Pair programming, refactoring, test driven development

[DT-0540] Metodi di sviluppo agile

Daniele Di Pompeo

daniele.dipompeo@univaq.it

UNIVERSITÀ DEGLI STUDI DELL'AQUILA A.A. 2024-2025

Pair programming

https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5283

Write all production programs with two people sitting at one machine

https://distantjob.com/blog/2017-08-16-pair-programming-why-you-should-care-about-it-and-how-to-do-it-remotely/

- Today pair programming is occasionally practiced
- Few companies have applied pair programming to “all ” their developments for very long.
- But many programmers have found some dose of pair programming beneficial, and the technique deserves to be known.

Pair programming concepts

The two partners "paired" should be closely involved in the work

- The Driver: handle the keyboard to compose the program, all the time expressing his her thought process and uncertainties aloud
- The Navigator: observer position, while the driver is typing. He/she reviews the code on-the-go, gives directions and shares thoughts. The navigator also has an eye on the larger issues, bugs, and makes notes of potential next steps or obstacles.

Pair programming concepts

This is a **peer process**, so the partners should **regularly reverse roles** advertised benefits keeping one another on task brainstorming on improvements clarifying ideas holding each other accountable enabling one partner to take the initiative when the other is stuck.

Pair programming main idea

Key idea: if two people produce software that is more than twice as good, then we get a productivity gain, not a loss

Pair programming main idea

Key idea: if two people produce software that is more than twice as good, then we get a productivity gain, not a loss

Typical productivity figures in the software industry is measured in SLOC (source lines of code, a metric that everyone criticizes — and that everyone uses) around 20 SLOCs per person per day.

Writing twenty lines of code takes only a few minutes

Explanation — clear to everyone in the field and confirmed by numerous studies developers spend most of their time on other tasks, in particular on thinking about the code they will write, and correcting code that was not right the first time around.

Pair programming vs Mentoring

Pair programming **should not be** used as a mentoring technique by pairing a junior programmer with an experienced one, as a training experience.

Mentoring is a fruitful technique, but its **primary** purpose is **education**, not software production.

Pair programming is peer programming

You get feedback from someone who is roughly at your own level of expertise

Pair programming vs Mentoring

Mixing pair programming with mentoring is no a good idea

The junior member will slow down the senior member, who instead of getting help for the most difficult challenges of the job will find himself repeatedly explaining the easiest parts.

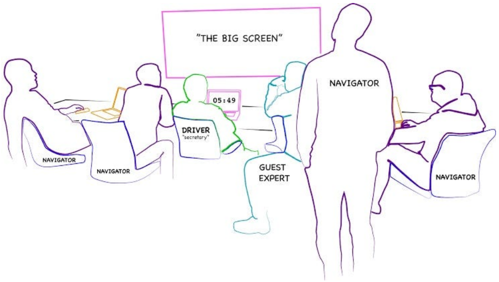
On the other side, the supposed teacher, thinking of the expected result and the deadline, will not explain more than strictly needed.

How to do effective pair programming

https://gds.blog.gov.uk/2018/02/06/how-to-pair-program-effectively-in-6-steps/

https://martinfowler.com/articles/on-pair-programming.html

Mob programming



The entire team works as a team together on one task at the time.

One team $\rightarrow$ one (active keyboard) $\rightarrow$ one screen

Mob programming

https://en.wikipedia.org/wiki/Mob_programming

In addition to software coding, a mob programming team can work together to tackle other typical software development tasks.

Some examples include: defining requirements, designing, testing, deploying software, and working with subject matter experts.

Almost all work is handled in working meetings or workshops, where all the people involved in creating the software are considered to be team members, including the customer and business experts.

Mob programming also works for distributed teams in the same virtual space using screen sharing technology

Refactoring

Coding standards

https://github.com/google/styleguide

Guidelines for a specific programming language Code conventions are important for a number of reasons:

Coding standards

Coding standards

https://github.com/google/styleguide

- Guidelines for a specific programming language
- Code conventions are important for a number of reasons:
  - 40%–80% of the lifetime cost of a piece of software goes to **maintenance**
  - Hardly any software is maintained for its whole life by the original author
  - Improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
  - If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

Refactoring

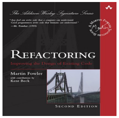
"[Refactoring is] the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" - *M. Fowler*

Problem: code is hard to evolve and maintain

Solution:

- refactor to keep the code simple
- good engineering keeps code simple and easy to understand

