MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Access Control Policy Language.

Mentor: prof. Cojuhari Irina
Students: Țărnă Cristina, FAF-211
Sclifos Tudor, FAF-211
Hariton Dan, FAF-211
Caminschi Leonid, FAF-211
Ceban Andrei, FAF-211

Chișinău, 2023

Abstract

According to our standards, the first paragraph of the abstract needs to contain the following:

- Name of the project,
- Author/-s,
- Institution,
- Report structure.

Keywords: LaTeX, TeX, Research Paper, Project.

A typical abstract is a 150 to 250 word paragraph that provides readers with a quick overview of the essay or report and the way it is organized. It should express the central idea of the project and the key points. It should also mention possible applications of the research being discussed in the paper.

Some of the things that can be included in the abstract are:

- The purpose of the project,
- The general objectives of it,
- Used methodologies,
- The obtained results.

Content

Introduction

The purpose of the introduction is to inform the reader/evaluator about the content and the general objectives of the paper. In this part of the paper, the following aspects should be covered:

- Studied domain,
- Motivation for choosing the topic,
- Degree of novelty/relevance of the topic or of the problem,
- General objectives of the project,
- Research methodologies that have been used,
- A brief sneak peek at the content of the subsequent chapters.

The volume of the introduction usually is about 2-3 pages.

1 Domain Analysis

The first chapter of the project report contains background information about the problem, the associated domain and the impact the problem has on the identified domain. Also, when knowing the domain, one could establish target groups and perform customer validation in order to justify the relevance of the problem and the need for the project itself.

1.1 Description of the domain

A domain-specific language (DSL) is a software language that's domain-specific. A software language is a language that's written and read by humans, but also automatically processable by computers, and supported by the software. In this section, we'll figure out what it means for a software language to be domain-specific, and what the key aspects of a software language are.

A domain is a particular, focused area of knowledge. It comes with a group of domain experts "inhabiting" or "owning" the domain: people possessing, shaping, extending that area's knowledge, and sharing it with business stakeholders and other interested parties. You could even argue that a domain often exists in the first place as such a group of domain experts, with that group defining the domain in an ad hoc manner.

Domain experts typically converse with each other using a domain-specific ubiquitous language – even if they don't realize they do. Although the term "domain-specific ubiquitous language" already gives us the S and the L, that language is not yet a proper DSL yet. The domain-specific ubiquitous language generally comes in both a verbal and written form. The written form is used for the domain's body of knowledge, and for task-specific documents such as the specification of a Runtime. The verbal form is used by the domain experts to discuss the body of knowledge and task-specific documents.

The domain-specific ubiquitous language is usually based on a natural language but is enriched with domain-specific terms which are given precise, domain-specific meanings. It's also often enriched with domain-specific notation. The ubiquitous language is not yet a proper DSL: it's not a software language, because it lacks a solid definition, as well as software tools.

This pre-DSL makes up fertile ground for a proper DSL: it's already domain-specific but needs work to become a software language. Looking at the key aspects of a software language, we can determine how we can "extract" a proper DSL from the pre-DSL. With that information, we can start implementing the DSL as a software language. That implementation takes the form of tools such as the Domain IDE, and the code generator. The Domain IDE is an application that allows domain experts to write (and read) DSL content using DSL editors.

Why use a DSL-based approach for software development? Adopting a DSL-based approach should

have several positive effects on the software development process.

Here are some potential motivations for using a DSL: Abstraction: A DSL allows developers to abstract complex programming concepts and focus on a specific domain. This can make the code more readable and easier to maintain.

Productivity: With a DSL, developers can create software quickly and efficiently. DSLs allow developers to express ideas in a concise and expressive way, which can speed up development time.

Improved Communication: DSLs can improve communication between developers and non-technical stakeholders. With a DSL, business analysts or domain experts can more easily express their needs and requirements, without requiring a deep technical understanding.

Safety: A DSL can help to catch errors early, before they cause problems. By providing a language that is designed specifically for a domain, developers can write code that is less error-prone.

Customization: A DSL can be customized to a specific problem or domain, making it more powerful and easier to use.

Domain-Specific Optimizations: A DSL can be optimized specifically for the domain it serves. This can lead to more efficient code and better performance.

