.

Exercise: Calculation of party price　—　To-Do List

- Make Dinner Reservation class
  - It have attributes for number and course.
- Price can be calculated with number of participants and course.
- When coupons are used.

Calculate price
When coupons are used
3 coupons limit

Getting ready: prepare Eclipse

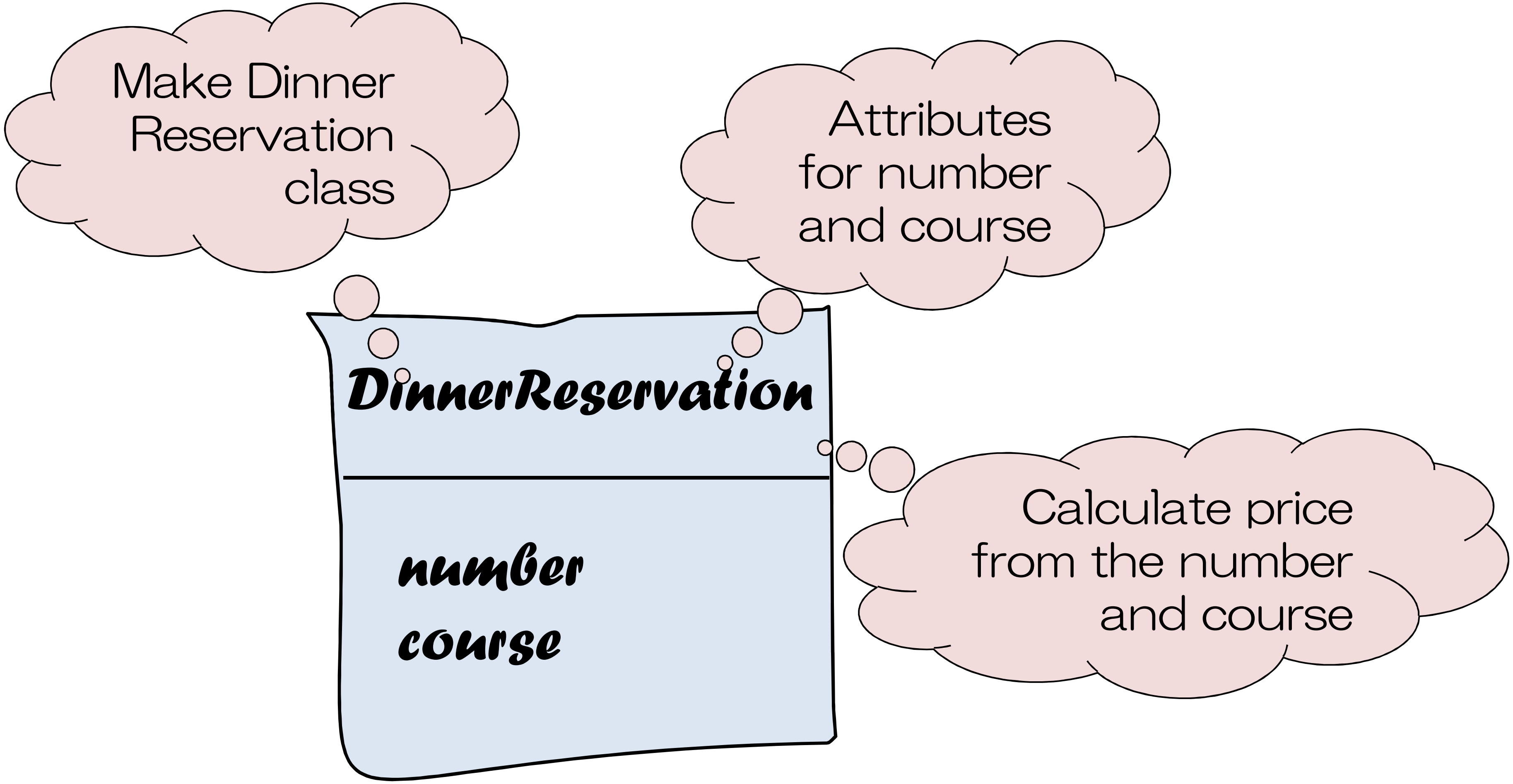
- Create a new Java project

Project name: Dinner

- Configure Build Path for JUnit

Libraries tab in Java Build Path => Add Library => select JUnit => select JUnit 4

Exercise: Calculation of party price（Basics） ー Design


Make Dinner Reservation class
Attributes for number and course
DinnerReservation
number
course
Calculate price from the number and course

Exercise: Calculation of party price（Basics） — the first test

- Since it needs much work to deal with all courses, divide cases by course.
- First write a test to express the specification.

Calculate price（Matsu）
When coupons are used
3 coupons limit
Calculate price（Take）
Calculate price （Ume）

```
import static org.junit.Assert.*;
import org.junit.Test;

public class DinnerReservationTest
{
    @Test
    public void testMatsu()
    {
        DinnerReservation resrv = new DinnerReservation();
        resrv.setUser(10);
        resrv.setCourse(1);
        assertEquals(70000, resrv.getCharge());
    }
}
```

🟥

Exercise: Calculation of party price （Basics） — the first code

- Make it compile

```
public class DinnerReservation
{
    public void setUser(int number)
    {
    }

    public void setCourse(int course)
    {
    }

    public int getCharge()
    {
        return 0;
    }
}
```

Calculate price（Matsu）
When coupons are used
3 coupons limit
Calculate price（Take）
Calculate price （Ume）

Exercise: Calculation of party price（Basics）　—　Fake it

- Minimum implementation for the test to pass

```
public class DinnerReservation
{
    public void setUser(int number)
    {
    }

    public void setCourse(int course)
    {
    }

    public int getCharge()
    {
        return 70000;
    }
}
```

<u>Calculate price（Matsu）</u>
When coupons are used
3 coupons limit
Calculate price（Take）
Calculate price （Ume）

🟩

Exercise: Calculation of party price（Basics） ー eliminate duplication

- 70000 is duplicated between the test and the code
- Refactor to remove this duplication

```
public class DinnerReservation
{
    private int _number;

    public void setUser(int number) {
        _number = number;
    }

    public void setCourse(int course) {
    }

    public int getCharge() {
        return 7000 * _number;
    }
}
```

<u>Calculate price（Matsu）</u>
When coupons are used
3 coupons limit
Calculate price（Take）
Calculate price （Ume）

Exercise: Calculation of party price（Basics） — the second test

- Since the first case is done, proceed to the next one.

```
@Test
public void testTake()
{
    DinnerReservation resrv = new DinnerReservation();
    resrv.setUser(10);
    resrv.setCourse(2);
    assertEquals(50000, resrv.getCharge());
}
```

~~Calculate price(Matsu~~)
When coupons are used
3 coupons limit
Calculate price（Take）
Calculate price（Ume）

🟥

Exercise: Calculation of party price（Basics） －Obvious Implementation

- Since it is clear how to implement, just do it.

```
class DinnerReservation
{
    private int _number;
    private int _course;
    public void setUser(int number) {
        _number = number;
    }
    public void setCourse(int course) {
        _course = course;
    }
    public int getCharge() {
        int charge = 0;
        switch (_course) {
            case 1:
                charge = 7000 * _number;
                break;
            case 2:
                charge = 5000 * _number;
                break;
        }
        return charge;
    }
}
```

~~Calculate price(Matsu)~~
When coupons are used
3 coupons limit
Calculate price（Take）
Calculate price （Ume）

Exercise: Calculation of party price（Basics） — the third test

- Write the third test.

~~Calculate price（Matsu）~~
When coupons are used
3 coupons limit
~~Calculate price（Take）~~
Calculate price （Ume）

```
@Test
public void testUme()
{
    DinnerReservation resrv = new DinnerReservation();
    resrv.setUser(10);
    resrv.setCourse(3);
    assertEquals(30000, resrv.getCharge());
 }
```

🟥

Exercise: Calculation of party price（Basics） — Implement

- Implementation

```
class DinnerReservation
{
    (omitted)

    public int getCharge() {
        int charge = 0;
        switch (_course) {
            case 1:
                charge = 7000 * _number;
                break;
            case 2:
                charge = 5000 * _number;
                break;
            case 3:
                charge = 3000 * _number;
                break;
            default:
                charge = 0;
                break;
        }
        return charge;
    }
}
```

~~Calculate price（Matsu~~）
When coupons are used
3 coupons limit
~~Calculate price（Take）~~
Calculate price（Ume）

Exercise: Calculation of party price（Basics)　—　review of the code

- Make it possible to calculate the price
- Code smell
  - The numbers representing courses do not make sense, so make them constants.
  - The body of getCharge() is redundant, so extract the unit price calculation to another method.

~~Calculate price（Matsu)~~
When coupons are used
3 coupons limit
~~Calculate price（Take)~~
~~Calculate price（Ume~~)
Course constants
Method for unit price

Necessity of TDD

- Whenever code is modified, it ensures that behaviors of the code has not changed.
  - You can try a little modification to some little part. If successful, you can apply it widely.

- It is good only necessary parts for tests can be compiled and run.

