# REDUCE DEVELOPMENT EFFORT WITH PATTERN-ORIENTED DEVELOPMENT


Jean Minnaar
jean@sumomobi.com
September 2013

# 1 Introduction

## 1.1 Problem Statement

In automobile manufacturing the manufacturer creates an assembly line and once done, assembles one vehicle after another. Then they reuse most of that assembly line for the next vehicle model -- the assembly line get designed right out of the gates as to where it can be reused for yet another vehicle model. This is repetitive manufacturing that software developers wish they can achieve. But instead, when developers need something to "pump tires" with, every developer tends to build his or her own pump.

## 1.2 Purpose of Article

This article promotes the idea of building the "assembly line" **first** after which one application (or one function of the application) can be assembled after the next. Similar to automobile manufacturing, it does not make sense in software to build the "assembly line" after you already are in the manufacturing phase. Fortunately in software an assembly line can, although not ideal, be injected piece by piece at a later date.

Parallels are drawn between the "assembly line first" approach and pattern-oriented development. In fact pattern-oriented development is the foundation for an "assembly line first" development model.

The purpose of this article is to explain pattern-oriented development and how it leads into the "assembly line first" practice.

## 1.3 Definitions

In software development we do not use terms like assembly line. Instead we call it Frameworks and Frameworks are usually accompanied by Libraries.

In case you need to be reminded of the difference between Frameworks and Libraries, you will see something similar to the below in related Internet articles:

- *A **software library** is essentially a set of functions that you can call, usually organized into classes. Each call does some work and returns control to the client.*
- *On the other hand, a **software framework** embodies some abstract design, with more behavior built in. In order to use it, you need to insert your behavior into various places in the Framework. The Framework's code can also call your code at given points.*

# 2 A Word on Patterns and Frameworks

## 2.1 Introduction

Before going much further, it is important to take a closer look at design patterns. Let's face it. A rather large portion of the developer community is not fascinated with design patterns.

At this time you may also think this article are about the known/formal software design patterns like the Decorator, Proxy, Adapter, etc. patterns. This is not the case though. At most this article will mention (but not discuss) a formal pattern from time to time. This article is about **current day patterns** found in Web Application development.

## 2.2 Pattern-Oriented Development Deficiencies

Design Patterns have been criticized widely and rightly so. The below shares with you apparent shortages in what is generally considered Pattern-Oriented development:

1. The need for design patterns resulted from using computer languages or techniques with insufficient abstraction ability. Peter Norvig provided a similar argument. He demonstrated that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in other languages.
2. Design patterns lack formal foundations. At an OOPSLA conference, the Gang of Four was (with their full cooperation) subjected to a show trial in which they were "charged" with numerous crimes against computer science. They were "convicted" by ⅔ of the "jurors" who attended the trial.
3. Design patterns do not differ significantly from other abstractions. Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary.
4. Design patterns lead to inefficient solutions. It is almost always a more efficient solution to use a well-factored **implementation** rather than a "just barely good enough" design pattern.

This article suggests overcoming such deficiencies by putting a retooled Pattern-Oriented approach to work.

## 2.3 A Distinctly Different Viewpoint of Design Patterns

**Focus on current-day patterns**. While focusing on the formal patterns, the present-day patterns could go unnoticed. Stated differently, put the focus on the problems at hand and do not allow yourself to get distracted by a few patterns that may not be relevant to your situation.

**Executable deliverables**. As stated in point (4) in the previous section, often patterns are not worth much unless implemented and provided to developers in some executable and usable format. .NET developers may at this time think about NuGet packages. That would be correct, but underneath the package (regardless how it is packaged and delivered) is what one can call, a Framework. Think of it as a

foundation or plumbing if you will – something the rest of the application gets built on top of. (Usually there is a Library component accompanying the Framework too.)

The above two approach differences are the key to putting a retooled Pattern-Oriented development approach to work. The rest of the article will confirm what this retooled approach is about.

## 2.4 Software Framework Redefined

Earlier on we saw a brief definition for Frameworks and Libraries. Before moving on, it is important to point out the author's definition of such:

> *A software framework is a set of design patterns (formal and/or informal) accompanied by the code necessary to take care of the common functionality of the design patterns and to expose the framework component functionality to the developers.*
>
> *Frameworks are almost always accompanied by what people describe as a **library**. For instance, you will find components that get registered with the framework which provide some functionality for the application.*
>
> *Software Frameworks are all about design patterns -- elimination of repetitive development work -- and are used to speed up the development cycle. They are used to streamline software development by allowing designers and programmers to devote their time to meeting software requirements rather than dealing with the common functionality and more standard low-level details of providing a working system. A software framework's purpose is to reduce overall development time.*
>
> *With Pattern-Oriented Development, the framework (a pattern driven software framework) is the forerunner deliverable leading the rest of development. It is generally understood that the most effective software frameworks are those that evolve from refactoring the common code of the enterprise. The software framework covered in this article certainly evolved this way – it was created by developers for developers.*

What the above reiterates, is not to stop at design patterns but to take it a step further by adding the code for the design patterns by which time we label it as a pattern driven software framework – a Framework with Pattern-Oriented origins.

