Coding Style and Refactoring

Jianyong Wang(王建勇)
Department of Computer Science and Technology
Tsinghua University, Beijing, China

Manifesto for Software Craftmanship

Manifesto for Software Craftsmanship

Raising the bar.

As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

Not only working software,
but also **well-crafted software**

Not only responding to change,
but also **steadily adding value**

Not only individuals and interactions,
but also **a community of professionals**

Not only customer collaboration,
but also **productive partnerships**

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

Sign the Manifesto

Manifesto for Software Craftsmanship

提高标准

作为有理想的软件工匠，我们一直身体力行，提升专业软件开发的标准，并帮助他人学习此工艺。通过这些工作，我们建立了如下价值观：

不仅要让软件工作，
更要**精益求精**

不仅要响应变化，
更要**稳步增加价值**

不仅要有个体与交互，
更要**形成专业人员的社区**

不仅要与客户合作，
更要**建立卓有成效的伙伴关系**

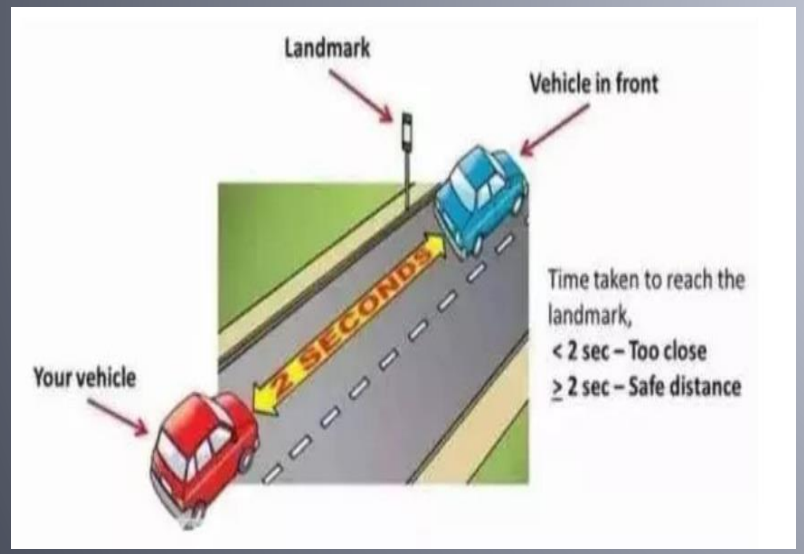
也就是说，左项固然值得追求，右项同样不可或缺。

签署宣言

Coding Style

- **Defensive programming**
- **Layout and presentation**
- **Naming convention**
- **Self-documenting code**
- **Code comments**

Defensive programming



害人之心不可有
防人之心不可无

The Murphy's Law

- "If anything can go wrong, it will" (怕什么来什么)
- "If there is more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way."
- ***If your code can be used incorrectly, it will be.***

The lessons of Ariane 5 accident

All of these conditions have to be satisfied at the same time for the Ariane 5 accident to happen:

- Araine 4 programmers decided to remove the overflow protection for the float to integer conversion, to cut memory footprint. This worked well in Araine 4.
- Arian5 has increased float to 64, which can overflow the conversion. Programmers left a comment about the the possibility of overflow, but the conversion code is designed to be used before the take-off, and shouldn't overflow before the take-off in theory. So no overflow protection in Ariane 5 either.
- (other) programmers decided to let the above code segment run for 40 seconds (to save some reset time) after the scheduled take-off. These programmers didn't see the comments left by the previous programmer.

Defensive Programming

```
int low_quality_programming(char *input){
  char str[1000+1]; // one more for the null character
  strcpy(str, input); // copy input
  ...
}
```

The function will crash when the input is over 1000 characters. However, many mainstream programmers may not feel that this is a problem because “Surely no one will enter that long of an input!”

```
int high_quality_programming(char *input){
  char str[1000+1]; // one more for the null character
  strncpy(str, input, 1000); // copy input, only copy a maximum length of 1000 characters
  str[1000] = '\0'; // add terminating null character
  ...
}
```

Defensive Programming

Assume nothing. Unwritten assumptions continually cause faults, particularly as code grows.

Garbage in, garbage out

Defensive protection

Garbage in, nothing out

Garbage in, error message out

No garbage allowed in

Defensive Programming

- Encourage programmers to include **as many checks as possible**
  - Check the values of all data from external sources
  - Check the values of all input parameters
  - Decide how to handle bad inputs

For and Against

AGAINST

- It eats into the efficiency of your code
- Each defensive practice requires some extra work

FOR

- Save you literally hours of debugging
- Working code that runs properly, but ever-so-slightly slower, is far superior to code that works most of the time but occasionally collapses in a shower of brightly colored sparks.

Coding Style

- Defensive programming
- Layout and presentation
- Naming convention
- Self-documenting code
- Code comments

The layout and presentation of source code

Know Your Audience

- The compiler
- Ourselves
- **Others**
  - **Most important, yet least considered**

"Understand the real audience for your source code: other programmers. Write for their benefit"

Coding style: bad case vs. good case

Bad case

```
if len(buffer) < (LOG_HEAD_LEN + LOG_TIME_LEN):
    # no enough data to get log_head and log_time
    return (False, None, buffer)
logHeader, buffer = _logHeadRead(buffer)
if logHeader == None:
    # fail to read logHead from buffer
    return (True, None, buffer)
if logHeader[LOG_HEAD_POS_COMPRESS_LEN] != 0:
    # some code is omitted
offset = LOG_HEAD_LEN + LOG_TIME_LEN + logHeader[LOG_HEAD_POS_UNCOMPRESS_LEN]
if len(buffer) < offset:
    # no enough data, wait for the next time
    return (False, None, buffer)
recordStr = buffer[(LOG_HEAD_LEN + LOG_TIME_LEN):offset]
buffer = buffer[offset:]
```

Good case

```
if len(buffer) < (LOG_HEAD_LEN + LOG_TIME_LEN):
    # no enough data to get log_head and log_time
    return (False, None, buffer)

# read loghead
logHeader, buffer = _logHeadRead(buffer)
if logHeader == None:
    # fail to read logHead from buffer
    return (True, None, buffer)

# check whether it is compressed record
if logHeader[LOG_HEAD_POS_COMPRESS_LEN] != 0:
    # some code is omitted

# check whether record is completely in the buffer
offset = LOG_HEAD_LEN + LOG_TIME_LEN + logHeader[LOG_HEAD_POS_UNCOMPRESS_LEN]
if len(buffer) < offset:
    # no enough data, wait for the next time
    return (False, None, buffer)

# get record out of the buffer
recordStr = buffer[(LOG_HEAD_LEN + LOG_TIME_LEN):offset]
buffer = buffer[offset:]
```

The Big Deal

- Presentation makes a real impact on the quality of code.
- Programmers read meaning into code based on its layout.
  →Good presentation: Illuminate and support your code's structure, helping reader understand what's going on
  →Bad presentation: Confuse, mislead, and hide the code's intent.
- Bad formatting not only makes code harder to follow; it may actually hide bugs from you.

Good Presentation

- Consistent
  - Keep the style consistent across the project
    - Indentation, braces, brackets….
- Conventional
  - To adopt one of the major styles currently in use in the industry rather than invent your own rules.
- Concise
  - Easy to follow and pick up

```
int k_and_r() {
    int  a =0, b=0;
    while (a != 10) {
        b++;
        a++;
    }
    return b;
}
```

```
int k_and_r()
{
    int  a =0, b=0;
    while (a != 10)
    {
        b++;
        a++;
    }
    return b;
}
```

```
int k_and_r()
    {
     int  a =0, b=0;
     while (a != 10)
        {
        b++;
        a++;
        }
     return b;
     }
```

```
 int k_and_r()
{
 int  a =0, b=0;
 while (a != 10) {
        b++;
        a++;
        }
 return b;
}
```

- control structure
- Individual statements
- Comments
- Routines
- Classes
- .....

Common Coding Standards

- Indian Hill
  - Indian Hill Recommended C Style and Coding Standards (Indian Hill AT&T Bell labs)
- GNU
  - GNU's Coding Standards influence most of the commonly used open source or free software (www.gnu.org)
- MISRA
  - UK's Motor Industry Software Reliability Association (MISRA) defined the well-known set of standards for writing safety critical embedded software in C/C++.
  - It consists of 127 guidelines, and a number of tools exist to validate your code against them.
- .....

Pick a single good coding style, and stick to it!

Java Code Convention

C++ Code Convention

In a Nutshell

GOOD PROGRAMMERS…

- Know how code layout impacts readability and strive for the clearest code possible
- Will adopt a house style even if it contradicts their personal preferences

BAD PROGRAMMERS…

- Are close-minded and opinionated – *My view is the right one*
- Have no consistent personal coding style
- Trample over others' code in their own style

Coding Style

- Defensive programming
- Layout and presentation
- Naming convention
- Self-documenting code
- Code comments

Naming

Giving meaningful things meaningful names

The Power of a Name

- People naturally read the meaning out of something based on its name

- A name describes
  - Identity
  - Behavior
    - Roles, relationships, and interactions
  - Recognition
    - Elevate something from ethereal concept to well-defined reality

What do we name

- Variables
- Constants
- Functions
- Types (classes, enums, structs, typedefs)
- Modules
- C++ namespaces and Java packages
- Macros
- Sub-systems
- Source files

How do we name

- Descriptive
- Technically correct
  - Whitespace, underscore, Unicode, ASCII
- Idiomatic
  - The common naming convention for specific language
- Appropriate
  - Length: favor clarity over brevity
  - Tone
    - Foo, bar,… never for production code

The Nuts and Bolts

- Naming Variables
  - Noun and "noun-ized" verb
    - Object, numerical, boolean
    - E.g. ok_button, count, widget_length
  - Local vs. global variables
    - e.g. C# : camelCase vs PascalCase
  - Type names vs. variable names
    - E.g. Window vs window
- Naming Functions
  - Verb
    - E.g. countApples ()
  - From the viewpoint of the use, hiding all the internal implementation

