Tony Marston's Blog About software development, PHP and OOP

Home About Me PHP/MySQL

Why I don't do Domain Driven Design

Posted on 1st March 2018 **by Tony Marston**

Amended on 24th January 2019

Introduction

What is Domain Driven Design (DDD)? If you read this Wikipedia article you will see that it talks about DDD consisting of a number of high-level concepts and practices. It refers to a ubiquitous language meaning that the domain model should form a common language given by domain experts for describing system requirements. It talks about breaking the domain into its entities, value objects, aggregates, events, services, repositories and factories. The primary focus of this process is the domain logic, the business rules for that domain. Any logic which is not specific to a particular domain, such as communicating with the database or the user interface, is left until last as it is considered to be nothing more than an implementation detail.

I have been designing and building enterprise applications for nearly 4 decades, and this is NOT the way that I do it. I analyse the business requirements in order to produce a logical design which identifies what business functions are needed, what front-end screens will be required, and what the database design should contain. Apart from identifying *what* business logic needs to sit between the screen and the database I do not bother with the *how* until the design has been approved and the implementation has started. To me it is the business logic which is the implementation detail and which can be left until last. Once I have designed the database I create a separate class for each table, then I build a user transaction for each operation which needs to be performed from a standard template. That standard template will provide the basic functionality so that I can run the transaction and communicate with the database, and I add in the specific business rules afterwards by filling the relevant hook methods with the necessary code.

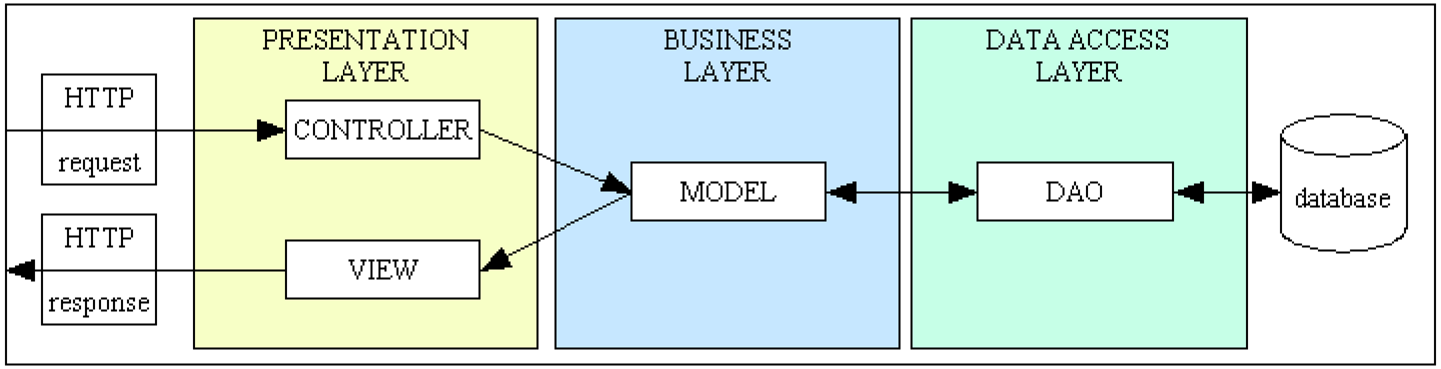
It was not until 2002 that I started using a language that had OO capabilities, and at that time the notions of Object Oriented Design (OOD) and Domain Driven Design (DDD) had either not yet been formulated, or were in their infancy and were not widely publicised nor generally accepted in the programming community. All that I new was that Object Oriented Programming (OOP) was a programming technique that was similar to procedural programming but with the addition of encapsulation, inheritance and polymorphism which could be used to increase code reuse and decrease code maintenance. I say *could* because the use of OO techniques does not automatically guarantee that the code will be *better*, the only guarantee is that it will be *different*. The result of using OOP is still down to the skill of the individual programmer. If you read any articles on how to make effective use of these three fundamental concepts you will see such ideas as coupling, cohesion and dependency being mentioned. These ideas existed in other languages before OOP was invented, so they are not unique to OOP. OOP itself is not a programming technique which is totally different from what came before, it uses the same basic principles, but adds encapsulation, inheritance and polymorphism into the mix. This is discussed in more detail in What is the difference between Procedural and OO programming?

The biggest differences with the most similarities

When a novice programmer first examines different domains within an enterprise application a casual glance will immediately highlight the differences - the inputs are different, the outputs are different, the database is different and the business rules are different. That novice will then assume that because each domain is so different from the others that it will require a totally different design and therefore totally different code to implement that design. However, when you have built as many database applications as I have you will quickly learn that the differences actually hide a number of similarities which provide opportunities for significant amounts of reusable and sharable code. An experienced programmer will tell you that the more reusable code you have then the less code you have to write. This leads to shorter development timescales, lower costs, quicker time-to-market (TTM) and easier maintenance.

