InstaDeep™

The Pragmatic Programmer

Presented By: **Lina Ben Salem**

Plan

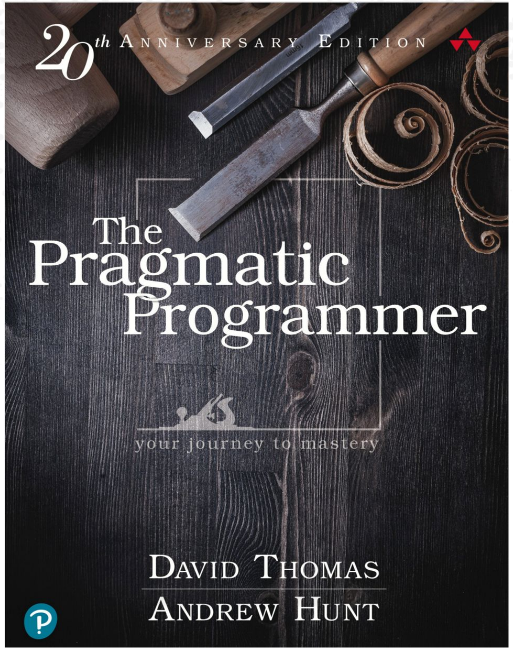
Introduction

Introduction

The Pragmatic Programmer: From Journeyman to Master (1999)

The Pragmatic Programmer: Your Journey to Mastery (2019)

Authors: **David Thomas & Andrew Hunt**



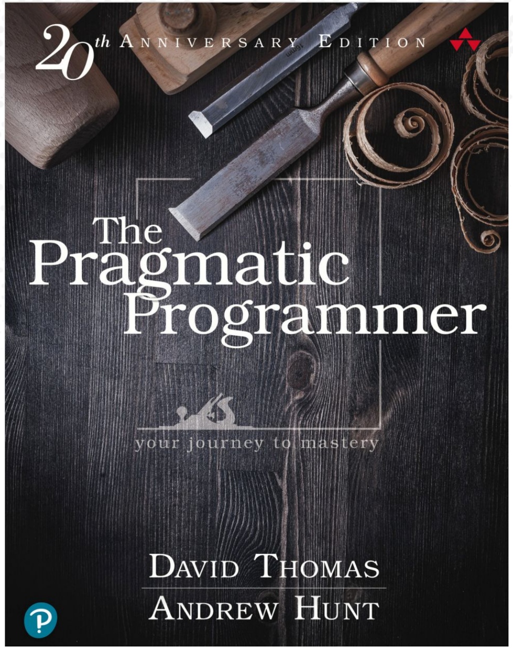
InstaDeep™

Introduction

How Did It Begin?

David and Andrew found themselves mentioning the same things whenever they worked on different projects, so they started writing down notes.

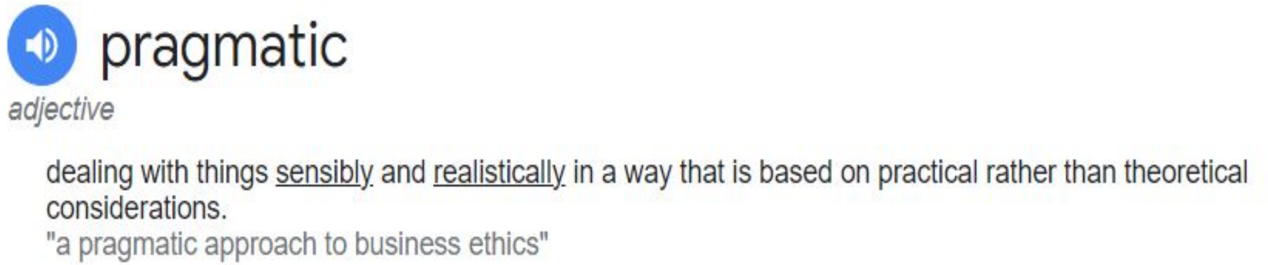
=> The Pragmatic Programmer



InstaDeep™

Introduction

Why The Pragmatic Programmer?