The Nuts and Bolts

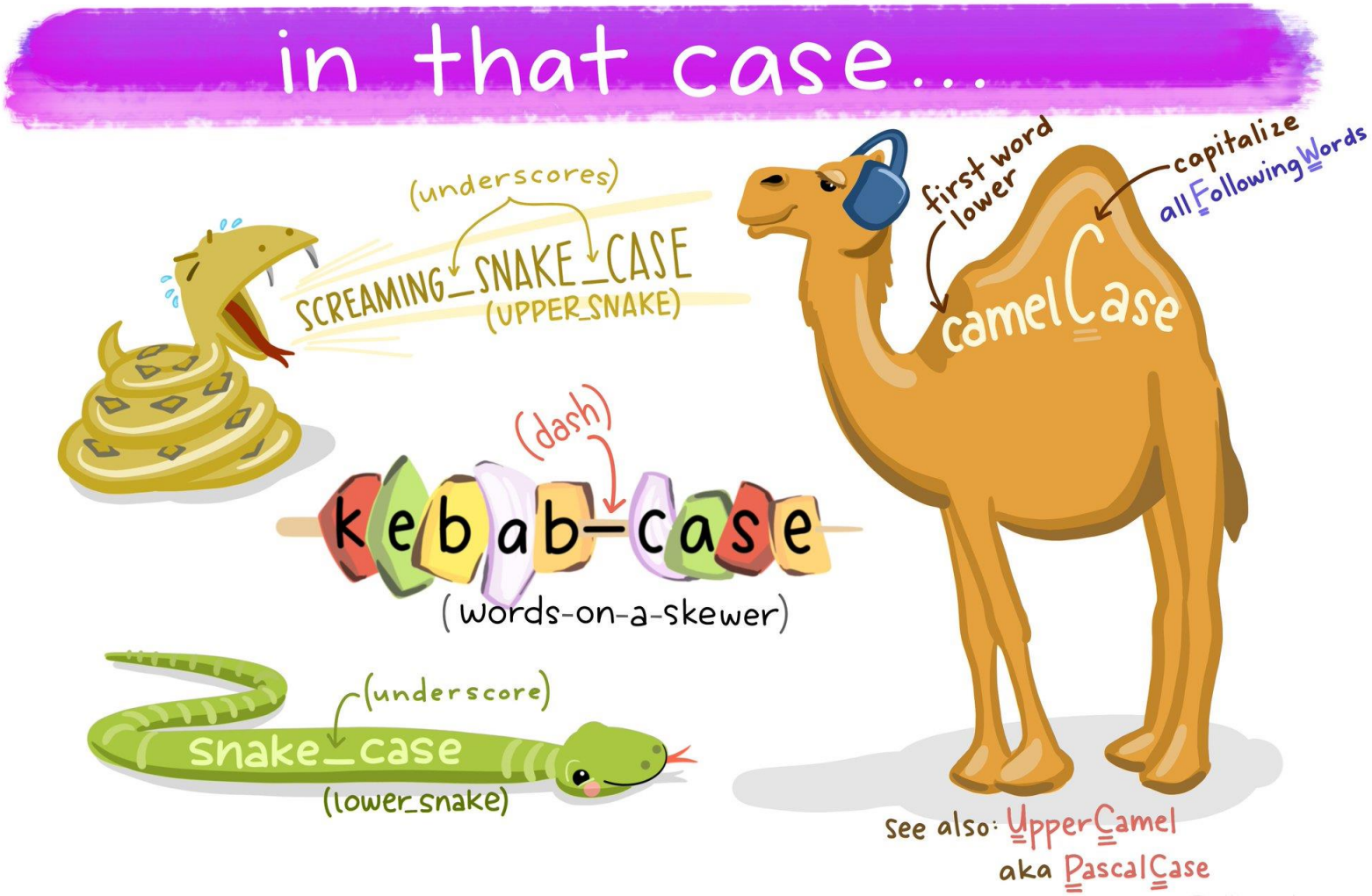
- Naming types
  - Typedefs, classes
  - Different purpose
    - E.g. interface classes, “Serializable”, “IPrintable”
  - **Avoid redundant words in type names**
- Naming Namespace (packages)
  - C++ and C# namespaces and Java packages are like grouping mechanisms, to avoid name collisions
  - **Naming scheme**
    - E. g. Java hierarchy of package names, nested like Internet domain names
    - ***Give namespace and packages names that reflect the logical relationships of their contents***

General DON'Ts

- **Cryptic**
  - Acronyms and abbreviations can appear quite random, and single letter names are far too magical.
- **Verbose**
  - Never, the_number _of_apples_before_I_started_eating
- **Inaccurate or misleading**
  - As obvious as it seems, make your names accurate.
- **Ambiguous or vague**
  - Don't use a name that could be interpreted in several ways.
  - Don't use a hopelessly vague name like data or value unless it's perfectly clear what it represents.
  - Don't differentiate names by capitalization or by changes of a single character. Be wary of names that sound similar.
  - Don't gratuitously create local variables with the same name as something in an outer scope.

```
void checkForContinue(bool weShouldContinue)
{
      if (weShouldContinue)   abort ();
}
```

Variable naming convention: team picks one and stick to it



In a Nutshell

GOOD PROGRAMMERS…

- **Realize the importance of names and treat them with respect**
- Think about naming and choose appropriate names for everything they create
- Hold name forces in balance: **name length, clarity, context,** and so on
- **Keep a view of the bigger picture, so their names hold together across a project (or projects)**

BAD PROGRAMMERS…

- Care little for the clarity of their code
- Produce write-once code that is quick to write and poorly thought out
- Ignore the language's natural idioms
- Are inconsistent in naming
- Don' think holistically, failing to consider how their piece of code fits into the whole

Coding Style

- Defensive programming
- Layout and presentation
- Naming convention
- Self-documenting code
- Code comments

Self-Documenting Code

Code Documentation

- Common wisdom
  - Tons of documents *about* the code
  - Tons of documents *in* the code
- The tedious work
  - A lot of extra work to write and to read
  - Separate documents may be out of sync with code changes
  - Hard to locate information and to manage
  - Important information in separate documents can easily be missed

Self-Documenting Code

- The only document that describes your code completely and correctly is the code itself
- Write your code to be read, By humans, Easily. Compilers will be able to cope.

```
int fval (int i) {
    int ret=2;
    for (int n1=1, n2=1, i2=i-3; i2>=0; --i2) {
        n1=n2; n2=ret;  ret=n2+n1;
    }
    return (i<2)? 1:ret;
}
```

The FACT

Even the smallest, most beautiful functions will need later maintenance.

```
int fibonacci (int position) {
    for (position <2) {
        return 1;
    }

    int previousButOne   = 1;
    int previous         = 1;
    int answer           = 2;

    for (int n=2;  n<position;  ++n) {
        previousButOne       = previous;
        previous             = answer;
        answer               = previous + previousButOne;
    }

    return answer;
}
```

Techniques

- Write simple code with good presentation
  - Make the "normal" path through your code obvious. Your *if-then-else* constructs should be ordered consistently.
    - Always place the "normal" case before the "error" case, or vice versa
  - Avoid too many nested statements.
    - Balance between exit points (Single Input, Single Output) and nesting levels.
  - Be wary of optimizing code so that it's no longer a clear expression of a basic algorithm.
    - *Never* optimize code unless you've proved that it is a bottleneck to acceptable program function.

```
int fibonacci (int position) {
    for (position <2) {
        return 1;
    }

    int previousButOne   = 1;
    int previous         = 1;
    int answer                   = 2;

    for (int n=2;  n<position;  ++n) {
        previousButOne       = previo
        previous         = answer;
        answer                   = previo
    }

    return answer;
}
```

Multiple exits,
Less nesting levels

```
int fibonacci (int position) {
    int answer = 1;
    if (position >= 2) {

        int previousButOne  = 1;
        int previous         = 1;

        for (int n=2;  n<position;  ++n) {
            previousButOne  = previous;
            previous         = answer;
            answer           = previous + previousButOne;
        }
    }
    return answer;
}
```

Single exit,
Deep nesting levels

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the

Techniques

- Decompose into Atomic Functions
  - One function, one action.
  - Minimize any surprising side effects
  - Keep it short
- Emphasize important code
  - Make sure that all important code stands out and is easy to read.
- Group related information
  - Use language features to make explicit grouping of related information.

Techniques

- Choose meaningful names
- Choose Descriptive Types
- Name constants
  - Avoid magic numbers. Use well-named constants instead.

```
typedef
   enum {
        NO_ERROR        = 0,    /* request valid and operation performed      */
        NO_ACTION       = 1,    /* status of system unaffected by request     */
        NOT_AVAILABLE   = 2,    /* resource required by request unavailable   */
        INVALID_PARAM   = 3,    /* invalid parameter specified in request     */
        INVALID_CONFIG  = 4,    /* parameter incompatible with configuration  */
        INVALID_MODE    = 5,    /* request incompatible with current mode     */
        TIMED_OUT       = 6     /* time-out tied up with request has expired  */
   } RETURN_CODE_TYPE;
```

```java
package Business;

import control.*;

public class Login3
{
    public int login32(String id,String pass)
    {
        Login4 cc = new Login4();
        if(cc.login43(id) == null)
            return 1;   //用户名不存在
        else
            {
                if(pass.compareTo(cc.login43(id).getpass()) == 0)//用户名匹配
                    {
                        if(cc.login43(id).getau() == 1)
                            return 3;   //管理员
                        else
                            return 4;   //普通用户
                    }
                else
                    return 2;   //密码不正确
            }
    }
}
```

```java
package control;

import frames.*;
import java.sql.*;

import userinterface.*;
import baseclass.LogInfo;

public class Login4
{
    public LogInfo login43(String id)
    {
        try{
            String sql = "select * from user where userID =\"" + id +"\"";

            ResultSet rs = DatabaseConnection.getDb().getDbOper().query(sql);

            if(rs.next())
```

```java
package userinterface;

import Business.*;
import baseclass.*;

public class Login2
    public String id,pass;
    public int login21(LogInfo log){
        id = log.getuser();
        pass = log.getpass();
        if(id.compareTo("") == 0)
            return 1;   //还未输入用户名
        else if(pass.compareTo("") == 0)
            return 2;   //还未输入密码
        else
        {
            Login3 bb = new Login3();
            if(bb.login32(id, pass) == 1)
                return 3;   //用户名不存在
            else if(bb.login32(id, pass) == 2)
                return 4;   //密码不正确
            else if(bb.login32(id, pass) == 3)
                return 5;   //用户名匹配,管理员
            else
                return 6;   //用户名匹配,普通用户
        }
    }
}
```

Magic Number

In a Nutshell

GOOD PROGRAMMERS…

- Seek to write clear, self-documenting code
- Try to write the least amount of documentation necessary
- Think about the needs of programmers who will maintain their code

BAD PROGRAMMERS…

- Try to avoid writing any documentation
- Don’t care about updating documentation
- Think, “if it was hard for me to write, it should be hard for anyone to understand. ”
- Are proud that they write unfathomable spaghetti （混乱复杂的程序）

Coding Style

- Defensive programming
- Layout and presentation
- Naming convention
- Self-documenting code
- Code comments

Code Comments

// I dedicate all
// have to suppor

//
// Dear maintainer:
//
// no comments fo
// it was hard to
// so it should b
//
// total_hours_wasted_he
//

// TODO: Fix th

// I am not respons
// They made me wri

// hack for ie br

Darlene, who will
dog once it gets

later

touch.

.==.
//^`\\
(__/)/^ ^^\\
/6 6\ ^^ ^ \\
(..)\^ ^ ^ \\
v""v |/\^ ^ ^\\
`~~` \ \/\^ ^\\

GONS

a browser)

《神秘的程序员们》-40 最佳注释

http://c.xiqiao.info

0

```
// 写这段代码的时候
// 只有上帝和我知道它是干嘛的
// 现在，只有上帝知道
```

1

```
// 虽然我也不想这样做，
// 但是有个讨厌的家伙就要我这样做，
// 迟早会证明，他是错的。
// 2011-12-21
```

2

```
/* 如果你能弄清楚这一段，求
求你务必发邮件至aaa@bbb.com
告诉我，不胜感激 */
```

3

```
/*勇士，乐观地接收这个现实吧！
2011-11-23*/
```

4

```
// 这个不知道是谁写的
// 看起来没用
// 但我不敢删
```

5

```
// 去他妈的。就这样发布吧。
```

6

```
/* 魔法。勿动！ */
```

7

```
/* 致终于来到这里的勇敢的人：
你是被上帝选中的人，英勇的、不辞劳苦
的、不眠不修的来修改我们这最棘手的代
码的编程骑士。你，我们的救世主、人中
之龙，我要对你说：永远不要放弃，永远
不要对自己失望，永远不要逃走。永远不
要哭啼，永远不要说再见。
永远不要说谎来
伤害自己。*/
```

8

```
// 亲爱的维护者：
// 如果你尝试了对这段程序进行‘优化’，
// 并认识到这种企图是大错特错，
// 请增加下面这个计数器的个数，
// 用于对后来人进行警告：
// 人们总共在这里浪费了 = 39h
```

原载于stackoverflow
翻译改编自《外刊IT评论》

西乔
设计师、项目经理。
2006年起携创业团队从事Web
技术外包开发及产品咨询顾问。

如果你有什么好玩的关于程序员的故事、对话、代码，
愿意通过漫画的形式分享，
请给西乔发邮件：arthur369@gmail.com

Different types of comments

```
// sum from 1 to num
sum = 0;

for (int  i = 1;  i < num; i++) {
      sum = sum +1;
}
 System.out.println ("Sum = " + sum);
```

X

Wrong

```
// set product
product = ba

// loop from 2 to "num-1"
for (int  i = 2;  i < num; i++) {
      // multiply "base" by "product"
      product = product * base;
}
System.out.println ("product =" + product );
```

Repeat of the code

```
// compute the square root of Num using the Newton-Raphson approximation

r = num/2;
while (abs (r-(num/r)) > TOLERANCE) {
      r = 0.5 * (r + (num/r));
}
System.out.println ("r =" + r );
```

Nice explanation

To Comment or Not to Comment

- To write **enough** comments, and **no more**.
- Comments are easier to write poorly than well, and commenting can be more damaging than helpful.
- We need to focus on comment **quality**, **not quantity**, so more important than the amount of comments we write are the contents of those comments.

