Programming: Good Practice Rules

Guillem Barroso and Sergio Zlotnik

guillem.barroso@upc.edu

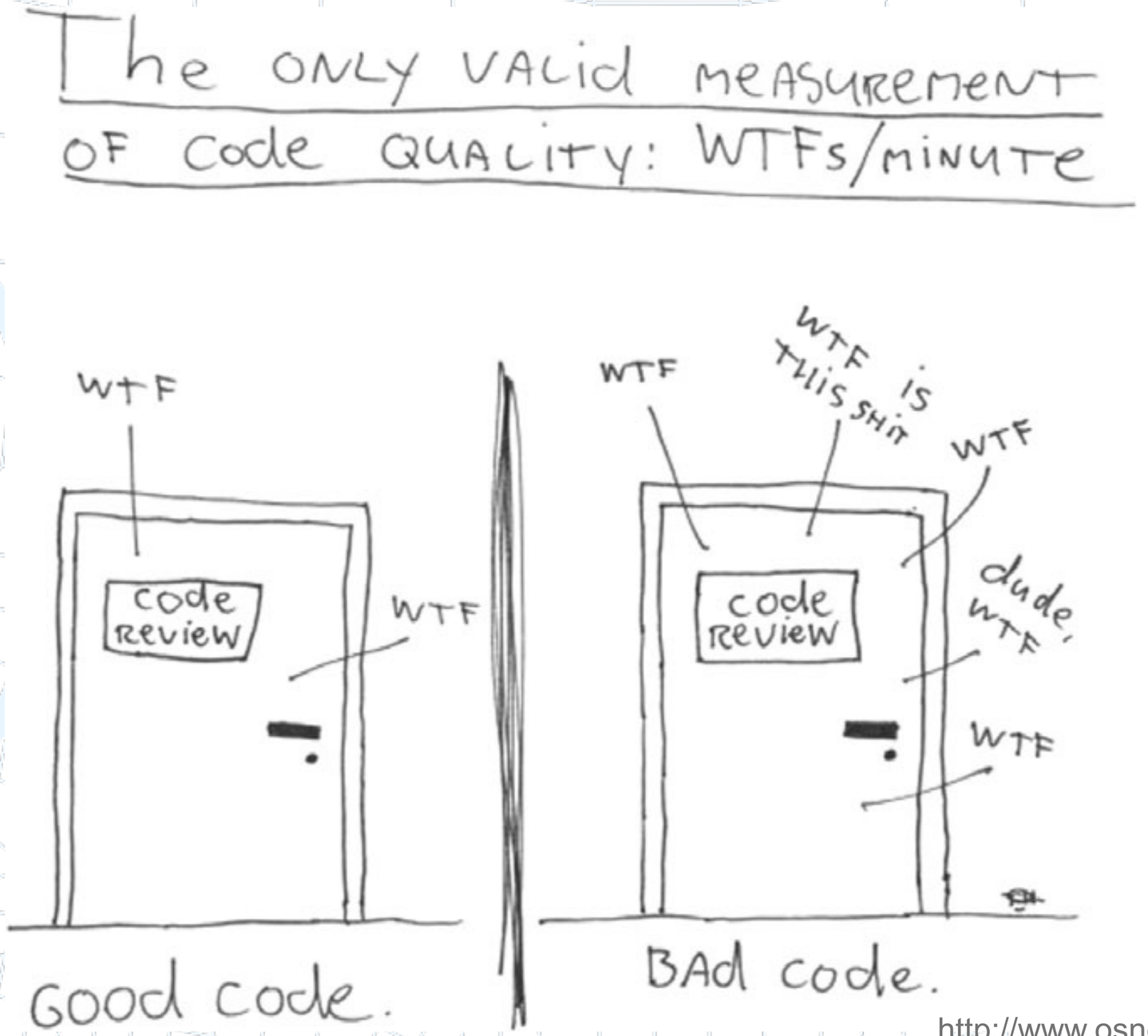
sergio.zlotnik@upc.edu

Laboratori de Càlcul Numèric (LaCàN)

