Anti Patterns

Assignment Presentation By:

20pw01- Abishek A
20pw02-Akash
20pw03-Akshaya J
20pw04-Anirudh Balakrishnan
20pw05-Britne Binu

Software Development Anti-Patterns

Cut-And-Paste Programming

- **Also Known As**: Clipboard Coding, Software Cloning, Software Propagation
- Code reused by copying source statements leads to significant maintenance problems.
- It comes from the notion that it's easier to modify existing software than program from scratch.
- This is usually true and represents good software instincts. However, the technique can be easily over used.
- Alternative forms of reuse, including black-box reuse, reduce maintenance issues by having common source code, testing, and documentation.

GENERAL FORM

- This AntiPattern is identified by the presence of several similar segments of code interspersed throughout the software project.
- Programmers are learning by modifying code that has been proven to work in similar situations, and potentially customizing it to support new data types or slightly customized behavior.
- This creates code duplication, which may have positive short-term consequences such as boosting line count metrics, which may be used in performance evaluations.
- It can quickly meet short-term modifications to satisfy new requirements.

SYMPTOMS AND CONSEQUENCES

- The same software bug reoccurs throughout software despite many local fixes.
- Lines of code increase without adding to overall productivity.
- It becomes difficult to locate and fix all instances of a particular mistake.
- Code can be reused with a minimum of effort.
- This AntiPattern leads to excessive software maintenance costs.
- Software defects are replicated through the system.
- Cut-and-Paste Programming form of reuse deceptively inflates the number of lines of code developed without the expected reduction in maintenance costs associated with other forms of reuse.

REFACTORED SOLUTIONS

- Cloning frequently occurs in environments where white-box reuse is the predominant form of system extension.
- In white-box reuse, developers extend systems primarily through inheritance.First, subclassing and extending an object requires some knowledge of how the object is implemented.
- In addition, typically, white-box reuse is possible only at application compile time (for compiled languages), as all subclasses must be fully defined before an application is generated.
- On the other hand, black-box reuse has a different set of advantages and limitations and is frequently a better option for object extension in moderate and large systems.