Commenting Techniques

- Explain Why, Not How
- Don't describe the code
- Keep it useful
  - Document the unexpected
  - Tell the truth
  - Are worthwhile
  - Are clear

```
int boundary = 0;              // upper index of sorted part of array
String insertVal = BLANK; // date elmt to insert in sorted part of array
int insertPos = 0; // position to insert elmt in sorted part of array
```

```
// swap the roots
oldRoot = root [0];
root [0] = root [1];
root [1] = oldRoot;
```

```
for (element = 0; element <elementCount; element++ )
      // use right shift to divide by two. Substituting the
      // right-shift operation cuts the loop time by 75%.
      elementList [element] = elementList [element] >> 1;
}
```

```
blockSize = optimalBlockSize ( numItems, sizePerItem);

/* The following code is necessary to work around an error in writeData () that appears only
when the third parameter equals 500. '500' has been replaced With a named constant for clarity.
*/

if ( blockSize == WRITEDATA_BROKEN_SIZE )
      blockSize = WRITEDATA_WORKAROUND_SIZE ;
}
WriteData ( file, data, blockSize );
```

Commenting Techniques

```
// if account flag is zero
 if (accountFlag == 0) ……
```

Not necessary comment:
repeat the code

Improve comments

```
// if establishing a new account
 if (accountFlag == 0) ……
```

Good comment to describe
the intent of the code

Improve code

```
// if establishing a new account
 if (accountType == AccountType.NewAccount) ……
```

Good Self-document code
Not necessary comments

Improve comments

```
// Establish a new account
 if (accountType == AccountType.NewAccount) ……
```

Improved comments to
summarize the whole block of
code

Commenting Techniques

- Choose a **low-maintenance** style

```
// variable          meaning
// ----------        ------------
//   xPos            XCoordinate Position (in meters)
//   yPos            YCoordinate Position (in meters)
```

```
/**********************************************
 * Class:  ABCD                               *
 *                                            *
 * XXXXXXX                                    *
**********************************************/
```

Style that is hard to maintain

Style that is Much easier to maintain

```
/**********************************************
  Class:  ABCD

  XXXXXXX
**********************************************/
```

Commenting Techniques

- File Header Comments
  - Describe the purpose and contents of each file
  - Legal notice

  - Avoid maintenance disaster
    - A list of every function, class, global variable…

      → Rapidly out of date
    - Versions, modification history, …

      → Version control system tells you

In a Nutshell

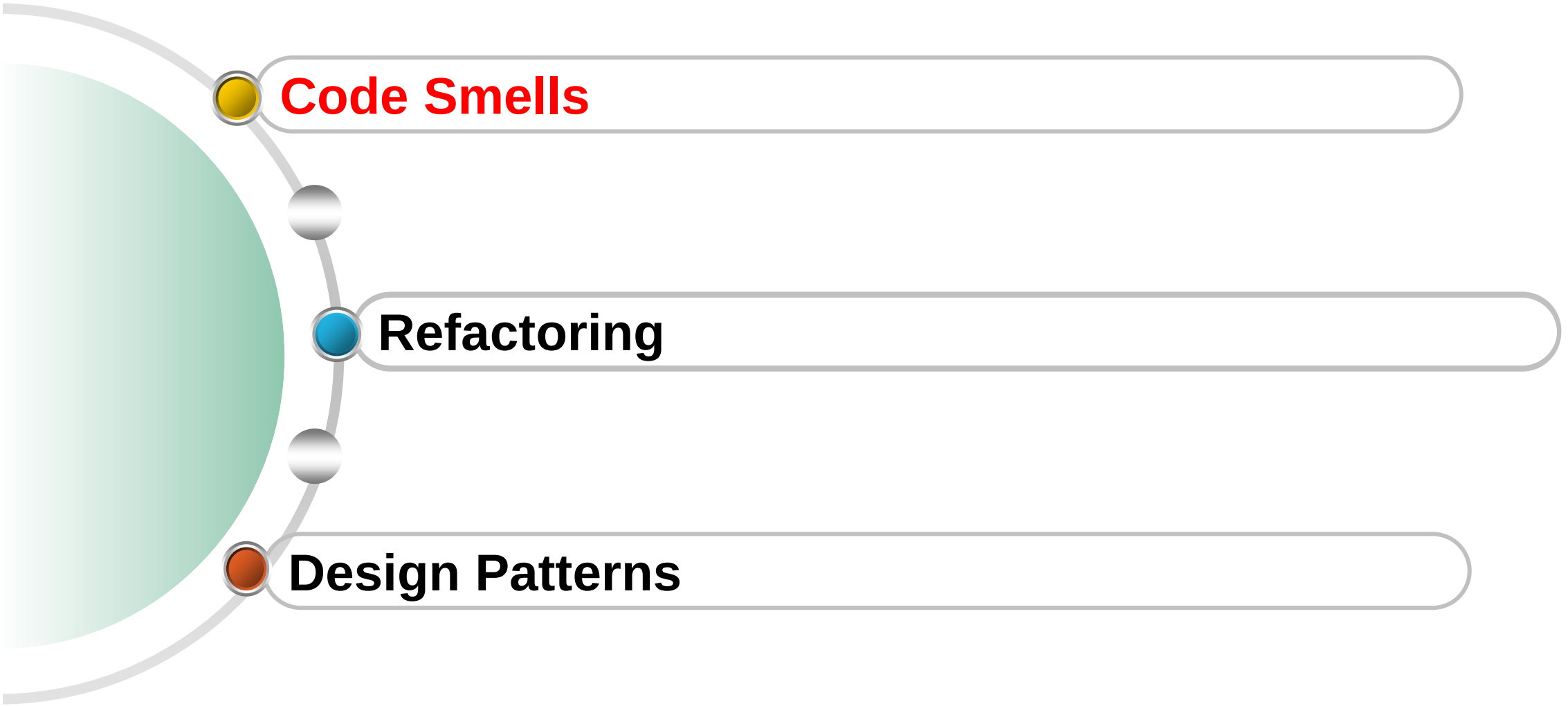
GOOD PROGRAMMERS…

- Try to write ***a few*** really good comments
- Write comments explaining ***why***
- Concentrate on writing good code rather than a plethora of comments
- Write ***helpful*** comments that make sense

BAD PROGRAMMERS…

- Can't tell the difference between good and bad comments
- Write comments explaining ***how***
- Don't mind if comments only make sense to themselves
- Bolster bad code with many comments
- Fill their source files with redundant information

Refactoring



Code Smells

Code Smell

"A code smell is a **surface indication** that usually corresponds to **deeper problem** in the system."

Martin Fowler
https://martinfowler.com/bliki/CodeSmell.html

"A code smell is a **hint** that **something has gone wrong somewhere** in your code. Use the smell to track down the problem."

Kent Beck
http://wiki.c2.com/?CodeSmell

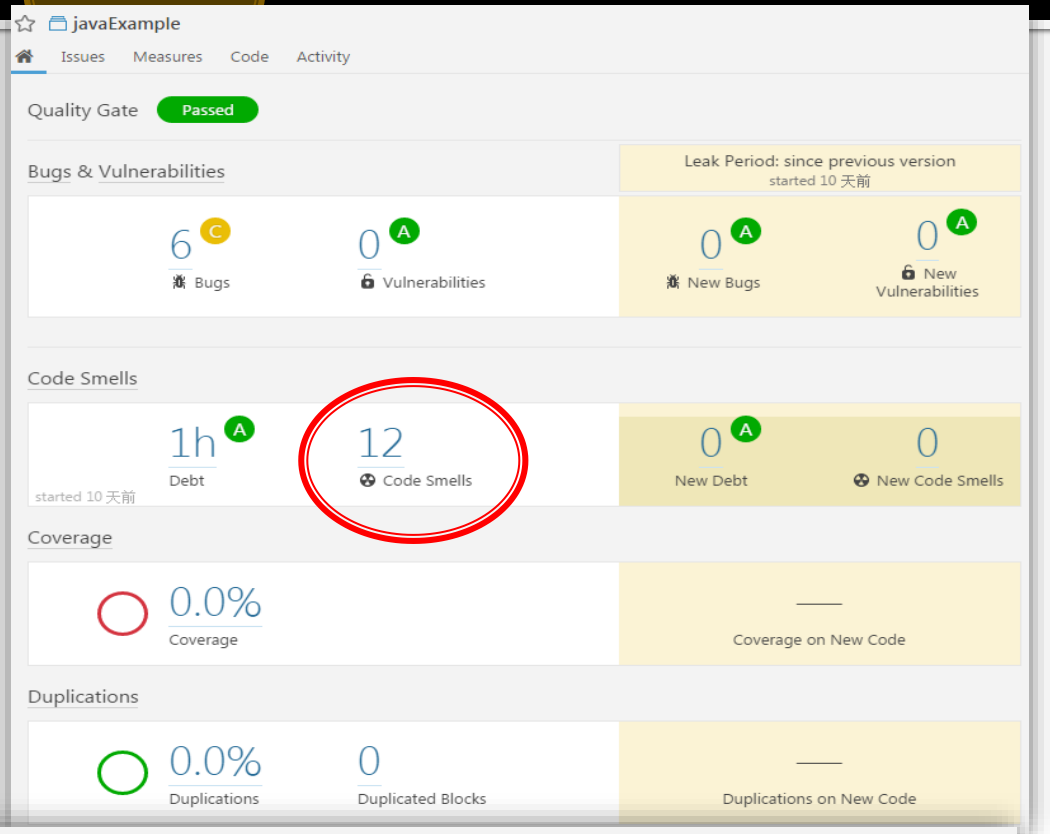
In computer programming, code smell is any **symptom** in the source code of program that possibly indicates a deeper problem. Code smells are usually **not bugs**, they are **not technically incorrect** and **do not currently prevent the program from functioning**.