Universitat Politècnica de Catalunya - BarcelonaTech (Spain)

http://www.lacan.upc.edu

Coding…



http://www.osnews.com/story/19266/WTFs_m

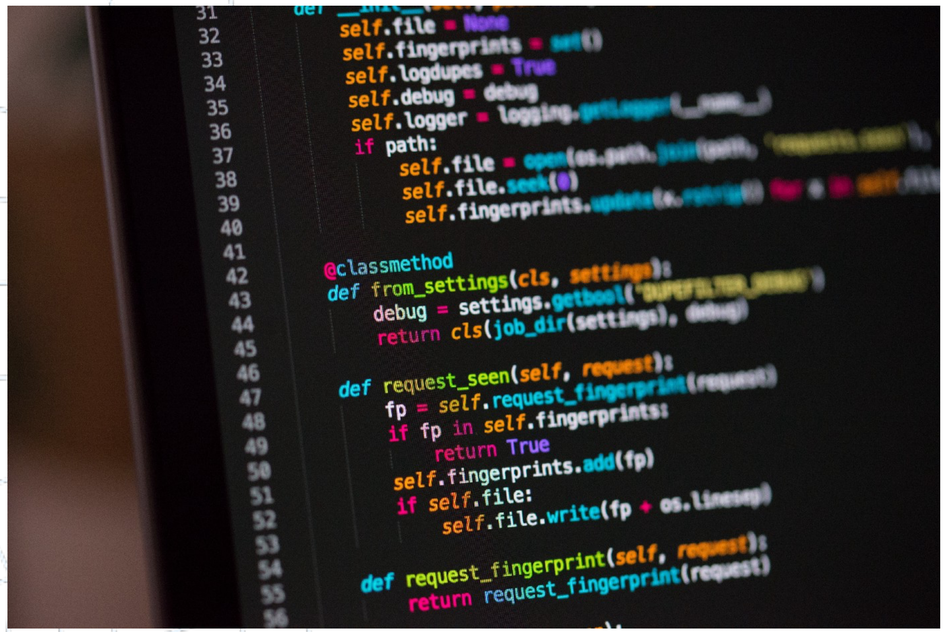
Outline: 3 ideas

- Writing code for humans!
- Use functions!
- Coding workflow tips…

- Writing code for humans!
  - The **importance of a readable code**. Code is read more often than it's written.
  - **Naming** is very important and deserves effort.
  - **Comments** and **formatting** can help to increase the code readability.

Second Idea

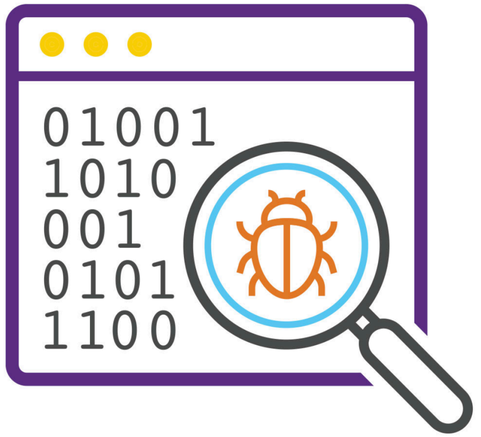
- Use functions!
  - Functions **do one thing**. Better **small**!
  - Functions **start once and end once**
  - Functions have **inputs** and **outputs**



Third Idea

- **Coding workflow tips…**
  - Versioning
  - Testing
- A good **reference**:
  “Clean Code” by Robert C. Martin

```
2.1.0
```



Programming for humans

"We are authors. We are responsible for communicating well with our readers"

Programming for humans

1. The importance of a readable code

2. Meaningful names

3. Comments

4. Formatting

The importance of a readable code

- A code **WILL be read by humans**, either yourself or others. Even if it is only you, it will look as it was coded by another person in 6 months.
- The **code** has to be kept **clean over time**. It is sometimes hard to take an active role in preventing code degradation.
- Mispercieved idea that most of the effort goes into writing a code instead of reading it.
- The **ratio reading/writting** is well over 10:1.
- Constantly **reading old code** in order to **write new code**.

Meaningful names

- Naming is a very large part when coding. **Names are everywhere**; variables, functions, arguments, classes, packages, source files, directories, …

- Different **naming conventions** when joining several words. If "`user`", "`login`" and "`count`" are three keywords that clearly define a variable, the most common conventions are:
    1. **Camel case**: `userLoginCount` (often for variables)
    2. **Pascal Case**: `UserLoginCount` (often for declaring classes)
    3. **Snake case**: `user_login_count` or `USER_LOGIN_COUNT` (often for constants)
    4. **Kebab case**: `user-login-count` (often used in URLs)

Meaningful names

- How to choose a name
  - Use **intention-revealing names**.

