ISO 9001:2008 Certified Institute

Java Institute for Advanced Technology

Department of Examinations

COURSE(S) – (LEADING TO)	BIRMINGHAM CITY BSC (HONS) SE - TOP-UP
SUBMISSION DATE	
UNIT NAME	OBJECT ORIENTED DESIGN PATTERN II
UNIT ID	JIAT/DP II
EXAMINATION ID	JIAT/DP II/AS/13
BRANCH	JAVA INSTITUTE, COLOMBO

NAME	: M.R.P.N.THARUKSHA RAJAPAKSHA (BLOCK CAPITALS)
BCU STUDENT ID	: 22178965
NIC NO	: 200019401866
SCN NO	: 207977608

Acknowledgments

First and foremost, I would like to express my deep and sincere gratitude to my lecturer and advisor for the object-oriented design patterns II subject, Ms. Narmadha Harischandra for the continuous support, motivation, and knowledge he has given to me to complete this project.

I am incredibly grateful to my parent for their love, care, and sacrifices they made to build a better future for me. And the encouragement they have given me to complete this project.

I would like to thank all of my friends for the support they have given and for the sleepless nights we were working together before deadlines.

I thank the management of Java Institute for their support in doing this project.

Finally, my thanks go to all the people who have helped me to complete this project directly or indirectly.

Table of Contents

1. What is Anti-Pattern

Software design principles are the set of guidelines or rules that assist software developers in making design-level decisions.

A bad design has four components:

- Fragile: Any change to the current application easily breaks the existing system.
- Immobile: An application is created in such a way that it is difficult to reuse.
- Viscose: To avoid complex architectural-level changes, the developer changes the code or environment itself.
- Rigid: An application is designed so that any minor change may result in the movement of too many software components.

If the aforementioned aspects of bad design are applied, they result in solutions that should not be implemented in software architecture or development.

The Anti Pattern could be the result of a manager or developer not knowing any better, not having enough experience or knowledge to solve a specific type of problem, or applying a perfectly good pattern in the wrong context.

An Anti-Pattern is a literary form that describes a common solution to a problem that has decidedly negative outcomes.

Anti-Patterns, like their design pattern counterparts, describe an industry vocabulary for common flaws in organizational processes and implementations. A higher-level vocabulary facilitates communication among software practitioners by allowing for the concise description of higher-level concepts.

Anti-Patterns offer real-world experience in identifying recurring problems in the software industry, as well as detailed solutions to the most common problems. Anti-Patterns highlight the most common issues confronting the software industry and provide tools to help you recognize these issues and determine their root causes.

Anti-Patterns also provides a detailed plan for reversing these root reasons and implementing product solutions.

Anti-Patterns effectively describe the measures that can be taken at various levels to improve application development, software system design, and software project management.

There are 3 major types of Anti-Patterns:

Software Project Management Anti Patterns

More than half of the work in the modern engineering profession involves human communication and problem-solving. Some of the key scenarios in which these issues are destructive to software processes are identified by management Anti Patterns.

Software Architecture Anti Patterns

Although the engineering discipline of software architecture is still in its infancy, software research and experience have repeatedly demonstrated the overarching importance of architecture in software development. Architecture Anti Patterns concentrate on the application and component structure at the system and enterprise levels.

Software Development Anti Patterns

Software refactoring is a type of code modification that is used to improve the structure of software in order to support future extensions and long-term maintenance. One of the primary goals of Anti Patterns development is to describe useful forms of software refactoring. Most of the time, the goal is to transform code without affecting its correctness.

2. Anti-pattern scenarios, bad programming practices & Solutions

2.1 Blob (God Object/ Class)

The Blob or God Object can be seen in architectures where one class dominates operations while other classes essentially encapsulate data. The main issue here is that the majority of the tasks have been assigned to a single class. The majority of the process is contained in the Blob, whereas the data is contained in the other objects. This Anti-Pattern is distinguished by a class diagram that has a single complicated controller class surrounded by basic data classes.