pragmatic

adjective

dealing with things sensibly and realistically in a way that is based on practical rather than theoretical considerations.
"a pragmatic approach to business ethics"

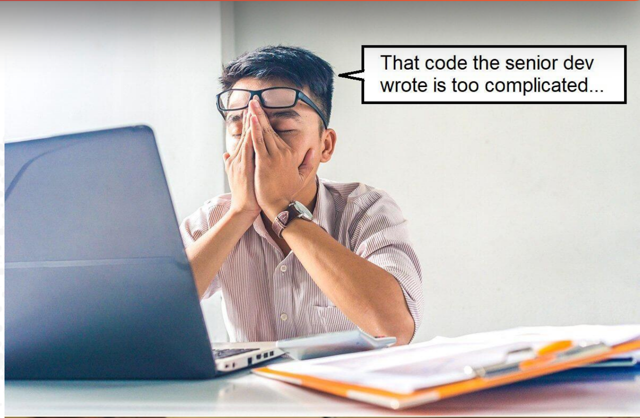
Introduction

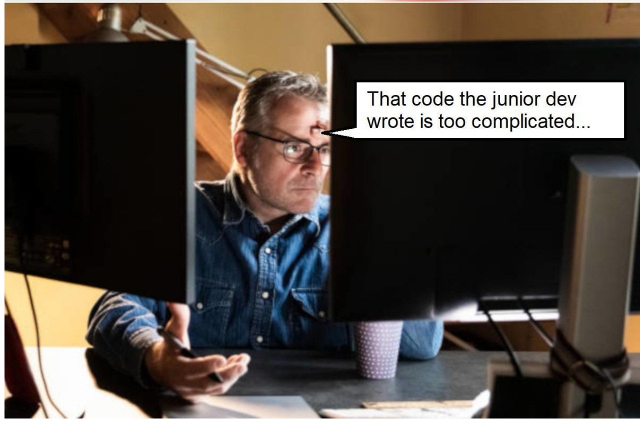
Story Telling

➢ The book uses analogies and short stories to present the development methodologies

Who Is It For:

➢ Beginners/Experienced Developers
➢ People who want to be more productive and care about their craft





InstaDeep™

I. Pragmatic Programmer Characteristics

I. Pragmatic Programmer Characteristics

- **Early/Fast Adapter:** passionate about experiencing technology and learning new things
- **Critical Thinker:** does not accept things as they are and try to understand the underlying nature of things
- **Realistic:** see things as they are and solve them
- **Jack-of-all-trades:** familiar with a broad of technologies

InstaDeep™

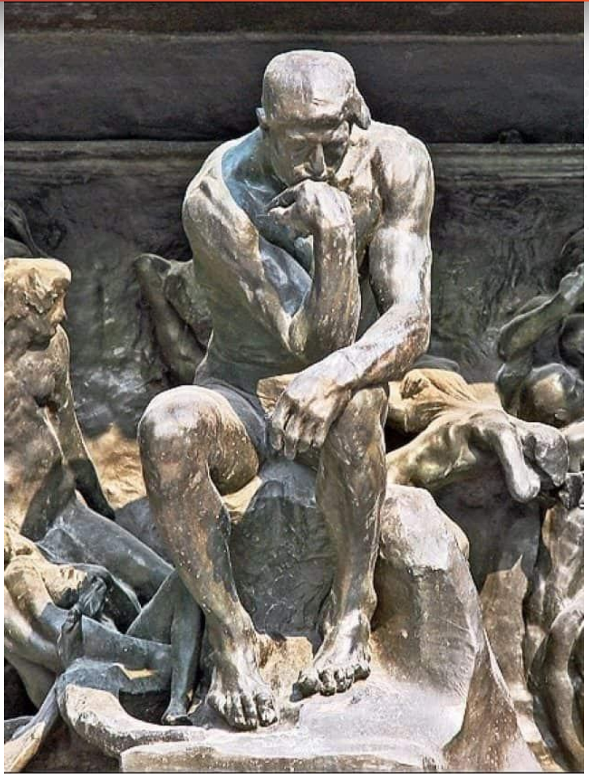
Key Principles

InstaDeep™

II. Keys Principles

1. Have a Pragmatic Philosophy

- ➢ Have the style of thinking beyond the immediate problem
- ➢ Always think about the bigger picture
- ➢ Be flexible with changes
- ➢ Never stop learning



