UNIT-3

Towards a Discipline of Exploratory Programming

Reverse Engineering, Reusable Software Design knowledge, Stepwise Abstraction, The problem of decompiling, Controlled Modifications, Structured growth.

Exploratory programming is an approach to s/w dev that emphasizes experimentation, iteration, and learning through hands-on coding.

Developing a discipline of exploratory programming involves incorporating certain principles and practices into your workflow.

Iterative Development :-

Embrace an iterative dev process where you build, test, and refine your code incrementally. Start with a minimal value product (MVP) and iterative based on feedback and insights gained during the exploration process.

Rapid Prototyping :

Use prototyping was a way to quickly experiment with ideas and gather feedback.

Iterative Dev Environment :

Choose programming languages and tools that support interactive dev, allowing you to test and modify code in real-time.

Utilize REPL (Read-Eval-Print Loop) environments for quick experimentation.

Continuous Feedback:

- Seek feedback early and often from stakeholders, users, and team members.

Exploratory Testing:

Integrate testing into your exploratory programming process to catch issues early.

Documentation and Reflection:

Document your exploratory process, including the rationale behind design decisions and any lessons learned.

Flexible Architecture:

Design your codebase with flexibility in mind, allowing for easy modification and adaptation as requirements evolve.

Version Control:

Use version control systems to track changes and roll back to previous states if needed.

Reverse Engineering :-

Reverse engineering is a process of moving backwards from an engineered product to extract the representations that would have been generated en route to the product had it been developed in accordance with a proper SDLC.

The classical AI programming languages primarily LISP, but Prolog too can now be so classified, offer the programmer a flexibility and freedom that facilitate the rapid evolution of s/w.

Type flow analysis is used to generate from the LISP code a representation of the type level structure implicit in the program.

This technique, which has been emplemented in a system called ESSIE.

The fundamental representation of type flow analysis is the braid which represent the set of execution strands through a function at the type level.

The FC - feasibility conditions represent the type-related constraints that must be satisfied in order for the execution path represented by the strand to be valid.

The CE - Constraint Environment holds the type-level properties of variables that must be satisfied whenever the execution path is traversed

The RT - Result Type indicates what type would be returned from the execution of the code represented by the strand.

A type-level is signalled when the composition of two execution paths is found to be feasible but the two CEs can not be successfully composed.

ESSIEs analysis of a Lisp system is expected to provide 3 benefits:

→ The verification of type safety.
→ The detection of type level errors.

→ The signalling of anomalies which may or may not be errors.

Using this reverse engg technique is an attempt to reconcile the implementation freedom that supports exploratory programming with implementation constraints, such as type-level consistency, that support the dev of robust s/w.

The classical scientist is primarily in reverse engineering.

Reverse engineering is a process in which a product, system or s/w is analyzed to understand its design, functionality, and components.

This can be done for various reasons, including understanding how a product works, creating interoperable systems, finding vulnerabilities, or improving upon existing designs.

Process of Reverse Engg:

Understanding the Objective:

This could include understanding the functionality of a s/w app^n, uncovering potential security vulnerabilities or creating compatible systems.

Decompilation and Disassembly:

For s/w reverse engg, decompilation and disassembly are common techniques.

Decompilation involves converting machine code back into a high-level programming language, while disassembly involves breaking down machine code into assembly language.

Code Review:
Examine the decompiled code (or disassembled assembly) to understand the logic algms, and data structures used in the s/w.

Dynamic Analysis:
Execute the s/w (or system) in a controlled environment and monitor its behavior.
This involves observing runtime interactions, system calls, and network communications.

Applications of RE:
→ Interpretability.
→ Security analysis
→ Legacy system support.
→ Product Improvement.
→ Malware Analysis.

Reverse engineering is a powerful tool but should be approached with legal and ethical considerations.

Reusable Software:
The term reusable s/w self-explanatory.
S/w reuse is the appn of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of dev and maintenance of that other system.
This reused knowledge includes artefacts such as domain knowledge, dev experience, design decisions, architectural structures, requirements, designs, code, documentation.

S/w reusability refers to the desire to minimize reinventions of the wheel & to avoid continual re-writing of much the same code, to be more precise.

S/w reusability has been very much more of a desire than a reality in the s/w world.

It is true that there are, and have been for a long time, substantial libraries of statistical and mathematical functions which are reused continually with a high success rate.

So finding code rather than writing it is the general advocated.

