ISSN 2228-9860 eISSN 1906-9642 CODEN: ITJEA8

International Transaction Journal of Engineering, Management, & Applied Sciences & Technologies

http://TuEngr.com

A Hybrid Approach for Recovering Use Case Models of MVC Web Applications

Emad Y. Albassam[1*]

[1] *Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, SAUDI ARABIA.*

**Corresponding Author (Email: ealbassam@kau.edu.sa).*

Paper ID: 12A13T

Volume 12 Issue 13

Received 24 June 2021
Received in revised form 28 September 2021
Accepted 15 October 2021
Available online 23 October 2021

Keywords:
Use Case Models; Reverse Engineering; Recovery; Architectural-Patterns; Design-Patterns; Model-View-Controller; MVC; Web Applications; Role-Based Access Control.

Abstract

In Model-Driven Software Engineering (MDSE), software systems are constructed from abstract models that are used to guide the development process. Use Case models are abstract models that aim to capture the intended behavior of a software system from the point of view of its actors. Use Case models are not only helpful during development time but can also assist in software evolution and modernization. Such post-development benefits can only be obtained with up-to-date Use Case models. However, manual maintenance of these models may cause divergence such that these models can become outdated as the software system evolves over time. Furthermore, legacy software systems may not have well-documented Use Case models. Therefore, it is beneficial to recover these models through reverse engineering of source code and supplementary documentation of the software system. This paper proposes a hybrid approach for recovering Use Case models for web applications. The proposed approach relies on defining recovery patterns for known architectural- and design patterns that are widely used to construct web applications. Each recovery pattern shows how a particular Use Case model element, including Use Cases, actors, relationships, and non-functional requirements of Use Cases, are to be recovered. Both static and dynamic analyses of the web application's source code are then performed based on recovery patterns to recover the various Use Case model elements. The proposed approach is applied to an open-source, real-world MVC web application. Results show adequate recovery of Use Case model elements of this application.

Disciplinary: Information System and Computer Science & Engineering (Software Engineering).

Cite This Article:

Albassam, E Y. (2021). A Hybrid Approach for Recovering Use Case Models of MVC Web Applications. *International Transaction Journal of Engineering, Management, & Applied Sciences & Technologies, 12*(13), 12A13T, 1-17. http://TUENGR.COM/V12A/12A13T.pdf DOI: 10.14456/ITJEMAST.2021.272

1 Introduction

Research has shown that maintaining software documentation after deployment through conventional, manual approaches can be costly, time-consuming, and may result in inconsistencies due to human errors and the continuous evolution of software systems [1] [4]. Furthermore, many legacy software systems are either partially documented or lack any documentation [3] [5]. As a result, certain post-development activities such as software evolution and modernization can become difficult in such cases [11]. Therefore, researchers have investigated approaches and techniques in which software documentation can be recovered from implementation artifacts such as source code and supplementary documentation of the software system [8]. The Unified Modeling Language (UML) [12] has become prominent for documenting software systems through abstract models such that each UML model provides a different view of the software system [13]. The purpose of UML Use Case models is to capture the functional requirements of a software system from the point of view of its actors. It has been shown that Use Case models aid in improving software maintenance [2]. In addition, recovery of Use Case models for existing and legacy software systems is considered important in software modernization since software requirements of such systems need to be considered and analyzed by requirement engineers so that they are either partially or fully fulfilled by the replacing software system.

This paper proposes an approach for recovering Use Case models for MVC web applications. The proposed approach is based on well-known architectural- and design patterns that are widely used to construct such applications. In this approach, recovery patterns are defined based on architectural and design patterns. Each recovery pattern defines how a specific concept related to Use Case models, such as actors, Use Cases, and relationships between Use Cases can be formally recovered. Both static and dynamic analyses of the application's source code are then performed to recover these elements based on the defined recovery patterns. Static and dynamic analyses are conducted in a phased approach to recover the various Use Case model elements incrementally. Furthermore, dynamic analysis is performed to refine the recovered Use Case model with additional details that otherwise cannot be recovered during static analysis. In addition, the paper describes the implementation details of a proof-of-concept tool, named REC-MVC, which realizes the proposed approach. The proposed approach is validated through a case study in which the Use Case model of an open-source, real-world MVC web application is recovered using REC-MVC.

2 Related Work

Recovery of software models from source code and/or supplementary software documentation has long been the subject of much research [8]. Such approaches vary based on whether recovery relies on the analysis of the application's source code (i.e. white-box recovery) or not (i.e. black-box recovery). Additionally, such approaches may utilize various techniques including (1) static analysis of the application's source code such that the source code is analyzed statically without running the actual application, (2) dynamic analysis in which the application is executed to collect runtime specific information such as execution traces, or (3) both static and dynamic analysis techniques. Finally, such approaches may target recovery of different abstractions, such as the recovery of Use Case models, state machines, class diagrams, Entity Relationship-Diagrams (ERDs), data-flow diagrams, and/or other abstractions. The remainder of this section presents approaches for Use Case model recovery as they are closely related to this work.

An approach by Ceponiene et al. [7] describes how a website's Use Case model can be recovered dynamically without accessing the internal source code. Their approach relies on recording and analyzing user preformed activities at runtime while the software system is executing. To accomplish this, users indicate beforehand their roles and the processes (i.e. use cases) they will perform. Similarly, work by Dugerdil et al. [10] presents a dynamic decision tree approach for recovering Use Case models in which human intervention is required to document system use cases. Compared to these approaches, we assume that use cases may require certain privileges for actors to access them and show how actors, Use Cases and Use Case access privileges can be automatically recovered using static analysis of source code without having the users indicate such information.

Li et al. [14] describe an approach in which dynamic analysis of the software system's source code is performed to mine and discover the system's Use Case model. Zhang et al. [9] show how use case models can be recovered from source code. However, these approaches require assistance from a domain expert to recover certain Use Case model elements such as actors and Use Case relationships.

Work by Santos et al. [6] presented an automated approach for recovering Use Cases of MVC web applications in which source code and endpoint URLs are analyzed to recover Use Cases. Miranda et al. [8] investigated how Use Case models can be recovered from Graphical User Interfaces (GUIs). However, their work relies solely on static analysis of source code and does not address invocation of polymorphic operations nor consider recovery of non-functional requirements such as authorization access of Use Cases.