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." *M. Fowler*



Refactoring

Changes made to a system that:

- Do not change observable behavior and the semantics of the problem
- Remove duplication or needless complexity
- Enhance software quality
- Make the code easier and simpler to understand
- Make the code more flexible
- Make the code easier to change, etc. . .
- Make the code aligned with design/architecture

Refactoring

Changes made to a system that:

- Do not change observable behavior and the semantics of the problem
- Remove duplication or needless complexity
- Enhance software quality
- Make the code easier and simpler to understand
- Make the code more flexible
- Make the code easier to change, etc. . .
- Make the code aligned with design/architecture

NOTE: the importance of

- Tests
- Metrics

Refactoring: Why?

- **Improve Code Quality:**
  - Refactoring makes code easier to read and understand.
  - Clean code is easier to maintain and extend.
- **Reduce Technical Debt:**
  - Over time, quick fixes and shortcuts can make the codebase harder to work with.
  - Refactoring addresses these issues to prevent long-term problems.
- **Facilitate Future Changes:**
  - Well-structured code makes it easier to add new features or fix bugs.
  - Refactoring ensures that future changes can be made with confidence.
- **Improve Collaboration:**
  - Refactoring helps create code that can be easily understood by others in the team.
  - Encourages shared ownership of the codebase.

by courtesy of ChatGPT

Refactoring: When

- **Before Adding New Features:**
  - If the code is hard to understand or messy, refactor first to make adding new functionality easier and safer.
- **After Fixing Bugs:**
  - If a bug was caused by unclear or poorly structured code, refactor it to prevent future issues.
- **When the Code is Repetitive or Complex:**
  - If you find yourself duplicating code or struggling to follow the logic, it's a good time to refactor.
- **During Code Reviews:**
  - If peers point out areas that are hard to understand, refactor to improve clarity.
- **Before Project Deadlines:**
  - Ensure the code is clean and organized before submitting, making it easier to explain and evaluate.

by courtesy of ChatGPT

When not to do refactoring?

- When the tests are failing
- When you should just rewrite the code
- When you have impending deadlines

What is a Bad Smell in Code?

Definition:

- refers to a symptom in the source code that indicates deeper problems.
- while not necessarily an error, a bad smell often points to potential issues with maintainability, readability, or flexibility of the code.
- typically make the code harder to understand or modify and increase the risk of introducing bugs during future changes.

Examples of Common Code Smells:
see others in Appendix

- Duplicated Code
- Long Methods
- Large Classes

by courtesy of ChatGPT

Refactoring Drawbacks

When taken too far

- Incessant tinkering with code
- Trying to make it perfect

Attempting refactoring when the **tests don't work** – or **without tests** – can lead to dangerous situations!

Refactoring published interfaces propagates to external users relying on these interfaces

Refactoring

Why Developers Fear Refactoring?

- "I don't understand the code enough to do it"
- Short-term focus (Adding a new working feature is cooler!)
- Not paid for overhead tasks such as refactoring?

Solutions:

- Test, test and test again, with good test code.
- Learn to appreciate elegant code (and find good metrics)
- Teach the benefits of better code (to your colleagues)

Code refactoring: external material

Videos + examples of code

- http://www.newthinktank.com/2013/01/code-refactoring/
- http://www.newthinktank.com/2013/01/code-refactoring-2/

Refactoring Kata

- https://kata-log.rocks/refactoring

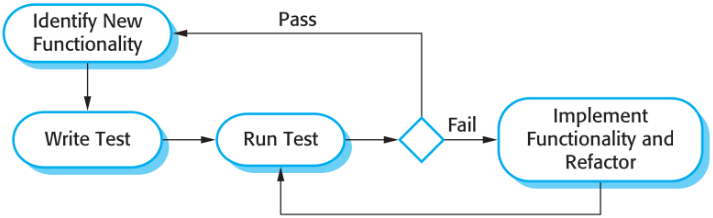
Test Driven Development

What is Test-Driven Development (TDD)?

Definition:

- TDD is a software development process where tests are written before the actual code.
- The cycle follows a repeated process of writing a test, writing the code to pass the test, and refactoring.

TDD Cycle


Identify New Functionality
Pass
Write Test
Run Test
Fail
Implement Functionality and Refactor

The TDD Cycle

Three Steps:

1. **Red:** Write a failing test (since the code doesn't exist yet).
2. **Green:** Write the minimum code necessary to pass the test.
3. **Refactor:** Clean up the code while keeping it functional.