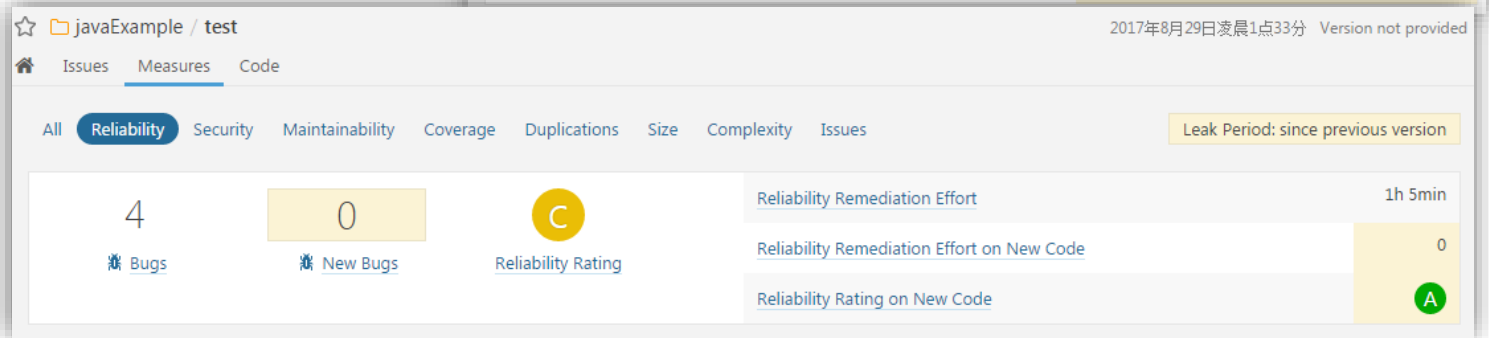
Instead, they **indicate weaknesses in design** that may be slowing down development or increasing the risk of bugs or failures in the future.

Code Smells = Warning Signs
Code Smell ≠ A problem

Sonarqube – Quality

- Continuous inspection
  - Automatic review with static analysis of code quality
- Health
  - Duplicated code
  - Coding standards
  - Code complexity
  - Comments
  - Bugs
  - Security vulnerabilities
  - ……





Some Simple Examples

```
int getRating() { return moreThanFiveLateDeliveries() ? 2 : 1; }
bool moreThanFiveLateDeliveries() { return _numberOfLateDeliveries > 5; }
```

```
int getRating() { return (_numberOfLateDeliveries > 5) ? 2 : 1; }
```

```
Double basePrice = anOrder.basePrice ();
Return (basePrice > 100)
```

```
Return (anOrder.basePrice() > 100)
```

Bad Smells in Code [Fowler 99]

- Duplicated Code
- Long Parameter List
- Large Class
- Long Method
- Shotgun Surgery
- Divergent Change
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class

Lots of Arguments is Bad

- Hard to get good testing coverage
- Hard to mock/stub while testing
- Boolean arguments should be a yellow flag
  - If function behaves differently based on Boolean argument value, maybe should be 2 functions
- If arguments “travel in a pack”, maybe you need to *extract a new class*
  - Same set of arguments for a lot of methods

Bad Smells in Code [Fowler 99]

- Duplicated Code
- Long Parameter List
- Large Class
- Long Method
- Shotgun Surgery
- Divergent Change
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class

Quantitative: ABC Complexity

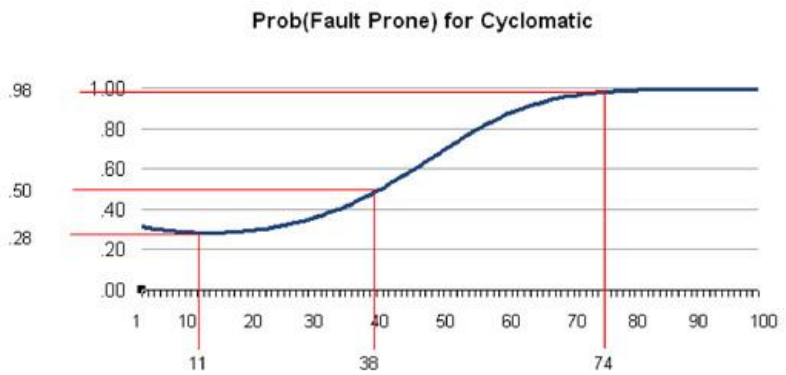
- Counts Assignments, Branches, Conditions
  - **A**ssignments: **=, *=, /=, %=, +=, <<=, >>=, &=, !=, ^=,++, --**
  - **B**ranches: function calls
  - **C**onditions: **<, >, <=, >=, ==, !=, 'else', 'case', 'default', '?'**
- Score = L2 Norm of sum of these
  = SquareRoot($A^2 + B^2 + C^2$)

- NIST (Natl. Inst. Stds. & Tech.): ≤20 /method

Cyclomatic complexity

- Cyclomatic complexity (CYC) is a software metric used to determine the complexity of a program. It is a count of the number of decisions in the source code

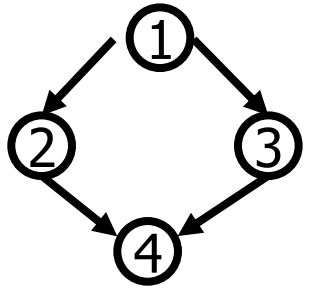
Cyclomatic complexity
= number of edges -
number of nodes +2
= number of decisions +1
= number of regions



< 10 per method (NIST)

