

Testing

- Manual testing
 - Automated Testing
 - Unit Testing
-
- What are they?
 - When they make sense?
 - How they interact with language features?
 - How they interact with us learning to code?

Coding, different learning steps

- 0 Hello world
 - Console output of constant arithmetic
- 1 Local variables (usually of String/int types)
 - side effects of assignments
 - general idea that execution goes top-bottom
- Most humans can handle this part pretty well:
 - common sense and minimal understanding of math lets you master this part.
 - no challenge = no need to reprogram your brain and fundamentally change **what** you are.
- Sadly, this set a **false expectation** about the rest of your learning experience.

Coding, different learning steps

- 2 if statement, nested ifs
 - Mind-blowing: parametric behavior, and you can nest instructions inside each others
 - Surprising expressive power just out of many ifs.
- Some humans just stop here: cit.
 - Oh, yea! I know Java, I know Java very well, I wrote tons of 'if's,... is the '**while**' that I have never used.
(Italian IT professional working with JSP)
- **Manual Testing** becomes relevant, as in:
 - run you code, read the output, if is equal to what you expect, smile, otherwise edit your code; repeat.

Coding, different learning steps

- 3 while, sequences and field-only classes (structs)
 - Mind-blowing:
 - short program produce large output
 - execution in your head: mentally unfold loops
- In the (far) past, secondary IT education stopped here:
 - using the standard library as if it was a language feature, many useful programs can be written with just this level of understanding.
- **Manual Testing** replaced by **Automated testing**:
 - simulate/provide input and then inspect output to check for the expected result. How? Why?

Automated testing -- How

- Common in the past: Console programs!
 - input file-> program run->output file
- In this (old) setting, a test is another program/script:
 - write down input file
 - run program
 - check generate output == expected output
- In Java you can just set System.in/System.out
 - meaningless in **unit testing** (see later)
 - still useful for (automated) integration testing
- Running your **testing program/script** will perform thousands of checks on your code.

Automated testing -- Why not

- Humans are incredibly persistent; some Excuses:
 - Running your code, typing input, read the output and check is what you expected just feels better.
 - I'm actually checking it, I'm not letting the machine check for me.
 - How I know the bug is not in the test.
 - Setting up the test is a waste of time, let me run it just once and I will see that it works as I expect.
 - Deciding the "precise" expected output is hard, my eyes do a better job of finding wrong behavior.

Automated testing -- Why is good

- Counter points:
 - Running your code, typing input, read the output and check is what you expected just feels better.
- Many feel repulsed by the very idea of **automated testing**
- Our primitive brains play tricks on us
- We feel a sense of “achievement” for seeing our code running
- Is a strong instinct, very hard to fight back.
- This is connected with a series of logic fallacies as we will see later.

Automated testing -- Why is good

- Counter points:
 - I'm actually checking it, I'm not letting the machine check for me
- Funny,
- I thought forcing the machine to work for us was the very goal of programming.
- Is a logic fallacy similar to want not to use library since you fell like
 - "you should be the one writing all the code"

Automated testing -- Why is good

- Counter points:
 - How I know the bug is not in the test
- Test code should be orders of magnitude simpler than tested code
- Test your testing: make some tests that are "supposed" to fail and (automatically) verify that they fail.

Automated testing -- Why is good

- Counter points:
 - Setting up the test is a waste of time, let me run it just once and I will see that it works as I expect
- A realistic program working at the first attempt is like winning the lottery!
- Gambler fallacy:
 - Try for free! if you "win" you believe you saved the time to write down your test.
 - If you fail, you can still chose to write the test, or to make a fast fix and then try to win again.
- Emotional roulette:
 - what you lose is the "time of your life"

Automated testing -- Why is good

- Counter points:
 - Deciding the "precise" expected output is hard, my eyes do a better job of finding wrong behavior
- Answer 1:
- Your program needs to follow a precise specification.
 - without it, you do not really know what you are coding toward.
 - you feel like you are doing progress, but you are not
 - If you do not know in your mind the expected output for all possible input, coding is only "prototype to explore possibilities"
 - Unexpected event: is this a bug or a feature?

Automated testing -- Why is good

- Counter points:
 - Deciding the "precise" expected output is hard, my eyes do a better job of finding wrong behavior
- Answer 2:
- Are you constant with (manual) testing criteria
Are you sure you are not moving the signpost?
- Are you just searching for 'justifications' on why you code is right?
- Setting expectations AFTER seeing the result:
 - a pointless way to feel less wrong
- Do not be ashamed of being wrong.

Automated testing -- Why is good

- Do not be ashamed of being wrong.
You could never be right in the first place.
- Not just "you", everyone, we are all wrong all of the time, and when it seems we are not wrong, is because we do not see a big enough picture.
- "I'm sure my code is 100% right, I think testing is a waste of time...."
- Of course you do..
- Only by accepting that we are **always wrong** we can fix our primitive brains.

Automated testing -- Why is good

- Don Knuth: this is the story of my life



[...]

Birds don't just fly

They fall down and get up

Nobody learns without getting it won

[...]

I wanna try everything

I wanna try even though I could **fail**

I'll keep on making those **new** mistakes

I'll keep on making them **every day**

Those **new mistakes**

Oh oh, try everything

Shakira - Try Everything

How to do **new** mistakes?

- Many features depend on common code.
- Manual testing = check again manually that all the features still works after any bugfix.
- Imagine you are writing a video-game that takes **several hours** to complete, even with cheats on.
- How many times to you plan to actually play your game top to bottom over and over again?
- Would it be fun?

How to do **new** mistakes?

