

Robert C. Martin Series

Clean Code

A Handbook of Agile Software Craftsmanship

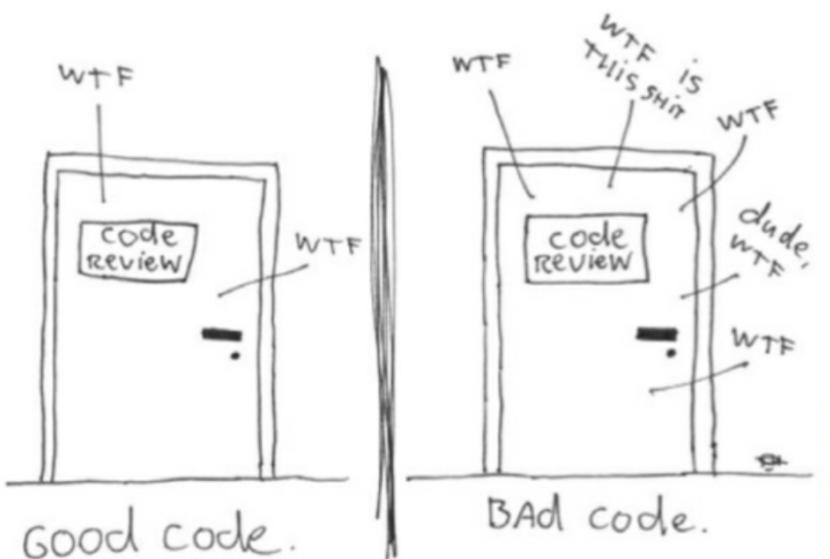


Foreword by James O. Coplien

Robert C. Martin

يجب الإهتمام ب software arch والإعتناء بالتفاصيل الصغيرة حيث يجب أن تكون متقدنة . في صناعة البرمجيات ٨٠٪ لعملية الصيانة . كذلك يجب التركيز على أن نبذل قصار جهدنا في عمل الكود والصدق بشأن حالة الكود بحيث لا يكون مزین بالواجهات لكن الهيكل مليء بالاخطاء . دائمًا سيكون الكود ليس مثالياً وبالتالي هناك حاجة لل clean code .

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift

Reproduced with the kind permission of Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m

** تشير الصورة انه مقبض الباب تالف والباب مليء بالمشاكل الغير متقد عملها يعكس الباب الآخر والذي ليس مثالياً لكنه أقل أخطاء .

أي باب يشبه كود فريقك او شركتك ؟
هل مراجعة الأكواد تكون عادية أم تكون بوجود عدد
من المشكلات الرهيبة ؟ كيف نعرف هل كودنا كما الباب
الأيمن (الأقل عدد من المشكلات) ؟
الجواب بالعمل الاحترافي (craftsmanship).
ولكسب هذه الحرفية نحتاج لشيئين :
١- المعرفة بمبادئ وأساسيات الانماط pattern .
٢- تطبيق المعرفة هذه بيده ورؤيتها بعينك بواسطة
العمل الجاد .

كما لو انك تعلمت فيزياء ركوب الدراجة لتصبح هاو
تحتاج للمارسة وستسقط مرات عدة قبل أن تتعلمها.
إن الكتاب ليس لمجرد القراءة العابرة فأنت بحاجة
للممارسة والتطبيق .

فالكتاب ٣ اجزاء :
الجزء الاول مبادئ وأنماط وممارسات ال clean code .
ولكنه غير كافي أبدا ولا يمكنك التوقف بعد قراءة هذا
الجزء .

الجزء الثاني : يحتوي على دراسة حالة للعديد من
الأكواد التي بها مشاكل لتصبح أكواد بها أقل مشاكل .

الجزء الثالث : يحتوي تفاصيل وخصائص وإرشادات لما
تم جمعه أثناء دراسة الحالة . وتعد قيمة جدا للقارئ تم
جمعها من تحويل الكود ل clean فى دراسات الحالة .

إذا تم قراءة الجزء الأول والثالث بدون الثاني فلن تكون استفدت من الكتاب تماما ولكن إذا بذلت جهدا بالجزء الثاني ستكون قد استفدت الكثير.

الفصل ١ clean code:

- There will be code سيكون دائما الكود موجود لتحويل متطلبات العميل لأوامر تفهمها الآلات.

ثم يتكلم المؤلف حول ال bad code من واقع الخبرة نعلم بأهمية الكود ال clean لأننا عانيمنا عندما لم تكن الأكواد كذلك. كمثال تم ذكر شركة قامت بعمل تطبيق كان ناجحا في الثمانينات ولكن بسبب الأخطاء التي تعذر إصلاحها في الإصدارات القادمة تم إغلاق المنتج وتوقفت الشركة. والسبب أنهم كانوا يضيفوا المميزات وذلك زاد المشاكل في الكود حتى أصبح من غير الممكن إدارة هذه المشاكل.

هل سبق أن صنعت كود سيء لأنك تريد إنجازه بسرعة؟ أو لأنك شعرت أنه ليس لديك الوقت الكاف لعمل جيد؟ أو لأنك سئمت من صنع البرامج أو لأنك نظرت لتراكم الأشياء حولك؟ إذا كان نعم كما فعلنا نحن سابقا لأننا ظننا أنها سنعود يوما ونصلحه لكن ذلك لم يحدث لأننا

لم نكن نعرف المقوله " later equals never ".

تكلفة الفوضى Mess في الكود , كلما زادت تقل الإنتاجية وتحتاج الإدارة لمزيد من الموظفين وقد تحدث تغيرات ليست ضمن أهداف التصميم بالعكس تحبط التصميم وبالتالي قد تسبب في توقف الإنتاجية.

ما هو دورك في المؤسسة التي يظن فيها أنه يتم تلبية رغبة متطلبات العملاء ورغبة المدراء حتى وإن تسببت في إخفاقات للكود ؟ يجب أن لا تخاف من معارضة المدراء فمعظم المدراء يريدون الحقيقة.

حتى إن كانوا يدافعون للحفاظ على متطلبات العميل والجدول الزمني للعمل بهذه وظيفتهم أما الحفاظ على الكود بهذه مسؤوليتك . لذلك من غير المهني أن يتنازل المبرمجين عن الـ clean في الكود لأجل إرادة المدراء لهذا.