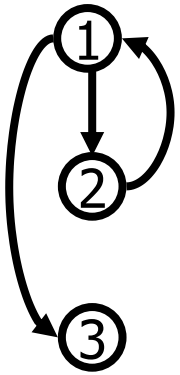
V=1-2+2
=1

Sequence



V=4-4+2
=2

If-Then-Else

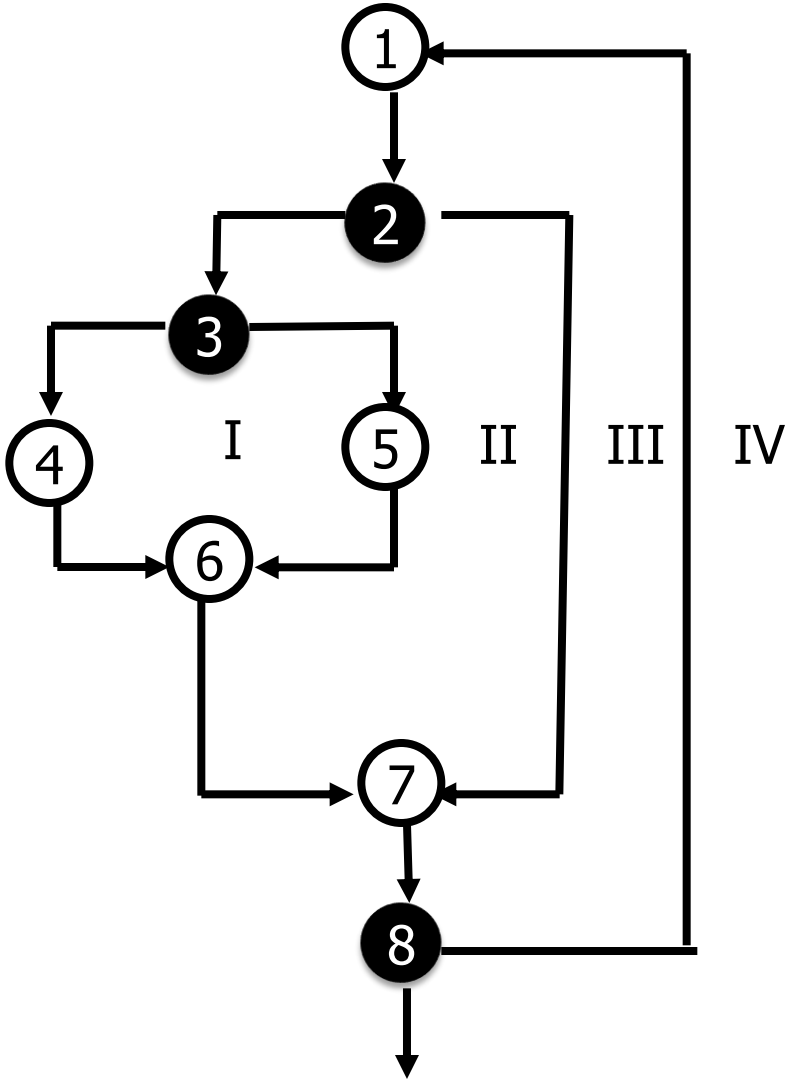


V=3-3+2
=2

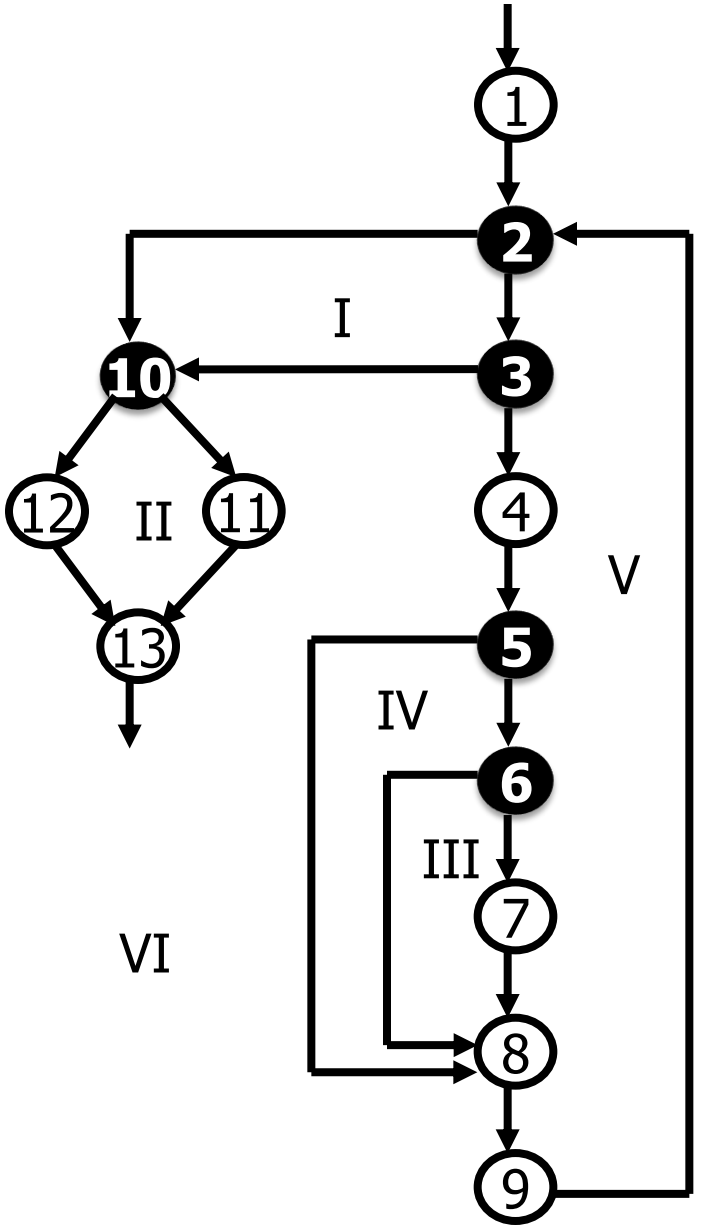
while

V=3-3+2
=2

Until



V=10-8+2=3+1=4



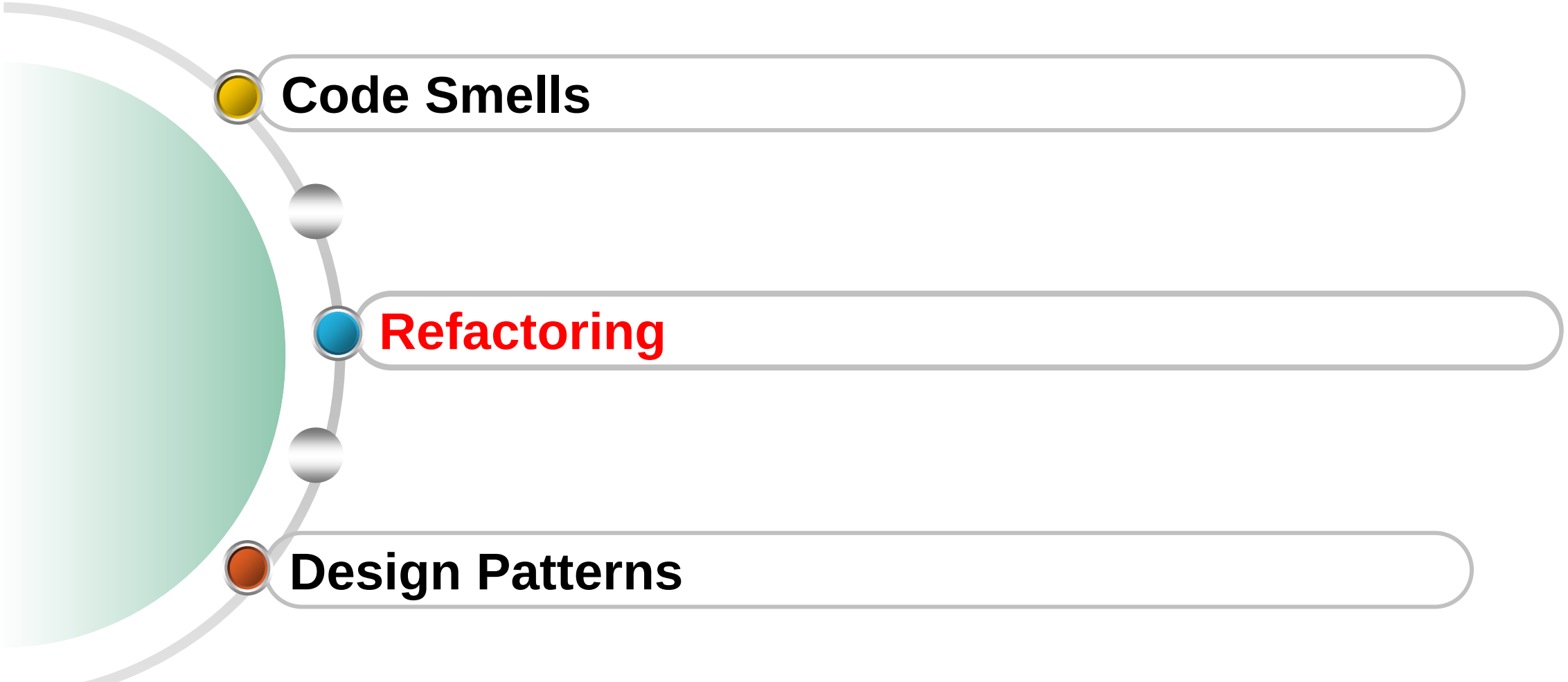
V=17-13+2=5+1=6

Quantitative: Metrics

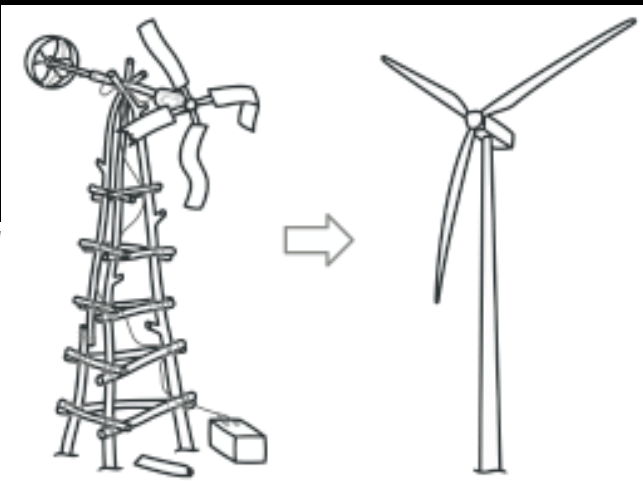
Metric	Target score
Assignment-Branch-Condition score	< 20 per method (NIST)
Cyclomatic complexity	< 10 per method (NIST)

- "Hotspots": places where *multiple metrics* raise red flags
- Take metrics with a grain of salt
  - Like coverage, better for *identifying where improvement is needed* than for *signing off*

Refactoring

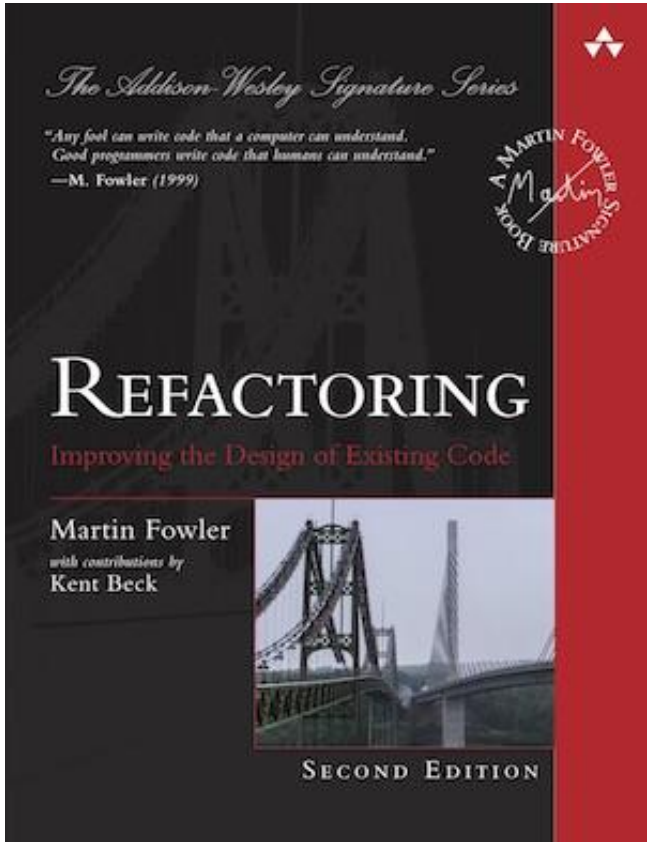


Software Refactoring



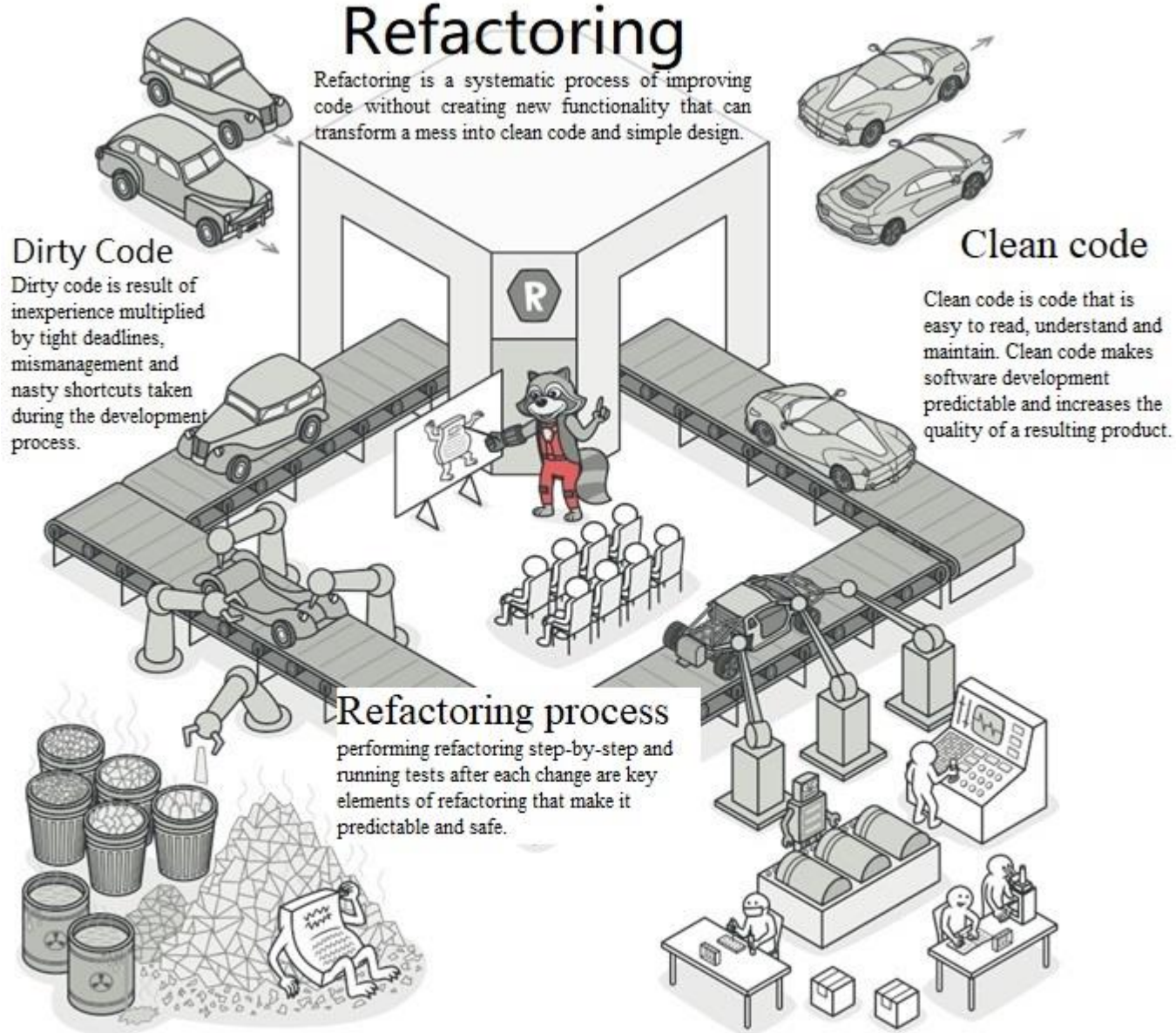
- A software transformation that
  - Preserves the external software behavior
  - Improves the internal software structure

- **Refactoring** is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
  - To enhance some non-functional quality – simplicity, flexibility, understandability, performance
  - Its heart is a series of small behavior preserving transformations. The system is kept fully working after each refactoring



Refactoring

Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design.



Dirty Code

Dirty code is result of inexperience multiplied by tight deadlines, mismanagement and nasty shortcuts taken during the development process.

Clean code

Clean code is code that is easy to read, understand and maintain. Clean code makes software development predictable and increases the quality of a resulting product.

Refactoring process

performing refactoring step-by-step and running tests after each change are key elements of refactoring that make it predictable and safe.