II. Keys Principles

2. Trust Your Intuition (The Lizard Brain)

➢ With time you will realize you'll be facing similar challenges, so trust your intuition with that

3. Don't Code By Coincidence

➢ You have to understand why the code worked at first to know how to debug it and fix it if something goes wrong

II. Keys Principles

4. Don’t Gather Requirements, DIG FOR THEM

➢ So often the user doesn't know his needs or even express them, so you have to know how to extract the information from the stakeholders and keep the feedback channel open with them to make sure their needs were met

➢ Programmers help people understand what they want (they focus on edge cases)

II. Keys Principles

5. Be Responsible

- “The cat ate my source code”
- Admit your ignorance and mistakes
- It helps with growth
- Build team trust

II. Keys Principles

6. Refactor Regularly

- ➢ Software entropy
- ➢ Software projects tend to get more complicated to handle
- ➢ Improve code without changing the external behavior
- ➢ This improves code readability, reduces complexity, and enhances maintainability

II. Keys Principles

7. Never Leave A Broken Window

- ➢ Whenever there's something broken fix it because it can be the beginning of the damage
- ➢ Abandoned house example
- ➢ Teams should not allow broken windows

II. Keys Principles

8. Stone Soup Story

➢ Whenever you feel you are out of resources remember the “stone soup” story

➢ The term “startup fatigue”

InstaDeep™

II. Keys Principles

9. Always Keep the Big Picture In Mind

- The “boiled frog story”
- Don’t be like the fabled frogs, always keep an eye on the big picture. Constantly review what’s happening around you, not just what you are doing

InstaDeep™

II. Keys Principles

10. Good Enough Software

- ➢ Meet requirements (user-centric approach)
- ➢ Continues delivery (user feedback)
- ➢ Avoid over-engineering
- ➢ Avoid excessive perfectionism
- ➢ Balance trade-offs (time, cost, functionality, quality)

II. Keys Principles

11. Reversibility

➢ Change is inevitable => There are no final decisions

➢ The ability to undo or roll back changes made to the software system

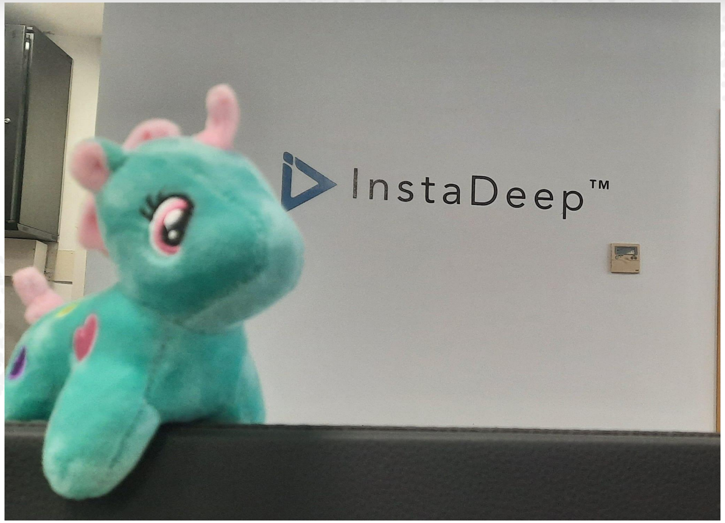
➢ Make sure that reversing the decision is cheap (flexible architecture)

InstaDeep™

II. Keys Principles

12. Debugging

- ➢ Focus on the problem, not on the blaming
- ➢ Don’t panic, think
- ➢ Fix the cause of failure, not its symptoms
- ➢ Use a good debugger with visualization capabilities
- ➢ Explain partial insights to someone else (or even the duck toy)
- ➢ Once you find the problem, add the test that would have caught it and look for further similar problems
- ➢ If it took too long, reflect on why (refactor your code)