But ensuring that the code is not fresh, and deciding how best to chunk-up s/w into reusable modules, are both open problems.

The gathering into one routine of all the connecting links b/w a program and a some function, it will need at many points, so as to form a single standard interface b/w the two.

A typical subject of modularization is an I/O package, which accepts all I/O req from a program in a machine-independent & OS independent form, and generates from them calls on the h/w & OS functions that actually provide the needed service.

Important factors in the success of this particular reusability,

→ making the library of reusable modules easy to use
→ Establishing the view that all programmers are either contributors to or consumers of library code.

Code reuse involves three steps:
(i) Accessing the code
(ii) Understanding it and
(iii) Adapting it.

Attribute	Metric
Program size	Lines of code
Program structure	Number of modules, number of links, and cyclomatic complexity.
Program documentation	Subjective overall rating.
Programming language	Relative language closeness.

Reusable s/w design involves creating modular, flexible, and well-organized code that can be easily reused in diff parts of a s/w system or even in diff. projects.

Modularity:
Breaking down a s/w system into independent and interchangeable modules.

Encapsulation:
Hiding the internal details of an object and exposing only what is necessary.

Abstraction:
Simplifying complex systems by modeling classes based on essential properties

Interfaces:

Defining clear and standardized interfaces for communication b/w diff s/w components.

Code Organization and Packaging:

structuring code in a way that facilitate easy navigation, maintenance, and reuse.

Community Collaboration:

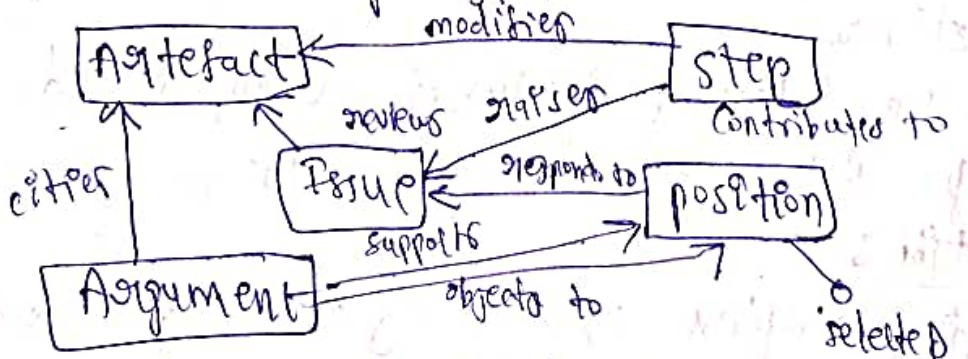
Engaging with open-source communities and sharing reusable components.

Design knowledge:-

Design knowledge is the process of storing the reusability modules in a machine-manipulable form.

To provide intelligent tool support for the early phases of system design, including s/w requirements analysis and domain modeling, it is necessary to develop a theory of the early design process that allows one to reason about design steps, sequences of steps, goals, open issues and so forth.

There are 5 legal binary relationships b/w the five entity types.



The generic model of potts.

→ Modifies - steps modify artefacts
→ Raises - steps raise issues
→ Reviews - issues review artefacts.
→ Responds to - a position is a response to an issue.
→ Supports - arguments support positions
→ Object to - arguments can also object to positions.
→ Cities - arguments can cite artefacts.
→ Contributes to : positions contribute to steps.

In general terms, the customization is accomplished by specialization of entity classes.

Stepwise Abstraction :-

Stepwise refinement & stepwise decomposition terms are used to label the general process of developing an intricate algm from a specification

(i) The effective communication of an AI based upon a sequence of abstraction steps.
(ii) The degree of conciseness and modularity of the abstract representation obtained.
(iii) During the process of stepwise abstraction, a notation which is natural to the program at hand should be used as long as possible.
(iv) Each abstraction implies a number of design decisions based upon a set of design criteria.
(v) The design of an effectively communicable sequence of representations from an AI program is not a trivial process.

Stepwise abstraction is a process in s/w design and dev where a complex system is gradually broken down into simpler and more manageable parts.

This iterative process involves creating layers of abstraction, each hiding the details of the underlying implementation while providing a well-defined interface.

Stepwise abstraction is a key principle in creating modular and reusable s/w.

Identify the problem:

Clearly define the problem or functionality you want to add in your s/w.

High-level Overview:

Provide a high-level overview of the entire system, outlining major components and their interactions.