References to the term, Framework, in the rest of this article pertain to the pattern-oriented [software] frameworks as described in this section.

## 2.5 Object Discovery versus Pattern-oriented Development

Assume the task given to you is to service Apache helicopters and C-130 cargo aircraft. How will you go about this? First you break the work up for each to see what must be done to service them.

For the helicopters you have to:

- Resurface the rotor blades
- Do a bunch of other stuff
- Pump the tires
- Do a bunch of more things

For the C-130s you have to:

- Do some stuff
- Pump the tires
- Do some more stuff

From the above domain narrative, object discovery shows that we need a pump. So we think we are in good shape – we know our objects. But what commonly occurring problems did this analysis exercise not reveal?

Probably stating the obvious here… Object discovery (together with responsibilities and collaboration) and pattern discovery are two totally different exercises.

## 2.6 Formal/Informal Patterns

From time to time this article refers to a given formal pattern. If you don't recognize a mentioned formal pattern, you probably encountered it numerous times in your life, implemented it and preached it to others, just not recognizing its name today. Go to the Formal Design Patterns appendix for a reminder, in layman's terms, of what the referenced formal patterns are all about.

Does this mean you have to look for where you missed the Adapter or Bridge, etc. pattern in your system? Looking for where you missed the formal design patterns is likely not where you will find the answers. The issue is more likely the lack of discovering and acting upon informal design patterns in your system.

Keep in mind that this article is rather about informal design patterns with the focus on Web Applications.

## 2.7 Summary

In short, patterns are about identifying the common application development challenges and solving/implementing them once.

# 3 Patterns by Illustration

## 3.1 Introduction

Now that the boring part of "definitions" is behind us, let's dig in and show examples of patterns as found in Web Development (regardless of technology stack).

Depending on the complexity of your application, there are anywhere between 30 and 60 patterns in a Web Application. However, this is an article and not a book on Web Application patterns. Therefore only a few patterns are mentioned in this article. The patterns selected for this article are the ones that do not require in-depth discussion and are then also easy to digest.

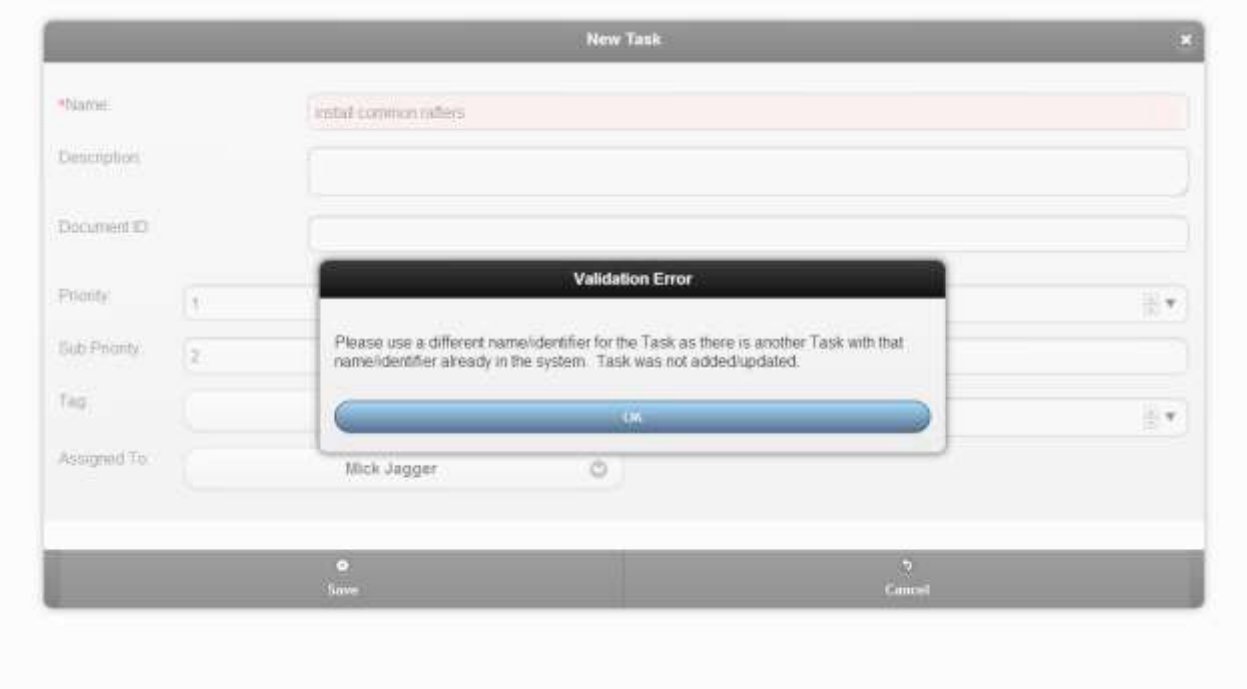
The intent is to illustrate the importance of Pattern-Oriented development and thereby instill the need in the reader to identify and take action on discovered design patterns.