من الخطأ الإستعجال بالكود لأجل التسليم بالموعد وعمل كود فوضوي Mess ، لأنه سوف يؤخرك عن موعد التسليم وسيبطئ تقدمك وإن الطريقة الأفضل للمضي قدما هي بعمل كود clean.

يعد الكود الـ clean إحساس وتطبيق لعدد من

الأساليب الصغيرة كما الفنان الذي يحدث فرقاً عبر تحويل الرسم لشيء أكثر أناقة.

رأي بعض المبرمجين ما هو الـ **clean code** مؤلف الـ **C++**

كون الكود أنيق وفعال يسهل معرفة الأخطاء فيه (تتبع الأخطاء) بأقل عدد ممكن التبعيات **dependency** يؤدي مهمة واحدة بشكل جيد .

مؤلف الـ **object oriented** الكود النظيف بسيط ومباشر يقرأ كالنثر يحتوي على ما هو ضروري و واقعي.

ل بيج ديف توماس انه يمكن لمطور اخر غير الذي كتب الكود ان يقرأ الكود ويتطوره ، كذلك تم اختبار الـ **unit , acceptance** عليه، التسمية فيه تحمل معنى العمل الذي تقوم به قابل للقراءة للبشر ، بأقل عدد من التبعيات ، بواجهة واضحة.

مؤلف كتاب التعامل الفعال مع الأكواد القديمة هو كود كتب بواسطة شخص يهتم و يفكر بالطرق الممكنة لتحسين الكود، بكلمة واحدة " الاهتمام ".

مؤلف آخر

يطبق عليه جميع الاختبارات

لا يتكرر فيه العمل

يمثل فكرة التصميم المطلوبة في النظام .

أقل عدد من الدوال والكيانات والوظائف.

أهم نقطة هي تكرار العمل عندما تقوم بعمل ثم تكرر العمل بشكل محسن هذا يدل على عدم قدرتنا على التعبير عما في أذهاننا . بالنسبة للنقطة الأخيرة يتم تقسيم الكيانات لكيانات متعددة عندما يكون لها أكثر من وظيفة يتم التقسيم بحسب العمل الذي تقوم به.

رأي بوب عن الكود الـ clean :

رأيه هو ما أورده ضمن هذا الكتاب وكما في مدارس الفنون لا توجد مدرسة هي الأفضل ولكن لكل مدرسة منها الذي تتبعه والطريقة التي يشرح بها الكود هي فن يمارس، اكتسب من خلال الخبرة في هذا المجال.

خلاصة الفصل الأول:

لا يعدك الكتاب يجعلك فنانا ولكنه يقدم حيل وتقنيات وأدوات يستخدمها المبرمجين . ستجد عدد من الأمثلة

من الأكواد تم تحويلها ل코드 جيد ولا بد من ممارسة هذه الأكواد .

الفصل ٢ : أسماء ذات معنى

استخدام أسماء تعبّر عن المقصد " نحن نستخدم الأسماء لتسمية المتغيرات، الدوال، البارامترات، المسارات، ملفات ال jar و والكثير .." لذا يجب أن تكون التسمية بشكل جيد. يجب العناية بإختيار الأسماء وإن أخذ ذلك وقتاً ثم تعديلها إن وجد إسم أفضل. اختيار الإسم يجب أن يجرب عن سبب وجود الكلاس أو الدالة.

int d; // days

هنا إسم المتغير لا يعبر عن المعنى. 

int elapsedTimelnDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;

الأسماء السابقة تعبّر عن وظيفتها وتعبّر عن المقصد. يصعب فهم الكود السابق لأنّه يصعب الإجابة عن

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

الأسئلة:

ما هي الأشياء الموجودة في ال list ؟
لما ال index يبدء من الصفر ؟
ما الهدف من القيمة 4 ؟

كيف أستخدم القيمة التي تم ارجاعها؟
بسبب غموض هذه الأسئلة أصبح الكود معقدا.
إفرض أن الكود السابق ضمن كلاس لعبة كاسحة الألغام
فالتعديل mine sweeper game
سيكون كالتالي :

لاحظ مع أن الكود لم يتغير إلى أنه أصبح أكثر وضوحا

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new ArrayList<int[]>();  
    for (int[] cell : gameBoard)  
        if (cell[STATUS_VALUE] == FLAGGED)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

وفهما لتعديل الكود وجعله أفضل حول المصفوفة
لكلاس بسيط إسم الكلاس يعبر عن العمل الذي يقوم به
وتم إستبدال الشرط if بدالة isFlagged كالتالي :

```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells = new ArrayList<Cell>();  
    for (Cell cell : gameBoard)  
        if (cell.isFlagged())  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

بالتأكيد أن الكود أصبح أبسط بكثير.

تجنب التضليل

عندما يتم كتابة المتغيرات تحت إسم مثل `hp,aix, sco`. حتى لو كان من وجه الإختصار والتبسيط فتعد تسمية مظللة.

لا تستخدم التعبير `accountList` إذا لم تكن في الواقع `List` لأنها ستكون مظللة للمبرمجين. و كمثال إذا كانت مجموعة من الحسابات وليس `list` أشر إليها بالإسم `accountGroup`.

**ستتعلم لاحقا أنه حتى وإن كانت من نوع `list` من الأفضل عدم ذكر النوع ضمن الإسم.

كذلك تجنب التسمية المتكررة والتي الفرق فيها بسيط كما في المثال التالي :

`XYZControllerForEfficientHandlingOfStrings`
`XYZControllerForEfficientStorageOfStrings`

في لغة الجافا يستعين المطورين بالاكمال التلقائي

لمعرفة الأكمالات المحتملة، لذلك من المفيد ترتيب الأسماء المتماثلة لتجنب اختيار المطورو لاسم مماثل بدون قصد.

من الأمثلة السيئة إستخدام الأسماء بالأحرف الصغيرة L أو الكبيرة LL خاصة في حالة التعامل مع الجمع

```
int a = 1;  
if ( 0 == 1 )  
    a = 01;  
else  
    l = 01;
```

يستخدم فروق ذات معنى

حين يصادف المبرمج شيئين مختلفين للإشارة لـ إسم واحد فقد يقع المطورو بخطأ وضع تسمية عشوائية فيها خطأ إملائي (للتفريق بين الشيئين) و حين يتم تصحيح الأخطاء ستظهر مشاكل أثناء ال compile . لا يكفي أن تضيف أحرف أو أرقام عشوائية للإسم لحل المشكلة فعندما تتغير الأسماء لابد أن تختلف الوظيفة.

** مثال لأن كلمة class مستخدمة قرر المطورو كتابة كـ اسم لمتغير klass

ذلك يجب أن تكون الأسماء يمكن قراءتها ونطقها . مثل