Identify Components:

Break down the system into distinct components or modules based on functionality.

Define Interfaces:

For each component, define a clear and well-documented interface.

Implement Basic Functionality:

Implement the basic functionality of each component, focusing on achieving the primary goals without unnecessary complexity.

Abstraction Layer 1:
Identify common patterns or functionalities shared among components and create a first abstraction layer.

Refine Interfaces:
Refine the interfaces of the components based on feedback and the evolving understanding of the problem.

Abstraction Layer-2:
Identify further commonalities or patterns emerging from the refined components. Create a second layer of abstraction to encapsulate shared functionalities, promoting reusability.

Benefits of Stepwise Abstraction:

→ Modularity.
→ Reusability
→ Maintainability
→ Scalability.
→ Collaboration.

By following a stepwise abstraction approach, developers can manage the complexity of slw systems and create designs that are both flexible and robust.

The problem of decompiling:

Compilers are common, and decompilers are somewhere blw scarce and nonexistent - it depends on the constraints employed.

There are a number of reasons why the

decompiling operations demanded by our SDLCs can be made to fall into the realm of factable problems.

→ We are not required to decompile from machine code.

→ We are not required to decompile from base code.

→ We are not required to decompile a complete s/w system.

→ We are not required to decompile a monolithic black box.

Decompiling is a simple and straightforward task.

Some degree of decompilation is unavoidable when designing and developing complex s/w systems.

The widely accepted sentiment that system development should focus on the specification and not the implementation.

Decompilation is an essential component of SDLCs. But decompilation is a difficult process.

Decompiling refers to the process of converting executable code such as machine code or bytecode back into a higher-level programming lang., often resembling the original source code.

While decompiling can be a useful tool for understanding and analyzing s/w.

Challenges of Decompiling:

→ Decompilation often results in the loss of

comments, variable names, and other symbolic information present in the original source code.
→ Compiled code is often optimized for performance, and decompiling may not accurately reconstruct the original high-level constructs.
→ Some information may be lost during the compilation process, making it impossible to fully reconstruct the original source code.
→ Decompiling may face challenges when dealing with platform-specific optimizations or features.

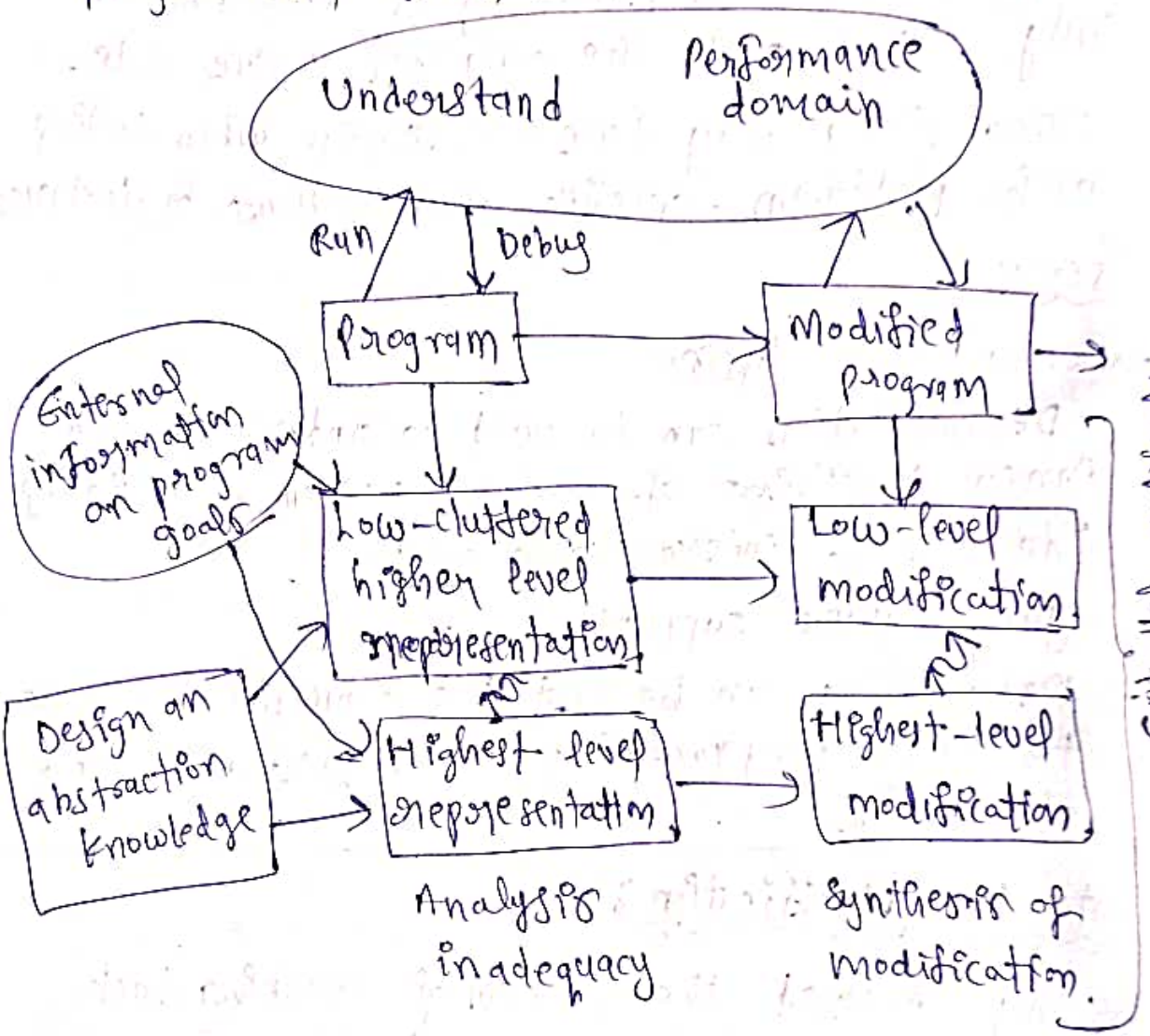
Uses:
→ security analysis
→ Decompilation can be used to understand the inner workings of a closed system, facilitating the dev of interoperable solutions.
→ Legacy system support.
→ Decompiling can be used for educational purposes to study programming techniques algos and sw design.