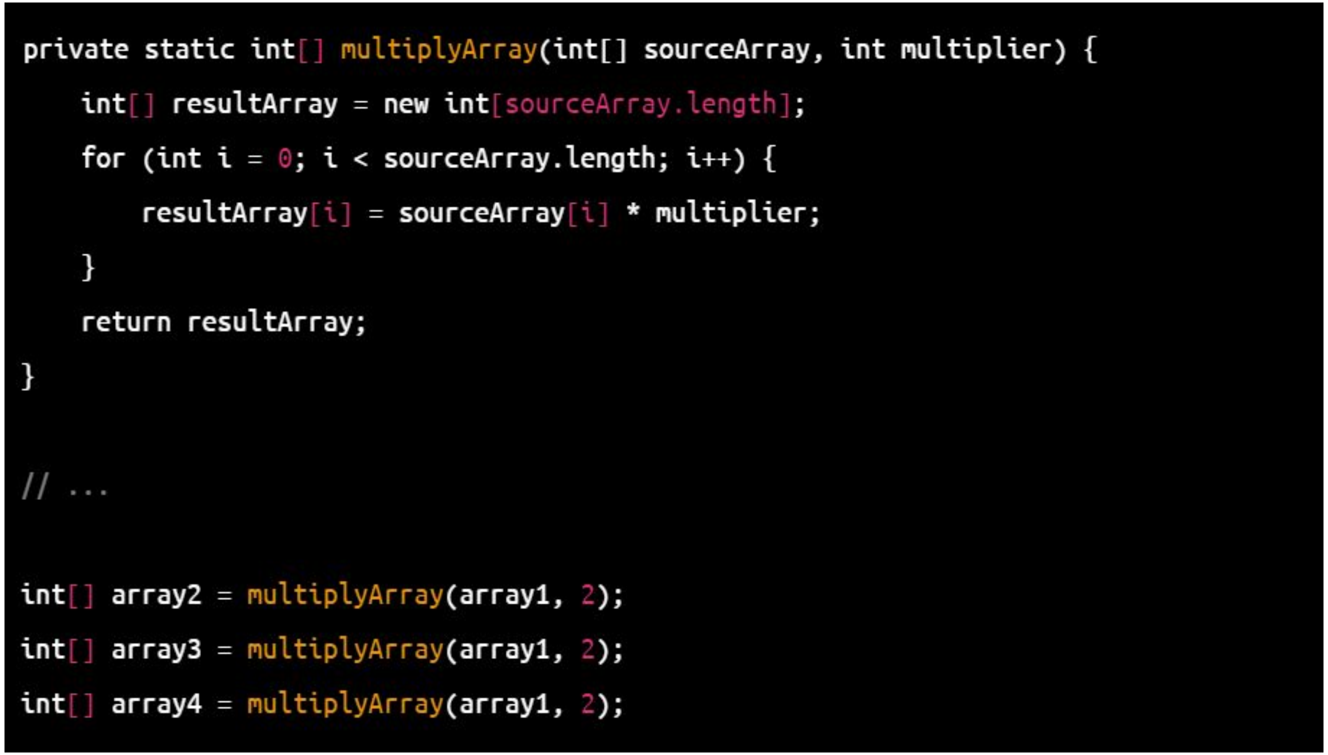
TYPICAL CAUSES

- It takes a great deal of effort to create reusable code, and organization emphasizes short-term payoff more than long-term investment.
- The context or intent behind a software module is not preserved along with the code.
- There is a lack of abstraction among developers, often accompanied by a poor understanding of inheritance, composition, and other development strategies.
- Reusable components, once created, are not sufficiently documented or made readily available to developers.
- There is a lack of forethought or forward thinking among the development teams.

- With black-box reuse, an object is used as-is, through its specified interface.
- The client is not allowed to alter how the object interface is implemented. The implementation of an object can be made independent of the object's interface.
- This enables a developer to take advantage of late binding by mapping an interface to a specific implementation at run time.
- But, changes to the interface typically must be made at compile time, similar to interface or implementation changes in white-box reuse.
- Restructuring software to reduce or eliminate cloning requires modifying code to emphasize black-box reuse of duplicated software segments. The most effective method of recovering your investment is to refactor the code base into reusable libraries or components.

```
for (int i = 0; i < array1.length; i++) {
    array2[i] = array1[i] * 2;
}

// ...

for (int i = 0; i < array1.length; i++) {
    array3[i] = array1[i] * 2;
}

// ...

for (int i = 0; i < array1.length; i++) {
    array4[i] = array1[i] * 2;
}
```

A better approach would be to refactor the code into a separate function that can be called multiple times with different parameters. This not only avoids code duplication but also makes the code easier to read and maintain.



```
private static int[] multiplyArray(int[] sourceArray, int multiplier) {
    int[] resultArray = new int[sourceArray.length];
    for (int i = 0; i < sourceArray.length; i++) {
        resultArray[i] = sourceArray[i] * multiplier;
    }
    return resultArray;
}

// ...

int[] array2 = multiplyArray(array1, 2);
int[] array3 = multiplyArray(array1, 2);
int[] array4 = multiplyArray(array1, 2);
```

The repeated code block is refactored into a separate function called multiplyArray()

Mushroom Management

MUSHROOM MANAGEMENT

Also known as Pseudo-Analysis and Blind Development.

Mushroom Management is often described by

"Keep your developers in the dark and feed them fertilizer."

An experienced system architect recently stated,

"Never let software developers talk to end users."

Without end-user participation, "The risk is that you end up building the wrong system."

In architecture and management circles, there is an explicit policy to isolate system developers from the system's end users.

Requirements are passed second-hand through intermediaries, including architects, managers, or requirements analysts.

Mushroom Management assumes that requirements are well understood by both end users and the software project at project inception. It is assumed that requirements are stable.

Mistaken Assumptions in Mushroom Management

- In reality, requirements change frequently and drive about 30 percent of development cost. In a Mushroom Management project, these changes are not discovered until system delivery. User acceptance is always a significant risk, which becomes critical in Mushroom Management.
- The implications of requirements documents are rarely understood by end users, who can more easily visualize the meaning of requirements when they experience a prototype user interface. The prototype enables end users to articulate their real needs in contrast to the prototype's characteristics.

- When developers don't understand the overall requirements of a product, they rarely understand the required component interaction and necessary interfaces. Because of this, poor design decisions are made and often result in stovepipe components with weak interfaces that do not fulfill the functional requirements.

Mushroom Management affects developers by creating an environment of uncertainty. Often, the documented requirements are not sufficiently detailed, and there is no effective way to obtain clarification.

In order to do their job, developers must make assumptions, which may lead to pseudo-analysis, that is, object-oriented analysis that takes place without end-user participation. Some Mushroom Management projects eliminate analysis altogether and proceed directly from high-level requirements to design and coding.