A query language is a programming language designed to retrieve and manipulate data stored in a database. It is a specialized language that allows users to ask specific questions, or queries, about data in a structured way. There are various query languages used for different types of databases, including SQL (Structured Query Language), which is used for relational databases, and NoSQL query languages like MongoDB query language and Cassandra Query Language, which are used for non-relational databases. Query languages typically consist of statements or commands used to interact with a database, such as SELECT, INSERT, UPDATE, and DELETE. These statements allow users to retrieve specific data from a database, add new data, update existing data, and delete data. Query languages are essential for working with databases, as they allow users to access and manipulate large amounts of data in a structured and efficient way[1].

1.2 Problem formulation

The problems that might occur, solved by the query language with role-based access control (RBAC) are the following:

1. Access Control Management: How can we efficiently retrieve and analyze information about users, roles, and permissions within an organization's system to ensure that access control policies are properly enforced and that users have appropriate permissions to perform their job responsibilities?

2. User Activity Tracking and Reporting: How can we retrieve and generate reports on user activity, such as login attempts, resource accesses, and changes to access control policies, to ensure compliance and identify potential security risks or policy violations?

3. Access Control Troubleshooting: When issues arise with access control, such as users being denied access to resources or unauthorized changes to access control policies, how can we use a query language based on RBAC to quickly identify the root cause of the problem and resolve it?

4. Access Control Auditing: How can we use a query language based on RBAC to track and audit changes to access control policies, such as the addition or removal of roles or permissions, to ensure that changes are made in a controlled and authorized manner and to maintain an audit trail for compliance purposes?

5. Role and Permission Assignment: How can we efficiently assign roles and permissions to users based on their job responsibilities, using a query language based on RBAC to automate the process and ensure consistency across the organization?

6. Access Control Policy Optimization: How can we use a query language based on RBAC to optimize access control policies, such as identifying roles that have overlapping permissions or roles that have been granted unnecessary permissions, to ensure that access control policies are as efficient and effective as possible?

1.3 Target Group

The target group for a query language based on RBAC would typically be IT professionals and developers who are responsible for managing and maintaining access control policies within an organization's system. This would include:

• Security Analysts: These professionals are responsible for monitoring and analyzing the security of an organization's systems, and a query language based on RBAC would be a valuable tool for them to retrieve and analyze information about access control policies and user activity.

• System Administrators: These professionals are responsible for managing the organization's system infrastructure, including access control policies. A query language based on RBAC would allow them to efficiently manage user roles and permissions and troubleshoot access control issues.

• Developers: These professionals are responsible for developing and maintaining the organization's system software, including access control features. A query language based on RBAC would allow them to efficiently build RBAC functionality into the software and troubleshoot any issues that arise.

• Compliance Officers: These professionals are responsible for ensuring that the organization's systems comply with relevant laws and regulations. A query language based on RBAC would be a valuable tool for them to track and audit access control policies and user activity for compliance purposes.

Overall, the target group for a query language based on RBAC would be professionals who need to manage and maintain access control policies and ensure the security and compliance of an organization's systems.

1.4 Comparative Analysis

1.4.1 ABAC

With Attribute-based Access Control (ABAC), the authorization process involves evaluating attributes rather than just relying on roles. The aim of ABAC is to secure objects, including data, network devices, and IT resources, against unauthorized access and actions by individuals who lack the ”approved” characteristics set forth in the security policies of the organization.

With ABAC, an organization's access policies enforce access decisions based on the attributes of the subject, resource, action, and environment involved in an access event.

The subject is the user requesting access to a resource to perform an action. Subject attributes in a user profile include ID, job roles, group memberships, departmental and organizational memberships, management level, security clearance, and other identifying criteria. ABAC systems often obtain this data from an HR system or directory or otherwise collect this information from authentication tokens used during login.

The resource is the asset or object (such as a file, application, server, or even API) that the subject wants to access. Resource attributes are all identifying characteristics, like a file's creation date, its owner, file name and type, and data sensitivity. For example, when trying to access your online bank account, the resource involved would be “bank account = ¡correct account number¿.”

