RASSP Information Management Appnote

Abstract

In an integrated product development environment which involves several vendor tools, diverse and incompatible configuration management mechanisms and authorization management mechanisms across tools can lead to the inefficiencies in the design process. We describe in this document a common model for configuration management and authorization management that may be adopted by an integrated product development environment. We have specified a common minimal set of configuration management mechanisms that need to be provided by the tools to support the proposed configuration and authorization models. We have completed a pilot implementation of the Configuration management (CM) and Authorization Management (AM) models for the Intergraph Asset and Information Management (AIM) software system. This pilot implementation proves the plausibility and usability of the model.

The RASSP configuration management model supports three types of workspaces -- private, shared, and global. The workspaces are organized hierarchically with global workspaces as the root of the hierarchy, shared workspaces as children of the global workspace, and private workspaces as children of the shared workspaces. The CM model supports three types of versions of design objects -- transient, working, and released. Transient design objects are created in the private workspaces, and when mature for sharing with the other members of the team, are promoted to working versions and checked in to a shared workspace. Once the working version has been thoroughly tested it is them promoted to a released version and checked into the global workspace.

The RASSP authorization management model supports three types of hierarchies -- an authorization object hierarchy, an authorization type hierarchy, and an authorization role hierarchy. The authorization object hierarchy is a composition tree of all the design objects in a project. The authorization type hierarchy is an implication tree, with a node of a certain access privilege implying a set of access privileges below it in the inverted tree. The authorization role hierarchy is also an implication tree with a node representing a certain user role subsuming all the access privileges of the user roles below it in the hierarchy. This hierarchical approach allows an administrator to specify authorizations in a succinct fashion by creating a three-way relationship between a node from each of the three hierarchies.

Purpose

The RASSP tools for information management were used for the RASSP benchmark programs at the Lockheed Martin Advanced Technology Labs. The tools supported a design program involving about ten engineers, and spanned about two years in duration. The metrics we have collected on the design process quantifies the benefits of the RASSP CM and AM tools.

This application note is intended for managers and administrators of design groups who are interested in streamlining the process of building, releasing, and supporting product designs.

Roadmap

RASSP Information Management Appnote

1.0 Introduction

Two critical aspects of information management within a computer-aided design environment are the configuration management and authorization management of the design data. Configuration Management (CM) is the management of the versioning of design objects. It includes creating, approving and releasing a new version of a design object; organizing the versions of a design object; and assembling compatible configurations of versions of design objects to form a release of a product. Authorization Management (AM) is the management of the granting and revoking of access to the design objects for the various users of the system. It involves specifying the types of operations that may be performed on data objects, partitioning the design object space at the appropriate granularity for specifying access controls, and partitioning the users of the system into the various roles for specifying access privileges. Different CAD vendor tools provide different mechanisms to support configuration management and authorization management. A mechanism is an individual function of a system. The mechanisms provided by the tools not only vary in scope, but also in their semantics. In an integrated product development environment (IPDE) which involves several vendor tools, such as RASSP, diverse and incompatible configuration management mechanisms and authorization management mechanisms across tools can lead to the inefficiencies in the design process listed below:

- There is no common way of handling the CM and AM of a product throughout its life cycle
- The design engineers working on a project have to learn several different paradigms for CM and AM
- The CM data on a product generated by one tool cannot be used by another tool
- The authorization information cannot be shared among the various tools used in the CAD environment. Authorizations have to be specified separately in the various tools and the management of the consistency of the authorization information among the tools is cumbersome.

We describe in this document a common model of configuration management and authorization management that may be adopted by an integrated product development environment. We have specified a common minimal set of mechanisms that need to be provided by the tools to support the proposed configuration and authorization models. An important criterion we CM and AM models have been implemented as part of the design environment developed for the Rapid Prototyping of Application-Specific Signal Processors (RASSP) program.

The Rapid Prototyping of Application-Specific Signal Processors (RASSP) is a Defense Advanced Research Projects Agency (DARPA)/Tri-Service program aimed at dramatically improving the process of design, manufacture, test and procurement of digital signal processors. The RASSP program will deliver an integrated system called the RASSP system, which integrates the CAD tools used in the RASSP design process under a framework referred to as the enterprise framework. An enterprise framework provides the facilities and services necessary to integrate the automated processes of an enterprise. In the RASSP system the enterprise framework provides support for workflow management, design data management and reuse data management. The workflow management subsystem of the RASSP enterprise system enables a RASSP system administrator to model and enforce a particular design methodology for a project. The data management subsystem of the enterprise framework provides facilities for configuration managing, and controlling access to design data files that may reside at various sites in a computer network. The reuse data management subsystem provides facilities for cataloging, classifying and storing reusable design components; as well as mechanisms for searching for reusable components.

In the next section we present the RASSP CM model, and in section 3 we present the RASSP AM model. In section 4 we present a pilot implementation of the models, and the lessons learned from the implementation.

The application of the CM and AM tools in a real world design project is the subject of section 5. In section 6 we compare the RASSP CM and AM facilities with an out-of-the-box product data management system.

 Dennis Basara

RASSP Information Management Appnote

2.0 The RASSP Configuration Management Model

2.1 Workspaces and Configurations

The RASSP Configuration Management (CM) model is based upon the concept of workspaces, and the concept of configurations. *Workspaces* are partitions of the design object space to allow designers working on the various parts of a project to selectively make their design objects visible to others in the project [Cattell_1991]. Workspaces are organized in a hierarchical fashion as shown in figure 2 - 1, with a *global workspace* at the root of the hierarchy, *shared workspaces* as the intermediate nodes in the hierarchy, and *private workspaces* as the leaves in the hierarchy. The links in a workspace hierarchy represent a *parent-child relationship* between the linked workspaces. For example, in figure 2 the workspace "Private WS2" is a child of the workspace "Shared WS1" (and the workspace "Shared WS1" is the parent of the workspace "Private WS2"). Workspaces provide for varying levels of sharing of data objects. A user of a workspace has visibility to all the objects residing in the workspace and the objects residing in the ancestor workspaces of the workspace. Thus all users of the database have visibility to data objects residing in the global workspace.

