# Cross-Cutting Concerns

## What is it?

The crosscutting concerns/aspects consist of security, caching, exception handling, logging, etc. These aspects transcend into other concerns (i.e., layers). In a simple layered architecture you have three common layers: Presentation, Business and Data. The cross-cutting concerns are applicable throughout the architecture layers and it affects the entire application. For example: logging, security and data transfer are the concerns which are needed in almost every module of an application, hence they are cross-cutting concerns.

## What is the value?

Without some framework, or extensible tools, cross-cutting concerns are piecemealed, forgotten, or a mish-mash of incongruous parts. Development often forgets these aspects because it is boring or just an after-thought. When you do need these aspects you patch only a portion or where there is a problem; but not consistently through-out. The foundations become tenuous and left neglected.

The value lies in being consistent, reliable, reusable and flexible. Understanding one size does not fit all.

## Why do it?

In many object orient languages you will find solid frameworks which support these aspects, for example, Java’s *Spring, Hibernate*, C#’s *Spring.Net, NHibernate*, just to name a few*.* However, I did not find what I was looking for in Apex. Second, I consider these aspects as foundational; where others consider them … *boring*, *an after-thought*, *or not important*.

Cross-cutting Concerns (CCC) attempts to provide functionality that is a bit more challenging in Apex than in other STOOLs[[1]](#footnote-1) . The goal was to apply [SOLID](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))[[2]](#footnote-2) principles to allow flexibility for change and reuse.

## How does it works?

Defining a set of common interfaces and custom metadata within the CCC framework allows one to change/augment aspects at different levels/granularity. The best way to understand the value is with a concrete example.

For example, if we look at a common concern, such as *Logging*, there are two aspects eluded to earlier. First, there is the logging interface, *ILogger*. The interface allows a myriad of logging behavior:

1. Logging supports four different functionalities
   1. Logging to Salesforce Debug-Logs via **System.Debug** (*accc\_ApexLogging -> accc\_ILogger*),
   2. Logging to a custom object, *ApplicationLog\_\_c* (*accc\_ApexObjectLogging -> accc\_ApexLogging*)
   3. Logging via Platform Events (*accc\_ApexPublishEventLogger -> accc\_ApexLogging*)
   4. No Logging via (accc\_*ApexNoOpLogger -> accc\_ApexLogging*).

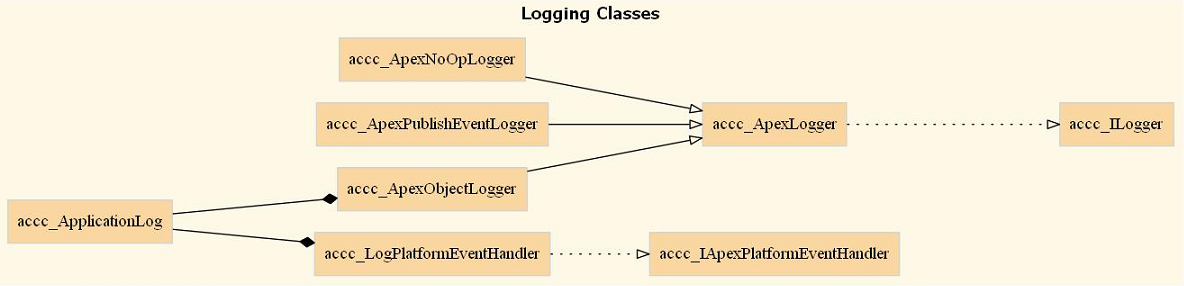


Figure 1 Logging

Second, there is the use of custom metadata. CCC framework uses two important components to facilitate change,

1. an Object Factory (*accc\_ApexObjectFactory-> accc\_IApexFactory*) and
2. Runtime environment (*accc\_ApexBaseRuntimeEnvironment ->* *accc\_IApexRunEnvironment*).