The action is what the user is trying to do with the resource. Common action attributes include “read,” “write,” “edit,” “copy,” and “delete.” In some cases, multiple attributes can describe an action. To continue with the online banking example, requesting a transfer may have the characteristics “action type = transfer” and “amount = 200.”

The environment is the broader context of each access request. All environmental attributes speak to contextual factors like the time and location of an access attempt, the subject's device, communication protocol, and encryption strength. Contextual information can also include risk signals that the organization has established, such as authentication strength and the subject's normal behaviour patterns.

When it comes to ABAC, the benefits far outweigh the costs. But there is one drawback businesses should keep in mind before implementing attribute-based access control: implementation complexity.

1.4.2 RBAC

RBAC – Role-Based Access Control

In computer systems security, role-based access control (RBAC) or role-based security is an approach to restricting system access to authorized users. It is an approach to implement mandatory access control (MAC) or discretionary access control (DAC).

Role-based access control is a policy-neutral access-control mechanism defined around roles and

privileges. The components of RBAC such as role permissions, user-role and role-role relationships make it simple to perform user assignments. RBAC can be used to facilitate the administration of security in large organizations with hundreds of users and thousands of permissions. Although RBAC is different from MAC and DAC access control frameworks, it can enforce these policies without any complication.

Design:

Within an organization, roles are created for various job functions. Members or staff (or other system users) are assigned particular roles, and through those role assignments acquire the permissions needed to perform particular system functions. Since users are not assigned permissions directly, but only acquire them through their role (or roles), management of individual user rights becomes a matter of simply assigning appropriate roles to the user's account; this simplifies common operations, such as adding a user or changing a user's department.

Role-based access control interference is a relatively new issue in security applications, where multiple user accounts with dynamic access levels may lead to encryption key instability, allowing an outside user to exploit the weakness for unauthorized access. Key-sharing applications within dynamic virtualized environments have shown some success in addressing this problem.

Rules

Three primary rules are defined for RBAC:

1. Role assignment: A subject can exercise a permission only if the subject has selected or been assigned a role.

2. Role authorization: A subject's active role must be authorized for the subject. With rule 1 above, this rule ensures that users can take on only roles for which they are authorized.

3. Permission authorization: A subject can exercise a permission only if the permission is authorized for the subject's active role. With rules 1 and 2, this rule ensures that users can exercise only permissions for which they are authorized.

Additional constraints may be applied as well, and roles can be combined in a hierarchy where higher-level roles subsume permissions owned by sub-roles.

When defining an RBAC model, the following conventions are useful:

• S = Subject = A person or automated agent

• R = Role = Job function or title which defines an authority level

• P = Permissions = An approval of a mode of access to a resource

• SE = Session = A mapping involving S, R and/or P

• SA = Subject Assignment

• PA = Permission Assignment

• RH = partially ordered Role Hierarchy. RH can also be written: ¿= (The notation: x ¿= y means

that x inherits the permissions of y.)

- A subject can have multiple roles.
- A role can have multiple subjects.
- A role can have many permissions.
- A permission can be assigned to many roles.
- An operation can be assigned to many permissions.
- A permission can be assigned to many operations.

A constraint places a restrictive rule on the potential inheritance of permissions from opposing roles, thus it can be used to achieve appropriate separation of duties. For example, the same person should not be allowed to both create a login account and to authorize the account creation.

1.4.3 XACML

XACML stands for ”eXtensible Access Control Markup Language”. The standard defines a declarative fine-grained, attribute-based access control policy language, an architecture, and a processing model describing how to evaluate access requests according to the rules defined in policies. The policy language is used to express access control policies (who can do what and when). The request/response language expresses queries about whether a particular access should be allowed (requests) and describes answers to those queries(responses). The reference architecture proposes a standard for the deployment of necessary software modules within an infrastructure to allow efficient enforcement of policies.

XACML supports Attribute-Based Access Control (ABAC) and evaluation can be done with the additional data retrieved from Policy Information Point (PIP) which is defined by the XACML reference architecture.