Blob-based architectures isolate processes from data; in other words, they are procedural-style designs rather than object-oriented systems. Despite the fact that it may be described using object notations and implemented in object-oriented languages, the Blob is a procedural design in general. A procedural design separates the process from the data, but an object-oriented design combines the process and data models, as well as partitions.

Consequences and Symptoms

- A single controller class with basic data-object classes attached.
- A class that has a diverse set of unconnected characteristics and functions. The Blob is characterized by an overall lack of coherence in the qualities and processes.
- A single class has several characteristics, actions, or both. The presence of the Blob is generally indicated by a class with 60 or more characteristics and actions.
- There is no object-oriented design. A software main loop associated with relatively passive data items inside the Blob class. Like a procedural main program, the single controller class frequently contains the application's complete functionality.
- Even for basic tasks, the Blob Class may be costly to load into memory, consuming unnecessary resources.

- Typically, the Blob Class is too complicated for reuse and testing. Reusing the Blob for parts of its capability may be wasteful or add unnecessary complexity.
- A legacy design that has been moved but has not been adequately refactored into an object-oriented architecture.
- The Blob undermines the fundamental benefits of object-oriented design. The Blob, for example, restricts the ability to alter the system without interfering with the operation of other contained items. Modifications to the Blob have an impact on the considerable software included within the Blob's encapsulation. Changes to other objects in the system are also likely to affect the Blob's software.

Common Causes

- Specified disaster: Sometimes the Blob is caused by the way requirements are defined. Defining system architecture as part of requirements study is typically improper and frequently results in the Blob Anti Pattern or worse. If the requirements mandate a procedural solution, architectural commitments may be made during requirements analysis that is difficult to modify.
- Too little interference: In iterative projects, developers tend to add small pieces of functionality to existing functioning classes rather than creating new classes or revising the class hierarchy for more effective responsibility allocation.
- Lack of architecture enforcement: Even when a decent design is in place, this Anti Pattern might evolve unintentionally. This might be the result of insufficient architectural evaluation as construction proceeds. This is especially true for development teams that are new to object orientation.
- Lack of object-oriented architecture: The designers may lack thorough knowledge of object-oriented principles. Alternatively, the team's abstraction abilities may be lacking.
- The lack of (any) architecture: The lack of specification of system components, their relationships, and the specific application of the chosen programming languages. Because programming languages are used for reasons other than their intended ones, programs can change ad hoc.

Known Exceptions

When wrapping legacy systems, the Blob Anti Pattern is appropriate. There is no need for software partitioning, all that is needed is a last layer of code to make the old system more accessible.

Solutions

As with most of the Anti-Patterns, the solution involves a form of refactoring. The idea is to deviate behavior from the Blob.

It may be necessary to reallocate functionality to some of the enclosed data items in order to make these objects more capable while making the Blob less complicated.

The procedure for refactoring responsibilities is as follows:

1. Contract-related qualities and operations should be identified and classified. These contracts should be coherent in the sense that they all directly relate to a single point of emphasis, behavior, or function within the entire system.
2. The next phase is to find "natural homes" for these contract-based groupings of functionality and relocate them there.
3. The next stage is to eliminate all "far-coupled," or redundant, indirect connections.
4. Following that, we move associations from derived classes to a single base class if applicable.
5. Finally, eliminate any transitory associations, replacing them with type specifiers for attributes and operations arguments where needed.

2.2 Boat Anchor

A Boat Anchor is a piece of software or hardware that is no longer relevant to the present project. The Boat Anchor is frequently an expensive buy, which adds to the irony of the purchase.

Solutions

Good engineering practice involves the provision of technological backup, which may be implemented with little software change.