Controlled Modification:

Having exposed the problem of working back from implementation to specification, for evolutionary system dev that pushes against the tide of the second law of program evolution.

Controlled modification is the process of generating system modifications by tackling each specific problem at its source (i.e in the specification or in the detailed design etc) and then redesigning and reimplementing the changes into the system code.

The style of modification introduced should be constrained by the principles of structured programming. The move from one version of the program to the next is accomplished on most occasions by a various abstractions from the programmed system.



Controlled modification as a part of RUDE cycle.

- The construction and maintenance of the design history of a slw system would be an onerous task.

Certain abstractions can be automatically generated provided the programmer observes certain rules in his/her programming activity.

modern programming languages in supporting both data structure abstraction and control abstraction bring this automatic abstraction goal nearer.

Careful system dev in accordance with a set of principles and guidelines for the distribution of coded structures would allow the appn of abstraction algm which can, on a purely syntactic basis, generate the algm and data structure abstractions embodied in the working system code.

The ADA package is an information-hiding mechanism, it has 2 parts: a public or specification part, and a private or implementation part.

The specification part constitutes the abstract specification of the package.

The implementation part contains an implementation of the abstract specification.

Notice that the detailed distribution procedure employed will be determined by both the abstraction algm available and the type of abstract representation required by the system developer.

ADA package comprises a collection of rules, and mechanisms for adding, accessing, and removing rules.

All code from the reserved word PRIVATE to the end of the package is inaccessible to users of the package.

The knowledge base is composed of rules. A rule is composed of a condition part and an action part.

The following operations on the knowledge base are possible:

→ Rules can be added to the KB.

→ The next rule can be obtained

→ Rules can be deleted.

The sort of automatic abstraction facility is implemented within the support environment of the commercially available object-oriented language.

Controlled modification generally refers to the intentional and systematic alteration of something in a managed or regulated manner.

Version Control:

The use of version control systems to manage and track changes to source code.

Feature Branching:

Creating isolated branches for developing new features or modifications.

CI/CD: Continuous Integration / C. Deployment

Automated processes for integrating code changes and deploying applications.

Prototyping:

Creating prototypes to test and modify designs before full-scale production.

This approach is crucial for managing complexity, minimizing risks, and ensuring that modifications are aligned with objectives and ethical standards.

Structured Growth :-

there is much more to be abstracted from a program than the original design, structures and specification.