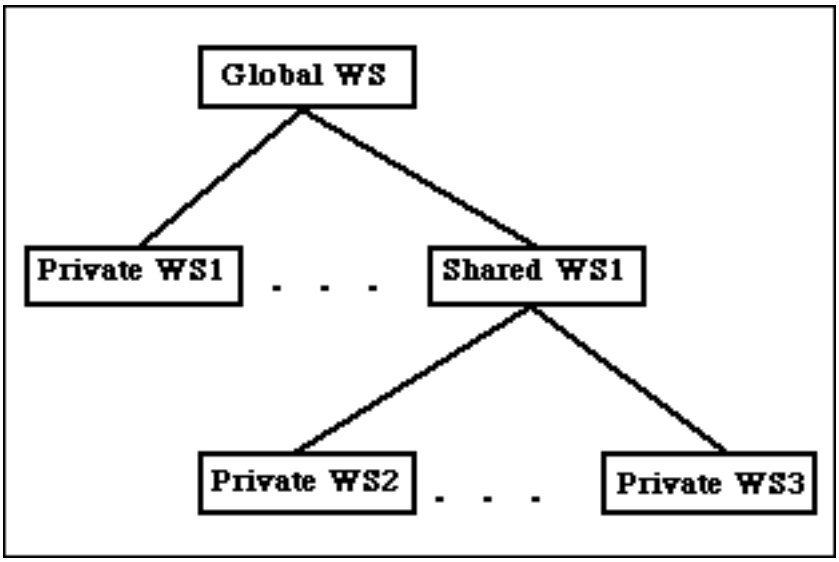

Figure 2 - 1: A Workspace Hierarchy

Configurations are related data objects that evolve at the same time and are grouped together for the purpose of versioning. New configuration objects are typically created in a private workspace, at which point the configuration is considered a *transient version* of the configuration. A transient version may be updated or deleted. Once the transient version of a configuration reaches a state of maturity suitable for sharing with other designers in a project, it is promoted to a *working version* of the configuration, by checking in the configuration from the private workspace where it resides, to its parent workspace. A working version may not be updated but may be deleted. Working versions of configurations that are considered to represent the final state of design are promoted to *released versions* by checking in them to the global workspace. A released version may not be updated nor deleted. We use the notation statei > statej to denote that statei is a higher state that statej. Thus *released* > *working* > *transient* .

A new transient version cj of a configuration c, may be created by checking out an existing version ci residing in a workspace wm to a workspace wn that is a direct or indirect descendent of wm. The source version ci may be in one of the three states prior to checkout -- released, working or transient. If the state of ci prior to checkout is released or working, the state of ci remains unchanged after checkout. However, if the state of ci is transient prior to the checkout, ci is promoted by the system to a working version as part of the checkout operation.

The versions of a configuration are organized as a directed acyclic graph (DAG), as shown in figure 2 - 2, and is commonly referred to as a version tree. The new nodes in the tree start out as a transient version and progressively become working versions and then released versions. A version in a version tree may be deleted only if it is a transient or a working version, and it is a leaf node in the tree. A directed link between two versions i and j in a version tree represents the *is-derived-from* relationship, i.e., version j is derived from version i. Also, version i is said to be the *parent* of version j, and version j is said to be a *child* of version i. We use the notation ci -> cj to represent the parent-child relationship between the two versions i and j of the configuration c.

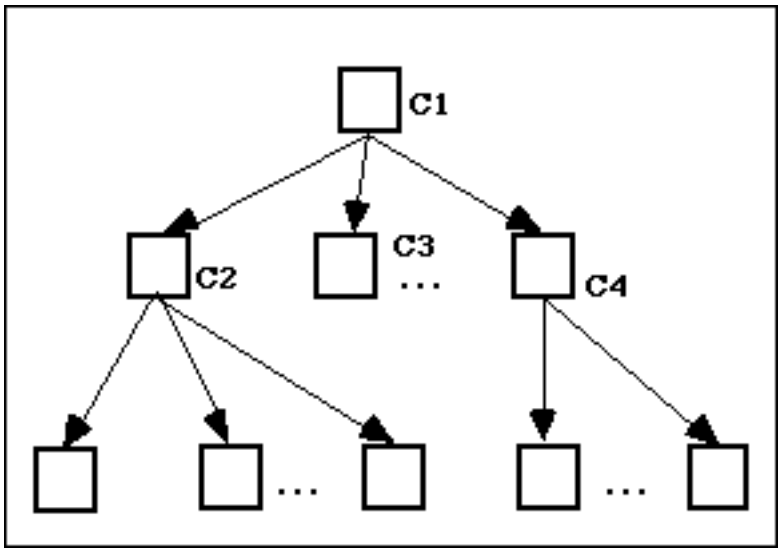

Figure 2 - 2: A Version Tree

The following rule applies to a version tree:
VT-Rule1: Given two version ci and cj of a configuration c, such that ci -> cj, then the state of cj should be less than or equal to the state of ci.

2.2 The RASSP CM Mechanisms

We describe in this section a minimal set of CM mechanisms that needs to be supported to implement the RASSP CM model. This set will provide the basis for configuration management of design data in the different phases of the RASSP design process. The set of CM mechanisms listed here is intended to serve as a common minimal set; an individual CAD tool may support more than the minimal set proposed here.

We categorize the CM mechanisms into two classes -- workspace functions and version management functions. In the following sections we use a C-like pseudo code to describe the functions. For example,

```
Bar my_func (Foo a_foo);
```

describes a function "my_func" that takes as a parameter an object of type "Foo" and returns an object of type "Bar". We use names starting with capital letters, such as "Workspace" to denote the type of object, and names starting with small letters, such as "create_workspace" and "parent_workspace" to denote a function

name or a parameter name.

```
Bar another_func (foo *a_foo=0);
```

describes a function "another_func" that takes as a parameter a pointer (denoted by the *) to an object of type "foo&334. The "= 0 " is a default value for the parameter if one is not provided. Thus the parameter "a_foo" is an optional parameter for the function "another_func", while it is a required parameter for the function "my_func".

The C-like style we are using to describe the mechanisms is for the brevity of the descriptions, and does not have any implications as to the implementations of these functions, nor the user interface provided by the systems to these functions.

2.2.1 Workspace Functions

2.2.1.1 Creating a workspace

```
Workspace *create_workspace (char *workspace_name,
  Workspace *parent_workspace=0);
```

Creates a child workspace of the specified parent workspace, and assigns it the supplied name. If a parent workspace is not specified, the new workspace is made a child of the global workspace. The global workspace is a system-created workspace that exists in every database, and has the name "global_workspace" assigned to it.