Next we will cover each selected pattern in its own section.

## 3.2 Pattern: Server Side Generated User Messages

We have situations where at the server-end your code recognizes the need to show a message to the user. This could occur during a page request using any HTTP method (GET, POST, etc.) or during an AJAX (GET, POST, etc.) request.

Ideally the server side application code should just turn a message over to the Framework and by the time control is back at the browser, the user sees the message:



Application developers in general [j1]should not have to be concerned with (1) what the message boxes look like or (2) how messaging should be implemented. There could have been a redirect between the time the message was turned over to the Framework and the time the user gets an opportunity to see the message. The developers shouldn't have to concern themselves with such details. Even if another message gets handed off to the Framework, be that elsewhere in the server-side code in the same HTTP request or the newly redirected-to page, the user should be presented, at the appropriate time, all applicable messages and the application developer should just know that the Framework supports that -- the Framework is responsible for making sure the user gets to see every message in a timely manner.

This makes sense, right? But you may ask what this has to do with patterns? As mentioned in the Software Framework Redefined section, design patterns are all about recognizing **common** problems plus implementing a reusable solution for them – the Framework.

The next question is, is "Server Side Generated User Messages" (the pattern under discussion here) one pattern or multiple patterns?

There actually are the following subordinate patterns:

1. The pattern telling the developers how to implement/use the Framework's messaging or actually how to turn messaging over to the Framework instead of each developer rolling his/her own messaging implementation.
2. Then, there is the pattern (part of the Framework itself) dealing with the actual implementation details of messaging (not described here).
3. Finally, underneath the second pattern are patterns like the Proxy pattern.

Do you really care that the Proxy pattern relates to the Server Side Generated User Messages pattern? Probably not. All we care about is that we have a tried and tested server side messaging component in our Framework which is also simple to use – there is another component that simplifies the application developers' life.

The patterns mentioned in the rest of this article pertain to the Framework component plus a brief mention of the usage "patterns" presented to the application developers (as opposed to the Framework developers[j2]).

The patterns geared towards the application developers ("usage patterns" if you will) should be very simple patterns – simplicity rules!

Earlier in the aircraft servicing example, the article mentioned that not all common problem areas get discovered from the problem/requirement narratives. When dealing with requirements or acceptance criteria, the messaging problem as seen here, would likely not be revealed as for one, the focus is on the "actual" problem at hand. The result could then be that later on, when a developer recognizes a need for messaging, he/she ends up implementing his

or her own custom messaging. Not good as we will certainly end up with redundant plus dissimilar code providing the "same" functionality.

In the case of the pump, who would initially have known that we need a cart to transport the pump from one aircraft to the next? When the service technician makes that discovery, he acquires a cart. Over time the service technician learns that a cart is needed for other tools too and now faced with turning it into a generic cart or chaining multiple carts to each other and finally figures out the optimum solution. At that time the solution is useful to many different and new applications.

In software we should also go through similar refinements until we end up with an optimal solution for the shared challenges. Developers need to make that a priority and put a conscience effort towards building the Framework. It is a matter of gathering what you learned from previous application development efforts and refining the Framework into something that ends up shaving off countless development hours.

## 3.3 Pattern: Client Side Generated User Messages

The out-of-the-box Web technology methods window.alert() and window.confirm() lack in functionality as well as professional user experience especially on mobile devices. Instead of using them, developers request the Framework to present messages to the user together with the button text plus user options (for multiple selection message boxes) and then leaves the rest up to the Framework. At the end the developer code gets called back and told which option the user selected (in the case of multi-selection message boxes).

It goes without saying that the previous pattern, Server Side Generated Messages, piggy backed on Client Side Generated Messages.