Refactoring is always with tests (TDD)

Exercise: Calculation of party price（Basics）　―　test modifications

- Modify current tests

```
@Test
public void testMatsu() {
    DinnerReservation resrv = new DinnerReservation();
    resrv.setUser(10);
    resrv.setCourse(DinnerReservation.Course.Matsu);
    assertEquals(70000, resrv.getCharge());
 }
@Test
public void testTake() {
    DinnerReservation resrv = new DinnerReservation();
    resrv.setUser(10);
    resrv.setCourse(DinnerReservation.Course.Take);
    assertEquals(50000, resrv.getCharge());
}
@Test
public void testUme() {
    DinnerReservation resrv = new DinnerReservation();
    resrv.setUser(10);
    resrv.setCourse(DinnerReservation.Course.Ume);
    assertEquals(30000, resrv.getCharge());
}
```

~~Calculate price（Matsu）~~
When coupons are used
3 coupons limit
~~Calculate price（Take）~~
~~Calculate price（Ume）~~
Course constants
Method for unit price

🟥

Exercise: Calculation of party price（Basics） — modify the code

- Constants representing courses

```
class DinnerReservation
{
    public enum Course {Matsu, Take, Ume}
    private Course _course;

    public void setCourse(Course course) {
            _course = course;
    }
    public int getCharge() {
        int charge = 0;
        switch (_course) {
            case Matsu:
                charge = 7000 * _number;
                break;
            case Take:
                charge = 5000 * _number;
                break;
            case Ume:

    (...)
```

~~Calculate price（Matsu）~~
When coupons are used
3 coupons limit
~~Calculate price（Take）~~
~~Calculate price（Ume）~~
Course constants
Method for unit price

Exercise: Calculation of party price（Basics） ― modify the code

- Extract method for unit price calculation

```
class DinnerReservation {
    (omitted)

    public int getCharge() {
      int price = 0;
      switch (_course) {
        case Matsu:
          price = 7000;
          break;
        case Take:
          price = 5000;
          break;
        case Ume:
          price = 3000;
          break;
        default:
          price = 0;
          break;
      }
      return price * _number;
    }
}
```

~~Calculate price（Matsu）~~
When coupons are used
3 coupons limit
~~Calculate price（Take）~~
~~Calculate price（Ume）~~
~~Course constants~~
Method for unit price

Refactoring catalog:
Use "Rename"
Tool
  -> Refactor
    -> Rename

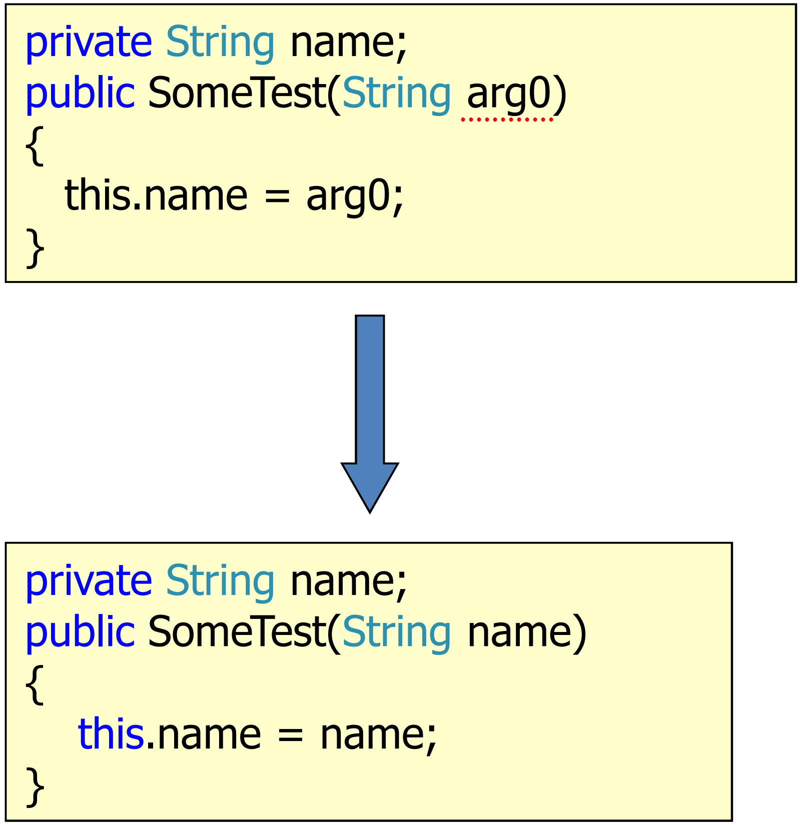
Rename(again)

Motivation

- Names of classes, methods, or variables are ambiguous or meaningless

Operation

- Rename it and make the code self descriptive



```
private String name;
public SomeTest(String arg0)
{
  this.name = arg0;
}
```

```
private String name;
public SomeTest(String name)
{
   this.name = name;
}
```

Exercise: Calculation of party price（Basics） ― modify the code

- Extract method for unit price calculation

```
class DinnerReservation {
   （omitted）

   public int getCharge() {
      int price = getPrice(_course);
      return price * _number;
   }
   private int getPrice(Course course) {
      switch (course) {
         case Matsu:
            return 7000;
         case Take:
            return 5000;
         case Ume:
            return 3000;
         default:
            return 0;
      }
   }
}
```

~~Calculate price（Matsu）~~
When coupons are used
3 coupons limit
~~Calculate price（Take）~~
~~Calculate price（Ume）~~
~~Course constants~~
Method for unit price

Refactoring catalog:
Use “Extract Method”
Tool
-> Refactor
-> Extract Method

Extract Method (again)

- Motivation
  - There is a code snippet to be grouped as a unit.
    - ➢ Long method
    - ➢ Cannot understand its intent without a comment.

```
void printOwing (double amount) {
    printBanner();

    // print detaills
    System.out.println("name:" + _name);
    System.out.println("amount:"  + amount);
}
```

- Important：Method name should represent its body.

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails(double amount) {
    System.out.println("name:" + _name);
    System.out.println("amount:" + amount);
}
```

- Operation
  - Make a code snippet a method and name it descriptively.

Exercise: Calculation of party price（Basics）　― next to-do

- Test for using a coupon

```
@Test
public void testCoupon()
{
    DinnerReservation resrv = new DinnerReservation();
    resrv.setUser(10);
    resrv.setCourse(DinnerReservation.Course.Matsu);
    resrv.setCoupon(1);
    assertEquals(60000, resrv.getCharge());
}
```

~~Calculate price（Matsu）~~
When coupons are used
3 coupons limit
~~Calculate price（Take）~~
~~Calculate price（Ume）~~
~~Course constants~~
~~Method for unit price~~

🟥

Exercise: Calculation of party price（Basics） ー implement

- Implement discount by coupons

```
class DinnerReservation
{
    (omitted)

    private int _coupon;

    public void setCoupon(int coupon) {
        _coupon = coupon;
    }

    public int getCharge() {
        int price = getPrice(_course);
        return (price * _number) - (10000 * _coupon);
    }

    (omitted)
```

~~Calculate price（Matsu）~~
When coupons are used
3 coupons limit
~~Calculate price（Take）~~
~~Calculate price（Ume）~~
~~Course constants~~
~~Method for unit price~~

Exercise: Calculation of party price（Basics）　—　exception test

- When more than 3 coupons are used, raise a exception.

~~Calculate price（Matsu）~~
~~When coupons are used~~
3 coupons limit
~~Calculate price（Take）~~
~~Calculate price（Ume）~~
~~Course constants~~
~~Method for unit price~~

```
@Test(expected=CouponException.class)
public void testCouponException()
{
    DinnerReservation resrv = new DinnerReservation();
    resrv.setCoupon(4);
}
```

🟥

Exercise: Calculation of party price（Basics）　―　exception code

- When more than 3 coupons are used, raise a exception.

```
class CouponException extends RuntimeException
{
}
```

```
class DinnerReservation
{
    (omitted)

    public void setCoupon(int coupon) {
        if (coupon > 3) {
            throw new CouponException();
        }
        _coupon = coupon;
    }

    (ommit)
}
```

~~Calculate price（Matsu）~~
~~When coupons are used~~
3 coupons limit
~~Calculate price（Take）~~
~~Calculate price（Ume）~~
~~Course constants~~
~~Method for unit price~~

What to learn in this chapter（again)

- Basics
  - Basic steps of TDD.
  - Three strategies achieving green
  - Tests for Exceptions
- Advanced 1
  - Evolutional design
  - Fake it in action
  - Express details by tests
- Advanced 2
  - Responsibility change with metaphor change

Chapter 4 Exercise: Calculation of party price

4.1 Basics

4.2 **Advanced 1**

4.3 Advanced 2

4.4 Summary

Exercise: Calculation of party price（Advanced1） – Specification

- What if multiple courses are ordered in one party?
  - e.g.） 3 users for Matsu and 10 users for Take
- Make it possible to set users by courses.
- Reset the to-do list.

Multiple courses

Exercise: Calculation of party price（Advanced1）　―　What to do

- Modify the code according to the contents of following slides.
- Program in pair, discussing with each other.
- At first, fake it

Multiple courses

- Take the following steps

1. Write a test to add one course to a dinner reservation using its new interface（addCourse）.
2. Implement for the test using addCourse.
3. Change existing tests to use the new interface.
4. Delete old interfaces（setUser, setCourse）.
5. Improve internal implementation（use a HashMap）
6. Write a test to add multiple courses.
7. Implement for the test to add multiple courses.

The purpose of this exercise is to take as small as possible steps and change the design safely. Proceed with fake it.

Exercise: Calculation of party price（Advanced1） – starting state

- Before