2.2.2 Accessing an arbitrary workspace

```
Workspace *get_workspace (char *workspace_name);
```

Returns a pointer to the workspace with the specified name.

2.2.3 Accessing child workspaces

```
Workspace_List child_workspaces (Workspace *a_workspace);
```

Returns a list of pointers to child workspaces of the specified workspace.

2.2.4 Accessing the parent workspace

```
Workspace *parent_workspace (Workspace *a_workspace);
```

Returns a pointer to the parent workspace of the specified workspace.

2.2.5 Making a workspace visible

```
void set_current_workspace (Workspace *a_workspace);
```

Sets the specified workspace to be the current workspace for the application program. The application program can access only objects that reside in the current workspace, or in the ancestor workspaces of the current workspace.

2.3 Version Management Functions

2.3.1 Creating a configuration

```
Configuration *create_configuration ();
```

Creates a new configuration and returns a pointer to it.

2.3.2 Inserting data objects into a configuration

```
void insert_into_configuration (Configuration *a_configuration,
  void *a_design_object);
```

Inserts the design object pointed to by "a_design_object" into the specified configuration.

2.3.3 Checkout

```
Configuration *checkout (Configuration *a_configuration,
  char *version_name=0);
```

Creates a new version of the configuration pointed to by "a_configuration", and makes the new version visible in the current workspace. The new version may also be provided an optional version name. The version name is an arbitrary name provided by the user. If the configuration pointed to by a_configuration is a transient version, it is promoted by the system to a working version, by performing a checkin operation (see section 2.3.4), before performing the checkout. The configuration pointed to by "a_configuration" may reside in the current workspace, or in any of the ancestor workspaces of the current workspace. This function returns a pointer to the newly created version of the configuration.

2.3.4 Checkin

```
void checkin (Configuration *a_configuration);
```

Checks in the configuration pointed to by "a_configuration". This results in the configuration being made visible to the parent workspace of the current workspace. If the parent workspace is the global workspace, then the configuration is promoted to a released version, otherwise it is promoted to a working version. A working version of a configuration that is checked in to a non global workspace is not promoted in the process of checkin, but is made visible to the parent workspace.

2.3.5 Accessing child versions

```
Configuration_List child_versions
  (Configuration *a_configuration);
```

Returns a list of pointers to the child versions of the configuration pointed to by a_configuration.

2.3.6 Accessing the parent version

```
Configuration *parent_version (Configuration *a_configuration);
```

Returns a pointer to the parent version of the configuration pointed to by "a_configuration".

2.3.7 Naming versions

```
void name_version(Configuration *a_configuration, char *a_name);
```

Assigns the character string pointed to by "a_name" as the name of the specified configuration.

2.3.8 Retrieving a named version

```
Configuration *get_named_version     (Configuration *a_configuration, char *a_name);
```

Returns the version of the specified configuration that has the name pointed to by "a_name". If no such version exists a null pointer is returned.

RASSP Information Management Appnote

3.0 The RASSP Authorization Model

3.1 Authorization Objects, Types, and Roles

An authorization is a triplet {*oi, rj, tk* } where *oi* is an authorization object in an authorization object hierarchy, *rj* is a authorization role in an authorization role hierarchy, and *tk* is an authorization type in an authorization type hierarchy [Rabbitti, 1991]. An *authorization object* is a data object on which an authorization may be specified. Authorization objects in a database are organized as a directed acyclic graph as shown in figure 3 - 1. An *authorization role* is a collection of users that have the same set of authorizations on the same set of objects. The authorization roles in an organization are also organized as a directed acyclic graph as shown in figure 3 - 2. An *authorization type* is a type of operation that may be performed on a data object. The authorization types for a database are also organized as directed acyclic graphs as shown in figures 3 - 3 and 3 - 4. In figure 3 - 3 the "Grant" authorizations are authorizations to grant an authorization to another role in the role hierarchy. The authorization type hierarchy for projects also contains the "Grant" authorizations, but are not shown in figure 3 - 4 to avoid cluttering the figure. The directed links between two nodes in a hierarchy represent an *implication relationship* between the nodes. For example, an authorization for the engineering manager to update design data, implies the authorization to update system definition data, architecture data, etc., as they follow design data in the authorization object hierarchy. The authorization implies the same authorization for the project manager also, as the engineering manager follows project manager in the authorization role hierarchy. Also, since the operation read follows the operation update in the authorization type hierarchy for data objects, the authorization to update design data implies an authorization to read design data as well.