## 3.4 Pattern: Web Forms and AJAX Post Backs

Here the Framework takes care of the core AJAX needs such as:

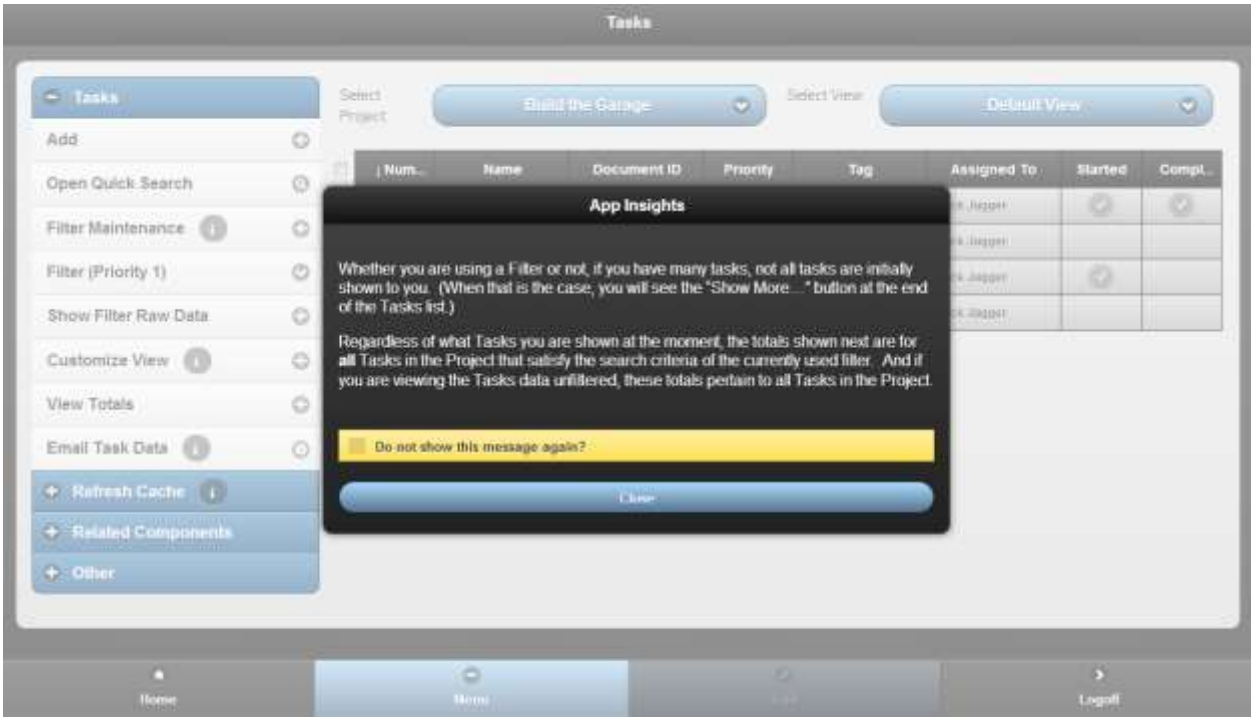
- Packaging up the form elements for post back.
- Making the AJAX call on behalf of the developer's code.
- Assisting at the server end with interpreting the received HTTP/Web Socket request and data.
- Dealing with any exception handling and resultant user messaging including validation error handling.

Developer code only needs to request the Framework to perform the AJAX call and on top of that developer code only needs to deal with success responses -- the Framework to handle non-normal responses.

## 3.5 Pattern: App Insights

App Insights is just a better term for what we usually call Help. However, Help is usually an on demand type Help except that you cannot clutter your UI everywhere with "information" type icons. The approach taken here is to show the user the App Insight when the user visits a given function of the system for the first time.

When a user views a particular app insight, the user can temporarily or permanently dismiss the shown message/insight. After permanently dismissing the message, the message will not be shown to the user again when he/she visits that particular functionality. Here is an example of App Insights after the user clicked on the **View Totals** menu item (of the particular app shown here) the first time:




Developers simply need to ask the Framework to show a particular **App Insight** at the appropriate place in their code. The Framework knows whether to show the App Insight or not and will turn control back over to the developer code when the Framework completed the app's App Insight request. There is no need for the developer code to track if an App Insight had already been dismissed by a particular user or not.

