UNIT-5

AI into Practical Software

* Support environments, Reduction of effective complexity, Moderately stupid assistance, An engineering toolbox, Self-reflective software, Overengineering software, Summary and what the Future Holds.

✓ Engineering practical AI s/w is a very difficult task.

Integrating AI into practical s/w involves several steps.

Define objectives and Use cases:

→ Clearly define the objectives of integrating AI into your s/w.

Data Collection and Preparation:

Gather relevant data for training and testing your AI model.

Choose the Right AI Model:

Select an appropriate AI model based on your objectives and the nature of your data.

Common models include ML algms, DL models, NLP models, and more.

Training the Model:

Use the collected data to train your chosen AI model.

Fine-tune the model to improve its performance.

Integration with S/w:
Once your AI model is trained, integrate it into your S/w.
APIs and SDKs:
Consider using pre-built APIs (or SDKs - S/w dev kits) provided by AI service providers.
Cloud services such as AWS, Azure & Google Cloud offer AI services and APIs for easy integration.
UI Design:
Design a UI that accommodates the AI features.
Testing and Validation:
Test the integrated AI system to identify and fix any bugs (or issues.
Validate the accuracy and performance of the AI model in real-world scenarios.
Security and Privacy:
Implement security measures to protect the AI model and user data.
Monitoring and Maintenance:
Set up monitoring tools to keep track of the AI model's performance over time.
Feedback and Iteration:
Collect feedback from users to identify areas for improvement.
Use feedback to iterate on your AI model and S/w, making continuous enhancements.

Support Environments :-

It was a major feature of AI programming in the days before conventional programming fully realized its importance.

In conventional programming we are translating a formal specification into an alternative notation that will result in an effectively computable algm.

programming :

programming from the conventional computer science viewpoint, is the transformation of a formal specification into a computational procedure for correctly realizing the given specification.

for Dijkstra " the programmer's main task is to give a formal proof that the program he proposes meets the equally formal functional specification "

AI programming has always been a process of exploratory prototyping and evolutionary system dev.

programming in AI is the exploration of behavorally adequate approximations to an incompletely specified problem.

LISP is a terrible programming language when considered in isolation and as a vehicle for conventional programming.

AI programming raises flexibility, as demanded by experimentation and evolutionary dev.

When integrating AI into practical sw, ensuring support for various environments is crucial.

Cross-Platform Compatibility:
Ensure that your s/w and AI components are compatible with diff OS such as Win, MacOS, etc

Web-Based Support:
Consider making your AI-powered s/w accessible through web browsers.

Mobile Devices:
If applicable, optimize your s/w for mobile devices.

Cloud Integration:
Ensure compatibility with popular cloud providers such as AWS, Azure, Google Cloud etc.

Edge Computing:
Optimize your models for deployment on edge devices like IoT devices, edge servers or embedded systems.

Integrating with Existing Systems:
Ensure that your AI s/w can integrate seamlessly with other existing systems and s/w commonly used in the industry.

Reduction of effective Complexity:-
With the task of programming not restricted to the problems of deriving a correct algorithm from a fixed and well-defined specification.
The lack of built-in constraints in typical AI languages, AI programming becomes an unmanageably complex process.

e programming environments of the first generation were designed to support just the task of actually writing and debugging code.

ny useful functionality that we can add to a system-dev environment may prove to be a true step down the long road to our final goal.

he benefits of a complete life-cycle environment: all useful design and dev information can be made available to the system maintainer.

There are a number of quite straight forward ways that support environments can be developed to reduce the effective complexity of sys dev.

Modular Design:

Break down the system into modular components with well-defined interfaces.

Abstraction:

Use abstraction to hide unnecessary details and expose only relevant information.

Documentation:

Maintain comprehensive documentation that explains the architecture, algms, and design decisions.

Feature Pruning:

Remove or deprecate features that are rarely used or do not contribute significantly to the systems goals.

Code Refactoring:

Periodically review and refactor code to eliminate redundancy and improve readability.

Automated Testing :-

Implement comprehensive automated testing to catch bugs and regressions early.

