Good coding practice

Agile development

One of a number of approaches to software development

Facilitates collaboration, robustness, and adaptability over time

These are *qualitative and social* considerations

The compiler doesn't care

Goal is to support working with an adviser or colleague and enhance reproducibility and reusability of your code

"The user" is either "your boss," "a reviewer," or (most commonly) "you in a few months"

Contract programming

Start each public function with a "contract":

- A list of preconditions that the function expects
- A description of what the function will then provide

Different from *defensive programming*, where your code tries to figure out what you might have meant and tries something. Defensive programming is more work and more likely to cause confusion.

Functions designed by contract are simpler to write. You let them "fail fast and hard" – they just give an error if preconditions are not met (i.e., wrong input was given).

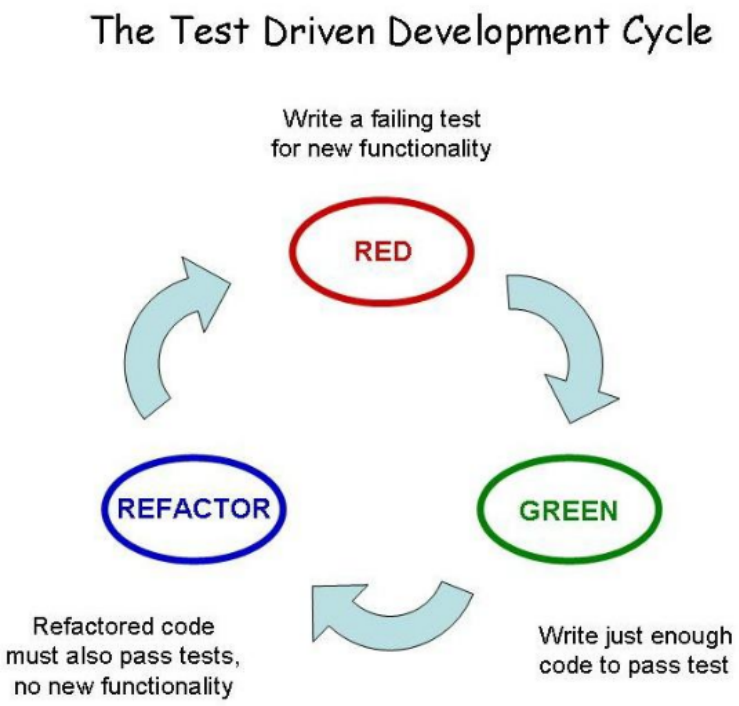
Test driven development

Development steps of TDD

1. Design: Write a test for a desired behavior
2. Implement: Whip up code that passes the test
3. Refactor: Eliminate all duplication in the code

Basic rules of TDD

- Write new code only if an automated test has failed
- Eliminate duplication



Continuous integration

For larger projects with many collaborators, the testing is often automated

Continuous integration tools can, for example, checkout your git repository or branch, compile software, run test suites on many different machines, and report to you.

This is most useful in industry contexts, where updated products are delivered constantly.

Sometimes useful for academics, for example with long-running online experiments, or if some collaborators need their code sanity-checked before it is added to the lab's shared codebase.

Refactoring

Martin Fowler (1999). *Refactoring – Improving the Design of Existing Code.*

Code smells

When to refactor? — When code smells.

DRY smell

Don't Repeat Yourself

- If a large chunk of code appears in multiple functions, it should probably be its own function
- If a small bit of code is repeated over and over again in a function, it should maybe be an intermediate variable

If a piece of code appears multiple times despite serving the same purpose, one risk is that one version is overlooked when your collaborator improves code.

A trickier DRY smell is when two methods do the same thing with different algorithms. Then you need to think about whether they are really doing the

same, and if so use the better algorithm both times – from its own function of course.

Long function smell

Programs written in one long file work just as well as tons of little function delegating to each other

However, collections of smaller functions that call each other are easier to maintain by future you

Splitting a long function into smaller ones that call each other is sometimes called *indirection*.

Advantages of indirection include

- The function can be used multiple places and only has to be maintained once
- If done well, smaller functions abstract away some of the code into more readable function calls (and the code almost comments itself!)

If a long function has comments like "in this section, we're going to do xyz..." that's a signal that some semantic distance is being created. That section should probably be its own function called `xyz_doer()`

Modern programming languages have essentially no performance penalty for function calls

---

Long parameter list smell

Get a load of this:

```
[a, b, c, d] = mySadFun(e, f, g, h, e3, t, y, dd);
```

This is hard to use and understand. Most likely the function is too complex to be easily read.

If you really need to do this, collect your input in a single parameter object, which could be a structure:

```
out  =  myFunFun(parameters) ;
...
```

or a class:

```
par  =  Parameter()    ;
out  =  par.myFunFun() ;
...
```

Code deodorant smell

Commenting code is good, but sometimes comments can smell bad.

If comments are full of warnings about strange ways a function needs to be conditioned to behave well, that implies the function may be prone to erroneous use or be poorly implemented.

This often indicates a scenario where the correct action is *not* to refactor, but to start over.

Other smells

- Shotgun surgery – if you're making a change, are you in one place in your code or are you updating in multiple places to do a single thing?
  - You should probably be putting those different bits of code into a class that you then call from those places.
- Divergent change – do you find yourself updating a function to work one way, and then later back to work another way because the data are now different?
  - That should probably be two functions.
- Speculative machinery – addition of functionality that isn't used but may be used one day.
  - It might break, confuse users, and why do now what you might not need to do later?

Assignment for today

Review @Norm2d as implemented by the student after you in the alphabet (JieWan review adri, adri review angela, etc.).

1. Clone their own repository and examine their solution in the assignment folder
2. Run the test suite and note any issues
3. Read the code line by line and add comments for these code smells:
   - Repetitive code / redundancy
   - Uninterpretable / long code that does many things
   - Broken functions / lack of functionality not picked up by test suite
   - Unnecessary complexity / possible simplification
4. Make improvements (and keep running the test suite)
5. Commit each change with an informative message, comment on *why* you made a change (be nice!)
6. Open a pull request to the forked repository

Optional: In addition to the student before you in the alphabet, conduct a similar review of the instructor's code (including the test suite) and see if you can find errors or inefficiencies. Finding errors in the instructor's code earns **no** credit.

Homework: Automated report generation

Imagine that we are keeping track of some behavioral data that is placed in a file on a website. There are two variables of interest, x and y, and we want to be able to rapidly report their mean, variance, and correlation. Add:

1. a static method `Norm2d().estimate(data)` that takes a 2xN matrix and returns a Norm2d object with Mean and Covariance computed from data (and all contingent properties updated)
2. `pseudonym/main.m % entry point function, has settings, calls required functions`
3. `pseudonym/getData.m % downloads a data file from a url`
4. `pseudonym/readData.m % reads the data file into a variable`
5. `pseudonym/report.m % prints a report to file in markdown format`
6. `pseudonym/test.m % tests each function and method in the package`
7. `pseudonym/Contents.m % prints help text for all functions`

- You may want to add additional functions, be sure to add+commit those.
- Follow principles of contract programming and test driven development.
- The url is http://www.cidlab.com/files/cogs205b.csv
- Name the output file `pseudonym-1.md` and save to your directory. Don't just print naked numbers – add some formatting and text.
- If a user deletes the report, then running `pseudonym.main()` should regenerate it, starting with downloading the data.