    `int d; // elapsed time in days` (name does not give any information. The comment tries to explain the variable). Instead,

    `int elapsedTimeInDays;` is self-explanatory.

  - Should **answer** all big **questions** (why, what it does, how it is used…)

    `theList` is a poor name selection.

    `productsList` could be a much better option if it is indeed a list

Meaningful names

- **Avoid disinformation**

  `productsList` would be misleading if it is not a list. Then, `products` is more concise and does not imply the type of the variable

- **Clearly recognisable**

  The names `XYZControllerForEfficientHandlingOfStrings` and `XYZControllerForEfficientStorageOfStrings` may be very informative but they have very similar shapes, which can be confusing.

- **Avoid** using **non-informative** names such as series `a1, a2, …, aN`.
- **Noise words are redundant**. `moneyAmount` and `money` are indistinguishable. Similar with `customerInfo` and `costumer`.

Meaningful names

- Use **pronounceable names**

  The use of `genymdhms` (generation date, year, month, day, hour, minute and second) can jeopardise human communication since it is not pronounceable. Instead, a descriptive name such as `generationTimestamp` would be better.

- Use **searchable names**

  Single-letter names and numeric constants are not easy to locate across a body of text. `MAX_CLASSES_PER_STUDENT` may be easily located, but the number `7` could be more troublesome. Similarly, the name `e` may appear in almost all lines of a code.

- To sum up, **CLARITY IS KING**

Comments

- Comments are **lines** in a code that are **not executed**. Thus, they are purely oriented to humans.

- Depending on the language, a line will be commented in a slightly different way such as

```
// This is a comment
% This is a comment
# This is a comment
```

Comments

- “The proper use of **comments is to compensate for our failure** to express ourself in code”.

- However, it is very **unlikely that we can express everything** we want without any comments on our code.

- Find a **good balance** between no comments and a code full of comments.

Comments

Why are **comments dangerous**?

- The older a comment is the more likely it is to be just plain wrong. **We can't realistically maintain** them when the code changes and evolves.
- **Inaccurate comments** may be far worse than no comments at all. They delude and mislead.
- **Comments do not make up for bad code**. It is better to invest efforts on cleaning and organising the code properly than to try to explain a messy code with comments.

Comments

When should we **use comments**?

- **Informative comments** can be used to provide basic information or a clarification of what the code is doing if it is not that straightforward.

- Comments can also provide information on the **code historic**, such as changes that where done over time.

Comments

- Other comments can indicate **assumptions** or **simplifications** made by the developer.

- **TODOs** and **FIXMEs** are also common comments that need to be reviewed later to improve or fix something.

  However, be careful with these since there is no point on adding something that is never reviewed again. It is always better to do it right in the first place.

Comments

- Sometimes a piece of code may be convoluted, but there is always a work around. What would you rather see?

```
//Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG)&&(employee.age > 65))
```

or

```
if (employee.isEligibleForFullBenefits())
```

In many cases **creating a function that says the same thing as the comment** works just fine.

Formatting

- Code formatting is about **communication**. As motivated before, the functionality that you create today has a good change of changing in the next release and the **style and readability of your code will have a high effect on all changes that will be made**.

- **Vertical formating**
  - Small files are usually easier to understand than large files
  - The name of a source file should be simple but explanatory.
  - The topmost parts of the file should provide the high-level concepts and algorithms.
  - Detail should increase as we move downward, until we find at the end all the details.

Formatting

- **Vertical openness.** Each line represents an expression or a clause, and each **group of lines represents a complete thought**. Then, thoughts should be separated from each other with blank lines.

- **Vertical density.** Lines that are tightly related should appear vertically dense.

- **Concepts** that are closely **related** should be kept **vertically close** to each other.

Formatting

- **Horizontal Formatting.** How wide should a line be? Programmers prefer short lines. You should **never have to scroll to the right**. However, monitors are too wide for that nowadays. It really depends, but **120 characters per line** seems to be a good limit.

