1-Introduction

Designing Software [for large systems]

1. – Principles
2. – Techniques
3. – Tools
4. – Practices

Libraries: very well tested code solving real problems

Library developers: help many heroes at once!

Dressed with the code developed by everyone else, Hero can win!

How to be a library developer

• Different mindset: no main!

code is parametric

no clear behavior

sort(List<T> list, Comparator<? super T> c)

How to use libraries

• Adapting code without modifying it, Objects as operations Reading documentation

2 – Refactoring and Version Control

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations

In computer programming, code smell is any symptom 症状in the source code of a program that possibly indicates a deeper problem

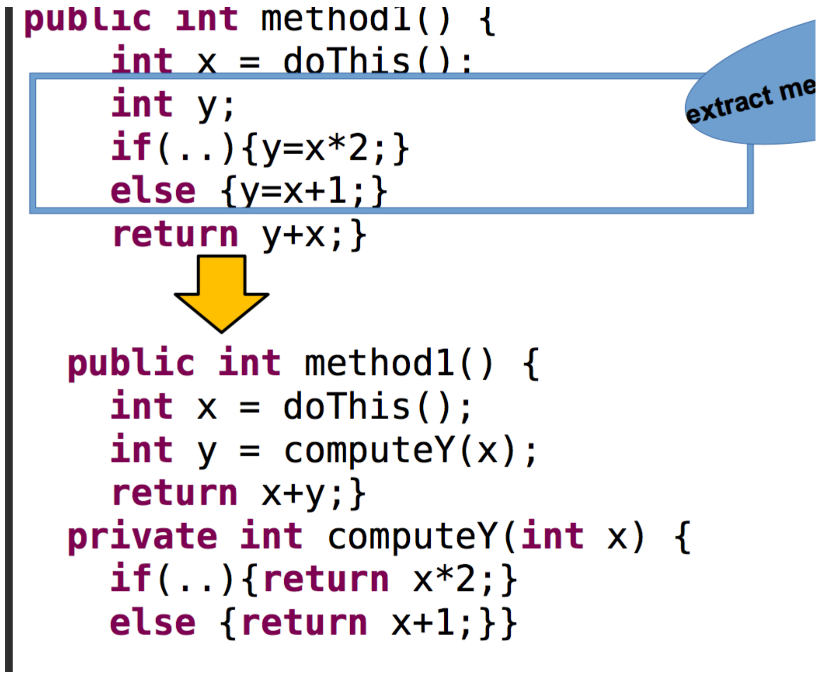
• Big Methods

1. – code too long
2. – code doing too much

• Big Classes

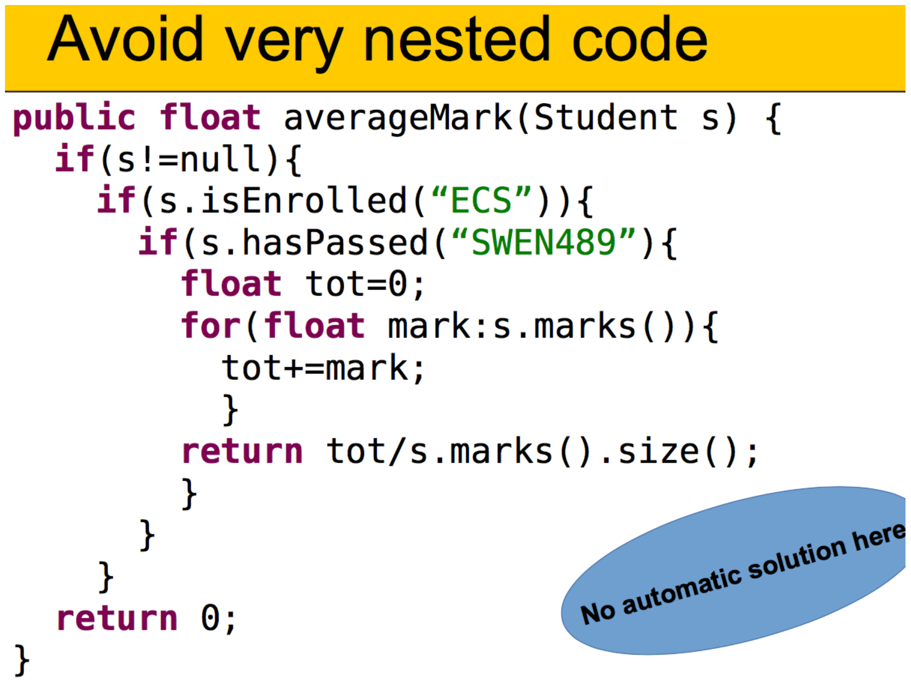
1. – too many variables
2. – “God” classes