```
public static void copyChars(char a1[], char a2[]) {  
    for (int i = 0; i < a1.length; i++) {  
        a2[i] = a1[i];  
    }  
}
```

source, destination a1 هذا لو استخدمنا بدلا عنه للبارامترات المرسلة.

يجب أن لا تستخدم كلمة **variable** للتعبير عن إسم المتغير او أن تستخدم **NameString** للتعبير مثلا عن ال Name لأنه لن يكون سوى string ، كذلك .. لأنه عندما يكون هناك على سبيل المثال **customer**, **customer object** كيف سيعرف المطور أي واحدا سيقوم باستدعائه . المشكلة هنا وجود فروقات بدون اختلاف في المعنى .

```
getActiveAccount();  
getActiveAccounts();  
getActiveAccountInfo();
```

ذلك يجب أن تكون الأسماء منطقية لأن ذلك أسهل للبشر أن يتعاملوا مع ما هو منطوق يمكن تهجئة الأحرف المستخدمة لأن البرمجة نشاط جماعي . في هذا المثال كانت الشركة تستخدم هذه المسميات

للتعبير عن `genymdhms` " generation date, year, month, day, hour, minute, and second " فتبدو " التسميات مزعجة ويحتاج المبرمجين لشرح التسميات للمطورين الجدد بدل من إستخدام تسميات إنجليزية واضحة، قارن المثالين لاحظ أن تبادل الآراء بين المطورين الآن سيكون 

```
class DtaRcrd102 {  
    private Date genymdhms;  
    private Date modymdhms;  
    private final String pszqint = "102";  
    /* ... */  
};  
  
to  
  
class Customer {  
    private Date generationTimestamp;  
    private Date modificationTimestamp;;  
    private final String recordId = "102";  
    /* ... */  
};
```

أسهل !!! أهلاً مایکی ، انظر على هذا السجل كيف بإمکانی ضبطه على تاريخ غدا ؟ !!!.

أسماء قابلة للبحث
من السهل ان تجد إسم الثابت `MAX_CLASS_PER_STUDENT` لكن لو تركناه رقما مثلا
من الصعب أن نجده أو نتعامل معه.

```

        for (int j=0; j<34; j++) {
            s += (t[j]*4)/5;
        }

to

int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
    sum += realTaskWeeks;
}

```

كذلك الحرف `e` إذ لا يمكن استخدام متغير بحرف واحد إلا في المتغيرات المحلية وفي الدوال القصيرة ، كما يجب أن يكون طول الإسم متوافق مع نطاقه الذي يستخدم فيه ، وكلما زاد طول الإسم أصبح من السهل أن نجده. إذا كان المتغير يستخدم أكثر من مرة أو قم باعطائه إسما يسهل البحث عنه بدل من `s`. تم استخدام الـ `sum` برغم طول الكود انه من السهل ان نعثر على `WORK_DAY_PER_WEEK` بدلا عن الرقم ^٥ داخل الكود.

لا تستخدم الترميزات لأجل التبسيط وتسهيل الكود ايضا لا تحتاج لبادئة مثلا `m` لأنه يجب أن تكون أسماء الكلاسات والدوال قصيرة.

شائع استخدام بعض الترميزات مع الـ `interfaces`

```
public class Part {  
    private String m_dsc; // The textual description  
    void setName(String name) {  
        m_dsc = name;  
    }  
}
```

```
public class Part {  
    String description;  
    void setDescription(String description) {  
        this.description = description;  
    }  
}
```

كمثال إذا كنا نبني الذي يصنع الأشكال هذا المصنع سيتمثل ب interface ونحتاج لعمل كلاس والذي يعمل implements ل دوال الواجهة.

يمكن كتابة ال interface هكذا "" / **IShapeFactory** والأفضل أن تكتبها كالأخرية لأنك لا تحتاج أن يعرف المطور أنه يتعامل مع ال interface وإن أحببت استخدام الترميز فلكن مع ال تطوير واكتب الأسماء هكذا **ShapeFactory** " برغم **ShapeFactoryImp / CShapeFactory**" هذا بالإضافة هنا ليست مفضلة .

تجنب استخدام تسميات أنت تعرفها يجب على المبرمج أن يكتب تعليمات يفهمها الآخرين فالأسماء الشائعة للمطور مثل i,j,k في الدوالات loops و ! الشائعة حتى إن كانت ضمن مدى scope صغير

تعتبر ممارسة غير جيدة.

أسماء الكلاسات

يجب ان تكون اسماء noun or noun phrase . كما يجب ان لا يكون فعلا verb .

Customer, WikiPage, Account, and AddressParser

أسماء الدوال

يجب أن تحتوي على verb or verb phrase . postPayment, deletePage, or save .

ويجب أن تلتزم بمعايير اللغات البرمجية مثلا في الجافا استخدام ال get, set مع إسم المتغيرات من الأفضل عند عمل overload لل constructor

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

is generally better than