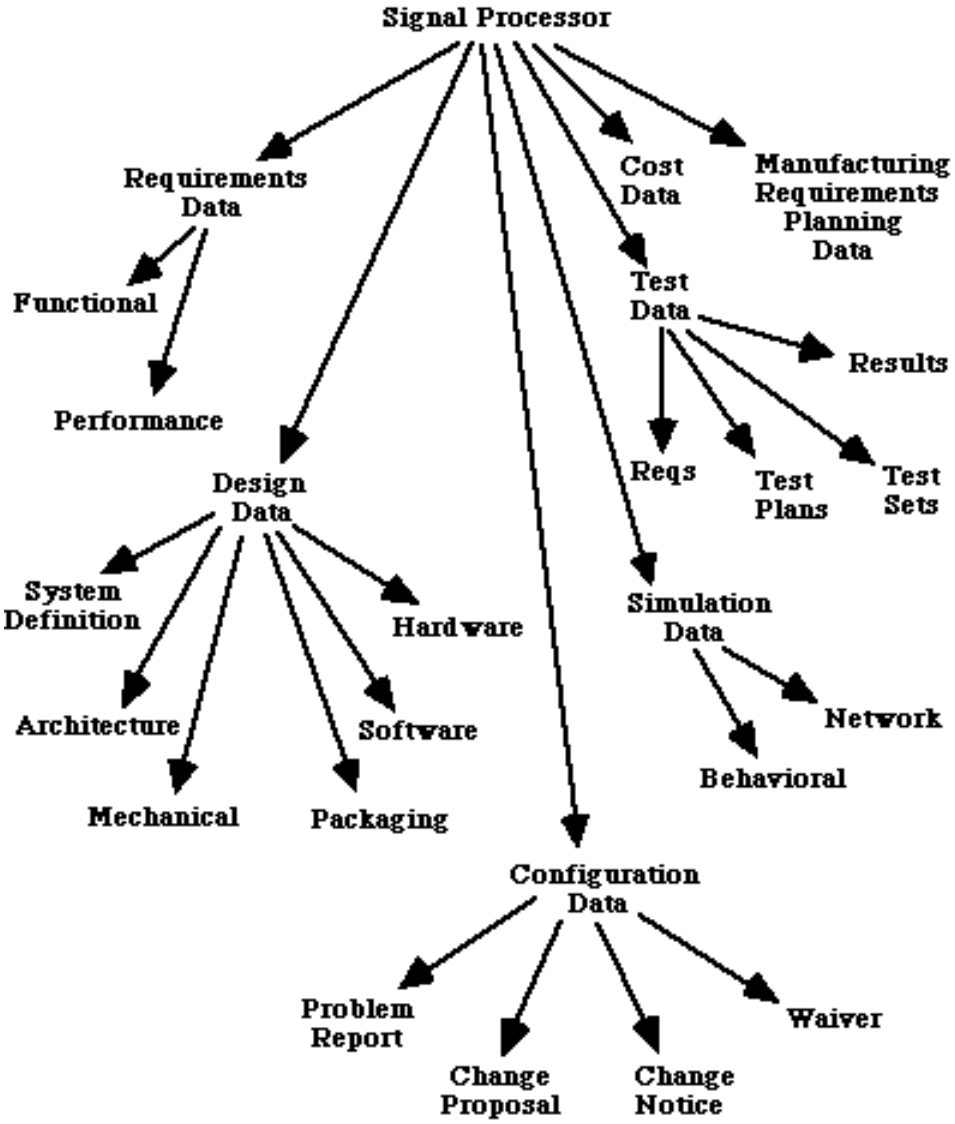

Figure 3 - 1: An Example Authorization Object Hierarchy

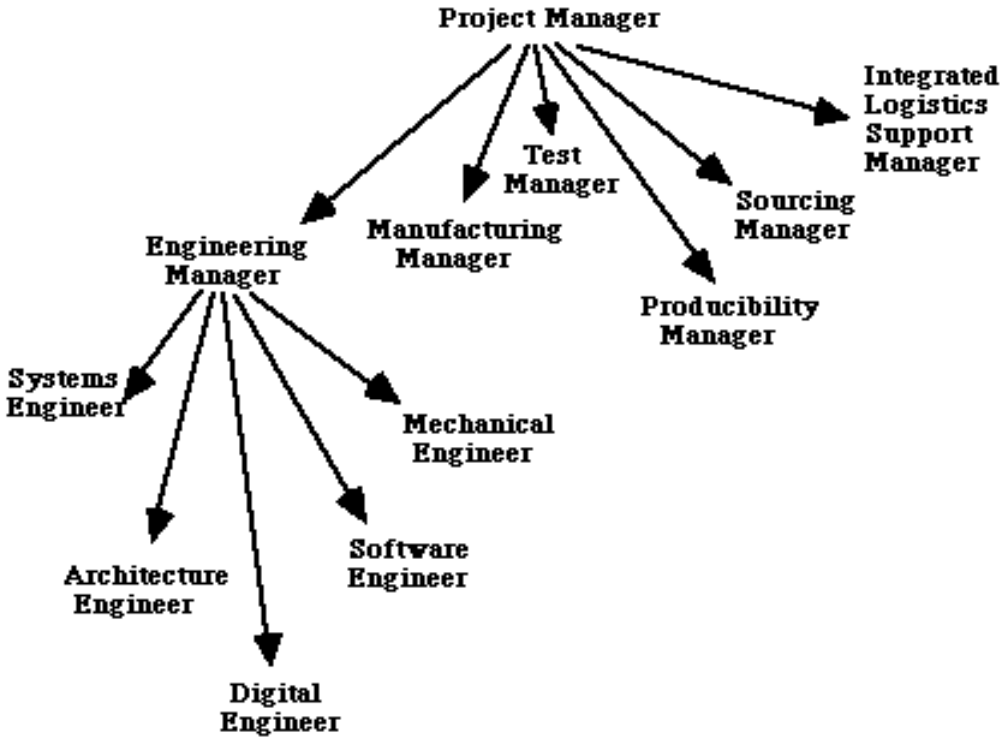

Figure 3 - 2: An Example Authorization Role Hierarchy

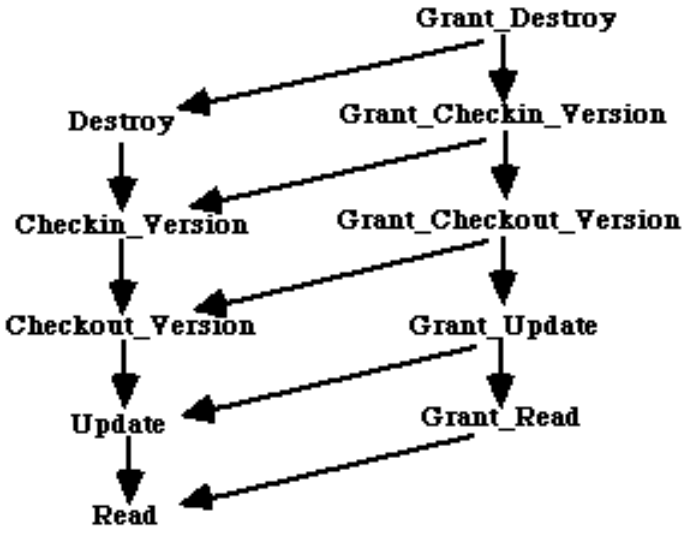

Figure 3 - 3: The Authorization Type Hierarchy for Design Objects

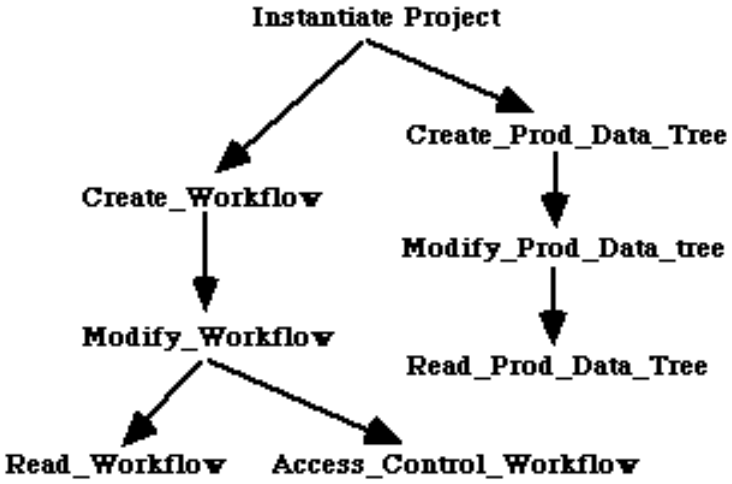

Figure 3 - 4: The Authorization Type Hierarchy for Projects

An authorization may be *positive* , granting an authorization, or *negative* , revoking an authorization. An explicit or an implicit positive authorization {*oi, rj, tk* } has to exist for an operation of type *tk* to be performed by a user belonging to role *rj* on a data object belonging to the authorization object *oi* . A positive (or negative) authorization specified on a node *ni* in an authorization hierarchy may be overridden by a negative (or positive) authorization on a node *nj* that follows *ni* in the authorization hierarchy. For example, a positive authorization for a particular role to update design data (in the example authorization object hierarchy in figure 3 - 1) may be overridden by a negative authorization for the same role to update mechanical design data. Similarly, a positive authorization for the engineering manager (in the example authorization role hierarchy in figure 3 - 2) to update configuration data (in the example authorization object hierarchy in figure 3 - 1) may be overridden by a negative authorization to update waiver data.

The authorization object hierarchy and the authorization role hierarchy for a project may be customized by a RASSP systems administrator. The authorization type hierarchies, however, are not customizable by a RASSP systems administrator. Each authorization type has an associated operation such as checkin, checkout, update etc. Thus, defining a new authorization type will typically involve adding a new functionality to the system.

3.2 Mechanisms to support the RASSP Authorization Model

We describe in this section four sets of mechanisms the data management subsystem of a CAD environment needs to provide in order to support the RASSP authorization model.

3.2.1 Mechanisms to manipulate the authorization object hierarchy

3.2.1.1 Creating an authorization object

```
Authorization_Object *create_authorization_object
  (Authorization_Object *parent=0, char *name);
```

Creates a new authorization object, as a child of the specified parent authorization object and assigns it the specified name. If no parent authorization object is specified then the root node of a new authorization object hierarchy is created.

3.2.1.2 Deleting an authorization object

```
void delete_authorization_object
  (Authorization_Object *an_authorization_object);
```

Deletes the subset of the authorization object hierarchy rooted at the specified authorization object, from the authorization object hierarchy.

3.2.1.3 Adding a child to an authorization object

```
void add_child (Authorization_Object *parent,
   Authorization_Object *child);
```

Adds the sub authorization object hierarchy rooted at the authorization object pointed to by "child" as a child of the specified parent authorization object. A particular sub authorization object hierarchy may be repeated at a number of nodes in an authorization object hierarchy.

3.2.1.4 Associating data files with authorization objects

```
void associate_data_file
  (Authorization_Object *an_authorization_object,
  char *file_path_name);
```

Associates an existing data file with a node in the authorization object hierarchy. A number of files may be associated with an authorization object. An authorization specified on an authorization object will apply to all the files associated with the authorization object.

