Agile development


I CAN FEEL YOUR TENSION, PROGRAMMER. YOU MUST LEARN PATIENCE IF YOU ARE TO BECOME AN AGILE MASTER.

Is it for you?

You may feel that agile development isn't for you, because it is about team work and industry practices, and you can't learn anything from it as an individual programmer

But it can increase your personal limit from a few hundred lines to a few thousand

Or...

Or maybe you just want to earn the title of
ninja developer

"Anyone who has watched a ninja movie knows that a small team of inspired, highly trained ninjas will defeat a near-infinite number of clumsy ninjas who might wear the right costume but don't truly pursue the righteous path."

Is it one thing?

No, it is a hodge-podge of tools, techniques, processes, dogmas, and especially anti-dogma dogmas, but it also has an element of karma about it

But there is a core which makes a coherent story and is extremely effective for individual and pair programming

We are going to have a look at that core, and just summarize issues which apply to teamwork and industry

Good programming

The starting point for agile development is good programming, it is not always made completely explicit, but here is golden rule number one

Take small steps

Go from working program to working program, adding features one at a time, compiling and testing often

Otherwise, you'll get the farmer's response

Components

The second golden rule is:

Use small components

That means small functions, methods, modules, classes, or whatever

Each should be responsible for doing "one thing", and should be ***bullet-proof***, i.e. nobody can tamper with the data from the outside (e.g. `private` fields)

Clean code

The best way to find out more about the modern approach to this is to read the book of the Clean Code cult:

Clean Code

To illustrate the change in thinking from the old days, let's have a look at an old program which was once thought OK called `GeneratePrimes`, adapted slightly to make it a complete program, and a modern version `Primes` cleaned by the book author and by me

Before cleaning

GeneratePrimes.java

```
/**
 * This class Generates prime numbers up to a user
 * maximum. The algorithm used is the Sieve of Erat
 * <p>
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Li
 * d. c. 194, Alexandria. The first man to calculat
 * circumference of the Earth. Also known for worki
 * calendars with leap years and ran the library at
 * <p>
 * The algorithm is quite simple. Given an array of
 * starting at 2. Cross out all multiples of 2. Fin
 * uncrossed integer, and cross out all of its mult
 * Repeat untilyou have passed the square root of t
 * value.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
 */
```

After cleaning

Primes.java

```
/* Generate primes up to a maximum
using http://en.wikipedia.org/wiki/Sieve_of_Eratost

import java.util.BitSet;

class Primes
{
  private int max;
  private BitSet crossedOut;

  public static void main(String[] args)
  {
    Primes program = new Primes();
    program.findMax(args);
    program.generatePrimes();
    program.printPrimes();
  }
```

Issues

Methods should be very small; rule of thumb - you can see all of a method at once in the editor

Methods should be ultra-readable - choose names carefully

Don't put comments at the ends of lines, or inside methods - and don't prettify them

A method should have a one line intro comment, or possibly no comment - throw away comments that don't say anything useful, or repeat what the code says

DRY code

Many of the issues in OO programming and OO design are aimed at this golden rule:

Keep your code DRY

DRY stands for "Don't Repeat Yourself"

If ever you see similar-looking code in two or more places, it is a symptom of potential problems – some people call these symptoms smells


I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself
I will not repeat myself

Auto testing

The beginner's blue belt for ~~ninja~~ ~~jedi~~ ~~judo~~ agile mastery is awarded for getting the hang of this rule:

Do automated testing

As a start, do ***whatever*** testing you normally do to make you feel confident, ***but*** make sure you automate it

It will cost you time to write the test code, but save you time when you want to repeat the tests

Unit testing

When you auto-test one particular class, that's called unit testing (in Java, unit = class)

To get the hang of it, you need to (a) see it in action so you know how to do it and then (b) just ***do*** it, like jumping into the deep end of a swimming pool

Text editor

Let's do some development, on a P.E.T. project

We'll build part of a Plain Editor of Text

It has a reasonably natural division into classes, using "Responsibility Driven Design"

- `Pet`: run
- `Files`: load, save, autosave, backup...
- `Text`: insert, delete, undo, redo, lines, search,...
- `Screen`: display, mouse, keys, menus, shortcuts,...