```
class DinnerReservation
{
    private int _number;
    private Course _course;

    public void setUser(int number) {
            _number = number;
    }
    public void setCourse(Course course) {
            _course = course;
    }

    public int getCharge() {
        int price = getPrice(_course);
        return (price * _number) - (10000 * _coupon);
    }

    (omitted)
```

```
@Test
public void testMatsu() {
    DinnerReservation resrv = new DinnerReservation();
    resrv.setUser(10);
    resrv.setCourse(DinnerReservation.Course.Matsu);
    assertEquals(70000, resrv.getCharge());
 }
```

Set party information with setUser(:int) and setCourse(:Course)

Exercise: Calculation of party price（Advanced1） – goal

- After (production code)

```
class DinnerReservation
{
    private HashMap<Course, Integer> _courseTable = new HashMap<Course, Integer>();

    public void addCourse(int number, Course course) {
        _courseTable.put(course, number);
    }

    public int getCharge() {
        int charge = 0;
        for (Map.Entry<Course, Integer> course :_courseTable.entrySet()) {
            int price = getPrice(course.getKey());
            int number = course.getValue();
            charge += price * number;
        }
        return charge - 10000 * _coupon;
    }
}

     (omitted)
```

Use a HashMap internally. HashMap is a collection consisting of pairs of key and value.

Exercise: Calculation of party price（Advanced1） – goal

- After (test)

```
@Test
public void testAddTwoCourse()
{
    DinnerReservation resrv = new DinnerReservation();
    resrv.addCourse(3, DinnerReservation.Course.Matsu);
    resrv.addCourse(10, DinnerReservation.Course.Take);
    assertEquals(71000, resrv.getCharge());
}
```

Set the information with addCourse(:int, :Course) to deal with multiple courses in one party

This time, program in pair

How to do

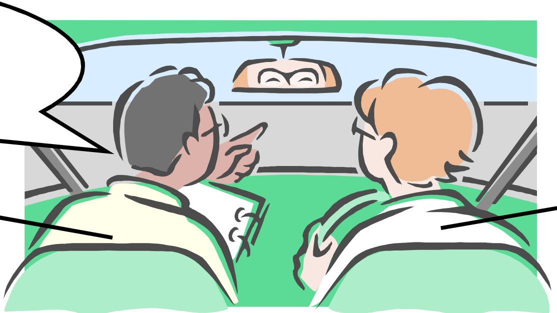
- Two programmers sit in front of one machine.
- Type codes in turn.

Merits

- Knowledge transfer to the team.
- Novice programmers can be trained.
- Make what you do not know clear

Notice

- Please feel free to ask questions to the trainer.
- Switch roles in moderate intervals.
- Discuss how to interpret your code in the way.



Navigator
・Always check
・Away from the goal?
・Code has defects?

Driver
・Type code
・Rewrite design

Exercise: Calculation of party price（Advanced1） ー start

Multiple courses

- Let’s begin

Hint

First, make it possible to add one course using the addCourse(:int,:Course) method.

Exercise: Calculation of party price（Advanced1） – write a test

- First, only for one course.
- Add an interface to add courses

Multiple courses
Add one course

```
@Test
public void testAddOneCourse()
{
    DinnerReservation resrv = new DinnerReservation();
    resrv.addCourse(10, DinnerReservation.Course.Matsu);
    assertEquals(70000, resrv.getCharge());
}
```

A example for fake it

🟥

Exercise: Calculation of party price（Advanced1） ― implement

- Product code to add one course(Red)

```
class DinnerReservation {
    （omitted）

    public void addCourse(int number, Course course)
    {
    }
}
```

🟥

Multiple courses
Add one course

- Real implementation(Green)

```
class DinnerReservation {
    （omitted）

    public void addCourse(int number, Course course)
    {
        _number = number;
        _course = course;
    }
}
```

🟩

Exercise: Calculation of party price（Advanced1） – modify existing tests

- Modify existing tests①

Multiple courses
Add one course

```
@Test
public void testMatsu() {
    DinnerReservation resrv = new DinnerReservation();
    //resrv.setUser(10);
    //resrv.setCourse(DinnerReservation.Course.Matsu);
    resrv.addCourse(10, DinnerReservation.Course.Matsu);
    assertEquals (70000, resrv.getCharge());
}

@Test
public void testTake() {
    DinnerReservation resrv = new DinnerReservation();
    //resrv.setUser(10);
    //resrv.setCourse(DinnerReservation.Course.Take);
    resrv.addCourse(10, DinnerReservation.Course.Take);
    assertEquals (50000, resrv.getCharge());
}
      (continued)
```

Exercise: Calculation of party price（Advanced1） – modify existing tests

- Modify existing tests ②

```
        (continued)
@Test
public void testUme() {
    DinnerReservation resrv = new DinnerReservation();
    //resrv.setUser(10);
    //resrv.setCourse(DinnerReservation.Course.Ume);
    resrv.addCourse(10, DinnerReservation.Course.Ume);
    assertEquals (30000, resrv.getCharge());
}
@Test
public void testCoupon() {
    DinnerReservation resrv = new DinnerReservation();
    //resrv.setUser(10);
    //resrv.setCourse(DinnerReservation.Course.Matsu);
    resrv.addCourse(10, DinnerReservation.Course.Matsu);
    resrv.setCoupon(1);
    assertEquals (60000, resrv.getCharge());
}
```

Multiple courses
Add one course

Exercise: Calculation of party price（Advanced1） – remove old implementation

- Remove unnecessary code

Multiple courses
<u>Add one course</u>

```
class DinnerReservation
{
    (omitted)
   //public void setUser(int number) {
   //    _number = number;
   //}

   //public void setCourse(Course course) {
   //    _course = course;
   //}
    (omitted)
}
```

These interfaces are changeable since they are internal (not public externally).
（As proceeding implementation, find the best design)

Exercise: Calculation of party price（Advanced1） – review the code

- In order to add multiple courses, use a HashMap internally.
  - Keeping existing way to store data unchanged, add a new attribute.

```
import java.util.HashMap;

class DinnerReservation {
    private HashMap<Course, Integer> _courseTable = new HashMap<Course, Integer>();
    private int _number;
    private Course _course;

    public void addCourse(int number, Course course) {
        _number = number;
        _course = course;
        _courseTable. put(course, number);
    }

(omitted)
}
```

Multiple courses
~~Add one course~~
Use HashMap

Exercise: Calculation of party price（Advanced1） – review the code

- Modify the price calculation logic to get data from the HashMap.

Multiple courses
~~Add one course~~
Use HashMap

```
class DinnerReservation {
    // private int _number;
    private HashMap<Course, Integer> _courseTable = new HashMap<Course, Integer>();
    private Course _course;

    public void addCourse(int number, Course course) {
        // _number = number;
        _course = course;
        _courseTable. put(course, number);
}

    public int getCharge() {
        int price = getPrice(_course);
        int number = _courseTable.get(_course);
        return (price * number) - (10000 * _coupon);
    }
}
```

Exercise: Calculation of party price（Advanced1） – add a test

- Write a test for multiple courses

Multiple courses
~~Add one course~~
~~Use HashMap~~

```
@Test
public void testAddTwoCourse()
{
    DinnerReservation resrv = new DinnerReservation();
    resrv.addCourse(3, DinnerReservation.Course.Matsu);
    resrv.addCourse(10, DinnerReservation.Course.Take);
    assertEquals(71000, resrv.getCharge());
}
```

🟥

Exercise: Calculation of party price（Advanced1） – implement

- Implement to add multiple courses

Multiple courses

~~Add one course~~

~~Use HashMap~~

```
import java.util.Map;

class DinnerReservation
{
    private HashMap<Course, Integer> _courseTable = new HashMap<Course, Integer>();
    // private Course _course;

    public void addCourse(int number, Course course) {
    // _course = course;
        _courseTable.put(course, number);
    }

    public int getCharge() {
        int charge = 0;
        for (Map.Entry<Course, Integer> course :_courseTable.entrySet()) {
            int price = getPrice(course.getKey());
            int number = course.getValue();
            charge += price * number;
        }
        return charge - 10000 * _coupon;
    }
}
```

Exercise: Calculation of party price（Advanced1） – What to do next

- What if add same courses multiple times?
  - In the current implementation, it seems that latest order overrides the existing number.
  - It is not specified.
  - Then express the situation by a test and check it.

~~Multiple courses~~
~~Add one course~~
~~Use HashMap~~
Add same courses

Exercise: Calculation of party price（Advanced1） – add a test

- A test to add same courses.

~~Multiple courses~~
~~Add one course~~
~~Use HashMap~~
Add same courses

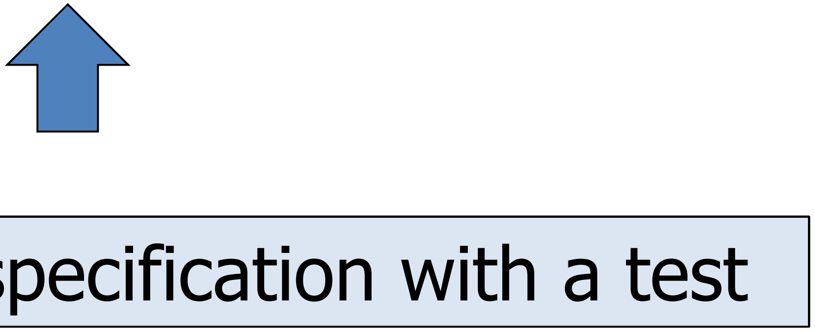
```
@Test
public void testAddSameCourse()
{
    DinnerReservation resrv = new DinnerReservation();
    resrv.addCourse(2, DinnerReservation.Course.Matsu);
    resrv.addCourse(3, DinnerReservation.Course.Matsu);