- Automated testing:
 - Summon a virtual you that can run that boring double checking over and over again,
 - virtual you is blazing fast!
 - virtual you has a perfect attention span!
 - virtual you is very methodic!
- Now you can focus on doing **new** mistakes, while virtual you takes care of the old ones over and over again.
- A test failing = virtual you calling for your attention, something **NEW** is happening!

How to do **new** mistakes?

- Running your automated tests over and over again whenever you change anything in the code, is

Regression testing!

- A good way to grow confidence in your code
- Also connected to the main mantra of developers:

Fail Faster my friend... Fail Faster!

Moving toward Unit testing

- Up to now:
 - we just discussed testing as in “testing your whole application”. This is often called
Integration testing
- Next:
 - how to test pieces of your application in isolation!
Unit testing

Coding, different learning steps

- 3 while, sequences and field-only classes (structs)
- 4 methods/functions

```
Xxx main() {  
    xxx xxxx  
    xxx xxxx  
    //handle simple case  
    xx xxxxxxxx {  
        xxxxxxxxxxxxxxxxxx  
    }  
    //handle harder case  
    xxxxxx xxxxxx {  
        xxx xxxxxx  
        xxx xxxxxxx  
        xx xxxxxxxx {  
            xxxxxxxxxxxxxxxxxx  
        }  
    }  
    xxxxxx xx  
}
```

divide in methods

```
xxxx handleSimpleCase(xxx xx) {  
    xx xxxxxxxx {  
        xxxxxxxxxxxxxxxxxx  
    }  
}  
void handleHarderCase(xxx xx) {  
    xxxxxx xxxxxx {  
        xxx xxxxxx  
        xxx xxxxxxx  
        xx xxxxxxxx {  
            xxxxxxxxxxxxxxxxxx  
        }  
    }  
}  
xxx main() {  
    xxx xxxx  
    xxx xxxx  
    handleSimpleCase(x);  
    handleHarderCase(y);  
    xxxxxx xx  
}
```

- Why is good?

Division in methods-- Why not

```
Xxx main() {  
    //handle simple case  
    xxxxxxxxxxxxxxxxxxxxxxxx  
    //handle harder case  
    xxxxxxxxxxxxxxxxxxxxxxxx  
    xxxxxx xx  
}
```

```
xxxx handleSimpleCase(xxx xx) {  
    xxxxxxxxxxxxxxxxxxxxxxxx  
}  
void handleHarderCase(xxx xx) {  
    xxxxxxxxxxxxxxxxxxxxxxxx  
}  
xxx main() {  
    handleSimpleCase(x);  
    handleHarderCase(y);  
    xxxxxx xx  
}
```

- Clearly, there is more code to the right!!
- The code reads better on the left, you just start from the top and read forward.
- Why should we learn yet another concept!!
- It does not seams to add any expressive power

Division in methods-- Why is good

- Clearly, there is more code to the right!!
→ Same amount of complexity, diluted to be more clear
- The code reads better on the left, you just start from the top and read forward.
→ Only for very short programs. Use methods to keep a constant abstraction level.
- Why should we learn yet another concept!!
→ Naming things is the main skill of humans, and the very base of all design
- It does not seams to add any expressive power
→ Now, **Unit tests** are possible!

Units of code, Units of testing

- Every method should have its well specified individual role behavior.
- For example, reading external input/messaging the user is a different role with respect to computation.
- Most methods should not use `System.out/System.in`, neither directly, nor indirectly (by calling other methods)
- Some humans are confused about
 - `System.in` vs method parameter
 - `System.out` vs return type
 - `System.err` vs leaking exception
- This confusion prevents division of roles, and thus unit testing.

Units of code, Units of testing

- Create objects: objects need to be easy to create in any valid state.
- Call a method: Testable methods will be called by providing parameters. (Not System.in)
- check the result: Testable methods will return informative value. (By 'return', not by System.out)

```
@Test public void youngerIsFaster() { //single case
    Person mike=new Person(25,75); //age, weight
    Person joe=new Person(60,85);
    assertFalse(joe.isFaster(mike));
}
@Test public void youngerIsFasterMany() {
    for(int i=0;i<10;i++) { //for each to capture many possibilities
        for(int j=0;j<10;j++) {
            Person mike=new Person(40+i,120);
            Person joe=new Person(20+j,75);
            assertFalse(joe.isFaster(mike));
        }
    }
}
```

Changing and learning is **painful**.

- variables, if, while, sequences:
 - fundamental, you just can not code without.
 - so humans **accept** this pain.
- Multiple methods/classes, multiple levels of abstraction, modularization...
 - is very inefficient, but you can survive without them, and just do manual testing
 - thus humans **resist** this pain
- It is like not wanting to go to the dentist.
- Many stay in discomfort for the **fear** of more pain.
- Embrace **modular code** and **unit testing**



Coding, different learning steps

- 4 methods/functions
- 5 classes, interfaces and subtyping
- Once division in functions is "accepted", naive use of classes is easier to teach.
- However, to properly test code with multiple interconnected classes is hard.
- Design patterns make it easier!
- Friction between encapsulation and testability!
- Dependency Injection (advanced topic)

Testing readings/videos

- <https://www.youtube.com/watch?v=4F72VULWFvc>

- <https://www.youtube.com/watch?v=wEhu57pih5w>

The Clean Code Talks -- Inheritance, Polymorphism, & Testing Misko Hevery of Testability corps @Google

- <https://www.youtube.com/watch?v=dvKeCcxD3rQ>

Half-life of knowledge

- https://www.youtube.com/watch?v=azoucC_fwzw

Kevlin Henney GUTs

<https://www.youtube.com/watch?v=5hJCxSiH4ts>

We look for evidence that reinforces our models
(please ignore the Nokia spot after minute 12:30)

- <https://www.youtube.com/watch?v=QOaaUHUnlz0>

Neil deGrasse Tyson on 'we are all wrong'