



Programming

for Today's Physicists and Engineers

ISOTDAQ 2019
Royal Holloway University London

April 4, 2019
Alessandro Thea
Rutherford Appleton Laboratory - PPD



Science & Technology
Facilities Council



Opening words

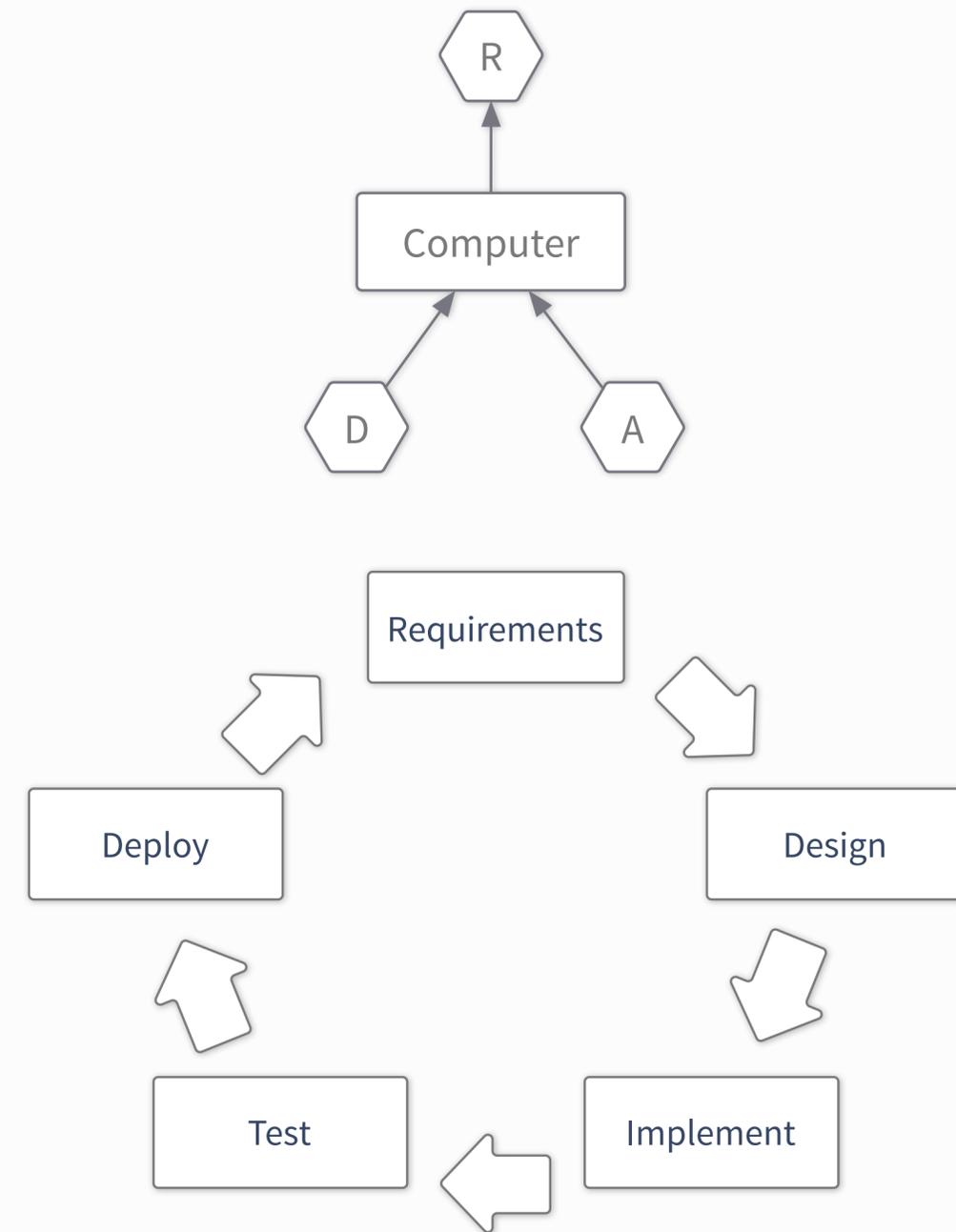
Disclaimer: This is more a collection of pointers* than a tutorial, it's a starting point...

(Almost) no code but a bias towards C++ and Python

Acknowledgment: Slides are based on previous lectures by Joschka Poettgen (Lingemann) and Erkcan Ozcan

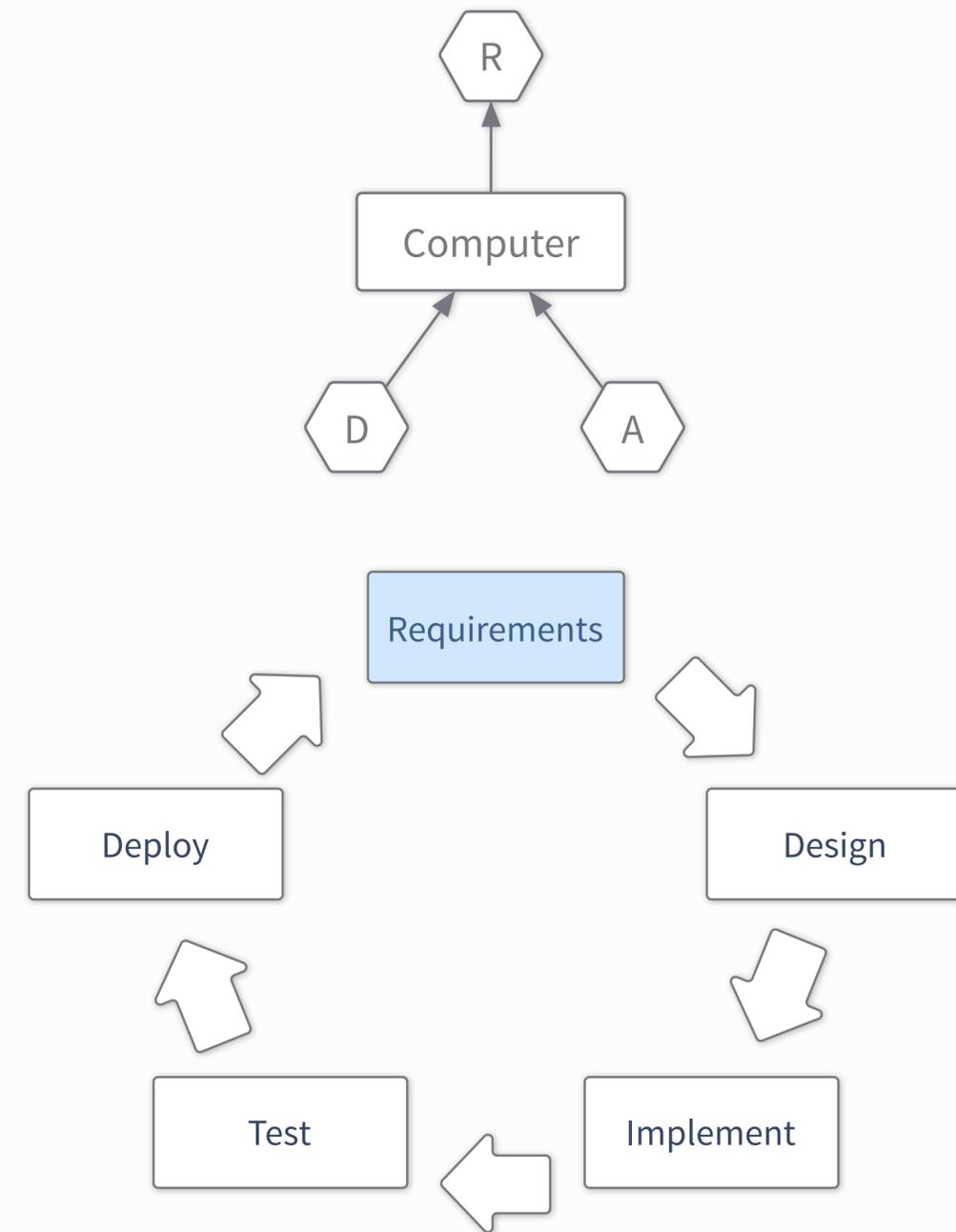
*further reading and tips in these boxes

What is programming?



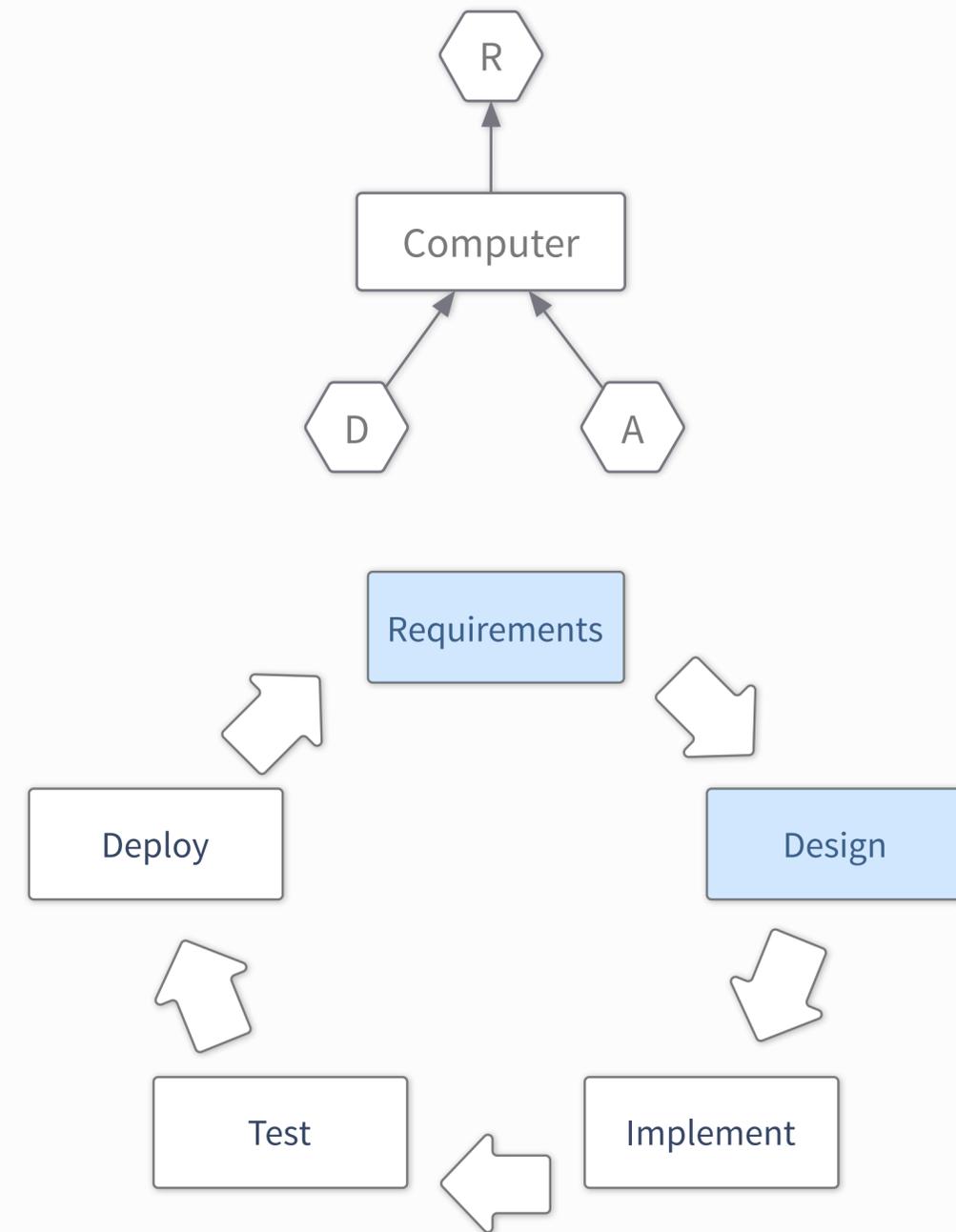
What is programming?

- **Understand** & define what you want to solve
 - ▶ Define the **requirements** for your software



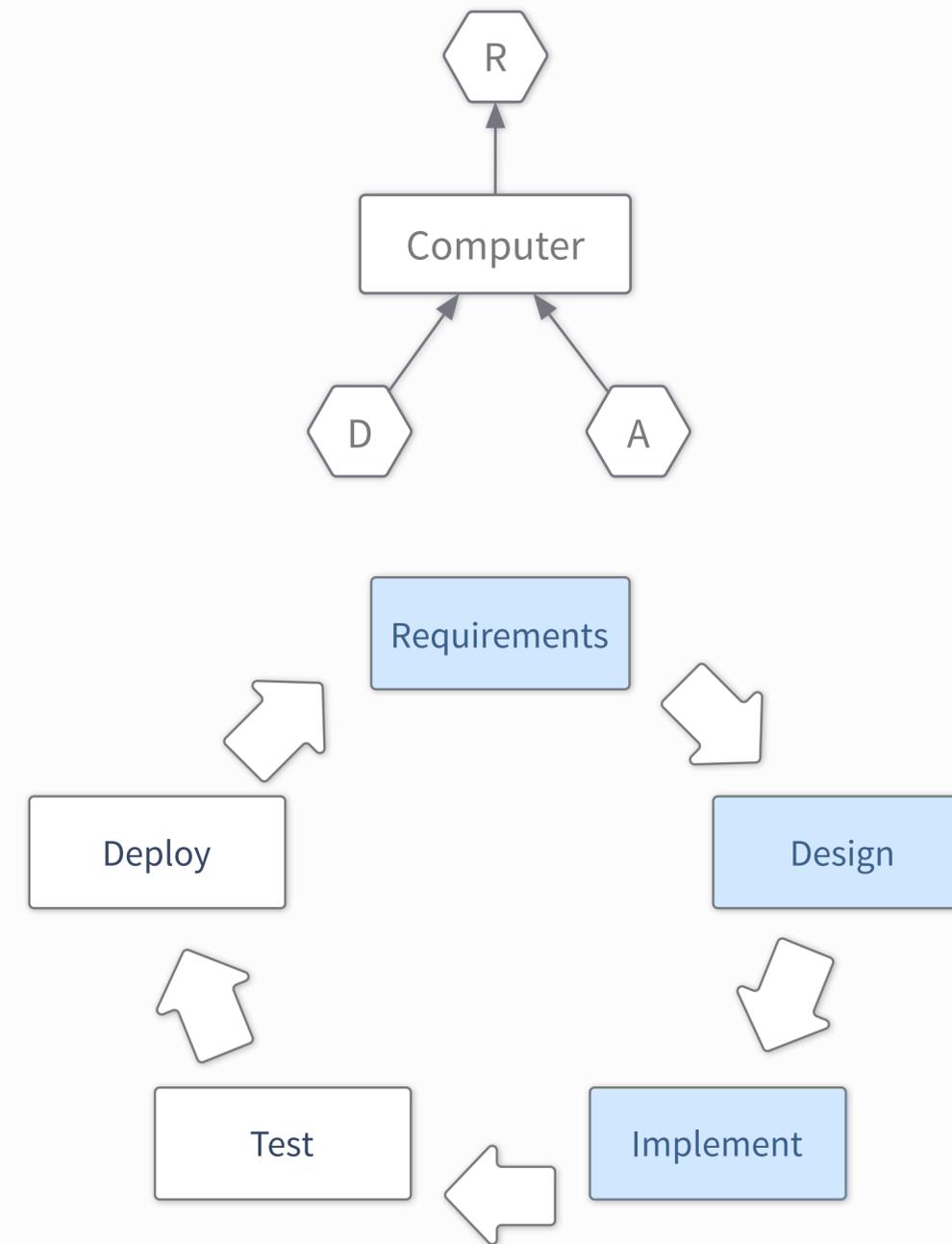
What is programming?