These two classes allow one to inject (via a custom metadata object) the object factory with the supported behavior. Thus, you can change the runtime behavior of *Logging as well as other cross-cutting concerns (exception handling, caching, configuration, fallback resources, etc.)*. At this time, CCC supports three runtime environments; each which can define different behavior:

1. **Debug** [i.e. when running in a Sandbox]
2. **Test** [i.e., when running in a Test (Test.isRunning)]
3. **Production** (neither item 1 nor 2)

## Core Components

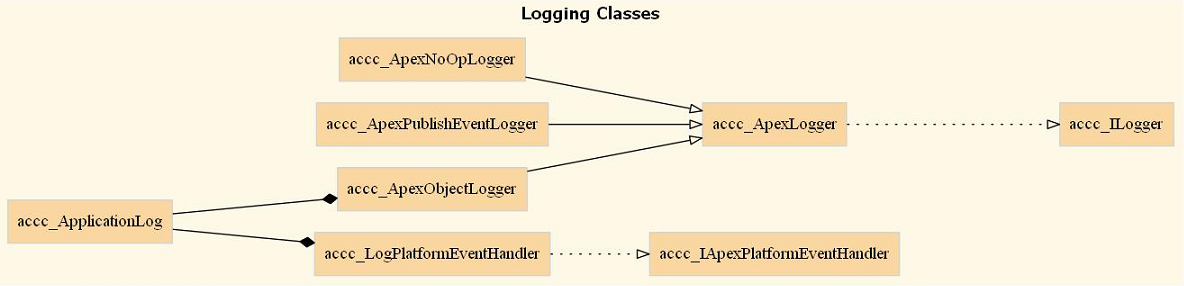
The core aspects of the CCC framework build upon four components:

1. Cross-cutting concerns (logging, caching, exception handling, providers, etc.)
2. Runtime environment(s)
3. An Object Factory (i.e. Design Patterns)
4. Custom Metadata Types

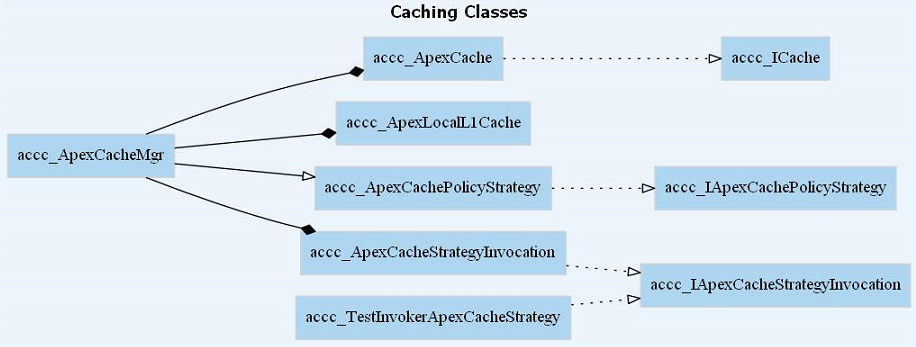
### Cross-cutting concerns

The following items below are the main cross-cutting concerns supported. Some cross-cutting concerns use aspects that will allow more flexibility in the future (i.e. notification/communication) and are in their nascent stage (denoted in red).