XACML policy language structure and syntax

Three top-level policy elements:

”Rule”: contains a Boolean expression that can be evaluated in isolation, but that is not intended to be accessed in isolation by a PDP. So, it is not intended to form the basis of an authorization decision by itself. It is intended to exist in isolation only within an XACML PAP, where it may form the basic unit of management

”Policy”: contains a set of ”Rule” elements and a specified procedure for combining the results of their evaluation. It is the basic unit of the policy used by the PDP, and so it is intended to form the basis of an authorization decision

”PolicySet”: contains a set of ”Policy” or other ”PolicySet” elements and a specified procedure for combining the results of their evaluation.

Rule-combining algorithms

The rule-combining algorithm specifies the procedure by which the results of evaluating the component rules are combined when evaluating the policy.

Some examples of standard combining algorithms are :

Deny-overrides (urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:deny-overrides)

— If any decision is “Deny”, the result is “Deny”.

Permit-overrides (urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:permit-overrides)

— If any decision is “Permit”, the result is “Permit”.

First-applicable (urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:first-applicable)

— Each rule is evaluated in the order in which it is listed in the policy. For a particular rule, if the target matches and the condition evaluates to Permit, Deny, or Indeterminate corresponding effect of the rule SHALL be the result of the evaluation of the policy. Otherwise, return to the next rule.

Attributes, Attribute Values, and Functions

As XACML is used in Attribute-Based Access Controlling, in XACML all the attributes are categorized into four main categories. But from XACML 3.0, custom categories are also supported.

Subject (urn:oasis:names:tc:xacml:3.0:attribute-category:subject)

Resource (urn:oasis:names:tc:xacml:3.0:attribute-category:resource)

Action (urn:oasis:names:tc:xacml:3.0:attribute-category:action)

Environment (urn:oasis:names:tc:xacml:3.0:attribute-category:environment)

A Policy resolves attribute values from the request or retrieves values from the PIP.

”AttributeDesignator”: lets the policy specify an attribute with a given name and type, and optionally an issuer as well. ”AttributeDesignator” can return multiple values. XACML provides a special attribute type called a “Bag”.

”AttributeValue”: contains a literal attribute value

XACML comes with a powerful system of functions. Functions can work on any combination of attribute values and can return any kind of attribute value supported in the system. Functions can also be nested, so we can have functions that consume the output of other functions, and this hierarchy can be arbitrarily complex.

”Apply”: denotes the application of a function to its arguments, thus encoding a function call.

1.4.4 MAC

Mandatory Access Control (MAC) is a type of access control mechanism in which the access to resources is strictly controlled by a central authority, such as the operating system or a security administrator. In a MAC system, the central authority determines the permissions for accessing each resource and assigns them to individual subjects (e.g., users, processes) based on their security clearance and the sensitivity of the resource.

In contrast to Discretionary Access Control (DAC), which allows the owner of a resource to control access to it, MAC systems provide a more centralized and secure approach to access control. This makes MAC particularly useful in high-security environments, such as military organizations, where there is a need to enforce strict security policies and protect sensitive information.

In MAC systems, the security policies are defined and enforced by the central authority and cannot be overridden by individual users or processes. Access to resources is based on the security classification assigned to the resource and the security clearance of the subject attempting to access it.

Examples of MAC systems include the Security-Enhanced Linux (SELinux) and the BSD MAC Framework. These systems use mandatory policies and access controls to enforce security policies, prevent unauthorized access to sensitive data, and limit the damage that can be done by malicious software.

Overall, MAC is a powerful and flexible tool for securing systems and controlling access to sensitive resources. However, it can also be complex and time-consuming to implement and manage, particularly in large organizations with many users and resources.

1.4.5 DAC

Discretionary access control is the principle of restricting access to objects based on the identity of the subject (the user or the group to which the user belongs). Discretionary access control is implemented using access control lists. A resource profile contains an access control list that identifies the users who can access the resource and the authority (such as read or update) the user is allowed in referencing the resource. The security administrator defines a profile for each object (a resource or group of resources), and updates the access control list for the profile. This type of control is discretionary in the sense that subjects can manipulate it, because the owner of a resource, in addition to the security administrator, can identify who can access the resource and with what authority.

DAC is easy to implement and intuitive but has certain disadvantages, including inherent vulnerabilities (Trojan horse), ACL maintenance or capability, grant and revoke permissions maintenance, and limited negative authorization power.