Refactured Solution

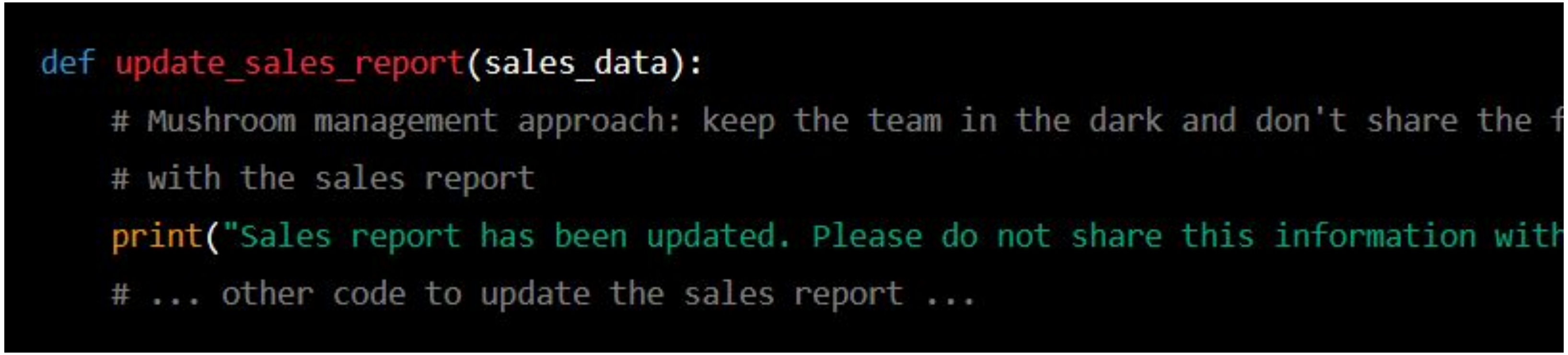
Risk-driven development is a spiral development process based upon prototyping and user feedback.

Risk-driven development is a specialization of iterative-incremental development process.

Every project increment includes extensions to user-interface functionality. The increment includes a user-interface experiment, including hands-on experience. The experiment assesses the acceptability and usability of each extension, and the results of the experiment influence the direction of the project through the selection of the next iteration.

Because the project frequently assesses user acceptance, and uses this input to influence the software, the risk of user rejection is minimized.

The Mushroom Management Antipattern is a term used to describe a situation where employees are kept in the dark and not given access to information, just like mushrooms are kept in the dark and fed with manure.



```
def update_sales_report(sales_data):
    # Mushroom management approach: keep the team in the dark and don't share the f
    # with the sales report
    print("Sales report has been updated. Please do not share this information with
    # ... other code to update the sales report ...
```

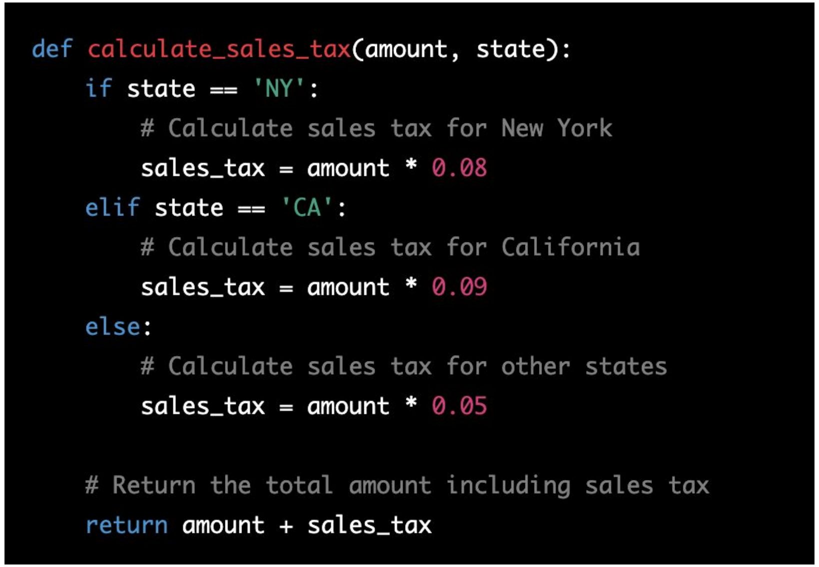
It can lead to communication breakdowns, misunderstandings, and overall decreased effectiveness of the team.

BOAT ANCHOR

Antipattern Problem

- A programming antipattern that occurs when a piece of code or functionality is retained in a program even though it is no longer necessary. This often happens when a feature is added to a program but is never used, or when a feature is used temporarily but is not removed after it is no longer needed.
- The term "Boat Anchor" refers to a useless object that is kept around despite its lack of value, like an anchor on a boat that is no longer being used. In the context of programming, a Boat Anchor can consume system resources, slow down the program, and make the code more difficult to maintain.