But where are these similarities I hear you ask? Where are the opportunities for reusable code? In my current ERP application I have personally designed and built nearly 20 domains, which I call "subsystems", using a development framework which I specifically designed for this purpose. If you read Anatomy of an Enterprise Application you will see that, regardless of the domain, each user transaction follows the same pattern - there is a User Interface (UI) at the front end, a database at the back end, and software in the middle which transfers data between the two, and which applies the business rules. A common way to deal with these three areas is to use an architectural pattern known as the 3-Tier Architecture (3TA) which has a Presentation layer for the User Interface (UI), a Business layer (sometimes known as the Domain layer) for all the business/domain logic, and a Data Access layer for communicating with the physical database. Another common pattern is the Model-View-Controller (MVC) which I have merged with 3TA by splitting the Presentation layer into a Controller and a View, with the Model being in the Business layer. This produces an application architecture which is shown in Figure 1:

Figure 1 - MVC plus 3 Tier Architecture



The following areas have similar characteristics which provide opportunities for reusable code:

- **Inputs**. In a web application all inputs come from an HTML form as an HTTP request which uses either the GET or POST method. In the PHP language the data for each of these methods is presented to the code as an associative array. It does not matter what data is contained in this array, it is a standard structure which can be accessed and manipulated using standard code. The HTTP request is passed to an initial component script which sends the request to a Controller which then translates the request into specific method calls on a nominated Model (or Models). When the Model has finished processing the request, which may or may not involve communicating the Data Access Object, it gives its raw data to the View which then produces the output which is returned to the user.
- **Outputs**. The most common form of output in a web application is an HTML form, and the only difference between one HTML form and another is its structure and/or its content. In a large application with thousands of forms you may be surprised to know that the number of different structures is actually quite small. I have produced a library of just over a dozen XSL stylesheets which identify the possible structures, and by using common code to put the application data into an XML document I can then perform an XSL Transformation which

produces the HTML output. This is all handled by a single View component which is aided by a small screen structure script which does nothing but identify which piece of data goes where.

I have a similar process to produce PDF output. I have a single PDF component which is aided by a small report structure script which does nothing but identify which piece of data goes where.

Creating CSV output is the easiest - simply extract the array of data from the Model component and write it to a disk file. I do not have to know what data this array contains, so I can extract different arrays of data from any number of different objects and the process of creating a CSV file from this data is exactly the same.

- **Databases**. While it is true that every different domain uses totally different sets of data, when it is stored in a relational database this data is contained in objects called tables. While each table has a unique name and a unique structure in the form of a particular collection of columns, this is where the differences end. Each column has a data type which comes from a restricted list, a size which can be varied, and a NULL or NOT NULL attribute which indicates whether it is optional or required. Regardless of its structure, every table in the database is accessed by the same set of SQL operations which are Create, Read, Update and Delete, or CRUD for short. Each of these operations is constructed as a query string which has a standard format, therefore can quite easily be handled in a sharable Data Access Object. I have a separate version of this object for each DBMS that I support.

Another common mistake that a novice programmer often makes is to consider each task in a domain as a series of business rules which incidentally touch the database. I learned a long time ago that the best approach is to take the opposite view - each task is mainly concerned with accessing one or more database tables using one or more of the standard CRUD operations, and only incidentally executing the business rules. This is why I always start with the standard code and add in the business rules later.

When you look at the four types of component in Figure 1 you should now be able to see how I provide them in my code:

- **Controllers** - I have one pre-written and sharable controller for each of the Transaction Patterns which I have recognised.
- **Views** - I have one pre-written and sharable object for each of the three types of output - HTML, PDF and CSV.
- **Models** - These are constructed separately once the domain database has been built, one class per table. This process has been automated in that the class file, which inherits large volumes of standard code from a single abstract table class, and structure file are produced by operations within my Data Dictionary.
- **Data Access Object** - I have one pre-written and sharable object for each DBMS that I support.

Each user transaction - and there may be hundreds, or even thousands - will therefore contain a mixture of code which falls into one of the following categories:

- **Domain-agnostic**. This is code which can be pre-written and then shared with any component within any business domain. An example would be the generation and execution of SQL queries, or the generation of HTML pages using a templating engine.

- **Domain-specific**. This is code which needs to be written specifically for a particular business domain in order to execute the business rules required by that domain. The business layer will contain a number of different domain objects, and each object will contain it own subset of business logic.

I have been building enterprise applications for nearly 4 decades, and my approach has always been to start building the components around the domain-agnostic logic and insert the domain-specific logic later. Why? Because in my experience the volume of domain-agnostic logic is far greater than the volume of domain-specific logic, and as there are large volumes of domain-agnostic logic which can be provided in reusable and sharable modules I have found it easier to build each user transaction by starting with the domain-agnostic logic and adding in the domain-specific logic later. Neither the front-end Presentation layer nor the back-end Data Access layer should contain any domain-specific logic, and the middle Business/Domain layer will always contain a mixture of standard domain-agnostic logic and non-standard domain-specific logic. With my technique I can build a raw user transaction from a template and add in the domain-specific logic afterwards at a faster rate than you can by starting with the domain-specific logic and adding in the domain-agnostic logic later.

In an enterprise application each user transaction has some sort of interaction with the database which, regardless of the table and the data it contains, follows a familiar and regular pattern. Each HTML document, while superficially different, has enough in common with other documents to enable it to be built from one of a small number of reusable XSL Stylesheets which implement a series of reusable modules called templates. Each Controller, which translates user requests into actions on a particular Model, can use the same pattern of actions on any number of different Models, so can be provided as a library of pre-written and reusable components.