InstaDeep™

II. Keys Principles

13. Assertions

- ➢ Assertions in programming are statements that check if a given condition is true at a specific point in the code
- ➢ They act as sanity checks to ensure that assumptions made during the development process are valid
- ➢ In Python, it uses the assert statement. The general syntax of an assertion is:
  asset condition, message
- ➢ **condition** is the expression or condition that is expected to be true. If the condition evaluates to false, an AssertionError is raised
- ➢ **message** is an optional string that can be used to provide additional information about the assertion

InstaDeep™

II. Keys Principles

13. Assertions

➢ **Example**

```
def divide_numbers(a, b):
    assert b !=0, "Cannot divide by zero!"
    return a / b
```

=> Catch potential division by zero errors during development and testing

II. Keys Principles

13. Assertions

➢ **Best practices:**

- Use assertions to check for conditions that are not supposed to occur
- Keep assertions focused on code assumptions and invariants
- Avoid using assertions for input validation or error handling that should be handled by proper exception handling mechanisms

➢ Don’t turn them off even in production, because even test can’t replace them even if you feel they make the program kinda heavy

InstaDeep™

II. Keys Principles

14. When to use Exceptions

- ➢ Junior programmers confuse exceptions and “if” statements
- ➢ Exceptions and if statements serve different purposes and are used in different contexts
- ➢ If statements should be used for making decisions and controlling the flow of execution based on conditions within the normal program flow
- ➢ Exceptions should be used for exceptional conditions or errors that deviate from the normal program flow

InstaDeep™

II. Keys Principles

14. When to use Exceptions

Here are some situations where you should consider using exceptions:

1. Resource management: Exceptions can be helpful in managing resources like files, database connections, or network sockets. If an exception occurs during the use of a resource, it can be caught and the necessary cleanup operations can be performed before terminating the program

Example: File Handling

```
try:
    file = open("example.txt","r")
    #Perform file operations
    file.close()
except FileNotFoundError:
    print("File not found! ")
except IOError as e:
    print("I/O error occured:", str(e))
```

II. Keys Principles

14. When to use Exceptions

2. Input validation: Exceptions can be used to validate input and handle invalid or unexpected data. For instance, if a function expects a positive integer as input and receives a negative value, it can throw an exception to indicate the invalid input

Example

```
try:
    age = int(input("Enter your age: "))
    if age < 0:
        raise ValueError("Age cannot be negative!")
except ValueError as e:
    print("Invalid input:", str(e))
```

II. Keys Principles

14. When to use Exceptions

3. Boundary conditions: Exceptions can be used to handle boundary conditions or exceptional situations that are not part of the normal flow of execution. For example, if an index is out of bounds in an array or a division by zero is attempted, an exception can be thrown

Example

```
try:
    result = x / y
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

InstaDeep™

II. Keys Principles

14. When to use Exceptions

4. Third-party interactions: When working with external libraries or services, exceptions can be used to handle errors or exceptional responses from those components. This allows you to gracefully handle unexpected behaviors and provide appropriate feedback or recovery mechanisms

Network request exceptions are a subset of third-party interaction exceptions that specifically pertain to communication over a network

InstaDeep™

II. Keys Principles

14. When to use Exceptions

In this example, the code performs a GET request to a URL using the Requests library. If an exception of type RequestException occurs during the network operation, it is caught and an error message is displayed

Example

```
import requests

try:
    response = requests.get("https://www.example.com")
    # Process the response

except requests.exceptions.RequestException as e:
    print("An error occured:", str(e))
```

II. Keys Principles

14. When to use Exceptions

In exceptions we can use, “try … except” and “try… if … raise”

- ➢ “try….except” is used to handle and catch exceptions that occur naturally preventing the program from crashing
- ➢ “try….if….raise” is used to raise an exception based on a specific condition within the code

Example

```
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Exception handling for ZeroDivisionError
    print("Error: Division by zero")
```