    assertEquals (21000, resrv.getCharge());
}
```

Exercise: Calculation of party price（Advanced1） – same courses

- We check the behavior when same courses are added with a test.

- The test code shows adding a same course overrides the existing number.
  - In another specification, you may have to add the number for the course.
  - In this exercise, adding same courses is overriding not adding.



Express detail specification with a test

What to learn in this chapter（again）

- Basics
  - Basic steps of TDD.
  - Three strategies achieving green
  - Tests for Exceptions
- Advanced 1
  - Evolutional design
  - Fake it in action
  - Express details by tests
- Advanced 2
  - Responsibility change with metaphor change

Chapter 4 Exercise: Calculation of party price

4.1 Basics

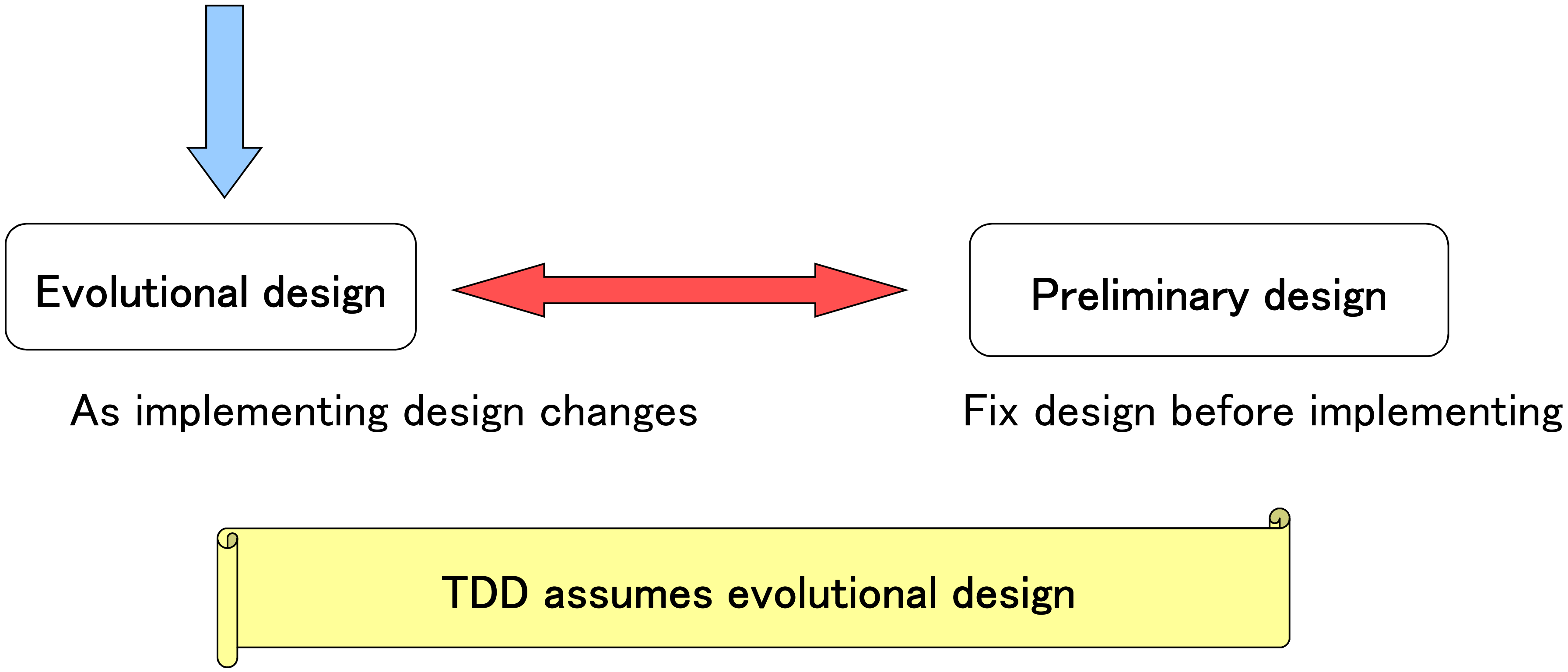
4.2 Advanced 1

4.3 **Advanced 2**

4.4 Summary

Assumption of TDD

- Good design makes extension and modification easier.
- You don’t know the best design until implementing it.



Evolutional design

Preliminary design

As implementing design changes

Fix design before implementing

TDD assumes evolutional design

Design in TDD

- Agile software development methods
  - Develop by a story
    - It assumes that developers have enough information to start developing
  - As little as design documents
    - They make only necessary documents to start developing
  - Friendly with object oriented
    - Encapsulation
    - Interfaces

- Design in TDD
  - Make behaviors clear
  - Design evolutionally
    - Evolve implementation of behaviors
    - Simple design
    - Metaphor

Design to make behaviors clear

- Object oriented design makes interfaces clear.
- Specify behaviors of classes with tests = design the classes
- Describe by tests not by documents
  - Test codes become “working specifications”.

Two meanings of evolutional design

- Grow up internal implementation evolutionally
- As simple as possible design each with to-do.
  - Dealing with to-dos, change the design little by little.
  - Avoid disadvantages of BDUF(Big Design Up Front).
  - Design to embrace change

Exercise: Calculation of party price（Advanced2） – Specification

- It was decided each course has three variations of Japanese, Western, and Chinese.
  - However, Chinese exists only in Matsu course, and extra charge of 500yen is needed.

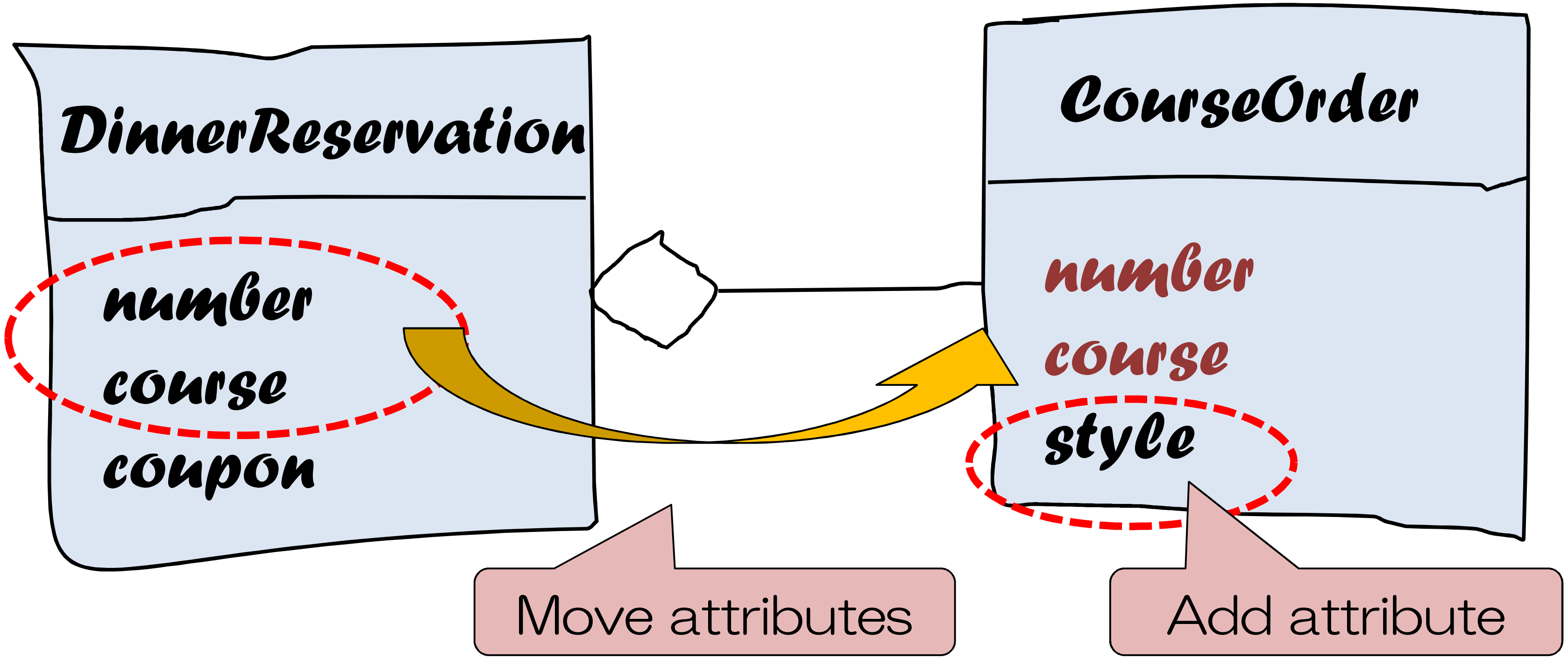
- Reset the to-do list.
  - Make Japanese/Western/Chinese
  - Error when Chinese and Take/Ume is combined.
  - Chinese course needs 500yen extra charge.

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu

Exercise: Calculation of party price（Advanced2） - Design consider

- Attributes associating orders such as styles are increasing.
- Applying a metaphor of CourseOrder, try to distribute responsibilities.
- Let DinnerReservation to focus on combining orders and calculating prices.

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
CourseOrder



Evolutional design: metaphor

- Metaphor
  - Change of responsibilities is described by a metaphor

- Responsibility
  - Structural design of classes to specify behaviors of the classes and their methods about what/how far they do.

- Example: draw money
  - When the responsibility includes interest calculation: metaphor of banks
  - When the responsibility does not include interest calculation: metaphor of safes

- Relationships
  - Relationships with other classes also change.

Exercise: Calculation of party price（Advanced2） – introduce a new metaphor

- Create CourseOrder class