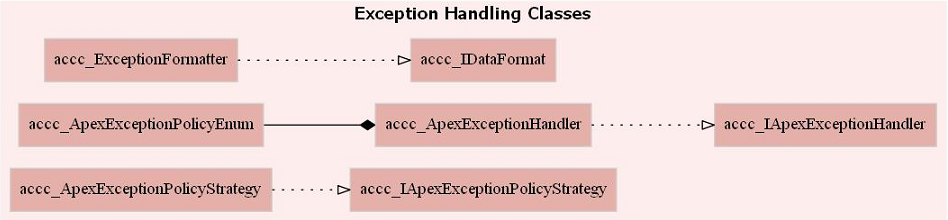
1. Logging



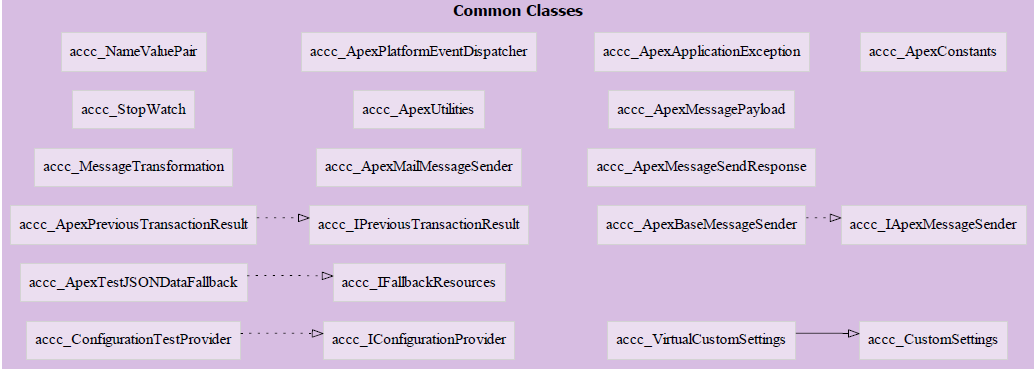
1. Caching



1. Exception Handling



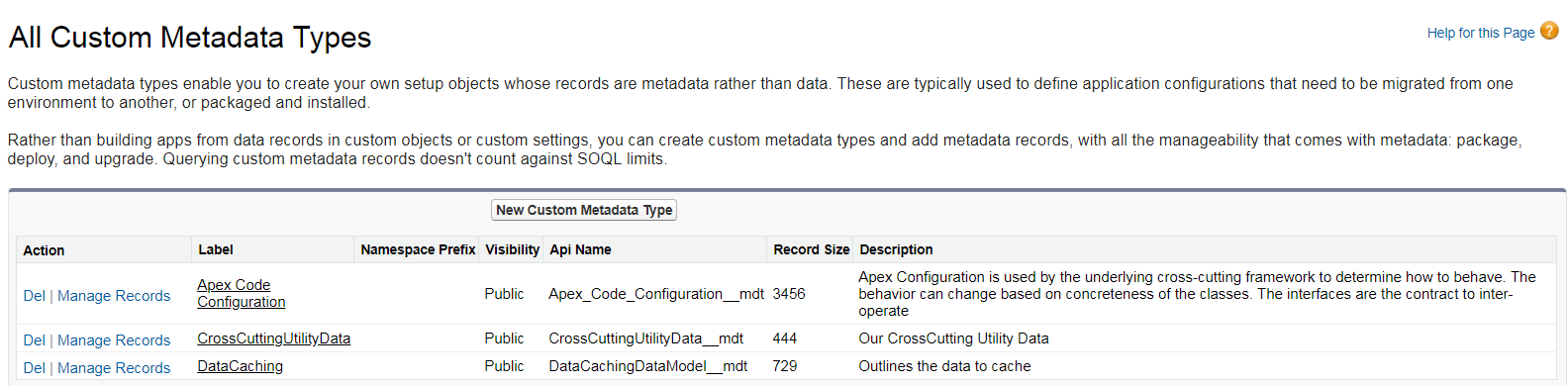
1. Configuration, Factory and Miscellaneous