```
try:
    x = 5
    if x > 10:
        raise ValueError("x should be less than or equal to 10")
except ValueError as e:
    print("Error:", str(e))
```

II. Keys Principles

14. When to use “If” statement

- ➢ **Purpose:** if statements are used for conditional branching and making decisions based on specific conditions
- ➢ **Conditional logic:** if statements evaluate a condition and execute a block of code if the condition is true
- ➢ **Control flow control:** if statements do not interrupt the normal control flow of the program
- ➢ **Usage scenarios:** Use if statements when you need to evaluate conditions and make decisions based on those conditions

II. Keys Principles

➢ **Examples** include checking user input, validating data, or implementing conditional behavior and checking multiple conditions

```
username = input("Enter your name: ")
password = input("Enter your password: ")
if username =="admin" and password =="password":
    print("Login successful!")
else:
    print("Invalid username or password.")
```

```
age = int(input("Enter your age: "))
if age>= 18:
    print("Your are eligible to vote!")
else:
    print("Your are not eligible to vote yet.")
```

```
num = int(input("Enter a number: "))
if num > 0:
    print("The number is positive.")
elif num == 0:
    print("The number is zero.")
else:
    print("The number is negative.")
```

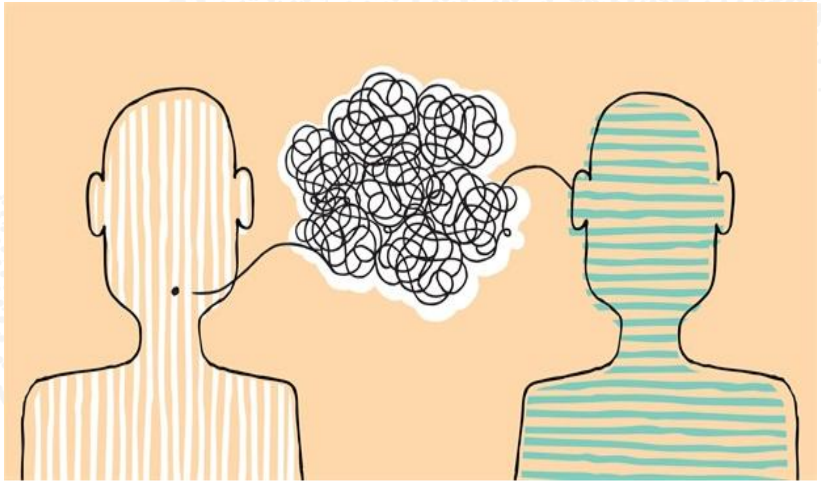
```
num = int(input("Enter a number: "))