```
class CourseOrder
{
    private int _number;
    private DinnerReservation.Course _course;

    public void setUser( int number) {
        _number = number;
    }

    public void setCourse( DinnerReservation.Course course) {
        _course = course;
    }
}
```

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
CourseOrder

Refactoring catalog:
You can use Introduce Parameter Object.

Exercise: Calculation of party price（Advanced2） – change an interface

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
<u>CourseOrder</u>

- Change one of existing tests

```
@Test
public void testMatsu() {
    DinnerReservation resrv = new DinnerReservati
    CourseOrder order = new CourseOrder();
    order.setUser(10);
    order.setCourse(DinnerReservation.Course.Matsu);
    resrv.addCourseOrder(order);
    assertEquals (70000, resrv.getCharge());
}
```

- Make it compile

```
class DinnerReservation {
     (omitted)

    public void addCourseOrder(CourseOrder order) {
    }
}
```

Exercise: Calculation of party price（Advanced2） – change an interface

- Add implementation①

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
CourseOrder

```
class CourseOrder {
    （omitted）

    public int getUser() {
        return _number;
    }
    public DinnerReservation.Course getCourse() {
        return _course;
    }
}
```

- Real implementation

```
class DinnerReservation {
    （omitted）

    public void addCourseOrder(CourseOrder order) {
        _courseTable.put(order.getCourse(), order.getUser());
    }
}
```

Exercise: Calculation of party price（Advanced2） – add creation methods

- To reduce cost need to fix test cases, add creation methods

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
~~CourseOrder~~
Creation method

```
@Test
public void testMatsu() {
    DinnerReservation resrv = new DinnerReservation();
    CourseOrder order = CourseOrder.createMatsuCourse(10);
    // CourseOrder order = new CourseOrder();
    // order.setUser(10);
    // order.setCourse(DinnerReservation.Course.Matsu);
    resrv.addCourseOrder(order);
    assertEquals (70000, resrv.getCharge());
}
```

Exercise: Calculation of party price（Advanced2） – add creation methods

- To reduce cost need to fix test cases, add creation methods

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
~~CourseOrder~~
Creation method

```
class CourseOrder
{
   （omitted）

   public static CourseOrder createMatsuCourse(int number)
   {
      CourseOrder order = new CourseOrder();
      order.setCourse(DinnerReservation.Course.Matsu);
      order.setUser(number);
      return order;
   }
}
```

static methods to create instances are called Creation Methods.

What is Creation method

- It is also called Factory method, but there a lot of interpretations.
  - GoF Factory method pattern
  - Methods to instantiate classes.
  - All methods to create objects.

- To make the meaning clear, this course calls following methods Creation Methods.
  - Methods to create instances of classes(normally static methods)

Exercise: Calculation of party price（Advanced2） – add creation methods

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
~~CourseOrder~~
Creation method

- Creation method for Take.

```
@Test
public void testTake() {
    DinnerReservation resrv = new DinnerReservation();
    CourseOrder order = CourseOrder.createTakeCourse(10);
    resrv.addCourseOrder(order);
    assertEquals(50000, resrv.getCharge());
}
```

🟥

- Implementation for Take

```
class CourseOrder
{
     (omitted)

    public static CourseOrder createTakeCourse(int number)
    {
        CourseOrder order = new CourseOrder();
        order.setCourse(DinnerReservation.Course.Take);
        order.setUser(number);
        return order;
    }
}
```

🟩

Exercise: Calculation of party price（Advanced2） – add creation methods

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
~~CourseOrder~~
Creation method

Creation Method for Ume

```
@Test
public void testUme() {
    DinnerReservation resrv = new DinnerReservation();
    CourseOrder order = CourseOrder.createUmeCourse(10);
    resrv.addCourseOrder(order);
    assertEquals(30000, resrv.getCharge());
}
```

🟥

Implementation for Ume

```
class CourseOrder
{
     (omitted)

    public static CourseOrder createUmeCourse(int number)
    {
        CourseOrder order = new CourseOrder();
        order.setCourse(DinnerReservation.Course.Ume);
        order.setUser(number);
        return order;
    }
}
```

🟩

Exercise: Calculation of party price（Advanced2） – add creation methods

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
~~CourseOrder~~
Creation method

- Modify existing tests to use creation methods (1)

```
@Test
public void testCoupon() {
    DinnerReservation resrv = new DinnerReservation();
    CourseOrder order = CourseOrder.createMatsuCourse(10);
    resrv.addCourseOrder(order);
    resrv.setCoupon(1);
    assertEquals (60000, resrv.getCharge());
}

// @Test
// public void testAddOneCourse() {
//     DinnerReservation resrv = new DinnerReservation();
//     resrv.addCourse(10, DinnerReservation.Course.Matsu);
//     assertEquals (70000, resrv.getCharge());
// }
```

Remove testAddOneCourse() as it is identical to testMatsu().

Exercise: Calculation of party price（Advanced2） – add creation methods

- Modify existing tests to use creation methods(2)

```
@Test
public void testAddTwoCourse() {
    DinnerReservation resrv = new DinnerReservation();
    CourseOrder order = CourseOrder.createMatsuCourse(3);
    resrv.addCourseOrder(order);
    CourseOrder order2 = CourseOrder.createTakeCourse(10);
    resrv.addCourseOrder(order2);
    assertEquals (71000, resrv.getCharge());
}

@Test
public void testAddSameCourse() {
    DinnerReservation resrv = new DinnerReservation();
    CourseOrder order = CourseOrder.createMatsuCourse(2);
    resrv.addCourseOrder(order);
    CourseOrder order2 = CourseOrder.createMatsuCourse(3);
    resrv.addCourseOrder(order2);
    assertEquals (21000, resrv.getCharge());
}
```

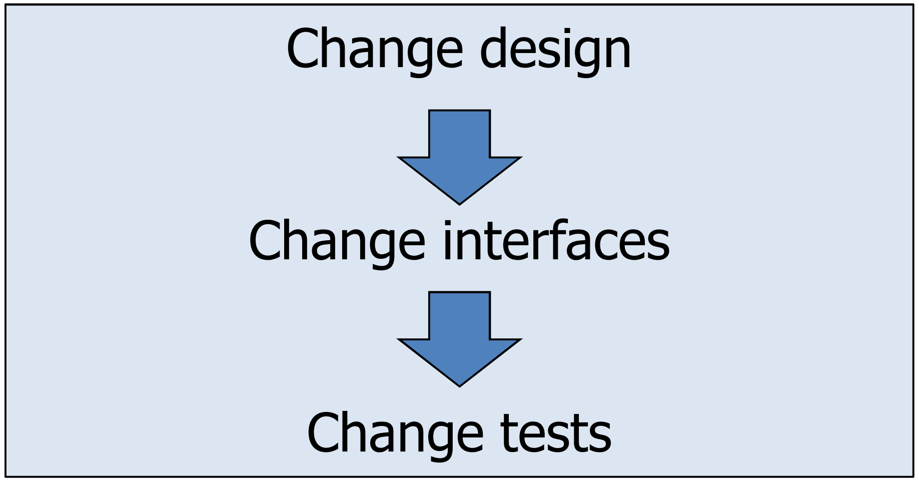
Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
~~CourseOrder~~
Creation method

※ Remove addCourse(:int :Course) in DinnerReservation class.

Exercise: Calculation of party price（Advanced2） – looking back

- What we have done
  - Introduce a new metaphor
  - Accompanied with it, change interfaces.

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
~~CourseOrder~~
~~Creation method~~



Change design

Change interfaces

Change tests

Exercise: Calculation of party price（Advanced2） – test for specification change

- Write a test to deal with styles
  - Add the style attribute to CourseOrder

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
~~CourseOrder~~
~~Creation method~~

```
@Test
public void testStyle()
{
    DinnerReservation resrv = new DinnerReservation();
    CourseOrder order = CourseOrder.createMatsuCourse(10);
    order.setStyle(DinnerReservation.Style.Japanese);
    resrv.addCourseOrder(order);
    assertEquals(70000, resrv.getCharge());
}
```

Exercise: Calculation of party price（Advanced2） – code for specification change

- Code for styles

Japanese/Western/Chinese
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
~~CourseOrder~~
~~Creation method~~

```
class CourseOrder
{
        (omitted)
    private DinnerReservation.Style _style;
    public void setStyle(DinnerReservation.Style style)
    {
        _style = style;
    }
        (omitted)
}
```

```
class DinnerReservation
{
        (omitted)
    public enum Style {Japanese, Western, Chinese}
        (omitted)
}
```

Exercise: Calculation of party price（Advanced2） – the next test

- Make Chinese and other than Matsu invalid.
  - Judge by isValid() method in CourseOrder.

~~Japanese/Western/Chinese~~
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
~~CourseOrder~~
~~Creation method~~

