IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 8 Issue 8, August 2021
ISSN (Online) 2348 – 7968 | Impact Factor (2020) – 6.72
www.ijiset.com

# On the Effects from Enforcing Standardization Through Abstractions in MCV

**Nikola Valchanov**[1]

[1] Chair of Software Engineering, Plovdiv University "Paisii Hilendarski",
Plovdiv, 4000, Bulgaria

**Abstract**

This paper discusses the benefits of enforcing standardization through abstractions when designing information systems. It is focused on known effects such as code quality and maintainability improvements and how those are reflected by standard quality metrics. In addition, it discusses how modern techniques simplify and speed up feature development and at the same time allow the introduction of a wide range of optimizations in the development process.

The paper describes some of those development optimizations and analyses their effectiveness when used for the development of document management systems.

***Keywords:*** *Design Patterns, Information Systems, Document Management Systems, Code Structure.*

## 1. Introduction

One of the main concerns of the modern software industry is cost effectiveness. Although cost effectiveness comes in different shapes and sizes, the best way to achieve it is either process simplification through decoupling complex problems into repetitive tasks of similar origin or partial/complete automation. This enables engineers to focus on architecture, system design and customizations leaving all standard features either completely generated or implemented by more inexperienced developers.

Contemporary software development frameworks and code generation platforms [1] have results in the field of standardization, abstraction, and automation [2]. The issue is that those platforms come with a tradeoff. They either are simple and straight forward, but do not cover complex scenarios, or they present a whole framework of interconnected features that can't be used individually thus introducing severe complexity overhead.

This paper discusses effects from enforcing standardization through abstractions such as simplified development process, cost effectiveness, and uncovering opportunity for introduction of further automation of the development process.

## 2. Common Code Reducing Abstractions

Two of the most common abstractions in modern software development that lay the foundations of most modern products are:

1. Data access layer abstractions
2. Service layer abstraction

### 2.1 Data access layer abstractions

Let's take this oversimplified definition of a data access abstraction layer – it is a construct or a library of constructs that isolates the implementation of data management (creating records, reading records, updating records, deleting records) from the rest of the codebase in such a way that other layers use the implemented functionality through the API the construct/library exposes where the API doesn't suggest in any way how or where data is being persisted.

The modern data access layers are very often based on object-to-relation mapping (ORM) libraries [3]. Those ORM libraries fall into the above definition thus themselves implement data access layer abstractions.

One of the main reasons why the encapsulation of ORM libraries using the Repository pattern [4, 5] is considered a good practice is due to evolutionary processes. While in the dawn of data source connectors each persistence platform supported their own set of connectors and the choice of the connector often came with the platform, Today ORM libraries support different ranges of persistence platforms. To go a step further due to the dynamics in the IT business such libraries are born,

gain maturity and decline in the matter of years. Adding this custom pattern-based layer between the codebase and the ORM ensures that the ORM library can be replaced relatively painless. And while historically such patterns existed and were abstracting the connectors, today those patterns are adapted to the structure and functionality of ORM libraries.
The foundations of ORM libraries are [5]:

- Models - entity classes that model the data that is to be persisted
- Mapping - descriptors that map the structure of the model to the structure of the persistence platform

Models are simple classes that contain only characteristics and no behavior. Based on the type of system the common characteristics of the models are usually abstracted in a base class. For the purposes of this paper, we'll consider an oversimplified case where only the identity field is extracted in the abstraction.

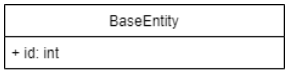


Fig. 1 Entity abstraction

There is a requirement for a basic tooling that supports mapping within the language/platform in use to identify entity members that are to be mapped and manipulate them using static descriptors loaded either at runtime or assigned at compile-time. This feature comes natively with some language (ex: JavaScript, PHP, etc.) and/or comes in the form of a platform feature (Java, .NET, etc.) such as Type Reflection.
Let's consider two of the most common software development platforms in the enterprise world – Java (Criteria API) [6] and .NET (EntityFramework) [7]. In both cases we can achieve passing the type of entity that is to be manipulated as a parameter and instantiating the ORM in the context of this specific entity type. The most common way of implementing this is through the Repository pattern [8, 9] by extracting the common repository operations in an abstract base class.
In the diagram below the type T is supplied as a parameter and denotes the type of entity that is to be managed by the ORM library.