Moderately Stupid Assistance :-

The four features of the 'assistant' are

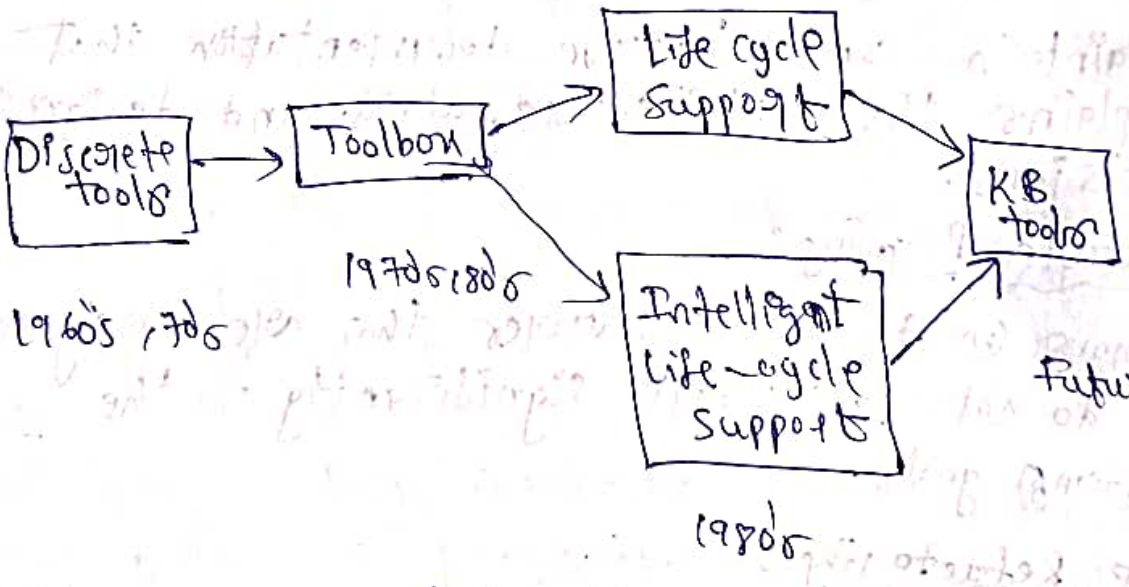
(i) Semantic (as well as syntactic) error checking could be a feature of such a system.

(ii) Answering Questions

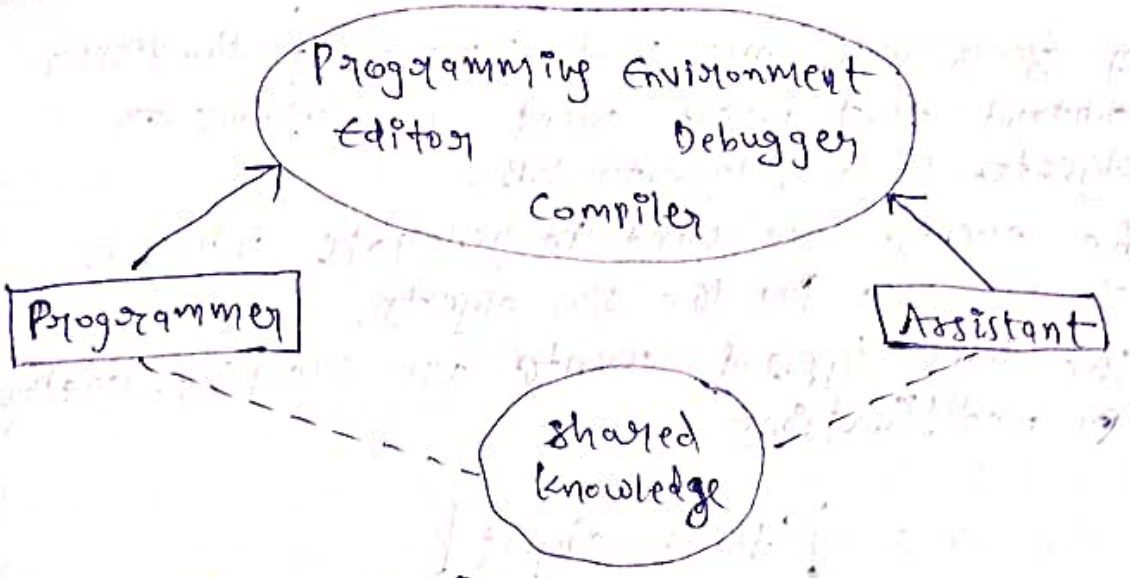
(iii) The system should be capable of filling in trivia without bothering the programmer with the actual details that it automatically generates.