if num % 2 == 0 and num > 0:
    print("The number is a positive even number.")
elif num % 2 != 0 and num > 0:
    print("The number is a positive odd number.")
else:
    print("The number is either zero or negative.")
```

II. Keys Principles

15. Communicate

- ➢ Developers communicate whether in meetings, conversations, presentations, or even code comments
- ➢ It is not about what you say, it is about :
- ● How to say it
- ● What you want to say
- ● Know your audience/person you are talking to (emotional intelligence)
- ● Choose your moment
- ● Be a listener
- ● Get back to people



Best Practices

InstaDeep™

III. Best Practices

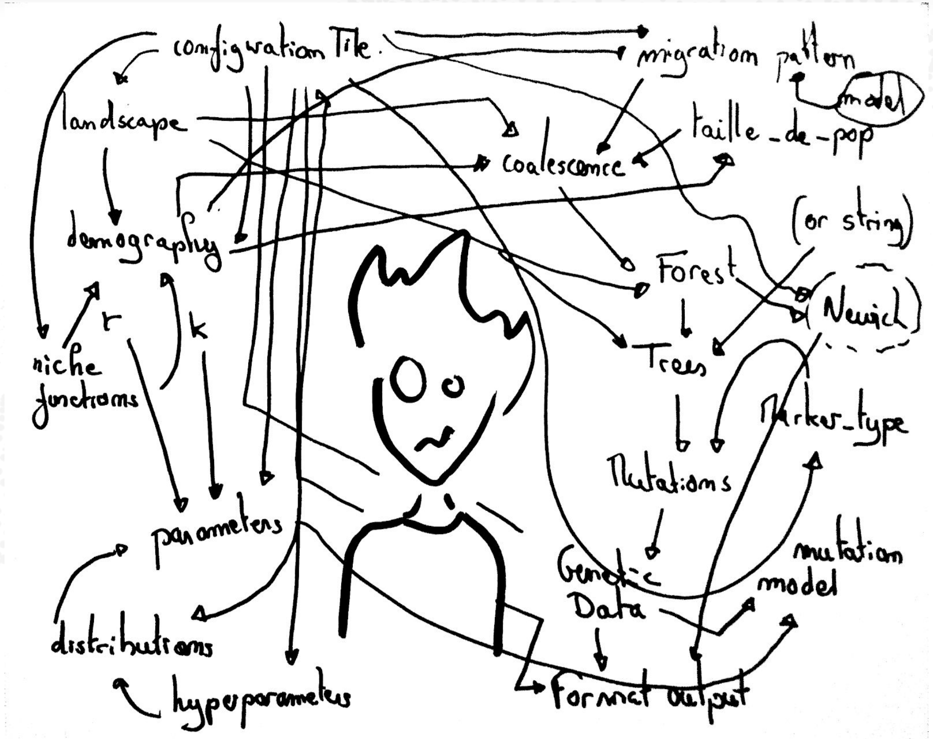
1. DRY (Don't repeat yourself)

- ➢ Avoid duplication and put things into functions and reuse them
- ➢ Keep refactoring to avoid dry code
- ➢ Often developers do it unintentionally that's why you have to pay attention during daily scrum meetings and read your colleagues' code and the project documentation

III. Best Practices

2. Orthogonality (Decoupling)

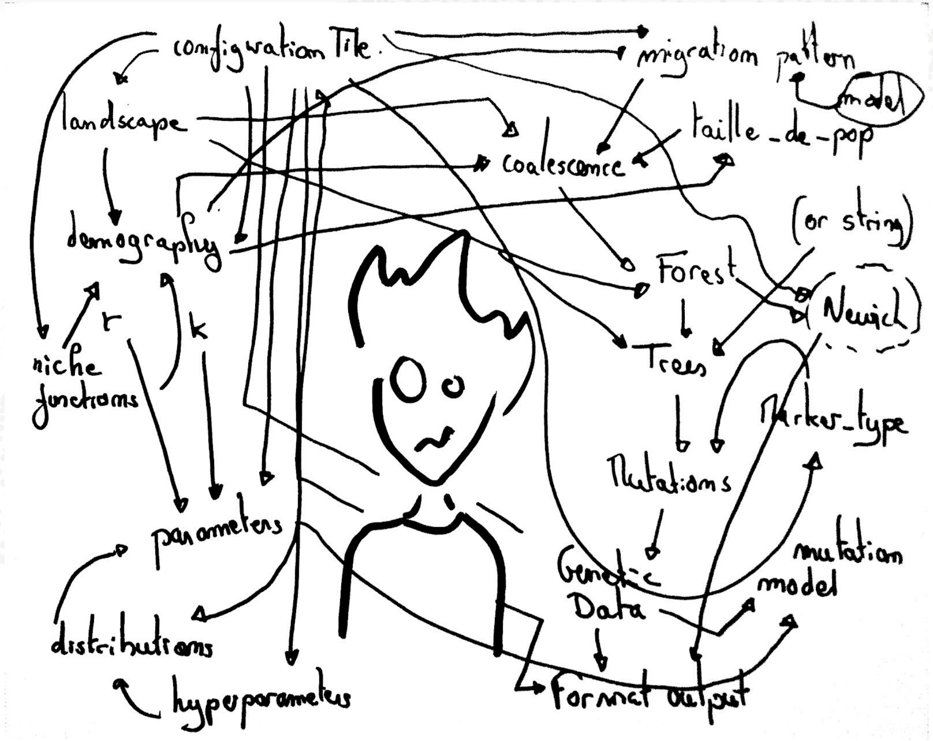
- ➢ Non-orthogonal is like a helicopter system, too complicated , touch one thing and something else gets affected
- ➢ Orthogonal systems = elements that aren't interconnected, interfaces are linked but deep inside each on its own so when you fix something you just focus on it
- ➢ Code becomes responsive to change



III. Best Practices

2. Orthogonality (Decoupling)

- ➢ Good design makes this ETC
- ➢ It increases productivity and reduces risks
- ➢ Tricky test to check orthogonality, make a slight change and see how it affects the rest of the system



III. Best Practices

2. Orthogonality (Decoupling)

Example 1: Interdependent Functions

These functions have a dependency on the `numbers` list

Non-orthogonal code