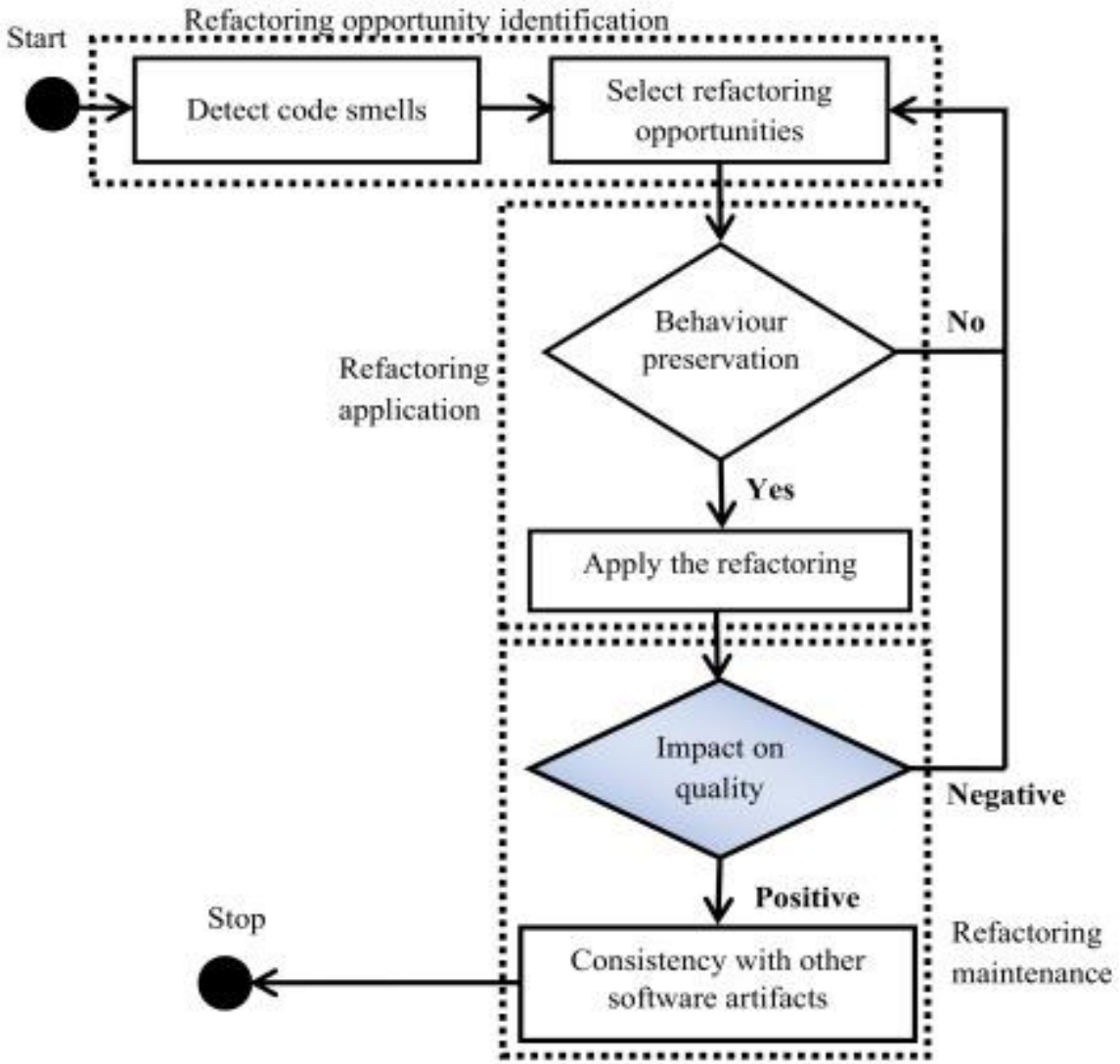
Code smells

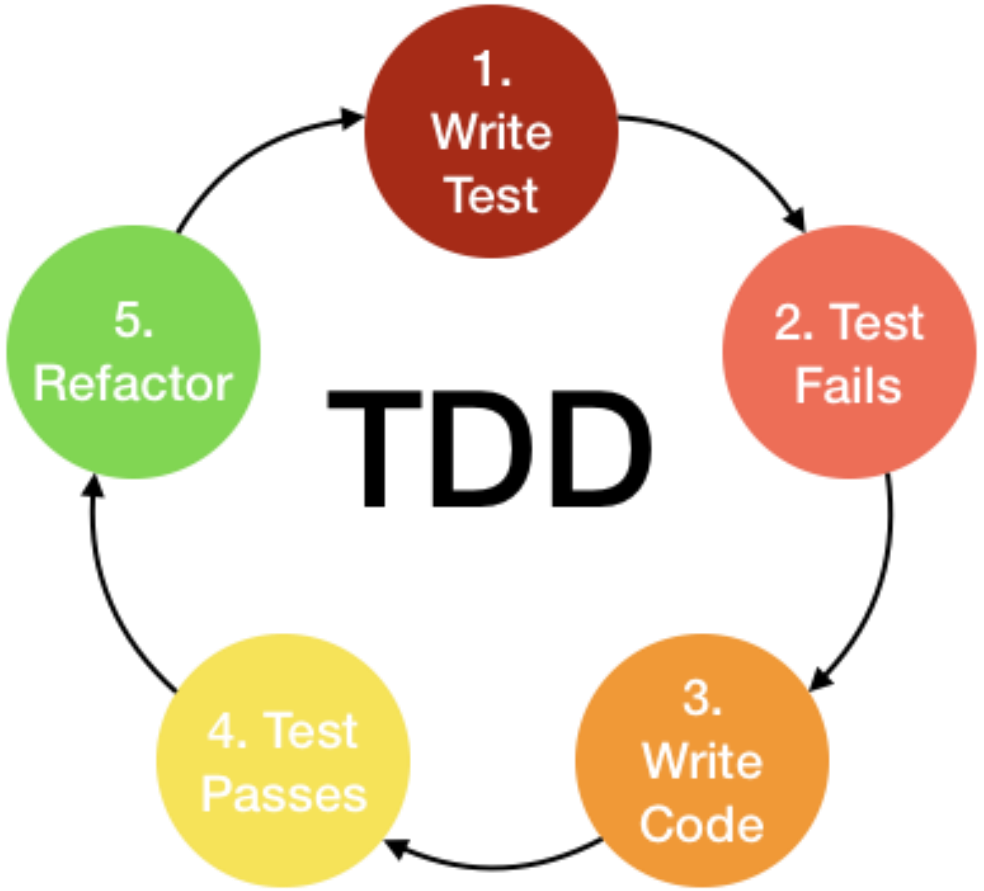
Code smells are indictors of problems that can be addressed during refactoring. Code smells are easy to spot and fix, but they may be just symptoms of a deeper problem with code.

Refactoring techniques

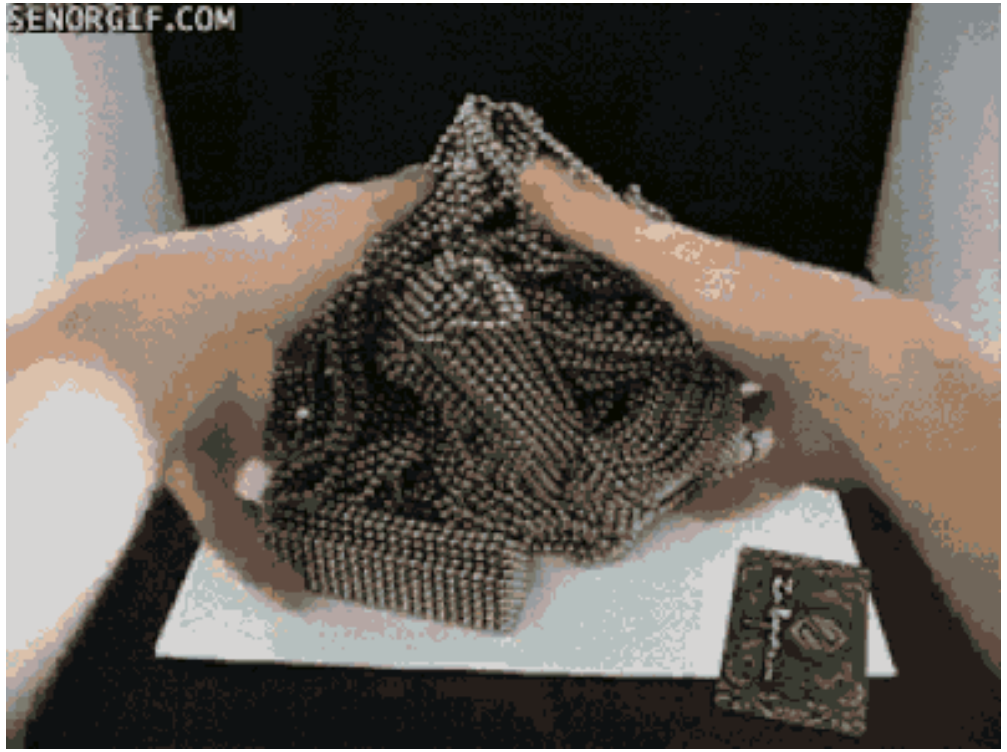
Refactoring techniques describe actual refactoring steps. Most refactoring techniques have their pros and cons. Therefore, each refactoring should be properly motivated and applied with caution.