A critical risk-mitigation approach is the selection of technological backup. Most infrastructure technologies (on which most software depends) and other high-risk technologies should have technical backups recognized. In the selection process, technical backups should be examined with critical-path technology. For both critical-path and backup solutions, prototyping with evaluation licenses (available from most suppliers) is advised.

2.3 Cut-and-Paste Programming

Cut-and-Paste Programming is a popular yet degenerate type of software reuse that leads to maintenance headaches. It stems from the idea that it is easier to change existing software than to create new software from the start. This is typically correct and shows sound software intuition. Furthermore, because the developer has complete control over the code used in his or her application, it is simple to make short-term changes to fulfill new requirements. The approach, however, can be overused.

The existence of multiple comparable portions of code spread across the software project identifies this Anti-Pattern. Typically, the project includes a large number of programmers who are learning how to design software by watching more experienced developers. They are learning, though, by altering code that has been shown to function in comparable scenarios and maybe adapting it to enable new data types or somewhat altered behavior. This results in code duplication, which may have short-term benefits such as increasing line count metrics, which may be utilized in performance assessments.

Consequences and Symptoms

- This Anti Pattern results in high software maintenance costs.
- Cut-and-Paste The programming form of reuse artificially inflates the number of lines of code created while failing to reduce maintenance costs as other types of reuse do.
- Reused assets are not translated into a format that is easily reusable and documented.
- Code is thought to be self-documenting.
- Lines of code grow without increasing overall productivity.
- It becomes tough to track down and correct all occurrences of a single error.
- Code evaluations and inspections are unnecessarily prolonged.
- Despite several local patches, the same software problem recurs throughout the product.
- Software flaws are propagated across the system.
- Developers generate several unique remedies for issues with no way of combining the variants into a common repair.
- Code may be reused with little effort.

Common Causes

- People are more prone to use Cut-and-Paste Anti-Pattern when they are inexperienced with new technology or techniques; as a consequence, they take a functional example and change it to meet their unique needs.
- The organization does not promote or reward reusable components, and development speed takes precedence over all other assessment criteria.
- It takes a lot of work to produce reusable code, and organizations prioritize short-term payback over long-term commitment.
- To be reused, the code must be an excellent match for the new assignment, according to the organization. Code is replicated to solve apparent shortcomings in meeting what is assumed to be a one-of-a-kind issue set.
- The context or intent of a software module is not saved with the code.

- Once constructed, reusable components are not fully documented or readily available to developers.
- The development teams are lacking in thoughtfulness and future thinking.
- Developers have a lack of abstraction, which is frequently coupled with a lack of understanding of inheritance, composition, and other development methodologies.

Known Exceptions

The cost of greater maintenance is paid and when the only goal is to get the code out the door as rapidly as possible, the Cut-and-Paste Programming Anti-Pattern is allowed.

Solutions

The distinction between white-box and black-box reuse is analogous to the distinction between object-oriented programming (OOP) and component-oriented programming (COP), where white-box sub classing is the traditional OOP signature and dynamic late binding of the interface to implementation is a COP staple.

Cloning is common in contexts where white-box reuse is the primary method of system expansion. To begin, sub classing and extending an object necessitates some understanding of how the object is implemented, such as the intended restrictions and usage patterns indicated by the inherited base classes. Inheritance is a fundamental aspect of object-oriented development, although it has significant limitations in big systems. White-box reuse is often only achievable at application build time (for compiled languages), as all subclasses must be completely specified before an application is built. Most object-oriented languages impose little constraints; kinds of extensions can be implemented in a derived class, resulting in inefficient usage of sub classing.

An object is utilized as-is through its given interface with black-box reuse. The client is not permitted to change the way the object interface is implemented. This allows a developer to use late binding by mapping an interface to a particular implementation at run time. Clients can be built to a static object interface while benefiting from more complex services that support the same object interface over time. Black-box reuse

provides a separate set of benefits and drawbacks and is typically a preferable alternative for object expansion in moderate and large systems. The main advantage of black-box reuse is that, with the use of tools such as interface definition languages, an object's implementation may be made independent of the object's interface. To decrease or eliminate cloning in software, code must be modified to emphasize the black-box reuse of duplicated program elements.