Repeat: This process is repeated for each small unit of functionality.

Why Use TDD?

Benefits of TDD:

- **Better Code Quality:** TDD encourages developers to write cleaner, simpler code.
- **Fewer Bugs:** Writing tests early catches bugs before they become embedded in the system.
- **Confidence in Refactoring:** Tests ensure that changes in the code do not break existing functionality.
- **Faster Debugging:** Failures are identified as soon as they occur, making them easier to fix.

Writing the First Test

Steps to Writing the First Test:

- Identify a small unit of functionality to be implemented.
- Write a test for that unit (start simple).
- Run the test, and confirm that it fails (because the code doesn't exist yet).

Example: Writing a test to check if a function returns the correct sum of two numbers.

Implementing the Code

Next Step After Writing the Test:

- Write just enough code to make the test pass.
- The focus should be on passing the test, not on writing the perfect solution.
- Keep the code simple; refactoring will come later.

Goal: Move the test from red (failing) to green (passing).

Refactoring the Code

Once the Test Passes:

- Clean up the code while ensuring the test remains green (passing).
- Refactor for readability, maintainability, and performance.
- Make sure no functionality changes during refactoring.

Test-Driven Refactoring: The tests give confidence that the refactor won't introduce **new** bugs.

Benefits of TDD for Teams

- **Improving Collaboration:** Everyone understands the expected behavior of the code.
- **Enabling Continuous Integration:** Tests act as documentation and reduce integration issues.
- **Building Trust:** Confidence in the codebase grows as developers see tests consistently passing.
- **Fostering Discipline:** TDD encourages small, manageable iterations.

Best Practices in TDD

Tips for Effective TDD:

- Keep tests small and focused on one piece of functionality.
- Use descriptive names for tests that describe the behavior being tested.
- Write tests that are easy to read and maintain.
- Run tests frequently to catch issues early.
- Refactor tests along with the code when necessary.

Challenges of TDD

Common Challenges:

- **Time-Consuming at First:** Writing tests before the code can feel slow for new adopters.
- **Changing Requirements:** Tests may need frequent updates if requirements change often.
- **Test Maintenance:** Over time, tests themselves may require refactoring or improvements.
- **Initial Learning Curve:** Developers need time to learn how to write good, testable code.

Pair programming, refactoring, test driven development

[DT-0540] Metodi di sviluppo agile

Daniele Di Pompeo

daniele.dipompeo@univaq.it

UNIVERSITÀ DEGLI STUDI DELL'AQUILA A.A. 2024-2025

Appendix - Bad Smells

Long Method

Description:

- A method that is too long and tries to do too much.
- Hard to understand and maintain.
- Makes the code harder to test.

Solution: Extract smaller methods.

Large Class

Description:

- A class that has grown too large and has too many responsibilities.
- Violates the Single Responsibility Principle.

Solution: Split the class into smaller, more cohesive classes.

Duplicated Code

Description:

- Code that is repeated in multiple places.
- Increases maintenance effort.
- Leads to inconsistency when the code is modified.

Solution: Refactor the code to use a single source.

Long Parameter List

Description:

- A method or constructor that takes too many parameters.
- Hard to read and understand.
- Increases the likelihood of errors.

Solution: Use objects to group related parameters.

Feature Envy

Description:

- A method that seems more interested in the data of another class than its own.
- Results in low cohesion.

Solution: Move the method to the class it is envious of.

Data Clumps

Description:

- Groups of data that tend to be passed together or appear together in several places.
- Indicates missing abstractions.

Solution: Create a class or structure to encapsulate the data.

Primitive Obsession

Description:

- The use of primitive types (e.g., int, string) to represent domain ideas.
- Leads to scattered behavior and inconsistent handling.

Solution: Create value objects for specific domain concepts.

Lazy Class

Description:

- A class that isn't doing enough to justify its existence.
- Adds unnecessary complexity.

Solution: Inline the class or remove it entirely.

Switch Statements

Description:

- Switch statements or large conditionals that appear in many places.
- Can become difficult to manage when new cases are added.

Solution: Use polymorphism to avoid repetitive switch statements.

Excessive Comments

Description:

- Over-reliance on comments to explain code.
- Usually indicates unclear code.

Solution: Write self-explanatory code by improving variable names and structure.