```
def add_number(numbers, number):
    numbers.append(number)
    return numbers

def remove_number(numbers, number):
    numbers.remove(number)
    return numbers
```

Orthogonal code

```
def add_number(numbers, number):
    new_numbers = numbers.copy()
    new_numbers.append(number)
    return new_numbers

def remove_number(numbers, number):
    new_numbers = numbers.copy()
    new_numbers.remove(number)
    return new_numbers
```

III. Best Practices

2. Orthogonality (Decoupling)

Example 2: Implicit Dependencies

These functions have a dependency implicit dependency on user information, If the authentication mechanism or user data structure changes, both functions may need to be modified

Non-orthogonal code

```
def authenticate_user(username, password):
    #Authenticates user credentials
    pass
def log_in_user(username):
    #Logs in the user and updates user status
    pass
```

Orthogonal code

```
def authenticate_user(username, password):
    #Authenticates user credentials
    pass
def log_in_user(user):
    #Logs in the user and updates user status
    pass
```

III. Best Practices

2. Orthogonality (Decoupling)

Example 3: Stateful functions

These functions have a shared state ‘the total variable’, and modifying it in one function can affect the other

Non-orthogonal code

```
total = 0
def add_to_total(amount):
    global total
    total += amount

def substract_from_total(amount):
    global total
    total -= amount
```

Orthogonal code

```
class TotalCalculator:
    def __init__(self):
        self.total = 0

    def add_to_total(self, amount):
        self.total += amount
def substract_from_total(self, amount):
    self.total -= amount
```

III. Best Practices

2. Orthogonality (Decoupling)

Example 4: Order-dependant functions

These functions have a specific order of execution, changing the order or skipping a function leads to incorrect results

Non-orthogonal code

```
def initilize_database():
    # Initializes the database

def import_data():
    # Imports data into the database

def perform_calculations():
    # Performs calculations based on imported data
```

Orthogonal code