- Duplicate Code
    1. – How many times should code do anything?
    2. – “Once and only once”



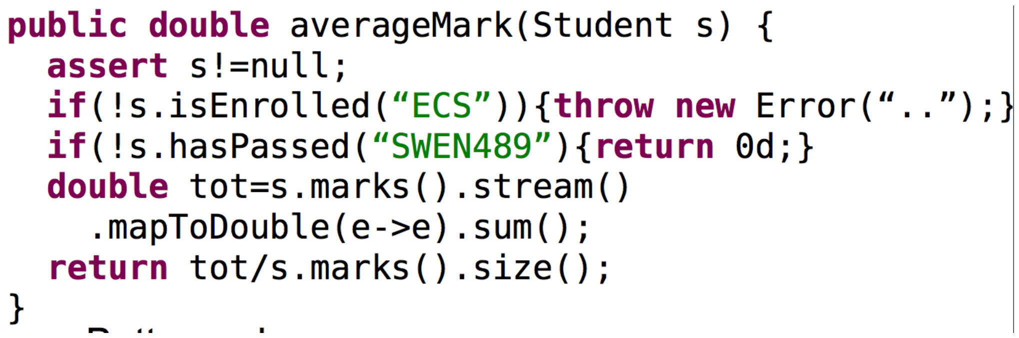
```
public int method1() {
    int x = doThis();
    int y;
    if(..){y=x*2;}
    else {y=x+1;}
    return y+x;}
```

```
  public int method1() {
    int x = doThis();
    int y = computeY(x);
    return x+y;}
  private int computeY(int x) {
    if(..){return x*2;}
    else {return x+1;}}
```


No automatic solution here

Avoid very nested code

```
public float averageMark(Student s) {
  if(s!=null){
    if(s.isEnrolled(“ECS”)){
      if(s.hasPassed(“SWEN489”){
        float tot=0;
        for(float mark:s.marks()){
          tot+=mark;
          }
        return tot/s.marks().size();
        }
      }
    }
  return 0;
}
```



```java
public double averageMark(Student s) {
  assert s!=null;
  if(!s.isEnrolled("ECS")){throw new Error("..");}
  if(!s.hasPassed("SWEN489"){return 0d;}
  double tot=s.marks().stream()
    .mapToDouble(e->e).sum();
  return tot/s.marks().size();
}
```

Better code

1. – be negative: start by checking the absence of errors (both assertions/preconditions and regular errors/exceptions)
2. – try to run away: consider simple cases FIRST
3. – be lazy: use libraries when possible (streams here)

Well defined roles

Difference between printing a string and returning a string?

Difference between exception throw and exception print/log?

Few methods (may be only main+tests) should print things. All the other methods should just throw exceptions and/or returning values or messages.

– leak an exception / assertion / error

– DO NOT PRINT AN ERROR MESSAGE

If a method can not complete its objective

– DO NOT PRINT A SUCCESS MESSAGE

2-Version Control

Git is a Version Control System

– It helps coordinate projects developed in teams

– It provides many useful pieces of functionality:

- Project files stored in distributed repositories – Repositories can be accessed remotely
- Complete history of all changes made
- Each change has a log entry associated with it
- Team members can "synchronize同步 "their code changes easily

– It replaces old and somewhat antiquated过时的 systems, like SVN and CVS
– It is freely available as an open source project

1. Must record changes made to enable integration of each others changes
2. must keep log of each changes they make
3. so if they need to undo one, they know why they made it in the first place
4. Also, good to keep original code before and after every change. So, can put back in working version

3-API Design

Extensible libraries

1. • Fixed task library 固定项目
   offers a precise set of functionalities and a precise behaviour that can not be customized
2. • Easier to write
   • eg. Math.pow, number cruncing, xml parsing,...
3. • Extensible library
   offers ways to adapt its behaviour and extends its functionalities.
4. • Harder to write, it is like defining a family of libraries.
   • eg. swing, collections, eclipse and plugings, frameworks,...