Code Snippet



```python
def calculate_sales_tax(amount, state):
    if state == 'NY':
        # Calculate sales tax for New York
        sales_tax = amount * 0.08
    elif state == 'CA':
        # Calculate sales tax for California
        sales_tax = amount * 0.09
    else:
        # Calculate sales tax for other states
        sales_tax = amount * 0.05

    # Return the total amount including sales tax
    return amount + sales_tax
```

Refactored Solution

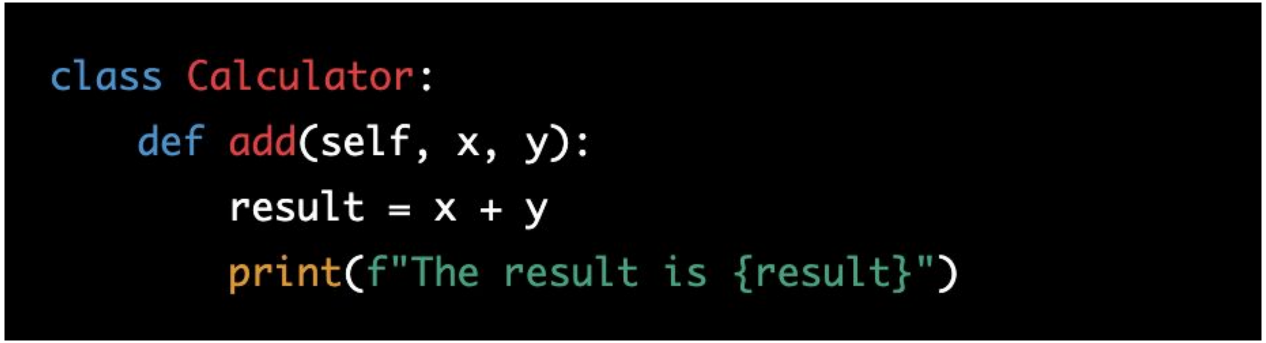
- Good engineering practice includes the provision for technical backup, an alternative approach that can be instituted with minimal software rework. The selection of technical backup is an important risk-mitigation strategy.
- To avoid the Boat Anchor antipattern, it is important to regularly review the codebase and remove any code that is no longer necessary. This will help to keep the code lean and efficient, and make it easier to maintain in the long run.

Poltergeists

Antipattern Problem

- The "Poltergeists Antipattern" is a programming antipattern that occurs when a piece of code or functionality is removed from a program, but its side effects and dependencies remain, causing unpredictable behavior and bugs. This antipattern is named after the idea of a poltergeist - a mischievous and troublesome ghost that causes disturbances even after it has been removed.
- For example, a method might be removed, but other parts of the codebase still rely on that method to function properly. This can cause the code to behave in unexpected ways, leading to bugs and other issues.

Code Snippet



```python
class Calculator:
    def add(self, x, y):
        result = x + y
        print(f"The result is {result}")
```

Refactored Solution

- To avoid the Poltergeists antipattern, it is important to carefully consider the side effects and dependencies of any code that is modified or removed. This includes identifying any potential impacts on other parts of the codebase and ensuring that all necessary changes are made to address those impacts.
- The key is to move the controlling actions initially encapsulated in the Poltergeist into the related classes that they invoked.

Spaghetti Code

Useful Information

- **AntiPattern Name**: Spaghetti Code
- **Most Applicable Scale**: Application
- **Refactored Solution Name**: Software Refactoring, Code Cleanup
- **Refactored Solution Type**: Software
- **Root Causes**: Ignorance, Sloth
- **Unbalanced Forces**: Management of Complexity, Change

Background

The Spaghetti Code Anti-Pattern is the classic and most famous AntiPattern; it has existed in one form or another since the invention of programming languages. Non object-oriented languages appear to be more susceptible to this AntiPattern, but it is fairly common among developers who have yet to fully master the advanced concepts underlying object orientation.

General Form

- Spaghetti Code appears as a program or system that contains very little software structure. Coding and progressive extensions compromise the software structure to such an extent that the structure lacks clarity, even to the original developer, if he or she is away from the software for any length of time.
- If developed using an object-oriented language, the software may include a small number of objects that contain methods with very large implementations that invoke a single, multistage process flow.
- Furthermore, the object methods are invoked in a very predictable manner, and there is a negligible degree of dynamic interaction between the objects in the system. The system is very difficult to maintain and extend, and there is no opportunity to reuse the objects and modules in other similar systems.

```
public void ProcessOrder(Order order) {
    if (order != null) {
        Customer customer = GetCustomerById(order.CustomerId);
        if (customer != null) {
            OrderItem[] items = order.Items;
            if (items != null && items.Length > 0) {
                for (int i = 0; i < items.Length; i++) {
                    OrderItem item = items[i];
                    Product product = GetProductById(item.ProductId);
                    if (product != null) {
                        decimal itemTotal = product.Price * item.Quantity;
                        customer.Balance += itemTotal;
                        UpdateProductInventory(product, -item.Quantity);
                    }
                }
                SaveCustomer(customer);
            }
        }
    }
}
```