Compared to these works, this paper investigates a hybrid approach for recovering Use Case models from MVC web applications. The approach is based on widely used architectural- and design-patterns in this domain to facilitate the recovery process since knowledge of such patterns can be exploited to address some of the limitations in previous works. The proposed approach is hybrid in that it incorporates both static and dynamic analysis such that the various Use Case model elements are recovered incrementally. Furthermore, dynamic analysis is performed to refine the discovered model by analyzing polymorphic operation invocations. In addition, this paper investigates how nonfunctional requirements, namely Use Case access privileges, can be automatically recovered and used to guide the recovery process.

3 Architectural- and Design-Patterns for Web Applications: Background

This section provides a background on existing concepts, such as architectural- and design-patterns, that are commonly used to construct web applications and form the basis of the proposed approach presented in this paper.

3.1 Model-View-Controller (MVC) Architectural-Pattern

In the Model-View-Controller (MVC) pattern [15], user requests are received and processed by controller actions (see Fig. 1). Upon receiving a request, the controller's action issues commands to related models and views to fulfil the user's request. Models are data structures that may encapsulate domain knowledge, business rules, and/or application data. Views on the other hand are responsible for presenting responses to the end-user given zero or more model instances. Therefore, MVC separates concerns related to coordination (i.e. controllers) from application data (i.e. models) and presentation (i.e. views).

Furthermore, it is common for many currently available MVC web frameworks, such as ASP.NET MVC Core and Spring framework, to build the MVC architectural-pattern on top of the HTTP protocol such that each controller action is bound to a specific HTTP method (e.g. HTTP Get, HTTP Post, and HTTP Delete).

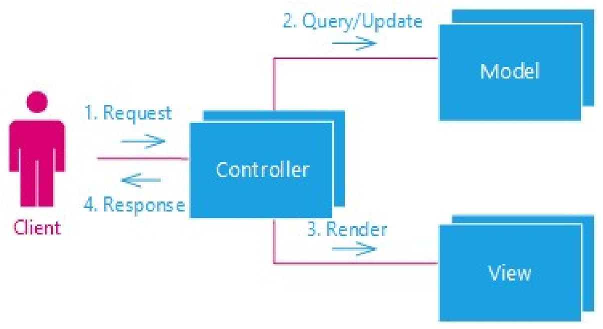


Figure 1. The Model-View-Controller (MVC) Pattern.

3.2 Service-Oriented Architecture (SOA)

The service-oriented architectural (SOA) pattern is widely used to compose software applications, including MVC web applications, from reusable services [13] such that accesses of these services are determined and coordinated by coordinator components (see Figure 2). As there are many approaches to implement service-oriented architectures, we consider in this paper implementation of SOA through the Simple Object Access Protocol (SOAP).

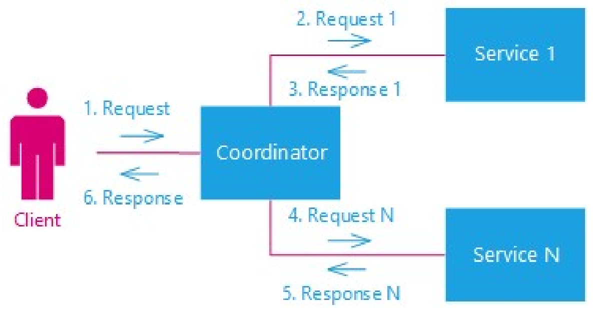


Figure 2: The Service-Oriented Architectural (SOA) Pattern. Example of sequential services access.

3.3 Role-Based Access Control (RBAC)

The Role-Based Access Control (RBAC) pattern [17] is commonly used to regulate accesses to web resources such as controller actions in the MVC architectural pattern. In this pattern (see Figure 3), roles are created to define sets of resource access policies. Each role is then assigned a set of permissions of allowed (or disallowed) resources by this role. Application users are assigned to roles either directly or indirectly via user groups. The application users are then authorized to access (or denied from accessing) resources by an authorization component based on their roles in the application.

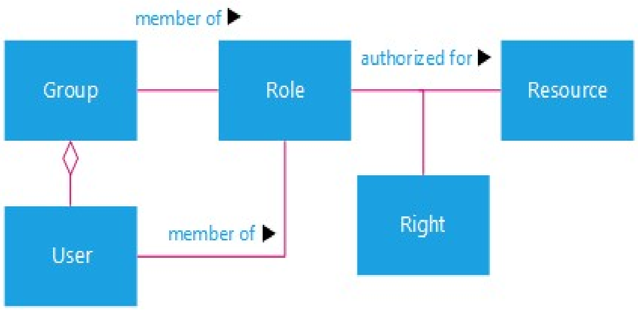


Figure 3: Role-based Access Control Pattern.

3.4 Layer Supertype Pattern

The Layer Supertype pattern [16] can be used in situations in which multiple controllers in the MVC architectural-pattern share common behaviors (e.g. authorization validation) such that these common behaviors are extracted into an abstract parent controller (i.e. the controller layer supertype). The controller supertype (see Figure 4) is then specialized into multiple concrete controllers that can reuse the common behaviors implemented by the super type.

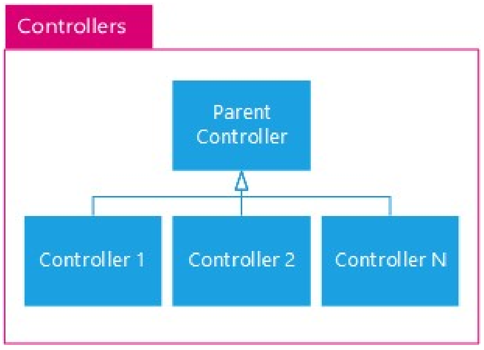


Figure 4: Layer Supertype Pattern.

4 Recovery Patterns For MVC Web Applications

This section defines recovery patterns for recovering Use Case models. Each recovery pattern is associated with a specific architectural- or design pattern and shows how a specific Use Case model element, such as actors, Use Cases, and relationships, are to be recovered.

4.1 Recovery of Actors

In Use Case models, actors are entities in the external environment that interact with the software system. An actor can be either a human user, an external system, or an I/O device [13].

Since it is common for human users and external systems to interact with web applications, this section shows how each type of these actors is recovered by the proposed approach.

1) Recovery of Human Actors

The recovery pattern of human actors is based on the role-based access control (RBAC) pattern which is widely used in MVC web applications to implement authorization mechanisms. In RBAC, access to MVC controller actions can be restricted to authorized users based on their permissions. Since human actors interact with the software system through controller actions, then recovery of human actors in the Use Case model can be based on analyzing RBAC data sources as follows.