If you've utilized Cut-and-Paste Programming extensively over the life of a software project, the most efficient way to recoup your investment is to rework the code base into reusable libraries or components that focus on black-box reuse of functionality. When performed as a single project, this refactoring process is expensive, time-consuming, and typically difficult necessitating the involvement of a strong system architect to oversee and execute the process, as well as mediate discussions about the benefits and limitations of various extended versions of software modules.

Code mining, refactoring, and configuration management are the three phases of effective refactoring to reduce numerous versions. The systematic detection of several versions of the same software component is known as code mining. Refactoring entails creating a standard version of the code segment and reintroducing it into the code base. Configuration management is a collection of policies designed to help avoid future instances of Cut-and-Paste Programming. This, for the most part, necessitates monitoring and detection procedures (code inspections, reviews, and validation), as well as teaching activities. Management support is required at all three levels to secure financing and support.

2.4 Dead End

Modifying a reusable component results in a Dead End if the updated component is no longer maintained and supported by the provider. When these changes are made, the duty of support is transferred to the application system developers and maintainers.

Improvements to the reusable component are difficult to incorporate, and support issues may be blamed on the update. A system integrator's decision to change a reusable component is frequently viewed as a workaround for the vendor's product deficiencies. As a short-term measure, this aids rather than hinders product development progress.

If the provider is a commercial vendor, this Anti-Pattern is often referred to as Commercial off-the-shelf (COTS) Customization. In fact, upgrading the customized component may be impossible due to factors such as cost and personnel turnover. If later product versions are made accessible, the required adjustments will have to be done again, if possible.

When dealing with future program versions and "reusable component" vendor releases, the longer-term maintenance load becomes unmanageable. The only time we saw this work was when the system integrator worked with the reusable component vendor to get the SI improvements included in the vendor's next release. It was pure coincidence that their goals were so similar.

Solutions

In testbeds that allow fundamental research, such as throwaway code, a Dead End may be an acceptable approach, and major advantages are achieved through customization.

Use an isolation layer when customization is unavoidable. Use isolation layers and other strategies to keep the majority of the application software's dependencies distinct from modifications and proprietary interfaces.

Avoid COTS customization and reusable software changes. Use mainstream platforms and COTS infrastructure to reduce the danger of a Dead End, and upgrade according to the supplier's release schedule.

2.5 Functional Decomposition

This Anti-Pattern is the consequence of experienced non-object-oriented developers designing and implementing an object-oriented application.

In class structure, the generated code resembles a structured language such as Pascal or FORTRAN. When developers are comfortable with a "main" procedure that calls a large number of subroutines, they may declare every subroutine a class, completely ignoring class hierarchy (and pretty much-ignoring object orientation entirely). It may be quite complex, as talented procedural engineers discover ingenious ways to reproduce their tried-and-true methods in an object-oriented design.

Consequences and Symptoms

- There is no apparent method to describe (or even explain) how the system works.
- Class models are completely illogical.
- Frustration and despondency among testers.
- There is no possibility of achieving software reuse in the future.
- All class attributes are private and only utilized within the class.
- Classes that perform a single operation, such as a function.
- This Anti-Pattern may be shown by classes having "function" names such as Calculate_Interest or Display_Table.
- A very defective design that entirely disregards the purpose of object-oriented architecture.
- There is no use of object-oriented notions such as inheritance or polymorphism.
- This may be exceedingly costly to maintain (if it ever worked in the first place; never underestimate the creativity of an elderly coder who is steadily losing the race to technology).

Common Causes

- Specified disaster: Those who produce specifications and requirements may not have real-world experience with object-oriented systems. If the system they

define makes architectural commitments prior to requirements analysis, Anti Patterns such as Functional Decomposition can and frequently do result.