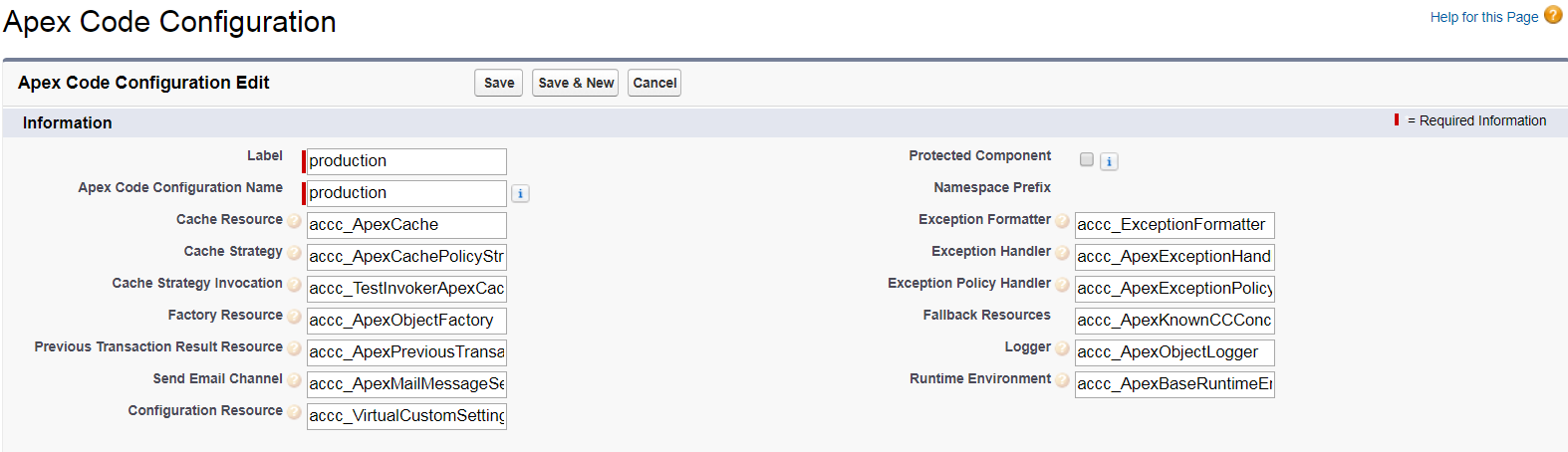
| Interface | Comments |
| --- | --- |
| accc\_IApexCacheStrategyInvocation | How to handle caching. Based on the policy (*DataCachingDataModel\_\_mdt*) and SOQL query determine if caching is allowed and for how long (milliseconds). |
| accc\_IApexExceptionHandler | How to handle exceptions. This could be logging the exception thru various channels and/or returning a more, customer friendly, or secure exception. |
| accc\_IApexExceptionPolicyStrategy | How to handle exceptions. Based on policy and exception type determine whether to re-throw the exception, return null, or provide a different exception (translated) |
| accc\_IApexFactory | The default factory used for creating new objects. One can add creators via *'add’*, look up the type based on name to create and instance. |
| accc\_IApexMessageSender | Send information over object boundaries/channels. The future goal here is the ability to send a message (sync/async) correctly. For example, if a message fails to send, should it queue it for resend, is there a *CircuitBreaker* involved, much yet to be determined. |
| accc\_IApexRunEnvironment | Provides the defined interfaces used in the runtime environment. This includes a configuration provider, logger, cache, fallback resources, exception handling, etc. |
| accc\_ICache | How to get/put information to the cache |
| accc\_IConfigurationProvider | Configuration providers allow common access to common sink such as custom settings, custom metadata types, custom objects, etc. |
| accc\_IDataFormat | How to format incoming data. This interface is used to format Exceptions (*accc\_ExceptionFormatter*). |
| accc\_IFallbackResources | This interface defines resources that one falls back to in the event their configuration data, object, or various sink is not available. For example, fallback resources could be a JSON configuration file (parsed Name/Value pairs). |
| accc\_ILogger | Defines the interface contract for logging |
| accc\_IPreviousTransactionResult | Defines the interface on how to process ***Database.SaveResults*** |
|  |  |
|  |  |

# Custom Metadata Types

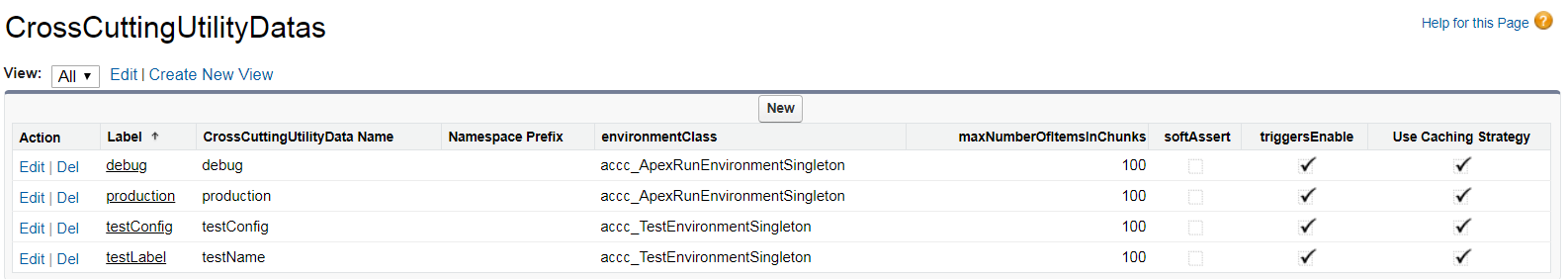
In order to allow a more reflective approach custom metadata types are used. There are three core custom metadata types,



