PROJECT MANAGEMENT

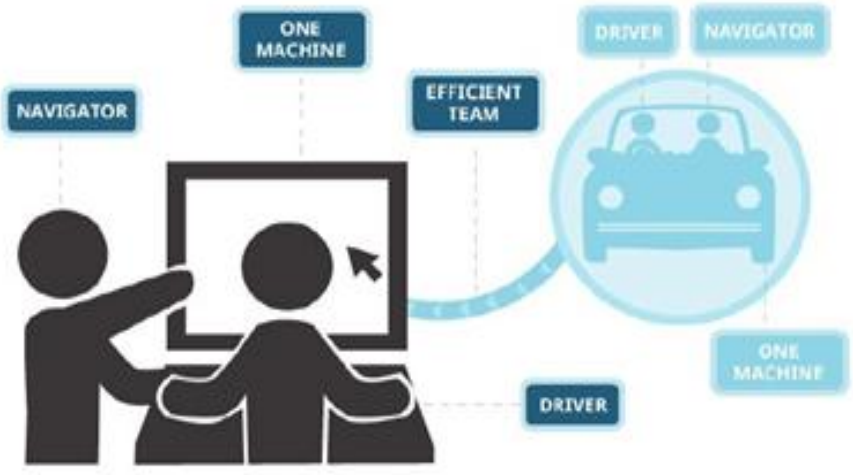
Part 2 :

Pair programming, Test Driven Development and Continuous Integration

Tianxiao LIU - Master IISC 1 – University of Cergy-Pontoise

PAIR PROGRAMMING PRINCIPLE

- **We help each other succeed.**
- Pair programming may seem weird when you are not used to it yet.
- **It's an extremely powerful approach !**
  - It may increase your brainpower.
- Basic principle of pair programming
  - Two roles : the driver and the navigator
  - Driver, the person who **codes**
  - Navigator, whose job is to **think**
- What do we think about, as navigator ?
  - What tasks (steps) to work on next …
  - How the work best fits into the overall design…



HOW TO PAIR

- **Frequently asked question : is pairing wasteful ?**
  - It is if you think that programming is just typing statements in a programming language…
- Pair frequently but not exclusively.
- **These are all normal feelings !**
  - Driver : Feel that your navigator is faster ☹ you are working on two things simultaneously !
  - Navigator : Expect to feel like you want to step in by taking the keyboard ☺ take your time to help your driver be more productive ! Look for the answers of your questions…
- The frequency for switching : **half hour**.
- Expect to feel tired at the end of the day of pairing.
- When a pair goes dark --- talks less, lower their voices or doesn't switch with other pairs --- it's often a sign of **technical difficulty**.

PAIRING TIPS

- Pair on everything you'll need to maintain.
- Allows pairs to form fluidly rather than assigning partners.
- Switch partners when you need a fresh perspective.
- Avoid pairing with the same person for more than a day at a time.
- Sit comfortably, side by side.
- Produce code through conversation. Collaborate, don't critique.
- Switch driver and navigator roles frequently.

UBIQUITOUS LANGUAGE

- We need to understand each other.
- The **Domain Expertise Conundrum**
  - **Problem** : try to describe the business logic of your system to a non programmer domain expert, avoiding programmer jargon (ex. design patterns, coding styles …)
- **Conundrum**
  - Domain experts are rarely qualified to write software.
  - Programmers don't always understand the problem domain.
- **The only solution**
  - Programmers should speak the language of their domain experts.

UBIQUITOUS LANGUAGE IN CODE SOURCE

- **Design your code to use the language of the domain.**
  - Use terms of the domain to name classes, methods, variables…
- This is the art !
  - Reflecting in code how the users of the system think and speak about their work.
- We refine our knowledge by encoding our understanding of the domain : gaps in our knowledge would result in bugs.
- Many standard processes are proposed
  - Domain modelling
  - Domain-centric design
  - Domain-driven design

} **True object-oriented design.**

REFINING THE UBIQUITOUS LANGUAGE

- Your ubiquitous language is a living language.
- Learning new things → improve the language

Agreement to the changes	Clarify your understanding ?	Update the design
• Whole team • Domain experts	• Do changes help ? • Remove all jargons	• With the changes • Synchronized refactoring