## 3.6 Pattern: Track Session Data per Tab and Browser Instance

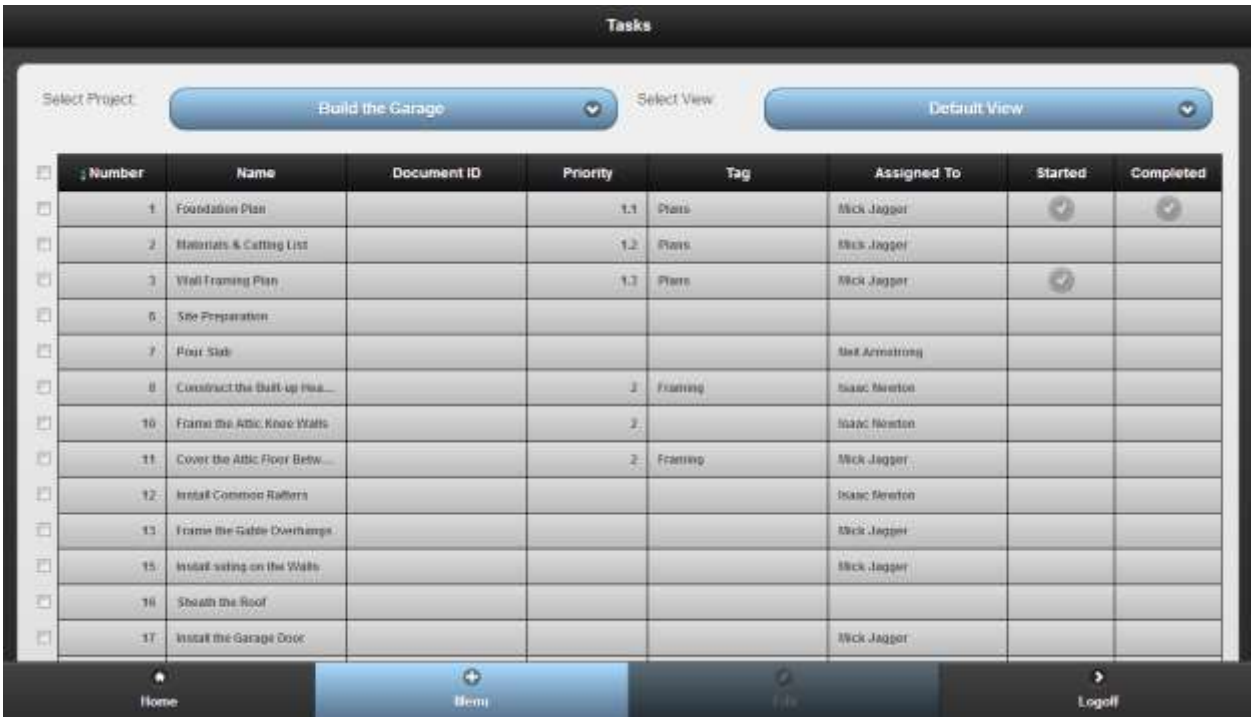
When you cache data in the middle tier the complication arises when a user opens the same Work Unit (see the Appendix discussing Definition of Terms) in multiple tabs or windows in the same browser instance. In the stateless HTTP world, the server generally does not know which tab the HTTP request came from and is thus not able to keep each tab's session storage separately.

If it is justifiable, you can opt to not allow the same Work Unit from being opened multiple times in the same browser instance. There are other alternatives like generating and sending tab identifiers to the server to be able to track each tab's session data separately.

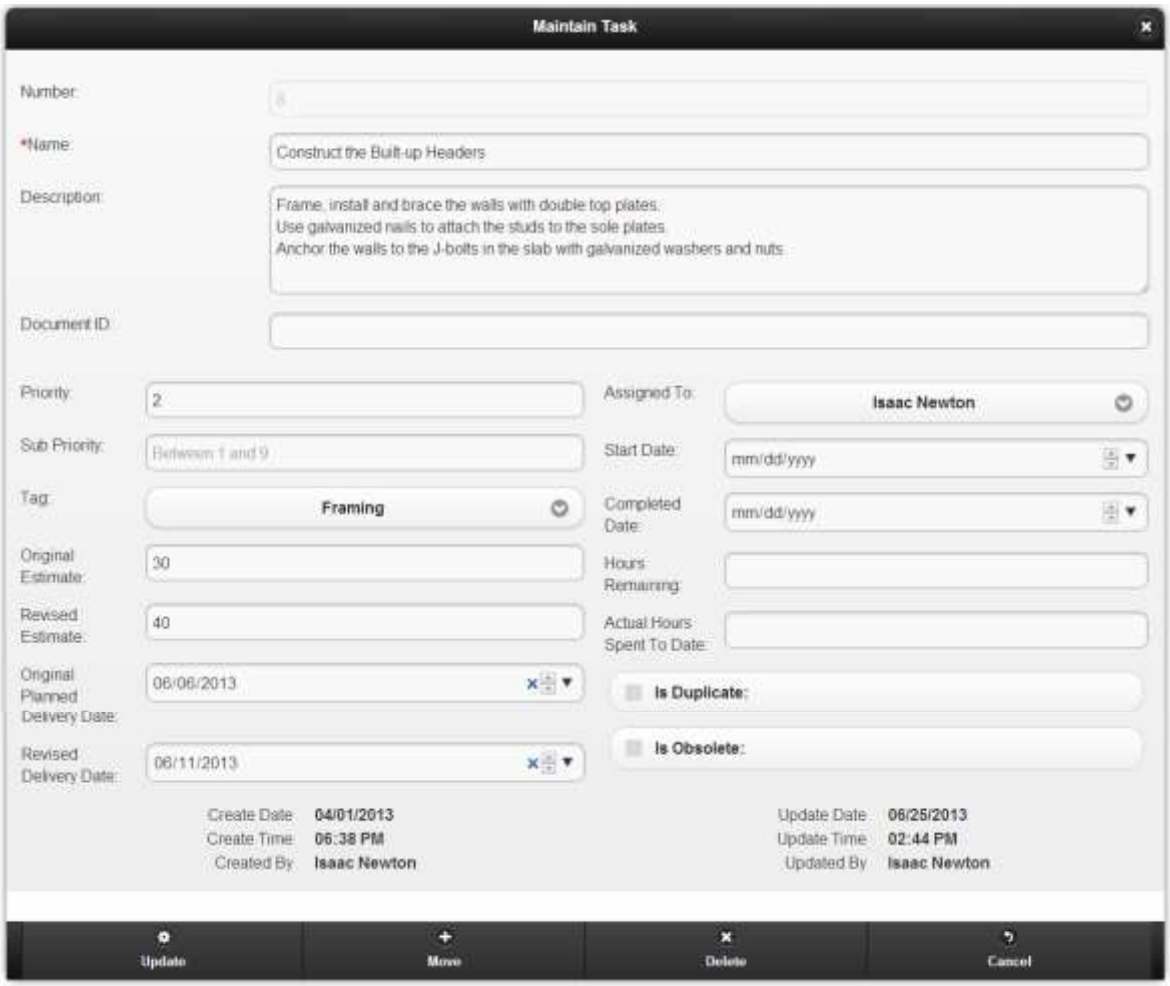
With this part of the Framework developers do not have to know how tabs are identified to the web servers and how the Framework keeps each tab's session data separately.