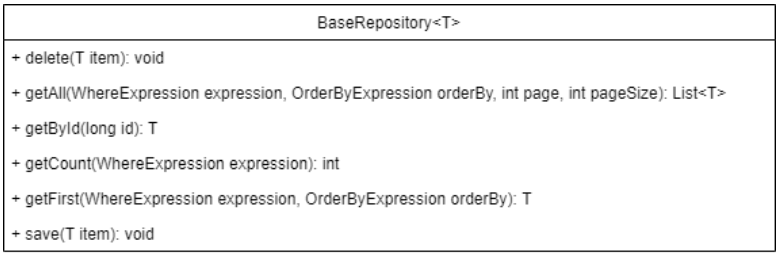


Fig. 2 Abstracting the base behavior of classic Repository pattern implementation

Enforcing the BaseRepository onto all classes in the data access layer effectively places a structure that is followed within the codebase and later enables other sections of the code to support further layers of abstraction.

## 2.2 Service layer abstraction

In practice the Repository layer is often omitted and implemented directly into the services layer. The reason is that the abstraction implementation is essentially the same and the additional business logic that needs to be injected can be implemented either with abstract methods that are called in the base class and implemented in the descendants or by exposing new methods.
Similar to the data access layer abstraction the diagram below the type T is supplied as a parameter and denotes the type of entity that is to be managed within the service. A service's scope should be limited to operations over a specific entity. Those operations could involve extracting and processing additional data from the data store, but essentially should return only information related to the entity in-scope.


Fig. 3 Abstracting the base behavior of Service implementation

The above diagram shows the case where Repository layer and Services layer are merged. In this case the only type parameter that is supplied is T that represents the type of Entity that is managed by the service class.


Fig. 4 Abstracting the base behavior of Service implementation with Repositories

When Services are built on top of Repository layer implementation then the abstraction needs to be parametrized, so any implementations of the abstraction supply the specific type of Repository that is to be used within the service. This way all descendants can have strongly typed methods, leverage compile time support and autocomplete. Different languages support different ways of additionally enforcing dependencies between the supplied type parameters. Example could be C# and Java where generic parameters can be used to define the type of other generic parameters within the same class.
The Service abstraction not only adds an additional business logic layer that can be leveraged to support complex behavior but additionally simplifies support as it adds additional point in the application that can be mocked, and unit tested [10].

## 2.3 MVC Controller abstraction

To complete the three most basic layers in an application we need to consider placing an abstraction within the layer that communicates with either the end-user or the user agent system. In frameworks based on the MVC design pattern the place for this abstraction is within the hierarchy of classes that form the Controllers.
In modern MVC based software framework, controllers expose endpoints that can be consumed via HTTP protocol. Those endpoints are sometimes annotated with descriptors, but in most cases come in the form of methods of the classes that implement controller logic.
Let's consider the following as basic behavior for user/user agent interaction in document management systems:

- Extract all records
- Load a record for modifications
- Create record
- Modify record
- Delete record

The list above can be applied to both set of endpoints for programmatic consumption (client-side rendering, mobile applications, automations, etc.) and a such that use server-side rendering (Razor, Thymeleaf, etc.).
One of the industry's most common practices associated with the MVC pattern is utilizing View Models for managing parameters, validation, and content generation.
To support the operations listed above, the following View Models are needed:

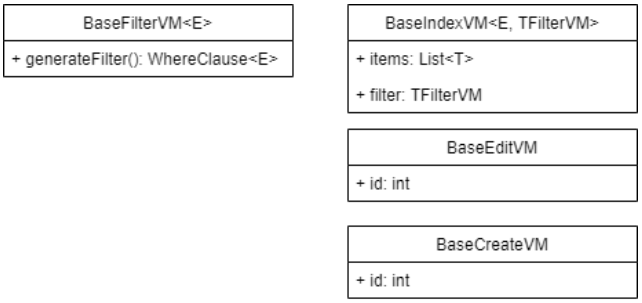


Fig. 5 View Model abstractions

The combination of the items described above allows the definition of a controller abstraction that combines all basic operations on entities.