- Lack of architecture enforcers: It doesn't matter how beautifully the architecture is built if the implementers don't comprehend what they're doing. Without the proper supervision, they will generally find a method to fudge anything using the tactics they do know.
- Lack of object-oriented understanding: The implementers did not "get it." This is extremely typical when developers go from a non-object-oriented programming language to an object-oriented programming language. Object orientation can take up to three years for a corporation to completely integrate due to paradigm changes in architecture, design, and implementation.

Known Exceptions

When an object-oriented solution is not required, the Functional Decomposition Anti Pattern is sufficient. This exception may be enhanced to cope with solutions that are purely functional in nature but are wrapped to offer an object-oriented interface to the implementation code.

Solutions

Create a design model that integrates the critical components of the present system. Focus on constructing a foundation for understanding as much of the system as feasible rather than on refining the model. The design model should ideally justify, or at least rationalize, the majority of the software components. It is realistic to anticipate that certain components of the system exist for reasons that are no longer understood and cannot be guessed at. Creating a design model for an existing code base is informative because it reveals how the whole system fits together.

If the essential needs for the program can still be determined, build an analytical model for the software to describe the key elements of the software from the user's perspective. This is critical for determining the underlying purpose for many of the software components in a given code base that has become forgotten over time.

Provide full documentation of the methods utilized as the foundation for future maintenance efforts for all phases in the Functional Decomposition Anti Pattern solution.

Use the following principles for classes that fall outside of the design model:

- Consider rebuilding the class as a function if it does not include any kind of state information. Some aspects of the system may be better depicted as functions that may be accessed from anywhere in the system without limitation.
- If the class just has one method, attempt to properly represent it as part of another class. Classes meant to aid another class are frequently better off being integrated into the base class they support.
- Make an attempt to merge numerous classes into a new class that meets a design goal. The objective is to combine the functionality of many types into a single class that represents a wider domain notion than the preceding finer-grained classes. For example, instead of having classes to manage device access, filter information to and from the devices, and operate the device, merge them into a single device controller object with methods that accomplish the previously distributed operations.

Examine the design and look for subsystems that are comparable. Engage in code restructuring as part of program maintenance to reuse code between comparable subsystems. These are candidates for reuse.

2.6 Lava Flow

Lava Flow is a programming anti-pattern that happens when code that works but is poorly documented or understood by its maintainers is retained in a system because it works.

In other words, lava flow is any code that is kept around because nobody has the time or desire to understand it, let alone approach it. Such code is said to "flow" through the system it lives in.

This Anti Pattern, on the other hand, is extremely common in innovative design shops where proof-of-concept or prototype code is rapidly moved into production.

It's a bad design for several reasons:

- As with many Anti Patterns, you lose many of the inherent benefits of an object-oriented design. In this case, you lose the ability to use modularization and reuse without further spreading the Lava Flow globules.
- Lava flows are highly-priced to analyze, verify, and test. In practice, verification and testing are rarely possible. All of this effort is completely futile and wasteful.
- Lava Flow code can be costly to load into memory, wasting valuable resources and negatively impacting performance.

Consequences and Symptoms

- There are numerous code areas that are "in flux" or "to be replaced."
- Existing Lava Flow code that is not removed may continue to spread as code is reused in other areas.
- There are entire blocks of commented-out code without an explanation or documentation.
- Unused, mysterious, or obsolete interfaces are found in header files.
- Unused (dead) code that has been left in.
- Unjustifiable variables and code fragments are frequently found in the system.
- The system architecture is very loose and "evolving."

- Complex, important-looking undocumented functions, classes, or segments that do not appear to be related to the system architecture.
- As the flows harden and compound, it becomes increasingly difficult to document the code or acknowledge its architecture well enough to make modifications.
- If the process that leads to Lava Flow is not checked, there may be exponential growth as succeeding developers, who are too rushed or intimidated to analyze the original flows, continue to produce new, secondary flows in order to work around the original ones, compounding the problem.