DAC attributes include:

• User may transfer object ownership to another user(s).

• User may determine the access type of other users.

• After several attempts, authorization failures restrict user access.

• Unauthorized users are blind to object characteristics, such as file size, file name and directory path.

• Object access is determined during access control list (ACL) authorization and based on user identification and/or group membership.

2 Language Overview

2.1 Computational Model

At a basic level, the computation that the DSL performs involves selecting, filtering, and transforming data with permission from the administrator. Basically, it performs the following computations:

1. SELECT: The SELECT statement is used to specify the columns to retrieve from one or more tables.

2. FROM: The FROM clause is used to specify the table(s) from which to retrieve data.

3. WHERE: The WHERE clause is used to filter the data based on specific criteria, such as values in certain columns or relationships between tables.

4. JOIN: The JOIN clause is used to combine data from multiple tables based on a specified relationship.

5. GROUP BY: The GROUP BY clause is used to group data based on one or more columns.

6. HAVING: The HAVING clause is used to filter the grouped data based on specific criteria.

7. ORDER BY: The ORDER BY clause is used to sort the data based on one or more columns.

Overall, the DSL provides a secured way to manage and manipulate data, making it a good fit for companies that are willed to restrict data access to some workers.

2.2 Data Structures

The DSL do not rely on the traditional tabular, relational model of storing data that SQL databases use. Some of the basic data structures in the DSL include:

1. Key-value stores: Key-value stores are the simplest type of non-relational database. Each piece of data is stored as a key-value pair, where the key is a unique identifier and the value can be any type of data, including structured, unstructured, or binary data.

2. Document-oriented databases: Document-oriented databases store data in the form of documents, which can contain a wide variety of structured and unstructured data. Each document is self-contained, and may be represented in formats such as JSON or BSON.

Non-relational databases also offer a variety of advanced features, such as automatic partitioning and replication, distributed processing, and high scalability, making them particularly well-suited for handling large volumes of unstructured or semi-structured data.

2.3 Error Handling

Programs can go wrong in many ways, including:

1. Syntax errors: Syntax errors occur when the code is written in a way that the programming language doesn't understand. These types of errors are usually caught by the compiler or interpreter, and

the language can communicate the error by indicating the line and column number where the error occurred, as well as providing a message that explains the issue.

2. Runtime errors: Runtime errors occur when a program is executing and encounters an issue that prevents it from running correctly. These can include issues like null pointer dereferences, divide-by-zero errors, or out-of-bounds array access. The language can communicate these errors to the user through error messages, which may include information about the line number where the error occurred, the type of error that was encountered, and suggestions for how to fix the issue.

3. Logical errors: Logical errors occur when the code is syntactically correct and runs without errors, but the output or behavior of the program is incorrect. These errors can be harder to detect, and may require debugging or code analysis to identify and fix. The language may not be able to communicate these errors directly to the user, but may provide tools like debuggers, profilers, or testing frameworks to help developers identify and resolve these types of errors.

Programming languages can communicate errors to the user in a variety of ways, including:

1. Error messages: Most programming languages provide error messages that are displayed when an error occurs. These messages typically include a description of the error, along with information about the location in the code where the error occurred.

2. Exceptions: Some languages use exceptions to handle errors. When an error occurs, the language raises an exception, which can be caught by the program and handled in a way that is appropriate for the situation.

3. Return codes: Some languages use return codes to indicate success or failure of a function or program. A value of zero may indicate success, while non-zero values indicate errors.

4. Logging: Some languages provide logging frameworks that can be used to log error messages, warnings, and other types of messages that can help developers diagnose and resolve issues.

One example of a DSL for error handling is the Assertive Programming Language (APL). APL is a domain-specific language that is specifically designed for writing assertions, which are statements that check the correctness of a program's behavior. APL allows developers to write concise and expressive assertions, which can help to catch errors early in the development process. APL also provides clear and informative error messages when an assertion fails, which can help developers quickly diagnose and fix problems.