- **Understand** & define what you want to solve
 - ▶ Define the **requirements** for your software
- Formulate a **possible solution**



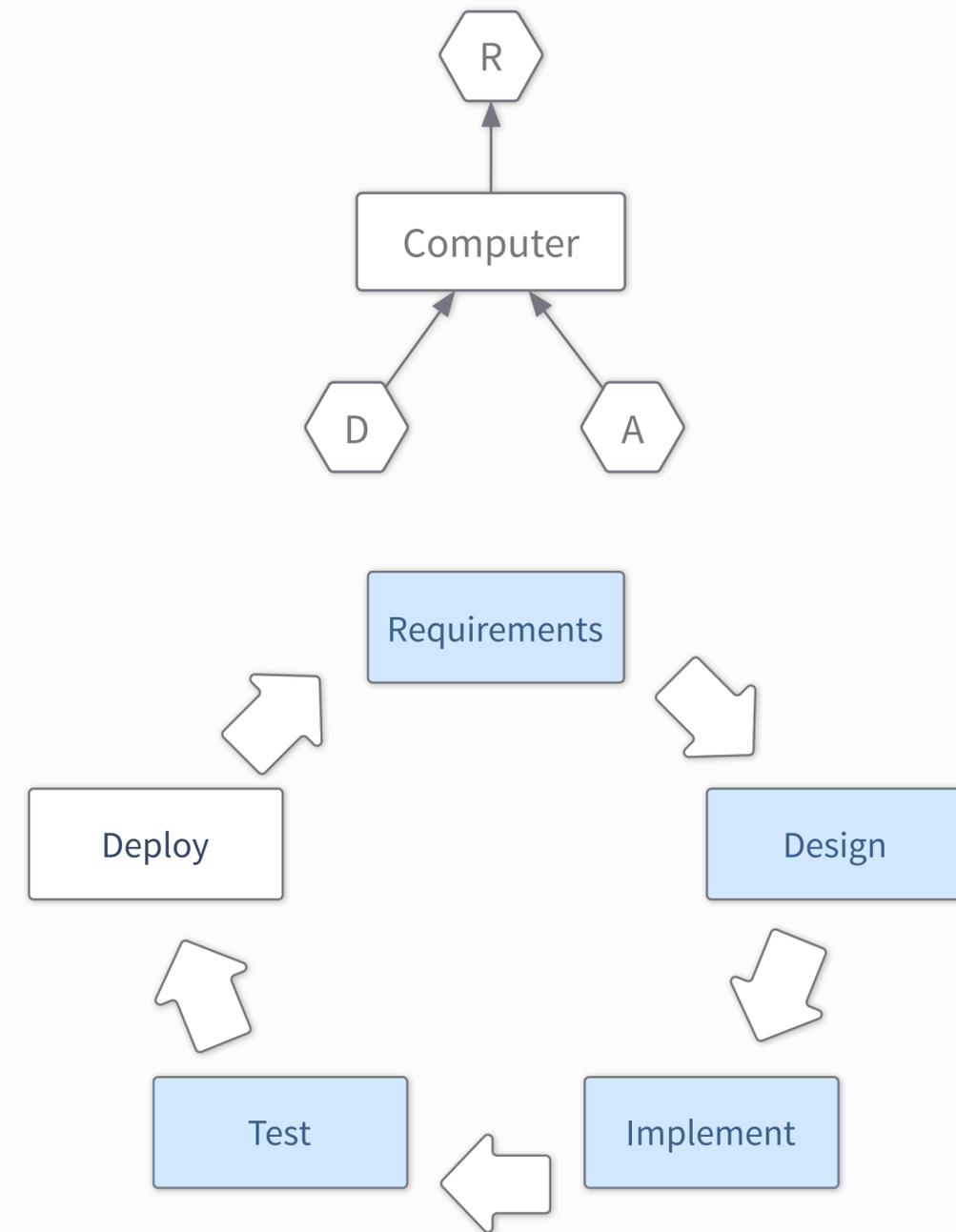
What is programming?

- **Understand** & define what you want to solve
 - ▶ Define the **requirements** for your software
- Formulate a **possible solution**
- **Implement** it
 - ▶ What the language?
 - ▶ Debug
 - ▶ Unit-test development
 - ▶ Documentation



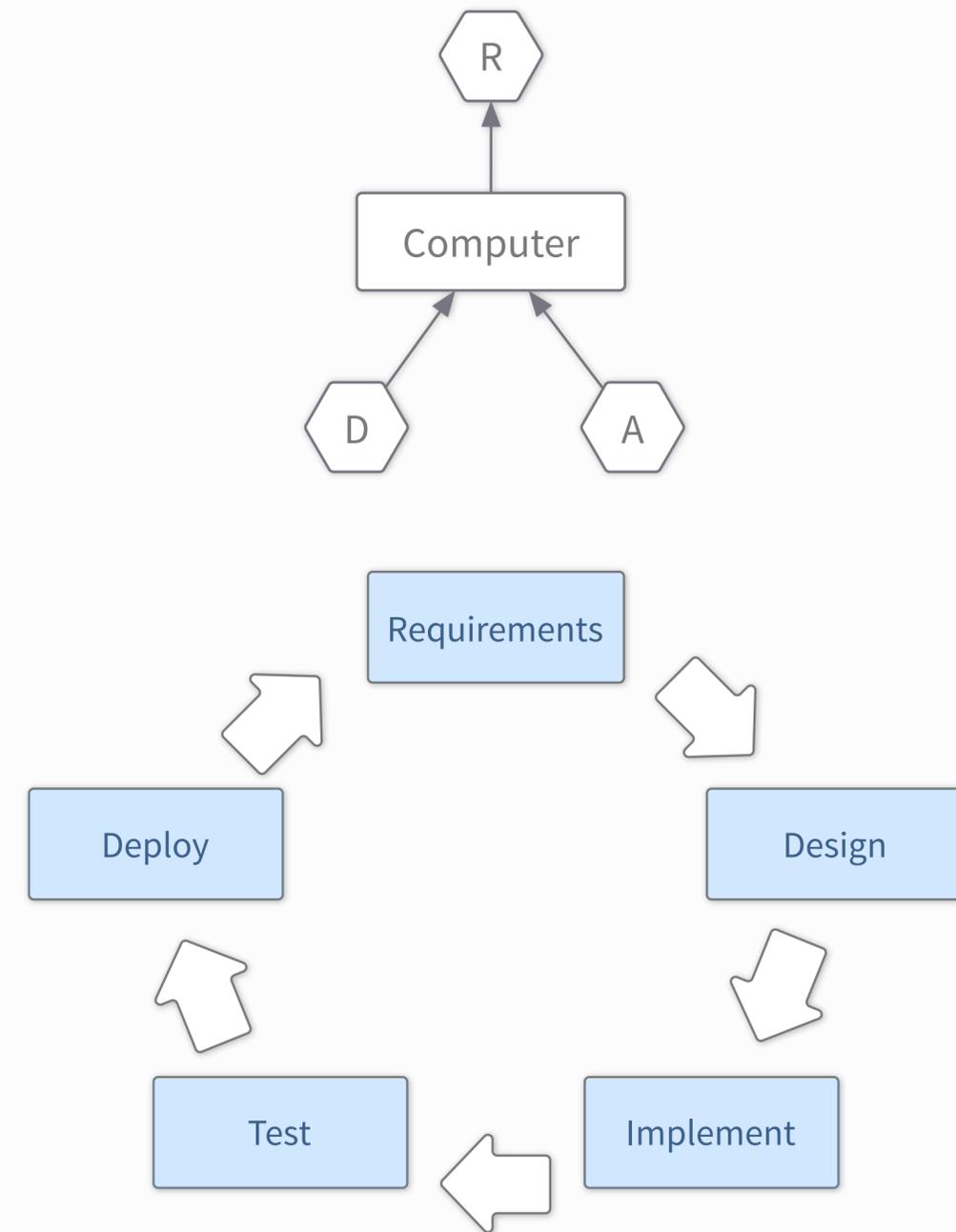
What is programming?

- **Understand** & define what you want to solve
 - ▶ Define the **requirements** for your software
- Formulate a **possible solution**
- **Implement** it
 - ▶ What the language?
 - ▶ Debug
 - ▶ Unit-test development
 - ▶ Documentation
- **Validate**
 - ▶ Verification and system tests

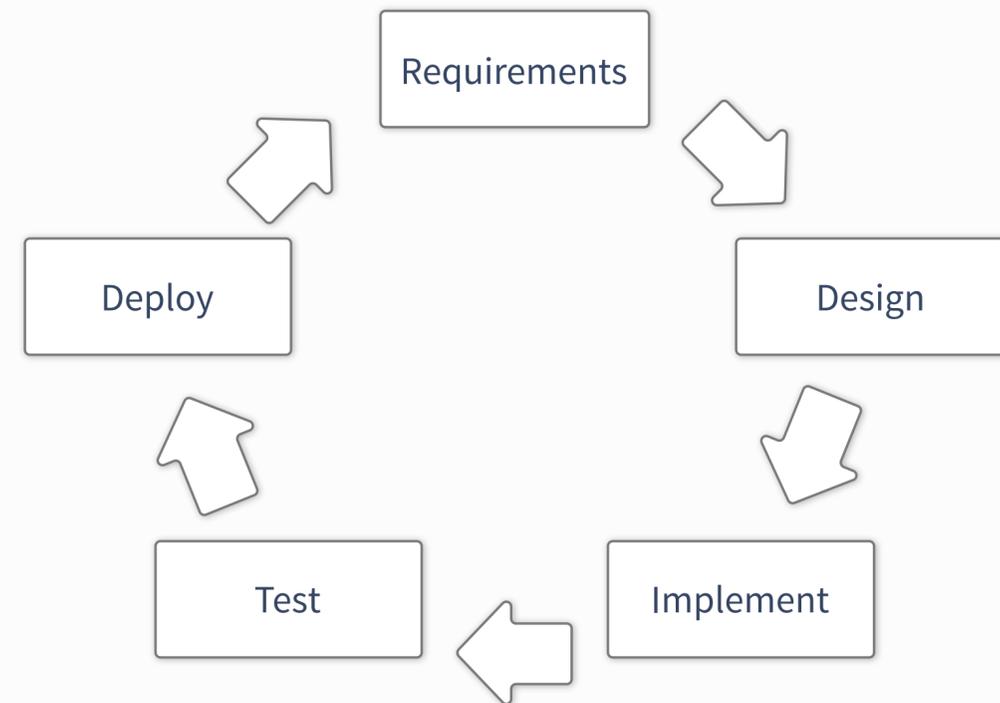


What is programming?

- **Understand** & define what you want to solve
 - ▶ Define the **requirements** for your software
- Formulate a **possible solution**
- **Implement** it
 - ▶ What the language?
 - ▶ Debug
 - ▶ Unit-test development
 - ▶ Documentation
- **Validate**
 - ▶ Verification and system tests
- **Deliver** the code
 - ▶ Collect feedback
 - ▶ Portability to different platforms?
- And then back to square 1

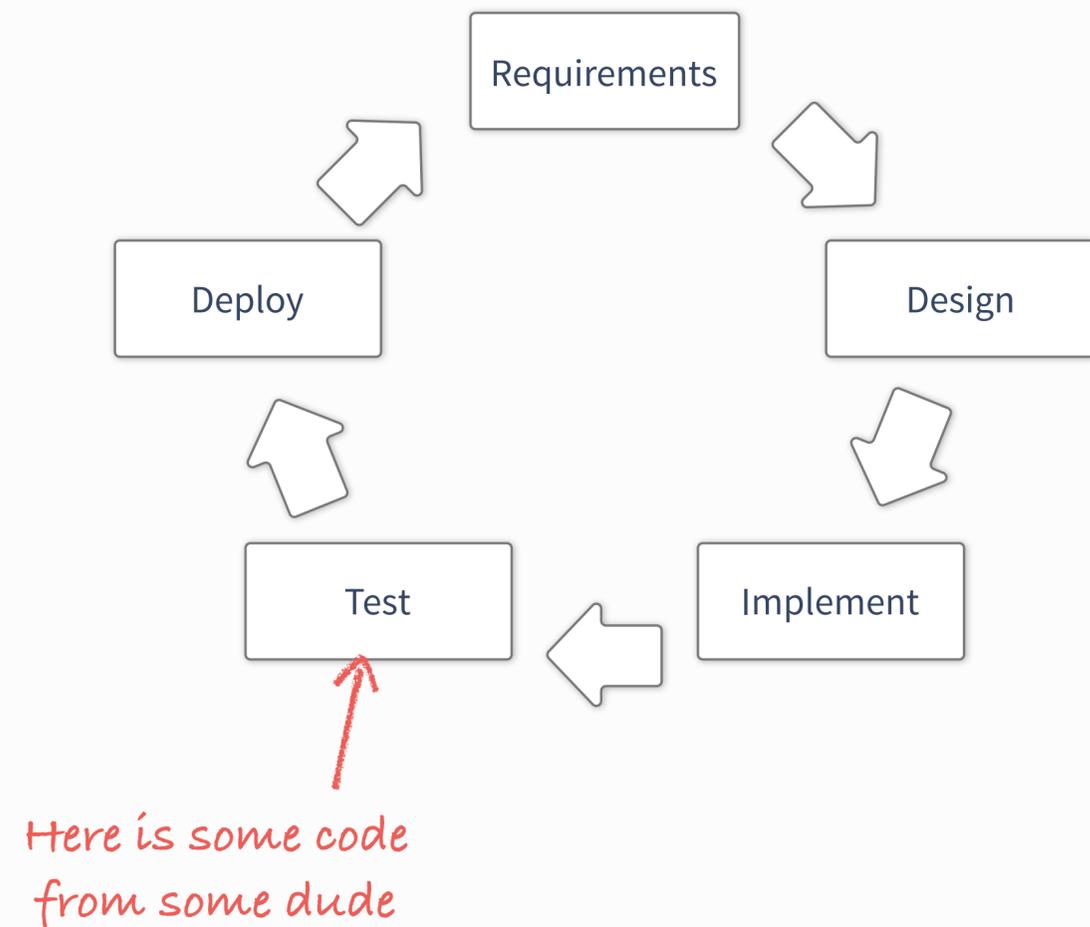


What programming is really like:



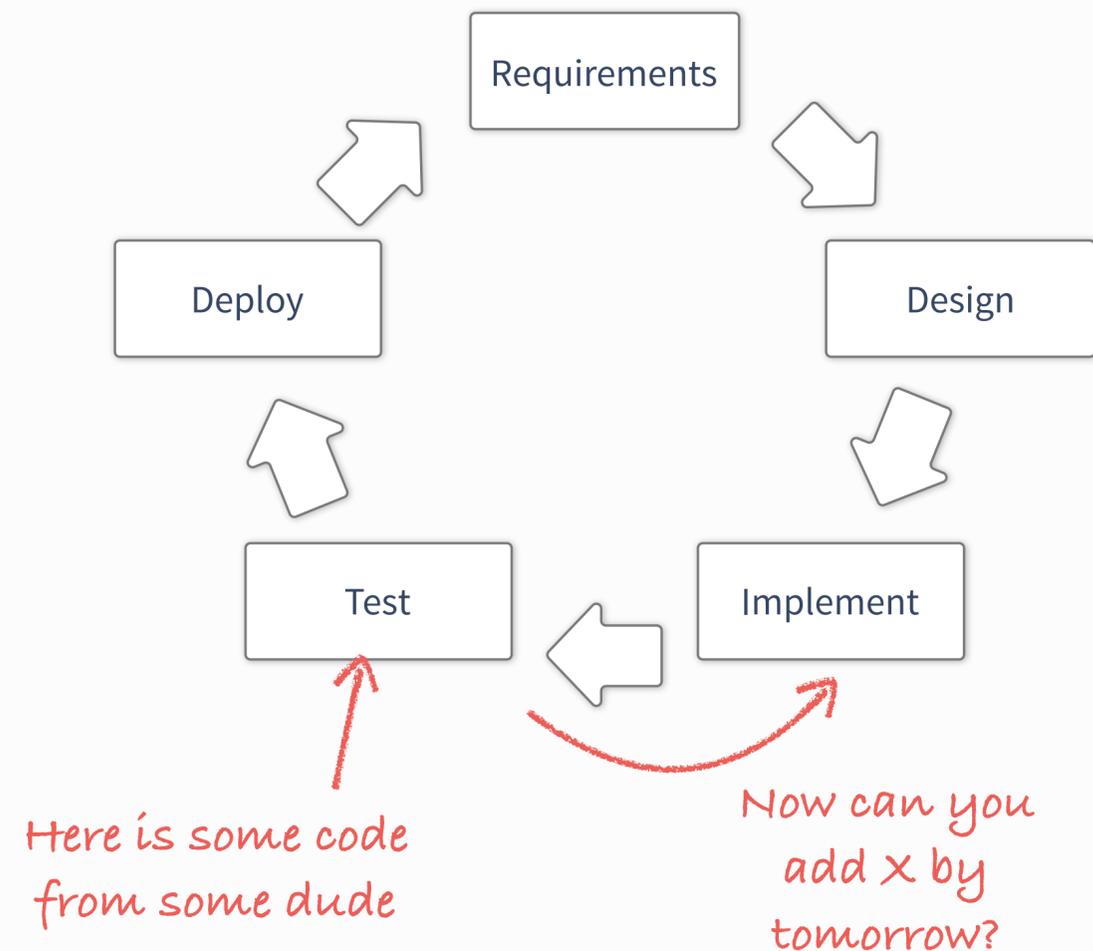
What programming is really like:

- Inherit **some code**
 - ▶ Run some tests to get the hang of it



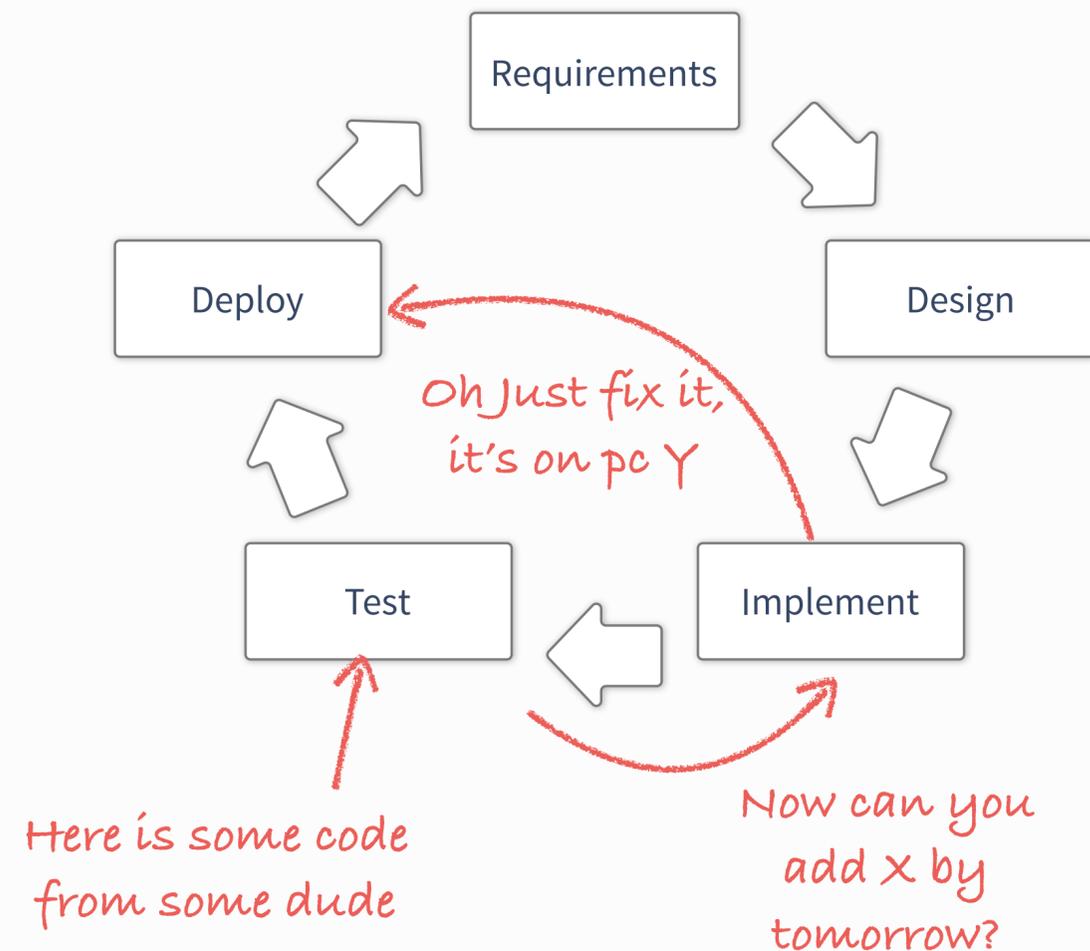
What programming is really like:

- Inherit **some code**
 - ▶ Run some tests to get the hang of it
- Add **some features**
 - ▶ whose purpose is not always completely clear
 - ▶ by patching some files



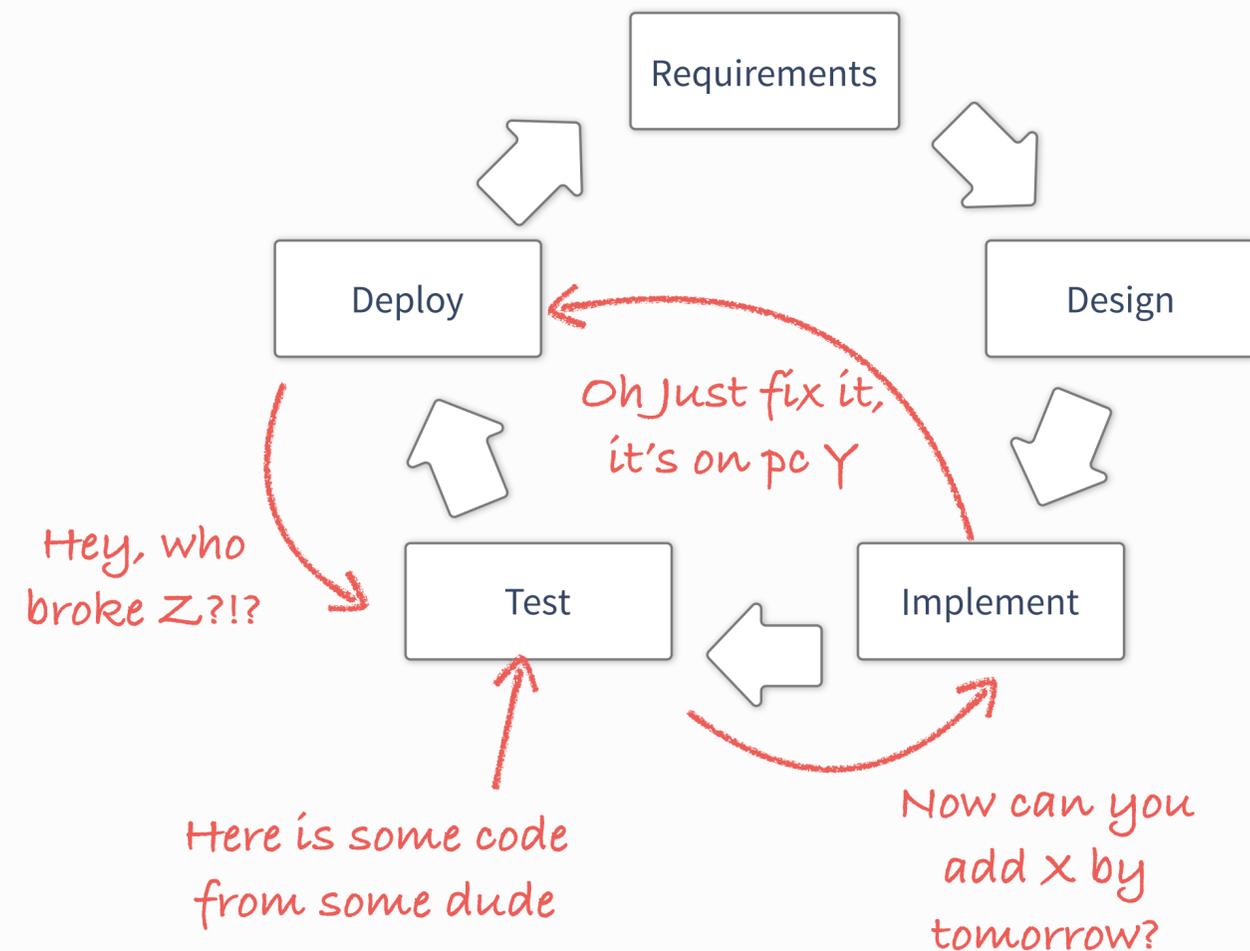
What programming is really like:

- Inherit **some code**
 - ▶ Run some tests to get the hang of it
- Add **some features**
 - ▶ whose purpose is not always completely clear
 - ▶ by patching some files
- On the **only working system**
 - ▶ well, it's the only place where the code runs, isn't it?



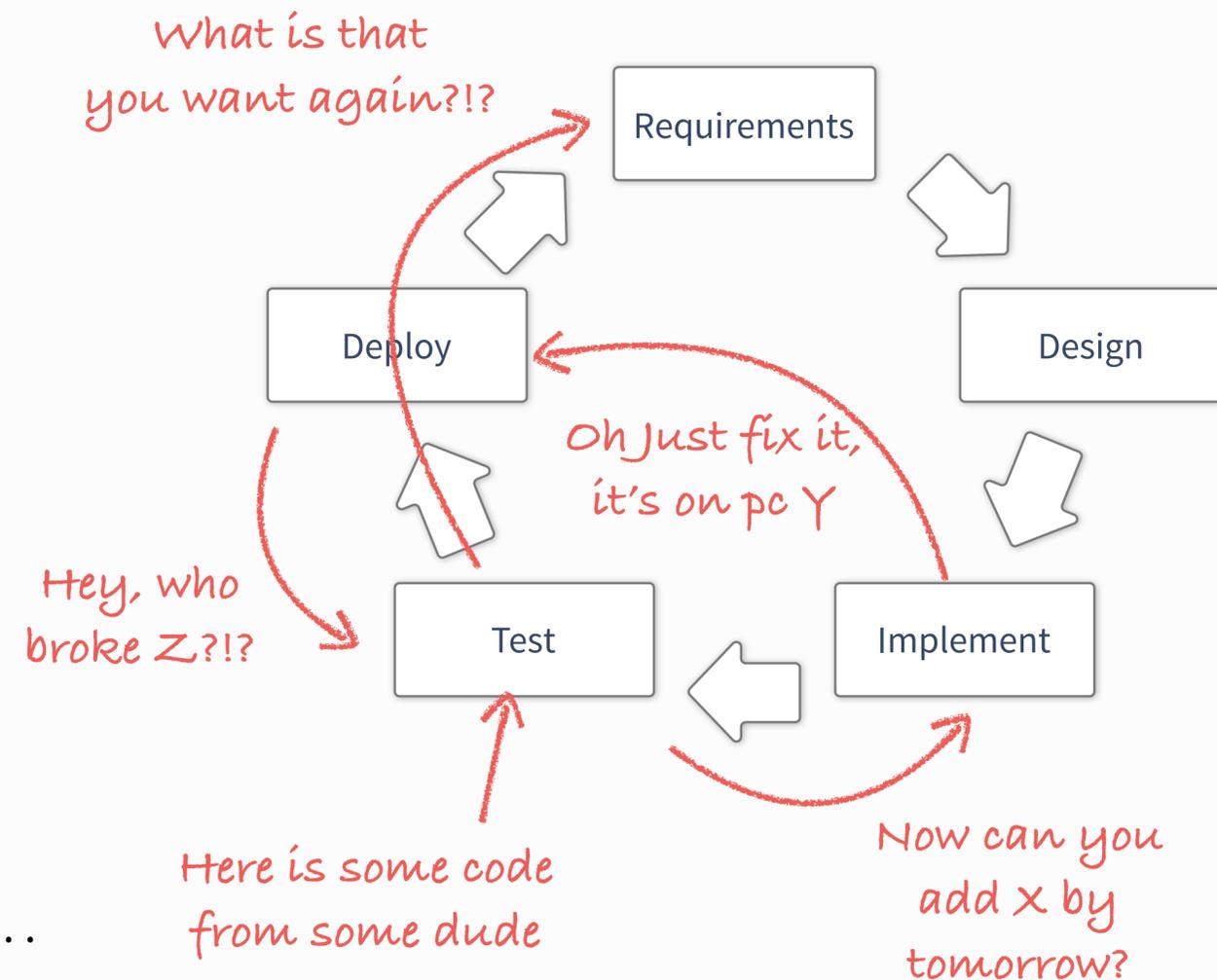
What programming is really like:

- Inherit **some code**
 - ▶ Run some tests to get the hang of it
- Add **some features**
 - ▶ whose purpose is not always completely clear
 - ▶ by patching some files
- On the **only working system**
 - ▶ well, it's the only place where the code runs, isn't it?
- **Break some other code** by accident
 - ▶ Desperately try to figure out why.



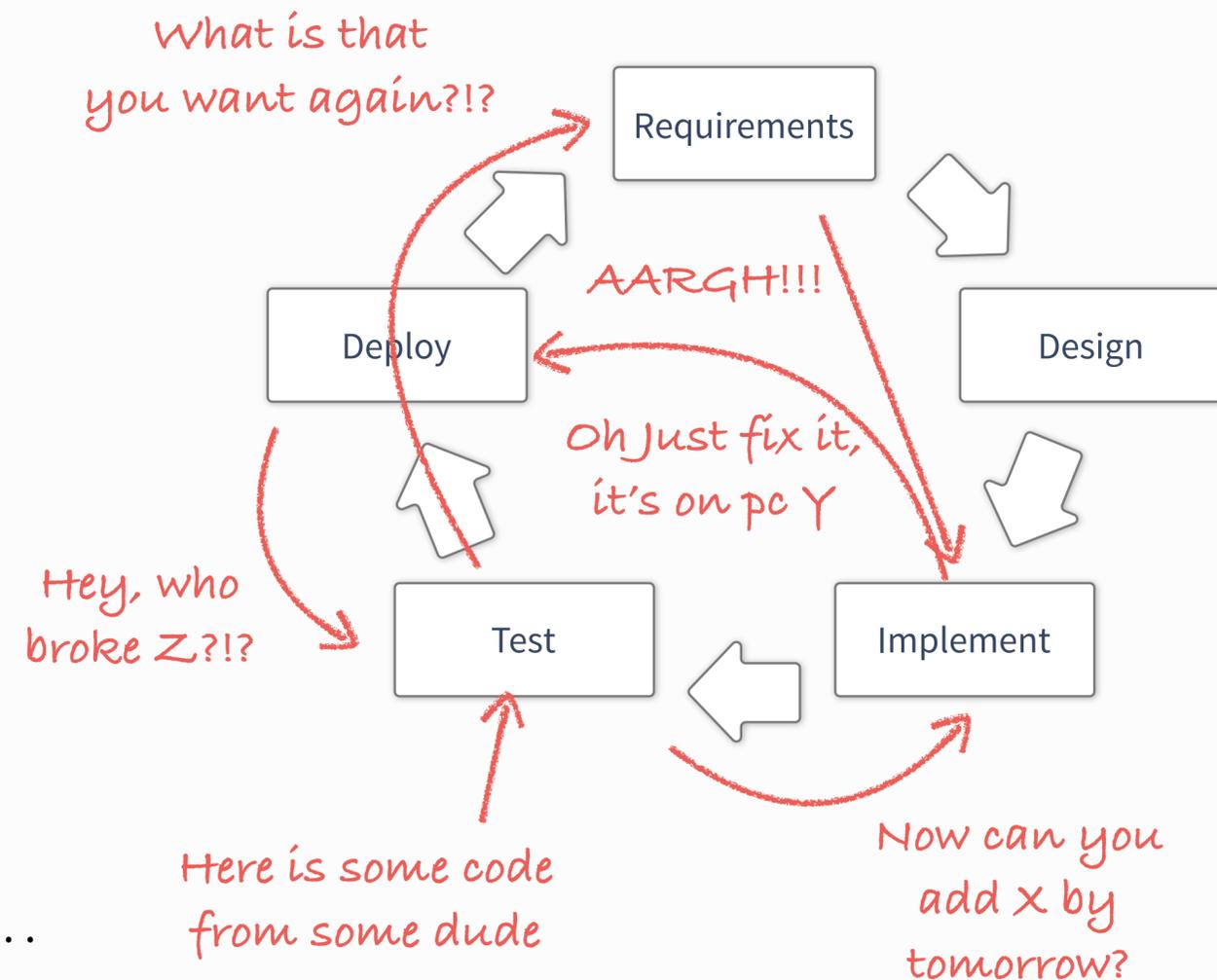
What programming is really like:

- Inherit **some code**
 - ▶ Run some tests to get the hang of it
- Add **some features**
 - ▶ whose purpose is not always completely clear
 - ▶ by patching some files
- On the **only working system**
 - ▶ well, it's the only place where the code runs, isn't it?
- **Break some other code** by accident
 - ▶ Desperately try to figure out why.
- Finally realise you **got it wrong** in the first place...