```
Complex fulcrumPoint = new Complex(23.0);
```

```
string name = employee.getName();  
customer.setName("mike");  
if (paycheck.isPosted()) ...
```

لا تختار الأسماء الترفيعية على حساب الأسماء الواضحة
لا تكن لطيفا باختيار الأسماء عبر استخدامك لأسماء

عافية مثال استخدام `whach()` بدلًا عن `kill()`.

استخدم كلمة واحدة للمفهوم الواحد من المربك أن تستخدم `fetch`, `get`, `retrieve` في نفس الكود لأن عمليات الإستدعاء لن تظهر تعليقات مطورة للدالة.

تجنب إستخدام التورية وهي إستخدام إسم مخالف لما يتم تنفيذه لتجنب التكرار مثلاً. لنفرض أن لدينا كلاس يقوم بعملية الإضافة باستخدام الدالة `add` وأردنا في كلاس آخر أن نقوم بعملية إدخال قيمة لمجموعة قيم فهل من الأفضل إستخدام الدالة `add`? برغم أنك قد تعتقد أن ذلك سيعزز الاتساق وتجنب للتورية ولكن لأن الوظيفة `insert` مختلفة من الأفضل إستخدام إسم مختلف مثل `append`. الهدف دائماً هو جعل الكود واضحًا يوضح نفسه.

استخدم أسماء يفهمها المطوريين بحيث لا تكون ضمن مفاهيم العميل ومن مصطلحات علوم الكمبيوتر والأنماط الشائعة والمعادلات الرياضية المتعارف عليها. غالباً ما يكون اختيار أسماء `technical`

هو الأنسب. وإذا لم يوجد إستخدام إسم يمكن للخبير معرفته، لأنه متعلق بالنظام.

يستخدم أسماء تحمل معاني قليلة هي الأسماء التي تحمل معنى بنفسها وتحتاج لعمل سياق يمثل معاني هذه الأسماء سواء كانت أسماء كلاسات أو دوال أو غيرها..

في المتغيرات `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state`, and `zipcode`

ستلاحظ أنها تدل على ال `address`. بخلاف لو كان متغير واحد فقط `state` مثلا. يمكن أن تعرف المتغيرات كالتالي `addrFirstName`, `addrLastName`, `addrState` والأفضل طبعاً أن تستخدم كلاس `Address` . إسمه

هل تحتاج المتغيرات بالشكل التالي إلى نصوص توضحها ؟ غالباً أسماء المتغيرات `number`, `verb`, `pluralModifier`.

برغم أن أسماء المتغيرات (عدد / فعل / مرات الجمع) تدل على 'تخمين الإحصاءات' لكن لازال يلزم تخمين السياق .

Listing 2-1

Variables with unclear context.

```

private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
        number = "no";
        verb = "are";
        pluralModifier = "s";
    } else if (count == 1) {
        number = "1";
        verb = "is";
        pluralModifier = "";
    } else {
        number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
    }
    String guessMessage = String.format(
        "There %s %s %s%s", verb, number, candidate, pluralModifier
    );
    print(guessMessage);
}

```

أيضا الدالة طويلة جدا لتبسيطها نحتاج لتقسيم هذه الدالة.

Listing 2-2

Variables have a context.

```

public class GuessStatisticsMessage {
    private String number;
    private String verb;
    private String pluralModifier;

    public String make(char candidate, int count) {
        createPluralDependentMessageParts(count);
        return String.format(
            "There %s %s %s%s",
            verb, number, candidate, pluralModifier );
    }

    private void createPluralDependentMessageParts(int count) {
        if (count == 0) {
            thereAreNoLetters();
        } else if (count == 1) {
            thereIsOneLetter();
        } else {
            thereAreManyLetters(count);
        }
    }

    private void thereAreManyLetters(int count) {
        number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
    }
}

```

```
private void thereIsOneLetter() {  
    number = "1";  
    verb = "is";  
    pluralModifier = "";  
}  
  
private void thereAreNoLetters() {  
    number = "no";  
    verb = "are";  
    pluralModifier = "s";  
}  
}
```

لا تضف سياق غير مبرر

لنفرض انك تعمل على نظام "Gas Station Deluxe" اذا قمت بإضافة GSD قبل أسماء الكلاسات فستواجهك مشكلة عندما تحاول إستدعاء أحد هذه الكلاسات بواسطة كتابة G وانتظار الإكمال ... ستظهر قائمة طويلة بجميع أسماء الكلاسات. لن تتمكن من الاستفادة من خدمات أدواتك وستصعب على الـ IDE مساعدتك ؟

خلاصة الكلام أصعب شيء في اختيار الأسماء أنها تتطلب مهارة وصفية وخلفية ثقافية وليس مسألة تقنية. لا تخف من أن يكون الإسم طويلا لأن الأدوات الحديثة تسهل الإكمال والعثور على أسماء الكلاسات والدوال. اتبع القواعد المذكورة لتحسين الكود الخاص بك و إذا كنت تقوم بصيانة كود شخص آخر فيمكنك

استخدام أدوات الـ refactoring لحل المشاكل الموجودة في الكود.

الفصل ٣ : الدوال

تعد الدوال أول خطوة لتنظيم البرنامج يتحدث هذا الفصل عن الكتابة الجيدة للدوال.

لاحظ الدالة في المثال من الصعب فهم الدالة فهي ليست فقط طويلة ولكن هناك كود مكرر والكثير من النصوص الغير مفهومة وأنواع البيانات وال APIs الغير واضح دورها .. خذ ٣ دقائق لفهم هذه الدالة.

على الأغلب أنك لن تفهم الدالة .. مع ذلك تمكنت من إعادة صياغة الدالة في ال ٩ الأسطر التالية:

خذ ٣ دقائق لفهمها غالبا لن تفهم جميع التفاصيل

Listing 3-1

HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup
) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup =
                PageCrawlerImpl.getInheritedPage(
                    SuiteResponder.SUITE_SETUP_NAME, wikiPage
                );
            if (suiteSetup != null) {
                WikiPagePath pagePath =
                    suiteSetup.getPageCrawler().getFullPath(suiteSetup);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -setup .")
                    .append(pagePathName)
                    .append("\n");
            }
        }
    }
}
```

```

}
WikiPage setup =
    PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
if (setup != null) {
    WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
    String setupPathName = PathParser.render(setupPath);
    buffer.append("!include -setup .")
        .append(setupPathName)
        .append("\n");
}
}
buffer.append(pageData.getContent());
if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
        PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
        WikiPagePath tearDownPath =
            wikiPage.getPageCrawler().getFullPath(teardown);
        String tearDownPathName = PathParser.render(tearDownPath);
        buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
    if (includesuiteSetup) {
        WikiPage suiteTeardown =
            PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME,
                wikiPage
            );
        if (suiteTeardown != null) {
            WikiPagePath pagePath =
                suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
            String pagePathName = PathParser.render(pagePath);
            buffer.append("!include -teardown .")
                .append(pagePathName)
                .append("\n");
        }
    }
}
pageData.setContent(buffer.toString());
return pageData.getHtml();
}
}

```

ولكن قد تفهم أن الدالة تقوم بتضمين صفحات الإعداد والتفكير في صفحة اختبار ومن ثم تحول الصفحات إلى HTML . وإذا كنت خبير بJUnit ستدرك أن الدالة تنتمي إلى `.test framework`