- **Horizontal Openness and density.** Horizontal white spaces are used to accentuate assignment operators.

- **Indentation**. A source file is a hierarchy. There is information that pertains to the file as a whole, to a individual class/function, etc. Each level of hierarchy is visible using indentation.

Formatting

Example of java code with vertical and horizontal formatting.

Formatting

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );

  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Functions

"Short and with one purpose"

Functions

- Functions are the **first line of organisation** in any program**.**
- “The first rule of functions is that **they should be small**. The second rule of functions is that **they should be smaller than that**”.
- Functions should only **do one thing**. They can either do or answer something, do it very efficiently, but they should only do one thing.
- A more advanced concept is **object-oriented programming** (out of the scope of this class). Where functions that do one thing are called from other object such as methods, classes, etc. For now, think of it as a source file that calls functions to do something.

Functions

- The **step-down rule**.

  Similar to vertical formatting, functions that are related to each other should be close together. If some of them are more important or **more general should appear first**, top to bottom.

- Use **descriptive names**.

  A long descriptive name is better than a short enigmatic name. A log descriptive name is better than a long descriptive comment. Same idea presented before when talking about meaningful names.

Functions

- Functions do or answer something and can have **inputs** and/or **outputs**. The number of **function arguments should be minimised**.

  The fewer the arguments the simplest it is to interpret.

- **Arguments are even harder from a testing point of view** (as we'll see later in this presentation).

  It is difficult to write all the test cases to ensure that all combinations of arguments work properly.

Functions

- **Avoid** using **flag arguments**.

Flag arguments are **booleans** (True or False) that are sometime used to tell the function to do something if it is true or something else if it is false.

This violates one of the main rules; **functions should do only one thing**. Most of the times, creating two different functions does the job just fine.

Functions

- If a function truly needs **more than two or three arguments** it is likely that some of them can be **wrapped together**.

  In Matlab this can be easily done using **structures**.

  In object-oriented programming, objects such as **classes** are usually used to group function arguments.

- **No side effects**.

  Side effects are lies; they promise to do one thing but it also does other hidden things (e.g., unexpected changes to variables).

Functions

- **Don’t repeat yourself.**

  Code duplication is by definition inefficient. It will require several modifications in several places if a change is required.

- Since functions should be small and only do one thing, they should have **clear starting and ending points**.

  Multiple `return`, `break` or `continue` statements are sometimes useful but they must be avoided if possible.

Functions

- **Writing software is like any other kind of writing**.

  The first draft may be clumsy, disorganised and difficult to read. Then, next iterations become cleaner, with less arguments, splitting out functions, eliminating duplication, etc.

- **Don’t over do it** (during developing stage).

  Reducing a function to the minimal expression is the objective as long as it is still descriptive. However, some extremely optimised codes can be difficult to understand.

```python
def create_user(email, password):
    try:
        user = User.objects.get(email=email)
    catch UserDoesNotExist:
        user = User.objects.create(
            email=email,
            password=password
        )
        auth_valid = authenticator(
            user=user,
            password=password
        )
        if auth_valid:
            user.last_login = datetime.now()
            user.save()
            msg_content = '<h2>Welcome {email}</h2>'.format(
                email=email
            )
            message = MIMEText(
                msg_content,
                'html'
            )
            message.update({
                'From': 'Sender Name <sender@server>',
                'To': email,
                'Cc': 'Receiver2 Name <receiver2@server>',
                'Subject': 'Account created'
            })
            msg_full = message.as_string()
            server = smtplib.SMTP('smtp.gmail.com:587')
            server.starttls()
            server.login(
                'sender@server.com',
                'senderpassword'
            )
            server.sendmail(
                'sender@server.com',
                [email],
                msg_full
            )
            server.quit()
            return user
```

What this function does ?

- If the user doesn't exist, it creates a new user.
- It creates an HTML Template.
- It initializes a SMTP connector.
- It sends an email.

Why it is wrong ?

1. Is doing more than One Thing.
2. How to test if the function works?
3. It's easier to introduce errors
4. Long function… losing readiness.
5. More things to do, it's harder to debug.

How to fix ?