Let $R = r_1..r_n$ be the set of roles in the software application and $P_i = p_1..p_m$ be the set of permissions that are assigned to role r_i. Since each role $r \in R$ defines a valid role of authorized users in the application, then each role is initially part of the actors set A. Therefore,

$$\forall r \in R \text{ then } r \in A$$

In use case models, an actor can be specialized from another actor through the generalization/specialization relationship. A specialized actor is associated with every Use Case as its parent actor but can also be associated with additional Use Cases. To recover the generalization/specialization relationship between the actors in A, the set of permissions P_i with $i = 1..m$ is analyzed as follows.

Let $Child_i$ denotes the set of child actors for actor $a_i \in A$. Then,

$$\forall a_i, a_j \in A \text{ with } i \neq j,\ P_i \subset P_j \Rightarrow a_i \in Child_j$$

That is, for every two actors a_i and a_j in A, if actor a_i's permissions is a proper subset of actor's a_j permissions, then actor a_i is a child of (i.e. specialized from) actor a_j.

2) Recovery of External Systems

An MVC web application may require services from external systems to provide its intended functionality to its users. For instance, a travel agency system may require the services of external airline, car rental, and payment gateway systems. Therefore, it is important to identify such external systems as actors in the recovered Use Case models.

In SOAs, software systems expose their services for consumption via endpoints. Let S be the set of services that are consumed by the software application. Each service $s_i \in S$ is bound to one or more endpoints through which s_i can be invoked. Therefore, the set of actors A is expanded as follows:

$$\forall s_i \in S \text{ then } s_i \in A$$

It should be noted that relying on the recovery pattern in the previous section for recovering human actors is not sufficient, since the software system may send output messages to the external system without receiving input messages from this system, and thus, such external system are not covered by the RBAC data sources.

4.2 Recovery of Use Cases and Their Relationships

In Use Case models, each use case represents a distinct functionality that provides a value to a system's actor. Furthermore, a use case may include or extend other use cases. This section defines how use cases and their relationships can be recovered in MVC web applications.

1) Recovery of Use Cases

A use case defines a sequence of interactions between one or more actors and the software system. Such interaction is initiated by the primary actor of the use case [13]. Since (1) use case interactions are initiated by primary actors and (2) all incoming requests to the MVC architectural-pattern are received by controller actions, then use cases of MVC web applications can be recovered by statically analyzing controller actions. Let $C = c_1..c_n$ be the set of controllers in the application and $T_{c_i} = t_1..t_m$ be the set of actions of controller $c_i \in C$, then the set U of use cases for the application can be recovered as follows:

$$\forall c \in C \text{ then } T_{c_i} \in U$$

2) Recovery of Extend Relationship

The extended relationship capture optional and/or supplementary interactions within use cases. For instance, lengthy alternatives and exceptional interactions of an extended use case can be extracted into an extending use case. Therefore, the extended use case is considered independent and may execute without triggering the interactions defined by the extending use case. As an example, an action responsible for creating a user's account may redirect the user request to another action for error handling if the connection to the backend database cannot be established. As a result, the use case corresponding to the action responsible for error handling extends the use case corresponding to the action responsible for user account creation.

Since in this approach (1) use cases are mapped to controller actions and (2) a controller's action can redirect the requests it receives to another action (within the same controller or another controller) for further processing, then the extended relationship can be recovered via static analysis of source code by identifying such request redirects.

Given the set C of controllers in the application and the set T_{c_i} of actions for controller $c_i \in C$, let $t.ST$ where $t \in T_{c_i}$ be the root of the syntax tree that defines the behavior of action t, then the extending use cases of the extended use case corresponding to action t, denoted as the set E_t, can be recovered by statically analyzing and searching for any expression that is of type controller action redirection. The arguments list of these expressions are inspected to find the target controller action (i.e. target of the redirect). The target controller action is then included in E_t.

3) Recovery of Include Relationship

The include relationship in use case models can be used to extract common interactions between different use cases into a shared-use case. The included use case is considered abstract such that it cannot be executed independently without including use cases.

We consider that common interactions among uses cases are implemented through the Layer Supertype pattern such that a parent controller (i.e. the controller layer supertype) implements the common operations which are inherited by child controllers using the generalization/specialization relationship. Therefore, to recover such actions, analysis of source code is performed to extract concrete controller classes that inherit from a parent controller class. Given the set of controllers C, then the set of included use cases for an including use case c_i, denoted as the set I_{c_i}, can be recovered as follows:

$$\forall c_i, c_j \text{ with } i \neq j,\ c_i \text{ parent of } c_j \Rightarrow c_i \in I_{c_j}$$

4.3 Recovery of Use Case Details

In the proposed approach, the details of each Use Case are recovered as a textual narrative description that defines the sequence of interactions between the actors of the software system and the software system, see Table 1.

Table 1: A subset of heuristic rules used to recover Use Case details

	Rule Name	Transformed Form	Description
1	Use Case Initiation	"The actor {ActorName} requests to {ActionName} {Plural(ControllerName)}"	Rule for transforming Use Case initiation by the primary actor, which maps to a user's request received by an controller action in the MVC pattern. identifier of the action
2	Model Query	"The system queries {ModelName} into [{variable name}]"	Rule for transforming models queries in the MVC architectural pattern
3	Model Query All	"The system retrieves all entries in {ModelName} into [{variable name}]"	Rule for transforming models queries in which all entries are retrieved
4	Model Updates	"The system update {ModelName}"	Rule for transforming models updates, including inserting new or updating existing elements
5	Model Delete	"The system deletes from {ModelName}"	Rule for transforming elements deletion from a model in the MVC architectural pattern
6	External System Interaction	"The system communicates with the external system {ExternalSystemName}"	Rule for transforming message exchanges between the system and external systems
7	Use Case Inclusion	"Include {Included Use Case Name}"	Rule for transforming Use Case include relationships
8	Use Case Extension	"{Extending Use Case} ”	Rule for transforming Use Case extend relationships

For each use case $c_i \in C$, the syntax tree $t_i.ST$ of the action that corresponds to c_i is statically analyzed. Nodes in the syntax tree that are architecturally significant are identified and classified using a set of heuristic rules for transforming such nodes into a textual, human-readable form. Table 1 lists a subset of the used heuristic rules. In this table, each rule contains the textual form for the node that matches the rule. Furthermore, each textual form may contain placeholders that are replaced by a token in the syntax tree. As an example, the statement "allAccount = Accounts.All()" where Account is defined as a Model matches rule 3 in table 1 and is transformed into "The system retrieves all entries in Accounts into allAccounts". Section V.C discusses in more details the algorithm used to transform source code into textual narrative description using static analysis.