(iv) The system should also posses some expertise in debugging programs.

An ideal s/w dev environment should provide a powerful language for the application (as well as appropriate methodology and tools for program dev.



"Classification of dev. environment".



A programming assistant.

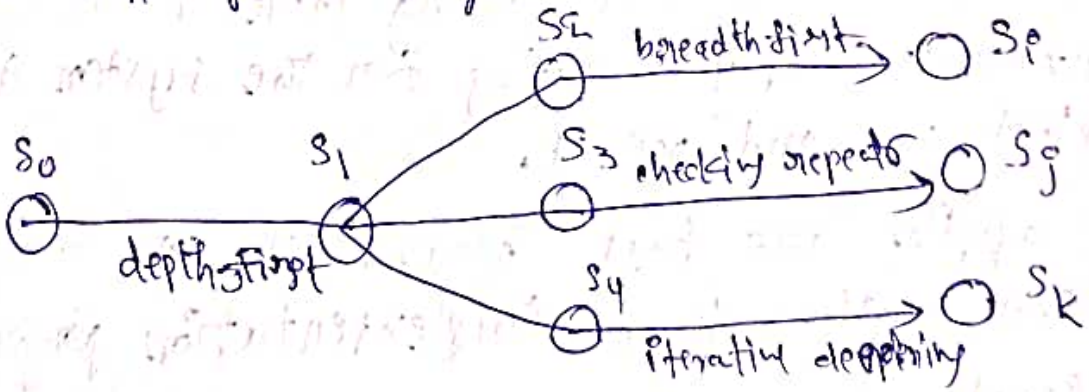
The assistant interacts with the tools in the environment.

A "plan" is an abstract representation of a program. The "plan formalism" is designed to represent two basic types of information:

→ The structure of particular programs
→ Knowledge about cliches.

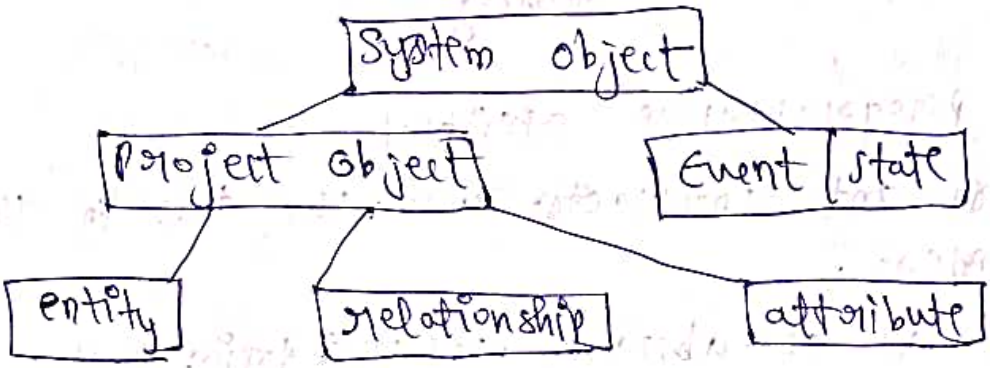
Each new version of the system has a parent and may be the parent of any number of descendents.

The basic arrangement of versions to be managed is that of a tree. Each node in the tree is a version of the system under dev.



"A tree of versions".

It is a new method of supporting s/w version control that uses event-annotations on objects in a project KB.
The events are used to generate intervals of existence for the s/w objects.
The basic types of events are creations, deletions, & modifications.



Taxonomy of s/w dev concepts in ESDE.
ESDE - Exploratory system Dev Environment.

✓ An engineering toolbox :-

The "assistant" metaphor emphasizes the active role that the computer may play, offering unsolicited help, advice and words of caution.
The toolbox metaphor promotes the more passive possibilities: a homogeneous collection of powerful tools just sitting there in the environment simply waiting for the system developer to select one and use it.
Some apps are best thought of as design problems, rather than implementation projects.
The first advance towards a power-packed personal programming environment was the

nduction within a single shell of standard
atures of the programming— text editors, emailetc.