Dynamic dispatch: execute future code

sort(List<T> list, Comparator<? super T> c)
• Sorts the specified list according to the order induced by the specified comparator.
• It can sort collections of types that do not exists yet, using comparison code wrote AFTER the sorting method was completed.
• It is mind blowingy影响深刻, it is reverting重复回到 the usual pattern of dependency, where code call only “pre existent code”.

• It is the key to modularize development.

sort(List<T> list, Comparator<? super T> c)

1. • What happen if tomorrow someone discovers a better sorting algorithm?
2. • They can update the sort method, and sorting around the world will go better;
3. • All sorting
4. • For all types
5. • Past and future
6. • No coding action required by individual programmers

What is a "Good API"?

Easy to learn

1. • Easy to use, even without documentation
2. • Hard to misuse
3. • Easy to read and maintain code that uses it
4. • Sufficiently powerful to satisfy requirements • Easy to extend
5. • Appropriate to audience
6. • Welldocumented

   – java doc, standard way

   – benefits of documentation standardization

Do one thing well

Behavior should be easy to explain

– If its hard to name - that's a bad sign – Good names drive development

• API should satisfy its requirements

• Look for high power-to-weight ratio

You can always add: you can never remove*

*(From a public*** API**) **(Often you can't even add) ***(In private, refactoring is your undo!)

Keep Implementation out of API

Implementation details

– Confuse users

– Stop you changing the API

What’s an implementation detail?

– Do not overspecify behavior (add, hashCode)

Minimise the accessibility of everything

– Maximise information hiding, encapsulation

– Classes & members as private as possible

– Public classes: no public fields (only constants) Simplicity

<u>How to develop an API</u>

Work in a group.Talk to people.

1. • Write to Your API Early and Often
   – “the rule of three”
2. • Start before you've implemented the API
   – Saves an implementation you'll throw away
3. • Start before you've specified it properly
   – Saves you from writing specs you'll throw away
4. • Continue writing to API as you flesh it out
   – Prevents nasty surprises
   – Code lives on as examples, unit tests

Design Forces

1. • Simple
2. • Expressive 有表现力
3. • Cohesive有凝聚力

Be Realistic

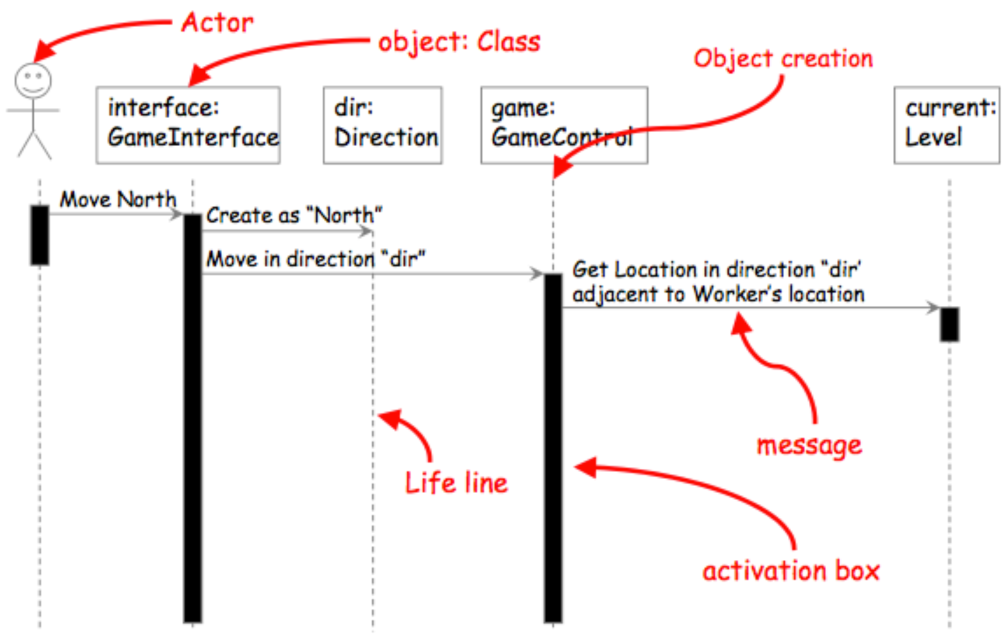
1. Most APIs are over-constrained（不变通）
   – There is no best design
   – You won’t be able to please everybody
   – Aim to displease everyone equally
2. • Expect to make mistakes
   – A few years of use will flush them out