5 Validation

The proposed approach described in Section 4 is validated through a case study. In this case study, the Use Case model of an open-source and cross-platform content management system, Piranha CMS, is automatically recovered. Piranha CMS is implemented using the ASP.NET MVC Core framework and consists of 27 related projects. Table 2 provides some relevant metrics related to Piranha CMS to show its complexity to interested readers.

Table 2: Code metrics results of the Piranha CMS solution.

Number of Projects	27
Maximum Cyclomatic Complexity	2,559
Maximum Depth of Inheritance	6
Lines of Source Code	143,940
Lines of Executable Code	42,112

In order to automate the recovery process of the Use Case model using the recovery patterns described in the previous section, a proof-of-concept tool, titled REC-MVC, has been implemented and applied to recover the Use Case model of Piranha CMS. REC-MVC recovers the various Use Case model elements incrementally in a phased approach such that the output of a phase is used as a subset of the input to the next phase. These phases include: the initial reflection of the MVC web application's source code, static analysis of the source code, and dynamic analysis of operation invocations and are described next.

5.1 REC-MVC: Recovering use Case Diagram via Reflection of MVC Web Applications

During the first phase of REC-MVC, reflection code is injected into the solution's startup class responsible for configuring the MVC web application. The solution is then executed to collect runtime information of the running application and query relevant class metadata. This phase consists of five activities as follows:

- Recovery of the application's Use Cases.
- Recovery of HTTP methods for each Use Case.
- Recovery of the application's actors.
- Recovery of Use Case relationships.
- Recovery of Use Case access privileges.

Figure 5 shows a fragment of the injected code responsible for code reflection and querying. As seen in Figure 5, reflection is used to obtain the list of all controller actions in the MVC application. Use Cases are recovered by REC-MVC from the action's name. Furthermore, Use Cases are grouped into UML packages such that each UML package corresponds to a controller in the MVC application. The HTTP method and actors are also recovered by inspecting attribute annotations of controllers and actions. Since each Use Case may require certain privileges to be accessed, such non-functional requirements are also recovered by inspecting the policies of authorization attributes. Finally, relationships between use cases are recovered as described in section IV (not shown in Figure 5).

To illustrate the recovery process performed by REC-MVC using reflection for recovering the various Use Case diagram elements, Figure 6 shows an example of a controller implementation in Piranha CMS. Since in this example the controller name is LanguageApiController, then a UML package is recovered by REC-MVC with the name LanguageApi. Since this controller implements 3 actions with the names Get, Save, and Delete, then these actions are recovered as Use Cases in the LanugageApi package. Finally, since the controller has an authorization attribute with Admin policy, then this is recovered by REC-MVC as the actor required to access the Use Case. It should be noted that controllers without an Authorize attribute are associated with the default User.

Figure 7 shows the recovered use case diagram of Piranha CMS using REC-MVC (where actors are associated with use cases of matching colors to reduce diagram cluttering). From Figure 7, REC-MVC is able to recover a total of 94 use cases and associate these uses cases to the 3 discovered actors. In addition, analysis performed by REC-MVC revealed recovery of Include relationships between the use cases of Role and User packages (the including use cases) and the use cases of the Manager package (the included use cases). To confirm this, an inspection of the RoleController and UserController revealed that these controllers inherit from a base controller, ManagerController, and therefore actions and methods of the former controllers can reuse the methods of the later controller.

During this phase, privileges required to invoke uses cases by actors are also recovered through reflection. Table 3 shows a fragment of the recovered privileges by REC-MVC for Piranha MVC. To recover Use Case access privileges, REC-MVC analyzes authorization policies associated with controllers and controller actions.

5.2 REC-MVC: Recovery of Use Case Details Via Static Analysis

In order to recover the details of each use case, REC-MVC performs static analysis of the MVC web application's source code using the Roslyn static analyzer. To avoid analysis of the entire source code, static analysis is guided by the results of the previous phase such that only controllers associated with recovered use case packages are analyzed.

Algorithm 1 shows how REC-MVC performs this phase. First, REC-MVC loads the solution of the entire MVC application. It is assumed that the solution may consist of more than one project. As an example, inspection of Piranha CMS solution revealed that it consists of 27 projects. Therefore, REC-MVC iterates over each of these projects and then runs static analysis to compile and obtain the syntax trees of these projects. Each syntax tree is then analyzed to retrieve method declaration nodes such that these nodes are declared by a controller associated with a recovered use case package. Each method declaration node is then statically analyzed further by analyzing its descendant nodes, which correspond to the implementation of this method. Descendant nodes of a method declaration node corresponding to statements such as method invocation expressions, for each statements, and if statements are then inspected and transformed to textual natural language.

To facilitate the transformation of source code into a natural language, transformation rules are applied. These rules are embedded into two dictionaries implemented by REC-MVC: the predefined rules dictionary and the custom rules dictionary. The predefined rules dictionary maintains a mapping between common MVC concepts, such as common Model operations, to their natural language counterpart. For instance, a model that is defined as a collection of objects provides operations to select, query, or remove objects from the collection. Such operations are defined in the dictionary as the key of the map while the value of each key is the natural language counterpart of the operation. REC-MVC also allows keys of the map to be defined as regular expressions to match different possible variances of implementations of the operations. Table 4 shows a fragment of the entries stored by the predefined rules dictionary.