Common Causes

- Lack of architecture, or development that is not architecture-driven. This is especially true for highly mobile development teams.
- Uncontrolled distribution of incomplete code Trial approaches to implementing some functionality is being implemented.
- Inadequate configuration management or adherence to process management policies.
- R&D code is deployed into production without regard for configuration management.
- The code was written by a single developer (lone wolf).
- The development process is iterative. Frequently, the software project's goals are unclear or change frequently. To adapt to the changes, the project must rework, backtrack, and create prototypes.
- In response to demonstration deadlines, there is a tendency to make hurried changes to code to address immediate issues. The code is never cleaned up, so architectural consideration and documentation are perpetually postponed.

Known Exceptions

Small-scale, disposable prototypes in a research and development environment are ideal for implementing the Lava Flow Anti-Pattern. It is critical to delivering quickly, and the outcome does not have to be long-term.

Solutions

There is only one definite method to avoid the Lava Flow Anti Pattern. Make sure that good architecture comes before production code development. This architecture must be supported by a configuration management process that maintains architectural compliance while allowing for "mission creep" (changing requirements). If an architectural concern is overlooked early on, code that is not part of the goal design may be produced, rendering it redundant or dead. Dead code becomes challenging for analysis, testing, and change over time since it does not receive the same amount of scrutiny as its intended successor. Management must postpone development until a clear architecture has been defined and disseminated to developers.

Bugs are introduced as assumed dead code is removed. When this happens, avoid the desire to treat the symptoms without first determining the root cause of the problem. Examine the interdependencies. This will assist you in better defining the desired architecture.

When compared to the cost of jackhammering away hardened globules of Lava Flow code, an investment in quality software interfaces can pay off handsomely in the long term. In cases where Lava Flow already exists, the cure can be hard. To avoid Lava Flow, it is critical to building robust, well-defined, and well-documented system-level software interfaces.

System discovery is required to locate the components that are really used and necessary to the system. Configuration management is aided by tools such as the Source-Code Control System (SCCS). SCCS comes standard with most UNIX systems and provides a rudimentary capability for tracking the history of changes to configuration-controlled files.

2.7 Poltergeists

Poltergeists are classes with relatively restricted responsibilities and life cycles. That is classes with limited obligations and functions to play in the system, hence their effective life cycle is fairly short. They frequently launch processes for other items. The refactored approach involves a reallocation of responsibilities to longer-lived objects, which eliminates the Poltergeists.

The Poltergeist Anti-Pattern is frequently deliberate on the side of some inexperienced architect who doesn't fully grasp the object-oriented notion. Poltergeists clog software architectures, adding unneeded abstractions; they are too complicated, difficult to grasp, and difficult to manage.

Poltergeist classes are undesirable design artifacts for three reasons:

- They are useless, so every time they "appear," they squander resources.
- They obstruct proper object-oriented design by unnecessarily cluttering the object model.
- They are inefficient because they make use of several redundant navigation pathways.

Consequences and Symptoms

- Objects and classes that exist only for a limited time.
- Classes without a state
- Classes that exist only to "seed" or "invoke" other classes via temporary relationships.
- Classes having operation names that seem "control-like," such as start_process_alpha.
- Temporary associations
- Navigation pathways that are redundant.

Common Causes

- During the requirements analysis process, management may make architectural promises. This is incorrect and frequently leads to issues such as this Anti-Pattern.
- Designers' lack of object-oriented architectural understanding.

Solutions

Poltergeists are eliminated from the class structure by Ghostbusters. However, when they are removed, the functionality "given" by the poltergeist must be restored. This is trivial to remedy with a simple tweak to the architecture. The key is to relocate the controlling activities that were formerly included in the Poltergeist into the relevant classes that they triggered.

2.8 Spaghetti Code

Spaghetti Code appears as a program or system with minimal software structure.

If the software is written in an object-oriented language, it may contain a small number of objects containing methods with very large implementations that invoke a single, multistage process flow. Furthermore, the system is hard to maintain and extend, and the objects and modules cannot be reused in other similar systems.

Coding and progressive extensions degrade the software structure to the point where it is unclear, even to the original developer, if he or she is away from the software for an extended period of time. The object methodologies are invoked in a very predictable manner, and the dynamic interaction between the objects in the system is negligible.

Consequences and Symptoms

- There are only a few relationships between objects.
- Object usage patterns are very predictable.
- Follow-up maintenance efforts exacerbate the problem.
- Methods are very process-oriented; in fact, objects are frequently named processes.

- Many object methods have no parameters and process data using class or global variables.
- Object implementation, not object clients, determines the flow of execution.
- Code is difficult to reuse, and when it is, it is frequently accomplished through cloning. However, in many cases, code is never considered for reuse.
- Software quickly reaches a point of diminishing returns; the effort required to maintain an existing code base exceeds the cost of developing a new solution from scratch.
- Object orientation benefits are lost; inheritance is not used to extend the system, and polymorphism is not used.
- Only parts of objects and methods appear to be suitable for reuse after code mining. Spaghetti Mining Code can often provide a poor return on investment; this should be considered before deciding to mine.

Common Causes

- This is frequently the result of developers working alone.
- There is no mentoring in place, and code reviews are ineffective.
- Inadequate knowledge of object-oriented design technologies.
- There will be no design prior to implementation.

Known Exceptions

When the interfaces are coherent and only the implementation is spaghetti, the Spaghetti Code Anti-Pattern is acceptable. This is similar to wrapping non-object-oriented code.

If the domain is unfamiliar to the software architects and developers, it may be better to build products to gain an understanding of the domain with the intention of later developing products with an improved architecture.

The reality of the software industry is that software concerns are usually subservient to business concerns, and business success is sometimes contingent on delivering a software product as soon as possible.

If the component's lifetime is short and it is cleanly isolated from the rest of the system, then some bad code may be acceptable.

Solutions

To avoid the accumulation of spaghetti code in their code base, every programmer and organization must take precautions. There are a few basic rules and methods to follow in order to maintain efficiency and avoid spaghetti code.

- Getting to Know the Code: When starting a new job at a company, it is common practice to learn their methods and styles before undertaking any significant programming-related work. This will help you understand how things work there and gain a better understanding of the structure of their code base.
- Use Light Frameworks: There are a plethora of frameworks and libraries available in all modern programming languages that allow to performance of hundreds of functions with only a few lines of code. As a result, the code becomes leaner, making it easier to find and fix bugs.
- Leave a comment: Coders regard writing comments as a very good practice. Comments benefit both the programmer writing the code and the person reading it. It clarifies what a particular section of code does and saves time.
- Perform Unit Tests: Can reduce the likelihood of spaghetti code occurring, by performing routine unit tests.
- Always double-check: It is better to go over a section of code one more time, especially if it will take you hours to find your bug in thousands of lines of spaghetti.

3. Anti-patterns and coding smells

Code smells are caused by bad or incorrect programming. These errors in the application code are frequently the result of mistakes made by the application programmer throughout the development process.

Code smells are typically caused by a failure to develop code in line with appropriate standards. Code smells may be caused by a variety of factors, including faulty module dependencies, erroneous method assignment to classes, and unnecessary repetition of code segments. Code that stinks might eventually create severe performance issues and make business-critical systems difficult to maintain. In other circumstances, it indicates that the documentation needed to properly outline the project's development standards and expectations was either insufficient, erroneous, or nonexistent.

Some instances of code smells are the following:

- Comments
- Data Clumps
- Dead Code
- Duplicate Code
- Long Methods
- Long Parameter List
- Unnecessary Primitive Variables

A code smell is not always a problem. Code odors are merely indicators of possible violations of code discipline and design standards. The code may still compile and function as planned. The root of a code smell may generate cascading errors and failures over time. It is also a good sign that code restructuring is required.

Differences between Anti Pattern and Coding Smells

- Code smells are more noticeable. For example, duplicate codes. The code makes excessive use of Switch situations. Anti-patterns, on the other hand, may or may not be easy to discern. It is dependent on the anti-pattern issue. Cargo cult programming, for example.

- Code smells are only seen in software issues. Anti-patterns extend beyond the stench of code. Anti-patterns, for example, are used to describe not only the most typical answer to software problems, but also organizational problems, configurational problems, and process difficulties. Anti-patterns can be expanded or modified to encompass a wider range of issues. For example, analysis paralysis overthinking, or overanalyzing a problem might inhibit decision-making.
- Natural language is used to explain code smells. As a result, code smells can be misleading and are dependent on how analyzers interpret them. After a thorough evaluation, anti-patterns are published in the structured semi-formal Pattern Specification Language. They are thus more formal than code smells. Anti-patterns are not published in mathematical formal languages to keep them accessible to the general public.
- It is possible for code smell and anti-pattern to overlap. For example, the blob/God class anti-pattern indicates that if the class has a lot of responsibilities, code maintenance will be tough. It is similar to the God Object code smell. However, the goal of the anti-pattern is to explicitly explain the pattern that leads to undesirable effects. The Code smell is a "feeling" that there is an issue with the code.
- The context is the most fundamental distinction between code smell and anti-pattern. Pattern universe includes anti-patterns. The phrase anti-pattern in the sense of characterizing patterns that have a negative outcome a code smell is used to describe a tip rather than a pattern.

4. References:

Wikipedia. (2021). *Anti-pattern*. [online] Available at: https://en.wikipedia.org/wiki/Anti-pattern.

freeCodeCamp.org. (2020). *Anti-patterns You Should Avoid in Your Code*. [online] Available at: https://www.freecodecamp.org/news/antipatterns-to-avoid-in-code/.

GeeksforGeeks. (2021). *6 Types of Anti Patterns to Avoid in Software Development*. [online] Available at: https://www.geeksforgeeks.org/6-types-of-anti-patterns-to-avoid-in-software-development/.

Stack Overflow. (n.d.). *terminology - What is an anti-pattern?* [online] Available at: https://stackoverflow.com/questions/980601/what-is-an-anti-pattern [Accessed 14 Oct. 2022].

Robinson, C. (n.d.). *Anti-Patterns vs Patterns: What is an Anti-Pattern?* [online] BMC Blogs. Available at: https://www.bmc.com/blogs/anti-patterns-vs-patterns/.

Besbes, A. (2021). *18 Common Python Anti-Patterns I Wish I Had Known Before*. [online] Medium. Available at: https://towardsdatascience.com/18-common-python-anti-patterns-i-wish-i-had-known-before-44d983805f0f [Accessed 14 Oct. 2022].

Ratnaparkhi, A. (2021). *Differences between Code Smells and Anti-patterns*. [online] Medium. Available at: https://alokratnaparkhi8.medium.com/differences-between-code-smells-and-anti-patterns-d19351539f7f#:~:text=The%20most%20basic%20difference%20between [Accessed 14 Oct. 2022].

Software Engineering Stack Exchange. (n.d.). *What is the difference between Code Smells and Anti Patterns?* [online] Available at: https://softwareengineering.stackexchange.com/questions/350085/what-is-the-difference-between-code-smells-and-anti-patterns.

Englund, E. (2019). *Anti-Patterns and Code Smells*. [online] Medium. Available at: https://itnext.io/anti-patterns-and-code-smells-46ba1bbdef6d [Accessed 14 Oct. 2022].