Symptoms and Consequences

- Many object methods have no parameters, and utilize class or global variables for processing.
- The pattern of use of objects is very predictable.
- Code is difficult to reuse, and when it is, it is often through cloning. In many cases, however, code is never considered for reuse.
- Benefits of object orientation are lost; inheritance is not used to extend the system; polymorphism is not used.

Ways to avoid Spaghetti Code

- Convert a code segment into a function that can be reused in future maintenance and refactoring efforts. It is vital to resist implementing the Cut-and-Paste AntiPattern. Instead, use the Cut-and-Paste *refactored* solution to repair prior implementations of the Cut-and-Paste AntiPattern.
- Reorder function arguments to achieve greater consistency throughout the code base. Even consistently bad Spaghetti Code is easier to maintain than inconsistent Spaghetti Code.
- Rename classes, functions, or data types to conform to an enterprise or industry standard and/or maintainable entity. Most software tools provide support for global renaming.

Software Architecture Anti-Patterns

COVER YOUR ASSETS

Antipattern problem

- “Document driven software processes often employ authors who list alternatives instead of making decisions.“
- “Cover Your Assets” is an anti-pattern that refers to a situation where a developer or a team focuses primarily on protecting their own interests, code or reputation rather than working towards the larger goal of delivering a quality product.
- This behavior can manifest in several ways, including:
    - Over-documentation
    - Blame-shifting
    - Hoarding knowledge
    - Lack of collaboration