The key to successful software development is the ability to spot these recurring templates and patterns, then to provide them with as much pre-written and reusable code as possible. This is the basis on which I developed my series of development frameworks. As far back as 1985 I built my first development framework in COBOL to provide as much domain-agnostic and sharable code as possible. In the 1990s I rewrote this in the UNIFACE language, and in 2003 I rewrote it again specifically for PHP. With each rewrite the additional capabilities of each new language allowed me to build additional features into my development framework and also to provide more reusable components as well as shorter development times. My latest PHP framework allows me to generate working user transactions which automatically contain all the necessary domain-agnostic code. The developer can then add in the necessary domain-specific code without having to touch the standard code which is provided by the framework. This is an easy job as the abstract table class, from which every concrete table class inherits, contains a series of hook methods specifically for this purpose.

In the introduction I identified the various components which are supposedly "required" when implementing DDD, but I have found a use for only some of them:

- **Entities** - these are implemented as Model classes with a separate class for each table in the database. Each of these classes inherits standard domain-agnostic code from an abstract table class.
- **Services** - these are pre-written and provided by the framework. They include Views and Controllers. There is a separate View for each output format - HTML, PDF or CSV. There is a separate Controller for each Transaction Pattern.
- **Events** - an event is something that happens in the real world which then has to be recorded in the application database. Each event is implemented as a user transaction which identifies a particular unit of work which will cause the database to be updated in the appropriate manner. Each has its own entry on the MNU_TASK table in the framework database. This data is then referenced in the MNU_MENU and MNU_NAV_BUTTON tables to provide clickable links to activate each task. The ROLE_TASK table is used to identify which tasks can be accessed by each USER_ROLE which in turn controls the links which are visible to each user. The task then points to a component script in the file system which identifies which action is to be performed on which object in the business/domain layer.
- **Repositories** - these are implemented as Data Access Objects with a separate DAO class for each supported DBMS. It is possible to switch an application from one DBMS to another simply by changing a single line in the configuration file.
- **Value objects** - PHP does not support them, so I don't use them.
- **Aggregates** - An aggregate is a collection of objects which are bound together by a root entity. These do not exist in any DBMS that I have used, so to put them in my software would introduce an artificial concept. Each table is a stand-alone object in the database, with its own structure, so in my software I create a stand-alone object in the domain/business layer to deal with that structure and apply its business rules.
- **Factories** - I have no need for any of these as each Model class has only a single configuration, and that configuration, which involves turning an abstract class into a concrete class, is performed within the class constructor.

For those of you who do not understand the difference between an *entity* and a *service*, an entity has *state* or data which can be loaded, changed and extracted over time. An example is a database record which has its current values extracted from the database, displayed in the UI where changes may be made and posted back, which causes the changed values to be updated in the database. A service on the other hand is *stateless* - data goes in, is processed in some way, then immediately spat out without hanging around for later extraction or manipulation. An example of a service is an XSL Transformation which, after having transformed an XML document into an HTML web page, simply dies as there is nothing else left to do. Domain logic should only exist within an entity. A service should only contain logic which is required for the provision of that service and which is not specific to any domain.

Which has priority, the software or the database?

Too many of today's programmers are taught to design their applications by starting with the software, which sits between the UI and the database, and leave the database till last as it is considered to be nothing more than "an implementation detail". In my early programming days I worked with badly structured software and badly designed databases, but I saw the light when I was taught that for a database application to be really successful the database must be properly designed by following the rules of Data Normalisation and, courtesy of a course on Jackson Structured Programming, that the software structure should be designed to follow the database structure as closely as possible. I personally witnessed the improvement that this combination of ideas made to both the initial software development and its subsequent maintenance, so I have absolutely no intention of turning my back on a winning formula. Just because someone has developed a different approach does not mean that I should follow it.

To me the reverse is true - the design of the database takes preference, and I build user transactions based primarily on the interaction they have with the database. The execution of business rules is totally secondary and can be plugged in later. It does not matter that the data and business rules for each business domain are totally different when the common factor across all domains is that the data is stored in the database as tables and columns which are manipulated with SQL queries using just 4 basic operations - Create, Read Update and Delete. Each table is treated as a separate object in the database, so I see no reason why each table should not have its own class/object in the application software. I particularly avoid the idea of object aggregation where a single object in the software is responsible for multiple objects (tables) in the database. In the real world the concept of an ORDER may exist as a single entity, but when it appears in the database after normalisation it is split into separate ORDER_HEADER and ORDER_LINE objects. Each of these is a separate object with a separate structure and business rules, so it deserves its own separate table class in the software. All the code which is common to every database table I put into an abstract class. Each table has its own name and collection of columns, so I specify these differences in the class constructor which inherits from the non-specific abstract class and turns it into a specific concrete class. While the collection of columns in a table's structure may be unique, their data types always come from a predefined list, so the validation of user input which checks that the value for a column matches that column's data type can also be predefined. Each table has its own primary key, optional unique keys and relationships with other tables, so these can also be identified in the class constructor and handled by code which is inherited from the abstract class.