الكود في 2-3 أسهل بكثير فما هو السبب ؟

Listing 3-2

HtmlUtil.java (refactored)

```

public static String renderPageWithSetupsAndTear downs(
    PageData pageData, boolean isSuite
) throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
        WikiPage testPage = pageData.getWikiPage();
        StringBuffer newPageContent = new StringBuffer();
        includeSetupPages(testPage, newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        includeTeardownPages(testPage, newPageContent, isSuite);
        pageData.setContent(newPageContent.toString());
    }
    return pageData.getHtml();
}

```

SMALL

كان سابقاً يقال بأن الدوال يجب أن تكون بحجم الشاشة لكن الآن مع تزايد حجم الشاشات يجب إطلاقاً أن لا تزيد الدالة عن ٢٠ سطراً. كم يجب أن يكون طول الدوال؟

يجب أن تتكون من سطرين أو ثلاثة أو أربعة أسطر فقط. لهذا الدالة السابقة يجب أن تبسط كالتالي :

Listing 3-3

HtmlUtil.java (re-refactored)

```

public static String renderPageWithSetupsAndTear downs(
    PageData pageData, boolean isSuite) throws Exception {
    if (isTestPage(pageData))
        includeSetupAndTeardownPages(pageData, isSuite);
    return pageData.getHtml();
}

```

حجم البلوك والمسافة الباردة

البلوك لل `if`, `else`, `while` يجب أن تكون بطول سطر واحد، لأن ذلك يبقى الدالة صغيرة، ويكون بمثابة

توثيق لما تفعله ما في البلاوك. وبالتالي لن تكون الدوال كافية للstrukture المترابطة nested.

المسافة البارزة قبل الدالة يجب ان لا تزيد عن مسافة واحد او اثنين لتسهيل القراءة والفهم.

مهمة واحدة

الدالة الطويلة 1-3 كانت تؤدي مهام متعددة ((انشاء بفر ، جلب الصفحات ، البحث وانشاء ال html)) ولكن الكود 3-3 لها وظيفة واحدة بسيطة (تضمين setup & teardowns صفحات الاعداد والتفكيك لصفحة الاختبار).

يجب ان تؤدي الدوال مهمة واحدة ويجب أن تؤديها جيدا ثم يجب أن تقوم بالعمل المطلوب فقط.

لا يزال الكود 3-3 يقوم بالوظائف :

١- تحدد هل الصفحة هي test page .
٢- اذا تحقق ذلك تقوم بتضمين ال setup &

teardowns

٣- ثم تقوم بعرض الصفحة لك.html

فهل الدالة لها ٣ وظائف ام وظيفة واحدة ؟

لاحظ ان الثلاث وظائف تقع ضمن اسم الدالة الذي وصفها التالي :

To RenderPageWithSetupsAndTeardowns

نقوم بالفحص هل الصفحة صحة اختبار اذا كان ذلك
نقوم بتضمين setup & teardowns في كلا الحالتين
عرض الصفحة في ال .html

اذا كانت الدالة تقوم ببعض الوظائف تحت مسمى هذه
الدالة نقول انها ذات وظيفة واحدة.

```
import java.util.*;  
  
public class GeneratePrimes  
{  
    /**  
     * @param maxValue is the generation limit.  
     */  
    public static int[] generatePrimes(int maxValue)  
    {  
        if (maxValue >= 2) // the only valid case  
        {  
            // declarations  
            int s = maxValue + 1; // size of array  
            boolean[] f = new boolean[s];  
            int i;
```

السبب اننا نحتاج لذلك لأجل تحليل المفهوم الأكبر الى
مجموعة خطوات لها مستوى واحد من التجزيد . الكود
if-2 له مستويين من التجزيد. إذا قمنا باستخراج ال
من الكود 3-3 في دالة

includeSetupsAndTeardownsIfTestPage

سيظل نفس مستوى التجزيد لكن مع تغيير في الكود.

لذلك إذا كان بالإمكان استخراج وظيفة من الدوال التي
لها أكثر من "وظيفة واحدة" تحت اسم ليس متعلق
بالدالة نقول أنها ذات أكثر من مستوى تجزيد.

أقسام في الدوال

```
// initialize array to true.  
for (i = 0; i < s; i++)  
    f[i] = true;  
  
// get rid of known non-primes  
f[0] = f[1] = false;  
  
// sieve  
int j;  
for (i = 2; i < Math.sqrt(s) + 1; i++)  
{  
    if (f[i]) // if i is uncrossed, cross its multiples.  
    {  
        for (j = 2 * i; j < s; j += i)  
            f[j] = false; // multiple is not prime  
    }  
}  
  
// how many primes are there?  
int count = 0;  
for (i = 0; i < s; i++)  
{  
    if (f[i])  
        count++; // bump count.  
}  
  
int[] primes = new int[count];  
  
// move the primes into the result  
for (i = 0, j = 0; i < s; i++)  
{  
    if (f[i]) // if prime  
        primes[j++] = i;  
}  
  
return primes; // return the primes  
}  
else // maxValue < 2  
    return new int[0]; // return null array if bad input.  
}
```

الكود من الفصل ٤ واضح أن الدالة 

generatPrimes

قسمت لل initialization, declaration بالتالي و واضح أنها لا تقوم بمهمة واحدة .

مستوى واحد من التجرييد لكل دالة

للتأكد من ان الدوال تقوم "وظيفة واحدة" يلزمها ان
نتاكد من ان الدالة بنفس مستوى التجريد
في المثال 3-1

Very high level of abstraction `getHtml()`
intermediate level `String pagePathName =
PathParser.render(pagePath);`
Low level `.append("\n")`

** يقصد بهذه مستويات التعقيد
لاحظ ان الخلط بين المستويات يمثله " نظرية النافذة
المكسورة" تراكمي وأمر مربك يسبب مزيد من الخلط
مع التفاصيل الأساسية.

جعل الكود متسلسل القراءة
كما لو كان مقال خطواته متسلسلة كما لو كان مجموعة
فقرات TO كل واحد منها يصف مستوى من التجريد
ويشير للمستوى التالي الذي يليه بالتجريد.

To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.

To include the setups, we include the suite setup if this is a suite, then we include the regular setup.

To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.

To search the parent...

برغم صعوبة الحيلة هذه إلا أن تعلمها مهم لإبقاء الدوال
قصيرة والتأكد أنها تقوم بعمل واحد.