- Split it in functions doing one thing:

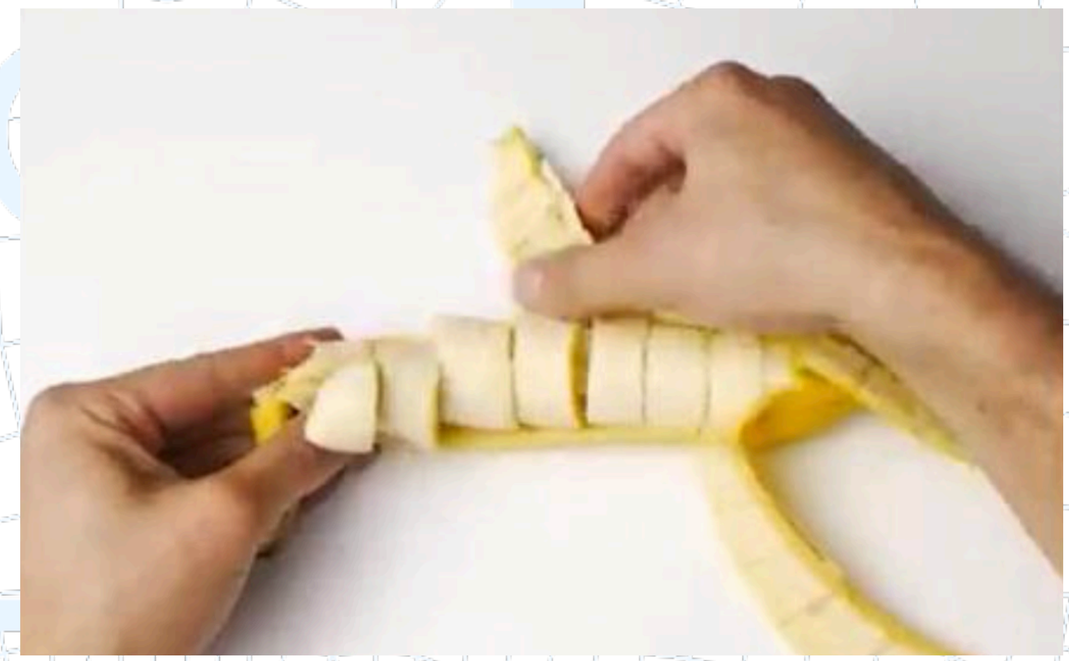
```
def signup(email, password):
```

```
def create_user(email, password):
```

```
def authenticate(user, password):
```

```
def send_welcome_email(email):
```

```
def send_email_to(email, email_msg):
```



Coding workflow tips

Coding workflow tips

- **Goal**: encourage you to do

1. Versioning
2. Testing

Versioning

- Versioning keeps track of the modifications you do in your code.

- Allows for several people programing in the same project simultaneously (very important but I am not interested in this aspect today.

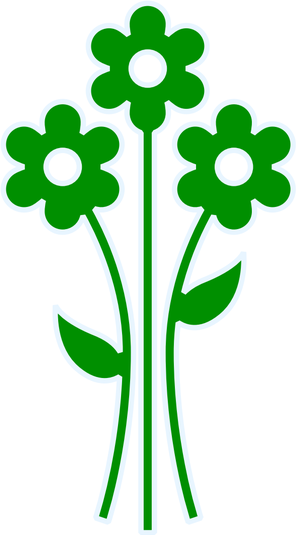
- **Force you to modify your code in an organized way!**

Versioning

- **Bad ideas to keep the history** (that we all did! 🙄)
  - commenting old code and leave it there… just in case.
  - coping files with _v0, _v1, _v3 suffix…
  - creating a v0, v1, v2 folder with the complete code…

Versioning

- **One solution: Versioning your code!**
  - It stores all the history of your code
  - It allows to track changes and locate when and where something got broken.
  - Importantly: it forces you to work in a more organized way…
  - 🙂😀😃😄😄



Versioning. How it works?

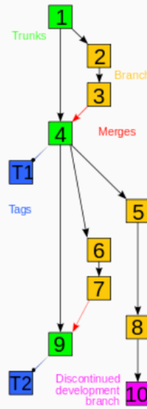
- **Usual workflow:**
  - You have your versioned code and need to do some modification.
  - You change the code.
  - You test your change.
  - When you are happy with the change,
    you tell the versioner to remember the actual version (commit).
    You must add a **description** for the modification you did.