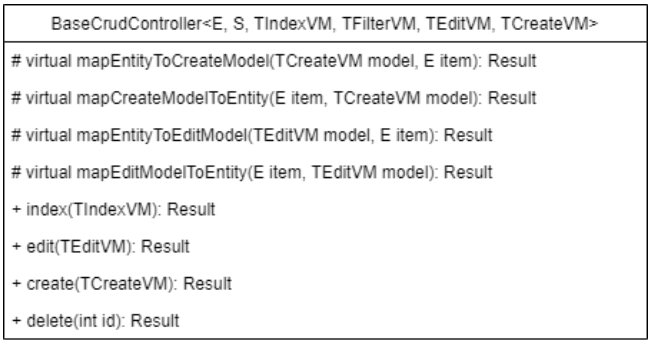


Fig. 6 Abstracting the base CRUD behavior of Controller implementation

To understand better the diagram above let's consider the separate components.

**Listing records**

To extract and return a list of record with the ability to filter data we need the ability to work with instances of:

- Entity class
- Service class that manages the specified entity
- Filter view model instance that can generate the filter descriptor at runtime

The above is achieved by several stacked pre-conditions.

First the abstract CRUD controller is expecting a type of parameter for the entity type that is to be managed by the controller, allowing only entity types that derive from BaseEntity thus ensuring the type has an "Id" field that is used when extracting single records by id.

Second the abstract CRUD controller is expecting as parameter the type of Service class that will be used for the management of the entity type "E". This parameter is limited to descendants of the abstract service class, thus within the base CRUD controller all base service features are available.

Third is the Filter view. While in the abstract CRUD controller the type of entity that is subjected to filtering is unknown, a strongly typed filter view model type can be supplied that is able to generate filter for the specific entity at hand. Although the implementation remains unknown at the level of the abstraction, the filter view model's ability to generate filter descriptor can be leveraged when building the result set with the list of records that are to be returned.

**Editing and Creating records**

The "E" parameter is defined as a class that derives from BaseEntity thus the Id property can be used within the abstract CRUD controller for identifying the item that is to be edited. In addition to that, Edit and Create features need only the ability to map data from the view model to the entity and back which can be achieved by either implementation of 1:1 mapping of properties with matching names using Type Reflection (in the abstraction as default implementation) or custom implementation of the virtual methods in the diagram above.

**Deleting records**

The delete feature need only the Id of the record that is to be deleted, which is available as “E” needs to be a class that derives from BaseEntity.

## 3. Affected metrics

In most of the cases the pattern above is applied in languages that support Generics. Since at compile time generics generate separate classes for each use of the generic class within the code, complexity metrics [11] are not affected as during this process new classes with fully functional methods are generated.
In the simplest case where Controllers need to implement basic create, read, update and delete functionality the rates are as follows:

- Services / Repositories - ~86% code reduction per class
  (from ~110 lines of code to ~15 lines of code)
- Controllers - ~85% code reduction per class
  (from ~120 lines of code to ~18 lines of code)
- Controllers with custom mapping- ~70% code reduction per class
  (from ~120 lines of code to ~36 lines of code)

The classes for supporting new controllers are done based on a standard template dictated by the structure of the entity in focus, thus can be built by developers with less experience ultimately achieving both significant reduction in the needed seniority and total lines of code written per feature.