## 3.7 Pattern: Prevent Post Back Hack Attempts

With the modern-day Single Page Application (SPA) application approach where the user views a list of entities and then identifies the entity the user wants to work with, there is the tendency to ship the entity's database identity column to the client. The user starts with something like:




(Note that the number column on the left above is just a sequential number within the task list shown and not the internal identifier of the task.) The user decides to work on task 8, clicks on task 8 and gets the following dialog:




When the user selected task 8, the system made sure task 8 still exists and that the user gets the latest version of task 8. It then shows the above dialog to the user. When the user is done making the necessary changes, the dialog's Update will send the task data to the server. The server now needs to know which task to update AND where it needs not be concerned with AJAX hack attempts.

Typically the developers tend to send the internal database row identifier of the entity (task 8's identifier in this case) to the client/dialog. On update, the idea is then for the AJAX post back to echo the identity and for the server side to use that identity to know which entity to update. This creates an opening for hackers to change the identity in the post back request and try to update an entity the user does not have access to. Additional security logic (with resource consumption and performance penalties of course) is then needed to detect and avoid such hack attempts.

The Prevent Post Back Hack Attempts pattern provides an alternative to sending identities to the client and ensures that a hacker can only get to its own entities without the need for additional resource consuming security code and checks.

## 3.8 Pattern: Work Unit Session Cache

For performance, security, and other reasons an application will remember given data for the Work Unit in the Middle Tier. When a user jumps from one Work Unit to the next, the previous Work Unit's data should be removed from server's session cache -- session cache needs to get cleaned up at appropriate times or else server memory gets stressed which leads to performance problems.

Let the Framework take care of timely disposing of unused session cache for the developers.

## 3.9 Pattern: Cross Work Unit Cache

The cache common across multiple Work Units should be cached in the Framework's Cross Work Unit Cache. Typically this cache will only get cleaned up after the user logs off. Developers will access the common cache items from the Framework and not roll their own solutions.

## 3.10 Pattern: Layer Integration

### 3.10.1 Background

Often developers report they are "90%" done with a given implementation and that they only need to add for instance concurrency conflicts. However, to add concurrency conflicts the Data Access layer gets worked on, so the Business Rules layer which impacts the Business Façade layer as well as the UI Tier. Not just do we violate the Open-Close principal left and right but these components that get changed, may be used elsewhere in other workflows. In that case those impacted components will have to be reworked plus retested too.

Adding the plumbing when the house is already built just does not make sense. When another bathroom needs plumbing, the plumber does not dig up the floor and add that room's plumbing. "Plumbing" (Concurrency Conflicts for instance) needs to be in place right from the start.

Layer Integration involves having particular return structures between the layers. Based on the content of the return structures, the code will know that a request was successful or not. In case of failure, the Framework takes over handling of the response. (This type of component is often best implemented using Aspect Oriented Programming constructs.)

These return types are there to accommodate challenges such as:

1. Concurrency conflicts
2. Updates with no changes made by the user
3. Attempts to create a duplicate entity
4. Validation

The Framework components for the above four mentioned items are all similar. Therefore only the Concurrency Conflict challenge and Validation get discussed.

### 3.10.2 Pattern: Concurrency Conflicts

The approach taken here is the pessimistic locking approach -- we want to know when an entity got updated since a user pulled a copy of it for update and inform the user of such at the time when the user tries to update the entity. The entity could also have been deleted in the meantime by another user.