We'll just look at `Text`, and make a first prototype

API

For this first prototype of `Text`, just support:

- `insert(i,s)`
- `delete(i,j)`
- `undo()`
- `redo()`
- `get()`

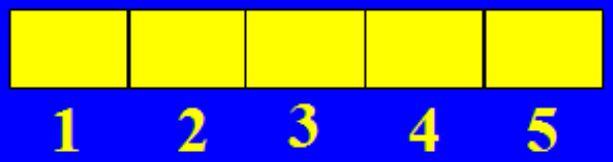
Positions

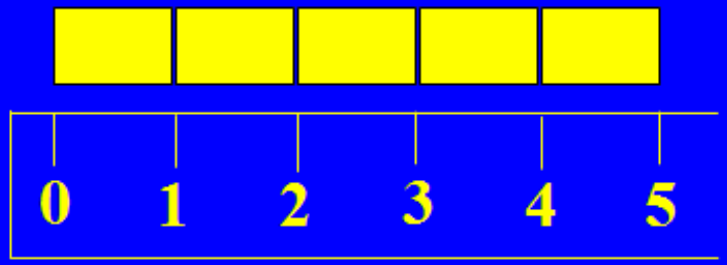
One detail is worth explaining:

`delete(i,j)` does not mean "delete the `i`'th to the `j`'th characters inclusive", i.e. `j-i+1` characters

It means "delete characters between positions `i` and `j`", i.e. `j-i` characters

Moral: don't count, measure:





Undo and redo

The hard part is to implement undo and redo

On insertion, you have to remember the inserted text (for redo) and on deletion, you have to remember the text that was deleted (for undo)

That suggests a simplification, which might keep the code DRYer:

```
insert(i, "x") === replace(i, "", "x")
delete(i, j)   === replace(i, "...", "")
undo()         === replace(...)
redo()         === replace(...)
```

The Op class

Let's define an `Op` class to remember a replacement operation for undo/redo purposes

Op.java

```
class Op {
  final int at;
  final String from, to;
  Op(int at0, String from0, String to0) {
    at = at0; from = from0; to = to0;
  }
}
```

I am breaking the normal rule of making fields `private`, and `final` ("read-only") is my justification

Using an array

To start with, let's store the text in a `char` array

Text.java

```
class Text {
  private char[] chars = new char[8];
  private int length = 0;
  ...
```

It seems better not to use a flexible array (`StringBuffer` or `StringBuilder`) because (a) the array can be indexed directly and (b) it is surprisingly easy to make the array flexible yourself with DIY code

Flexibility

Here's the code (only needed in `insert`) to make the array flexible:

Text.java

```
...
while (length + s.length() > chars.length) {
    chars = Arrays.copyOf(chars, chars.length * 2);
}
```

This follows the rule of thumb "start small and double"

Stacks

The operations to undo or redo are stored in stacks:

Text.java

```
import java.util.Stack;
...
    private Stack<Op> undos = new Stack<Op>();
    private Stack<Op> redos = new Stack<Op>();
...
```

It is usually a good idea to use library classes, unless there is a specific reason not to

The `Stack` class is generic, i.e. `Stack<X>` means "stack of X", just as `X[]` means "array of X"

insert

The code for `insert`, excluding flexible array code, is:

Text.java

```
void insert(int i, String s) {
  if (i < 0 || i > length)
    throw new Error("Bad insert");
  ...
  replace(i, "", s);
  save(i, "", s);
}
```

There is a defensive check that the position is in range

The change to the text is done by a call to `replace`, and the undo aspect is handled by `save`

delete

The code for `delete` is:

Text.java

```
void delete(int i, int j) {
  if (i > j) { delete(j, i); return; }
  if (i < 0 || j > length)
    throw new Error("Bad delete");
  String from = new String(chars, i, j-i);
  replace(i, from, "");
  save(i, from, "");
}
```

One of two approaches is taken if the args are the wrong way round, then there is a defensive check, then the real work is done by `replace` and `save` again

replace

The code for `replace` is:

Text.java

```
private void replace(int at,
                     String from, String to) {
    System.arraycopy(
        chars, at + from.length(),
        chars, at + to.length(),
        length - at - from.length());
    to.getChars(
        0, to.length(),
        chars, at);
    length = length + to.length() - from.length();
}
```

`private` ensures safety, and library calls shift/copy

save

The code for `save` is:

```
private void save(int at,                                  Text.java
                  String from, String to) {
    Op op = new Op(at, from, to);
    undos.push(op);
    redos.clear();
}
```

`private` ensures safety

The usual approach is taken to redo

undo & redo

The code for undo and redo is:

Text.java

```
void undo() {
    if (undos.isEmpty()) return;
    Op op = undos.pop();
    replace(op.at, op.to, op.from);
    redos.push(op);
}
void redo() {
    if (redos.isEmpty()) return;
    Op op = redos.pop();
    replace(op.at, op.from, op.to);
    undos.push(op);
}
```

Testing

My favourite tool for testing is this function:

```
static void is(Object x, Object y) {
    tests++;
    if (x == y) return;
    if (x != null && x.equals(y)) return;
    throw new Error("Test failed: " + x + ", " + y);
}
```

It can compare two objects of (almost) ***any*** type

Since Java 5, it also compares `int`s, `char`s, etc.

It is worth breaking rules, e.g. using `static`, to make tests short and convenient

One-liners

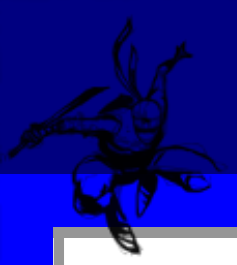
I have a strong preference for one-line tests

```
tests = 0;
text = new Text();
...
text.insert(0,"x"); is(text.get(), "x");
```

This is one line, and would probably do for this class, but it isn't compact or convenient enough for my taste

It is often convenient to arrange tests which have a string as input, and a string as expected output:

```
test("insert 0 x", "x");
```



The test method

Text.java

```
...
private static void test(String in, String out) {
  String[] words = in.split(" ");
  if (words[0].equals("insert")) {
    text.insert(Integer.parseInt(words[1]), words[2]);
  }
  else if (words[0].equals("delete")) {
    text.delete(Integer.parseInt(words[1]),
                Integer.parseInt(words[2]));
  }
  else if (words[0].equals("undo")) {
    text.undo();
  }
  else if (words[0].equals("redo")) {
    text.redo();
  }
  is(text.get(), out);
}
```

How much testing?

It is OK to think of testing as having only one purpose:
to make *you* confident that *your* code works properly

Text.java

```
test("insert 0 xy", "xy");
test("insert 0 a", "axy");
test("insert 2 b", "axby");
test("insert 4 c", "axbyc");
test("delete 0 1", "xbyc");
test("delete 1 2", "xyc");
test("delete 2 3", "xy");
test("delete 0 2", "");
test("undo", "xy");
test("undo", "xyc");
test("undo", "xbyc");
test("undo", "axbyc");
test("redo", "xbyc");
```