```
@Test
public void testValidation()
{
    CourseOrder order = CourseOrder.createTakeCourse(10);
    order.setStyle(DinnerReservation.Style.Chinese);
    assertFalse(order.isValid());

    order = CourseOrder.createUmeCourse(10);
    order.setStyle(DinnerReservation.Style.Chinese);
    assertFalse(order.isValid());

    order = CourseOrder.createMatsuCourse(10);
    order.setStyle(DinnerReservation.Style.Chinese);
    assertTrue(order.isValid());
}
```

🟥

Exercise: Calculation of party price（Advanced2） – the next code

- Implement the method to validate

~~Japanese/Western/Chinese~~
Error when Chinese and Take/Ume
Charge When Chinese and Matsu
~~CourseOrder~~
~~Creation method~~

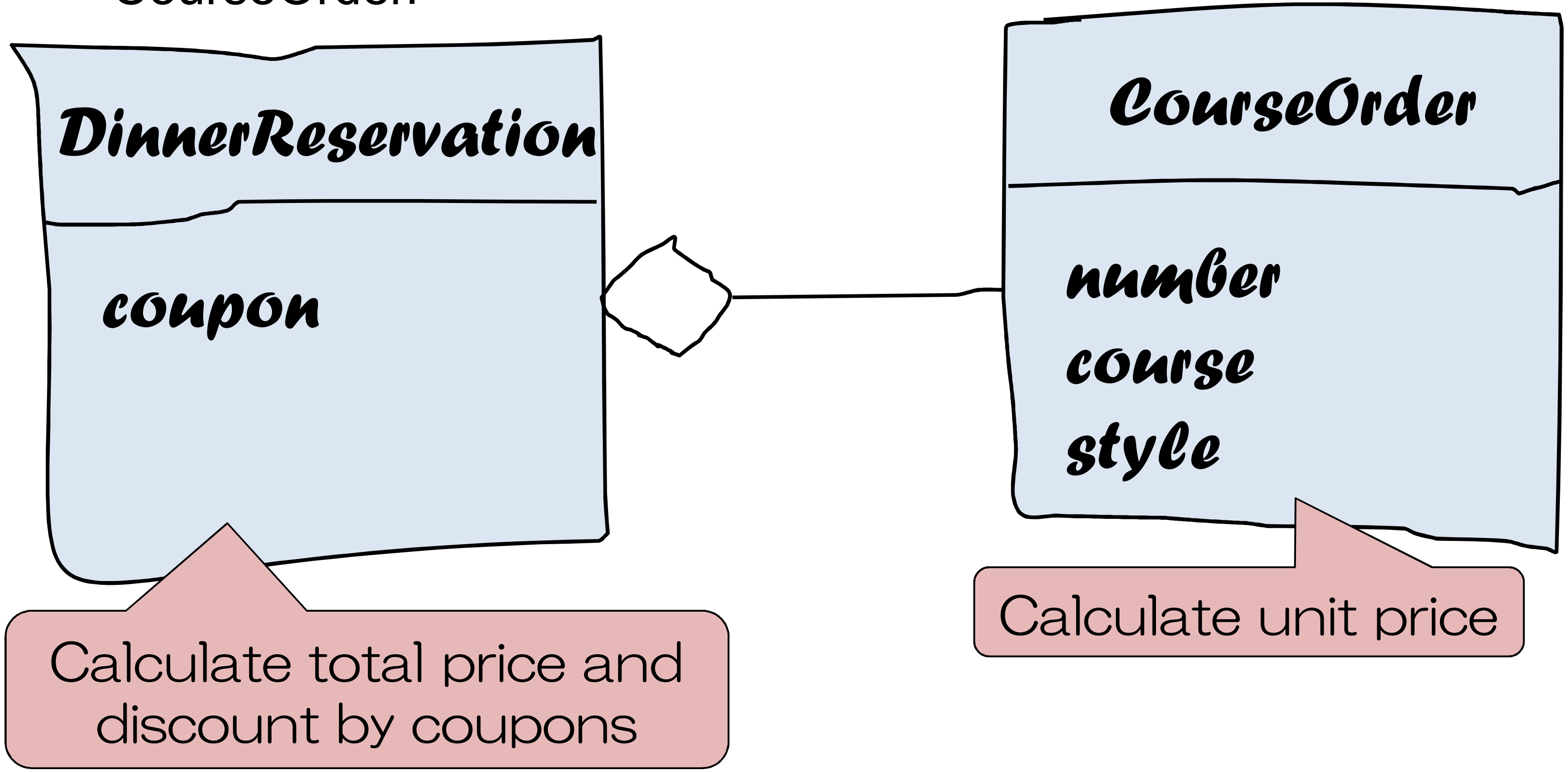
```
class CourseOrder
{
        (omitted)

    public boolean isValid() {
        if (_style == DinnerReservation.Style.Chinese) {
            if (_course == DinnerReservation.Course.Matsu) {
                return true;
            } else {
                return false;
            }
        } else {
            return true;
        }
    }
}
```

Exercise: Calculation of party price（Advanced2） – Change responsibilities

- Calculate Matsu course with Chinese style.
  - Attributes in CourseOrder decide unit price for the course, so calculation method should exist in CourseOrder.

~~Japanese/Western/Chinese~~
~~Error when Chinese and Take/Ume~~
Charge When Chinese and Matsu
~~CourseOrder~~
~~Creation method~~
Change calculation method



Exercise: Calculation of party price（Advanced2） － change responsibilities

~~Japanese/Western/Chinese~~
~~Error when Chinese and Take/Ume~~
Charge When Chinese and Matsu
~~CourseOrder~~
~~Creation method~~
Change calculation method

- As interfaces do not change, there is no need to change tests.

```
class CourseOrder {
      (omitted)

    public int getOrderCharge() {
        int price = getPrice(_course);
        return price * _number;
    }
    private int getPrice(DinnerReservation.Course course)   {
        switch (course) {
            case Matsu:
                return 7000;
            case Take:
                return 5000;
            case Ume:
                return 3000;
            default:
                return 0;
        }
    }
}
```

Exercise: Calculation of party price（Advanced2） – change responsibilities

- Also change DinnerReservation

```
class DinnerReservation {
    private HashMap<Course, CourseOrder> _courseTable = new HashMap<Course, CourseOrder>();
      (omitted)

    public void addCourseOrder(CourseOrder order) {
        _courseTable.put(order.getCourse(), order);
    }
    public int getCharge() {
        int charge = 0;
        for (Map.Entry<Course, CourseOrder > course :_courseTable.entrySet()) {
            charge += course.getValue().getOrderCharge();
        }
        return charge - 10000 * _coupon;
    }

    //private int getPrice(Course course)  {
    //    switch (course) {
          (omitted)
}
```

~~Japanese/Western/Chinese~~
~~Error when Chinese and Take/Ume~~
Charge When Chinese and Matsu
~~CourseOrder~~
~~Creation method~~
Change calculation method

A fairly big change has become possible because tests exist as a safety net.

Exercise: Calculation of party price（Advanced2） – add a new feature

- Write a test for additional charge.

~~Japanese/Western/Chinese~~
~~Error when Chinese and Take/Ume~~
Charge When Chinese and Matsu
~~CourseOrder~~
~~Creation method~~
~~Change calculation method~~

```
@Test
public void testAdditionalCharge()
{
    DinnerReservation resrv = new DinnerReservation();
    CourseOrder order = CourseOrder.createMatsuCourse(10);
    order.setStyle(DinnerReservation.Style.Chinese);
    resrv.addCourseOrder(order);
    assertEquals(75000, resrv.getCharge());
}
```

🟥

Exercise: Calculation of party price（Advanced2） – add a new feature

~~Japanese/Western/Chinese~~
~~Error when Chinese and Take/Ume~~
Charge When Chinese and Matsu
~~CourseOrder~~
~~Creation method~~
~~Change calculation method~~

- Code for the additional charge