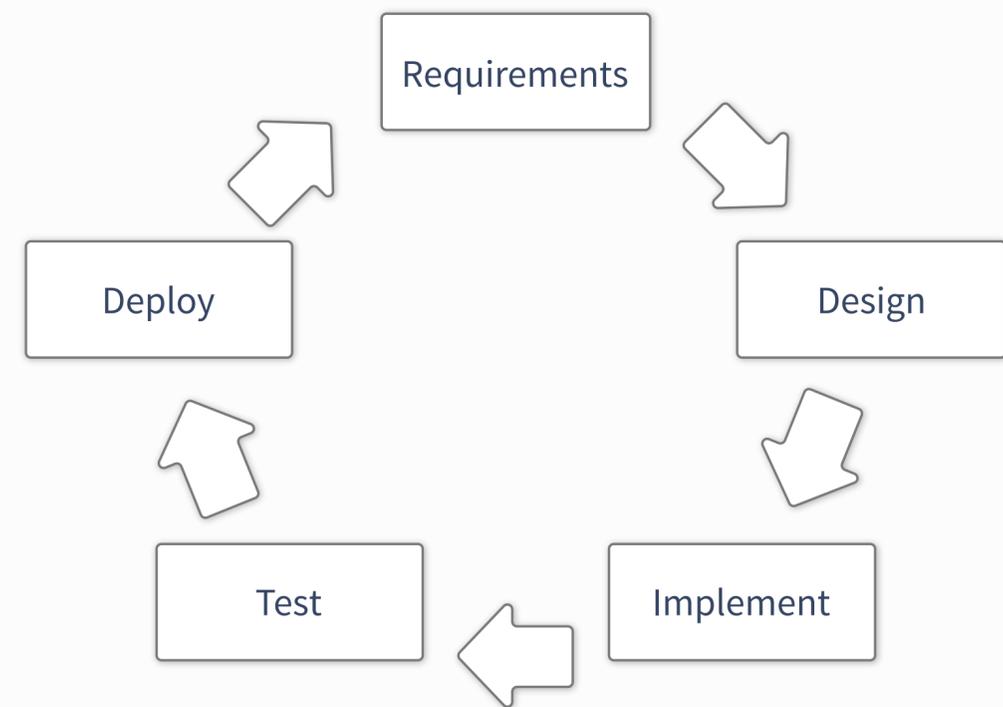


What programming is really like:

- Inherit **some code**
 - ▶ Run some tests to get the hang of it
- Add **some features**
 - ▶ whose purpose is not always completely clear
 - ▶ by patching some files
- On the **only working system**
 - ▶ well, it's the only place where the code runs, isn't it?
- **Break some other code** by accident
 - ▶ Desperately try to figure out why.
- Finally realise you **got it wrong** in the first place...
 - ▶ and so on and so on...



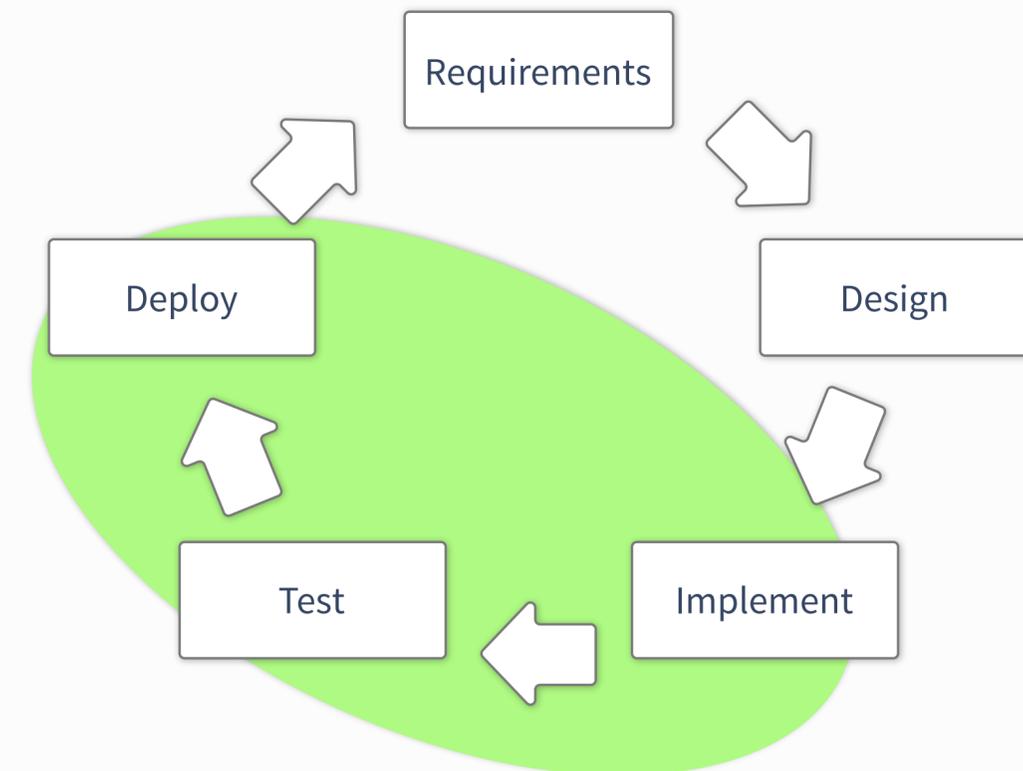
(some) Typical programming contexts



(some) Typical programming contexts

Small projects

- Shortened dev-cycle: Implement, Test, Deploy
 - Requirements and design already defined
- Mostly self contained
 - no /few external interfaces and dependencies
- Few developers (typically 1)



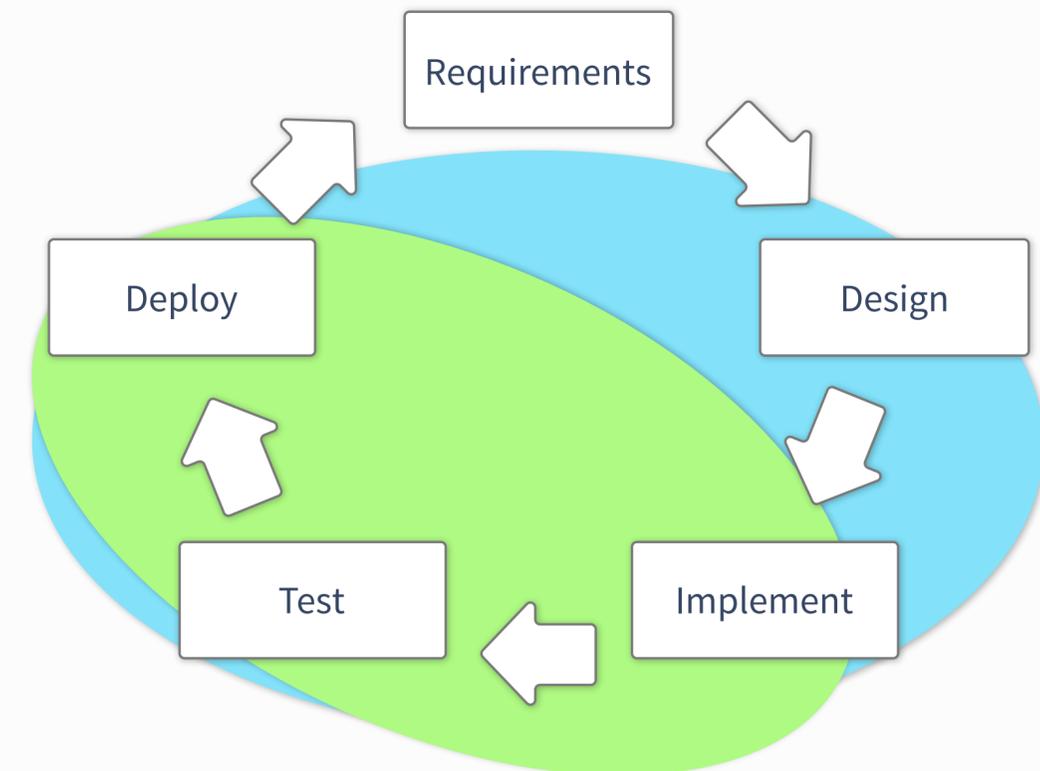
(some) Typical programming contexts

Small projects

- Shortened dev-cycle: Implement, Test, Deploy
 - Requirements and design already defined
- Mostly self contained
 - no /few external interfaces and dependencies
- Few developers (typically 1)

Medium projects

- Design becomes unavoidable
- Well defined interfaces and dependencies
 - e.g. external frameworks
- Many developers
- **Maintenance issues** make their appearance



(some) Typical programming contexts

Small projects

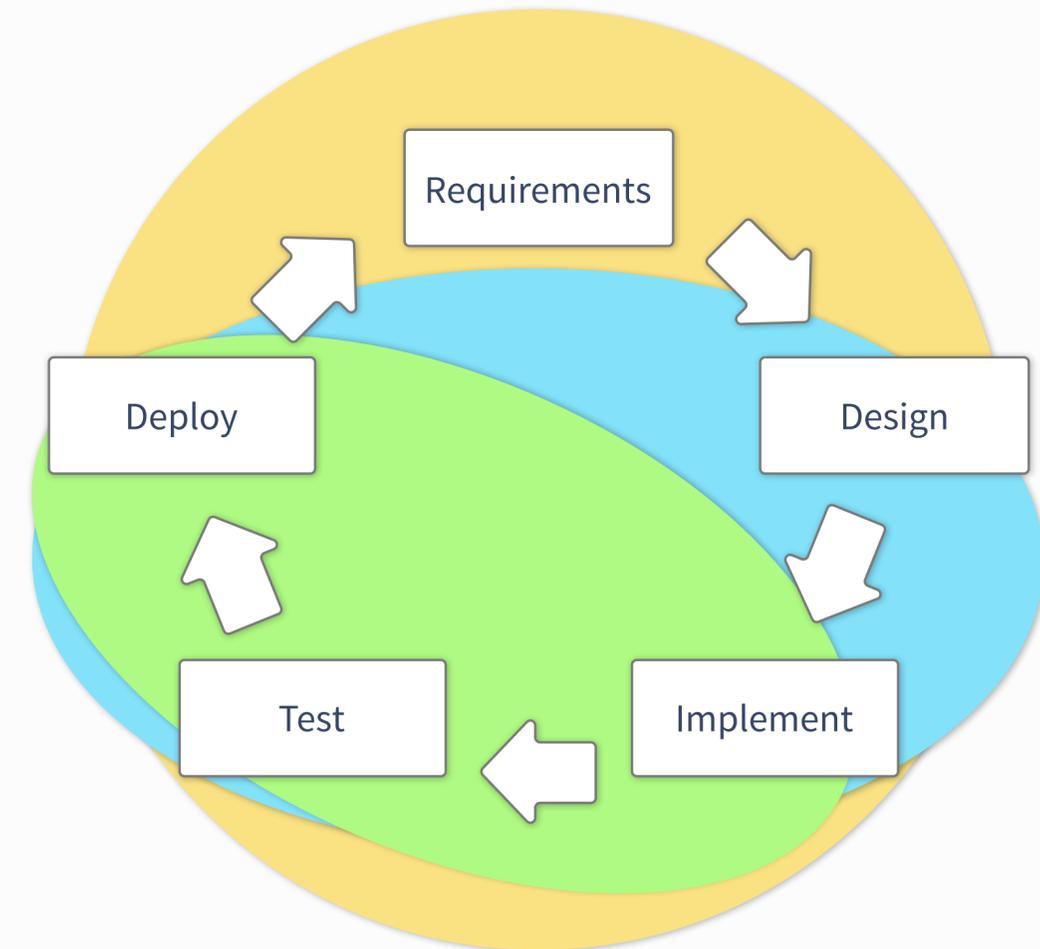
- Shortened dev-cycle: Implement, Test, Deploy
 - Requirements and design already defined
- Mostly self contained
 - no /few external interfaces and dependencies
- Few developers (typically 1)

Medium projects

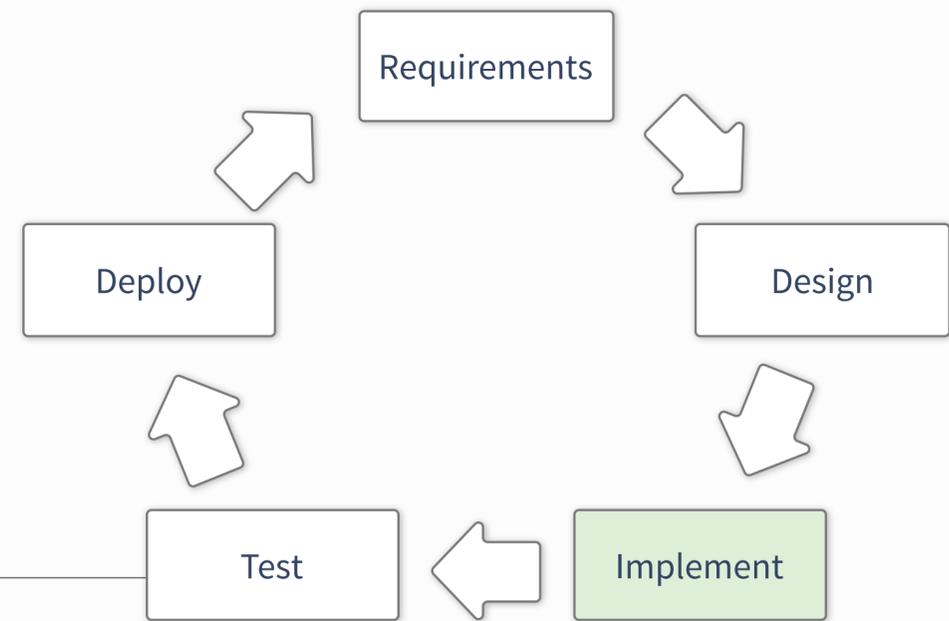
- Design becomes unavoidable
- Well defined interfaces and dependencies
 - e.g. external frameworks
- Many developers
- **Maintenance issues** make their appearance

Large projects (TDAQ)

- Requirements become crucial
- Many interfaces, complex dependencies
- Sizeable userbase
 - Support becomes your worst nightmare



Implementation



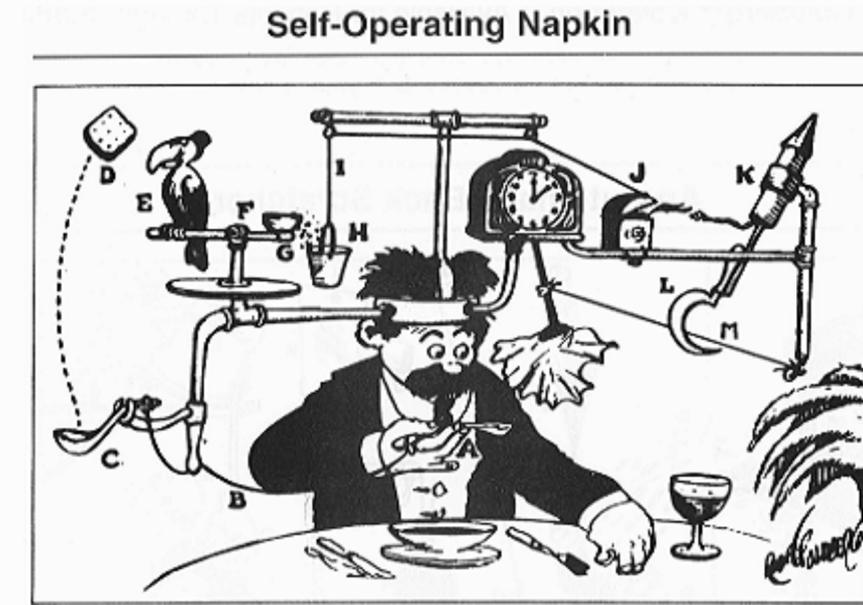
Look around for existing solutions

Do not reinvent the wheel

- Many problems have already been solved
- (Sometimes necessary — avoid dependencies)
 - ▶ Do not reject a library because of too many features
- Look for libraries where:
 - ▶ Active community? Well maintained? Tested?
 - ▶ Rule of thumb: Last commit a few days ago, most over a year old

Getting to know new frameworks:

- Try the simple tools and then ask for advice
 - ▶ Read the docs (RTFM)
 - Investing time in the beginning will pay off
 - ▶ Are there wikis? Has it been asked on StackOverflow?
 - ▶ python packages: try the ipython “help”



“Prof. Lucifer Butts and his Self-Operating Napkin”,
by Rube Goldberg

- Start with a simple test
(existing examples -> what you want to do)
 - ▶ Does the code do what you expect?

before looking at external libraries:
Look at the STL / python standard library

When coding - Avoid feature bloating

If you do everything at once:

- You'll probably end up doing nothing right
- **Write specialised tools / libraries**

Define features by writing a test that needs to be passed

- Do not implement more than you need to pass that test.