ال Switch

من الصعب جعل جملة ال **Switch** صغيرة حيث يفضل ان تكون بلوك واحد او دالة واحدة تفعل شيء واحد فقط، لأن عادة تستخدم ال **switch** لعمل N من الأشياء. لا يمكننا ان ندع استخدام **switch** ولكن يمكننا ان نتأكد من ان كل **switch** بداخل كلاس من مستوى .**polymorphism** ولا تتكرر. تستخدم مع ال

الدالة التالية تقوم بوظيفة واحدة :

Listing 3-4

Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
        case COMMISSIONED:
            return calculateCommissionedPay(e);
        case HOURLY:
            return calculateHourlyPay(e);
        case SALARIED:
            return calculateSalariedPay(e);
        default:
            throw new InvalidEmployeeType(e.type);
    }
}
```

ال코드 السابق فيه عدد من المشاكل ، أولاً أنها دالة كبيرة ، وإذا أضفنا موظفين آخرين ستصبح أكبر. ثانياً واضح أن للدالة أكثر من مهمة. ثالثاً أنها تنتهي مبدأ **Single Responsibility Principle**. لأنه يجب تغيير الكود عند إضافة أنواع جديدة. والمشكلة الأسوأ أنه سيكون

هناك عدد من الدوال لها نفس التركيب مثلا

`isPayday(Employee e, Date date)`

or

`deliverPay(Employee e, Money pay)`

الصياغة الجيدة في الكود 3-5 بواسطة إخفاء

استخدام ال `Abstract Factory` عبر ال `Switch` ولا تجعل مرئية.

Listing 3-5

Employee and Factory

```
public abstract class Employee {  
    public abstract boolean isPayday();  
    public abstract Money calculatePay();  
    public abstract void deliverPay(Money pay);  
}  
-----  
public interface EmployeeFactory {  
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;  
}  
-----  
public class EmployeeFactoryImpl implements EmployeeFactory {  
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {  
        switch (r.type) {  
            case COMMISSIONED:  
                return new CommissionedEmployee(r);  
            case HOURLY:  
                return new HourlyEmployee(r);  
            case SALARIED:  
                return new SalariedEmployee(r);  
            default:  
                throw new InvalidEmployeeType(r.type);  
        }  
    }  
}
```

قاعدتي العامة هي أنه يمكن التسامح مع استخدام ال `polymorphism` مرة واحدة في حالة ال `Switch` وأن تكون مخفية بالوراثة وبالتالي لا يمكن لباقي النظام رؤيتها.

استخدام أسماء معبرة

في القائمة 3-7 قمت بتغيير الاسم من `testableHtml` للاسم `SetupTeardownIncluder.render` وتسمية الدوال ال `private` للأسماء التالية :
`isTestable` `includeSetupAndTeardownPages`

وتذكر "أن الكود سيكون `clean` عندما يكون ما تراه هو ما تتوقعه ". ولتحقيق هذا المبدأ لا بد من اختيار أسماء جيدة للوظائف الصغيرة التي لها وظيفة واحدة، فكلما كانت أصغر كان من السهل اعطاء إسم لها . وتذكر أن الإسم وإن كان طويلاً أفضل من إسم قصير غامض بالإضافة للتعليقات . تحتاج لبعض الوقت لتحديد إسم الدوال وأن تجرب عدد من الأسماء وقراءة الكود بمختلف الأسماء حتى تحدد الإسم الأكثر وصفية بقدر ما يمكنك فعله . كما يجب أن تكون الأسماء المستخدمة متسقة تستخدم نفس الاسم مع الدالة الذي تم استخدامه مع ال `.module`

بارامترات الدوال

العدد المثالي للبارامترات المرسلة هو عدم إرسال أي بارامترات يليه واحد ثم اثنين يجب تجنب استخدام ثلاثة بارامترات ، أما أكثر من ثلاثة لابد من سبب مقنع ولا تستخدمه باي حال . تعدد البارامترات يجعل عملية

الاختبار `test` صعبة ، في حالة عدم وجود اي بارامترات سيكون من السهل عمل الاختبار ويزداد التعقيد مع تزايد البارامترات. من العادة ان البارامترات تستخدم كمدخل للدالة والقيمة الراجعة هي مخرج الدالة ، فمما لو تم استخدام بارامترات كقيمة مخرجة من الدالة ، ذلك سيصعب فهم الدالة. في المثال استخدمنا بارامتر واحد وهذا جيد والافضل عدم تواجد اي بارامترات

`SetupTeardownIncluder.render(pageData)`

واضح ان البارامتر `pageData` سيتم عليه عملية التحويل `.render`.

صور شائعة للبارامتر الواحد قد يمرر بارامتر واحد للدالة بسبب ان هذا ما يتوقع من اسم الدالة مثل

`boolean fileExists("MyFile"),`

`InputStream fileOpen("MyFile")`

كذلك احد الامثلة الاقل شيوعا هو استخدام ال `event` الذي يستقبل بارامتر ولا يعيد مخرجات للتعبير ان التغيير يحدث على النظام

`void passwordAttemptFailedNtimes(int attempts)`

يجب ان يكون واضحا للقارئ ضمن سياق متسلق. كما يجب ان تتجنب الدوال ببارامتر واحد والتي لا تتبع

هذه النماذج مثل

`void includeSetupPageInto(StringBuffer pageText)`

استخدم قيمة راجعة بدلاً من التعديل على القيم لانه مربك. اذا كانت عملية التحويل تتم على البارامتر المدخل فإن ناتج الدالة يجب ان يتم ارجاعه في الـ return للدالة.

`StringBuffer transform(StringBuffer in)` 

هذا الشكل أفضل من الشكل التالي حتى لو كان تطوير الدالة التالية أسهل

`void transform(StringBuffer out)`

استخدام الـ Flag كبارامتر

يعتبر استخدام الـ flag ممارسة غير صحيحة وشكل غير ملائم، لأن الدوال التي تستلم flag يجب ان تكون لها أكثر من وظيفة .. في حالة ان المؤشر صحيح ووظيفة أخرى إذا المؤشر غير ذلك.

في الكود 3-3 لم يكن هناك خيار اخر فقد أردت تقليل خطوات اعادة صياغة الكود وبالتالي يكون استدعاء الدالة `render(true)` وهذا امر مربك للقارئ. رغم ان تسمية بارامتر الدالة `render(boolean isSuite)` سيساعد بعض الشئ لكن يجب تقسيم الدالة لذاتين `renderForSuite()` & `renderForSingleTest()`

الدوال ببارامترین

تعد اصعب من استخدام دالة ببارامتر واحد مثال:

`writeField(name)`

`writeField(outputStream, name)`

لأنه يحتاج لوقت قصير حتى تتجاهل البارامتر الأول،
في حين أن الأشياء التي نقوم بتجاهلها تكثر فيها
الأخطاء.

في بعض الحالات يكون من الأنسب وجود بارامترین
حتى في بعض الحالات الشائعة مثل

`Point p = new Point(0,0);`

لكن التالي قد يحدث ليس بترتيب البارامترین

`assertEquals(expected, actual)`

لأنه لا يوجد تماسك ولا ترتيب طبيعي في البارامترات
المرسلة.

بالتأكيد سوف تكتب دوال ببارامترین ولكن ستدرك
ان لهذا الاستخدام تكلفة لذا ينصح بان تحاول ان
تستخدم الادوات المتوفرة التي تحولها لدالة ببارامتر
واحد.مثال في الدالة `writeField` يمكن جعلها عضو
من الكلاس `OutputStream.writeField(name)`
او ان تجعل ال `OutputStream` عضوا في الكلاس
الذي توجد فيه هذه الدالة وبالتالي لن تحتاج لتمرير هذا

الكلas . او يمكنك عمل كلاس FieldWriter والذي يستقبل ال outputStream في دالة ال write . و توجد في الكلas الدالة constructor .

الدوال بثلاثة بارامترات

يجب ان تفك بعانياة قبل أن تسمح بالدوال بثلاثة بارامترات بسبب المشاكل الكثيرة والتي ستواجهك .

: assertEquals overload للدالة

assertEquals(message, expected, actual)

بالنظر للدالة نظن أن المتغير message هو ال .message ثم إننا نتجاهل المتغير expected

الكائن كبارامتر

اذا كان لابد من ثلاثة من البارامترات قم بعمل هذه البارامترات في كلاس من نوع هذه المتغيرات كالتالي:

```
Circle makeCircle(double x, double y, double radius);  
Circle makeCircle(Point center, double radius);
```

بدلا من y , x لابد من ان يكون هناك اسم بدليلا .

القوائم كبارامترات

تمرير عدد متغير من البارامترات يكافي تمرير بارامتر واحد من نوع List .لذلك لكي نحوال اي دالة سنستخدم

ال object كما فعلنا سابقا.

`String.format("%s worked %.2f hours.", name, hours);`

يصبح :

`public String format(String format, Object... args)`

اختيار الأسماء

عند اختيار إسم لدالة ب بارامتر واحد يجب ان يشكل زوج من الاسم / الفعل كالتالي `write(name)` ذلك يفهم بان حدث الدالة سيكون على الاسم .

كما يجب ان يكون هناك ترابط فبدلا عن اسم الدالة السابق استخدام `writeField(name)` يخبر عن نوع المتغير `name` انه `field` . هذا يطلق عليه keyword `name` في هذا الشكل نقوم بتسمية المتغيرات كاسم `form` الدالة . يمكن تعديل الدالة كالتالي `assertEquals` `assertExpectedEqualsActual(expected, actual)`.

هذا سوف يحل مشكلة تذكر البارامترات (سوف نتذكر `act` عند الاستدعاء ان ال `ex` هو البارامتر الاول وال `actual` هو البارامتر الثاني).

بدون تأثيرات جانبية

اي عمل يخالف مسمى الدالة يعد كذب ليس ضمن ما تدعى الدالة لعمله . وقد يؤدي لتغييرات غير متوقعة ما

يؤدي لمشاكل في الكود.

انظر الدالة التي قد تبدو انها غير ضارة و تستخد
خوارزمية قياسية لمطابقة اسم المستخدم بكلمة المرور
ولك لها اثار جانبية ماهي ؟

Listing 3-6

UserValidator.java

```
public class UserValidator {  
    private Cryptographer cryptographer;  
  
    public boolean checkPassword(String userName, String password) {  
        User user = UserGateway.findByName(userName);  
        if (user != User.NULL) {  
            String codedPhrase = user.getPhraseEncodedByPassword();  
            String phrase = cryptographer.decrypt(codedPhrase, password);  
            if ("Valid Password".equals(phrase)) {  
                Session.initialize();  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

لاحظ وجود ال `Session.initialize()` برغم أن إسم
الدالة هو فحص كلمة السر لا يشير لعملية تهيئة الجلسة،
لذا عندما يتم استدعاء الدالة فان بيانات الجلسة
الحالية ستتحذف اثناء التحقق من المستخدم.
هذا يسبب coupling ارتباط غير مرغوب. في مثل هذه
الحالة إسم الدالة يجب أن يكون:
checkPasswordAndInitializeSession()
رغم أن ذلك ينتهك وظيفة واحدة لكل دالة.

البارامترات كمخرجات

تفسر البارامترات على انها مدخلات في اغلب الحالات .
كان وجود البارامترات كمخرج امر ضروري قبل وجود
ال OOP لكن مع ظهورها حلت المشكلة باستخدام ال
كباقيه فبدلا عن الاستدعاء this

```
public void appendFooter(StringBuffer report)  
report.appendFooter();
```

بشكل عام تجنب إستخدام البارامتر كمخرج تغييره الدالة
اذا كانت الدالة تغير حالة احد المتغيرات فاجعلها تغير
حالة ال object.

فصل الاستعلامات

الدالة يجب أن تقوم بأحد أمرين إما أنها تغير حالة
متغير (change state of object) او أنها ترجع
. (return some information) return
ولكن ليس كلا الأمرين لأن ذلك يسبب الارتباك.

```
public boolean set(String attribute, String value);
```

تعلم سابقا بدور دوال ال set والقيمة الراجعة تحدد اذا
نحوت العملية ام فشلت (بسبب عدم وجود بيانات
مرسلة).