Build a library of reusable components

When I started my programming career with COBOL in the 1970s a common complaint was that the language was so verbose with its numerous divisions (identification division, environment division, data division and procedure division) that it took a long time to write even a simple program. Then a person much older and wiser than me pointed out a simple fact - a good programmer need only ever write *one* program from scratch, after which he could copy this code and amend it as necessary to create a different program. As COBOL was a language designed for business applications, and all these applications used some sort of database, there was a great deal of code which moved the data from the database to the screen and from the screen to the database by constructing and executing queries. Note that SQL did not exist at that time, so a "query" was actually a hard-coded function call that used the non-relational database that was provided with the operating system. The differences between one program and the next were then limited to the name of the database table and the names of its columns, plus the business rules which were unique to that table. My goal then became to write code in such a way that it was easy to identify that which was similar from that which was different so that I could minimise the changes necessary to make the code work with a different table and its different set of columns. Note that I use the word "similar" and not "identical". Each program may have a block of code which moves data from the database to the screen, and another block which moves data from the screen to the database, but each block contains a series of `MOVE source TO destination` statements where the "source" and "destination" names were hard-coded and could not be parameterised.

All I could do initially was to put identical code into a shared library so that the library function could be called instead of the code within that function being duplicated. I also began to write library functions which dealt with "similar" conditions by passing the differences as arguments so that the function body could be kept as static as possible. The results of this work is identified in Library of Standard Utilities on my COBOL page.

Turning a library into a framework

As explained in What is a Framework? you have to write your own code in order to call a library routine, whereas a framework contains components which call your code. A framework should also allow you to create application components by writing less code.

I converted my library into a framework while working on a project in 1985. Up until that point the practice had been to hard-code each of the menu screens so that the user could see a list of options and choose one to be executed. A menu had to fit within the screen's dimensions (no scrolling was allowed), so each menu page had to be written as a separate subprogram. Note the difference between a "program" and a "subprogram" - a program can only be activated from the command line, while a subprogram can only be called from within a program or subprogram. The starting point for each application was always a logon screen which accepted a user ID and password, and from then on it passed control to the top-level menu where it waited for the user to choose which option to run next. Each application therefore had its own copy of the logon screen, its own set of hard-coded menu screens, plus its own set of application components. Not all users had access to all components, so there was a primitive mechanism which identified who could access what. Unfortunately this condition could only be evaluated *after* the component was loaded, so while all users could see all options on the menu screen, they sometimes selected an option only to be told "You don't have access to this option".

This all changed when the customer made a simple request: "If a user does not have access to an option, then that option should not appear on the menu screen". This meant that I had to change from using *static* menus to a system of *dynamic* menus. The request was made on a Friday, so I spent a couple of hours over the weekend in designing a solution and started implementing it on the Monday. By the following Friday the implementation was complete. The solution involved the creation of a new set of database tables plus the programs to maintain them. These are documented in the Menu and Security System User Manual as follows:

- The M-TRAN table identified every option available in the application.
- The M-USER table identified every person who was allowed to access the application.

- The D-TRAN-USER table identified which options each user was allowed to access.
- The D-MENU table identified which option was available on which menu screen.

This arrangement allowed menu screens to be built or modified by changing the contents of a few database tables without having to change any program code. When a menu option was selected the contents of the D-MENU table was read, and any options which were not on the D-TRAN-USER table for that user were filtered out. This meant that the menu screen never displayed an option which the user could not select. This also removed the need to have any security checking within any subprograms as such checking was now performed by the framework *before* the subprogram was called instead of within the subprogram *after* it was called. This software became a framework by virtue of the fact that it provided a standard set of pre-written code which could be reused in any application without having to be redesigned and rewritten each time. The programmer no longer had to write any code to call his application components as the standard code did this all for him. This meant that the programmer could concentrate on writing the application components and not waste time on the code which called those components.

This database structure was carried forward when I redeveloped the framework in each new language, first UNIFACE in the 1990s and then PHP in 2003. Each of these languages came with its own set of new facilities which either allowed more options to be provided, or allowed more code to be transferred to reusable functions. The results of the rewrite in PHP have proved to be very successful as I am now able to build application components at a much faster rate and with more features built in as standard than I could with its previous incarnations.

The challenge with PHP was that it was the first language I had used which had Object Oriented (OO) capabilities, so I had to learn how to use these things called Encapsulation, Inheritance and Polymorphism to maximum advantage. According to some people my implementation of these principles is totally wrong, but their arguments are futile when you consider that my results are superior to theirs.

The steps I took to build the components to maintain the first database table in my framework are detailed in Building the first components. This resulted in the creation of 1 Model class and 6 Controllers.

I then copied these 7 scripts and renamed them to perform the same actions on a second database table, as described in Duplicating and Refactoring components. As this resulted in a huge amount of duplicated code I then went through a refactoring exercise in order to put as much common code as possible into reusable modules. For the two Model classes I created an abstract table class for the common code which could be inherited, then moved code from the table class into the abstract class. When I was finished I had nothing left in each table class except the constructor which did nothing but supply the table name and its structure.

Then I turned my attention to the Controllers. Instead of having a separate Controller which contained hard-coded references to a particular Model class I found a way to make them work with a class name that was not supplied until run-time. This mechanism is called a component script, and there is one of these for every user transaction in the application. If you study my implementation carefully you should see that each Controller is dependent on a Model (sometimes two or three models), but rather than this dependency being hard-coded it is in fact injected at run-time, thus demonstrating a form of Dependency Injection. Because the methods called by each Controller are defined within the abstract table class which is inherited by every concrete table class this means that each Controller is capable of calling the same methods on each of the 400+ table classes in my enterprise application, thus demonstrating one of the goals of OOP which is called Polymorphism.