Another example of a DSL for error handling is the Error Model and Notation (EMN) language. EMN is a domain-specific language for modeling and analyzing errors in software systems. It provides a set of graphical notations that can be used to represent the different types of errors that can occur in a software system, as well as a set of analysis tools that can be used to identify and diagnose these errors.

Compared to these DSLs, the approach taken by mainstream programming languages such as Python,

Java, and C++ for error handling may differ in terms of syntax, but they provide similar functionality, including error messages, exceptions, return codes, and logging. The key advantage of these mainstream languages is that they are widely used, well-documented, and have large communities, making it easier for developers to find help and resources when dealing with errors.

2.4 Implementation plan

Define the scope and objectives of the project: This includes determining the target audience, the purpose of the language, and the features it will offer.

Choose the programming language: The team should choose a programming language that is suitable for the project and can deliver the desired features.

Plan the development phases: The team should plan and divide the project into different development phases such as design, implementation, testing, and deployment.

Build a development environment: Set up an environment that supports the development process, including a code repository, issue tracking, and testing environment.

Develop and test the language: The team should develop the language's core features and test it for functionality and performance.

Deploy and maintain: After successful testing, the language can be deployed and maintained to ensure that it remains functional and updated.

Teamwork Plan:

Discuss and plan the project: The team should meet and discuss the project's scope, objectives, and features to ensure everyone understands the project's vision.

Divide the tasks: The team should divide the project into different development phases and assign specific tasks to each member. Every team member should work on every aspect of the project, including design, implementation, evaluation, and documentation.

Collaborate and communicate: The team should collaborate and communicate regularly to ensure that everyone is on the same page, share feedback, and address any challenges or issues that arise.

Review and test: After completing their assigned tasks, each team member should review and test their work and provide feedback to other team members.

Document and report: The team should document and report on the project's progress, including the tasks completed, the challenges faced, and the solutions implemented. This documentation will help future team members or stakeholders to understand the project and its implementation.

Task description:

1. Design The design of a new programming language involves several important considerations. Here are some key design considerations that need to be taken into account:

Purpose: First, you need to determine the purpose of the language. Is it intended for general-purpose

programming or for a specific domain or application? The design of the language should align with its intended use.

Syntax: The syntax of the language is critical in making it easy to learn and use. The syntax should be clear, concise, and easy to read and write.

Data Types: The data types that the language supports must be well defined and appropriate for the purpose of the language. This includes fundamental types such as integers, floating-point numbers, and characters, as well as more complex data types such as arrays, structures, and objects.

Expressiveness: The language should be expressive enough to allow developers to write code in a concise and understandable way. This includes support for abstractions, lambdas, closures, and other high-level programming constructs.

Tooling: Good tooling is critical for a successful language. This includes compilers, debuggers, and other development tools that make it easy to write, debug, and deploy code in the language.

Performance: The language's performance is another important consideration. The language should be designed to be fast and efficient, both in terms of runtime performance and memory usage.

Backward Compatibility: If the language is intended for long-term use, backward compatibility is important. This means that the language must be designed in a way that makes it possible to maintain compatibility with older versions of the language.

Overall, designing a new programming language is a complex process that requires careful consideration of many factors. The language designer must balance the needs of developers with the technical constraints of the system to create a language that is expressive, efficient, and easy to use.

2. Implementation

Once the design of a new programming language has been defined, the implementation part involves the creation of the language itself, including the development of a compiler or interpreter that can transform code written in the new language into executable machine code.

Here are some key steps involved in the implementation of a new programming language:

Lexer and Parser: The first step is to implement a lexer and parser that can take the language's source code and break it down into individual tokens and grammar rules. This creates an Abstract Syntax Tree (AST) that represents the program's structure.

Semantic Analysis: The next step is to perform semantic analysis, which involves analyzing the AST and applying the language's rules and constraints to ensure that the code is syntactically and semantically correct. This includes type checking, scope analysis, and other checks.

Code Generation: The final step is to generate executable code from the AST. This can involve the creation of an intermediate representation (IR) that can be optimized and transformed into machine code.

Tooling and Infrastructure: Once the core language implementation is complete, it's important to

develop tooling and infrastructure to support the language. This includes development tools such as IDEs, debuggers, and profilers, as well as documentation, tutorials, and a community of users and contributors.