(i) Environments concerned with the entire life cycle of a program in system.
these environments are generally focused on managing the dev of large systems built by many programmers.
They serve as the repository of all information on the system and thus contain DB of the relevant information from each stage of the dev process.

(ii) Environments concerned primarily with the coding phase and whose tools are relatively independent from each other.

(iii) Environments which regard coding, debugging, testing and maintenance as a single process of program dev through incremental enrichment..

One last point about sw support environments is that it appears to be the case that support for effective exploratory programming is also support for speculative code hacking.
There is no need to restart the the computation from the beginning and no need to recompile after a modification of the code.

Creating an engineering toolkit involves assembling a collection of tools, resources, and information that can assist engineers in their work.

Prototyping and Testing equipment:
3D printers and CNC machines for prototyping.

Testing equipment for: materials, electronics etc.

Programming and Scripting Langs:
Depending on the field, knowledge of languages like Python, MATLAB, & C++ can be valuable.

Data Analysis and Visualization:
Data analysis tools like Excel, Python with Pandas & MATLAB.
Visualization tools for creating graphs and charts.

Project management tools:
Project management software like Microsoft Project & Trello.

Collaboration and Communication Tools:
Communication platforms like slack & Microsoft Teams.
Version control systems like Git for collaborative coding.

Code Repositories:
Access to repositories like GitHub for sharing and collaborating on code.

Self Reflective Software :-

"Self-reflecting software" typically refers to s/w app^n & systems that possess the ability to introspect, analyze, and adapt their own behavior & performance.

Monitoring and Logging:
Improvement robust monitoring and logging mechanisms to collect data on the s/w behavior,

performance, and interactions.

Telemetry and Metrics:
Incorporate telemetry and metric collection to gather quantitative data about the systems performance, resource usage (and user interactions.

Diagnostic Tools:
Include diagnostic tools that can analyze the internal state of the s/w, helping to identify and troubleshoot issues.

Self-Analysis Algorithms:
Develop alg^ms that enable the s/w to analyze its own performance, detect patterns, and identify areas for improvement.

Adaptive Learning:
Implement ML (or adaptive alg^ms that allow for s/w to learn from its own data and adjust its behavior over time.

Dynamic Configuration:
Enable the s/w to dynamically adjust its configuration based on changing conditions (or performance metrics.

Self-Optimization:
Build mechanisms for the s/w to optimize its own processes, such as resource allocation, to enhance efficiency and performance.

Autonomous Decision-Making:
Integrate decision-making capabilities that enable the s/w to make autonomous choices based on its self-analysis and predefined criteria.

Fault Tolerance:

Implement features that allow the s/w to detect faults or errors.

User-Feedback Integration:

Incorporate mechanisms to gather user feedback and preferences.

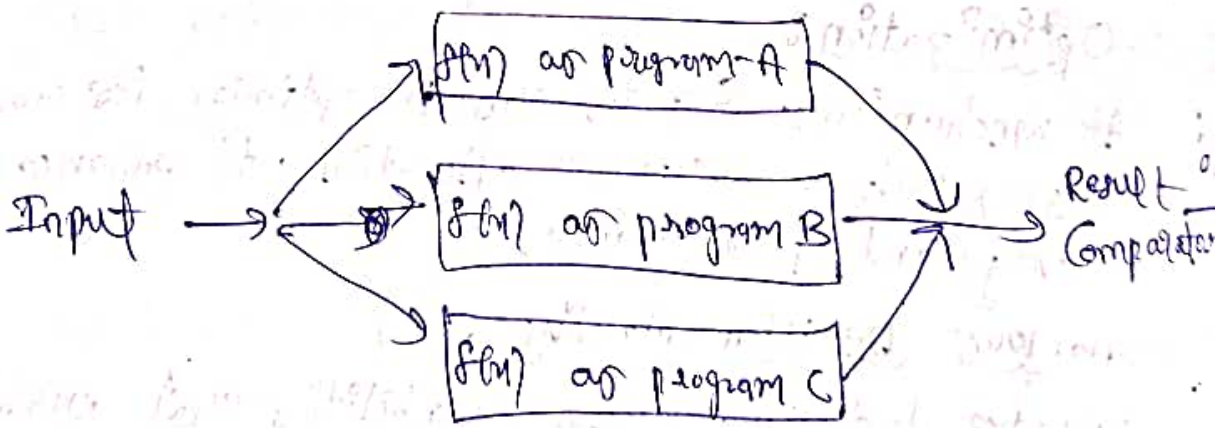
Developing self-~~adap~~ reflective s/w involves a combination of advanced technologies, including AI, ML, and sophisticated algm.