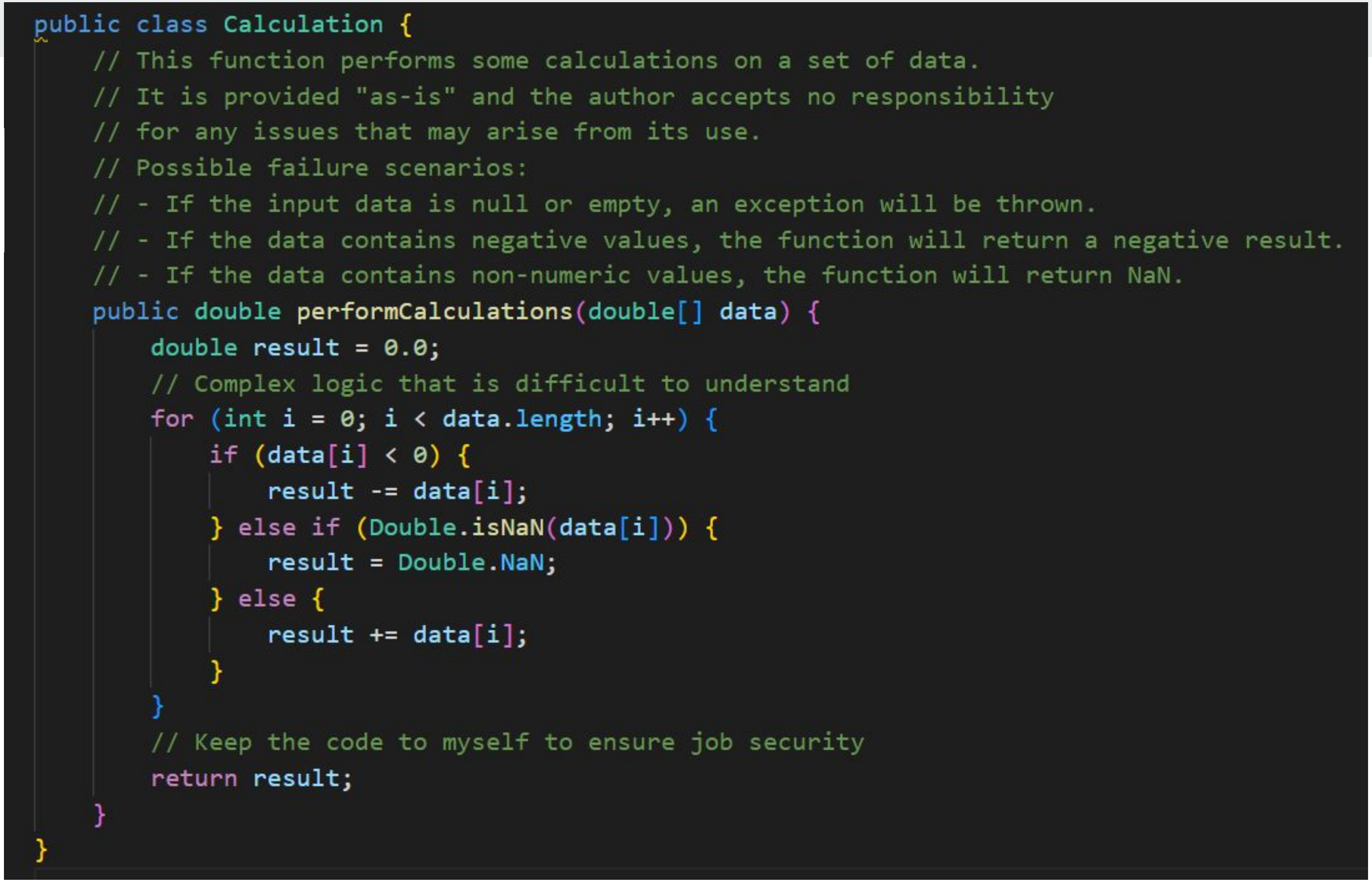
Antipattern problem

- **Over-documentation**: A team may spend an excessive amount of time documenting every aspect of their work to protect themselves in case of errors or issues that may arise later.
- **Blame-shifting**: When something goes wrong, the team members may try to deflect blame onto each other or external factors, rather than taking responsibility and working together to fix the issue.
- **Hoarding knowledge**: In an effort to maintain job security or ensure recognition for their contributions, team members may refuse to share knowledge or expertise with others, leading to knowledge silos and communication breakdowns.
- **Lack of collaboration**: A team may become so focused on their own tasks and responsibilities that they fail to work effectively with other teams or departments, leading to missed opportunities and suboptimal outcomes.

Antipattern problem

- To avoid making mistakes, the authors often describe multiple options, which makes the documents long and confusing.
- The documents don't effectively convey the author's intentions because there is no clear summary or abstract.
- People who need to read the documents, such as those with contractual obligations, may find them difficult to understand because of their length and complexity.
- This is unfortunate because the documents are important for ensuring that the software development process runs smoothly and meets the necessary requirements.

Example



```java
public class Calculation {
    // This function performs some calculations on a set of data.
    // It is provided "as-is" and the author accepts no responsibility
    // for any issues that may arise from its use.
    // Possible failure scenarios:
    // - If the input data is null or empty, an exception will be thrown.
    // - If the data contains negative values, the function will return a negative result.
    // - If the data contains non-numeric values, the function will return NaN.
    public double performCalculations(double[] data) {
        double result = 0.0;
        // Complex logic that is difficult to understand
        for (int i = 0; i < data.length; i++) {
            if (data[i] < 0) {
                result -= data[i];
            } else if (Double.isNaN(data[i])) {
                result = Double.NaN;
            } else {
                result += data[i];
            }
        }
        // Keep the code to myself to ensure job security
        return result;
    }
}
```

Example

- In that example the developer has written a function that performs some calculations on an input array of double values. However, they have also added a lengthy comment block at the top of the function that describes all of the possible failure scenarios.

- Additionally, the function is written in a way that makes it difficult for other team members to understand, using complex logic and variable names that only make sense to the original author.

- Finally, the developer has kept the code to themselves, refusing to share it with other team members and ensuring that they are the only ones who know how it works.

Refactored solution

- Architecture blueprints are abstractions of information systems that facilitate communication of requirements and technical plans between the users and developers.

- An architecture blueprint is a small set of diagrams and tables that communicate the operational, technical, and systems architecture of current and future information systems.

- A typical blueprint comprises no more than a dozen diagrams and tables, and can be presented in an hour or less as a viewgraph presentation.

Jumble Anti-Pattern

Anti-Pattern Problem

- When horizontal and vertical design elements are intermixed, an unstable architecture results. Vertical design elements are dependent upon the individual application and specific software implementations. Horizontal design elements are those that are common across applications and specific implementations.
- By default, the two are mixed together by developers and architects. But doing this limits the reusability and robustness of the architecture and the system software components. Vertical elements cause software dependencies that limit extensibility and reuse. Intermingling makes all the software designs less stable and reusable.

Ways to avoid Jumble in code

- Add vertical elements as extensions for specialized functionality and for performance.
- Incorporate metadata into the architecture.
- Trade off the static elements of the design (horizontal and vertical) with the dynamic elements (metadata).

Proper balance of horizontal, vertical, and metadata elements in an architecture leads to well-structured, extensible, reusable software.