Testing and Debugging: Finally, thorough testing and debugging are critical to ensure that the language implementation is correct and performs well. This involves writing and executing test cases, debugging issues that arise, and optimizing the language implementation for performance.

Overall, implementing a new programming language is a complex process that requires a deep understanding of computer science and programming languages. It can be a challenging and rewarding task that requires a lot of hard work and dedication.

3. Evaluation

The evaluation part of creating a new programming language involves assessing the language's performance, usability, and overall effectiveness in meeting its intended goals. Here are some key factors that are typically evaluated:

Performance: The language's runtime performance is a critical factor in determining its effectiveness. This includes both the speed of executing code and the memory usage of the language.

Usability: The language's usability is another important factor to consider. This includes the ease of learning and using the language, as well as the quality of its development tools and documentation.

Expressiveness: The language's expressiveness is a measure of its ability to allow developers to write code that is concise, readable, and maintainable. This includes support for high-level programming constructs such as closures, lambda functions, and other abstractions.

Backward Compatibility: If the language is intended for long-term use, backward compatibility is important. This means that the language must be designed in a way that makes it possible to maintain compatibility with older versions of the language.

Adoption: The language's adoption by developers and organizations is another important factor. This includes the number of developers using the language, the number of libraries and frameworks available for the language, and the overall ecosystem around the language.

Feedback: Gathering feedback from developers and users of the language is critical in understanding how well the language is meeting its intended goals. This can include feedback on usability, performance, and overall effectiveness.

Overall, evaluating a new programming language requires careful consideration of many factors, and it may take time to determine whether the language is meeting its intended goals. By gathering feedback and assessing the language's performance, usability, and effectiveness, language designers can improve the language and make it more useful for developers.

4. Documentation

The documentation part of creating a new programming language involves creating high-quality

documentation that helps developers understand how to use the language and its associated tools effectively. Good documentation can help developers learn the language quickly, avoid common mistakes, and write code that is easy to read and maintain. Here are some key aspects of documenting a new programming language:

Language Reference: A language reference is a comprehensive guide that explains the syntax and semantics of the language. It should cover all the language's features and provide clear examples of how to use them. This can include details about the language's keywords, data types, and control structures.

Tutorials and Guides: Tutorials and guides are a great way to help developers get started with the language quickly. They should provide step-by-step instructions for performing common tasks in the language, and should be written in a clear and easy-to-understand style.

API Documentation: If the language includes an API or library, it's important to document the API's functions, parameters, and return values. This can include code examples that demonstrate how to use the API in various scenarios.

Examples and Best Practices: Providing examples of good programming practices and common pitfalls can help developers write better code in the language. This can include guidelines on structuring code, naming conventions, and best practices for error handling.

Tooling and Environment: Documentation should also cover any tools or development environments associated with the language. This can include instructions for setting up and configuring development environments, as well as guides on using tools such as debuggers, profilers, and performance analysers.

Community Resources: Finally, it's important to provide links to community resources such as forums, mailing lists, and online communities. This can help developers get help and support from other users of the language, and can foster a sense of community around the language.

Overall, creating good documentation is an essential part of creating a new programming language. By providing clear, comprehensive, and accessible documentation, language designers can help developers learn the language quickly and use it effectively.

Team tasks:

Every team member will participate at meetings and discuss the objectives and their opinion regarding project's vision.

Design:

1. Tudor - purpose, syntax, data types, expressiveness
2. Cristina - purpose, syntax, expressiveness
3. Leonid - purpose, syntax, data types, tooling, performance
4. Dan - purpose, syntax, expressiveness
5. Andrei - purpose, syntax, tooling, performance

Implementation:

1. Tudor – parser, semantic analysis, testing and debugging
2. Cristina – semantic analysis, testing and debugging
3. Leonid - parser, testing and debugging
4. Dan - code generator, testing and debugging
5. Andrei – code generator, testing and debugging

Evaluation:

1. Tudor – performance, usability
2. Cristina – usability
3. Leonid – expressiveness
4. Dan – performance
5. Andrei – expressiveness

Documentation:

All – language reference, examples, tooling

Every member will participate in the report writing, as everyone will have work on every part of the project.