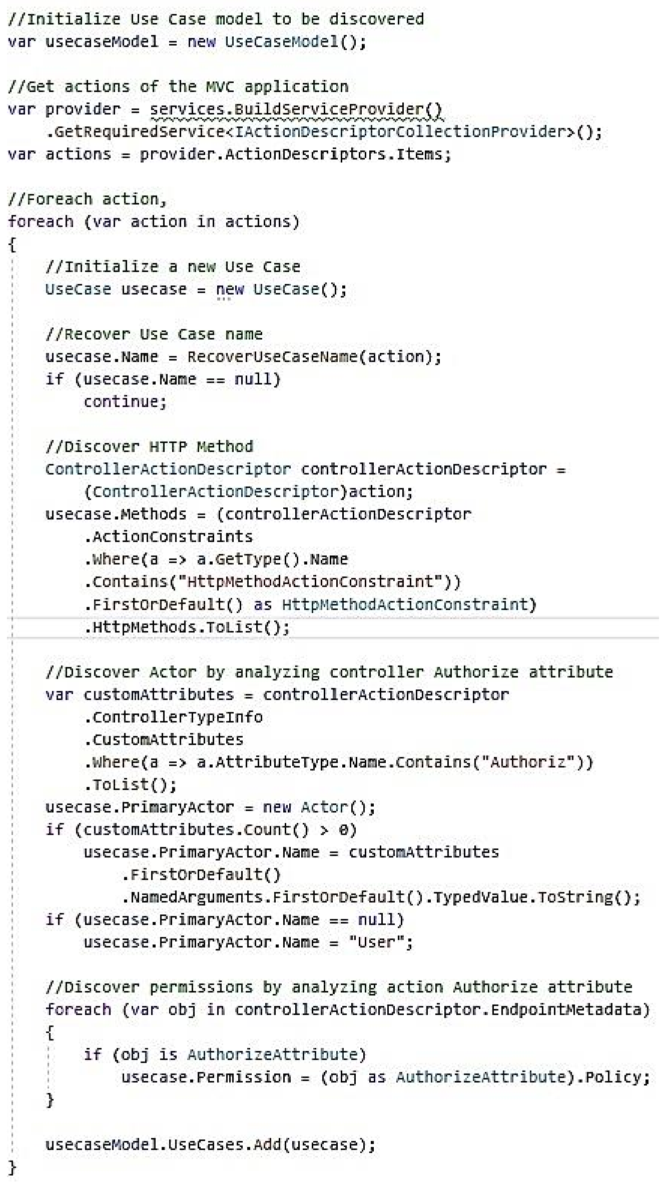
Table 3: Fragment of recovered privileges required to invoke use cases in Piranha CMS.

Actor	Package	Use Case	Required Privilege
User	CMS	Archive	none
User	CMS	Page	none
User	CMS	PageWide	none
User	CMS	Post	none
User	CMS	TeaserPage	none
PiranhaAdmin	AliasApi	List	PiranhaAliases
PiranhaAdmin	AliasApi	Save	AliasesEdit
PiranhaAdmin	AliasApi	Delete	PiranhaAliasesDelete
PiranhaAdmin	CommentApi	List	PiranhaComments
PiranhaAdmin	CommentApi	Approve	PiranhaCommentsApprove
PiranhaAdmin	CommentApi	UnApprove	PiranhaCommentsApprove
PiranhaAdmin	CommentApi	Delete	PiranhaCommentsDelete
PiranhaAdmin	ContentApi	GetBlockTypes	PiranhaAdmin
PiranhaAdmin	ContentApi	CreateBlockAsync	PiranhaAdmin
PiranhaAdmin	ContentApi	CreateRegionAsync	PiranhaAdmin
PiranhaAdmin	ContentApi	List	PiranhaContent
PiranhaAdmin	ContentApi	Get	PiranhaContent
PiranhaAdmin	ContentApi	Create	PiranhaContentAdd
PiranhaAdmin	ContentApi	Save	PiranhaContentSave
PiranhaAdmin	ContentApi	Delete	PiranhaContentDelete

Table 4: Fragment of entries stored by the predefined and custom rule dictionaries.

Key	Value	Comment
.**.Where.*	The system queries	a regular expression key in the predefined rules map
Single	The system retrieves a single entry from	a simple key in the predefined rules map
Language Service	Languages	a simple key in the custom rules map to rename a class

On the other hand, the custom rules dictionary allows for customizing the transformation process by REC-MVC by manually 1) defining custom mapping entries and/or 2) overriding existing rules defined by the predefined rules dictionary. For instance, the custom rules dictionary can be used to map variable names into a readable form. It should be noted that multiple rules from both dictionaries can be matched during the transformation process. In such cases, REC-MVC applies matched rules from the custom rules dictionary over the predefined rules dictionary. Furthermore, if multiple rules are matched from the same dictionary, REC-MVC applies the first matching rule based on the entry index in the list.