Over time I discovered that some user transactions were becoming more complicated than could be covered by my original 6 Controllers, so I gradually added to their number until I now have over 40 of them which are documented in Transaction Patterns for Web Applications

Use the framework to build components

Initially I had to generate the scripts for new table classes and user transactions by hand, but as all these followed a standard pattern I decided to automate them as much as possible. To handle the class files I followed the steps described in Generating class files. To handle the user transactions I followed the steps described in Generating user transactions. This then meant that I could create a new table in my database, then generate *and run* the 6 basic user transactions in 5 minutes without writing a single line of code - no PHP, no HTML, no SQL. All the domain-agnostic code is provided by the framework while it is only the domain-specific code which needs to be written by the developer. This is why my productivity is higher than my rivals both for new development and subsequent maintenance.

Creating a new subsystem/domain follows the same basic steps:

- The analysis and design phase identifies the database design, the tasks and the business rules.

- Design and build the database.
- Import the database structure into my Data Dictionary.
- Export each table from the Data Dictionary to produce a separate Model class for each table.
- Select a table, link it with a Transaction Pattern, then press a button to generate a working transaction. Repeat using as many different transaction patterns as required.
- Modify table class files to add any business rules or task-specific behaviour. This can be done by using the hook methods which are already provided by the framework. These are empty methods which are waiting to be filled with custom code.

Note here that because so much work is provided by pre-built and reusable framework components the bulk of the developer's time can be spent in dealing with the most important part of the software in any application - the business rules. The developer doesn't have to design any software around those business rules, the framework creates the class files for each database table and the developer adds code into the relevant hook method in each class file to process the business rules.

Use Cases require tasks, not methods

Time after time I see OO tutorials which say that after a use case has been identified a class should be constructed with a method name specifically to execute that particular use case. I do not. Instead of using method names which are unique to the operation within each domain I use names which are common to the underlying database activity that will be performed during the execution of that use case. All the method names that can be used by my Controllers when communicating with Models have already been predefined as public methods in my abstract table class which is inherited by every concrete table class. This simple practice allows any of my Controllers to be used with any of my Models. This is a prime example of loose coupling, which is supposed to be good. If I were to manually create a method in a Model to execute a particular use case then I would have to manually create a Controller to call that method on that Model. As that method would be unique to that Model then that Controller could be used only with that Model, thus making that Controller inextricably tied to that Model and thus not reusable with any other Model. This is a prime example of tight coupling, which is supposed to be bad.

Instead I create a task in the framework's database for each and every use case. Each task points to a component script in the file system which is very small as all it does is identify which Model should be used with which View and which Controller. This means that instead of custom method names I use generic names, such as:

OO purists	effect on the database	the Tony Marston way

createProduct()	insert a record into the PRODUCT table	`$table_id = 'product';`
		`....`
		`require "classes/$table_id.class.inc";`
		`$dbobject = new $table_id;`
		`$result = $dbobject->insertRecord($_POST);`

createCustomer()	insert a record into the CUSTOMER table	`$table_id = 'customer';`
		`....`
		`require "classes/$table_id.class.inc";`
		`$dbobject = new $table_id;`
		`$result = $dbobject->insertRecord($_POST);`

createInvoice()	insert a record into the INVOICE table	`$table_id = 'invoice';`
		`....`
		`require "classes/$table_id.class.inc";`
		`$dbobject = new $table_id;`
		`$result = $dbobject->insertRecord($_POST);`

payInvoice()	insert a record into the PAYMENT table	

```
$table_id = 'payment';
....
require "classes/$table_id.class.inc";
$dbobject = new $table_id;
$result = $dbobject->insertRecord($_POST);
```

A significant point to notice here is that I do NOT have a separate Controller for each Model which calls methods which are specific to that Model as this would require each Controller and Model to be hand-crafted and tightly coupled to each other. All my Controllers are supplied by the framework as services, and each Model is generated by the framework as an entity. All my Controllers access their Model(s) by using method names which have been defined in my abstract table class, and each Model inherits from this same abstract table class. This means that no Controller is inextricably tied to a single Model, which would produce tight coupling, but instead any Controller can be used with any Model, thus becoming as loosely coupled as is possible.

Anybody who knows anything about OOP should see that my method is a shining example of polymorphism in action. Every one of my table classes uses the same method signatures by virtue of the fact that they are provided by the same abstract table class. Every one of my page controllers calls these methods, so if I have 40 controllers and 400 table classes that means that I have 40 x 400 = 16,000 - yes SIXTEEN THOUSAND - opportunities for polymorphism. How many frameworks have you seen which can match that?

Optimise the User Interface

If you show the average programmer a collection of different HTML pages they will tell you that each is totally unique and therefore needs to be designed and built independently. An experienced programmer should be able to see the similarities as well as the differences. The similarities are elements which appear in multiple pages and which look and act in the same way, and it should be possible to create a catalog of patterns which show the different arrangements of these similar elements. Among these similar and repeating elements are the following:

- The Menu Bar which contains the current set of menu options.
- The Title Bar which displays the name of the current task.
- The Navigation Bar which provides a number of menu options which are only applicable to the current task.
- A line of Column Headings which appear above rows of data in LIST screens.
- A Data area which contains application-specific data.
- A Message Area which shows any error messages in red and informative messages in green.
- A Pagination Area which appears under those areas in LIST screens which contain multiple rows displayed horizontally.
- A Scrolling Area which appears under those areas which contain a single row displayed vertically.
- The Action Bar which identifies actions such as SUBMIT or CANCEL.