3.2.1.5 Retrieving an authorization object

```
Authorization_Object *find_authorization_object
  (Authorization_Object *root=0, char *name);
```

Returns a pointer to an authorization object with the specified name, in the authorization type hierarchy pointed to by "root". If no root is specified, a pointer to a root of an authorization object hierarchy with the specified name is returned.

3.2.1.6 Retrieving the children of an authorization object

```
Authorization_Object_List get_children
  (Authorization_Object *an_authorization_object);
```

Returns a list of pointers to authorization objects that are children of the specified authorization object.

3.2.2 Mechanisms to manipulate the authorization role hierarchy

3.2.2.1 Creating an authorization role

```
Authorization_Role *create_authorization_role
  (Authorization_Role *parent=0, char *name);
```

Creates a new authorization role, as a child of the specified parent authorization role. If no parent authorization role is specified then the root node of a new authorization role hierarchy is created.

3.2.2.2 Deleting an authorization role

```
void delete_authorization_role
  (Authorization_Role *an_authorization_role);
```

Deletes the subset of the authorization role hierarchy rooted at the specified authorization role from the authorization role hierarchy.

3.2.2.3 Adding a child to an authorization role

```
void add_child (Authorization_Role *parent,
  Authorization_Role *child);
```

Adds the sub authorization role hierarchy rooted at the authorization role pointed to by "child" as a child of the specified parent authorization role. A particular sub authorization role hierarchy may be repeated at a number of nodes in the authorization role hierarchy.

3.2.2.4 Associating users with authorization roles

```
void associate_user
  (Authorization_Role *an_authorization_role,
  char *user_name);
```

Associates a user with an authorization role. A number of users may be associated with an authorization role. An authorization specified on an authorization role will apply to all the users associated with the authorization role.

3.2.2.5 Retrieving an authorization role

```
Authorization_Role *find_authorization_role
  (Authorization_Role *root=0, char *name);
```

Returns a pointer to an authorization role with the specified name, in the authorization type hierarchy pointed to by "root". If no root is specified, a pointer to a root of an authorization role hierarchy with the specified name is returned.

3.2.2.6 Retrieving the children of an authorization role

```
Authorization_Role_List get_children
  (Authorization_Role *an_authorization_role);
```

Returns a list of pointers to authorization roles that are children of the specified authorization role.

3.2.3 Mechanisms to manipulate the authorization type hierarchy

3.2.3.1 Retrieving an authorization type

```
Authorization_Type *find_authorization_type
  (Authorization_Type *root=0, char *name);
```

Returns a pointer to an authorization type with the specified name, in the authorization type hierarchy pointed to by "root". If no root is specified, a pointer to a root of an authorization type hierarchy with the specified name is returned.

3.2.3.2 Retrieving the children of a node in the authorization type hierarchy

```
Authorization_Type_List get_children
  (Authorization_Type *an_authorization_type);
```

Returns a list of pointers to authorization types that are children of the specified authorization type.