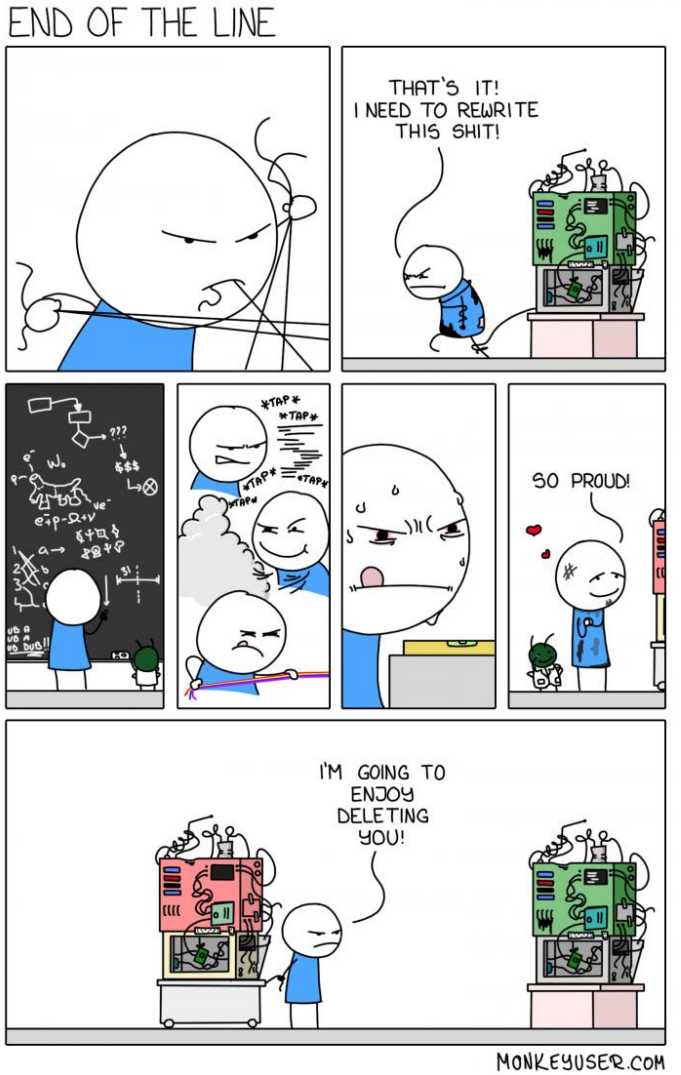
Refactoring Activities





Refactoring : Good vs Bad

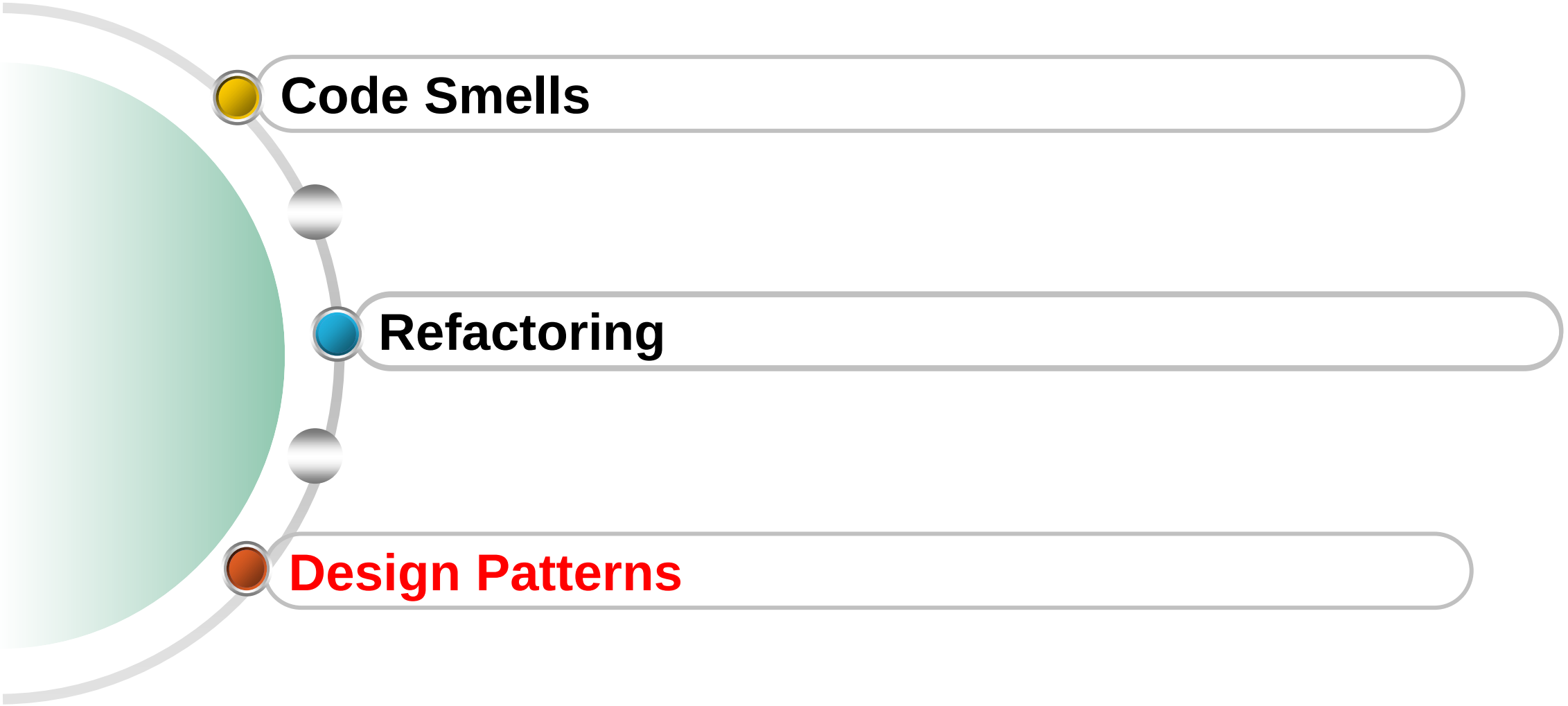




Smells & Refactoring Remedies

Large class	Extract class, subclass or module
Long method	Decompose conditional Extract method Replace loop with collection method Extract enclosing method Replace temp variable with query Replace method with object
Long parameter list/data clump	Replace Parameter with Method Introduce Parameter Object Preserve Whole Object
Shotgun surgery; Inappropriate intimacy	Move method/move field to collect related items into one place
Too many comments	Extract method introduce assertion replace with internal documentation
Inconsistent level of abstraction	Extract methods & classes
Duplicate code	Extract Method, Extract Classes, Pull Up Method, Form Template Method

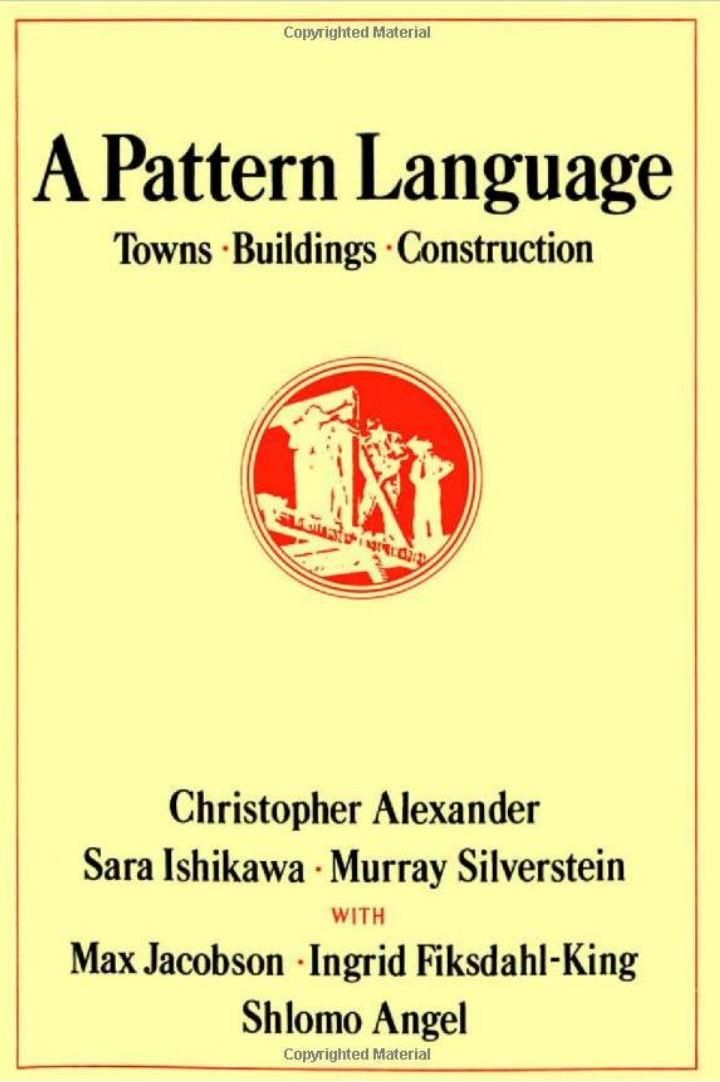
Refactoring



Design Patterns Promote Reuse

"A pattern describes a problem that occurs often, along with a tried solution to the problem" - Christopher Alexander, 1977

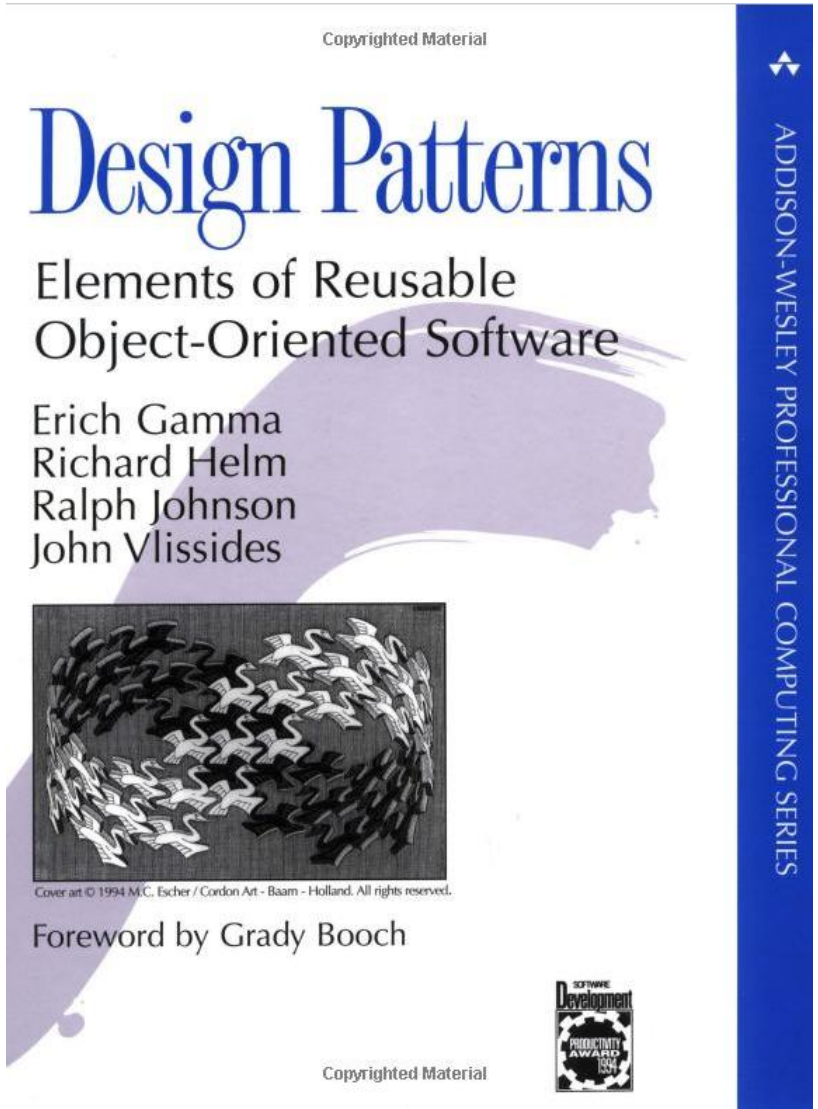
- Christopher Alexander's 253 (civil) architectural patterns range from the creation of cities (2. distribution of towns) to particular building problems (232. roof cap)
- A pattern language is an organized way of tackling an architectural problem using patterns
- *Separate the things that change from those that stay the same*



Software Design Patterns Book by The Gang of Four (GoF)

- 23 *structural* design patterns
- description of communicating objects & classes
  - captures common (and successful) solution to a *category* of related problem instances
  - can be customized to solve a specific (new) problem in that category

- Pattern ≠
  - individual classes or libraries (list, hash, ...)
  - full design—more like a blueprint for a design



The GoF Pattern Zoo

1. Factory
2. ***Abstract factory***
3. Builder
4. Prototype
5. ***Singleton/Null obj***

Creation

6. ***Adapter***
7. ***Composite***
8. ***Proxy***
9. Bridge
10. Flyweight
11. ***Façade***
12. ***Decorator***

Structural

Behavioral

13. ***Observer***
14. Mediator
15. Chain of responsibility
16. Command
17. Interpreter
18. ***Iterator***
19. Memento (memoization)
20. State
21. ***Strategy***
22. ***Template***
23. Visitor