```
def initilize_database():
    # Initializes the database

def import_data(database):
    # Imports data into the database

def perform_calculations(data):
    # Performs calculations based on imported data
```

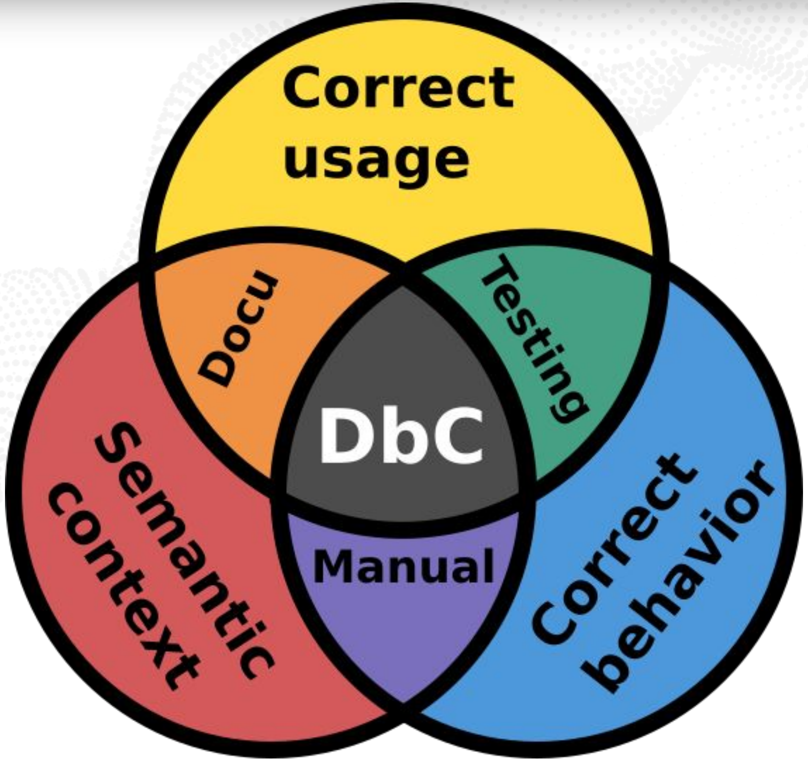
III. Best Practices

Orthogonality + DRY = Flexibility

III. Best Practices

3. DBC (Design By Contract)

- ➢ Explicitly define the expected behavior, preconditions, and postconditions of a component which would lead to catching errors early ensuring components interact correctly



III. Best Practices

4. Write Tests

➢ It is mandatory to verify the correctness and robustness of the code, catch bugs early, facilitate refactoring, and provides documentation for expected behavior

➢ **Types of tests**:

- Unit testing: for each module individually
- Integration testing
- Performance testing (stress testing)
- Testing the tests: make sure they would catch bugs

Tools & Technics

InstaDeep™

IV. Tools & Technics

Plain Text :

Human understandable structure, it is best for analysis and manipulation

Shell

It is important, plus you can personalize it

Editor

Shortcuts save so much time

Version Control

Go back in history
give access to all the team to participate in the same version of the project

IV. Tools & Technics

1. Prototyping

- ➢ It is about creating a partial or scaled-down version of software to gather feedback, validate ideas, explore potential design solutions, demonstrate a functionality or interaction flow of the final product
- ➢ It helps stakeholders visualize and understand the intended outcome
- ➢ Details to ignore in prototyping: correctness, completeness, robustness
- ➢ It is not real, it is a prototype
- ➢ If it is misleading, use tracer bullet approach

IV. Tools & Technics

2. Tracer Bullets

- ➢ Small focused implementations of specific features/ functionalities
- ➢ End-to-end implementation in a simple form
- ➢ Allows the user to see something working early (better feel of the progress)
- ➢ Something to demonstrate

InstaDeep™

IV. Tools & Technics

3. Tracer Code

- ➢ It is added temporarily during development or debugging and can be removed once it serves its purpose
- ➢ Insertion of code snippet/log statement to track software execution and behavior
- ➢ Allows the user to see something working early (better feel of the progress)
- ➢ Allows the user to collect runtime information: sequence of method calls, input/output data, or performance metrics

IV. Tools & Technics

4. Post-It Notes

- ➢ Are used for brainstorming, capturing ideas, organizing tasks(kanban boards)
- ➢ Provide a tangible and adaptable medium for collaboration, planning, and tracking progress

TIPS

InstaDeep™

V. Tips

1. Your Knowledge Portfolio

Invest in your portfolio by following these tips:

- ➢ Diversification is key: keep learning and experimenting with different technologies
- ➢ Take classes
- ➢ Learn at least one language every year
- ➢ Read books each quarter
- ➢ Read non Technical books too
- ➢ Update your portfolio periodically

InstaDeep™

V. Tips

2. Dealing with Estimation

➢ What to say when asked for an estimation:
“ I’ll get back to you”, you need think before giving one

➢ The only way to determine the timetable for a project is by gaining experience on that same project.

➢ Scale it properly :

● 15+	days - weeks
● 8+	weeks - months

InstaDeep™

V. Tips

3. Expect Failure

➢ Expect failure in software systems. It highlights that failures are inevitable and that developers should anticipate and plan for them rather than assuming everything will work perfectly

4. Analyze

➢ Critically analyze what u hear and read (ask why 5 times, it will often lead you to the real reason)

5. Project glossary

➢ Use a project glossary for everyone to understand the terms used in the project, because each project has its jargon

V. Tips

Not a software engineer, nor a software developer, you are a PROBLEM SOLVER!

Thank you!

InstaDeep™