3.2.4 Mechanisms to grant and revoke authorizations

3.2.4.1 Granting authorizations

```
int grant_authorization
  (Authorization_Object *an_authorization_object,
  Authorization_Role *an_authorization_role,
  Authorization_Type *an_authorization_type);
```

Grants an authorization of type "an_authorization_type" on the object pointed to by "an_authorization_object" to the authorization role pointed to by "an_authorization_role".

3.2.4.2 Revoking authorizations

```
int revoke_authorization
  (Authorization_Object *an_authorization_object,
  Authorization_Role *an_authorization_role,
  Authorization_Type *an_authorization_type);
```

Revokes an authorization of type "an_authorization_type" on the object pointed to by "an_authorization_object" from the authorization role pointed to by "an_authorization_role".

Next Up Previous Contents

Next: 4 Implementation of the RASSP Configuration and Authorization Management Models **Up:** Appnotes Index
Previous:2 The RASSP Configuration Management Model

RASSP Information Management Appnote

4.0 Implementation of the RASSP Configuration and Authorization Management Models

A pilot implementation of the Configuration management (CM) and Authorization Management models have been completed using Intergraph Corporation's Asset and Information Management (AIM) software system. This implementation proves the plausibility of the model and also provides insight into how the models may be implemented on product data management systems.

AIM is an enterprise-wide electronic object management system. It provides an object-oriented framework with functionality provided for storage, query, security, and usage control. AIM's architecture provides a graphical environment to assist users in quickly locating and using objects. AIM manages information by ensuring that access is controlled and integrity is preserved throughout the life cycle of an object. AIM can be tailored to provide the following functions: administration of user, groups, and hosts; creation of a generic set of object classes and relationships; object creation, storage, vaults, and queries; and rule-driven security.

The workspace hierarchy is implemented in AIM using the features of *users* and *vaults* . Relationships between workspaces are enforced by defining *groups* , which contain related users, and limiting the access of these groups through the use of *rules* .

In AIM, each user has a private workspace. That is, there is a one-to-one mapping between a user and a private workspace. Rules are used to enforce the privacy of individual workspaces. Shared workspaces are implemented through the use of vaults. A vault is a logical collection of shared objects. Rules are used to control access to a vault. User-to-vault relationships can be established to allow visibility of ancestor workspaces. The global workspace consists of selected data from all shared workspaces (vaults), obtained through the AIM *query* capability. The AIM implementation of the RASSP workspace hierarchy is shown in Figure 4 - 1.

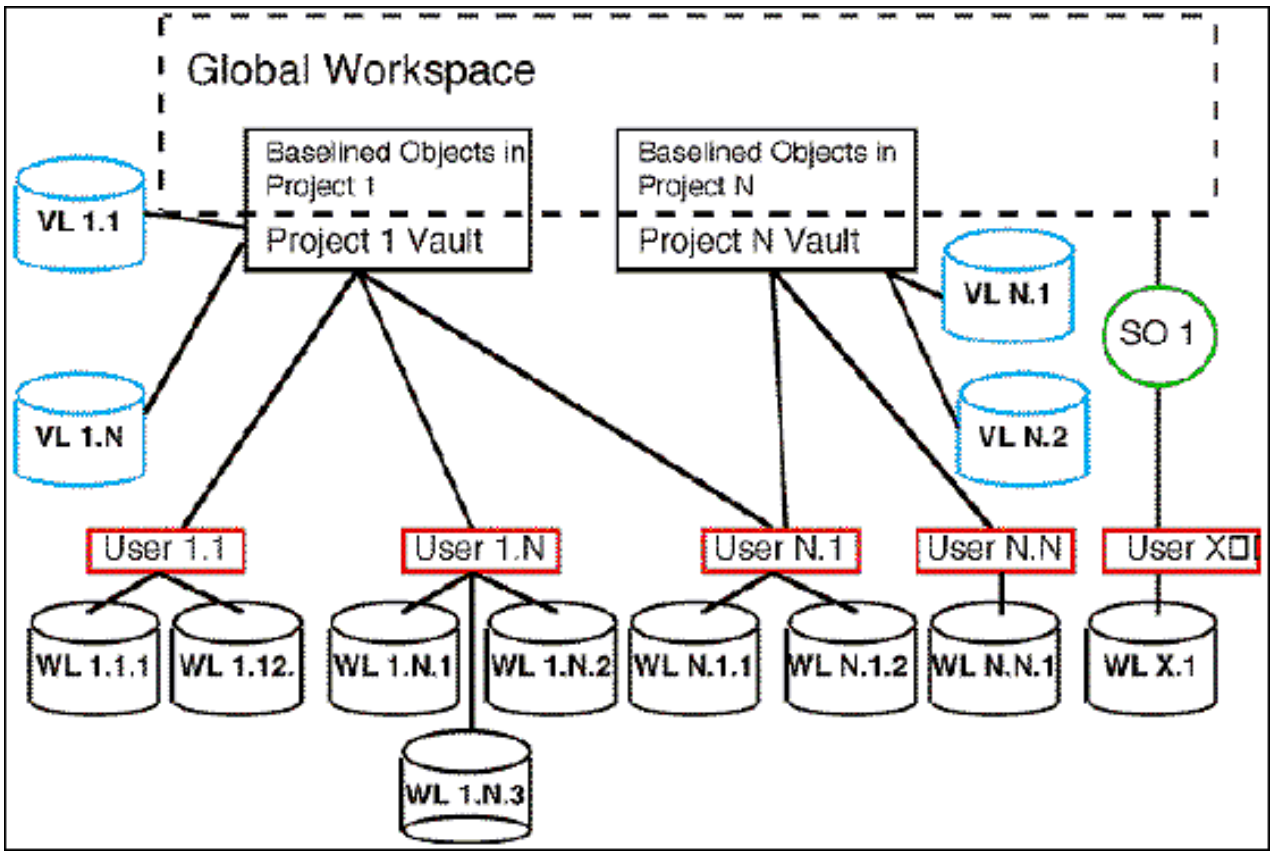

Figure 4 - 1: The AIM Workspace Hierarchy

*** Legend:**

=Represents the physical file system location for the items residing in the associated vault (shared workspace) or user (private workspace). Vault and work locations may be on the same host machine or on seperate machines wihin a network.
VL=Vault Location: Each vault location will be associated with only 1 vault.
WL=Work Location: Each work location will be associated with only 1 user.