```
class CourseOrder {
        (omitted)
    public int getOrderCharge() {
        int price = getPrice(_course);
        if (_course == DinnerReservation.Course.Matsu) {
            if (_style == DinnerReservation.Style.Chinese) {
                price = 7500;
            }
        }
        return price * _number;
    }
}
```

What to learn in this chapter（again）

- Basics
  - Basic steps of TDD.
  - Three strategies achieving green
  - Tests for Exceptions
- Advanced 1
  - Evolutional design
  - Fake it in action
  - Express details by tests
- Advanced 2
  - Responsibility change with metaphor change

Chapter 4 Exercise: Calculation of party price

4.1 Basics

4.2 Advanced 1

4.3 Advanced 2

4.4 **Summary**

What to learn in this chapter（again）

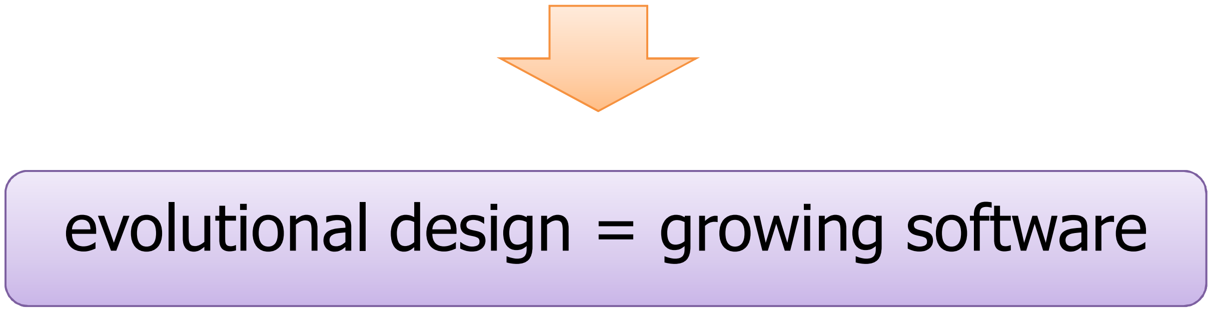
- Basics
  - Basic steps of TDD.
  - Three strategies achieving green
  - Tests for Exceptions
- Advanced 1
  - Evolutional design
  - Fake it in action
  - Express details by tests
- Advanced 2
  - Responsibility change with metaphor change

How to proceed TDD/Refactoring

- Pile up reliable parts little by little

- Switch strategies(piling up granularity) according to the situation.
  - Fake it
    - When complex implementation is to be needed, make it green by fake values and eliminate duplications later.
    - Making it green has the highest priority.
  - Obvious Implementation
    - When there seems to be an obvious solution, try it
    - When the solution turns out to be bad, go back and start over with fake it.

- Refactoring is a part of the cycle of TDD.
  - Implement just enough to current to-dos.
  - In the scope, make code as clean as possible.



evolutional design = growing software

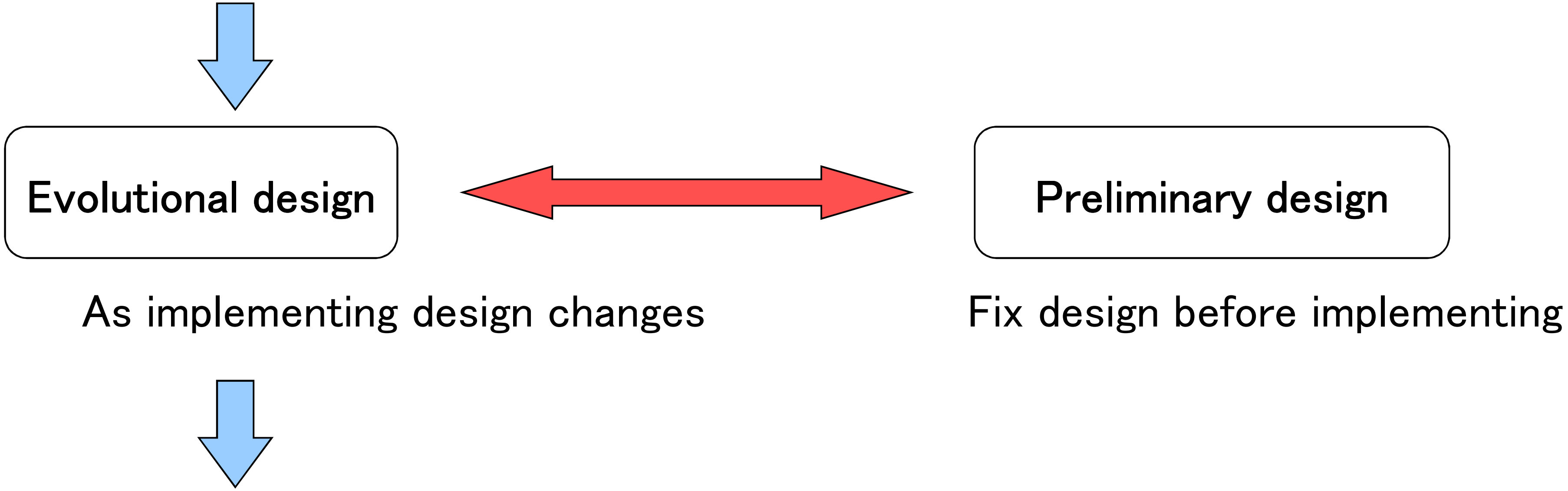
Meaning of test of TDD

- How far should we test?
  - Until gaining courage
    - Refactoring does not cause to degrade
    - Can change design evolutionally
  - Until the point that tests clarify specification
    - Tests are clarifying behavior of the software.
    - Team members can understand the intent by reading the tests.
  - Until meeting requirements from quality assurances
    - Test coverage as a goal requirement for the iteration
    - Test automation as a goal requirement for the iteration

The team determines how many things should be done with TDD, and leave other tests to later stages.

TDD and evolutional design

- Good design makes extension and modification easier.
- You don’t know the best design until implementing it.



Evolutional design

Preliminary design

As implementing design changes

Fix design before implementing

Evolve at different levels large or small

- Grow up internal implementation evolutionally.
- For each to-do, design as simple as possible.

Chapter 5 Summary of the Day

What is TDD?

- Definition