Be pragmatic

- Generalising a problem before solving it:
 - ▶ Probably not a good idea
 - ▶ Only do it when you have a use case
- Only do the abstract cases when it is likely that they will be used
- Try to make everything as concise as possible (better readability)
- **Keep it simple!**



**Don't reinvent
the wheel**

Tools of the Trade: Editor, Terminal and IDEs

Whatever you do, you'll end up using:

- Editor
 - ▶ Know* at least one “omnipresent” editor: **nano**, **vi (m)**, **emacs**, etc.
 - ▶ More modern solutions: have a number of clear benefits for development
 - ▶ Depending on the language / platform (e.g. Java): IDEs are the best choice **Eclipse**, **Netbeans**
- Terminal
 - ▶ Learn about shortcuts (minimal set: **tab**, **ctrl+r**, **ctrl+e**, **ctrl+a** ... have a look)
 - ▶ Knowing about some basic command line-tools will come in handy

* at least know how to save and exit :)
for the more daring: try **ed**

A few words on editors: Choose what suits you and be effective

The choice of editor is yours to make...¹

- Do you want “a great operating system, lacking only a decent editor”
- Or one with two modes: “beep constantly” and “break everything”²

Both are versatile and learning them is worthwhile

However: modern alternatives have a less-steep learning-curve

- Some are commercial ([Sublime Text](#), TextMate,...)
- Some are open: github’s [Atom](#) & Microsoft’s [VSCode](#)
 - Plugins, git integration, active communities, more plugins...

Once you decided which one is best for you:

- Spend some time learning about features and keybindings
- Many things that might require dozens of keystrokes can be done with 2 (5 in emacs ;))
- Learn about: Linters, extensibility — look at existing plugins

1. an insightful guide available here: [Text Editors in the Lord of the Rings](#)
2. from the [Editor war](#)



**Use what you find most comfortable
and learn to be efficient with it**

The Terminal - Get used to it

At the beginning might think: Quicker with GUI, don't need terminal

- After learning about some command line tools... probably not
- What if you don't have a GUI?

Searching files: grep, find — example:

```
$ find . -name "*.cc" -exec grep -A 3 "foo" {} +
```

- Displays all matches of “foo” (+3 lines below) in all .cc files from the current work dir

Once you learn some tools it becomes very versatile:

- **sed, head, tail, sort... awk** (a turing-complete interpreted language)
- At the beginning: note down often used commands...
- After a tutorial dump your history* (increase cache size for max usage)

Shell-scripting:

- Anything you do with the shell can just be dumped in a script
- Alternative: Can solve most things more conveniently with an interpreted language
 - ▶ Con: interpreters / bindings might not always be available

```
* dump the last 100 steps:  
$ history | tail -n 100 > steps.txt  
log the terminal “responses”:  
$ script # press ctrl+d to stop
```

```
tune your bashrc / bash-profile  
see additional material
```

Interlude: Working on the road — SSH

SSH — very, very versatile, more than you think:

- Tunneling
 - ▶ Secure connections to other machines
 - ▶ Use with VNC to avoid man-in-the-middle vulnerability
- Generate keys for authentication
- Working around bad latency / shaky connection
 - ▶ Always use **tmux/screen** or similar
 - ▶ Alternative: **mosh** (mosh.mit.edu)
 - mitigates intermittent connectivity, roaming or just moving to the next meetings...

SSHFS (AFS)

- Work locally but have files on remote host

SSH tunnel for VNC connection:

```
ssh -L 5902:<VNCServerIP>5902 <user>@<remote>\
  vncserver :<session> -geometry\
  <width>x<height> -localhost -nolisten tcp
```

SSH authentication via kerberos token. In ~/.ssh/config:

```
GSSAPIAuthentication yes
GSSAPIDelegateCredentials yes
HOST lxxplus*
  GSSAPITrustDns yes
```

Lots of things possible with the ssh-config:

```
HOST <host>
  USER <remote-user>
  ProxyCommand ssh <tunnel> nc <host> <port>
```

more on (auto-)tunnelling:

https://security.web.cern.ch/security/recommendations/en/ssh_tunneling.shtml

tmux guides and courses:

<https://robots.thoughtbot.com/a-tmux-crash-course>
<http://www.hamvocke.com/blog/a-quick-and-easy-guide-to-tmux/>

The right tool for many jobs - interpreted languages

Keep your code as short as possible

while maintaining readability

- Sometimes means to use the right language
- Often quicker / nicer: interpreted languages
 - ▶ python, perl, ruby, tcl, lua
- Used as binding languages:
 - ▶ Performance critical code in C/C++
 - ▶ Instantiated within python
(e.g. in CMS, ATLAS & LHCb offline Software)
 - ▶ Best of both worlds
- Python: large standard library & very expressive!

```
from __future__ import print_function
from argparse import ArgumentParser

parser = ArgumentParser(description="Get number of days")
parser.add_argument("month", type=str, nargs='+', help="Name of month")
args = parser.parse_args()

months = {"january": 31, "february": 28, "march": 31,
          "april": 30, "may": 31, "june":30,
          "july": 31, "august": 31, "september": 30,
          "october": 31, "november": 30, "december": 31}

for usermonth in args.month:
    if usermonth in months:
        print("{0} has {1} days.".format(usermonth, months[usermonth]))
    else:
        print("sorry. month '{0}' not known.".format(usermonth))
```

Keep it
easy to read

Easier to maintain; Easy to re-use

Interlude: iPython

```
ArrayType = class array(__builtin__.object)
| array(typecode [, initializer]) -> array
|
| Return a new array whose items are restricted by typecode, and
| initialized from the optional initializer value, which must be a list,
| string or iterable over elements of the appropriate type.
|
| Arrays represent basic values and behave very much like lists, except
| the type of objects stored in them is constrained.
|
| Methods:
|
| append() -- append a new item to the end of the array
| buffer_info() -- return information giving the current memory info
| byteswap() -- byteswap all the items of the array
| count() -- return number of occurrences of an object
| extend() -- extend array by appending multiple elements from an iterable
| fromfile() -- read items from a file object
| fromlist() -- append items from the list
```

Interlude: iPython

> ipython

```
In [1]: import array
```

```
In [2]: help (array)
```

```
In [3]: import ROOT
```

```
In [4]: help (ROOT.TH1D)
```

Interlude: iPython

```
class TH1D(TH1, TArrayD)
|   Method resolution order:
|   TH1D
|   TH1
|   TNamed
|   TObject
|   TAttLine
|   TAttFill
|   TAttMarker
|   TArrayD
|   TArray
|   ObjectProxy
|   __builtin__.object
|
|   Methods defined here:
|
|   AddBinContent(self, *args)
|       void TH1D::AddBinContent(int bin)
|       void TH1D::AddBinContent(int bin, double w)
```

Interlude: iPython

> ipython

```
In [1]: import array
```

```
In [2]: help (array)
```

```
In [3]: import ROOT
```

```
In [4]: help (ROOT.TH1D)
```

```
In [4]: run myscript.py
```

Documentation: Do it while it's fresh

Two sides of the same coin: Internal and external documentation

- Both necessary to make your programs easy to use
- They have different purpose!

Internal documentation:

- Explain interfaces, i.e. function signatures
- Make note of possible future problems (better: prevent them)
- Sometimes might be good to document your reasoning
- Do not “over-comment”
- Clean code: **You write it once and you read it many times**

External documentation:

- Again: Explain your interfaces (can be derived from internal, e.g. doxygen.org)
- For large projects: **Explain the big picture**
 - ▶ Wiki pages with use-cases and examples
 - ▶ Consider using UML (unified modelling language)

```
class TheClass(object):
    """ Documentation of this class. """
    def __init__(self, var):
        self.var_ = var
        ## @var var_
        # my member variable

        ## Documentation of this function.
        # More on what this function does.
        ## @param arg1 an integer argument
        ## @param arg2 a string argument
        ## @returns a list of ...
    def some_function(self, arg1, arg2):
        pass
```

```
if a > b: # when a is greater than b, do...
```

Documentation: Do it while it's fresh

Two sides of the same coin: Internal and external documentation

- Both necessary to make your programs easy to use
- They have different purpose!

Internal documentation:

- Explain interfaces, i.e. function signatures
- Make note of possible future problems (better: prevent them)
- Sometimes might be good to document your reasoning
- Do not “over-comment”
- Clean code: **You write it once and you read it many times**

External documentation:

- Again: Explain your interfaces (can be derived from internal, e.g. doxygen.org)
- For large projects: **Explain the big picture**
 - ▶ Wiki pages with use-cases and examples
 - ▶ Consider using UML (unified modelling language)

```
class TheClass(object):
    """ Documentation of this class. """
    def __init__(self, var):
        self.var_ = var
        ## @var var_
        # my member variable

        ## Documentation of this function.
        # More on what this function does.
        ## @param arg1 an integer argument
        ## @param arg2 a string argument
        ## @returns a list of ...
    def some_function(self, arg1, arg2):
        pass
```

```
if a > b: # when a is greater than b, do...
```

Document while coding

You write it once, read it many times

Write build scripts to ease your life

Makefiles — makes compilation **easier and faster**

- Makefiles might look complex
- More than one source file: Useful!
 - ▶ Again: Think about yourself in 2 years
- Write your own for a small project
- Automatically allows parallel compilation (option -j)

Abstraction layer on top: CMake and others

- Might look like overkill; Makes things easier in the long run
 - ▶ CMake is easier to read and better documented
 - ▶ Improved **portability**
 - ▶ Support different build-systems: ninja, GNU make, ...
- At least you should learn how to compile with it

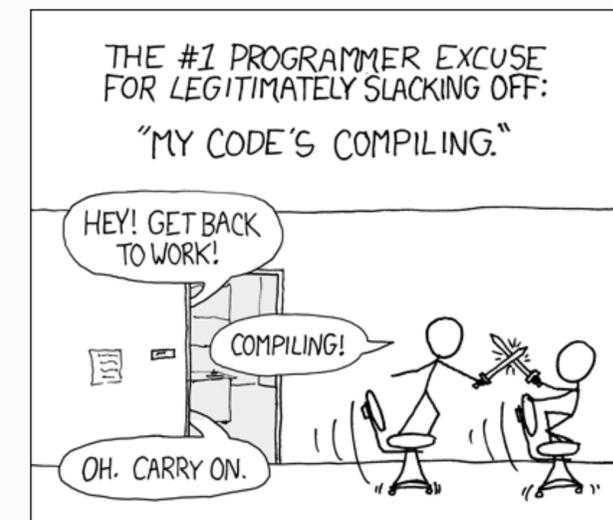
```
CC=clang++
CCFLAGS=-Wall -pedantic -std=c++14
SOURCES=src/howmanydays.cc
OBJECTS=$(SOURCES:.cc=.o)
EXE=howmanydays

all: $(SOURCES) $(EXE)

$(EXE): $(OBJECTS)
    $(CC) $(CCFLAGS) $(OBJECTS) -o bin/$@

%.o: %.cc
    $(CC) $(CCFLAGS) -c -o $@ $<

.PHONY: clean all
clean:
    rm -f $(OBJECTS) bin/$(EXE)
```



“Compiling” by Randall Munroe
xkcd.com

Use appropriate tools for debugging

While running your code:

- printing to console: only suitable for (very) small code base
- Sooner or later have to use a debugger: **gdb** (GNU debugger) — better sooner than later
 - ▶ basic commands: **run**, **bt**, **info <*>**, **help**
 - ▶ very useful trick - attach to a running program: **gdb <executive> <pid>**
- Python debugger (**pdb** or rather **ipdb***):

General hints for debugging

- Segmentation violations due to memory management
 - ▶ Life-time vs. scope
 - ▶ Only use raw pointers when you have to!
(I.e. when performance becomes crucial and *you know what you're doing*)
 - ▶ Look at smart pointers (part of C++11/14 standards, alternative: boost)
- Even if you don't have crashes: Memory Leaks. Try **valgrind** (valgrind.org)

```
*import ipdb; ipdb.set_trace() # set a breakpoint
```

Static Code Checking

While writing your code:

- There are static code analysis tools that can help you
- Try out a linter for your preferred editor (e.g. atom: <https://atom.io/packages/linter>)
 - ▶ Highlights potentially problematic code
 - ▶ Your code will be more reliable

Static checking at compile time:

- Clang has a nice suite of static checks implemented <http://clang-analyzer.llvm.org>
 - ▶ Can also enforce coding styles
- Takes longer than compiling; HTML reports with possible bugs
- Might flag some false-positives

Static code checking helps you avoid problems!

```
errors.py
1 from __future__ import print_function
2 import os, sys, allthings
3
4 def main():
5     i = 1000
6     for k in range(j):
7         print(k)
8
9 if __name__ == "__main__":
10     main()
11
```

Undefined name 'j', undefined name 'j', Line 6, Column 23 0 misspelled words

```
Example.m
12 void foo(int x, int y) {
13     id obj = [[NSString alloc] init];
14
15     switch (x) {
16     case 0:
17         [obj release];
18         break;
19     case 1:
20         // [obj autorelease];
21         break;
22     default:
23         break;
24     }
25 }
```