Project Management Anti-Patterns

FEAR OF SUCCESS

Fear of Success is a phenomenon which often occurs when people and projects are on the brink of success.

Some people begin to worry obsessively about the kinds of things that *can* go wrong. Insecurities about professional competence come to the surface.

When discussed openly, these worries and insecurities can occupy the minds of the project team members.

Irrational decisions may be made and inappropriate actions may be taken to address these concerns.

These discussions may generate negative publicity outside the project team that affects how the deliverable is perceived, and may ultimately have a destructive effect on the project outcome.

Fear of Success is related to *termination* issues. Group dynamics progress through a number of phases, for both one-week projects and efforts of lengthier duration. The first phase addresses group acceptance issues.

In the second phase, as relationships are formed, individuals assume various roles in the group, including both formal roles in the organization and informal self-determined roles.

After the roles are established, the work is accomplished (third phase).

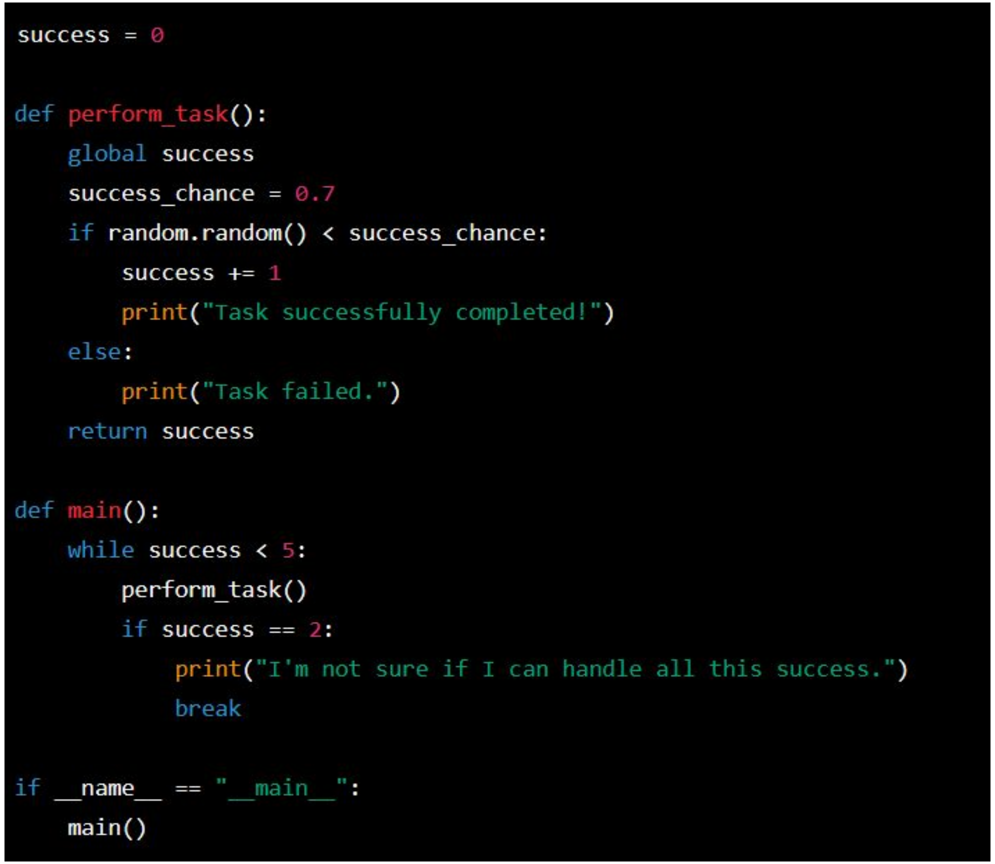
In the fourth phase many personality issues may arise. Because project completion may result in dissolution of the group, these issues often emerge as project termination approaches .

In the termination phase, concerns about the outcome of the project, its future life cycle, and the group's subsequent activities are often expressed in indirect ways.

Refactored Solution

One important action that management can take near the end of a project is to *declare success.* Statements by management that support the successful outcome of the project are needed, even if the actual outcome is ambiguous. It is important to help the project team to accept the significance of their achievements and lessons learned.

Declaring success also helps to mitigate termination issues, maintain the team's commitment to the organization, and pave the way for future project activities.