✓ Overengineering Software:-

The terms "overengineering" or "adding redundancy" might be same.

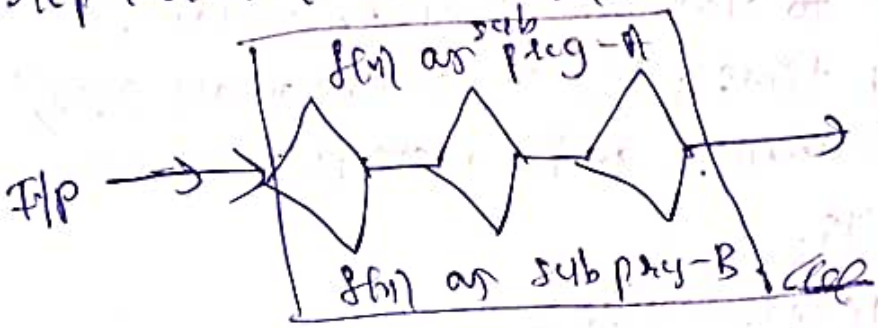
A relevant blossoming field of logic programming is that of constraint logic programming. (CLP).

A frequently used technique for improving s/w system reliability is the replication of the critical s/w component together with a 'comparator' module that cross-checks for consistency.

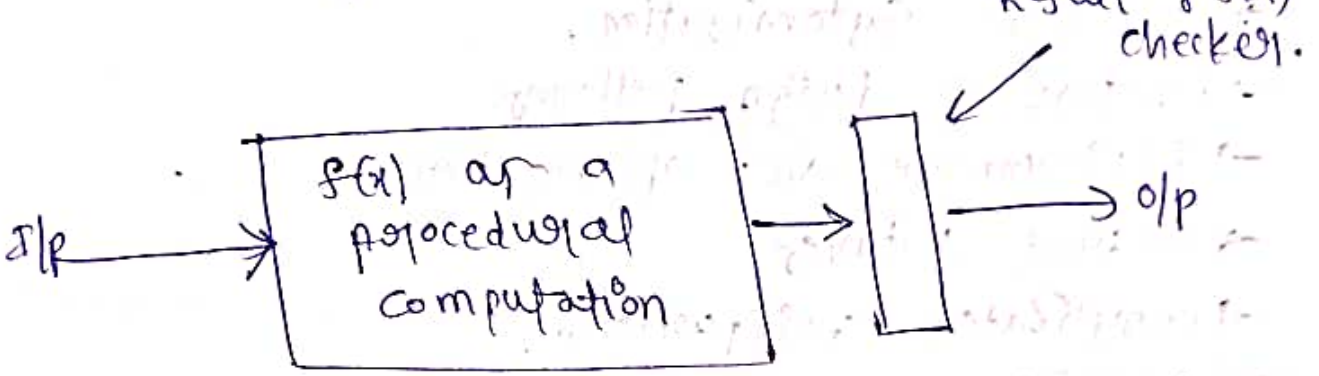


Simple s/w diversity providing system redundancy.

The bifacial language used abstract data types to solve the problem of determining when two results were equal when their concrete representations differed.



Redundancy through bifacial computation.



The object-oriented paradigm provides us with an up-to-date perspective on the assertion based approach to s/w reliability.

A deferred class is a class that contains at least one deferred routine, which is a routine whose implementations will only be provided by descendents.

Features such as polymorphism and late binding, which permit considerable type flexibility; give the programmer significant freedom to design systems using deferred classes and routines.

Overengineering has long been recognized as a route to reliable s/w systems.