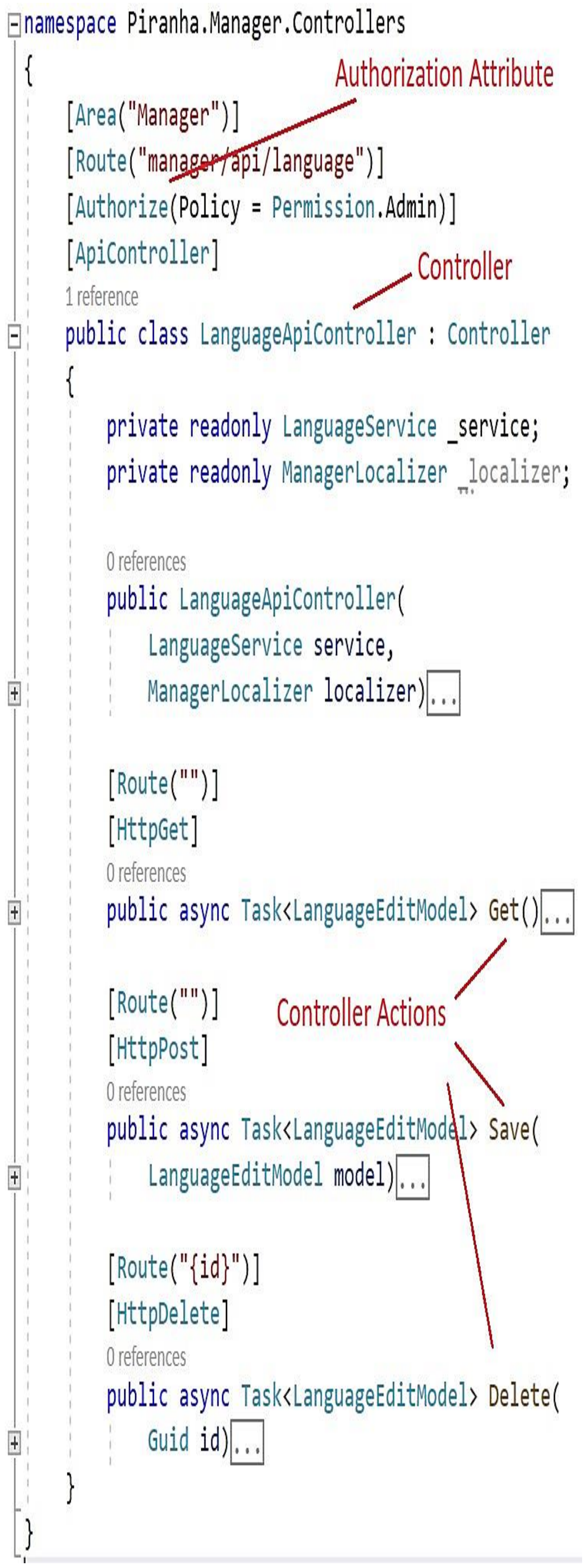


```
//Initialize Use Case model to be discovered
var usecaseModel = new UseCaseModel();

//Get actions of the MVC application
var provider = services.BuildServiceProvider()
    .GetRequiredService<IActionDescriptorCollectionProvider>();
var actions = provider.ActionDescriptors.Items;

//Foreach action,
foreach (var action in actions)
{
    //Initialize a new Use Case
    UseCase usecase = new UseCase();

    //Recover Use Case name
    usecase.Name = RecoverUseCaseName(action);
    if (usecase.Name == null)
        continue;

    //Discover HTTP Method
    ControllerActionDescriptor controllerActionDescriptor =
        (ControllerActionDescriptor)action;
    usecase.Methods = (controllerActionDescriptor
        .ActionConstraints
        .Where(a => a.GetType().Name
        .Contains("HttpMethodActionConstraint"))
        .FirstOrDefault() as HttpMethodActionConstraint)
        .HttpMethods.ToList();

    //Discover Actor by analyzing controller Authorize attribute
    var customAttributes = controllerActionDescriptor
        .ControllerTypeInfo
        .CustomAttributes
        .Where(a => a.AttributeType.Name.Contains("Authoriz"))
        .ToList();
    usecase.PrimaryActor = new Actor();
    if (customAttributes.Count() > 0)
        usecase.PrimaryActor.Name = customAttributes
            .FirstOrDefault()
            .NamedArguments.FirstOrDefault().TypedValue.ToString();
    if (usecase.PrimaryActor.Name == null)
        usecase.PrimaryActor.Name = "User";

    //Discover permissions by analyzing action Authorize attribute
    foreach (var obj in controllerActionDescriptor.EndpointMetadata)
    {
        if (obj is AuthorizeAttribute)
            usecase.Permission = (obj as AuthorizeAttribute).Policy;
    }

    usecaseModel.UseCases.Add(usecase);
}
```

Figure 5: Code snippet showing implementation details of REC-MVC for recovering use case diagrams.

Algorithm 1: Algorithm executed by REC-MVC for recovering Use Case details.

```
Result: Discovered Use Case diagram
Input:  solutionPath: solution path of the MVC web
        application.
        usecaseModel: discovered Use Case model
        from previous phase.
solution ← LoadSolution(solutionPath);
// loads the solution of the MVC web
   application. The solution may
   consist of multiple projects.
model ← usecaseModel.UseCases;
// loads recovered Use Cases.
foreach project ∈ solution do
    compilation ← project.GetCompilation();
    // Runs Roslyn static analyzer to
       obtain the compilation result
       for this project.
    foreach tree ∈ compilation.SyntaxTrees do
        root ← tree.GetRoot();
        // Gets the root node of this
           syntax tree
        semanticModel ←
         compilation.GetSemanticModel(tree);
        // Runs static analyzer to
           obtain the semantic model
           for this syntax tree.
        foreach usecase ∈ model do
            methods ← root.GetRoot();
             .OfType<MethodDeclarationSyntax>();
            // search descendant nodes
               of root for method
               declarations
            if methods.Ancestors()
             .OfType<TypeDeclarationSyntax>()
             .First() <> usecase.ControllerName
             then
                continue;
            foreach method ∈ methods do
                foreach node ∈
                 method.DescendantNodes() do
                    if node.Kind() ==
                     InvocationExpression then
                        TransformInvocation(node);
                    else if node.Kind() ==
                     ForEachStatement then
                        TransformForEach(node);
                    else if node.Kind() == IfStatement
                     then
                        TransformIf(node);
                    // other cases omitted
                end
            end
        end
    end
end
```



```
namespace Piranha.Manager.Controllers
{
    [Area("Manager")]
    [Route("manager/api/language")]
    [Authorize(Policy = Permission.Admin)]
    [ApiController]
    1 reference
    public class LanguageApiController : Controller
    {
        private readonly LanguageService _service;
        private readonly ManagerLocalizer _localizer;

        0 references
        public LanguageApiController(
            LanguageService service,
            ManagerLocalizer localizer)...

        [Route("")]
        [HttpGet]
        0 references
        public async Task<LanguageEditModel> Get()...

        [Route("")]
        [HttpPost]
        0 references
        public async Task<LanguageEditModel> Save(
            LanguageEditModel model)...

        [Route("{id}")]
        [HttpDelete]
        0 references
        public async Task<LanguageEditModel> Delete(
            Guid id)...
    }
}
```

Figure 6: Piranha CMS Controller code snippet. (obtained from https://piranhacms.org).

5.3 Refining use Case Recovery Through Dynamic Analysis

In the previous section, REC-MVC relies on static analysis for recovering the details of Piranha CMS Use Cases. However, one limitation of static analysis is the recovery of operation invocations that can only be resolved at the run time of the MVC web application due to dynamic dispatching and polymorphic operations.

For instance, Piranha CMS provides multiple mechanisms for handling media content. Therefore, the *IStorage* interface defines common operations, such as opening a new storage session and getting a resource name, for such mechanisms. Different storage mechanisms are then implemented by different concrete classes that realize the *IStorage* interface such that each class defines the actual behavior for a specific mechanism. For instance, the *FileStorage* class implements the *IStorage* interface for handling media content on the webserver hosting the actual MVC web application. On the other hand, the *BlobStorage* class implements the *IStorage* interface for handling media content stored on the cloud via a service provider.