- Do not introduce technical debt (mismatch) → ugly bugs

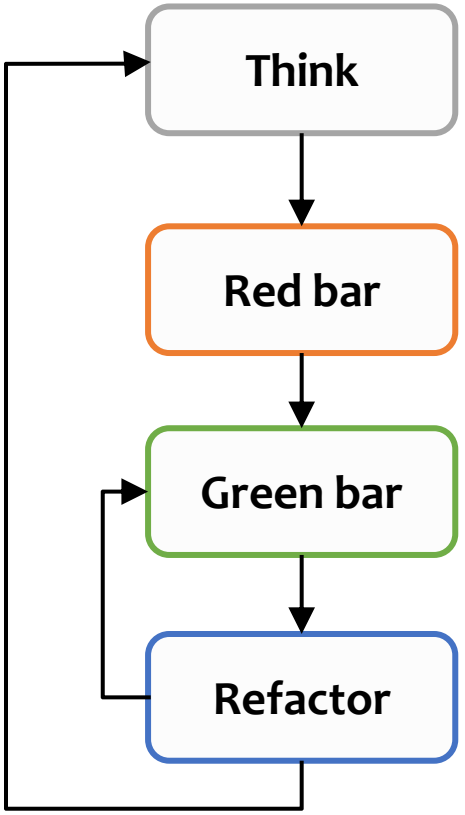
TEST DRIVEN DEVELOPMENT (TDD)

- **We produce well-designed, well-tested, and well-factored code in small, verifiable steps.**
- Programming is a demanding work that requires perfection.
- People are not good at perfection. → Software is buggy.
- We need a process that alerts us to programming mistakes.

- **TDD : a rapid cycle of testing, coding and refactoring.**
- When TDD is used properly, it'll help us improve the design, document public interfaces and guard against future mistakes.

TDD STANDARD CYCLE

- TDD takes moments to learn and a lifetime to master.
- It operates in a **very short** cycle that repeats over and over again.
- Each cycle will only take a few minutes, ideally.
- TDD uses small tests to force your to write your code to make them pass.



STEP 1 : THINK

- Imagine what behavior you want your code to have.
- Think of a small increment (ex. about five lines of code).
- Think of a test
  - **This test will fail unless that behavior is present.**
- Challenge
  - It can be difficult to think in small increments.
- Solution : **pair-programming**
  - **Driver** : try to make the current test pass.
  - **Navigator** : stay a few steps ahead, thinking of tests that will drive the code to the next increment.

STEP 2 : RED BAR

- **Write the test.**
  - It's only enough test code for the current increment of behavior.
  - It takes only a few lines. → If not, try to take a smaller increment next time.
- Code in terms of the class' behavior and its public interface.
- **Before behavior implementation (step 3), you test uses inexistent classes and methods → it forces you to design them in step 3.**
  - The internals of a class : perspective of implementer.
  - The externals of a class (API) : perspective of a user of the class.
- Run the test (in the test suite) → New test fails. → Very good (red bar)
- If the test passes, or it fails in a different way than you expected → Troubleshoot the problem → predict what's happening with the code

STEP 3 : GREEN BAR

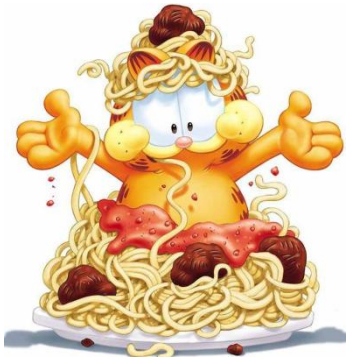
- **Now write just enough production code to get the test to pass.**
- Don't worry about design purity or conceptual elegance for now.
- Just do what you need to do to make the test pass !
- Run the test again → All the tests pass. → Green bar
- It fails ?!
  - Your partner (navigator) sees the problem. → good!
  - Nobody sees the problem. → Erase the code and try again.
- Key point : **remaining in control**
  - Revert the code to *known-good code*
  - Switch pair roles

STEP 4 : REFACTOR

- Refactor without worrying about breaking anything (the tests pass.).
- Review the code and look for possible improvement
  - Ask your navigator if you are pairing
- List all problems (need improvement)
  - A series of very small refactorings
  - Normally , one or two minute will be enough for each one. (< five min.)
  - **Run the test after each one, that should still pass.**
  - No ? Undo all and get back to *known-good* code.
- Do you best.
  - Refactor as many times as you like.
  - **Refactorings are not supposed to change behavior !**

CODE REFACTORING

- **Every day, our code is slightly better than it was the day before.**
- Entropy → Chaos → Mess of spaghetti
- Refactoring → reversible work



- **Reflective design**
  - Analyze the design of existing code
  - Improve it
  - We need **code smells :** condensed nuggets of wisdom
- Code smell does not necessarily mean that there's a problem
  - It may indicate that it's time to *"take out the garbage from the kitchen"*.