Over engineering in s/w dev occurs when a solution is excessively complex & feature-rich for the problem it aims to solve.
This can lead to various issues, including increased dev time, higher maintenance costs, reduced performance, and a steeper learning curve for users.

Signs of overengineering:

→ Unnecessary Complexity.
→ Excessive Customization.
→ Overuse of design patterns
→ Performance over optimization.
→ Unused features.
→ Complicated configuration.

Reasons for overengineering:

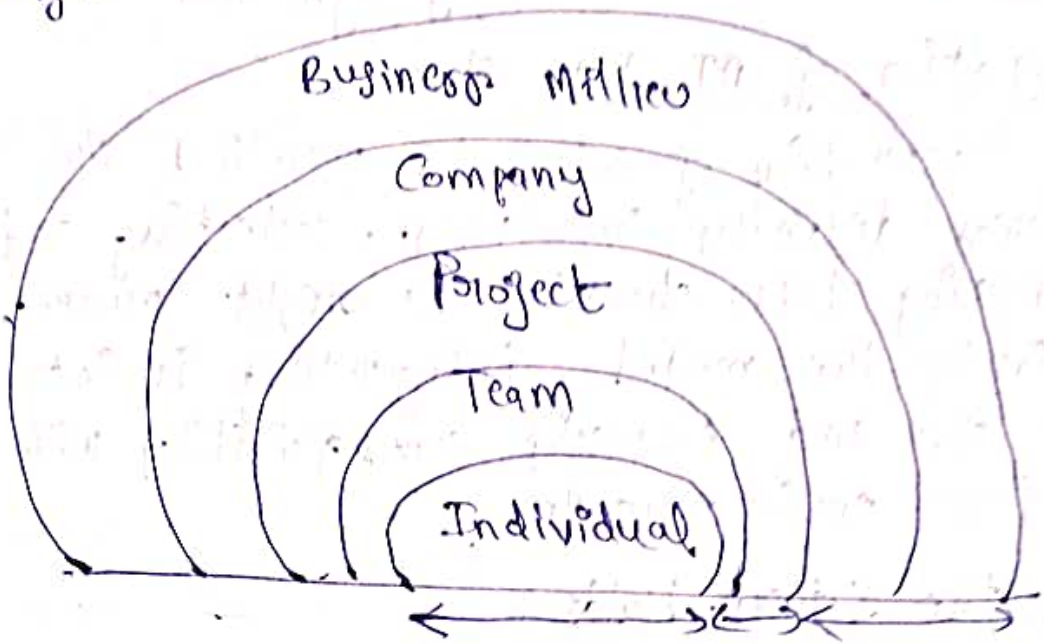
→ Gold plating
→ Fear of Missing out (FOMO)
→ Lack of clarity in requirements.
→ Lack of communication.

By recognizing the signs of overengineering and addressing the root causes, dev teams can create more efficient, maintainable, and user-friendly s/w solutions.