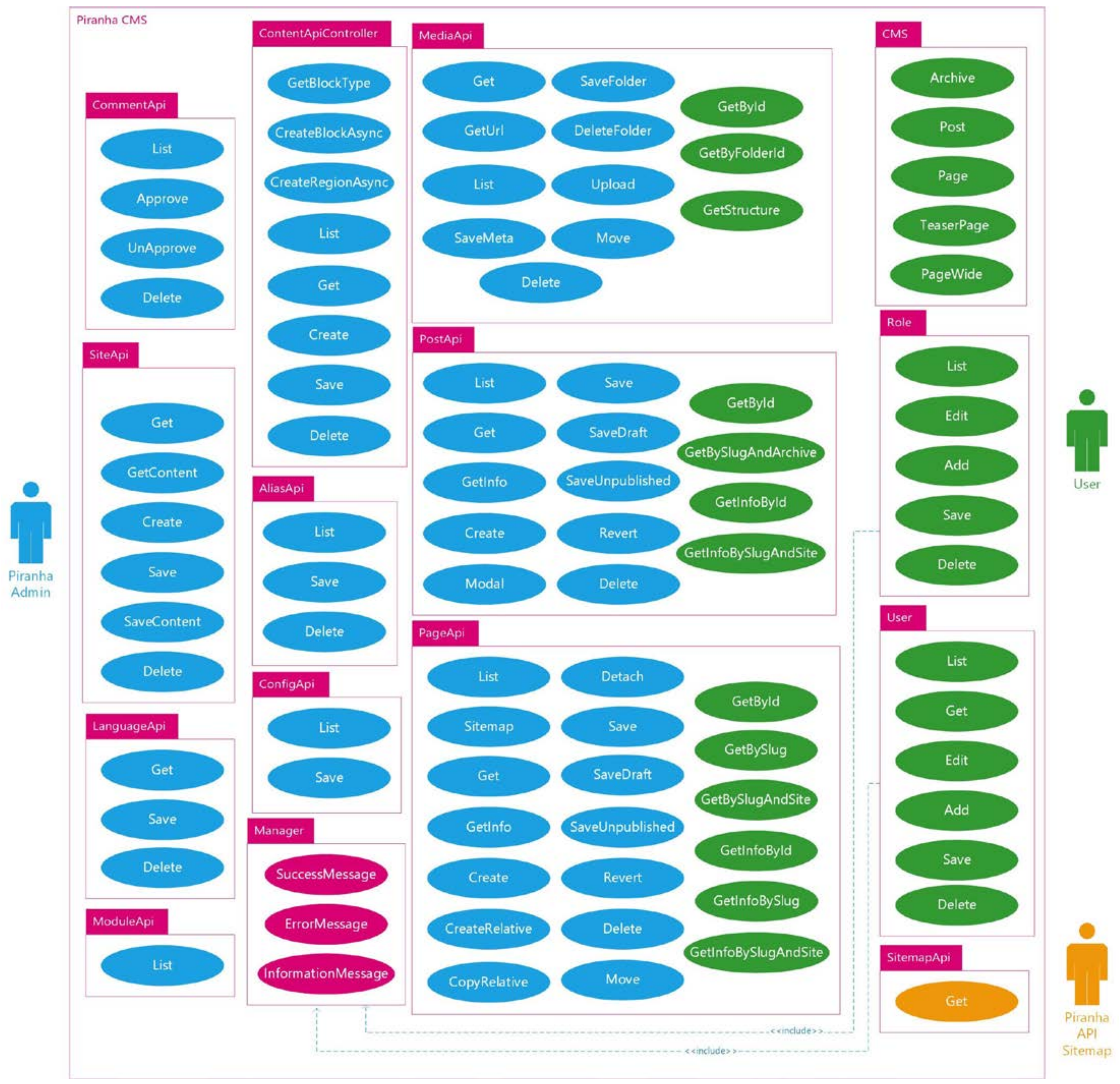


Figure 7: Recovered Use Case diagram of Piranha CMS. Actors and Use Cases in this diagram are associated based on their colors.

Since such concrete classes define different behaviors of the same operations and invocation of concrete operations can only be resolved at runtime due to dynamic dispatching, then the details of some Use Cases cannot be fully discovered by REC-MVC by relying solely on static analysis. Furthermore, different mechanisms may represent mutually exclusive or alternative functionality such that only a subset of these mechanisms are configured to be part of the running MVC web application. For instance, although both FileStorage and BlobStorage are available statically, it is possible to configure the MVC web application at deployment time to only use FileStorage.

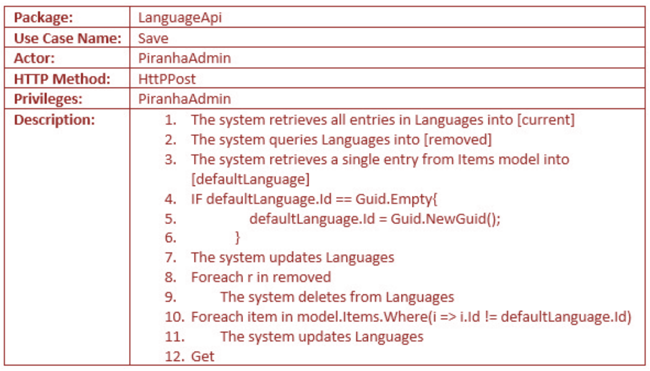
Therefore, to enhance the Use Case model recovery process, the MVC web application is instrumented to collect execution traces related to method invocations. REC-MVC then performs dynamic analysis guided by results obtained from the previous activities as follows:

1) Run the instrumented MVC web application.
2) If the recovered use case indicates an authorization privilege is needed to access the use case, log into the system as a user with the appropriate authorization privileges.
3) Execute the use case by accessing the action that corresponds to this use case.
4) Collect runtime information for this run.
5) Analyze the runtime information and generate the updated Use Case details 1.

6 Discussion and Threat to Validity

The proposed approach is based on architectural patterns such as the MVC and SOA patterns which are widely adopted in web applications. Since UML Use Case models are by nature abstract and do not involve implementation details, then recovery of such models from an architectural point of view provides an appropriate level of abstraction.

The proposed approach assumes that Use Case model concepts, such as the Include and Extend relationships, are mapped to and implemented through specific design concepts, such as the Layer Supertype pattern. Indeed, such concepts can be implemented differently at a more detailed level such as the invocation of utility class libraries. In this case, the proposed approach does not recover such concepts. Nevertheless, essential concepts such as actors and use cases can be recovered by the proposed approach since these concepts are reflected at the architectural level.