Data Class

Description:

- Classes that only have fields and getters/setters but lack behavior.
- Often a sign of missing encapsulation.

Solution: Move behavior that uses the data into the class.

Speculative Generality

Description:

- Code that is overly generalized for potential future use cases.
- Leads to unnecessary complexity.

Solution: Remove unused abstractions and make the code simpler.

Inappropriate Intimacy

Description:

- A class that knows too much about another class's internal details.
- Violates encapsulation and leads to high coupling.

Solution: Reduce direct interaction between the two classes.

Message Chains

Description:

- When objects call methods on other objects, creating long chains.
- Makes the code fragile and tightly coupled.

Solution: Use method extraction or introduce intermediate objects.

Middle Man

Description:

- A class that delegates almost all of its work to another class.
- Adds unnecessary layers to the system.

Solution: Eliminate the middle man and delegate directly.

Refused Bequest

Description:

- A subclass inherits methods or data that it does not need or use.
- Violates the Liskov Substitution Principle.

Solution: Refactor the inheritance hierarchy or consider composition.

Temporary Field

Description:

- Fields that are only used in certain situations or configurations.
- Leads to complex class logic and confusion.

Solution: Use a separate class or method to handle the specific case.

Appendix - Refactoring Catalog

Extract Method

Description:

- Move part of a long method into a new method to make the code more readable and reusable.

Use when:

- A method is too long or doing too many things.
- You see repeating code fragments.

Inline Method

Description:

- Replace a method call with the content of the method to reduce unnecessary indirection.

Use when:

- A method's body is just as clear as the name.
- The method is used only once or the logic is trivial.

Move Method

Description:

- Move a method from one class to another if it's more related to another class.

Use when:

- A method seems more interested in another class than its own.
- To reduce dependency between classes.

Rename Method

Description:

- Change the name of a method to better reflect its purpose.

Use when:

- A method name does not clearly describe its function.

Extract Class

Description:

- Move part of a class's responsibilities into a new class to simplify the original class.

Use when:

- A class has too many responsibilities or is becoming too large.

Inline Class

Description:

- Merge a class into another class when it no longer justifies its existence.

Use when:

- A class is too small or serves as a middleman for another class.

Replace Temp with Query

Description:

- Replace temporary variables with methods that calculate the value.

Use when:

- A temporary variable holds a value that can be derived from other methods.

Introduce Parameter Object

Description:

- Encapsulate a group of parameters into a single object.

Use when:

- A method or constructor takes too many parameters.

Replace Conditional with Polymorphism

Description:

- Use subclasses to replace conditionals that switch behavior based on object type.

Use when:

- Conditional logic depends on an object's type.

Remove Dead Code

Description:

- Delete unused code that no longer serves any purpose.

Use when:

- Code is never called or used anymore.

Introduce Null Object

Description:

- Create a null object to handle cases where an object has no value, avoiding null checks.

Use when:

- You have frequent null checks in the code.

Decompose Conditional

Description:

- Extract complex conditionals into separate methods to clarify intent.

Use when:

- Conditionals become too complex or unreadable.

Replace Magic Numbers with Constants

Description:

- Replace hardcoded numbers with named constants to improve readability.

Use when:

- You find numbers in the code whose meaning isn't immediately clear.

Encapsulate Collection

Description:

- Ensure that a collection is only modified through methods that encapsulate it.

Use when:

- A collection should be protected from external modification.

Push Down Method

Description:

- Move a method from a superclass to a subclass if only one subclass uses it.

Use when:

- A method in the superclass is only relevant to certain subclasses.

Pull Up Method

Description:

- Move a method from a subclass to the superclass when it is common to all subclasses.

Use when:

- Multiple subclasses share the same method.

Encapsulate Field

Description:

- Replace public fields with getter and setter methods to control access.

Use when:

- You want to hide the internal representation of a class.

Introduce Assertion

Description:

- Add assertions to document and enforce assumptions about program behavior.

Use when:

- Critical assumptions about the program should be enforced during execution.

Consolidate Conditional Expression

Description:

- Combine multiple conditionals that result in the same outcome into a single condition.

Use when:

- Several conditionals lead to the same result.

Replace Constructor with Factory Method

Description:

- Replace a complex constructor with a static factory method to simplify instantiation.

Use when:

- Object creation logic is complex or differs based on conditions.

Remove Middle Man

Description:

- Remove a class that delegates all its work to another class.

Use when:

- A class serves as an unnecessary intermediary between two other classes.