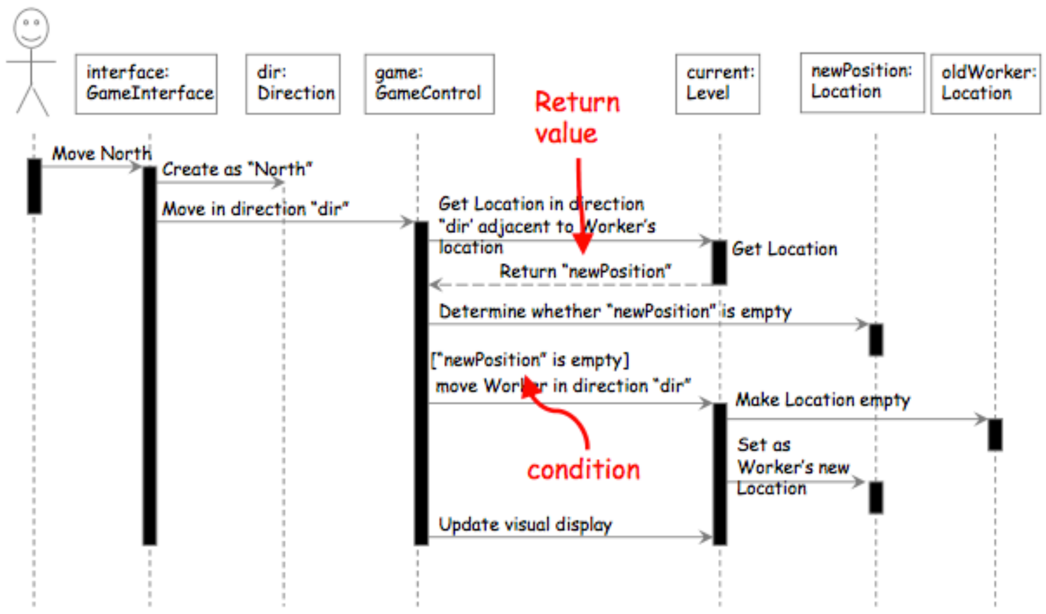
– Expect to evolve your API

Over Engineering

Overengineering (or over-engineering) is the designing of a product to be more robust or complicated than is necessary for its application, either (charitably) to ensure sufficient factor of safety, sufficient functionality, or because of design errors. Overengineering can be desirable when safety or performance on a particular criterion is critical, or when extremely broad functionality is required, but it is generally criticized from the point of view of value engineering as wasteful. As a design philosophy, such overcomplexity is the opposite of the less is more school of thought

Premature Optimisation *（早期优化）*




interface: GameInterface
dir: Direction
game: GameControl
current: Level
newPosition: Location
oldWorker: Location
Return value
Move North
Create as "North"
Move in direction "dir"
Get Location in direction "dir' adjacent to Worker's location
Get Location
Return "newPosition"
Determine whether "newPosition" is empty
["newPosition" is empty]
move Worker in direction "dir"
Make Location empty
condition
Set as Worker's new Location
Update visual display