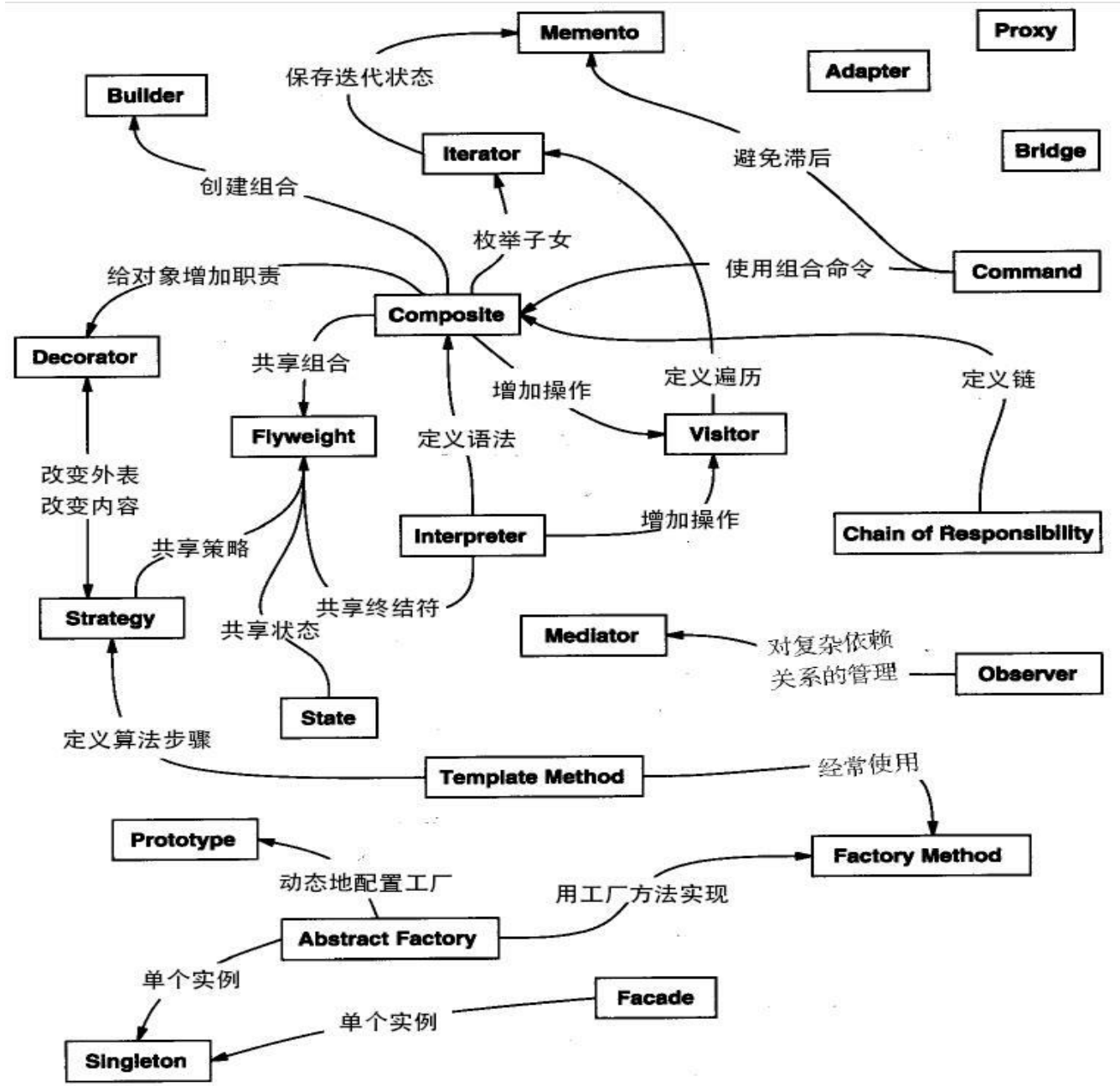
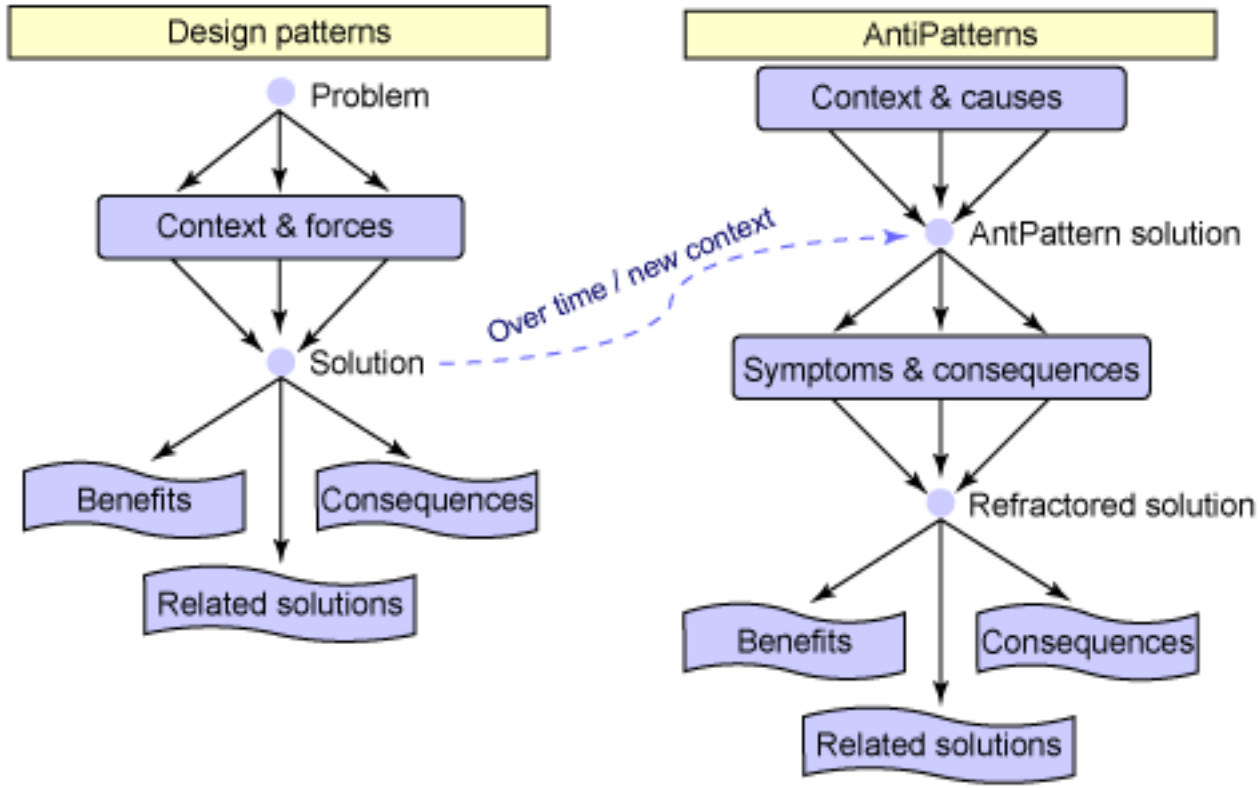

Memento
Proxy
保存迭代状态
Adapter
Builder
Iterator
避免滞后
Bridge
创建组合
枚举子女
给对象增加职责
使用组合命令
Command
Composite
Decorator
共享组合
定义遍历
定义链
增加操作
Flyweight
定义语法
Visitor
改变外表
改变内容
Interpreter
增加操作
Chain of Responsibility
共享策略
共享终结符
Strategy
共享状态
Mediator
对复杂依赖
关系的管理
Observer
State
定义算法步骤
Template Method
经常使用
Prototype
Factory Method
动态地配置工厂
用工厂方法实现
Abstract Factory
单个实例
Facade
单个实例
Singleton

图　设计模式之间的关系

Antipattern

- Code that looks like it should probably follow some design pattern, but doesn't
- Often result of accumulated *technical debt*



Refactoring & Design Patterns

Methods within a class	Relationships among classes
Code smells	Design smells
Many catalogs of code smells & refactorings	Many catalogs of design smells & design patterns
Metrics: ABC & Cyclomatic Complexity	Metrics: Lack of Cohesion of Methods (LCOM)
Refactor by extracting methods and moving around code within a class	Refactor by extracting classes and moving code between classes
SOFA: methods are **S**hort, do **O**ne thing, have **F**ew arguments, single level of **A**bstraction	SOLID: **S**ingle responsibility per class, **O**pen/closed principle, **L**iskov substitutability, **I**njection of dependencies, **D**emeter principle

SOLID OOP principles
(Robert C. Martin, co-author of Agile Manifesto)

Motivation: minimize cost of change

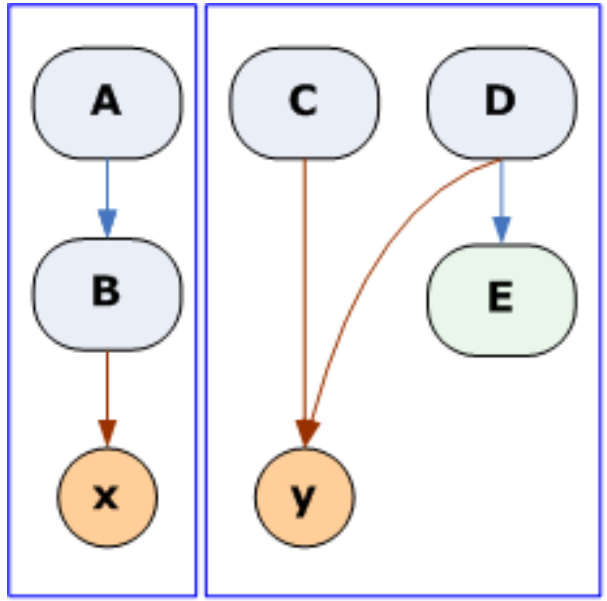
- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Injection of dependencies
- Demeter principle

Single Responsibility Principle (SRP)

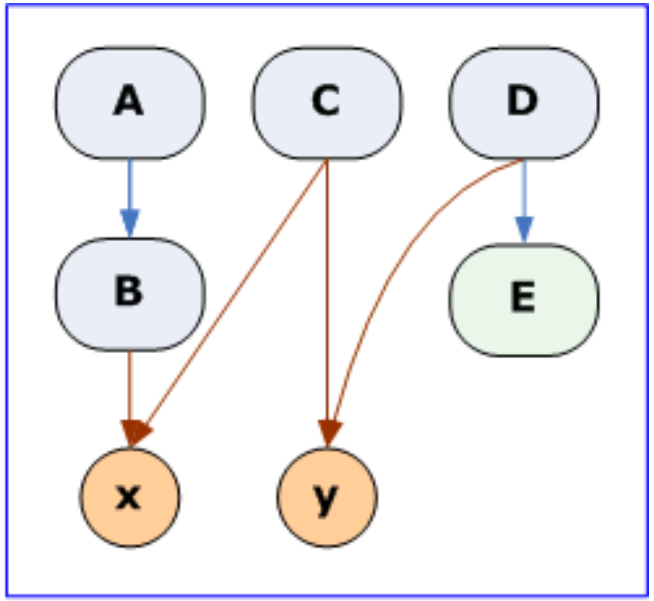
- A class should have *one and only one* reason to change
  - Each *responsibility* is a possible *axis of change*
  - Changes to one axis should not affect others
- What is class's responsibility, in ≤25 words?
  - Part of the craft of OO design is *defining* responsibilities and then sticking to them
- Models with many sets of behaviors
  - E.g., a user is a moviegoer, and an authentication principal, and a social network member, ...etc.
  - really big class files are a tipoff

Lack of Cohesion of Methods

- LCOM4=counts # of connected components in graph where related methods are connected by an edge
  - High LCOM4 suggests possible SRP violation
  - Long class with “cliques” of methods
- Patterns/refactorings:
  - Extract class



LCOM4 = 2



LCOM4 = 1

Open/Closed principle

- **What:** *extending* functionality of a class shouldn't require *modifying* existing code, just *adding* to it

- **Symptoms:**
  - `case` statements based on class or other property that doesn't change after assignment

- **Design patterns/refactorings:**
  - Abstract factory pattern combined with…
  - Template and strategy patterns (capture outline of algorithm's steps, or of overall algorithm)
  - Decorator (add behaviors to a base class)

Liskov Substitution principle

- **What:** instance of subclass/subtype of type T can always be safely substituted for a T
- **Symptoms:**
  - Refused bequest: no meaningful way to implement a behavior of your superclass
- **Design patterns/refactorings:**
  - Composition: rather than *inheriting* from T, create class that has a T as a *component*
  - *Explicitly delegate* method calls on T to that component (inheritance≈implicit delegation)

Injection of Dependencies Principle

- **What:** rather than have class A depend on interface of B, have both depend on an injected common interface (level of indirection)
- **Symptom:** can't extend/add variants of B without *modifying* code in A (violating Open/Closed)
- **Design pattern/refactoring:**
  - Insert a new interface that is always the same from A's point of view, but adapts to different Bs
  - Interface usually abstract class whose concrete subclasses implement variants of B, even with very different interfaces

Demeter Principle

- **What:** talk to friends & friends of friends; everyone else is a stranger
- **Symptom:** long chains of method calls, leading to *mock trainwrecks* in tests
- **Design pattern/refactoring:**
  - Delegate the *behavior* to the right place Visitor pattern (separate traversal from computation)
  - Observer pattern (be aware of important events)

Design Patterns Summary

- Design patterns represent *successful solutions* to classes of problems
  - Reuse of design rather than code/classes

- Separate what changes from what stays the same
  - program to interface, not implementation
  - prefer composition over inheritance
  - delegate!

Thank you!