Long ago, after having designed thousands of different forms, I recognised that each screen/form could be described in the following ways:

- **Structure** - what it looks like, how many elements it contains, how they are positioned and how they are related.
- **Behaviour** - what it does, what actions it performs, either automatically or by user selection.
- **Content** - what data from which database table(s) is visible within each element.

Unlike with my previous languages which required the forms to be built and compiled using a built-in process, the building of HTML pages is far easier simply because each page is constructed as a text file which does not have to be compiled and therefore can be constructed on the fly. This text file can be constructed in many ways, but my method of choice was to put all the relevant data into an XML document and then transform it into HTML using an XSL stylesheet. I started off with a separate XSL stylesheet for each HTML page, but following a process of refactoring I managed to reduce this down to a small set of Reusable XSL Stylesheets. I created a single reusable View object to handle both the construction of the XML document and the XSL transformation, which then meant that I did not have to write any code, apart from the small screen structure script, in order to create any web page.

Some pages produce PDF output instead of HTML, so again I have a single View object which handles all the processing with the aid of a small report structure file.

How can specific business rules be inserted into the application?

Some of the business rules are implemented in the structure of the database which identifies what data elements need to be stored for each entity. Each of these elements has a specific type and size, and can be designated as being either NULL (optional) or NOT NULL (required). A competent programmer should therefore be able to create a standard routine which validates that each piece of user input matches the database definition before any attempt is made to add it to he database. In my framework this functionality is provided by my validation class, so it is an example of more code which does not have to be written by the developer.

Each method call from the Controller to the Model will initiate a database operation in a series of predefined steps which have been built into the abstract table class as shown in this set of UML diagrams. Those methods with a "_cm_" prefix, which are known as customisable methods or "hook" methods, are defined within the abstract class but are empty, which means that when they are executed nothing happens. However, if any of these methods is copied from the abstract class to the concrete class then the copy in the concrete class will override, or be executed instead of, the empty original in the abstract class. If any code is inserted into the method in the concrete class then this code will be executed when that method is called.

The use of these "hook" methods is a prime example of the Template Method pattern.

How can the same method execute different code for different tasks?

It is quite possible that it some circumstances you may wish to perform some processing in a particular hook method for a particular task that is totally different from the code used in the same method for a different task. In this situation I use a standard OO technique called subclassing. I create a subclass of the table class with a name such as `<table>_sNN` where "s" denotes a subclass and "NN" is an incrementing number. This automatically inherits all the methods in the superclass but allows me to override any of those methods with a different implementation. Note that the subclass deals with the same table as the superclass, it just provides different business rules for that table. I then change that task's component script to point to this subclass. You can see working examples of this in the Data Dictionary subsystem of the framework where you will find the following class files:

- `dict_table.class.inc` handles the standard processing for "dict_table".
- `dict_table_s01.class.inc` is used to import table details from the database schema into the dictionary database.
- `dict_table_s02.class.inc` is used to export table details from the dictionary database into the PHP application.

Conclusion

An enterprise application is comprised of a series of subsystems or modules each of which serves a different business area or domain. Each subsystem contains a series of user transactions which perform an action or unit of work for the user. Each transaction has a user interface at the front end, a database at the back end, an a component in the middle which moves data between the two ends as well as executing the business rules which are specific to that domain and/or user transaction. Of the two types of logic - domain-specific and domain-agnostic - it should become apparent to the experienced developer that the domain-agnostic logic is the larger of the two, and contains elements which are common to many transactions regardless of their business domain and which could be placed into reusable components so that they can be shared instead of duplicated. If every user transaction follows a familiar pattern with the only difference being the domain-specific logic then it would seem a good idea, at least it would to me, to start building each user transaction from a pre-defined pattern into which the domain-specific logic can be added afterwards.

This situation appears to me as being a prime candidate for the Template Method pattern. In my implementation of this design pattern my framework provides a series of Transaction Patterns each of which handles a different combination of database interaction and user interface. In the enterprise application which I built using my framework I have 17 subsystems which contain 3,000 user transactions which were built from 40 patterns. While the sharable Controllers contain fixed logic each Model contains a combination of fixed logic, which is defined in and inherited from the abstract class, and customisable logic which can be defined in the various "hook" methods within each concrete class. These hook methods are initially defined as empty methods in the abstract class, but which can be copied into the concrete class and filled with suitable code.

This means that I do not have to waste valuable time in designing classes and methods for each object in the business layer as each object is a database table whose basic operations can be provided by methods in the abstract table class. Each table therefore has its own concrete class which I can generate, by means of my Data Dictionary, from the database schema. I do not have any class hierarchies as every concrete table class (Model) inherits all the standard code it requires from my abstract table class. Domain-specific logic can be added into a Model class using any of the numerous hook methods. As each user transaction has its own combination of user interface and interaction with the database I have managed to create a library of 40 Transaction Patterns each of which covers a different combination. Unlike design patterns which exist only as abstract descriptions my Transaction Patterns exist as concrete implementations which can be reused over and over again. In my framework it is possible to generate and then run a working user transaction simply by saying "Take table X, link it with pattern Y and produce transaction Z" thus eliminating the need to write any of that boring boilerplate code.