Versioners

- git - https://git-scm.com
  - github / gitlab - https://www.mercurial-scm.org
  - subversion or svn - https://subversion.apache.org
  - mercurial - https://www.mercurial-scm.org

Just for you to know

V·T·E	Version control software	[hide]
Years, where available, indicate the date of first stable release. Systems with names *in italics* are no longer maintained or have planned end-of-life dates.		
Local only	**Free/open-source**	RCS (1982) · SCCS (1973)
	Proprietary	PVCS (1985) · *QVCS* (1991)
Client–server	**Free/open-source**	CVS (1986, 1990 in C) · CVSNT (1998) · QVCS Enterprise (1998) · Subversion (2000)
	Proprietary	AccuRev SCM (2002) · Azure DevOps (Server (via TFVC) (2005) · Services (via TFVC) (2014)) · ClearCase (1992) · *CMVC* (1994) · Dimensions CM (1980s) · *DSEE* (1984) · Endevor (1980s) · Integrity (2001) · Panvalet (1970s) · Perforce Helix (1995) · SCLM (1980s?) · Software Change Manager (1970s) · StarTeam (1995) · Surround SCM (2002) · Synergy (1990) · Team Concert (2008) · Vault (2003) · *Visual SourceSafe* (1994)
Distributed	**Free/open-source**	*ArX* (2003) · BitKeeper (2000) · Breezy (2017) · Darcs (2002) · *DCVS* (2002) · Fossil (2007) · Git (2005) · *GNU arch* (2001) · GNU Bazaar (2005) · Mercurial (2005) · Monotone (2003)
	Proprietary	Azure DevOps (Server (via Git) (2013) · Services (via Git) (2014)) · *TeamWare* (1992) · *Code Co-op* (1997) · Plastic SCM (2006)
Concepts	Baseline · Branch · Changeset · Commit · Data comparison · Delta compression · Fork (Gated commit) · Interleaved deltas · Merge · Monorepo · Repository · Tag · Trunk	
Category · **Comparison** · **List**		https://en.wikipedia.org/wiki/Distributed_version_control



→ Tell me what files did I change?

```
MBP-2020:algebraicPGD zlotnik$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   pgdLinearSolve.m
```

→ You changed this file…

Tell me the modifications I did…

```
MBP-2020:algebraicPGD zlotnik$ git diff pgdLinearSolve.m
diff --git a/pgdLinearSolve.m b/pgdLinearSolve.m
index c73d5d3..2e4ebdf 100644
--- a/pgdLinearSolve.m
+++ b/pgdLinearSolve.m
```

```
@@ -110,6 +127,7 @@
 addParameter(p, 'tolTruncateLastTerm1', 1E-12);
 addParameter(p, 'tolTruncateLastTermE', 1E-3);
 addParameter(p, 'verbose', true);
+addParameter(p, 'paramOrder', []);
 parse(p,varargin{:});
 x0          = p.Results.x0;
 maxModes    = p.Results.maxModes;
```

→ You added this line…

```
@@ -374,7 +408,7 @@


       %% param
-      for jdim = 2:ndim
+      for jdim = getDimOrder(paramOrder, iMode, iter, ndim) %2:ndim

           alpha = zeros(dimSize(jdim), 1);
```

→ … and changed this other line

- Ok, save this version in the history…

```
[MBP-2020:algebraicPGD zlotnik$ git add pgdLinearSolve.m
[MBP-2020:algebraicPGD zlotnik$ git commit -m "add option for dimension solve order"
 [master 0367581] add option for dimension solve order
  1 file changed, 74 insertions(+), 1 deletion(-)
```