Conventional s/w engg spurred on by the s/w crisis. Structured programming is the common name for the collection of rules and looser guidelines that the responsible system developer uses in order to maximize the clarity of his or her programmed artefacts.

Structured growth meaning as follows " an initial program with a pure and simple structure is written, tested, and then allowed to grow by increasing the ambition of its modules.--- The growth can occur both " horizontally", through the addition of more facilities and "vertically" through a deepening of existing facilities and making them more powerful in some sense.

One of the key ideas behind the effective use of techniques of structured growth is that you are only introducing modifications to a stable, well-structured and well-tested basic system.

"Structured growth" refers to a deliberate and organized approach to expansion in dev, where the process is carefully planed, guided, and managed.

Strategic Planning :

Developing a detailed strategy that outlines goals, milestones, and the steps needed for growth.

Provides a roadmap, for decision-making, resource allocation, and progress evaluation.

Resource Allocation:

Allocating resources such as finances, manpower, and technology in a systematic manner to support growth initiatives.

Risk Management:

Identifying potential risks and implementing strategies to mitigate or mange them.

Customer-Centric Approach:

Placing a strong emphasis on understanding and meeting the needs of customers (or end users.

Examples:

→ Business expansion
→ Technology development
→ Personal development
→ Organizational scaling
→ Product life cycle mangement.

Benefits:

→ Efficiency
→ Minimized risks
→ Scalability
→ Sustainability
→ Improved decision making
→ Adaptability.

Gyla 04-01-2024 10:15 PM.

05-01-2024.

Machine Learning: Much promise, Many Problems

3 self-adaptive software, the promise of increased software power, the threat of increased software problems.

ML indeed holds significant promise, revolutionizing various industries and driving technological advancements.
It also comes with its fair share of challenges and concerns.

Much Promise:-

Automation and Efficiency:

ML enables automation of tasks that traditionally required human intelligence, leading to increased efficiency.

Pattern Recognition:

ML excels at recognizing patterns and making predictions based on data, which is valuable for decision-making and problem-solving.

Personalization:

ML algm can personalize user experience, such as in recommendation systems, providing tailed content or suggestion.

Medical Advances:

ML contributes to medical diagnostics, drug discovery, and personalized medicine, potentially

saving lives and improving healthcare outcomes.

Enhanced UI:

NLP and computer vision improve UI, making interactions more intuitive and user-friendly.

Predictive Analytics:

ML enables businesses to make data-driven predictions, helping in forecasting trends, demand, and market behavior.

Many Problems:-

Bias and Fairness:

ML models can inherit and perpetuate biases present in training data, leading to unfair and discriminatory outcomes.

Lack of Transparency:

Many ML models, particularly DL models, operate as "black boxes", making it challenging to understand how they reach specific decisions.

Data Privacy and Security:

ML systems often require vast amounts of data, raising concerns about privacy and the security of sensitive information.

Robustness and Reliability:

ML models can be sensitive to changes in i/p data, and adversarial attacks can manipulate them.

Interpretable Models:

Understanding and interpreting complex ML

models is difficult, especially for non-experts.

Ethical Concerns:

ML app^n's raise ethical questions, such as accountability for automated decisions and the potential for job displacement.

Generalization Issues:

Models that perform well on training data may struggle with new, unseen data.

ML mechanisms will be a necessary part of many AI systems. Engineering artificially intelligence s/w is a well understood and boring procedure which is in need of some injection of excitement.

Self-adaptive s/w:-

The code that actually constitute the program will vary as the program runs through an execution sequence.

Debugging is not one of the favored pursuits of most s/w engineers. They are fundamentally creative people, system designers and builders. Looking for errors in such creations is an. exercise.

The advent of high-level languages has done much to improve the bleak picture.

The principle was:

"Never write self-modifying code."

Self-adaptive software refers to systems (or app^n's) that have the ability to autonomously adjust their behavior and configurations in response to changing environmental conditions, user

requirements or other factors.

The goal is to enhance system performance, optimize resource usage, and improve overall adaptability.

Here are some key aspects and characteristics of self-adaptive s/w.

Monitoring and sensing:

Self-adaptive systems continuously monitor their environment, internal state, and relevant external factors.

Decision-Making:

Based on the information gathered through monitoring, self-adaptive s/w makes decisions autonomously.

Feedback loops:

Self-adaptive systems often operate on feedback loops. They assess the impact of their decisions and use this feedback to refine and adapt their strategies over time.