People should remember that the primary objective of a software developer is to develop cost-effective software for the benefit of the paying customer. It is *NOT* to follow a set of arbitrary or artificial rules created by fellow developers to promote their idea of "purity" or "perfection". I have encountered many of these rules in the past, and I have found that I can write better software by ignoring them. The only universal rule that I follow comes from a book called "The Structure and Interpretation of Computer Programs" which was published by H. Abelson and G. Sussman in 1984:

> Programs must be written for people to read, and only incidentally for machines to execute.

Other rules that I follow are "Keep It Simple, Stupid" and "If it ain't broke don't fix it".

In OOP one of the most important principles is that of **abstraction** from which we can build abstract classes to identify the high-level concepts, and concrete classes which supply the low-level details. It could be said that an abstract class provides a basic pattern while a concrete class provides a working implementation of that pattern. In order to fulfil one of the aims of OOP, which is to provide more reusable code, it should be noted that creating a large abstract class which is inherited hundreds of times is far better than creating a small abstract class which is only inherited once or twice.

If an enterprise application is comprised of a mixture of domain-specific logic and domain-agnostic logic, and the domain-agnostic logic can be seen to contain a set of repeating patterns of behaviour, it would make sense to me to create some sort of abstraction for each pattern so that it can be converted into any number of concrete implementations when building the individual application components. It should then be obvious that the domain-specific logic resides in the concrete implementation and not the abstract pattern. Since the abstraction should be defined BEFORE the concrete implementation then the followers of DDD have got it arse-backwards by starting with the domain-specific logic and leaving the domain-agnostic logic until last. This to me sounds as daft as building a house by starting with the roof and leaving the foundations until last.

When I add another domain to my enterprise application it is implemented as another subsystem which has its own subdirectory under the application's root directory, has its own database to hold its collection of tables, and has its own set of tasks (user transactions) in the MENU database. After analysing the requirements of the new subsystem I first build the database, construct the class file for each database table, then use the framework to build the tasks which perform operations on each table class. Specific business logic is added in afterwards by inserting code into any of the pre-defined hook methods.

The differences between the "approved" DDD approach and my heretical approach can be summarised in the following comparison matrix:

	the "approved" way	**the Tony Marston way**
1	Start by focusing on the domain logic and design the software around that logic.	No. After analysing the requirements I have a preliminary database design and a list of tasks (use cases). I then build the database and design the software based on actions that will be performed on those tables.
2	Design classes which have a method/operation for each use case.	No. Each use case has a separate task in the MENU database which points to a separate component script which in turn identifies a Model, View and Controller which will be used to implement that task.
3	Design complex class hierarchies.	No. I never inherit from one Model class to create another Model class. Each Model class inherits from a single abstract class.
4	Use object aggregation to create objects which are responsible for groups of database tables.	No. The idea of a "root" table through which you have to go in order to access individual tables within that group simply does not exist in any RDBMS that I have used. Each table is a stand-alone entity, so I give each table a stand-alone class.
5	Use object composition instead of inheritance.	No. There is nothing wrong with inheritance *when it is used properly*, but it would appear that far too many of today's young programmers simply do not know what "properly" means. According to the book Design Patterns: Elements of Reusable Object-Oriented Software problems can be created when you have deep inheritance hierarchies (which I don't) and extend a concrete class to create a totally difference concrete class (which I don't). You can avoid these problems by only ever inheriting from abstract classes (which I do). I have a single abstract table class which is inherited by every concrete table class.

6	Create objects to deal with domain events.	No. Objects in the Business/Domain layer are centered around entities which have properties (data) and methods (operations). These objects represent nouns while the operations represent verbs. Objects in the Presentation or Data Access layers which perform standard operations on the data obtained from business objects are called Services. Domain objects have state while Services do not. Domain objects are large in number and are specific to their individual domains. Services are far smaller in number, are domain-agnostic, and are provided by the framework. Domain Events are therefore implemented as operations within the relevant domain object and not as separate service objects.
7	Create service objects to deal with operations which conceptually do not belong to any object.	No. All such objects have already been built and come supplied with the framework. These are Controllers, Views and Data Access Objects.
8	Use factories to create domain objects.	No. I don't need factories to configure any domain object as each object has only a single implementation, and that is handled with the class constructor.
9	Create a separate repository for each domain object.	No. I have a separate DAO for each RDBMS which can deal with any table with that RDBMS. I do not have a separate DAO for each table (domain object). Every table in the database is subject to the same operations - Create, Read, Update and Delete - which are specified using a query string, so a competent programmer should have no difficulty in building a single object which can handle those four operations on any database table.

The advantage of my heretical approach is that I am able to cut out a great deal of manual effort. All the domain-agnostic logic is provided by components in the framework, which means that I can spend the bulk of my time in dealing with that which has the most importance for the paying customer - the domain-specific logic. Amongst the things that I do not have to send time on are:

1. I do not have to spend time in designing classes, methods and hierarchies as each database table has its own class. This is constructed by the Data Dictionary subsystem in my framework which uses information that is imported directly from the database schema.

2. I do not have to spend time in defining the logic which handles the basic input/output operations for each table class as all this common code is inherited from a standard abstract table class.