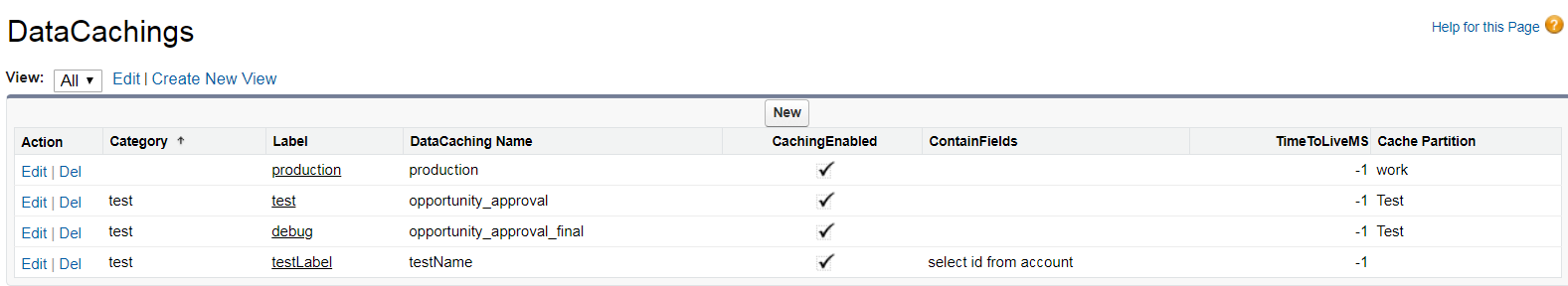
1. **Apex Cross-Cutting Configuration** – holds the concrete classes per environment. There are three supported environments (a) *production*, (b) *debug*, (c) *test*.



1. **Cross-Cutting Utility** –holds the common information per environment as well as global settings.



1. **Data Caching** – provides the cache strategy information on the name of the cache element, what to cache, how long to cache, the partition and if enabled.



# How does this work?

There is quite a bit of information to consume unfortunately; however, once you understand the core scenario’s it should be more palatable. The best way is to present a scenario to help understand the flow.

The logging scenario applies to all of the cross-cutting concerns.

## High-Level Flow of the Runtime Environment (Logging)

The framework relies on interfaces and custom metadata type to control behavior. In this scenario, the developer is asked to log/write information in their function/method as follows:

* In the test environment write to the Debug Log.
* In debug environment, write to the Application Log (custom object).
* In production, do not write information (no-op).

Most developers would be writing specific code with entangled logic; making it tightly coupled, difficult to maintain, with no reuse, no consistency and error-prone. However, with the use of the CCC framework, you write it ONCE (the same way).

Static void someFunction(String param1, Object param2)

{

*accc\_ApexRunEnvironmentSingleton.getInstance().logger().write(‘Entered “someFunction” , param1=’ + param1);*

try {

*accc\_ApexRunEnvironmentSingleton.getInstance().logger().write(‘Entered before “dowork”’);*

// do some work

this.dowork(param1, param2);

*accc\_ApexRunEnvironmentSingleton.getInstance().logger().write(‘Entered after “dowork”’);*

*:*

*:*

} catch (Exception excp) {

*accc\_ApexRunEnvironmentSingleton.getInstance().exceptionHandler().process(excp);*

*}*

*accc\_ApexRunEnvironmentSingleton.getInstance().logger().write(‘Exited “someFunction”’);*

}

Figure 2 Sample Code to Log

The above code, can work differently, because of inheritance and custom metadata types. Here is how:

1. *ApexRunEnvironmentSingleton* usesa runtime environment*, accc\_ApexBaseRuntimeEnvironment* which inherits from accc\_*IApexRunEnvironment.*
2. The runtime environment uses a factory, *accc\_ApexObjectFactory*, to define the cross-cutting behavior.
3. The cross-cutting behavior, is extracted from the custom metadata type (**Apex Code Configurations)** via the *accc\_CrossCuttingConfigMgr.*
4. The *accc\_CrossCuttingConfigMgr* class determines which environment (production, test, debug) and reads the data that defines which concrete class to support logging.

In the custom metadata type below (**Apex Code Configurations),** three environments (based on the ‘*Label’*) are defined with three different loggers (see the red outlined below). The three loggers have different behavior,

1. **[DEBUG]** *accc\_ApexObjectLogger*, writes to the Application Log object,
2. **[PROD]** *accc\_ApexNoOpLogger*, does not write anything,
3. **[TEST]** *accc\_ApexLogger*, writes to the **system.debug** log.

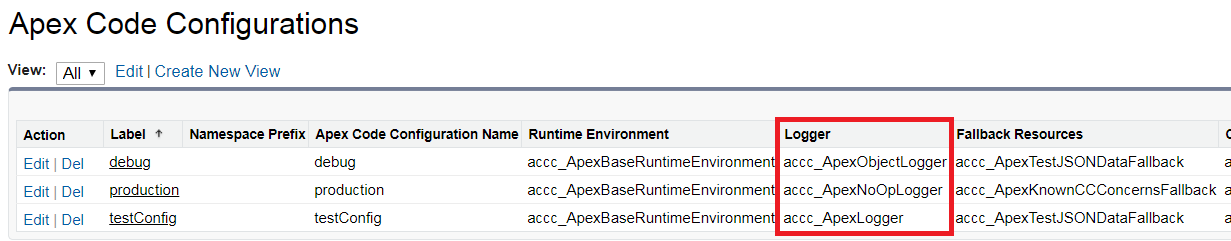
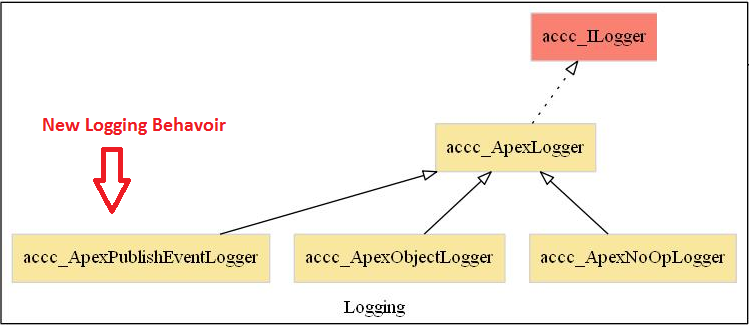


Figure 3 Cross-Cutting Concerns that holds the concrete classes

### 

Figure 4 Runtime Environment dynamically changing Log behavior

If the behavior needs to be different you can change the associated concrete class in the custom metadata type. For example, in the above scenario, there could be a use case to publish and log events. This can easily be accomplished by defining a new logger, *accc\_ApexPublishEventLogger*, which publishes log events (*Log\_e*).



Furthermore, if you wanted ALL logging to go to the default Debug Log, ensure ALL environments used *accc\_ApexLogger*.

Figure 5 accc\_ApexPublishEventLogger added

## Debug Scenario

In production your code should be stable, well tested and functioning. However, this sometimes is not the case. You don’t wish to run the debugger in production nor are you certain where things are breaking.

The two aspects that come into play are (1) *Exception Handling* (2) *Logging*. These two aspects are common aspects, if handled correctly, can make most debugging efforts easier.

### Exception Handling[[3]](#footnote-3)

Properly wrapping challenging code in a try/catch block is the first step. Please note, handling an exception with an exception handler is NOT always needed. Exception handling is primarily designed to simplify exception handling and management at your application or layer boundaries. The information below provides a high-level view.

#### High-Level Exception Scenario

In this bad scenario below an exception is caught and not handled.

Static void someFunction(String param1, Object param2)

{

try {

// do some work

this.dowork(param1, param2);

} catch (Exception excp) {

// do nothing

}

}

Figure 6 Bad Scenario

In the above scenario catching an exception without proper handling does not provide help when a problem arises. Sometimes methods/function that use a try/catch block will return a value that provides a clue; however, in non-return methods, it leaves a gap. Instead, a simple change provides the needed information.

Static void someFunction(String param1, Object param2)

{

try {

// do some work

this.dowork(param1, param2);

} catch (Exception excp) {

*accc\_ApexRunEnvironmentSingleton.getInstance().exceptionHandler().process(excp);*

*}*

}

Figure 7 Better Scenario

The above code uses the exception handler to process the exception. The processing of the exception means:

* Wrap the exception,
* Replace the exception,
* Allow the exception to propagate,
* Log the exception and re-throw a different (user-friendly) exception

There are additional parameters to the exception handler however, for simplicity, the exception is logged and handled (which can be changed).

1. Static Object Orient Languages [↑](#footnote-ref-1)
2. **S**ingle Responsibilities, Open-Closed, **L**iskov Substitutability, **I**nterface Segregation, **D**ependency Inversion. [↑](#footnote-ref-2)
3. Exception Handling was modeled after [Microsoft Enterprise Library](http://download.microsoft.com/download/D/2/0/D204F45F-4B44-4A0D-BC08-F9FA10BA0896/Developer's%20Guide%20to%20Microsoft%20Enterprise%20Library%20-%202nd%20Edition.pdf) [↑](#footnote-ref-3)