Dynamic Reconfiguration:

self-adaptive s/w can dynamically reconfigure its components or settings to respond to changing conditions.

Goal Specification:

self-adaptive systems are typically designed with predefined goals and objectives.

These goals guide the system in making decisions that align with the desired outcomes, such as maximizing performance, minimizing energy consumption, or ensuring reliability.

Adaption Mechanisms:
various adaption mechanisms are employed, including rule-based systems, ML algms, & other computational intelligence techniques.

Fault Tolerance:
self-adaptive s/w often includes mechanisms to detect and respond to faults & failures. It may dynamically reconfigure itself to maintain functionality.

Verification and Validation:
Ensuring the correctness and effectiveness of self-adaptive systems can be challenging. verification and validation techniques are essential to confirm that the adaptive behavior aligns with the system goals ~~without~~ &.

Examples of self-adaptive s/w can be found in various domains, including autonomous systems, network management, cloud computing, and IoT applications.

The Promise of Increased s/w Power:-

Many significant enhancements of s/w power require a move from static, conten-free systems to dynamic content-sensitive systems.

Intelligence is not a context-free phenomenon, and AI can not be either.

The need for self-adaptive s/w derives from several sources: there is a need for s/w that is reactive to changing circumstances, and at the more mundane level there is a need for mechanisms

to lessen the difficulty of the task of incrementally upgrading knowledge bases

The promise of increased s/w power encompasses the potential for s/w to deliver enhanced capabilities, efficiency, and transformative impacts across various domains.

Computational Power:

Advances in s/w design and optimization contribute to increased computational power.

This enables app^ns to perform complex calculations, simulations, and data processing tasks more efficiently.

Scalability:

Scalable s/w can handle increased workloads, user interactions, or data volumes without significant performance degradation.

AI and ML:

The integration of advanced alg^ms and ML techniques into s/w empowers app^ns to learn, adapt, and make intelligent decisions.

Parallel and Distributed Computing:

S/w power is augmented by the effective utilization of parallel and distributed computing.

Real-time Processing:

High-powered s/w enables real-time processing of data, facilitating instant decision making and response.

User Experience:
Powerful s/w enhances the user experience by providing responsive, feature-rich, and visually engaging interfaces.

Data Processing and Analytics:
s/w power plays a crucial role in handling vast amounts of data.
Advanced analytics and data processing capabilities enable organizations to derive valuable insights, make informed decisions.

Security and Robustness:
s/w power is also associated with robust security features. Advanced encryption, authentication mechanisms, and secure coding practices contribute to building resilient and secure s/w app's.

The threat of Increased s/w problems:-

→ systems are designed and developed more systematically using appropriate abstract representations which reduce the effective complexity by several orders of magnitude.

→ With the dev of elaborate machine-manipulate data structures program complexity can be traded out of the alg^m.
As a result s/w adaptability can be gained.

→ s/w can now be developed within a sophisticated support environment, which can remove a myriad of trivial considerations from the concern of the programmer.

ML does not have to be self-modifying code.

The increasing complexity and ubiquity of s/w bring about various challenges and threats that can lead to s/w problems.

Security Vulnerabilities:

Complex codebases may contain undiscovered flaws & loopholes that malicious actors can exploit, leading to data breaches, unauthorized access & other cyber threats.

Bugs and Errors:

These issues can lead to system crashes, unexpected behavior, & data corruption, affecting the overall reliability of s/w app^ns.

Performance Degradation:

~~Highly complex s/w may~~

this can result in slower response times, increased resource consumption, and diminished overall user satisfaction.

Compatibility Issues:

s/w problems may arise when app^ns need to interact with other s/w & hardware components.

Inadequate Testing:

Inadequate testing may lead to the release of s/w with undetected issues, compromising its stability and security.

Software Bloat:

s/w may accumulate unnecessary features and functionalities, leading to s/w bloat.

Addressing the threat of increased s/w problems requires a comprehensive approach that includes robust testing practices, ongoing maintenance, security measures, and user feedback.

The state of art in ML :-

The ML mechanisms with some promise of near-term practical utility can be divided into the classical ones such as inductive generalization, and the network learning models such as back propagation of an error signal.

A PDP-parallel Distributed Processing system is typically a network of primitive processing elements that receive 'activity' values from other elements that are directly connected into them.

The general strategy of ML in PDP systems is one of error-signal feedback.

The general strategy for self-adaption based on inductive generalization.