ستكون عملية الاستدعاء مربكة، هل عملية الفحص
للتأكد انه تم اعطاء المتغير القيمة سابقا() هل القيمة

هي 'unclebob'؟ أم الفحص هل تم بنجاح تخصيص القيمة للمتغير؟ لن يكون واضح هل الـ `set` هي فعل أم صفة. يقصد المؤلف استخدام `set` كفعل لذلك ممكن أن نحل المشكلة عبر تغيير الإسم والحل الأفضل فصل عملية التحقق كالتالي:

```
if (attributeExists("username")) {  
    setAttribute("username", "unclebob");  
    ...  
}
```

تفضيل استخدام الاستثناءات بدلاً عن ارجاع خطأ لا نريد أن تظهر الأخطاء في أكوادنا فنحتاج للاستثناءات

لكن المشكلة في الكود وجود تراكيب متداخلة من `if`

```
if (deletePage(page) == E_OK) {  
    if (registry.deleteReference(page.name) == E_OK) {  
        if (configKeys.deleteKey(page.name.makeKey()) == E_OK) {  
            logger.log("page deleted");  
        } else {  
            logger.log("configKey not deleted");  
        }  
    } else {  
        logger.log("deleteReference from registry failed");  
    }  
} else {  
    logger.log("delete failed");  
    return E_ERROR;  
}
```

بدلاً عنها يمكن استخدام التعبير بالصورة التالية:

```

try {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
    logger.log(e.getMessage());
}

```

فصل الـ try/catch

شكل الكود مع try /catch قبيح يسبب الارتباك بين معالجة الأخطاء والكود الأساسي لذلك من الأفضل فصل بلوك ال try /catch لدوال خارجية.
بالشكل التالي توجد دالة delete خصيصا للتعامل مع الاستثناءات كالتالي

```

public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    }
    catch (Exception e) {
        logError(e);
    }
}

private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}

```

معالجة الاستثناء تعالج شيء واحد

الدالة التي تعالج الأخطاء يجب ان لا تفعل شيء آخر.
يجب ان تكون try في بداية الدالة ويجب ان يوجد

شيء بعد ال catch/finally تستخدم في الكود class /enum والتي تحوي انواع الاخطاء الممكنة.

```
public enum Error {  
    OK,  
    INVALID,  
    NO_SUCH,  
    LOCKED,  
    OUT_OF_RESOURCES,  
    WAITING_FOR_EVENT;  
}
```

لا تكرر

اعد النظر في الكود 1-3 ستجدد الكود تكرر 4 مرات

[SetUp](#), [SuiteSetUp](#), [TearDown](#), [SuiteTearDown](#)

وتم اخفاء هذا التكرار في 7-3 . التكرار مشكلة سببها في تضاعف كبير لل kod .

البرمجة المنظمة

يتبع بعض البرمجيين Edsger Dijkstra's الخاصة بتنظيم البرمجة rules

كل دالة يجب ان يكون لها مدخل و مخرج وحيدان.

وهذا يعني عدم وجود break او continue او goto . يمكن التغاضي في حالة الدوال الصغيرة لذلك من المهم

ان يجعل دوالك صغيرة ثم لن تسبب هذه التعليمات اي مشاكل في الكود ويجب ان تتجنب استخدام ال goto لأن الحاجة لها تظهر مع الدوال الكبيرة.

كتابة الكود مثل كتابة المقال في كل مرة تقوم بتحسين الكتابة للأفضل عندما اكتب كود تكون الدوال طويلة ثم أقوم بتحسين الكود وتقسيم الدوال وعمل الاختبار عليها.

الاستنتاج

لكل نظام مجموعة مصطلحات خاصة به نحتاج في البرمجة لوضع التسمية يعطى للكلاس اسم وللدوال فعل. تذكر ان هدفك هو سرد وظائف النظام بطريقة قابلة للقراءة.

SetupTeardownIncluder

Listing 3-7

SetupTeardownIncluder.java

```
package fitness.html;

import fitness.responders.run.SuiteResponder;
import fitness.wiki.*;

public class SetupTeardownIncluder {
    private PageData pageData;
    private boolean isSuite;
    private WikiPage testPage;
    private StringBuffer newPageContent;
    private PageCrawler pageCrawler;

    public static String render(PageData pageData) throws Exception {
        return render(pageData, false);
    }

    public static String render(PageData pageData, boolean isSuite)
```

```
    throws Exception {
        return new SetupTeardownIncluder(pageData).render(isSuite);
    }

    private SetupTeardownIncluder(PageData pageData) {
        this.pageData = pageData;
        testPage = pageData.getWikiPage();
        pageCrawler = testPage.getPageCrawler();
        newPasswordContent = new StringBuffer();
    }

    private String render(boolean isSuite) throws Exception {
        this.isSuite = isSuite;
        if (isTestPage())
            includeSetupAndTeardownPages();
        return pageData.getHtml();
    }

    private boolean isTestPage() throws Exception {
        return pageData.hasAttribute("Test");
    }

    private void includeSetupAndTeardownPages() throws Exception {
        includeSetupPages();
        includePageContent();
        includeTeardownPages();
        updatePageContent();
    }

    private void includeSetupPages() throws Exception {
        if (isSuite)
            includeSuiteSetupPage();
        includeSetupPage();
    }

    private void includeSuiteSetupPage() throws Exception {
        include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
    }

    private void includeSetupPage() throws Exception {
        include("SetUp", "-setup");
    }

    private void includePageContent() throws Exception {
        newPasswordContent.append(pageData.getContent());
    }

    private void includeTeardownPages() throws Exception {
        includeTeardownPage();
        if (isSuite)
            includeSuiteTeardownPage();
    }

    private void includeTeardownPage() throws Exception {
        include("TearDown", "-teardown");
    }

    private void includeSuiteTeardownPage() throws Exception {
        include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
    }

    private void updatePageContent() throws Exception {
        pageData.setContent(newPasswordContent.toString());
    }

    private void include(String pageName, String arg) throws Exception {
        WikiPage inheritedPage = findInheritedPage(pageName);
        if (inheritedPage != null) {
            String pagePathName = getPathNameForPage(inheritedPage);
            newPasswordContent.append(inheritedPage.getContent());
        }
    }
}
```

```
        string pagePathName = getFullNameForPage(inheritedPage);
        buildIncludeDirective(pagePathName, arg);
    }

    private WikiPage findInheritedPage(String pageName) throws Exception {
        return PageCrawlerImpl.getInheritedPage(pageName, testPage);
    }

    private String getPathNameForPage(WikiPage page) throws Exception {
        WikiPagePath pagePath = pageCrawler.getFullPath(page);
        return PathParser.render(pagePath);
    }

    private void buildIncludeDirective(String pagePathName, String arg) {
        newPageContent
            .append("\n!include ")
            .append(arg)
            .append(" .")
            .append(pagePathName)
            .append("\n");
    }
}
```