… this is the description of the modification saved.

```
MBP-2020:algebraicPGD zlotni$ git log
commit 0367581de78b5b63c21ad[illegible]44093b4 (HEAD -> master)
Author: sergio.zlotnik@upc.edu <sergio.zlotnik@upc.edu>
Date:   Sun Jan 23 20:25:05 2022 +0100

    add option for dimension solve order

commit fc4e398808a12605a4a9ae239cf010f84da4faa4 (origin/master)
Author: sergio.zlotnik@upc.edu <sergio.zlotnik@upc.edu>
Date:   Mon Jan 10 09:45:30 2022 +0100

    add cell2sepataredTensor

commit 6fd783d9f76c26641e5aa77e638ebb391dc01809
Author: sergio.zlotnik@upc.edu <sergio.zlotnik@upc.edu>
Date:   Fri Oct 8 10:56:58 2021 +0200

    fix problem with deleteRowsDBC

commit bd9ddb203c74d805678ca39baeb7192755cae94d
Author: sergio.zlotnik@upc.edu <sergio.zlotnik@upc.edu>
Date:   Wed Sep 22 13:17:29 2021 +0200

    fix stopping criteria in compression

commit 6cf224adc8c781845435338a9713e705b0b68958
Author: sergio.zlotnik@upc.edu <sergio.zlotnik@upc.edu>
Date:   Sun Jul 25 19:38:46 2021 +0200

    isMatrix

commit 051b2be2a3ff3dab5389124f2cc7cd3d7bc16e30
Author: sergio.zlotnik@upc.edu <sergio.zlotnik@upc.edu>
Date:   Tue Jun 22 11:40:43 2021 +0200

    error message

commit ee310e5e203bf9bfe142a8ad0b1d685fff0fe6f2
Author: sergio.zlotnik@upc.edu <sergio.zlotnik@upc.edu>
Date:   Thu Mar 25 10:29:50 2021 +0100

    comment

commit ec0a116b913bf162f53800da6f0ac3a6f0d9d1d1
Author: sergio.zlotnik@upc.edu <sergio.zlotnik@upc.edu>
Date:   Thu Mar 25 10:25:48 2021 +0100

    formating code
```

Tell me what happened recently with the code

Testing…

IDEA: We want to try our code before committing a change in the history… every time…

Definition

- *Testing*: an investigation conducted to provide information about the quality of the code under test.
- Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs.
- Testing **cannot** identify all the defects within software

Types of testing

- **Unit testing**: component/function testing
- **System testing**: test a completely integrated system
- **Regression testing**: find defects after major code modifications.
- Many other types of tests exists. Ton of material available on the Internet.

Basic testing tips

- Some basic tips that I use to test functions or operations.

1. Use some analytical result or a previous trusted result stored in a file.
2. Create a small program that calls the tested function and compares the result with the expected.
3. If difference above some tolerance, test fails.

Basic testing tips

- Some basic tips that I use to test functions or operations.
- Better something than nothing…
  - Make tests that run fast. Otherwise you will be lazy to run them.
    - Separate fast and slow test.
  - Don’t worry about listing all errors, just abort assert() is your friend.

Example

```
1  function test002() % ST + c
2
3  tol = 1e-10;
4  ndim    = randi([2 5], 1, 1);
5  nmodesA = randi([1 10], 1, 1);
6  dimSize = randi([10 20], 1, ndim);
7  c       = rand(1) * 10;
8
9  A = separatedTensor;
10 A.sigma = rand(nmodesA, 1);
11 for iDim = 1:ndim
12    for iMode = 1:nmodesA
13       A.sectionalData{iDim,iMode} = rand(dimSize(iDim), 1);
14    end
15 end
16 fullA = fullTensor(A);
17
18 S = A + c;
19 fullS = fullTensor(S);
20
21 maxS    = max(abs(fullS(:)));
22 diff    = abs(fullS - (fullA + c));
23 maxDiff = max(diff(:));
24 err     = maxDiff / maxS;
25
26 assert(err < tol, ...
27    'Error in A+c, got error of %e with a tolerance of %e', err, tol)
28 end
```

Setup test.
Including some random variables

Run operation

Compute difference

Abort if not passed

Main testing code…

```
test_addition
test_division
test_evaluate
test_LSE
test_product
test_separation
test_compression
test_sqrt
test_reshape
test_index
test_normalization
test_diag
test_full
test_sparse
test_separatedTensorOperations

disp('Don''t worry be happy, baby :)')
```

You could have two testing suites:

- run_test_fast
- run_test_full

Basic testing tips

- Try to test every main operation / option of your code.
- New features should have a new testing

Summary

- **Versioning**
  - Allows you to keep and manage history.
  - Force you to be organized.
- **Testing**
  - Even in the simplest form of testing provides some degree of early detection of error.
- **Use them!**
  - your coding-life will get easier…