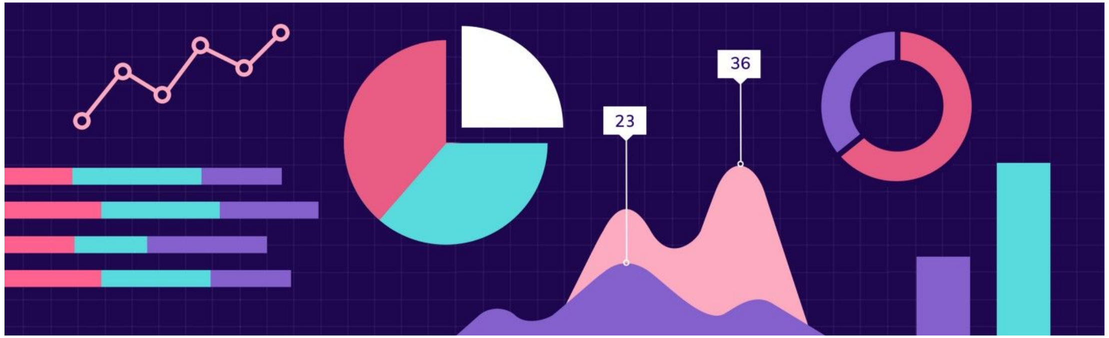


```python
success = 0

def perform_task():
    global success
    success_chance = 0.7
    if random.random() < success_chance:
        success += 1
        print("Task successfully completed!")
    else:
        print("Task failed.")
    return success

def main():
    while success < 5:
        perform_task()
        if success == 2:
            print("I'm not sure if I can handle all this success.")
            break

if __name__ == "__main__":
    main()
```

Viewgraph Engineering

Antipattern problem

The Viewgraph Engineering anti-pattern is a situation where a development team spends an excessive amount of time creating viewgraphs or other forms of visual aids instead of focusing on actual software development.



Antipattern problem

This can happen when management places too much emphasis on documentation and presentation rather than actual product development.

As a result, engineers may end up using tools like office automation software to create pseudo-technical diagrams and papers instead of using proper development tools.

This can be frustrating for the team as it does not allow them to fully utilize their skills and can lead to their skills becoming outdated.

Refactored solution

- Many of the developers caught in viewgraph engineering roles should be redirected to construct prototypes.
- Prototyping, underutilized by many organizations, has many roles in a project beyond the creation of a sales tool. It is a key element of iterative, incremental development processes.
- Prototypes can answer technical questions that cannot be deduced through paper analyses, and can be used to reduce many types of risks, including those related to technology and user acceptance.
- Prototyping helps to shorten learning curves for new technologies.

Refactored solution

- Two principal kinds of prototypes are: mockups and engineering prototypes.
- A **mockup** is a prototype that simulates user-interface appearance and behavior.
- **Engineering** prototypes include some operational functionality, such as application services, databases, and integration with legacy systems.
- Prototyping a system in an alternative language prior to production system development is a useful exercise.

Analysis Paralysis

Useful Information

- **AntiPattern Name**: Analysis Paralysis
- **Also Known As**: Waterfall, Process Mismatch
- **Most Frequent Scale**: System
- **Refactored Solution Name**: Iterative-Incremental Development
- **Refactored Solution Type**: Software
- **Root Causes**: Pride, Narrow-Mindedness
- **Unbalanced Forces**: Management of Complexity

General Form

- Analysis Paralysis occurs when the goal is to achieve perfection and completeness of the analysis phase. This AntiPattern is characterized by turnover, revision of the models, and the generation of detailed models that are less than useful to downstream processes.
- Many developers new to object-oriented methods do too much up-front analysis and design. Sometimes, they use analysis modeling as an exercise to feel comfortable in the problem domain. One of the benefits of object-oriented methods is developing analysis models with the participation of domain experts. Otherwise, it's easy to get bogged down in the analysis process when the goal is to create a comprehensive model.

Assumptions

Analysis Paralysis usually involves waterfall assumptions:

- Detailed analysis can be successfully completed prior to coding.
- Everything about the problem is known a priori.
- The analysis models will not be extended nor revisited during development. Object-oriented development is poorly matched to waterfall analysis, design, and implementation processes.

Symptoms and Consequences

- Design and implementation issues are continually reintroduced in the analysis phase.
- Cost of analysis exceeds expectation without a predictable end point.
- The analysis phase no longer involves user interaction. Much of the analysis performed is speculative.

Solution to overcome Analysis Paralysis

- Key to the success of object-oriented development is incremental development. Whereas a waterfall process assumes a priori knowledge of the problem, incremental development processes assume that details of the problem and its solution will be learned in the course of the development process.
- In incremental development, all phases of the object-oriented process occur with each iteration—analysis, design, coding, test, and validation. Initial analysis comprises a high-level review of the system so that the goals and general functionality of the system can be validated with the users.

THANK
YOU