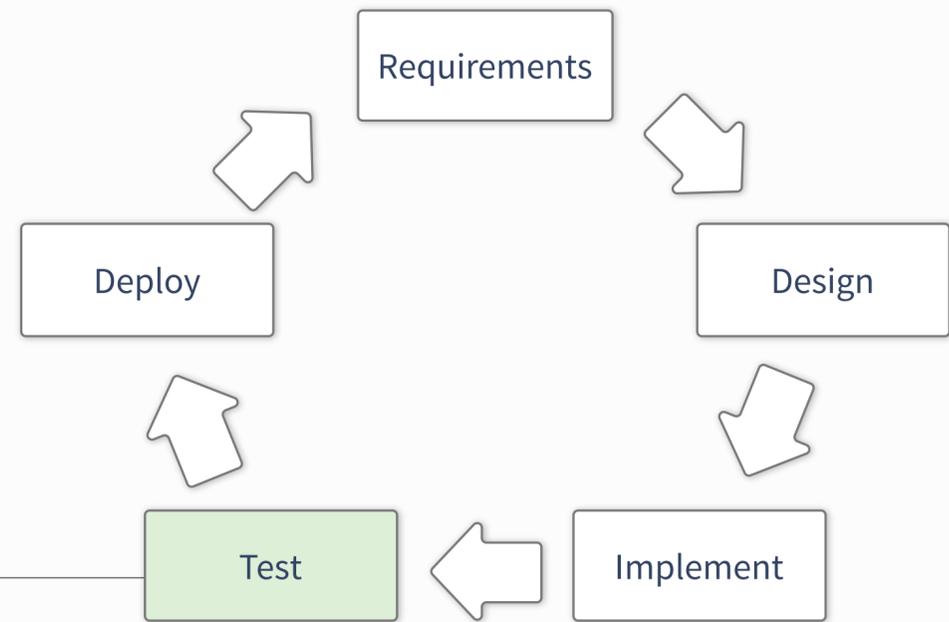
1 Method returns an Objective-C object with a +1 retain count (owning reference)

2 Control jumps to 'case 1' at line 18

3 Execution jumps to the end of the function

4 Object allocated on line 13 is no longer referenced after this point and has a retain count of +1 (object leaked)

Testing



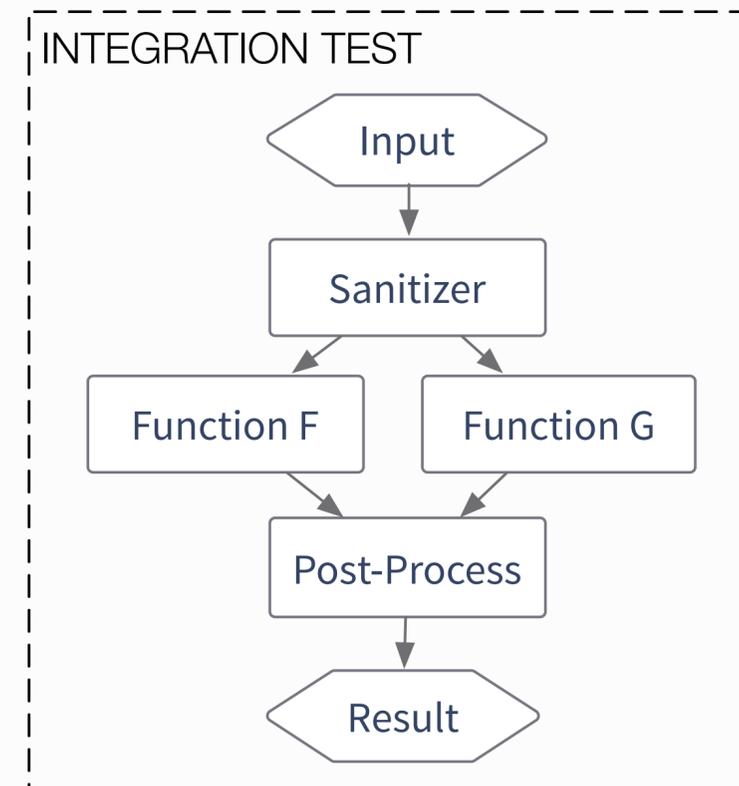
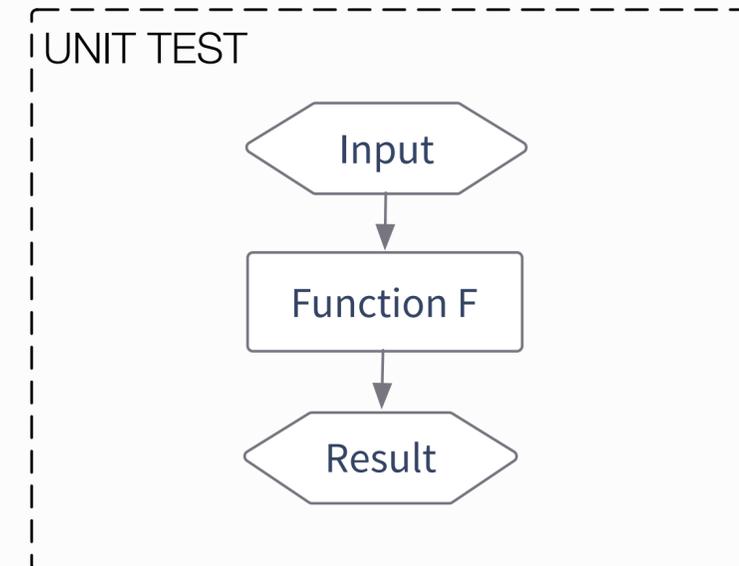
What do we mean with tests?

Different tests, different purposes:

- **Unit test**
 - ▶ Testing “units of code”, e.g. a function or class
 - ▶ Given a defined input => expected output?
- **Integration test**
 - ▶ Testing a larger part of your software
 - ▶ For example running an example and checking output

Do not mix it up with verification

- ▶ Checking if specifications are met



Writing good tests is hard

How to come up with tests?

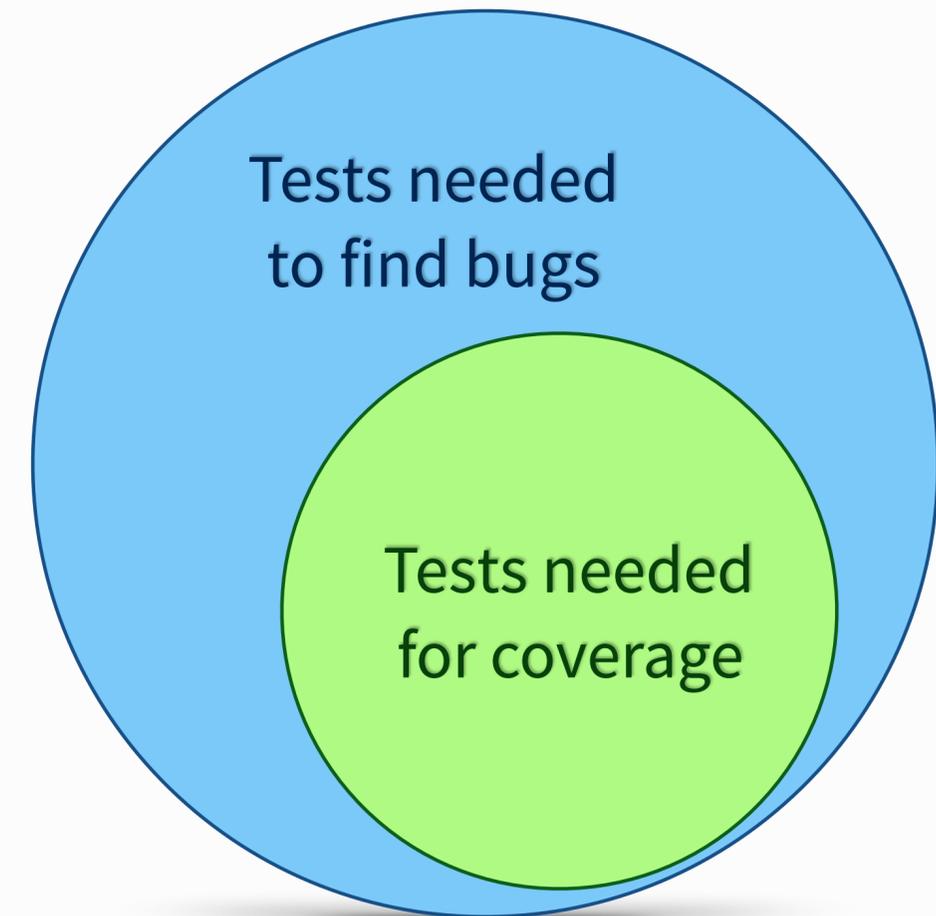
- What should the algorithm do?
 - ▶ Check if well defined input produces correct result
- How should the algorithm fail?
 - ▶ Check if wrong input fails in the way you want

You'll ~~probably~~ miss corner cases:

- Once you discover them, implement a test!
 - ▶ **Only let a bug hit you once**
- Have beta-testers / users help you
 - ▶ Use issue tracker
 - ▶ Be responsive!

Look at existing solutions to implement tests

- Python: [doctest](#) and [unittest](#) packages
- C++: [CTest](#) (integrated with cmake) & [Catch](#)



Interlude: doctest

> python testfib.py

```
def fib(n):
    """ Returns the fibonacci series at n
    >>> [fib(n) for n in range(6)]
    [0, 1, 1, 2, 3, 5]
    >>> fib(-1)
    Traceback (most recent call last):
        ...
    ValueError: n should be >= 0
    """
    if n < 0: raise ValueError("n should be >= 0")
    if n == 0: return 0
    a, b = 1, 1
    for i in range(n-1):
        a, b = b, a+b
    return a

import doctest
doctest.testmod()
```

Interlude: doctest

```
> python testfib.py  
>
```

```
def fib(n):  
    """ Returns the fibonacci series at n  
    >>> [fib(n) for n in range(6)]  
    [0, 1, 1, 2, 3, 5]  
    >>> fib(-1)  
    Traceback (most recent call last):  
        ...  
    ValueError: n should be >= 0  
    """  
    if n < 0: raise ValueError("n should be >= 0")  
    if n == 0: return 0  
    a, b = 1, 1  
    for i in range(n-1):  
        a, b = b, a+b  
    return a  
  
import doctest  
doctest.testmod()
```

Interlude: doctest

> python testfib.py -v

```
def fib(n):
    """ Returns the fibonacci series at n
    >>> [fib(n) for n in range(6)]
    [0, 1, 1, 2, 3, 5]
    >>> fib(-1)
    Traceback (most recent call last):
      ...
    ValueError: n should be >= 0
    """
    if n < 0: raise ValueError("n should be >= 0")
    if n == 0: return 0
    a, b = 1, 1
    for i in range(n-1):
        a, b = b, a+b
    return a

import doctest
doctest.testmod()
```

Interlude: doctest

```
> python testfib.py -v
> Trying:
>     [fib(n) for n in range(6)]
> Expecting:
>     [0, 1, 1, 2, 3, 5]
> ok
> Trying:
>     fib(-1)
> Expecting:
>     Traceback (most recent call last):
>         ...
>     ValueError: n should be >= 0
> ok
```

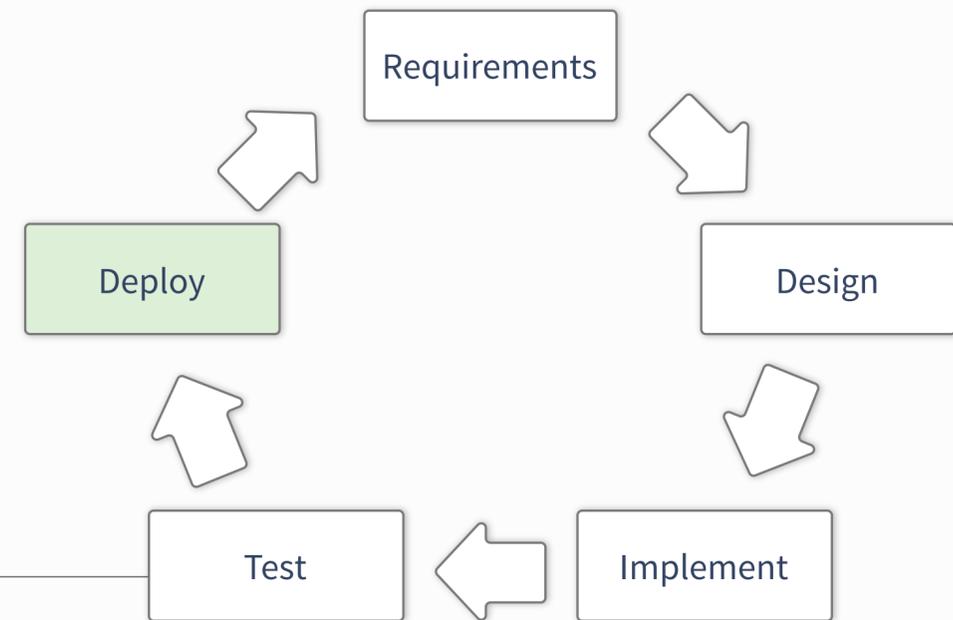
```
def fib(n):
    """ Returns the fibonacci series at n
    >>> [fib(n) for n in range(6)]
    [0, 1, 1, 2, 3, 5]
    >>> fib(-1)
    Traceback (most recent call last):
        ...
    ValueError: n should be >= 0
    """
    if n < 0: raise ValueError("n should be >= 0")
    if n == 0: return 0
    a, b = 1, 1
    for i in range(n-1):
        a, b = b, a+b
    return a

import doctest
doctest.testmod()
```

Test your software

and not just in production!

Deploying your software



Releasing the Software

When you release your software:

- Tag the repository
 - ▶ Ensure everyone has the same code
- Test in the target environment
 - ▶ Fresh virtual machine
- Accompanying documentation
 - ▶ Produce Doxygen pages
 - ▶ Update wikis (new version)
 - ▶ Make sure all examples work



Ideal case: All this is done every single night!

Continuous integration

Working in groups on software can be hard:

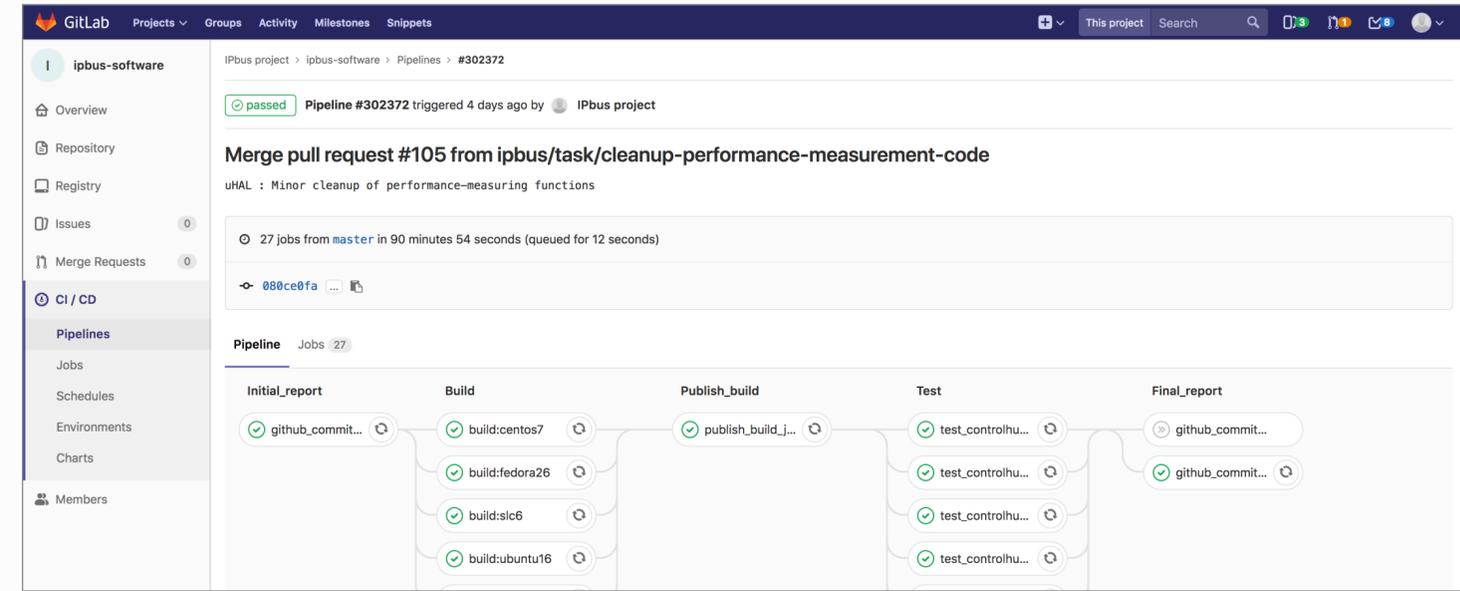
- Somebody changes something: Other code breaks
- **Avoid such nuisances by testing regularly!**

New contribution to the code base:

- Check everything works
 - ▶ Can do this by hand.. Tedious
 - ▶ Better: Automate it.