=Represents a DM2 saved query.
SQ=Saved Query: Every user will have access to the Global Workspace through a pre-defined query.

=Represents a logical collection of objects based on ownership. An owner of an object is either a user or a vault. Each vault will have 1 or more vault locations associated with it. Each user may be associated with 1 or more vaults. Each user will have 1 or more work locations associated with it.

A *global workspace* in AIM is mapped to all baselined objects in all vaults within a database. Items are visible in a global workspace by use of the AIM Saved Query Object Class. *Shared workspaces* in AIM are mapped to a *vault/vault location* and can be accessed by performing transfer, check in, and check out operations. A vault is a logical collection of shared objects. A vault may contain data objects or actual file system items. A vault location provides a file system location for storing physical files owned by the vault.

In AIM, each user has a *private workspace* . Based on projects, a parent-child relationship can be established between private and shared workspaces. A private workspace may have more than one *work location* . Similar to vault locations, a work location provides actual file system space for objects residing in a private workspace. Note: A private workspace may have relationships with more than one shared workspace.

AIM provides out-of-the-box implementation of the RASSP Authorization model by using the AIM feature of message access rules. Message access rules are granting in nature. An explicit rule must exist for an action to be performed on an object class. If a rule exists, then the actions granted by the rule cannot be limited by another rule. In AIM, if two or more rules conflict or overlap, then the more permissive rule is always followed. Due to the object-oriented nature of AIM, *inheritance* plays an important part in maintaining the authorization hierarchy. Message access rules may be inherited throughout the AIM data model.

In addition to class and rule authorization, AIM provides additional enforcement of the RASSP Authorization Model by allowing roles to be divided into various groups of users. By dividing the users into five types of groups, administrative authority is distributed over the entire RASSP System. Figure 4 - 2 depicts the design of the RASSP Authorization user group hierarchy, and the association of the roles to the workspaces.

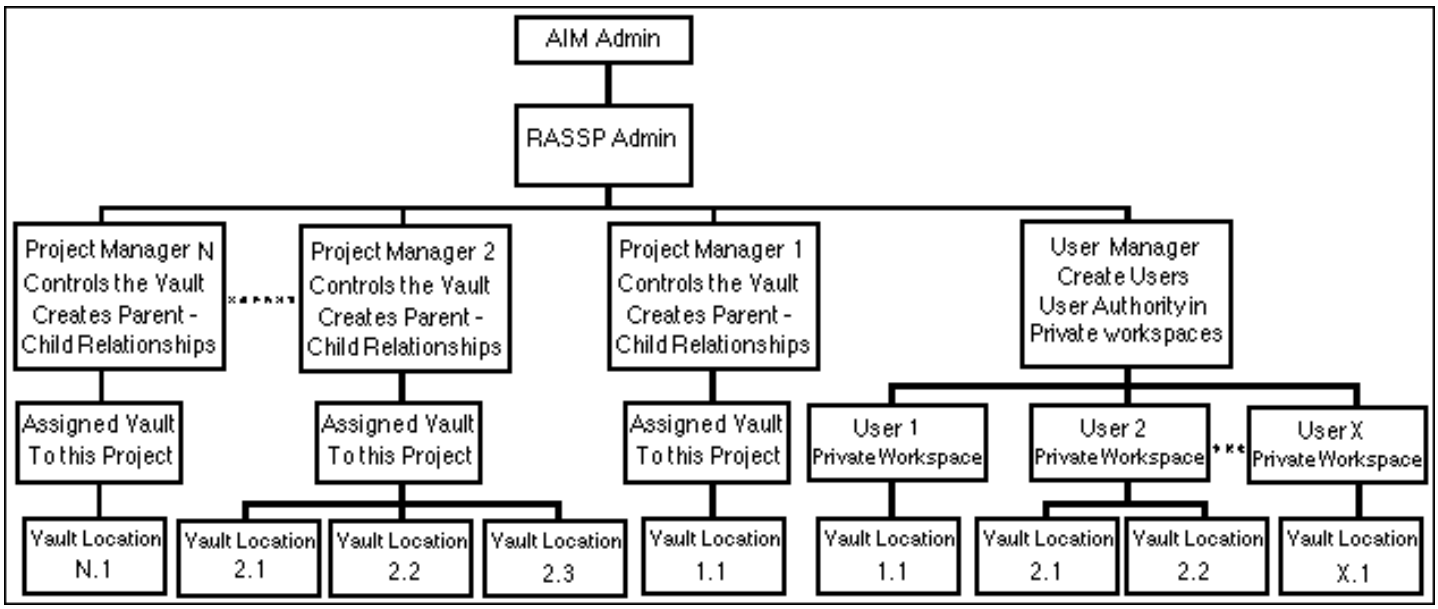

Figure 4 - 2: RASSP User Group Hierarchy Design

The AIMAdmin user commands the most authority over the RASSP Enterprise Framework. From his/her position in the hierarchy, he/she may perform any available action on any object, and thus perform the actions defined for any other user in the system. AIMAdmin controls and defines the RASSP environment. The RASSPAdmin user has limited super user authority. The RASSPAdmin can perform actions which govern the RASSP environment without being able to change the existing rules that

define it. RASSPAdmin may perform the actions of all users in the system except the AIMAdmin user.

The User Manager grp is a group of assigned users that governs and maintains the user accounts in the RASSP environment. RASSPAdmin, or AIMAdmin, must assign a user to the User Manager position. The User Manager commands authority over all the user accounts in the system. From his position in the hierarchy, he may perform any action available in any user account. This is a useful feature when conditions arise where users are unavailable due to vacation, sickness, or other absence.

The Project Mgr grp is a group of assigned users that control an instantiated project in the RASSP environment. The RASSPAdmin or AIMAdmin must assign a user to the Project Mgr grp. By doing this, a project manager is created. Each project manager is all powerful within the context of that project and maintains the associated Project Vault in the system.

The Project grp, and its associated rules, are created when the RASSPAdmin instantiates a new project. It is a group of users assigned to support an instantiated project in the RASSP environment. The project manager, (or RASSPAdmin or AIMAdmin) must select an existing user from the user grp, and assign that user to the Project grp. By doing this, the project manager has added a user to the project support group. Members of a project create design objects and are allowed to share these objects by placing them in the associated project vault.