With the appropriate usage of the Framework, developer code (or Framework injected AOP code) just needs to recognize a concurrency conflict and turn it over to the Framework. From there the Framework takes over resulting also in appropriate propagation of the situation through every layer of the code all the way up to the appropriate user experience – the developer did not have to roll any additional lines of code to handle concurrency conflicts.

### 3.10.3 Pattern: Adapt Validation Fields

When the system detects, at the client or sever end, invalid data in one of more form fields, those fields, on top showing the user relevant validation failure messages, have to be styled as being in error. Developers should only provide the validation logic for the fields and leave the rest to the Framework to handle.

## 3.11 Pattern: Logging

There are four types of logging:

- Exception
- Utilization
- Performance
- Audit trail

Of course it is difficult to add logging after the fact. It should be part of the Framework from the start.

Exception logging is the most bottom component used by all. Even your email code will report exceptions to the exception logger. So what if the logging database is down? Who does the exception logger report that to? A proper logger framework solves this problem for you.

If email is the preferred communication mechanism for fatal exceptions, your inbox could become (hopefully not) flooded. Again, a proper logger framework will accumulate exceptions and periodically reports on the exceptions since the last time it reported the exceptions.

## 3.12 Pattern: Session Expiration

Assume your system relies on an active session or active authentication session. Secondly assume the user brought up an entity to update, saw the entity details in the update dialog, but then went inactive until after the inactivity timeout expired. The user then attempts to update the entity. The AJAX call arrives at the server end and sees that the session expired. With a proper Framework there is no need

for developers to code for session expiration during any type of HTTP call. The Framework takes care of that even taking to the user the Login page if re-login is appropriate.

### 3.13 Pattern: Handling of Unsupported Browsers

Developers should be assured that by the time a browser reaches their pages, the browser is a supported browser. Leave the responsibility, to detect compatible browsers, up to the Framework.

### 3.14 Pattern: Cookies Configuration

Checking to see if browser cookies are turned on or off should not be a developer task. It should solely be the responsibility of the Framework.

# 4 Conclusion

## 4.1 Framework Scope

A comprehensive Framework has numerous more components than mentioned so far. This article is only aimed to touch on a few patterns and not to provide complete documentation on a complete Framework.

## 4.2 Framework Observations

From the bits of the Framework exposed in this article, you would have recognized that exception case handling was part of the application space right off the bat. Habitually developers actually write for the flawless conditions first. Why not? They get told, "*We have to have this next Tuesday!*" When they are done with "Tuesday's" deliverable, they claim they are nearly done.

Then the developers start adding the code for the exception cases, logging and so on. This often results in the code having to be re-opened (violating the open-close principal). It also results in code paths that worked before, to be broken now and the system remains in an "almost done" state for a very, very long time.

Not meaning to deviate from the topic, but let me also mention that the developers often take cover under the Agile methodology and call these next efforts, refactoring. There is a big difference between refactoring and building systems in a flawed and damaging sequence that needs major overalls later on.

Pattern-Oriented Development eliminates the mentioned issues – "exception cases" are handled right from the start, along with logging and more. Furthermore, these challenges are handled consistently across the application. The Framework takes care of the heavy lifting for the developers, thereby freeing them up so that they only need to focus on their domain.

What is very important to take note of, is that an established Framework already covers the common components found in any Web application. That in itself is an excellent start for any Web app project.

## 4.3 Why Are Assembly Lines Absent?

Even in modern day applications we very seldom encounter any significant "assembly lines". The reasons most likely are:

- Developers do not think along these lines. Especially the inexperienced developers where the main focus for them is on mastering the technologies first.
- Business tells developers that they need some application by "next Tuesday" (and there is no room for planning let alone creating the Framework first). Inability of development to push back on business wanting something quick instead of convincing business to invest in the assembly line.

- Developers at large do not believe in themselves. They think Frameworks should come from companies with huge budgets like Google. Just because they can throw in huge efforts, does not mean they can do a better job.

## 4.4 Development Inefficiencies

Our apologies for the somewhat negative trend of this section. It would have been wrong though for the author to make it appear as if the "assembly line first" approach would solve all development problems.

Certainly the "assembly line first" development approach makes a huge contribution towards development productivity. In some cases a 50% reduction in the number of lines of code. But it would be dumb to think this would solve ALL your project overrun issues. It is out of scope for this article to cover these other cost overrun topics in detail. (The author discusses some of these in a separate article.) To **mention** a few reasons for development cost overruns, the following:

**Reinvent the Wheel**. Instead of capitalizing on available software components and/or libraries, developers want to create their own. That is a recipe for disaster.

**OOP**. Just because a developer can articulate the Object-Oriented concepts and principals, does not mean the person thinks along the Object-Oriented lines let alone implement the application in an Object-Oriented way. The quickest way to know if a team is Object-Oriented, is to look at their database(s). Are there physical or at least logical boundaries between the different entities? Often that is not the case. And if the foundation, the database design, is not Object-Oriented, OOP attempts in the Middle Tier will fail too. Similarly SOA without an Object-Oriented database architecture is at best a "collection" of services where updates to one service likely impacts another service.