## 4. Opportunities from enforcing structure

The model above leads to opportunities with optimization and automation of the development process. The standardization of the classes needed for feature implementation leads to the following:

**Rapid development**

Extracting abstractions on all levels (data access, services, controllers) result in reduced development time. After implementation of the Entity class the Repository, Service and Controller simply derives from the base with no additional code.
The View Model classes follow the same template where the only places where customizations are needed is the Filter view model and Create/Edit view models. Although those three classes need customizations, the Create/Edit view models are copies of the Entity class and the only customization needed is validation descriptors (based on the MVC framework in use). The filter is of the exact same format and follows a standard template as per the example (in C#) below:

```
public class FilterVM : BaseFilterVM<User>
{
  [DisplayName("Username: ")]
  public string Username { get; set; }
  [DisplayName("First Name: ")]
  public string FirstName { get; set; }
  [DisplayName("Last Name: ")]
  public string LastName { get; set; }
  [DisplayName("Email: ")]
  public string Email { get; set; }

  public override Expression<Func<User, bool>> GetFilter()
  {
    return i =>
      (string.IsNullOrEmpty(Username) || i.Username.Contains(Username)) &&
      (string.IsNullOrEmpty(FirstName) || i.FirstName.Contains(FirstName)) &&
      (string.IsNullOrEmpty(LastName) || i.LastName.Contains(LastName)) &&
      (string.IsNullOrEmpty(Email) || i.Email.Contains(Email));
  }
}
```

The filter expression above is of C# syntax, but same filter descriptor can be built via similar tools in the different languages (ex: Java's Criteria API).

The above reduces the cost of bootstrapping a new basic feature both in terms of hours spent and level of seniority needed for the implementation.

**Code generation**

If we analyze the results from utilizing the described abstractions, it is evident that the only difference the controller implementations (including entities, view models, repository, service, and controller) is reduced to:

- Structure of the Entity
- Structure of the Filter, Create and Edit view models
- Methods for mapping data between models and entities

Considering the implementation of the items in the list above it is visible that the only difference comes in the form of a list of the properties of the Entity in scope. This means that all code for implementing CRUD controllers can be organized in a very simple standard template that can be used for generation of the complete entity management feature.

The required input for generation of a single entity is a set of static descriptors that frame the structure of entity with property types and validation rules. Any information regarding entity relationships can be further leveraged in the generation process and reflected in resulting codebase.

**Customizations**

The abstractions described above are over simplified. If used in commercial environment the patterns need to provide means to enrich the behavior, implemented in the base classes.

A very simple scenario would be adding abstract methods before and after each one of the basic operations as per the diagram below:

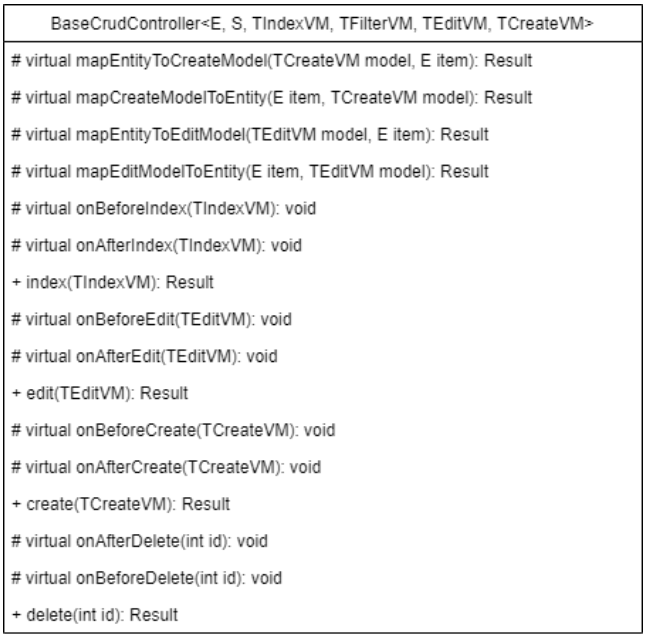


Fig. 7 Adding ability to customize behavior of base features of CRUD Controllers

The utilization of the above is obvious for standard development done by software engineers. Customizations can be implemented via overriding the virtual methods to add/modify data in view models or perform operations based on information from the view models prior to the execution of the base operation.
Business logic customizations when using code generation techniques is not that straight-forward. The two most obvious approaches to solving this are:

- Building an action-based framework
- Injecting code directly into the template

Building an action-based framework would mean that for each transformation or data manipulation the code generation platform builds a separate library that implements the set of actions/transformations on the model that are to be completed as part of the customization. This could be a single library (action) or a set of libraries that are to be executed in a particular order. The libraries are loaded at runtime and executed in the configured order. The benefit from this approach is that each customization can be reviewed separately by the code generation platform, actions can be organized in a common exchange, shared and easily versioned. This way non-technical personnel can use actions from a common exchange that complete specific tasks like for example sending an email or calling an API (storing a record in an external system like Salesforce).
Injecting code directly into the template is the simpler approach where the code generation tool adds the customization code in the respective virtual method. Although simpler this approach operates on snippets, rather than encapsulated plugins. This complicates the maintenance of versions, validation, and general use by non-technical personnel.

## 5. Conclusion

The analysis shows that the suggested abstractions reduce the development time for basic entity management features. In addition, they improve code organization, testability and reduce maintenance cost.
The described model allows the creation of templates which can be used for code generation with the ability to generate well features with well-organized codebase, based on best practices. The model also allows customizations both in classic software development and code generation scenarios.

## References

[1] G. Paolone, M. Marinelli, R. Paesani, P. Felice, Automatic Code Generation of MVC Web Applications, Computers Vol., 9, No. 3, 2020, pp. 56-85.
[2] D. Khelladi, B. Combemale, M. Acher, O. Barais, On the power of abstraction: a model-driven co-evolution approach of software code, in ICSE-NIER '20, 2020, pp. 85–88.
[3] M. Gorodnichev, M. Moseva, K. Poly, K. Dzhabrailov, R. Gematudinov, Exploring Object-Relational Mapping (ORM) Systems And How To Effectively Program A Data Access Model, PJAEE, Vol. 17, No. 3, 2020, pp. 615-27.
[4] B. Gorman, Repository and Unit of Work Patterns, Apress, 2020.
[5] J. Griffin, DTOs, Entities, and Value Objects, Apress, 2020.
[6] M. Keith, M. Schincariol, M. Nardone, Pro JPA 2 in Java EE 8, Apress, 2018.
[7] B. Gorman, Practical Entity Framework, Apress, 2020.
[8] https://docs.microsoft.com/en-us/dotnet/architecture/microservices/ microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design, 2018.
[9] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2012.
[10] S. Kapferer, O. Zimmermann, Domain-driven service design - context modeling, model refactoring and contract generation, SummerSOC 2020 CCIS, Springer, 2020, Vol. 1310, pp. 189–208.
[11] O. Masmali, O. Badreddin, Code Quality Metrics Derived from Software Design, in ICSEA 2020, 2020, pp. 141-150.
[12] A. Golev, Textbook on algorithms and programs in C#, University Press “Paisii Hilendarski”, 2012.
[13] A. Iliev, N. Kyurkchiev, A. Golev, A Note on Knuth’s Implementation of Extended Euclidean Greatest Common Divisor Algorithm, International Journal of Pure and Applied Mathematics, Vol. 118, 2018, pp. 31–37.
[14] G. Bogdanov, N. Pavlov, A. Rahnev, GENERATING REPORT DOCUMENTS FROM TEST ANYTHING PROTOCOL IN JAVASCRIPT, in Innovative ICT in Research and Education: Mathematics, Informatics and Information Technologies, 2018, pp. 55-64.
[15] N. Kyurkchiev, A. Iliev, A. Golev, A. Rahnev, Some Non-standard Models in "Debugging and Test Theory", Plovdiv University Press, 2020.
[16] O. Rahneva, A. Golev, G. Spasov, Investigations on Some New Models in Debugging and Growth Theory, LAP LAMBERT Academic Publishing, 2020.
[17] A. Rahnev, N. Pavlov, N. Valchanov, T. Terzieva, Object Oriented Programming, Lightning Source UK Ltd., 2013.