The user grp is a group of users that exist in the RASSP environment and are available for assignment to the User Manager grp, Project Mgr grp, or Project grp. The User Manager, (or AIMAdmin or RASSPAdmin) must create new users, validate them on the appropriate Host(s), and place them in the user grp. The user grp commands the least authority in the administrative hierarchy.

Next Up Previous Contents

Next: 5 Lessons Learned **Up:** Appnotes Index **Previous:** 3 RASSP Manufacturing Interface Usage Scenarios

RASSP Information Management Appnote

5.0 Lessons Learned

The following lessons were learned from the application of the RASSP enterprise system to the benchmark programs:

1. It was discovered that the concept of a private workspace, as defined within the RASSP CM model, is very difficult to apply to the domain of electronic systems design. This is because many of the CAD tools, used for designing electronic systems, create a single design database and partition it internally to permit access by multiple designers. So the concept of each designer storing portions of a CAD tool design database within his private workspace will not work. For example, the schematic for a PCB cannot be stored in one designer's private workspace while the layout information is stored in another designer's private workspace because the CAD tool stores all the information regarding the PCB design within one design database.

   The solution to this problem was to modify the RASSP CM model to include the concept of a group workspace. A group workspace is defined as the place where shared data resides during development. It exists at the same level as a private workspace within the workspace hierarchy. The difference between a group workspace and a private workspace is that all designers have access to the group workspace. It is implemented at the file system level as a common directory to which all designers have access. An example of shared data, which would reside in the group workspace, is a Mentor Graphics Corp. design database. Upon completion of the design cycle, the shared data is then promoted up to either the shared or global workspace, as defined in the RASSP CM model, by checking it into the AIM PDM system.

2. The RASSP electronic design workflows, in general, represent the ideal "best case" design cycle. The benchmark programs did not follow best case design scenarios and customization of the workflows was required to handle unique situations. Even with the customizations in place, situations were encountered that the workflows were not set up to handle. On-the-fly modifications were required because it is difficult to predict and anticipate what problems will be encountered during the design cycle. When this happens, designers cannot wait for the workflow changes to be implemented and will work outside of the system to stay on schedule and to accomplish their goals. The lessons learned here are:
   - more communication is required between the domain experts (e.g. FPGA designers) and the workflow developers so that workflows may be developed which better models their design process
   - workflows must provide for as many contingencies as possible, i.e. be "worst case" vice "best case" design that uses the philosophy of "what could go wrong will go wrong"
   - further advances in workflow tool technology are required to provide a capability to better handle exceptions to design cycles and ad-hoc situations so designers are not forced to work outside of the workflow tool

3. A complex system such as the RASSP enterprise system is going to require fundamental changes during its use on any project. Examples of fundamental changes include encapsulating new CAD tools, modifying the workflows, and adding new users. These types of changes cannot be avoided because things like new people being added to a project, or changes in project scope often occur. In order to make these types of changes to the enterprise system, it had to be taken off line. After making the modifications, the enterprise system had to be brought back on line and restored back to its

original state. The lessons learned here include:

- the need to better plan for changes, perform them after hours so as not to adversely affect the benchmark teams
- the need for better communication between the benchmark teams and the enterprise system developers to minimize the frequency of these changes

4. System administration of the RASSP enterprise system needs to be performed by someone who is part of the regular systems administration group. Due to the developmental nature of the Enterprise system, the role of RASSP enterprise system administrator was performed by a member of the enterprise development team during the benchmark programs. The lack of communication between the RASSP enterprise system administrator and the services group caused some problems. For example, the upgrade of one designer's machine from the SunOS 4.1 operating system to the Solaris operating system resulted in that designer not being able to use the enterprise system. The way to prevent these types of problems from occurring is to have the RASSP enterprise system administration functions performed by someone who is cognizant of system administration and is part of an organization's services group. This can easily be accomplished through training and is not expected to be a large burden on the system administration staff.

5. The RASSP enterprise system was still in development while being used on the benchmark programs. Consequently, the benchmark teams encountered problems that should have been discovered and corrected during a normal testing phase. An example of a problem encountered was the inability of the DMM workflow tool to permit multiple designers to execute the same workflow step. Encountering problems like this often resulted in the benchmark teams working outside the enterprise system until they were corrected. The lesson learned here is that if a system, such as the RASSP enterprise system, is to be used while still in development, a plan must be put in place governing its use. The plan must specify the current and planned capabilities along with the development schedule. Upgrades to the system, such as installing new versions of Oracle, AIM, and DMM need to be performed during off-hours because they involve taking the system off line during the time the upgrade is being performed.

6. New users of the RASSP enterprise system should receive at least one week of training before using it. During the benchmark programs, insufficient time was allocated for training the users of the enterprise system. Due to ambitious project schedules, members of the benchmark teams were only able to attend a 1 hour training class before using the system.

RASSP Information Management Appnote

References

Bsharah, F.L., and Willis, J.F., "Build 1 Application Interpreted Model Report", Rockwell, December 1994. [AIMRI_94]

Application Interpreted Model, December 1994 [AIMI_94]

Bsharah, F.L., and Willis, J.F., "Build 1 Application Reference Model Report", Rockwell, December 1994. [ARMRI_94]

Application Interpreted Model, December 1994 [ARMI_94]

Bsharah, F.L., and Willis, J.F., "Build 1 Enterprise Data Model Report", Rockwell, December 1994. [EDMRI_94]

RASSP Enterprise Data Model, December 1994 [REDMI_94]

Kalathil, B "Library Management Report" Lockheed Martin ATL, June 1995.Application Interpreted Model, June 1995 [LMR_95]

Bsharah, F.L., and Willis, J.F., "Build 2 Application Interpreted Model Report", Rockwell, March 1996. [AIMR2_96]

Bsharah, F.L., and Willis, J.F., "Build 1 Enterprise Data Model Report", Rockwell, March 1996. [EDMR2_96]

Blanchard, D. "The Configuration Management Model for the RASSP System, Version 3 ", Lockheed Martin ATL, August 1996. [BLANCHARD_96]

Application Notes

Enterprise Framework Implementation
Reuse Methodology and Implementation

 Dennis Basara