CODE SMELL AND REFACTORING SOLUTIONS

- Divergent Change (1) and Shotgun Surgery (2) : cohesion problems
  - (1) unrelated changes affect the same class : the class involves too many concepts → **Split it**
  - (2) You have to modify multiple classes to support changes to a single idea : the concept is represented in many places in the code → **A single home**

- Primitive Obsession (3) and Data Clumps (4)
  - (3) Represent high-level design concepts with primitive types → **Encapsulate the concept in a class**
  - (4) Several primitives represent a concept as a group. Batches of variables consistently passed around together. → **Encapsulation**

CODE SMELL AND REFACTORING SOLUTIONS

- Time Dependencies (5) and Half-Baked Objects (6)
  - **(5)** A class' methods must be called in a specific order.
  - **(6)** Objects must first be constructed, then initialized with a method call, then used. → For both, the class may have too many responsibilities. → **Split**
- Coddling Nulls (7)
  - Null references : a particular challenge to programmers
  - Problem : method, that may receive null reference, will return null itself
  - **Null reference should not be propagated.**
  - **We need a fail fast strategy**
    - Do not allow null as a parameter to any method, constructor or attribute.
    - Null can be allowed only if it has explicitly defined semantics.
    - Throw exceptions rather than returning null.

F.A.Q. ABOUT CODE REFACTORING

- How often should we refactor ?
  - Constantly, every day.
- Shouldn't we design our code correctly from the beginning rather than refactoring (rework) ?
  - Perfect design is impossible for large systems.
  - Don't bemoan design errors, celebrate your ability to fix them !
- Will large design changes conflict with other team members ?
  - Yes, it may conflict. So you need a better management of timing issue.
- Do we need to do **test** refactoring ?
  - Yes absolutely. Tests have to be maintained just as much as production code does, so they are valid targets for refactoring, too.

CONTINUOUS INTEGRATION (CI)

- **We keep our code ready to ship.**
- Problem : hidden delay
  - Between when the team says "we're done" and when the software is ready to ship.
  - Examples of little things to do before shipping : merging everyone's pieces together, creating an installer, prepopulating the database…
- We often forget how long these things take !
- **The ultimate of CI is to be able to deploy at any time.**
- **Key point** : Be technologically ready to release even if you are not functionally ready to release.

PRACTICING CI

- Integrate your code every **few hours**.
- Keep your build, tests and other release infrastructure **up-to-date**.
- Good practice in some industrial projects : a firm rule (optional)
  - You have to integrate before going home.
  - If you can't integrate → something goes wrong → abandon what you did
  - Start fresh the next day.
- This is a harsh rule but it may actually work very well.
- **Never (almost never) break the build and agree with that as a team !**
- CI needs strict environment configuration : hardware and software
  - https://depinfo.u-cergy.fr/~tliu/ens/gpi/gpi6-intégration-continue.pdf

SUPPORTING CI : UNIT TESTS (UT) & MOCK OBJECTS (MO)

- **Unit tests focus just on the class or method at hand.**
- They run entirely in memory, which makes them very fast.
  - Average UT run speed : **100 UTs per second**.
- A test is not a UT if :
  - It talks to a database.
  - It communicates across a network.
  - It touches the file system.
  - You have do to special things to your environment to run it. Ex. editing configuration files)
- Mock object allow your test to substitutes its own object (MO) for an object that talks to the outside world (DB, network, etc.).
  - Don't abuse → well designed, decoupled system

SUPPORTING CI : FOCUSED INTEGRATION TEST (FIT) & END-TO-END TESTS (EET)

- **FIT** are the tests which test just **one interaction** with the outside world (DB, network, file system, etc.)
- Prepare the external dependency carefully :
  - Tests should run exactly **the same way** every time
  - Intermittent failures of FIT are technical debts.
  - We don't need too many FIT with N-tiers architecture (well decoupled).
- To ensure that UT and FIT mesh perfectly : **EET**
  - EETs exercise large swaths of the system.
  - EETs are very slow : seconds even minutes per test
  - **Attention** : don't use exploratory testing to find bugs !
- The proportion of EETs should be minimized
  - **We need a well design system, constructed by TDD strategy.**

SUPPORTING CI : COLLECTIVE CODE OWNERSHIP

- **"Of course nobody can understand it… its job security !" --- old joke.**
- **We are all responsible for high-quality code.**
- **A real risk** : What happens when a critical person goes on holiday, gets sick ? How much time will you spend training a replacement ?
- Fix problems no matter where you find them
  - If you encounter code duplication, unclear names, or even poorly designed code, we don't care about who wrote it. Fix it !
- Always leave the code a little better than you found it.