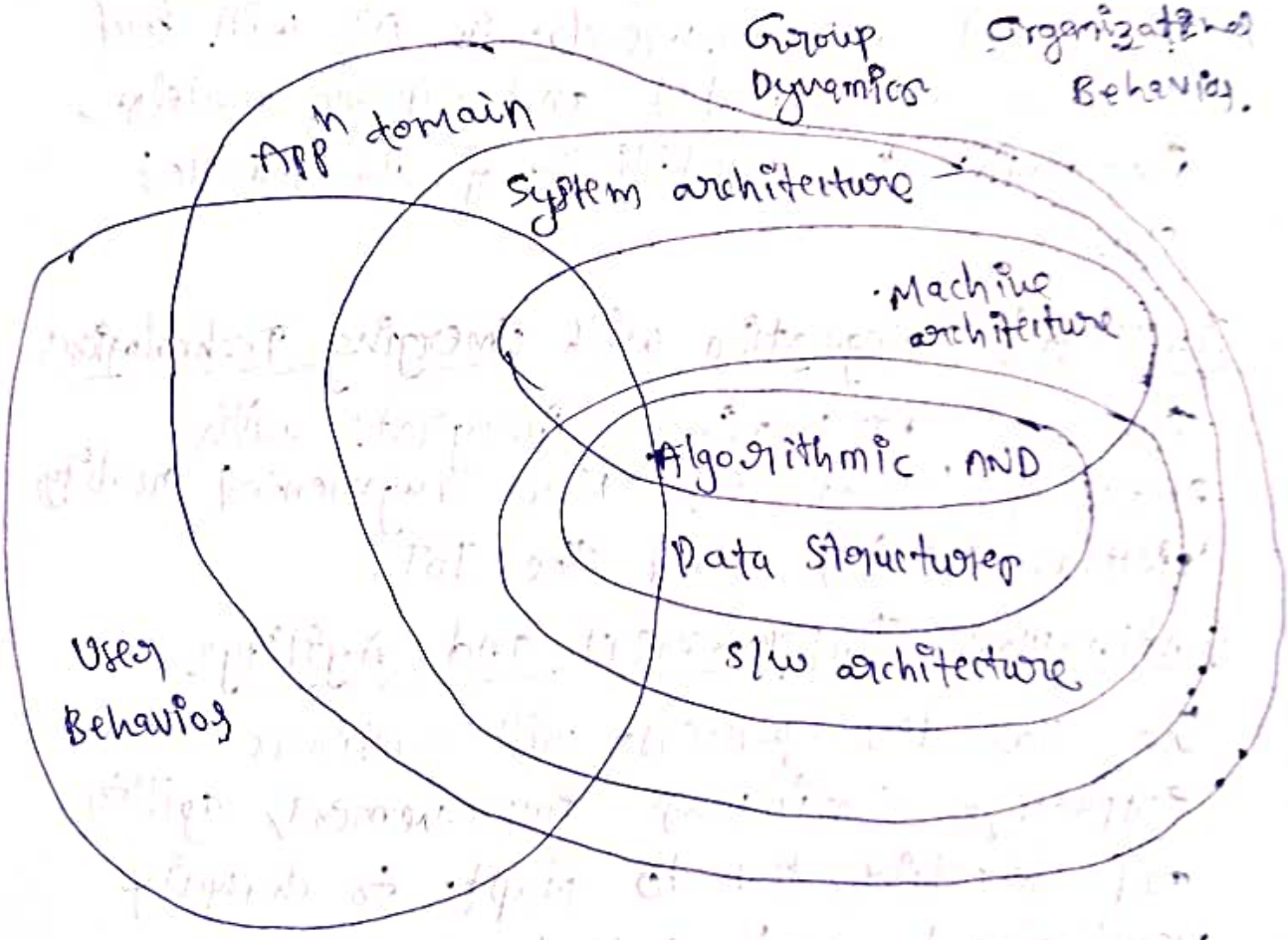
Summary and What the future Holds:-

S/w dev is not a purely technical issue. S/w products operate in society.

Many s/w systems are built by a team of people, and not by just one lone hacker.
Once a group of people are involved certain problems are reduced, but certain new problems emerge.



Content of analysis → cognition & motivation



"Knowledge domains involved in system building"...

Finally, s/w design and dev should be more like engineering.

S/w artefacts, especially AI-ish ones are different from conventionally engineered artefacts in a number of significant respects.

Integration of AI Into S/w:

The integration of AI into practical s/w involves defining objectives, collecting and preparing data, choosing the right model, training the model, integrating it into the s/w, and ensuring compatibility with various environments.

What the future Holds:

Advancements in AI:

Continued advancements in AI will lead to more sophisticated and capable models, improving the capabilities of AI-powered s/w.

Increased Integration with Emerging Technologies:

S/w will increasingly integrate with emerging technologies like augmented reality, virtual reality, and the IoT.

Continuous Improvement and Agility:

The s/w dev process will continue to emphasize continuous improvement, agility, and iterative dev to adapt to changing requirements and technologies.

Decentralized and Edge Computing:

With the rise of edge computing, s/w will be optimized for decentralized processing, enabling real-time data analysis and reducing latency.

Open Source Collaboration:

Collaboration within the open-source community will remain essential, fostering innovation, knowledge sharing, and the dev of robust and sustainable s/w solutions.

As technology continuous to evolve, the future of s/w dev will be shaped by a combination of innovation, adaptability, and a focus on creating solutions that enhance efficiency, security, and user experience.

GSLd 05/01/2024
9:00 PM