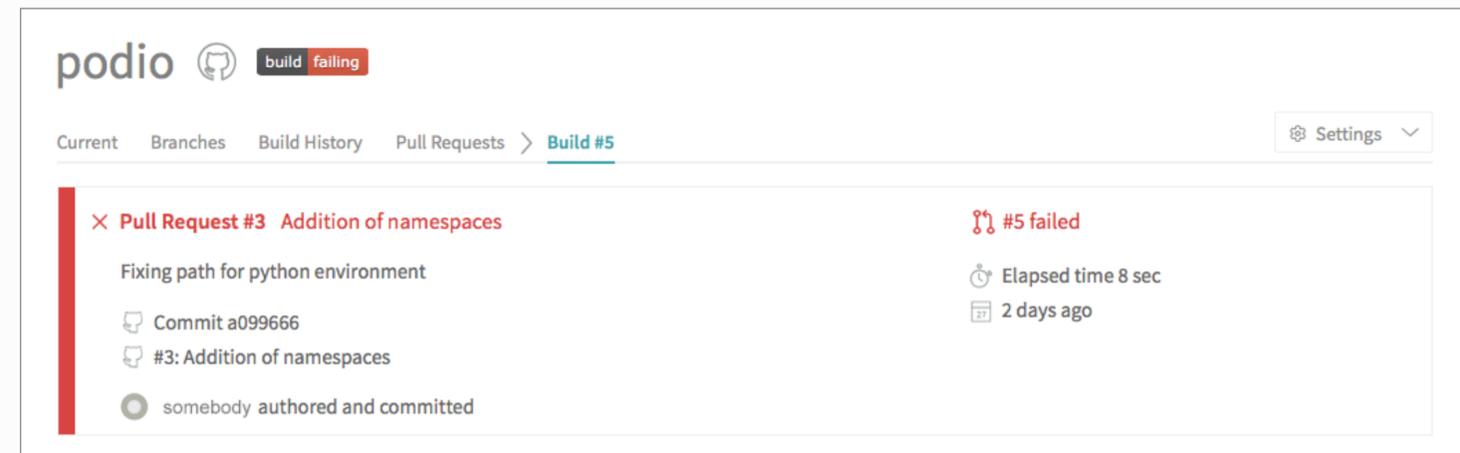
CI/CD



Gitlab CI - <https://about.gitlab.com/features/gitlab-ci-cd/>

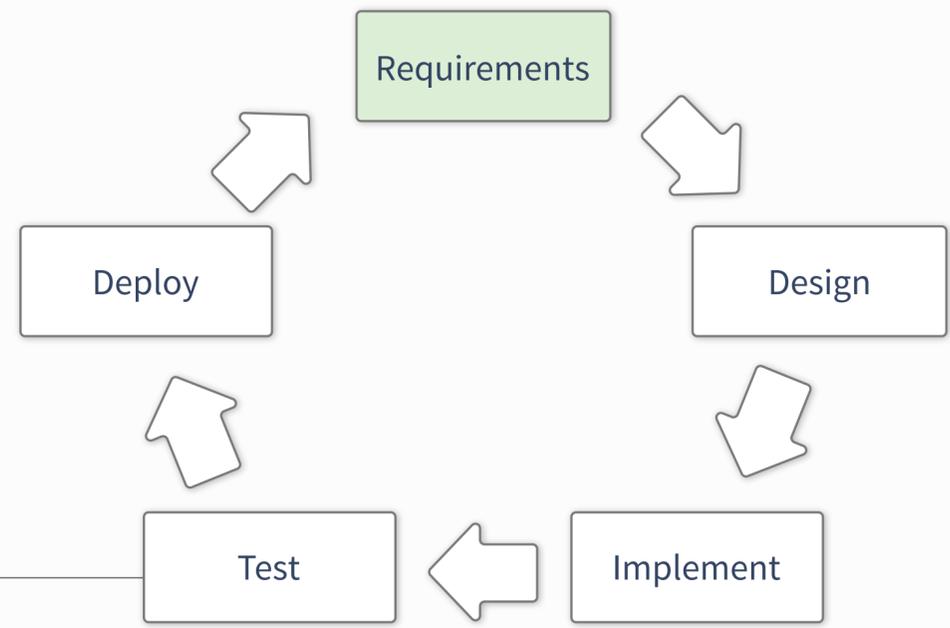
Many solutions exist that automatically test things:

- Check compilation
- Check all defined test cases
- Write nice summaries

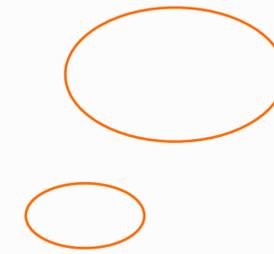


Travis CI - <https://travis-ci.org>

Requirements



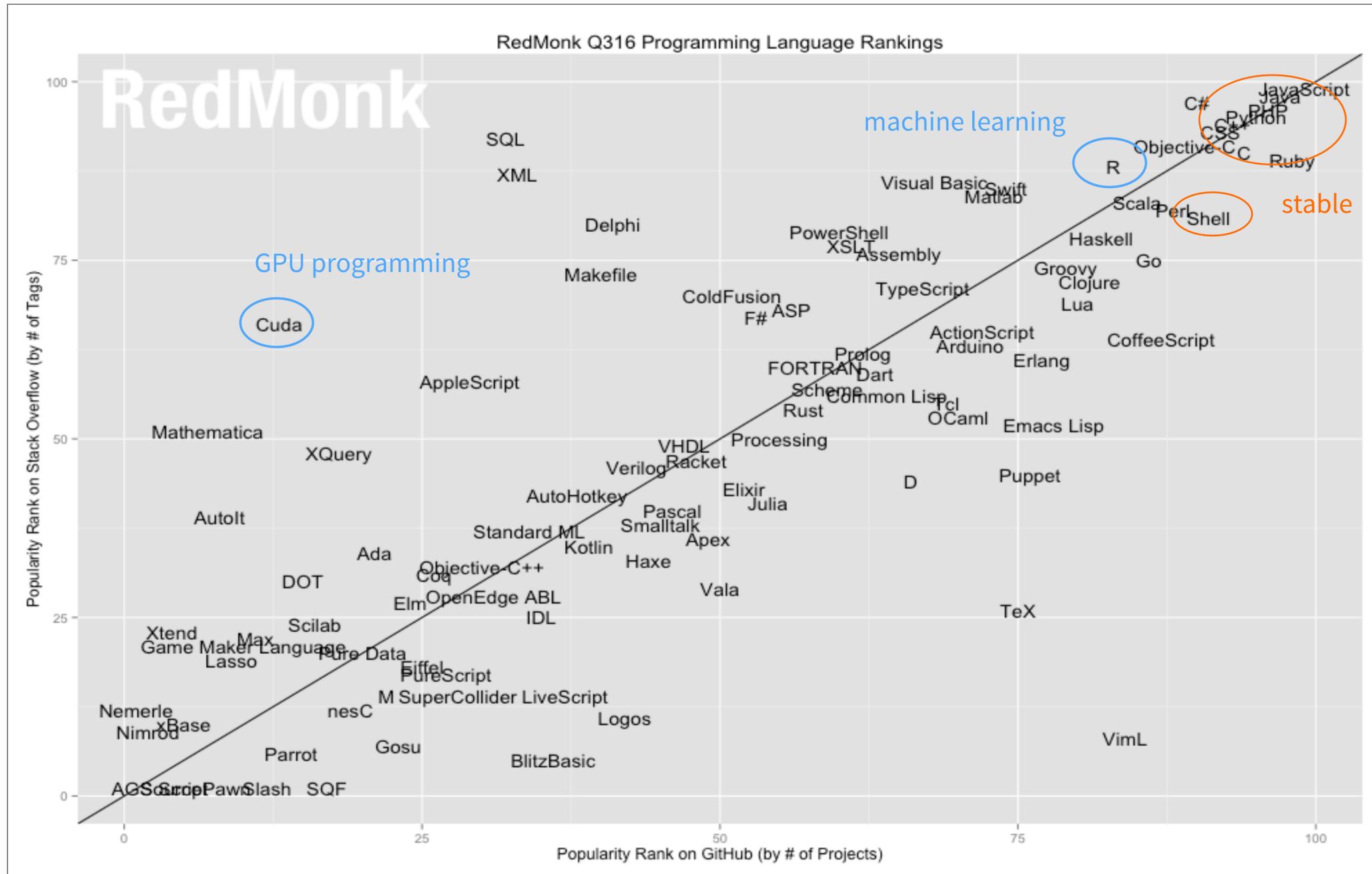
Choosing the programming language



The answer depends:

- Analysis?
- DAQ / Trigger?
- *External conditions?*
 - *Can you choose?*

Choosing the programming language



The answer depends:

- Analysis?
- DAQ / Trigger?
- External conditions?
 - Can you choose?

Choosing the programming language



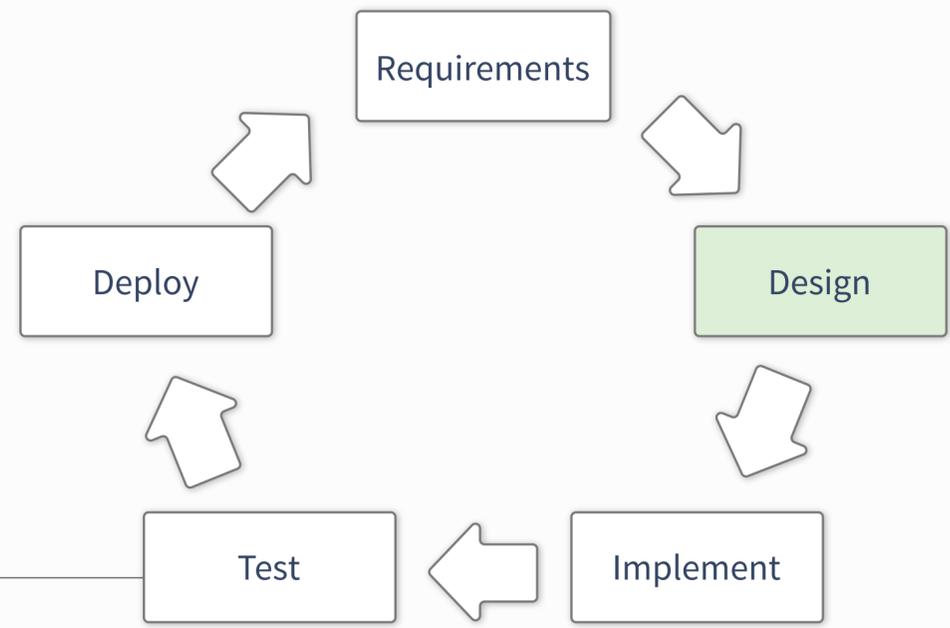
Choose wisely

- Favour documentation and support over features
- Favour large user-bases

Do you really
have to program?

Or has somebody already done it for you?

Design



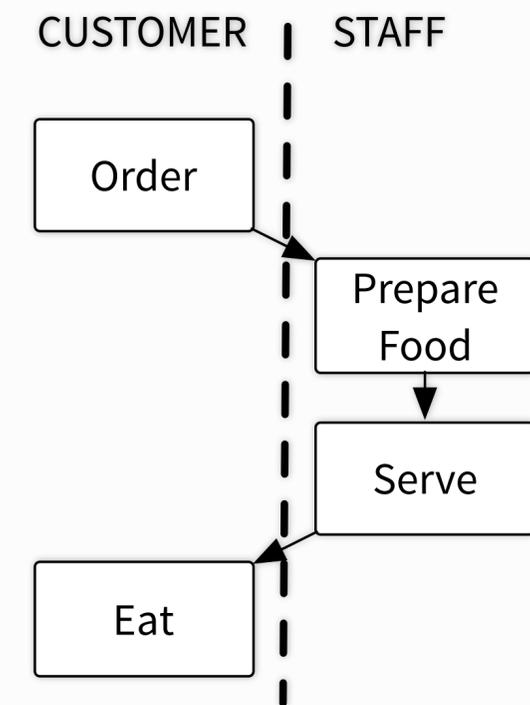
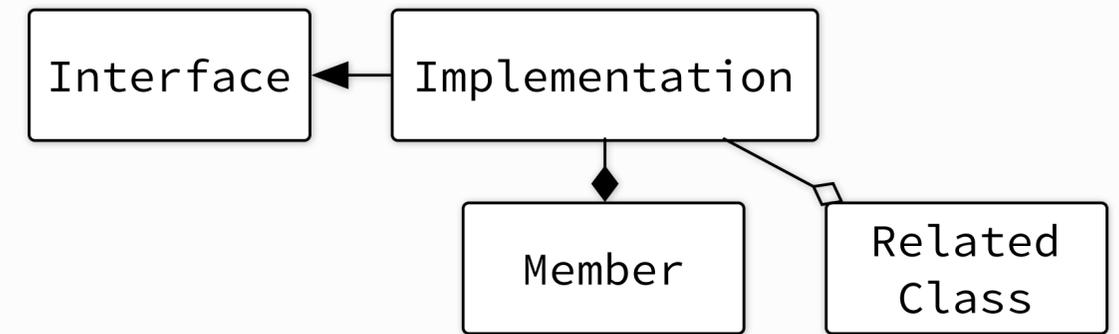
UML Diagrams

Unified Modelling Language: sketch a design

- Probably everyone has seen structure diagrams
 - ▶ Relationships of classes (or larger components)
- Behaviour diagrams
 - ▶ What does the user do and what should be the result?
- Interaction diagrams
 - ▶ How does data and control flow?

Forces you to be concrete!

to make them, look at draw.io or lucidchart.com



Things to keep in mind when designing

Maintainability

- Is it easy to adapt to changed environment?
- Can you cope with (slightly) changed requirements?