Package:	LanguageApi
Use Case Name:	Save
Actor:	PiranhaAdmin
HTTP Method:	HttPPost
Privileges:	PiranhaAdmin
Description:	1. The system retrieves all entries in Languages into [current] 2. The system queries Languages into [removed] 3. The system retrieves a single entry from Items model into [defaultLanguage] 4. IF defaultLanguage.Id == Guid.Empty{ 5. defaultLanguage.Id = Guid.NewGuid(); 6. } 7. The system updates Languages 8. Foreach r in removed 9. The system deletes from Languages 10. Foreach item in model.Items.Where(i => i.Id != defaultLanguage.Id) 11. The system updates Languages 12. Get

Figure 8: Example of recovered Use Case specification by REC-MVC.

Automatic recovery of Use Case details via static analysis of source code can be a challenging task. This is due to reasons such as complex flow of control (e.g. nested conditional statements) and complex business logic and processing. As a result, classifying nodes in the syntax tree as shown in section IV.C can be non-trivial. In addition, recovering the extended relationship as shown in section IV by statically analyzing redirects is challenging, since request redirects can be implemented at the View level or the frontend (e.g. javascript) of the web application. In such cases, the involvement of domain experts is needed to refine the recovered Use Case details.

The implementation of REC-MVC described in section V is platform-dependent since Piranha MVC is implemented using the C# language. However since the proposed approach is based on architectural- and design patterns, such concepts are platform-independent. Therefore, re-implementing REC-MVC to target other platforms (e.g. Spring MVC Framework) can be achieved with these patterns.

7 Conclusion

It has become common to construct software systems from architectural- and design patterns since these patterns promote acceptable solutions by researchers and practitioners to recurring problems that appear in specific contexts. This paper discussed a hybrid approach for recovering Use Case models for MVC web applications in which recovery patterns are defined from architectural- and design patterns that are widely used to construct web applications. Each recovery pattern shows how a specific Use Case model element is to be recovered. Static and dynamic analyses of the MVC web application are then performed according to defined recovery patterns to recover the complete Use Case model of the web application. The proposed approach has been implemented as a proof-of-concept tool, title REC-MVC, and applied to recover the Use Case model of an open-source MVC web application.

Our future work includes investigating recovery patterns in other domains such as real-time and embedded software systems which may adopt different architectural patterns, including control and master/slave patterns. Furthermore, we are interested in investigating the recovery of non-functional requirements such as security and performance, including response time and throughput, of software systems. Finally, we are interested in investigating challenges related to the recovery of Use Case models for software product lines in which products of the same family may vary in the provided functionality.

8 Availability of Data and Material

Data can be made available by contacting the corresponding author.

9 References

A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

F. U. Rehman, B. Maqbool, M. Q. Riaz, U. Qamar and M. Abbas, "Scrum software maintenance model: efficient software maintenance in Agile methodology," *2018 21st Saudi Computer Society National Computer Conference (NCC)*, Riyadh, 2018, pp. 1-5, DOI: 10.1109/NCG.2018.8593152.

A. M. Fernández-Sáez, D. Caivano, M. Genero and M. R. V. Chaudron, "On the use of UML documentation in software maintenance: Results from a survey in industry," *2015 ACM/IEEE 18th International Conference* *on Model Driven Engineering Languages and Systems (MODELS)*, Ottawa, ON, 2015, pp. 292-301, DOI: 10.1109/MODELS.2015.7338260.

L. Moreno, "Summarization of complex software artifacts," *2014 Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014),"* Association for Computing Machinery, New York, NY, USA, 654–657. DOI: 10.1145/2591062.2591096.

A. M. Fernández-Sáezez, M.R.V. Chaudron and M. Genero, "An industrialcase study on the use of UML in software maintenance and its perceivedbenefits and hurdles", *Empirical Software Engineering*, pp. 32813345,2018. DOI: 10.1007/s10664-018-9599-4.

C. S. Joanna Santos, S. Moshtari and M. Mirakhorli, "An Automated Approach to Recover the Use-case View of an Architecture," *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, Salvador, Brazil, 2020, pp. 63-66, doi: 10.1109/ICSAC50368.2020.00020.

L. Ceponiene, V. Drungilas, M. Jurgelaitis, J. Ceponis, "Method for reverse engineering UML Use Case model for websites," *Information Technology And Control*. 2018, vol. 47, no. 4, doi: 10.5755/j01.itc.47.4.21264.

E. Miranda, M. Berón, G Montejano, D. Riesco, "Using reverse engineering techniques to infer a system use case model," *Journal of Software: Evolution and Process*. 2018, vol. 31, no. 2, doi: 10.1002/smr.2121.

L. Zhang, T. Qin, Z. Zhou, D. Hao, "Identifying use cases in source code," *J Syst Softw. Journal of Systems and Software*, vol. 79., 1588-1598, 2006.

P. Dugerdil, D. Sennhauser, "Dynamic decision tree for legacy use-case recovery," *In Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*, Association for Computing Machinery, New York, NY, USA, 2013 1284–1291.

B. Ulziit, Z. A. Warraich, C. Gencel, K. Petersen, "A conceptual framework of challenges and solutions for managing global software maintenance," *J. Softw. Evol. and Proc.*, 27: 763– 792, 2015.

OMG. "The Unified Modeling Language. Documents associated with UML version 2.3," 2010. http://www.omg.org/spec/UML/2.3.

H. Gomaa, "Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures," *Cambridge: Cambridge University Press*, 2011.

Q. Li, S. Hu, P. Chen, L. Wu and W. Chen, "Discovering and Mining Use Case Model in Reverse Engineering," *4th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007)*, Haikou, 2007, pp. 431-436, DOI: 10.1109/FSKD.2007.255.

J. Bucanek, Model-View-Controller Pattern. In: Learn Objective-C for Java Developers. *Apress*, 2009.

M. Fowler. Patterns of Enterprise Application Architecture, AddisonWesley Professional, 2002.

M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, P., Security Patterns; Integrating Security and Systems Engineering, *John Wiley Sons, Inc.*, 2005, Hoboken, NJ, USA.

Dr.Emad Albassam is an Assistant Professor at the Computer Science Department, King Abdulaziz University. He is a Vice Dean for Applications at the Deanship of Information Technology. He received a B.S. degree in Computer Science from King Abdulaziz University, Saudi Arabia. He received an M.S, and a Ph.D. in Software Engineering and Information Technology with Software Engineering concentration, respectively, from George Mason University, Fairfax, Virginia. His research interests include Software Engineering, Autonomous Software Systems, Software Product Lines, and Video Game Development.