3 Grammar

Grammar

For a better understanding, further is represented the grammar for the Query language according to a very simple and textual program. The language permits the user to access, modify, insert and delete elements from a non-relational database. Next, is shown in detail each feature of grammar.

The DSL design includes several stages. First of all, the definition of the programming language grammar is G = (Vn, Vt, P, S):

- Vn – is a finite set of non-terminal symbols;
- Vt - is a finite set of terminal symbols.
- P – is a finite set of the production of rules;
- S - is the start symbol;

In Table 1 are meta-notations used for specifying the grammar.

Table 1

Meta notation

<query> -> <selectquery> | <generatekeyquery> | <createuserquery> |
<loginquery> | <createquery> | <insertquery> | <updatequery> | <deletequery>
| <help>
<selectquery> -> <select_clause> <from_clause> <where_clause>?
<order_by_clause> ?
<select_clause> -> ”SELECT” <field_name> ”,” <field_name>* | ”*”
<from_clause> -> ”FROM” <collection_name>
<where_clause> -> ”WHERE” <condition>
<order_by_clause> -> ”ORDER BY” <field_name> (”ASC” | ”DESC”)?
<generatekeyquery> -> ”generate_key” <privilege> <privilege> <privilege>
<privilege> <time>?

Notation (symbol)	Meaning
<**foo**>	means foo is a nonterminal
”foo” — ’f’	means that foo and f are terminal i.e.,a token or a part of a token
x?	means zero or one occurrence of x, i.e., x is optional
x*	means zero or more occurrences of x
x+	one or more occurrences of x
{}	large braces are used for grouping
—	separates alternatives

<privilege> -> <boolean>

<time> -> <int>

<createuserquery> -> ”create_user” <username> <password> <mail> <token>

<mail> -> <string> ”@” <string> ”.” <string>

<token> -> <string>

<loginquery> -> ”log_in” <username> <password>

<createquery> -> <create_clause> <codeblock>

<create_clause> -> ”CREATE TABLE” <collection_name>

<codeblock> -> ”(” <table_fields> ”)”

<table_fields> -> <collection_collumn> ”,” <collection_collumn>*

<collection_collumn> -> <type> <field_name>

<insertquery> -> <insert_clause> <values_clause><insert_clause> -> ”INSERT INTO” <collection_n

<values_clause> -> ”VALUES” ”(” <value> ”,” <value>* ”)”

<updatequery> -> <update_clause> <where_clause> <set_clause>

<update_clause> -> ”UPDATE” <collection_name>

<set_clause> -> ”SET” <field_name> ”=” <variable>

<deletequery> -> <delete_clause> <where_clause>?

<delete_clause> -> ”DELETE FROM” <collection_name>

<help> -> ”HELP”

<username> -> <string>

<password> -> <string>

<condition> -> <comparison> | <logical_op> | ”(” <condition> ”)” |

<comparison> -> <variable> <comparison_op> <variable>

<variable> -> <field_name> | <value>

<logical_op> -> <condition> <logical_op> <condition>

<comparison_op> -> ”==” | ”!=” | ”>” | ”>=” | ”<” | ”<=”

<field_name> -> <string>

<collection_name> -> <string>

<value> -> <string> | <number> | <NULL>

<NULL> -> ’0’ | ”

<type> -> <string> | <int> | <float> | <char> | <digit> | <boolean>

<string> -> ”” <char>* ””

<int> -> <digit>+

<float> -> <digit>+ ”.” <digit>+

<char> -> ’a’ | ’b’ | . . . | ’z’ | ’A’ | ’B’ | . . . | ’Z’

<digit> -> ’0’ | ’1’ | ... | ’9’

<boolean> -> ’True’ | ’1’ | ’False’ | ’0’

Parse Tree

Statement example:

SELECT * FROM foo WHERE id=prev_id ORDER BY id

Figure 1. Parse tree for the statement example

Conclusions

Here go your conclusions..

Bibliography

[1] Fowler, M. (2010). Domain-specific languages. Addison-Wesley Professional Accessed February 14, 2023. https://freecontent.manning.com/the-what-and-why-of-domain-specific-lanugages/.

[2]