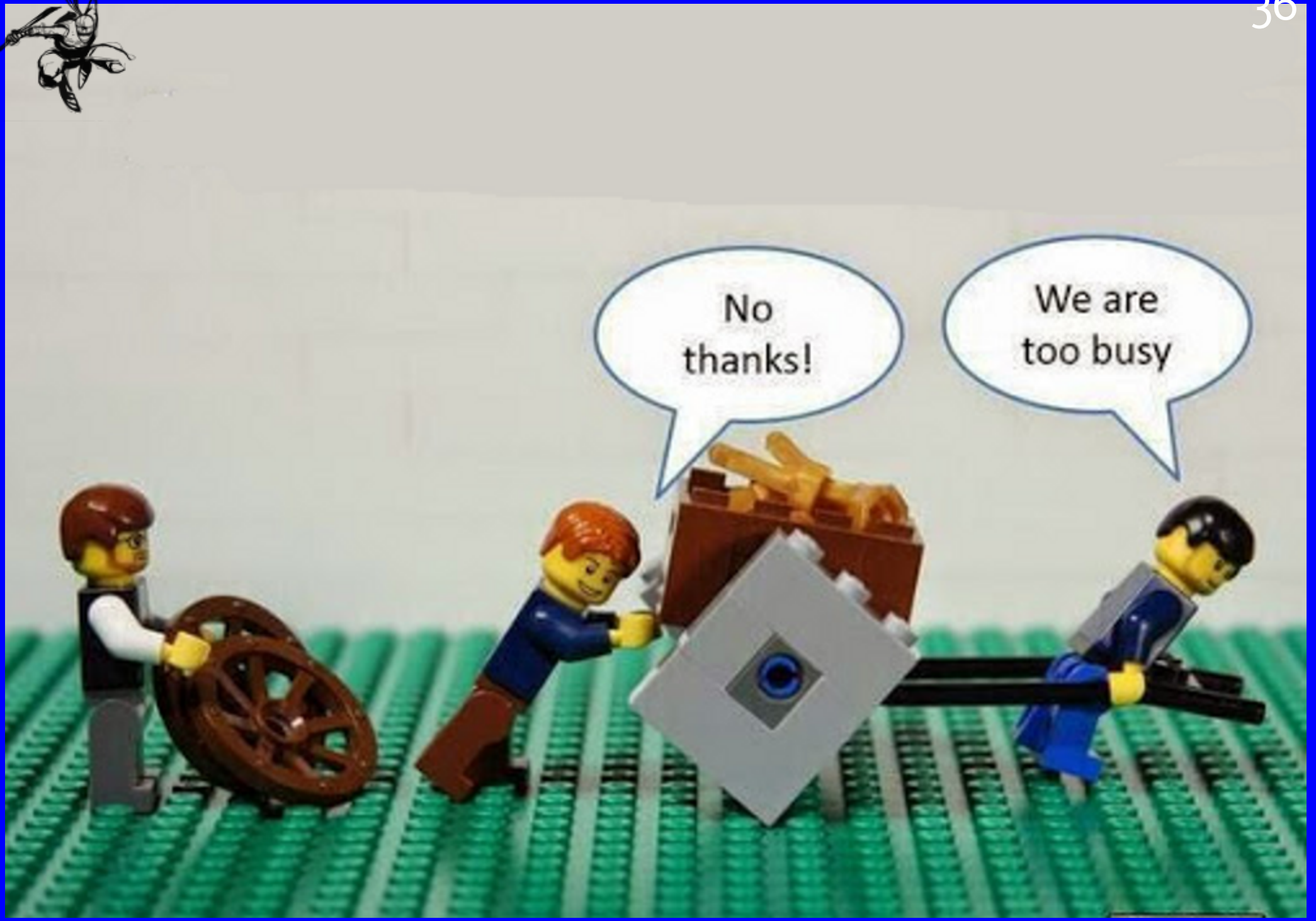
Refactoring

For your agile mastery white belt:

Refactor your projects

That means re-organise it, ready for the next stage

That means ***breaking*** your program, which works ***if*** your auto-testing tells you when it is working again


No thanks!
We are too busy

KISS

For your agile mastery brown belt:

Keep it Sweet and Simple

Don't over-engineer

Do the simplest thing that ***works properly***

Create value-for-money code

It works ***if*** you trust yourself to refactor when needed

Versioning

For your agile mastery black belt:

Use a versioning system

Your editor backs up files, ***but*** overwrites old backups

You need to backup a whole project, in a working state

Long term, don't backup on the same disk, or another disk on the same computer, or a memory stick – use your Uni filestore or the cloud, and don't make silly mistakes

Git

Seems the most popular

On Linux, it is installed already, on MacOS it is part of XCode, on Windows, install git-for-windows

Try the octocat tutorial – good, but it's a survey, leaving one question unanswered

What's the ***simplest*** way of using git?

Using Git

Start versioning a folder: (once per folder)

```
> git init
```

Start tracking a file: (once per file)

```
> git add Pet.java
```

Commit changes: (every time the project works)

```
> git commit -a -m "added main"
```

When you want to backup in the cloud, use `github`

What next?

It turns out that a black belt is a lowly achievement

Now you have to work from 1st dan, ..., to 10th dan, ...

That means taking on board integration testing, pair programming, collective ownership, coding standards, the fixed working week, the planning game, on site clients, small releases, and user stories

When you reach user stories, you realize that its all about naming things again