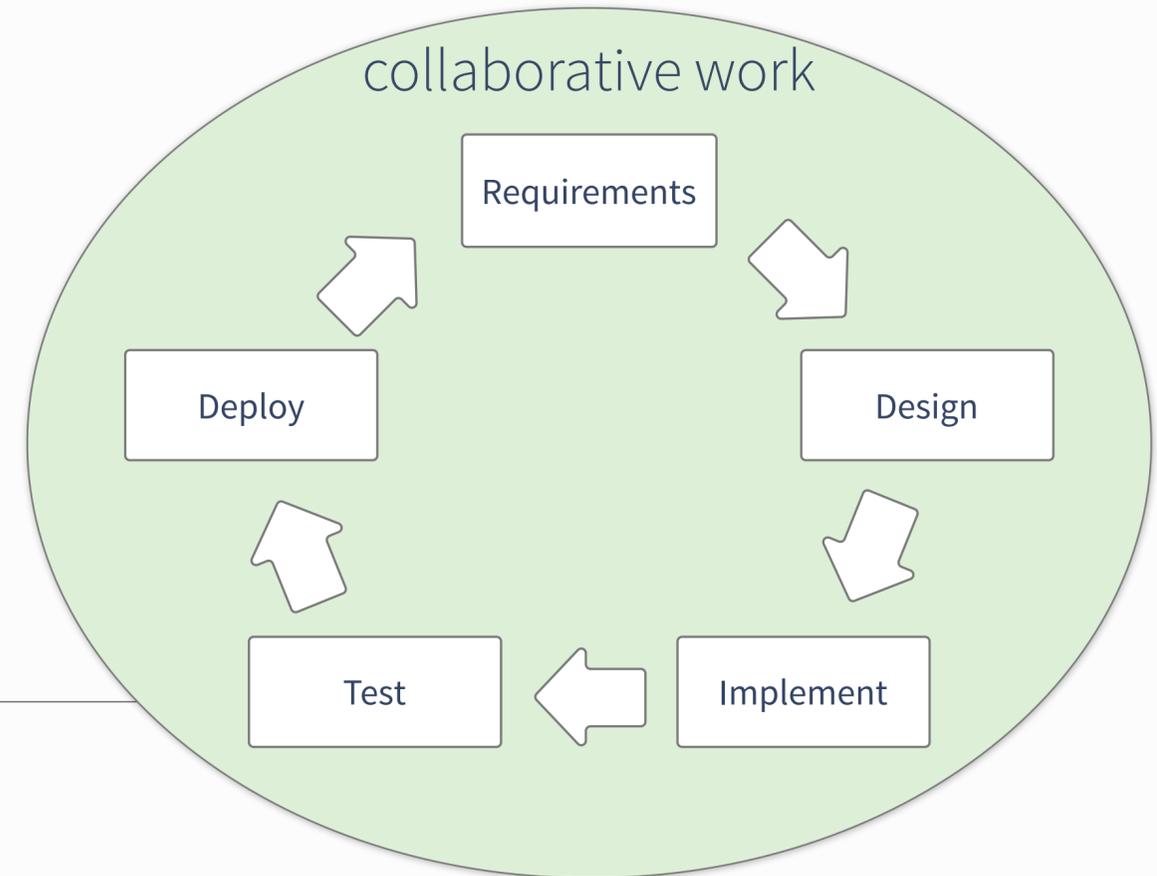
Scalability

- Large data volumes
 - ▶ Think about data-flow and data layout
 - ▶ Try to avoid complicated data structures

Re-usability

- Identify parts of the design that could be used elsewhere
- Could these be extracted in a dedicated library?

Collaborative programming



Development Cycles

Developing software efficiently:

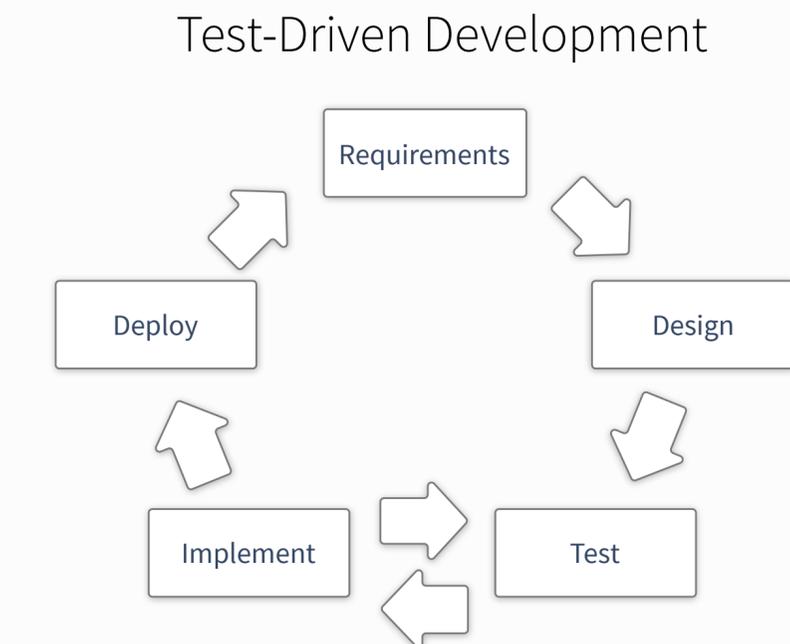
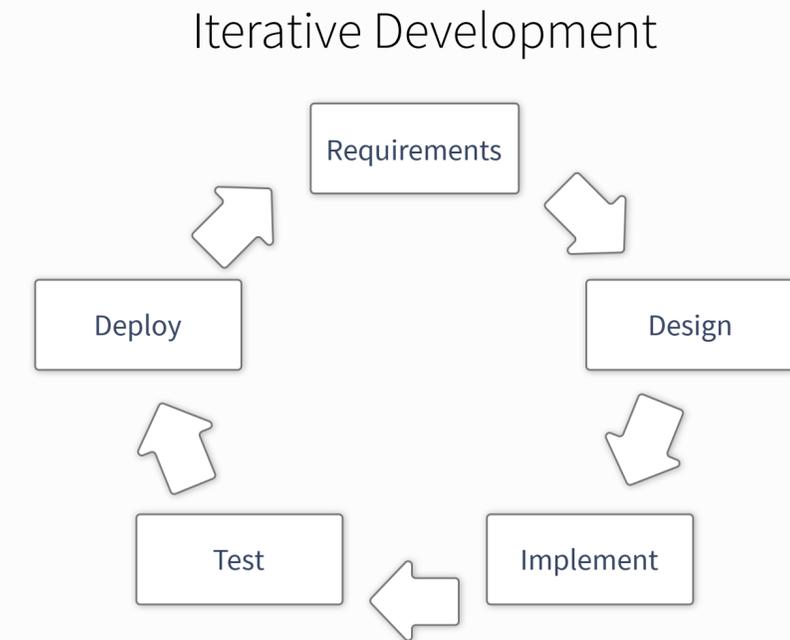
- Avoid duplication of work
- Avoid feature bloating
- Ensure code quality
- Deliver code timely

Many approaches to accomplish this:

- Examples: Iterative and Test-Driven Development

Similar principles, different focus

- on team management (agile development)
- on actual programming style (lean development / TDD)
- broad guidelines to deliver (iterative development)



Revision control software

Revision control: **Essential for collaboration...** but not only

Once upon a time: CVS and Subversion [“CVS done right”]*

Nowadays: Distributed revision control - Great for personal use

- Easy to work on the go
- Your local copy has everything (including history)

Probably the most popular **git**: git-scm.com

[“there is no way to do CVS right”]*

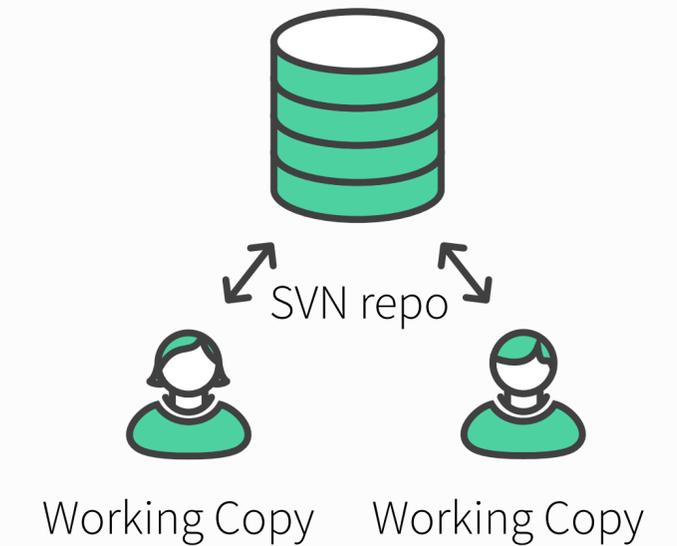
- Other distributed solutions are: Mercurial, bazaar...
- Easy to learn...

Graphics from: <https://www.atlassian.com/git/tutorials/>
Ultimate git guide: <https://jwiegley.github.io/git-from-the-bottom-up/>

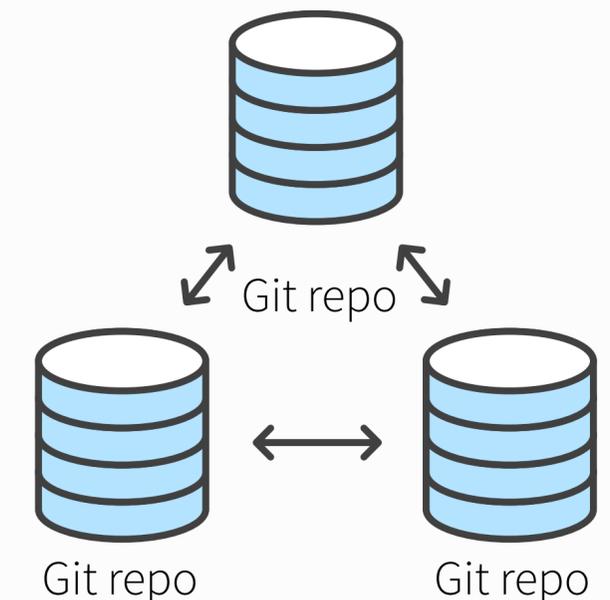
Git tutorials:
<http://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>
<http://pcottle.github.io/learnGitBranching/>

* paraphrasing Linus Torvalds

Central-To-Working-Copy Collaboration



Repo-To-Repo Collaboration



Interlude: git basics

```
> git init  
Initialized empty Git repository in /TestDirectory/.git/  
> vim README.md  
skipping this part.
```

Interlude: git basics

```
> git init
Initialized empty Git repository in /TestDirectory/.git/
> vim README.md
skipping this part.
> git add README.md
```

Interlude: git basics

```
> git init
Initialized empty Git repository in /TestDirectory/.git/
> vim README.md
skipping this part.
> git add README.md
> git commit -m "Initial commit of readme."
```

Random github commit messages:
<http://whatthecommit.com/>

The git ecosystem

Easy to host & share your projects:

- Setting up a shared repo can be done via any cloud service, e.g. dropbox
- Many open-source hosting sites, biggest: github.com
- Not open to public but CERN users: [GitLab.cern.ch](https://gitlab.cern.ch)
 - ▶ Both include fairly usable issue-tracking
- The beauty of pull-requests*:
 - ▶ Do builds on pull-requests (combine with CI)
 - ▶ Review contributed code on pull-requests



Git is widely used — **de-facto community standard**

- Exception: Python uses Mercurial

The more you learn the more you'll like it!

* merge-request in GitLab

The git ecosystem

Easy to host & share your projects:

- Setting up a shared repo can be done via any cloud service, e.g. dropbox
- Many open-source hosting sites, biggest: github.com
- Not open to public but CERN users: [GitLab.cern.ch](https://gitlab.cern.ch)
 - ▶ Both include fairly usable issue-tracking
- The beauty of pull-requests*:
 - ▶ Do builds on pull-requests (combine with CI)
 - ▶ Review contributed code on pull-requests



Git is widely used — **de-facto community standard**

- Exception: Python uses Mercurial

The more you learn the more you'll like it!

* merge-request in GitLab

General Tips & Pointers

Learning about software development

Coursera — courses by universities (Caltech, Johns Hopkins, Stanford and more)

- <https://www.coursera.org/courses>
- Large variety of courses
 - ▶ Not only technology / programming
 - ▶ Also physics, biology, economics... and more
 - ▶ Also in different languages

Udacity — courses from industry (Google, Intel, Autodesk)

- <https://www.udacity.com/courses#!/all>
 - ▶ Mixed courses: Some free, recently switched to a payed model with monthly fees

University Homepages — have a gander... many courses available through YouTube etc.

- e.g.: [Programming Paradigms, Stanford University](#)

<http://ureddit.com/> — University of Reddit

Closing Advice

Before you write trigger / DAQ software, you should know the ins and outs:

- What is: compiler, interpreter, linker, terminal, object, class, pointer, reference
- If these concepts are not clear: Excellent material on the web (previous slide)

Before (and while) implementing: Think

- Smart solutions can take significant amount of time...
put it on the back-burner if you have other things to work on

Read! Ask! Write! The internet is full of information... Blogs, tutorials, StackOverflow, also Wikipedia can be very useful to get a grasp of new concepts

Conclusion

These slides were full of starting points: You have to follow up to get something out of it

- Most of it are tools to make your life easier
 - ▶ Bonus: If you know them you'll have an easier time to follow nerd-talk
- Nothing is free
 - ▶ You'll have to invest some effort to learn
 - ▶ If you do that this week: We'll be here to help!

Homework:

- Install git, start a repository. Try branching on the web
- Run tmux, kill the connection, reconnect and see if you can continue where you left off
- Tune your .bashrc / .bash_profile to get a more useful prompt
- Try out vim / emacs / atom / vscode and learn what suits you best
 - ▶ Download a shortcut summary...
 - ▶ Learn how to block-select, indent multiple lines, rename occurrences of text

Master by doing

Don't forget: Have fun while doing so!

Random Things

6 Stages of Debugging:

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?
 - <http://plasmasturm.org/log/6debug/>

Go-language: Designed with threading in mind
<http://tour.golang.org/welcome/1>

Want to try your programming skills?
Google code jam (registration open):
codingcompetitions.withgoogle.com/codejam
Also you can just practice
by solving nice problems.

Guru of the Week: (Not any more)
regular C++ programming problems
with solutions by Herb Sutter
<http://www.gotw.ca/gotw/>

“Debugging is like being the
detective in a crime novel where
you are also the murderer.”
– @fortes

About JavaScript:
<https://www.destroyallsoftware.com/talks/the-birth-and-death-of-javascript>
<https://www.destroyallsoftware.com/talks/wat>

2014 lecture has complementary stuff:
<http://indico.cern.ch/event/274473/session/21/material/0/0.pdf>

More Random Things

In HEP probably no way around ROOT / RooFit

- Maintained at CERN, used in LHC experiments

GNU R — www.r-project.org

- Used widely among statisticians (including finance and others)
- Interpreted language + software for analysis and graphical representation
- ROOT bindings now available (use it through TMVA)

SciPy — <http://www.scipy.org/>

- Collection of python libraries for numerical computations, graphical representation and containing additional data structures

Sci-kitlearn: — <http://scikit-learn.org/stable/>

- Python library for machine learning
- ROOT bindings available (usable through TMVA)

Data visualisation:

Matplotlib (part of SciPy)

- histograms, power spectra, scatterplots and more.. extensive library for 2D/3D plotting

ROOT

- Again, probably no way around it... Sometimes a little unintuitive

Other:

JaxoDraw — <http://jaxodraw.sourceforge.net/>

- Feynman graphs through “axodraw” latex package

tex2im — <http://www.nought.de/tex2im.php>

- Need formulas in your favourite WYSIWG presentation tool?

GraphViz — <http://www.graphviz.org/> or MacOS: <http://www.pixelglow.com/graphviz/>

- Diagrams / Flowcharts with auto-layout

SAGE — www.sagemath.org

- Open source alternative to Matlab, Maple and Mathematica

GNUPlot — <http://www.gnuplot.info/>

- Quick graphing and data visualisation

Wolfram Alpha — <http://www.wolframalpha.com/>

- Wolfram = Makers of Mathematica.. A... ask me anything?:
 - ▶ <http://www.wolframalpha.com/input/?i=how+much+does+a+goat+weigh>
 - ▶ Answer: Assuming “goat” is a species specification. Result: 61 kg

```
# tune your prompt:
if [ "$PS1" ]; then
    PS1="\[\033[1;29m\]\[\033[0;34m\] \u\[\033[0;34m\]@\[\033[1;34m\]\h : \[\033[0m\]: \w \
\[\033[0;36m\] \$(git branch 2>/dev/null | grep '^*' | colrm 1 2) \[\033[0m\] ] \n \[\033[0;31m\]\$\
\[\033[0m\] "
fi

# do not put duplicate lines into history:
export HISTCONTROL="ignoredups"

# default to human readable file sizes
alias df='df -h'
alias du='du -h'

# get some color
alias grep='grep --color'

# more file listing:
alias l='ls'
alias ll='ls -lt -h -G -c -r'

# fool proof cp - asks for each file, use fcp if you're sure
alias fcp='cp'
alias cp='cp -i -v'

# never remember those..
alias untgz='tar -xvzf'
alias tgz='tar -pczf'

#never install root:
source /path/to/your/working/root/bin/thisroot.sh
alias root='root -l'

# Mac OS stuff
alias wget='curl -O'
```

resulting prompt

```
[ user@host :: pwd current git-branch ]
[ joschka@local :: ~/test master ]
$
```