  - Development method to design and build programs by repeating the small steps of test and implementation to produce each small function of software.

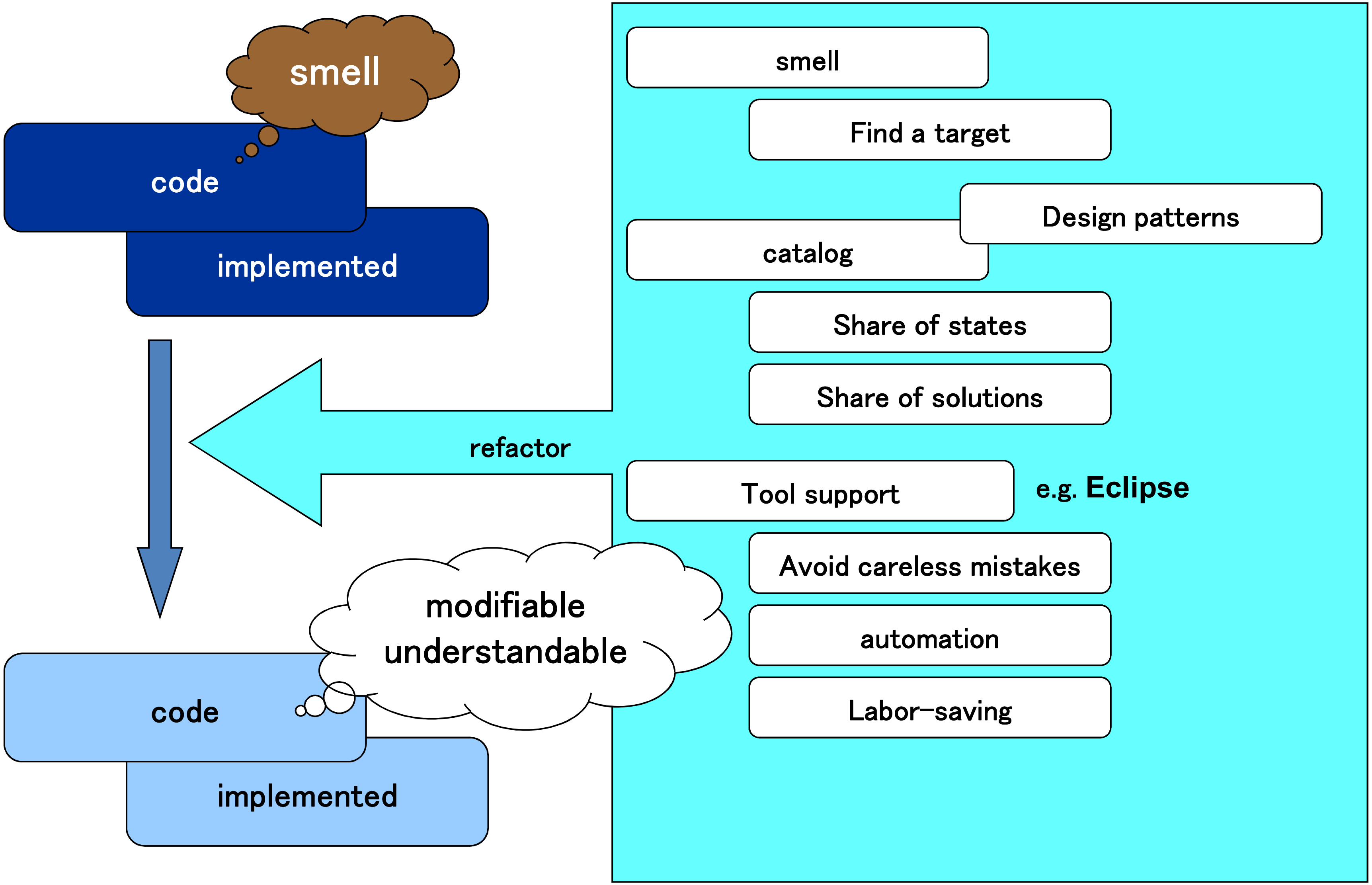
- How to proceed
  - Manage implementing functions with to do lists.
    - Write down behaviors needed and tasks to notepads as to−dos.
  - For each to−do, repeat the following
    - Firstly write a test.
    - Run the test ⇒ failed (RED)
    - Next write a production code to pass the test.
    - Run the test ⇒ passed (GREEN)
    - Refactor the code to the code as it should be.
    - Run the test ⇒ ensure it still passes (REFACTOR)

Essence of Test Driven Development (TDD)

- Design technique
  - Though writing tests, make how the production code behaviors clearer.
  - It assumes evolutional design.

- Development approach to pile up reliability
  - It is a technique to encourage development by piling up verifiable implementation little by little, which is possible only by the tests.
  - Developers write the tests by themselves.
  - Developers decides the range to test.
  - To stay a fixed rhythm, it needs rapid build and test environment.

Significance of refactoring



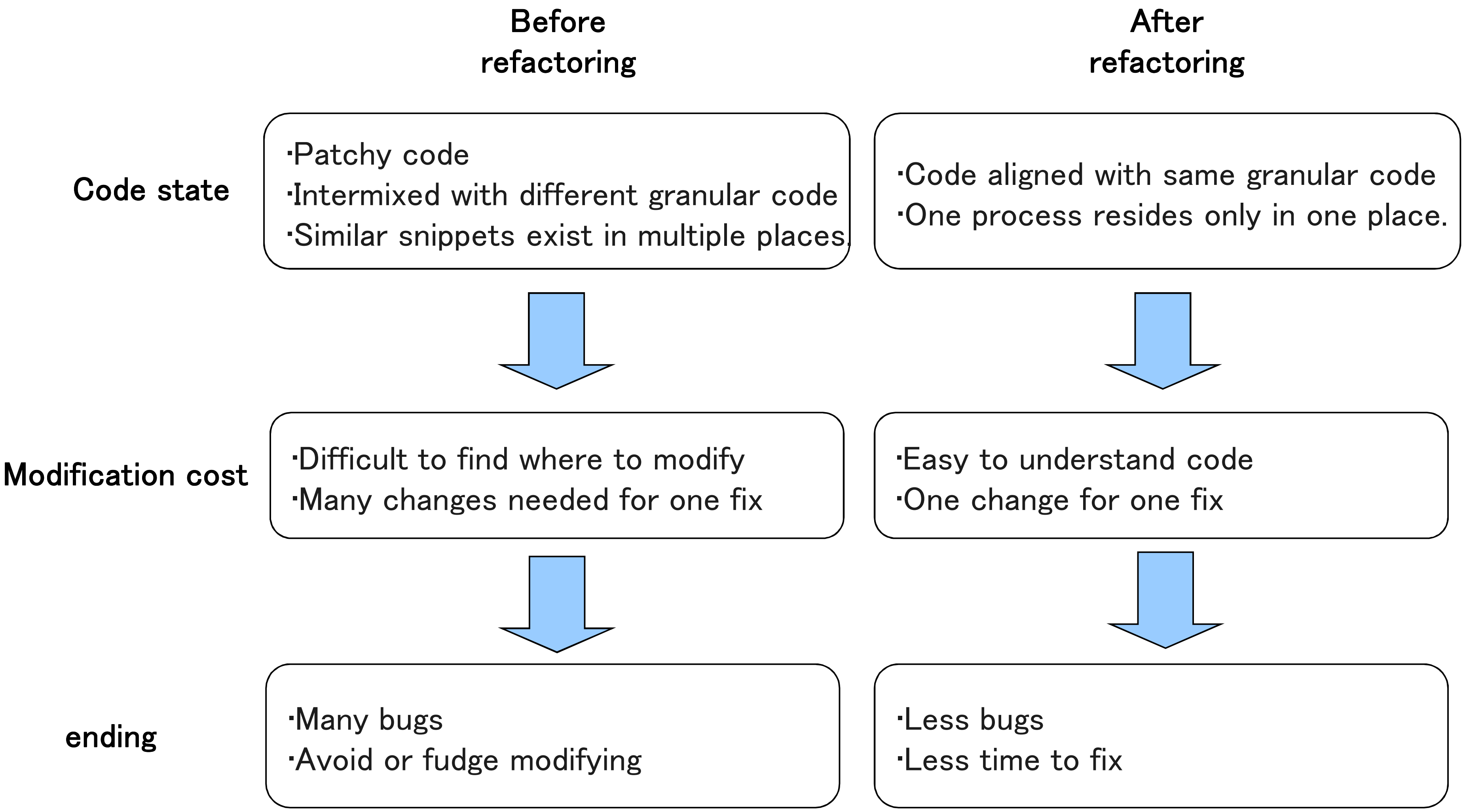
What is refactoring

Refactoring is

- To change internal structure with keeping external behavior
- To change internal structure so that the code becomes easy to understand and modify

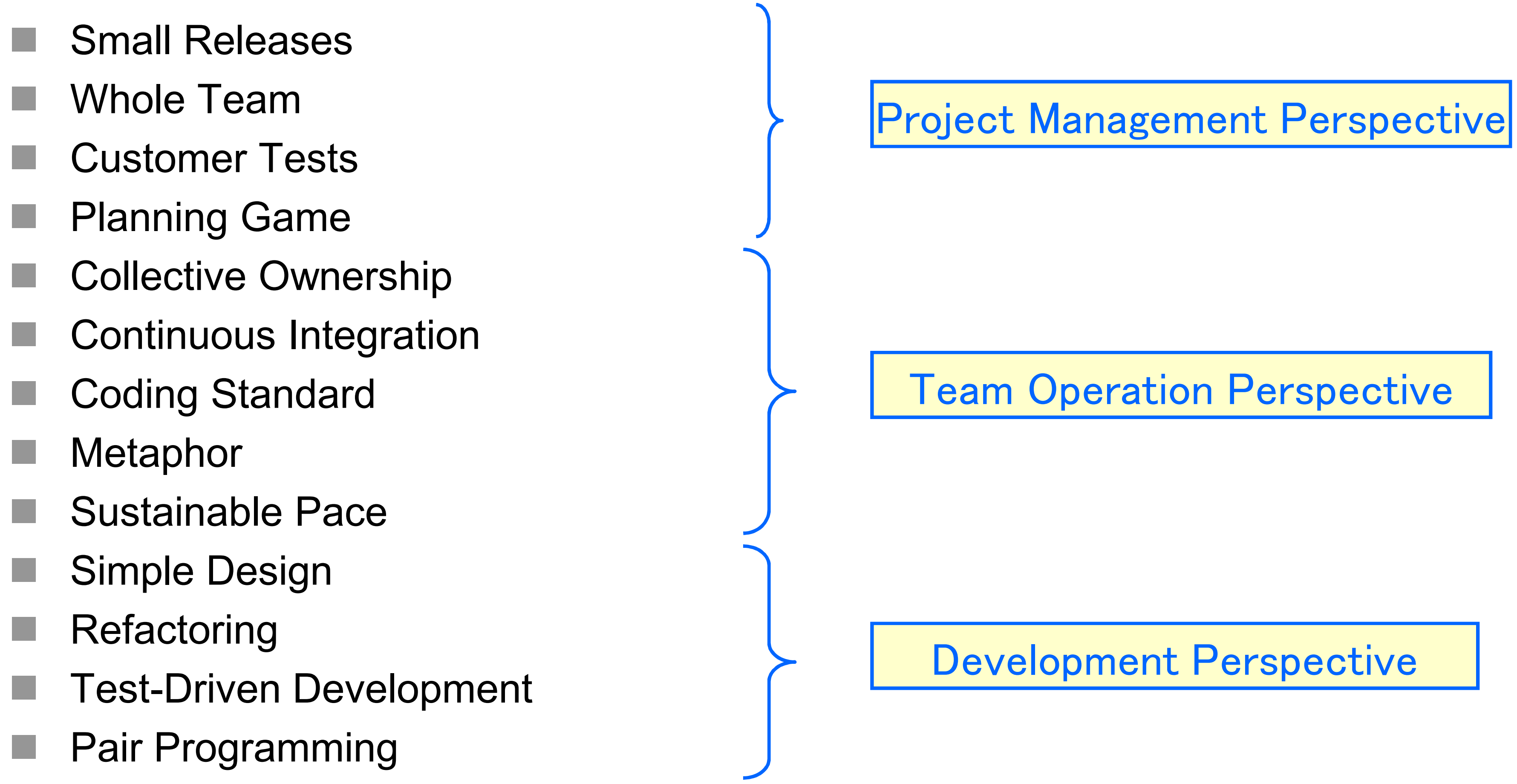
It does not increase current value of the software (there is no change in the external visible behavior)
It improves the future value.

Consequences of refactoring



	Before refactoring	After refactoring
Code state	･Patchy code ･Intermixed with different granular code ･Similar snippets exist in multiple places.	･Code aligned with same granular code ･One process resides only in one place.
Modification cost	･Difficult to find where to modify ･Many changes needed for one fix	･Easy to understand code ･One change for one fix
ending	･Many bugs ･Avoid or fudge modifying	･Less bugs ･Less time to fix

Positions of XP practices



- Small Releases
- Whole Team
- Customer Tests
- Planning Game
- Collective Ownership
- Continuous Integration
- Coding Standard
- Metaphor
- Sustainable Pace
- Simple Design
- Refactoring
- Test-Driven Development
- Pair Programming

Project Management Perspective

Team Operation Perspective

Development Perspective

References(1)

- Test Driven Development: By Example
  - Author：Kent Beck
- Extreme Programming Explained: Embrace Change
  - Author：Kent Beck

References(2)

- Refactoring: Improving the Design of Existing Code
  - Author：Martin Fowler

- Refactoring to Patterns
  - Author： Joshua Kerievsky

Introduction of Test Driven Development

(1st day)

Technologic Arts Incorporated

Contact

E-mail: training@tech-arts.co.jp

株式会社テクノロジックアート
TECHNOLOGIC ARTS INCORPORATED