3. I do not have to spend time in designing mechanisms to handle domain-specific logic as the methods inherited from the abstract table class already implement the Template Method Pattern by providing a series of methods with concrete implementations interspersed with empty hook methods into which the developer can insert custom logic.

4. I do not have to spend time in designing custom Controllers which call methods which are specific to individual Model classes as each Controller calls only those generic public methods which are defined in the abstract table class.

5. Each task (use case) within the application performs a series of operations on a database table, and as there are different tasks which perform identical operations on different database tables I have managed to encapsulate the common code into a series of Transaction Patterns. A facility within my Data Dictionary allows tasks to be generated by linking a particular table class with a particular pattern.

6. I do not have any Model names hard-coded into any Controller as these are passed down by each task's component script using a form of Dependency Injection. Because I can now use any of my 40 Controllers with any of my 400 Models this provides an enormous amount of code reuse through polymorphism.

7. I do not have to spend time in designing and building individual View components for each domain object (Model) as each HTML page is constructed from a standard reusable XSL stylesheet. A single View object, which is provided by the framework, can extract the data out of any Model and insert it into the XML document which is then used in the XSL Transformation process. The differences between one web page and another have been isolated in a screen structure file

8. I do not have to spend time in writing code to provide user authentication, to construct menus, to control an individual user's access to specific tasks in the application as this is already handled by the Role Based Access Control (RBAC) system which is the backbone of the framework.

I have been building enterprise applications across numerous domains for several decades, and I have built a development framework for these applications in three different programming languages. This framework makes me very productive by providing huge volumes of domain-agnostic logic as standard from which I can then create user transactions into which I can add domain-specific logic. I could never be as productive using an arse-backwards technique such as Domain Driven Design, so forgive me if I consign it to the toilet bowl.

Here endeth the lesson. Don't applaud, just throw money.

References

Here are some other heretical articles I have written on the topic of OOP:

- Development Standards - Limitation or Inspiration?
- What is/is not considered to be good OO programming
- In the world of OOP am I Hero or Heretic?
- Object-Oriented Programming for Heretics
- What is Object Oriented Programming (OOP)?
- Design Patterns - a personal perspective
- Design Patterns are dead! Long live Transaction Patterns!
- What are Transaction Patterns?
- Object Relational Mappers are EVIL
- Dependency Injection is EVIL
- Not-so-SOLID OO Principles
- Not the three greatest paragraphs ever written on encapsulation
- Table Oriented Programming (TOP)
- A minimalist approach to Object Oriented Programming with PHP
- Your code is crap!
- Using object composition for "has-a" relationships is not such a good idea
- OO Design is incompatible with Database Design
- How NOT to Validate Data
- Let's Make It More Complicated Than It Really Is Just To Prove How Clever We Are
- Object Oriented Database Programming
- On not using the "right" standards
- DB or not DB, that is the question
- What is the difference between Procedural and OO programming?
- Singletons are NOT evil
- The concept is OK but your implementation is not
- Your rules are RUBBISH!
- Re: Objects should be constructed in one go
- Re: What's so great about OOP?
- Anatomy of an Enterprise Application

Here are some articles on my framework:

- What is a framework?
- A development infrastructure for PHP
- A Role-Based Access Control (RBAC) system for PHP
- User Guide to the Menu and Security (RBAC) System
- The Model-View-Controller (MVC) Design Pattern for PHP
- An activity based Workflow Engine for PHP

- Creating an Audit Log with an online viewing facility
- A Data Dictionary for PHP Applications
- Transaction Patterns for web applications
- RADICORE for PHP - Functions, Methods and Variables
- FAQ on the Radicore Development Infrastructure
- How Radicore's Hook System Works.

Amendment History

24 Jan 2019	Added The biggest differences with the most similarities
02 Dec 2018	Added a Comparison Matrix to summarise the differences between the "approved" approach and my "heretical" approach.
01 Oct 2018	Added Build a library of reusable components Added Turning a library into a framework Added Optimise the User Interface Reworded the article to emphasize the fact that DDD starts with the domain-specific logic and adds in the domain-agnostic logic afterwards whereas my framework does the exact opposite. It is an implementation of the Template Method pattern which starts with domain-agnostic logic and allows domain-specific logic to be added in later via a pre-defined collection of "hook" methods.

0 Comments **tonymarston.net** 1 **Login**

Recommend Tweet Share **Sort by Best**

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON **TONYMARSTON.NET**

Dependency Injection is EVIL

2 comments • 6 months ago

Tony Marston — Most certainly. The idea that I should use DI for every dependency whether I need it or not does not make sense to me. I use it where it provides genuine benefits, as in my Controllers and Views, but I

Not-so-SOLID OO Principles

2 comments • 6 months ago

Tony Marston — I am not confused about DIP simply because I have read the article written by the guy who invented it. His sample COPY program quite clearly shows that any benefits of using this principle only appear

Re: Objects should be constructed in one go

12 comments • 7 months ago

disqus_ZJZZTbBtWo — tony_marston 5++

Object Oriented Database Programming

2 comments • 6 months ago

Tony Marston — If you look back at the responses to my articles as far back as 2003 you will see that I have had nothing but personal abuse for my "heretical" ideas. That is why I respond in kind. People keep telling me