**Technologies**. Let's pick on CSS. Does an application have thousands of lines of CSS because the developers did not understand how to separate positioning from style and how to "extend" CSS classes? Lack of technology skills certainly can be a huge contributor to time-to-market delays.

**Scope Creep**. Developers are quick to blame scope creep. Why do we see scope creep and see it that often? With Use Cases the requirements were described in the form of a solution with the result that the actual requirements are just plain missed. Agile replaces Use Cases with Acceptance Criteria. This is one of the most important changes that came with Agile. Unfortunately if you do not take your time with Acceptance Criteria and therefore do not have clear requirements and scope defined, scope creep will stick its head out again even under Agile.

**Refactoring**. Agile gets criticized by some saying Agile is too costly for startups and small organizations. Why is there such a perception out there? With the Agile methodology, if you do not look further than the current sprint, you will build something that has to be redone with the very next sprint. That instead of using the previous sprint's code base as foundation for the next sprint. When there is major refactoring needed, the developers' justification is, "It is Agile."

That happens when the development team does not have a decent enough view of where they were supposed to end up.

## 4.5 Jambalaya

Say you went to New Orleans and got introduced to Jambalaya. Assuming you are very impressed with it and want your friends to experience it too. How do you describe a good meal to a friend? You tell them, “You should TRY it!” After all, is there a better way? Not really.

Therefore, you should just try “assembly line first”!

## 4.6 Obtainable Framework

This article is based on a Framework that applies to Web apps running on over 100 device types including desktops, tablets and smartphones.

# 5 Appendix A: Definition of Terms

## 5.1 Work Unit

The quickest way to explain a Work Unit is to look at a few examples of Work Units.

In a Project Management app one of the Work Units is the Project Work Unit. (This Project Work Unit follows the Single Page Application pattern) In this Work Unit the user starts off by being presented with a list of tasks under a particular project. The user can select a task and, via a dialog, modify the task. After modifying the task, the dialog closes and the user is back at the task list with the relevant task now containing the updates. The user can search for tasks, delete a task (via dialogs again), change the view of the list data, and much more. The Project Work Unit encompasses all these project related functions.

Similarly, in the same app, there is a Projects Work Unit where the user initially gets shown a list of projects. Again, the user can perform all kinds of project related functions while in the Projects Work Unit. These could be to add or modify projects, assign or remove users from a project, and more.

An example of a Wizard type Work Unit is where you encounter a multi-step user experience. The first to the last page in this work flow constitutes the Work Unit.

In looking at the Work Unit related patterns you will come to the realization why establishing Work Unit boundaries are very important.

## 5.2 Tiers and Layers

The following on Tiers (physical separation) and Layers (logical separation):

> **UI Tier:** Comprised of the UI layer running in the browser on a client machines.
>
> **Middle Tier:** Comprised of the Business Façade, Business Rules and Data Access layers running on the Web/HTTP servers.
>
> **Database Tier:** Comprised of the Database layer running on the database servers.

The following statement does not necessarily apply to a J2EE environment but it does apply to a .NET environment: More than one Tier can run on the same machine but Layers of the same Tier should not be spread out to run on different machines. (It is not the intent to explain the J2EE versus .NET statement as it involves .NET's virtual processes, where the HTTP.SYS driver resides and more – a topic for another article).

# 6 Appendix B: Formal Design Patterns

## 6.1 Introduction

This article referenced a few formal patterns. Below you will find an explanation of each of those.

The author borrowed ideas to explain these patterns in non-software terms from *Emergent Design* by Scott Bain.

## 6.2 Adapter

We want to use an existing class but via a different interface than its current interface. We want to make an existing object exchangeable.

A Light Socket Adapter is an excellent example of an Adapter pattern implementation. We still want the power from the light socket but for a different application/use.

## 6.3 Bridge

Separate something from what it is from what it does.

A waitress sees the persons at the restaurant tables as just patrons. Each may be unique but she does not care about that. The waitress asks each patron the same question, “What would you like to eat?” Each patron responds with potentially a different answer.

Who the patrons are, vary and the available food may vary but all patrons can use a menu. From the waitress’ point of view, she just wants an answer.

## 6.4 Decorator

Dynamically add additional behaviors to an entity.

A camera is an excellent example where we encounter the Decorator pattern. Without changing the camera and without changing the outside world being photographed, we can put a filter on the camera and alter the resultant image – we can decorate the camera.


## 6.5 Proxy

We want to add additional behavior to an existing class but we also need to be able to use the existing class at times without the additional behavior.

Take a garden hose as an example. We have a sprayer connected to the hose but now we want to add fertilizing functionality. For that we add something like the below: