# JJ’s Reference Architecture

***-***

*Author: Jan-Joost van Zon*

*Date: December 2014 – June 2017*

*Under Construction*

## Contents

[Contents 1](#_Toc486534350)

[Introduction 8](#_Toc486534351)

[Definition of Software Architecture 8](#_Toc486534352)

[Inter-Disciplinary Aspects 8](#_Toc486534353)

[From Definite Choices to Possible Choices 9](#_Toc486534354)

[The Technical Approach 9](#_Toc486534355)

[Fundamental Principles 10](#_Toc486534356)

[Top 12 Code Improvements 11](#_Toc486534357)

[Layers 11](#_Toc486534358)

[Data Layer 12](#_Toc486534359)

[Presentation Layer 13](#_Toc486534360)

[Business Layer 16](#_Toc486534361)

[Perpendicular Layers 17](#_Toc486534362)

[Alternatives 18](#_Toc486534363)

[Namespaces, Assemblies and Folder Structure 18](#_Toc486534364)

[General Structure 18](#_Toc486534365)

[Root Namespace / Company Name 19](#_Toc486534366)

[Main Layers 19](#_Toc486534367)

[Business Domains 19](#_Toc486534368)

[Technologies 19](#_Toc486534369)

[Test Projects 20](#_Toc486534370)

[Details 20](#_Toc486534371)

[One Class, One File 20](#_Toc486534372)

[Lone Classes (bad) 20](#_Toc486534373)

[‘Scramled’ Technical and Functional Concerns 20](#_Toc486534374)

[Patterns 21](#_Toc486534375)

[Data Access Patterns 21](#_Toc486534376)

[Entity 21](#_Toc486534377)

[Mapping 22](#_Toc486534378)

[DTO 22](#_Toc486534379)

[Repository 22](#_Toc486534380)

[Repository Interfaces 22](#_Toc486534381)

[Business Logic Patterns 23](#_Toc486534382)

[Business layer 23](#_Toc486534383)

[Repository Wrappers 23](#_Toc486534384)

[Validators 23](#_Toc486534385)

[Side Effects 24](#_Toc486534386)

[LinkTo 24](#_Toc486534387)

[Cascading Extensions 25](#_Toc486534388)

[Manager / Facade 25](#_Toc486534389)

[Visitor 26](#_Toc486534390)

[String Resources 27](#_Toc486534391)

[Presentation Patterns 27](#_Toc486534392)

[ViewModel 27](#_Toc486534393)

[Lookup Lists 28](#_Toc486534394)

[ToViewModel 29](#_Toc486534395)

[ToEntity 30](#_Toc486534396)

[Presenter 30](#_Toc486534397)

[ToEntity-Business-ToViewModel Round-Trip 30](#_Toc486534398)

[NullCoalesce (ViewModels) 31](#_Toc486534399)

[Views 32](#_Toc486534400)

[First Full Load – Then Partial Load – Then Native Code 32](#_Toc486534401)

[Temporary ID’s 33](#_Toc486534402)

[Stateless and Stateful 33](#_Toc486534403)

[Considerations 34](#_Toc486534404)

[Presentation Patterns (MVC) 34](#_Toc486534405)

[Controller 34](#_Toc486534406)

[Post-Redirect-Get 34](#_Toc486534407)

[ValidationMessages in ModelState 35](#_Toc486534408)

[Polymorphic RedirectToAction / View() 35](#_Toc486534409)

[Html.BeginCollection 36](#_Toc486534410)

[Return URL’s 37](#_Toc486534411)

[Back Buttons 38](#_Toc486534412)

[Data Transformation Patterns 39](#_Toc486534413)

[Converter 39](#_Toc486534414)

[TryGet-Insert-Update 39](#_Toc486534415)

[TryGet-Insert-Update-Delete / Full-CRUD Conversion / Collection Conversion 39](#_Toc486534416)

[DocumentModel 41](#_Toc486534417)

[Selector-Model-Generator-Result 41](#_Toc486534418)

[Other Patterns 42](#_Toc486534419)

[Accessor 42](#_Toc486534420)

[Adapter 42](#_Toc486534421)

[Anti-encapsulation 42](#_Toc486534422)

[Initialization and Finalization 43](#_Toc486534423)

[Constructor Inheritance 43](#_Toc486534424)

[DebuggerDisplays 43](#_Toc486534425)

[Executor 43](#_Toc486534426)

[Inheritance-Helper 43](#_Toc486534427)

[Factory 44](#_Toc486534428)

[Factory-Base-Interface 44](#_Toc486534429)

[TryGet 45](#_Toc486534430)

[Get-TryGet-GetMany 45](#_Toc486534431)

[Helper 46](#_Toc486534432)

[Info 46](#_Toc486534433)

[IsSupported 46](#_Toc486534434)

[Mock 46](#_Toc486534435)

[Name Constants 46](#_Toc486534436)

[NullCoalesce 47](#_Toc486534437)

[Plug-In Model 47](#_Toc486534438)

[Progress and Cancel Callbacks 47](#_Toc486534439)

[Singular, Plural, Non-Recursive, Recursive and WithRelatedEntities 47](#_Toc486534440)

[Wrapper 49](#_Toc486534441)

[Alternatives 49](#_Toc486534442)

[Rich Models 49](#_Toc486534443)

[Aspects 50](#_Toc486534444)

[Authoring & Reviewing 50](#_Toc486534445)

[Caching 50](#_Toc486534446)

[Calculation 50](#_Toc486534447)

[Cascading 50](#_Toc486534448)

[Circular References 51](#_Toc486534449)

[Cloning 51](#_Toc486534450)

[Collections / List Processing 51](#_Toc486534451)

[Specialized Lists 51](#_Toc486534452)

[Concurrency 51](#_Toc486534453)

[Alternatives 51](#_Toc486534454)

[Configuration 52](#_Toc486534455)

[Custom Configuration Sections 53](#_Toc486534456)

[appSettings 54](#_Toc486534457)

[connectionStrings 54](#_Toc486534458)

[Conversion 55](#_Toc486534459)

[Defaults 55](#_Toc486534460)

[Debugging 55](#_Toc486534461)

[Entity Model / Data Model 55](#_Toc486534462)

[Entity Status Management 55](#_Toc486534463)

[Alternatives 55](#_Toc486534464)

[Enums 56](#_Toc486534465)

[General Rules 56](#_Toc486534466)

[Enum-Like Entities 57](#_Toc486534467)

[Localization 58](#_Toc486534468)

[TODO 59](#_Toc486534469)

[Errors 59](#_Toc486534470)

[Exceptions 59](#_Toc486534471)

[Facades 60](#_Toc486534472)

[Inverse Relationship Management / Inverse Property Management 61](#_Toc486534473)

[IO 61](#_Toc486534474)

[Logging 61](#_Toc486534475)

[Multi-Language / Translations / Culture 62](#_Toc486534476)

[Separate Content Items 62](#_Toc486534477)

[Loosely Linked Translation Entities 62](#_Toc486534478)

[Many Foreign Keys 62](#_Toc486534479)

[Comparison Loosely Linked vs Many Foreign Keys 62](#_Toc486534480)

[Naming 63](#_Toc486534481)

[Paging 63](#_Toc486534482)

[Parsing 63](#_Toc486534483)

[Performance 63](#_Toc486534484)

[Persistence 64](#_Toc486534485)

[Platform Compatibility 64](#_Toc486534486)

[Reflection 64](#_Toc486534487)

[Scheduling 64](#_Toc486534488)

[Security 64](#_Toc486534489)

[Side Effects 66](#_Toc486534490)

[Text Processing 66](#_Toc486534491)

[Transactions 66](#_Toc486534492)

[Unit Testing 66](#_Toc486534493)

[User Interface 67](#_Toc486534494)

[Utilities 67](#_Toc486534495)

[Validation 68](#_Toc486534496)

[API’s 68](#_Toc486534497)

[AJAX 69](#_Toc486534498)

[Embedded Resources 69](#_Toc486534499)

[Entity Framework 5 69](#_Toc486534500)

[JavaScript / TypeScript 69](#_Toc486534501)

[NHibernate 69](#_Toc486534502)

[ORM 69](#_Toc486534503)

[SQL 70](#_Toc486534504)

[With NHibernate 72](#_Toc486534505)

[Files instead of Embedded Resources 73](#_Toc486534506)

[Strings instead of Embedded Resources: 73](#_Toc486534507)

[XML 73](#_Toc486534508)

[Framework.Business Relationships 73](#_Toc486534509)

[Practices & Principles 74](#_Toc486534510)

[Responsibilities 74](#_Toc486534511)

[Separation of Concerns 74](#_Toc486534512)

[Combination of Concerns 76](#_Toc486534513)

[2 API’s for the Same Thing (bad) 76](#_Toc486534514)

[Copy-Paste Programming (bad) 76](#_Toc486534515)

[Do Not Repeat Yourself (DRY) 76](#_Toc486534516)

[Dump Code Line Here (bad) 76](#_Toc486534517)

[God-Object (bad) 76](#_Toc486534518)

[God Base Class (bad) 77](#_Toc486534519)

[Granularity 77](#_Toc486534520)

[Helperitis (bad) 77](#_Toc486534521)

[Spread Responsibility (bad) 77](#_Toc486534522)

[Too Many Responsibilities (bad) 77](#_Toc486534523)

[Two Solutions for the Same Thing (bad) 78](#_Toc486534524)

[Vague Justification (bad) 78](#_Toc486534525)

[Error Checking 78](#_Toc486534526)

[Be Stict 78](#_Toc486534527)

[Error Hiding / Null-Tollerance (bad) 78](#_Toc486534528)

[Null-Checks 78](#_Toc486534529)

[Process that Checks Itself (generally bad) 80](#_Toc486534530)

[Reject, Don’t Correct 81](#_Toc486534531)

[Interfacing 81](#_Toc486534532)

[‘All’ and ‘Many’ 81](#_Toc486534533)

[Blackboxing and Whiteboxing 81](#_Toc486534534)

[Clarity over Brevitity 82](#_Toc486534535)

[Conceptual Names (bad) 82](#_Toc486534536)

[CRUD 82](#_Toc486534537)

[Delegitis (bad) 83](#_Toc486534538)

[Dependency Injection 83](#_Toc486534539)

[Entity Design 85](#_Toc486534540)

[Execution Order Dependence (bad) 85](#_Toc486534541)

[Handy Extras / Ya Ain’t Gonna Need It (bad) 85](#_Toc486534542)

[Hatch / ‘Doorgeefluik’ (generally bad) 85](#_Toc486534543)

[Hollow Interface 86](#_Toc486534544)

[Interface Contamination 86](#_Toc486534545)

[Interface Neutrality 86](#_Toc486534546)

[Interface Stability 86](#_Toc486534547)

[IO Transparency 87](#_Toc486534548)

[Kama-Sutra Pattern (bad) 87](#_Toc486534549)

[Leaky Abstractions (bad) 87](#_Toc486534550)

[Loose Coupling 88](#_Toc486534551)

[Lying Names 88](#_Toc486534552)

[Magic (bad) 88](#_Toc486534553)

[Magic Defaults (generally bad) 88](#_Toc486534554)

[Magic Numbers / Magic Strings (bad) 89](#_Toc486534555)

[Method Self-Sufficient 89](#_Toc486534556)

[Ripple-Effect 89](#_Toc486534557)

[See from Name, not from Arguments 89](#_Toc486534558)

[Spooky Action (at a Distance) / Cause and Effect too Far Apart 89](#_Toc486534559)

[Syntactic Sugar 89](#_Toc486534560)

[Unclear Interfaces 90](#_Toc486534561)

[The Unwritten Agreement (‘het onderonsje’) (bad) 90](#_Toc486534562)

[Wrapperitis (bad) 91](#_Toc486534563)

[Variables and Parameters 91](#_Toc486534564)

[Double Negatives (bad) 91](#_Toc486534565)

[High-Throughput (bad) 91](#_Toc486534566)

[Keep the Names Consistent 91](#_Toc486534567)

[Many Properties (generally bad) 92](#_Toc486534568)

[Methods Instead of Parameters (good) 92](#_Toc486534569)

[Pass on and Assign (generally bad) 92](#_Toc486534570)

[Returning a Parameter (bad) 92](#_Toc486534571)

[Temporary Variables (good) 93](#_Toc486534572)

[Unused Parameter (bad) 93](#_Toc486534573)

[Variable not Declared Where it is Used (bad) 93](#_Toc486534574)

[Variables that Change Meaning (bad) 94](#_Toc486534575)

[Method Bodies 94](#_Toc486534576)

[Auto-Instatiation 94](#_Toc486534577)

[Constructor Calls an Overridable (bad) 94](#_Toc486534578)

[Cross-Referencing (generally bad) 95](#_Toc486534579)

[Empty If-Block (generally bad) 95](#_Toc486534580)

[Foreach with i 95](#_Toc486534581)

[Last Loop Item 95](#_Toc486534582)

[Method too Long / Class too Long 95](#_Toc486534583)

[Nested Loops (sometimes bad) 96](#_Toc486534584)

[Nesting too Deep 96](#_Toc486534585)

[Toilet-Role Principle 97](#_Toc486534586)

[Strategy 97](#_Toc486534587)

[Abstract / Concrete 97](#_Toc486534588)

[Anti-programming 97](#_Toc486534589)

[Asymmetry (bad) 98](#_Toc486534590)

[Bug Solving 98](#_Toc486534591)

[Bottom-Up and Top-Down 98](#_Toc486534592)

[Cartesian Product of Features Problem 98](#_Toc486534593)

[Chicken and Egg 99](#_Toc486534594)

[Consistent Stupidity 99](#_Toc486534595)

[Core of the Problem 99](#_Toc486534596)

[Delete + Insert != Update 99](#_Toc486534597)

[Distortion (usually bad) 100](#_Toc486534598)

[Dot It Right, Or Don’t Do It At All 100](#_Toc486534599)

[Double Stitch 100](#_Toc486534600)

[First Try Specific, Then Try Generic 100](#_Toc486534601)

[Fluff 100](#_Toc486534602)

[Ghost Hunt 100](#_Toc486534603)

[GNUID (bad) 101](#_Toc486534604)

[Hard-Coding and Soft-Coding 101](#_Toc486534605)

[Hit F5 and See 101](#_Toc486534606)

[Ideal Solution 101](#_Toc486534607)

[Inheritance not Always Good 101](#_Toc486534608)

[It Works, Doesn’t It? (‘Maar het werkt toch?’) (bad) 101](#_Toc486534609)

[Least Possible Effort Strategy (bad) 102](#_Toc486534610)

[Liskov-Substution Principle (SOLID) 102](#_Toc486534611)

[Unused Functions (bad) 102](#_Toc486534612)

[Open/Closed Principe (SOLID) 102](#_Toc486534613)

[Open Ends (bad) 103](#_Toc486534614)

[Power of Abstraction / Power of Generalization 103](#_Toc486534615)

[Quick and Dirty / Dirty (usually bad) 103](#_Toc486534616)

[Readable, Writable and Rewritable Code 103](#_Toc486534617)

[Subtractive and Additive 103](#_Toc486534618)

[Testing 104](#_Toc486534619)

[Too Difficult / Disproportional Effort 104](#_Toc486534620)

[Tooleritis 104](#_Toc486534621)

[Trade-Offs 104](#_Toc486534622)

[Whirlpool Anti-Pattern / Inappropriate Conversions 105](#_Toc486534623)

[SOLID 105](#_Toc486534624)

[Service Architecture 105](#_Toc486534625)

[The ESB Concept 105](#_Toc486534626)

[Canonical Model 106](#_Toc486534627)

[Less Integration Code 106](#_Toc486534628)

[Clearer Integration Code 107](#_Toc486534629)

[In Practice 107](#_Toc486534630)

[Standard ESB vs Custom ESB 108](#_Toc486534631)

[ESB Model 108](#_Toc486534632)

[Enterprises 108](#_Toc486534633)

[ConnectionTypes 108](#_Toc486534634)

[Connections 108](#_Toc486534635)

[Keys 108](#_Toc486534636)

[Transmissions 108](#_Toc486534637)

[Service Implementations 108](#_Toc486534638)

[Multi-Dispatch 109](#_Toc486534639)

[Namespaces 109](#_Toc486534640)

[Service-Related Patterns 110](#_Toc486534641)

[Facade 110](#_Toc486534642)

[Hidden Infrastructure 110](#_Toc486534643)

[TODO 111](#_Toc486534644)

[Database Conventions 111](#_Toc486534645)

[Developing a Database 111](#_Toc486534646)

[Naming Conventions 112](#_Toc486534647)

[Rules 112](#_Toc486534648)

[Upgrade Scripts 113](#_Toc486534649)

[Excel Sheet 113](#_Toc486534650)

[Scripts 115](#_Toc486534651)

[Deployment 115](#_Toc486534652)

[SqlScripts table 116](#_Toc486534653)

[C#-Based Migrations 117](#_Toc486534654)

[Code Style 118](#_Toc486534655)

[Casing, Punctuation and Spacing 118](#_Toc486534656)

[Tri vial Rules 121](#_Toc486534657)

[Miscellaneous Rules 123](#_Toc486534658)

[Namespace Tips 125](#_Toc486534659)

[Member Order 125](#_Toc486534660)

[Naming 125](#_Toc486534661)

[Boolean Names 125](#_Toc486534662)

[Class Names 126](#_Toc486534663)

[Collection Names 127](#_Toc486534664)

[DateTime Names 127](#_Toc486534665)

[Enum Names 127](#_Toc486534666)

[Event Names / Delegate Names 127](#_Toc486534667)

[Method Names 128](#_Toc486534668)

[File-Related Variable Names 129](#_Toc486534669)

[Miscellaneous Names 131](#_Toc486534670)

[Software Lifecycle 131](#_Toc486534671)

[Branching, Versioning & Release Management 131](#_Toc486534672)

[Team Management 131](#_Toc486534673)

[Focus Points 131](#_Toc486534674)

[Dialog 131](#_Toc486534675)

[Periodic Individual Meetings 131](#_Toc486534676)

[Reporting to Management 132](#_Toc486534677)

[Limiting Dialog 132](#_Toc486534678)

[Support 132](#_Toc486534679)

[Question Rotation 133](#_Toc486534680)

[Support & Direction 133](#_Toc486534681)

[Developer Skills 133](#_Toc486534682)

[Direction 133](#_Toc486534683)

[Miscellaneous Pointers 133](#_Toc486534684)

[Vary a Developer’s Tasks 134](#_Toc486534685)

[Upward Management 136](#_Toc486534686)

[Budgeting 136](#_Toc486534687)

[Planning 136](#_Toc486534688)

[Tasks 136](#_Toc486534689)

[Strategy 137](#_Toc486534690)

[Keep Your Cool, Be Open to Other Views 137](#_Toc486534691)

[Server Architecture 137](#_Toc486534692)

[DTAP 139](#_Toc486534693)

[Folders 139](#_Toc486534694)

[Development Workstation 141](#_Toc486534695)

[Backups 141](#_Toc486534696)

[Appendices 141](#_Toc486534697)

[Appendix A: Layering Checklist 141](#_Toc486534698)

[Appendix B: Knopteksten en berichtteksten in applicaties (resource strings) (Dutch) 142](#_Toc486534699)

[Hoofdletters, interpunctie, spelling 143](#_Toc486534700)

[Assemblies 143](#_Toc486534701)

[Tips 144](#_Toc486534702)

## Introduction

### Definition of Software Architecture

Software architecture has thousands of definitions. If you believe them all, then software architecture is about everthing that has to do with software development.

Mostly it is about actually building the software.

It is about modular design of software components, which can be done by splitting up code into functionalities, like a shopping basket, or financial reports, or splitting up code into technical aspects, like validation, security and persistence. You can also split up by both technical aspects and functional aspects at the same time, giving you a more fine-grained (2-dimensional!) separation of concerns.

Software architecture also has a strong focus on maintainability and being prepared the future as a system grows.

The programming side of software architecture is about making frameworks, coding the functionalities, combining different technologies and using best practices. It also involves technical design, which by the way you can do in your head as well as on paper.

### Inter-Disciplinary Aspects

Software architecture also involves technical details outside of software programming, such as the basic outlining of hardware infrastructure, collaborating with infrastructure technicians, server administrators, hosting providers, software vendors.

Software architecture also includes soft-skills that do not have much to do with technology. Planning the development of software both in rough outlines as well as task details, guarding that work, prioritizing, organizing and replanning, making concessions, work preparation, managing software lifecycle, going from design to implementation to test to production and after care, having proper source control in place, managing the team that codes, the team that tests, discussing functionalities, goals and planning with management, stakeholders, staff and end-users. Basically talking to anyone even slightly involved in the development of the software. Coaching developers, expanding the teams knowledge, making the team work optimally together and efficiently, and give people room to focus, so a lot of work gets done well. It can involve managing budgets for hardware and software and also functional designing.

Fortunately this does not need to come down to one person. Even though a software architect can overview the whole process, lots of tasks can be delegated to other team members, so you can make software architecture work as a team.

### From Definite Choices to Possible Choices

This is kind of a personal note on where this document stands right now.

Originally I described a fixed way of working here, that generally works well if you want to build large dependable systems with a lot of flexibility. I applied these methods of working in a team under my lead. It worked, but required a lot of discipline of team members to do things the way the boss wants.

I want to move away from this a little bit, and see the methods described here more like a suggestion box of different ways to do things. I will try to describe different alternatives next to the one I prefer and highlight the pros and cons, so you can perhaps see why I came to the conclusion that one method is better than the other.

Much of the document is still described in definites, rather than suggestions. As I find the time to work on this documentation, I will be changing the tone.

What you will also find is that I describe a lot of things you could do wrong. The suggestion is often that there is a better way to do it. I will try to reformulate things so they start with a positive approach rather than starting with the negative.

Currently (2017-06-28) it is full of TODO’s that indicate texts I still want to write or rough texts to polish up. So please be forgiving of those.

But, now: back to business.

### The Technical Approach

This document mostly goes into detail about technical aspects of software architecture. It talks about those aspect of software development, that go beyond the individual application: techniques and best practices that can be applied to the development of *any* application. You could call it ‘*functionality-independent software architecture’*.

This architecture could also be called a ‘*pattern stack’*, because it takes a technical-first approach, rather than functional-first in that a layering is described where fixed design patterns are used from one layer to the next. The rationale behind this is that even though technology changes fast, functionality changes faster, so a technical-first order will be less likely to change, and this results in a more stable subdivision into parts upon which the functionality builds.

<TODO: Use some of these phrasings: Preservability of functionality, versus technique. Functionality changes even faster than the technology itself. That is one of the reasons why the code has a technical-first ordering, not functional-first. And to accommodate quickly changing technology, we use abstraction of these technologies to be able to replace them and not have to reprogram the whole application. >

<TODO: Possibly Use this phrase about extensibility: (Dutch)“Alles *past* erin, ook al zit het er nog niet in.” “Everything fits in it, but not everything is in it yet.”>

There are two parts of this software architecture:

* Application architecture
* Service architecture

The *application architecture* is the main part. It is about business domains and everything you could show on a screen, including a framework of reusable parts. The *service architecture* is explained separately and is mostly about linking different systems together.

The way of working described here is just a suggestion. It describes *a* way of working, not *the* way of working. The described principles and practices can be used at will.

### Fundamental Principles

The main principles of this software architecture are:

* Maintainability
* Code scalability
* Platform and protocol independence

Code scalability does not refer to hardware scalability, but rather that the code base can grow and grow, while keeping it maintainable. 50 apps should be as maintainable as 5. This means that quality demands are high. As a companies’ amount of software products grows, software architecture is necessary to create an economically viable amount of software maintenance, or a company might run into problems.

Another way of putting this is: The next software change should not be more difficult than the previous one, regardless of how large the system has become.

Platform and protocol independence is something given extra attention in this software architecture. A lot of split up into parts is, due to the fact that not every technology is supported on every platform. This allows us to take our pick from technologies more easily.

This software architecture also puts a lot of focus on fixed patterns of working. These patterns are proven to work well, and if we all work the same way and understand the system of organization, we can more easily navigate the code, regardless of who wrote it.

### Top 12 Code Improvements

This document goes into detail about a lot of best practices. But to keep focus on what goes wrong most of the time, and would offer the greatest improvement of code, here is a list of practices, that if done right, may greatly increase the quality of your software.

1. Avoid code duplication
2. Avoid error hiding
3. Use clear names
4. Use correct indentation
5. Separation of concerns
6. Proper encapsulation

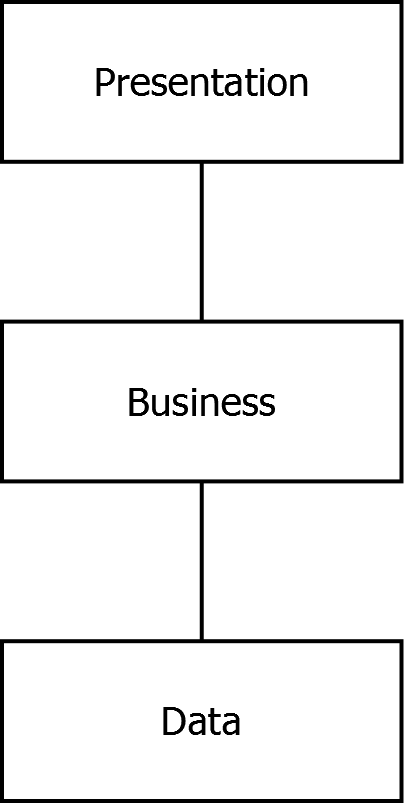
And second in line:

1. Reject, don’t correct
2. Clear interfaces
3. ‘No handy extras’ / ‘Ya ain’t gonna need it’
4. Solve a problem at its core, instead of making work-arounds.
5. To solve a bug, first reproduce it
6. Proper use of encoding

## Layers

This is a suggestion of how to split up your software into layers.

The software is split up into 3 layers:



The presentation layer contains the screens of the system.

The presentation layer calls the business layer, which is non-visual. It defines and enforces the rules of the system. Those are like the internal, mechanical parts of the system.

The business layer talks to the data layer, which models the business domain but does not process anything: it just stores and retrieves the data.

Data layer and presentation layer are programmed using fixed patterns. The business layer uses patterns too, but it gets a little more creative. If anything special needs to happen, this belongs in the business layer, since that is the machinery of the system.

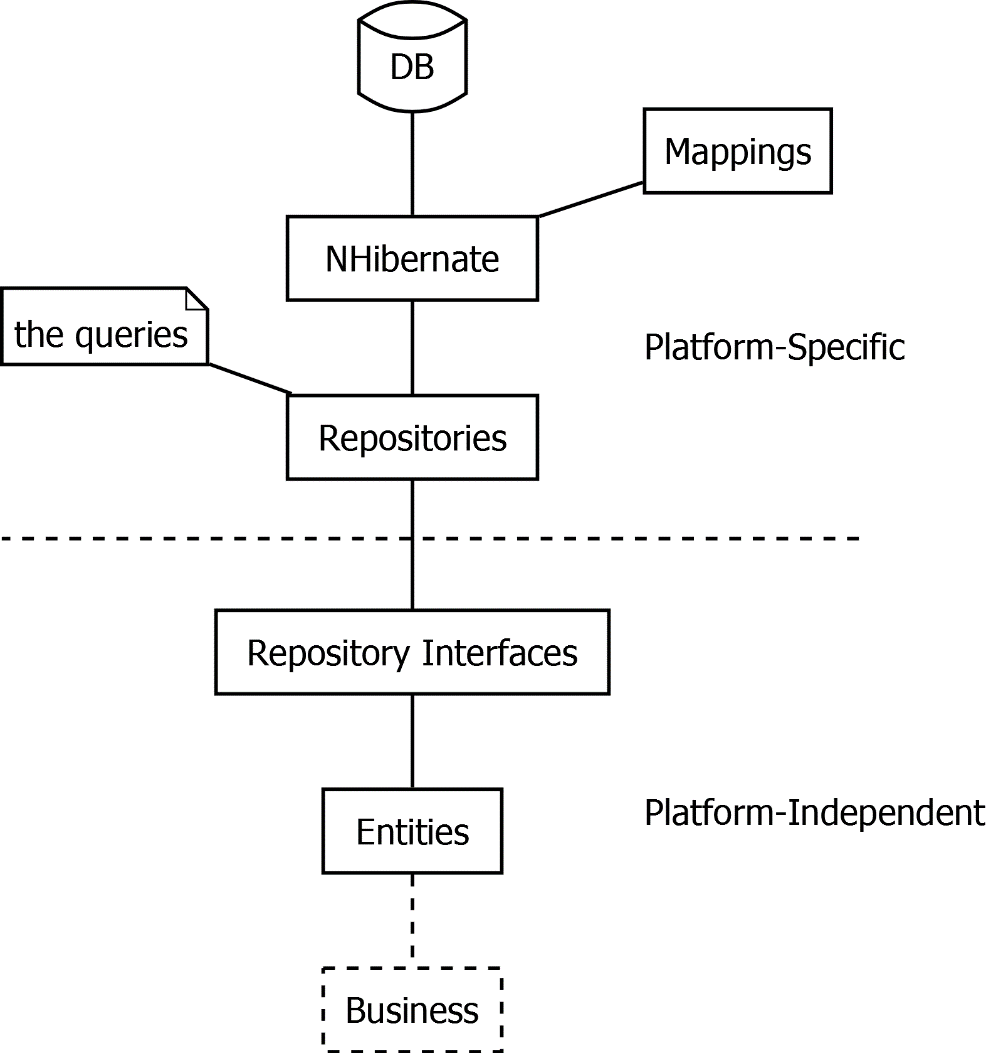
The data layer is also called the ‘data access layer’ or ‘persistence layer’.

The business layer is also referred to as ‘business logic’.

The presentation layer is sometimes referred to as the ‘front-end’.

### Data Layer

The data layer is built up of the following sub-layers:



It all starts with the database. The database is not directly accessed by the rest of the code, but the database is talked to through NHibernate, an object-relational mapper. NHibernate will translate database records to instance of classes. Those classes have properties, that map to columns in the database, and properties that point to related data. NHibernate needs to be given mappings, that define which class maps to which table and which columns map to which properties.

The data classes are called entities.

The entities are not directly read out of NHibernate by the rest of the code. The rest of the code talks to NHibernate through the repositories. You can see the repositories as a set of queries. Next to providing a central place to manage a set of optimal queries, the repositories also keep the rest of the code independent of NHibernate, in case you would ever want to switch to a different data storage technology.

The repository implementations are not used directly, but accessed through an interface, so that we can indeed use a different data access technology, just by instantiating a different repository implementation. The repository interfaces are also handy for testing, to create a fake in-memory data store, instead of connecting to a real database.

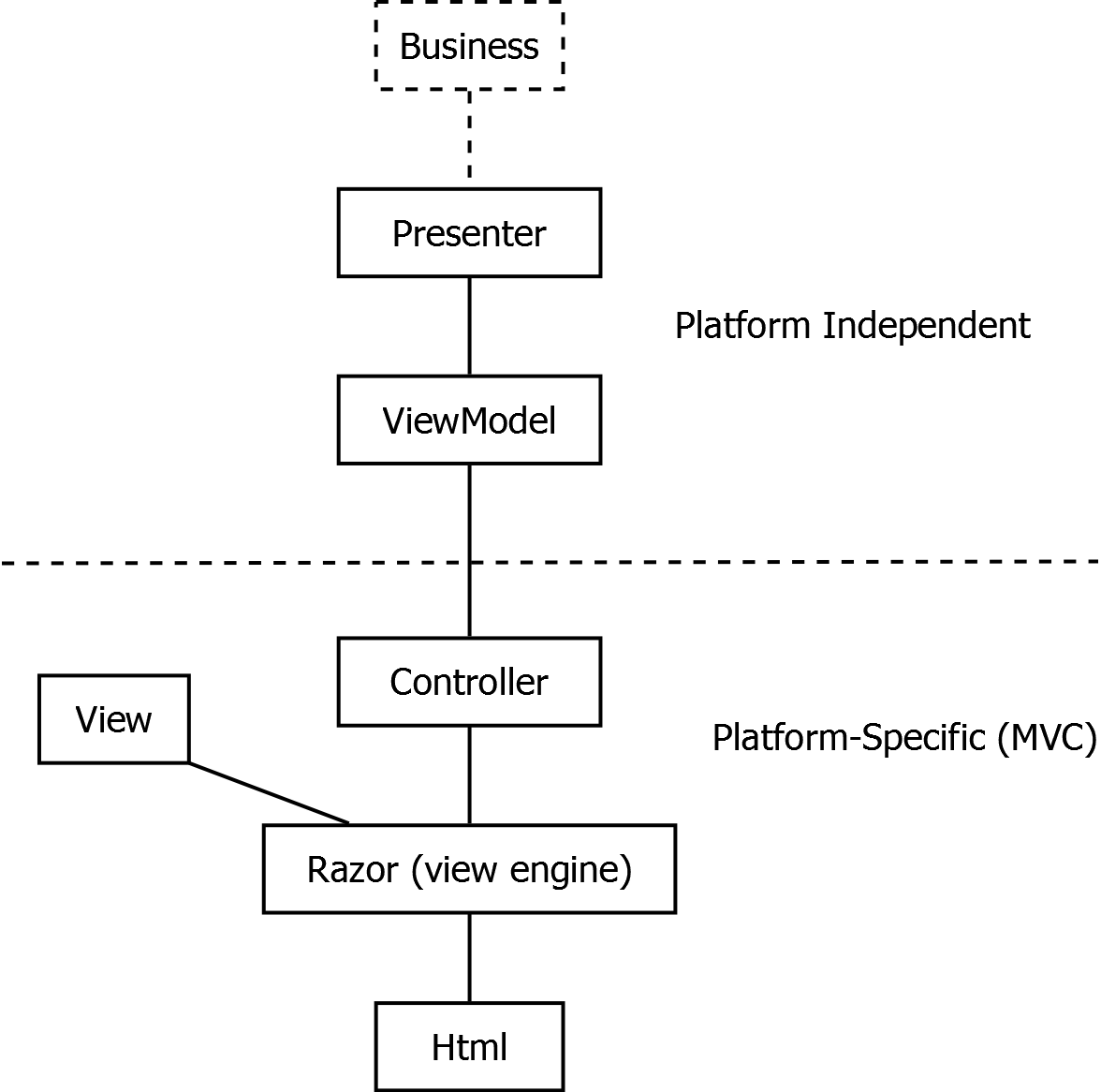
The dashed line going right through the diagram separates the platform-specific code from the platform independent code. The platform-specific code concerns itself with NHibernate and SQL Server, while the platform independent code is agnostic of what the underlying storage technology is. You may as well stick an XML file under it and not use SQL Server and NHibernate at all. This allows you to program against the same model, regardless of how you store it. This also allows you to deploy this code in any environment that can run .NET code, such as a mobile phone.

Because the architecture is multi-platform, the labels in the diagram are actually too specific:

* ‘DB’ can actually be any **data store** – that is the proper term for it: ‘data store’: an XML file, flat file or even just in-memory data.
* ‘NHibernate’ can be an any **persistence technology**: another ‘ORM’ (‘object relational mapper’), like Entity Framework, a technology similar to NHibernate. The persistence technology can also be simply writing to the file system, or an XML API, or SqlClient with which you can execute raw SQL.

### Presentation Layer

The presentation layer is built up of the following sub-layers:



<TODO: Put ToEntity en ToViewModel in the diagram.>

The presentation layer calls the business layer, which contains all the rules that surround the system.

The data that is exactly shown on screen is called the *view model*.

*Presenter* classes talk to the business layer. The presenter is responsible for translating the data and the results of the business logic to a subset of data that is shown on screen: the view model.

The presenter is also responsible for translating user input back to data, passing it through the business logic. The business logic then executes validations and side effects around the data access.

<TODO: Write about ToEntity / ToViewModel, etc., to indicate that the presenter is a combinator of things.>

The presenter layer forms a model of your program navigation. Each screen has its own presenter and each method in that presenter is a specific user action.

MVC is the web technology of choice we use for programming user interfaces. In our architecture the MVC layer builds on top of the presenter layer.

In MVC we use controllers, which are similar to presenters in that they group together related user actions and each user action has a specific method.

<TODO: This pragraph is all over the place. Split it up into paragraphs and add some a few sentences that ease you into it.>

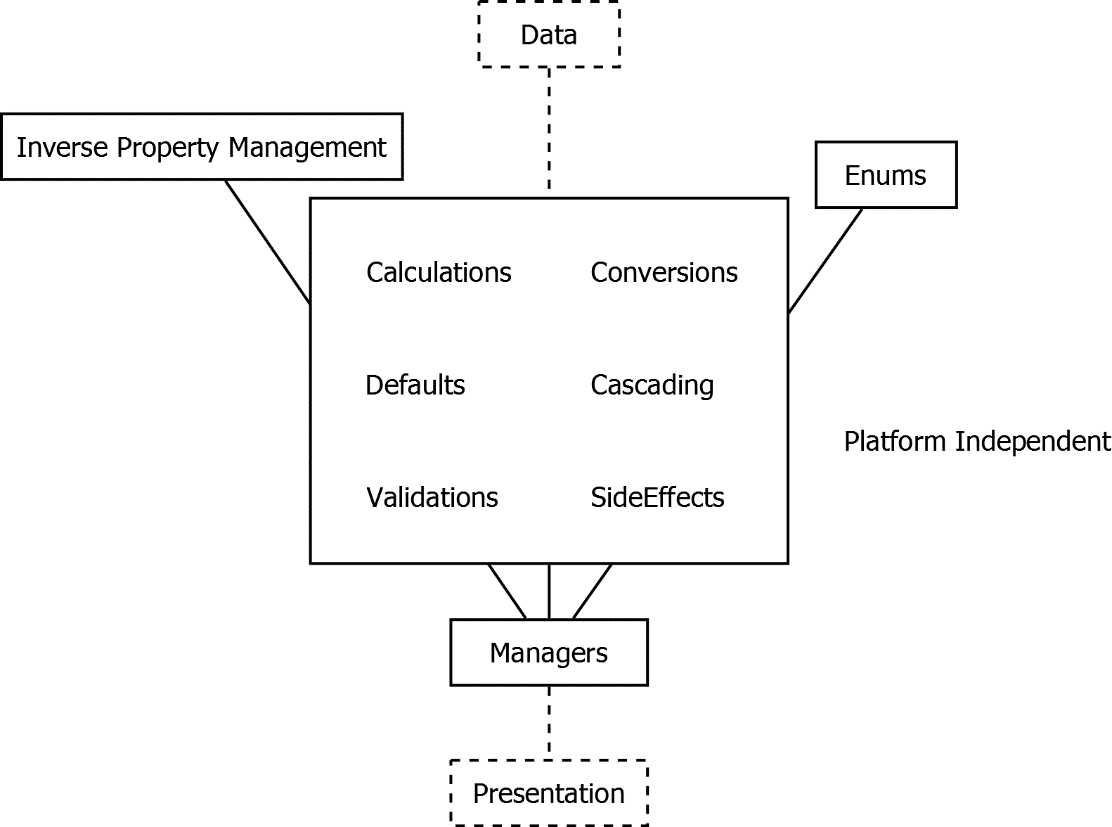
Each method in a controller represents a URL. The request from the web browser will lead to the controller method going off, which eventually results in a view engine rendering a piece of HTML. The view rendering goes off automatically. The view engine we use is Razor, which offers a concise syntax for programming views, in which you combine C# and HTML. Razor has tight integration with MVC. The view engine combines a view model and a view to form a specific piece of HTML.

The dashed line going right through the diagram separates the platform-specific code from the platform independent code. The platform-specific code concerns itself with MVC, HTML and Razor, while the platform independent code is agnostic of what presentation technology we use. That means that we can use multiple presentation techniques for the same application navigation model, such as offering an application both web based as well as based on WinForms. This also provides us the flexibility we need to be able to deploy apps on mobile platforms using the same techniques as we would use for Windows or web.

Because the architecture is multi-platform, the labels in the diagram above are actually too specific:

* The *controller* is very specific to MVC and an equivalent might not even be present on other presentation platforms, even though it is advisable to have a central place to manage calls to the presenter and showing the right views depending on its result.
* The views in WinForms would be the *Forms and UserControls*. It is advised that even if a view can have ‘code-behind’ to only put dumb code in it and delegate the real work elsewhere.
* ‘Html’ can be replaced by the type of presentation output. In WinForms it is the controls you put on a form and their data. But it can also be a generated PDF, or anything that comes out of any presentation technology.

### Business Layer



<TODO: Keeping ‘Inverse Property Management’ and ‘Enums’ separate like that does not seem to add much to the clarity of the diagram. It is true that some aspects are used directly besides through the manager, but even those lines going from presentation directly to those aspects, are not present in the diagram, so that is kind of a fail. I think you could express that better.>

<TODO: Include ‘Resources’. in the diagram >

What is business logic? Basically anything that is not presentation or data access, is business logic.

<TODO: Layers: Say something about infrastructure, next to persistence, business and presentation. Because then you can say: everything that is not persistence, presentation or infrastructure, is business logic.>

The business layer resides in between the data access and the presentation layer. The presentation layer calls the business layer for the most part throught the Manager classes. The manager classes are combinators that combine multiple aspects of the business logic, by calling validators, side effects, cascadings and other things. They are ‘CRUD-oriented facades’.

The business layer executes validations that verify, that the data corresponds to all the rules. Also, the business layer executes side effects when altering data, for instance storing the date time modified or setting default values when you create an entity, or for instance automatically generating a name. The business layer is also responsible for calculations and many other things as represented in the diagram above.

The business layer uses entities, but sometimes will call repositories out of the data access layer, even though your first choice should be to just use the entities. The presentation layer uses the business layer for anything special that needs to be done. Often when something special is programmed in the presentation layer, it actually belongs in the business layer instead.

The business layer is platform independent and the code can be deployed anywhere. This does sometimes require specific API choices or using our own framework API’s. These choices are inherently part of this architecture. But because most things are built on entities and repository interfaces, the business logic is very independent of everything else, which means that the magic of our software can be deployed anywhere.

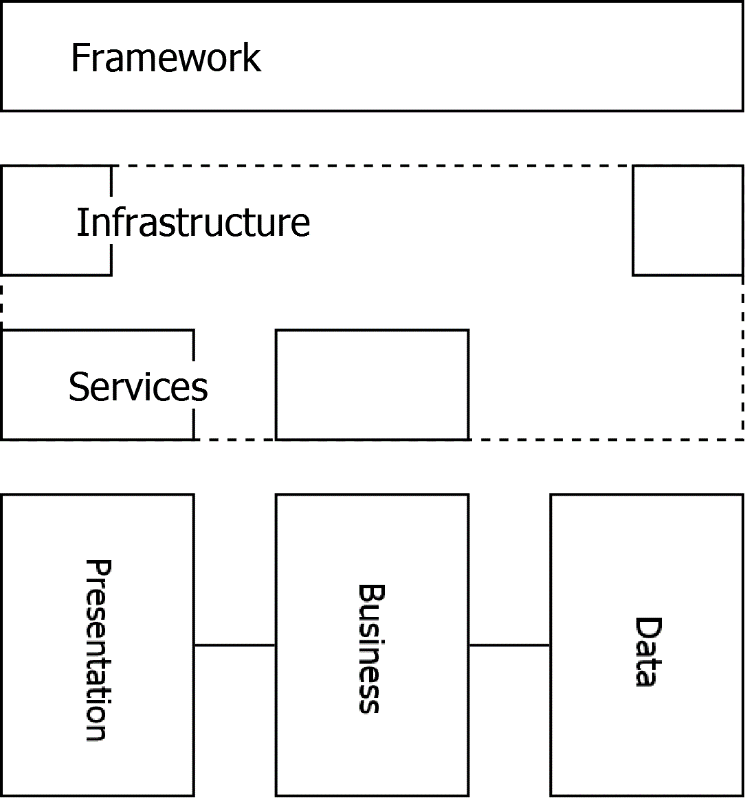
<TODO: Add ‘Cloning‘ to big block in the diagram? It might stay too vague if you mention it there. >

<TODO: Consider this:

- Mention in the layering diagrams that Inverse Property Management is also called LinkTo and Unlink in our architecture and that Cascading is also called UnlinkRelatedEntitiesExtensions and DeleteRelatedEntitiesExtensions. Whether you should pollute the diagrams with that is an open question, because it is a really specific choice that may be broken in the future. On the other hand, the diagrams serve to clarify and are specific to this architecture already.>

### Perpendicular Layers

The subdivision into data, business and presentation is just about the most important subdivision in software design. But there are other additional layers, called perpendicular layers:



The Framework layer consists of API’s that could support any aspect of software development, so could be used in any part of the layering. That is why it stretches right from Data to Presentation in the diagram.

Infrastructure is things like security, network connections and storage. The infrastructure can be seen as part at the outer end of the data layer and part at the outer end of the presentation layer, because the outer end of the data layer is actually performing the reading and writing from specific data source. However it is the presentation layer in which the final decision is made what the infrastructural context will be. The rest of the code operates independent of the infrastructure and only the top-level project determines what the context will be.

<TODO: Encorporate this phrase: It is hard to explain what the position of infrastructure is in the architecture. One thing you can say is that the infrastructure should be loose coupled. >

Services expose business logic through a network interface, often through the SOAP protocol. A service might also expose a presentation model to the outside world. Because it is about a specific network / communication protocol, the service layer is considered part of the infrastructure too.

Another funny thing about infrastructure, for example user right management, is that a program navigation model in the presenter layer can actually adapt itself to what rights the user has. In that respect the platform-independent presentation layer is dependent on the infrastructure, which is a paradox. The reason the presenter layer is platform-independent is that it communicates with the infrastructure using an interface, that may have a different implementation depending on the infrastructural context in which it runs.

### Alternatives

|  |  |  |
| --- | --- | --- |
|  | Benefits | Downsides |
| Data and Business in one layer | - Might be easier to understand | More likely for data access and business to get entangled |
| No repositories |  |  |

## Namespaces, Assemblies and Folder Structure

### General Structure

Solution files are put in the code root.

Assembly names, namespaces and folder structure are similar to eachother. An assembly’s name will be its root namespace. The folder structure will also correspond to the namespacing.

An assembly name is built up as follows:

Company.SoftwareLayer.BusinessDomain [.Technology] [.Test]

Internally in an assembly each pattern can get its own sub-folder:

Company.SoftwareLayer.BusinessDomain [.Technology] [.Test] [.DesignPattern]

If a project is very small, you might use a single sub-folder ‘Helpers’, instead of a folder for each design pattern.

When a project gets big, a design pattern folder might again be split up into partial domains or main entities:

Company.SoftwareLayer.BusinessDomain [.Technology] [.Test] [.DesignPattern] [.PartialDomain]

### Root Namespace / Company Name

In this architecture the root namespace will be the ‘company name’, for instance:

**JJ**

### Main Layers

The second level in the namespacing consists of the following parts:

|  |  |
| --- | --- |
| JJ.**Data** | The data layer including the entity models and persistence. |
| JJ.**Business** | The business logic layer |
| JJ.**Presentation** | The presentation layer |
| JJ.**Framework** | Contains any reusable code, that is independent from any domain model. Any layer in the software architecture can have reusable framework code to support it. |

And the less important:

|  |  |
| --- | --- |
| JJ.**Demos** | Demo code for educational purposes |
| JJ.**Utilities** | Processes that are not run very often. Utilities contains small programs for IT. E.g. load translations, things to run for deployment. |

### Business Domains

The third level in the namespacing is the business domain. A business domain can be present in multiple layers, or missing in a specific layer, an app can use multiple business domains, a single business domain can have multiple front-ends. Examples:

JJ.Data.**Calendar**

JJ.Business.**Calendar**

JJ.Presentation.**Calendar**

The ‘business domain’ of the framework layer is usually a technical aspect. Examples:

JJ.Framework.**Validation**

JJ.Framework.**Security**

JJ.Framework.**Logging**

### Technologies

The fourth level in the namespacing denotes the used technology. It is kind of analogus to a file extension. You can often find two assemblies: platform-independent and one platform-specific.

JJ.Data.Calendar

JJ.Data.Calendar.**NHibernate**

JJ.Presentation.Calendar

JJ.Presentation.Calendar.**Mvc**

JJ.Framework.Logging

JJ.Framework.Logging.**DebugOutput**

This means that the platform-indepent part of the code is separate from the platform-specific code. This also means, that much of the code is shared between platforms. It also means, that we can be very specific about which technologies we want to be dependent on.

### Test Projects

Every assembly can get a Test assembly, which contains unit tests. For instance:

JJ.Business.Calendar.**Tests**

JJ.Presentation.Calendar.Mvc.**Tests**

### Details

#### One Class, One File

The general rule is that all classes, interfaces, enums, etc. get their own file. The rule can be broken if the amount of classes really becomes big and also the rule does not count for nested classes. Also a single class can be spread among files, if they are partial classes.

#### Lone Classes (bad)

It is unhandy to have a whole bunch of your assembly’s folders just containing one class or very few classes. Consider moving those classes into other folders. Another solution could be to put them all together, for instance in a folder called ‘Helpers’, if they indeed are just simple helper classes.

#### ‘Scramled’ Technical and Functional Concerns

In our namespacing, the technical and functional pieces seem scrambled:

JJ.Business.Ordering.Validation.Products

These are the functional (or commercial) concerns:

**JJ**.Business.**Ordering**.Validation.**Products**

These are the technical concerns:

JJ.**Business**.Ordering.**Validation**.Products

The reason for ‘scrambling’ of technical and functional concerns, is rooted in that we are trying to project something 2-dimensional (funtional vs. technical) onto something sequential (written text). We could artificially keep functionality together and technical things together:

***JJ.Ordering.Products***.Business.NHibernate.Validation

JJ.Ordering.Products.***Business.NHibernate.Validation***

But this does not help us do our job.

What we instead try to do is organize things into bigger and smaller chunks. The split up into companies' intellectual property is the largest concern, while the second most important concern is the split up into main software layers (Data, Business, Presentation, etc.) A business domain is a larger concern than the specific technology used (e.g. NHibernate, Mvc). And a design pattern is a level of detail even below that.

Also: in the less recommended namespacing it is not obvious that JJ.Ordering.Products is about validating the products, while if you put the Products sub-namespace at the end (…Validation.Products) this is obvious.

Another problem with starting with JJ.Ordering.Products instead of JJ.Business is that it suggests that Ordering has a Data, Business and Presentation layer, while really Ordering does not need to be present in all layers. It gives a false sense that you create a Presentation layer it must be put into an already existing business domain or that a business domain must always have al three layers present. It would also suggests that a presentation layer in one business domain can only use one business layer. The reality is, that a presentation layer can use multiple business layers.

It is less confusing from a software design perspective to have all layers present and whether a business domain is present in a layer is optional. That makes it more obvious that there is an n-to-n relationship between layers and business domains.

But the ordering in the namespace is arbitrary. It just helps us to ‘scramble’ the namespace parts wisely, so that it goes from one level of detail to the next.

## Patterns

Design patterns are coding techniques to solve common programming problems. They bring consistency to the code. They help us reuse best practices and prevent code from becoming messy. They also are an extension to the software layering.

### Data Access Patterns

#### Entity

These are the classes that represent the domain model.

The entity classes simply contain properties of simple types or references or lists to other entities.

There will be no logic in the entity classes in our architecture.

Collections should be created in the constructor, because NHibernate does not always create them, and you do not want to check whether collections are null all over your code.

All public members should be virtual, otherwise persistence technologies can often not work with it.

Do not use inheritance within your entity model, because it can make using persistence technologies harder, error prone, and it can actually harm performance of queries.

##### Alternatives

Generally avoided, but not prohibited:

* Use inheritance anyway with the aforementioned downsides.
* Use interfaces for polymorphism instead.
* Instead of inheritance, consider a composition solution, rather than an inheritance solution.

##### Considerations

<TODO: Write some more about the difficulties of inheritance in entity models.>

#### Mapping

Mappings are classes programmed for a particular persistence technology, e.g. NHibernate, that map the entity model to how the objects are persisted in the data store (e.g. an SQL Server database).

#### DTO

DTO = Data transfer object. DTO’s only contain data, no logic. They are used solely to transfer data between different parts of the system, particularly in cases where passing an entity is not handy or efficient.

For instance: A specialized, optimized SQL query may return a result with a particular record structure. You could program a DTO that is a strongly typed version of these records. In many cases you want to query for entity objects instead, but in some cases this is not fast / efficient enough and you should resort to a DTO.

DTO’s can also be used for other data transfers than for SQL queries.

#### Repository

A repository is like a set of queries. Repositories return or save entities in the data store. Parameters of the methods must be simple types, not entities. The repository pattern is a way to put queries in a single place. The repository’s job is also to provide an *optimal* set of queries.

Typically, every entity type gets its own repository.

DO NOT expose types from the underlying persistence technology.

#### Repository Interfaces

Any repository type will get an associated repository interface. This keeps our system loosely coupled to the underlying persistence technology.

### Business Logic Patterns

#### Business layer

Presentation, entity model and persistence should be straightforward. If anything special needs to happen this belongs in the business layer. Any number of different patterns can be used.

The business layer externally speaks a language of entities or sometimes data transfer objects (DTO’s). Internally it can talk to repository interfaces for data access.

It is preferred that business logic works with entities rather than repositories (even though there is a large gray area). This improves testability, limits queries and limits interdependence, dependency on a data source and passing around a lot of repository variables.

#### Repository Wrappers

Passing around lots of repositories creates long parameter lists, that are prone to change. To combat that problem, combine sets of repositories into repository wrappers and pass those around instead. This keeps the parameter lists short and less prone to change.

You can make a single RepositoryWrapper with all your domain model’s repositories.

But that is not always enough. Some logic will use repositories out of multiple domains, so sometimes you are well off making a custom repository wrapper in that case. You could also choose to simply pass around multiple repository wrappers: one per domain model.

Also, you may want to create different, more limited repository wrappers. For instance ones for partial domain models. This keeps the width of dependency narrow, so logic that has nothing to do with certain repositories, do not become dependent on all of them.

An alternative to repository wrappers is dependency injection. See ‘dependency injection’. There you will find some criticism about the techique, but those might be due to not using a very good dependency injection API. Repository wrappers and dependency injection could well be used in combination with eachother.

#### Validators

Use separate validator classes for validation. Make specialized classes derived from JJ.Framework.Validation.FluentValidator<T>.

Try to keep validators independent from eachother.

If multiple validators should go off, call them individually one by one. Try not to make them delegate to eachother.

If you do decide to make a complex validator, add a prefix or suffix to the class name such as ‘Recursive’ or ‘Versatile’ to make extra clear that it is not just a simple validator.

Next to validators saying that user input is wrong, validators can be used to generate warnings, that are not blocking, but help the user do their work.

Validators can also be used for (complex) delete constraints, for instance when an entity is still in use, you might not be able to delete it.

#### Side Effects

The business layer executes side effects when altering data, for instance storing the date time modified or setting default values when you create an entity, or for instance automatically generating a name.

We implement the interface ISideEffect for each side effect. It only has one method: Execute, but it allows us to have some sort of polymorphism over side effects so it is easier to execute multiple of them in one blow, or allows other more generic handlings of the side effects.

Using a separate class for side effects, creates overview over those pieces of business logic, that are the most creative of all, and prevents those special things that need to happen from being entangled with other code.

Side effects should evaluate the conditions internally as much as possible. So the called of the side effect class does not know what conditions are tied to it doing anything at all. This makes the side effect fully responsible for what happens. The side effect’s doing anything can also be dependent on entity status. See ‘Entity Status Management’.

#### LinkTo

This pattern is about inverse property management. Inverse property management means for instance that if a parent property is set: Product.Supplier = mySupplier, then automatically the product is added to the child collection too: Supplier.Products.Add(myProduct).

To manage inverse properties even when the underlying persistent technology does not have inverse property management, you can link entities with LinkTo methods, instead of assigning properties or adding or removing from related collections directly. By calling the LinkTo methods, both ends of the relationship are kept in sync. Here is a template for a LinkTo method that works for 1-to-n relationships. Beware that all the checks can come with performance penalties.

public static void LinkTo(this Child child, Parent parent)

{

if (child == null) throw new NullException(() => child);

if (child.Parent != null)

{

if (child.Parent.Children.Contains(child))

{

child.Parent.Children.Remove(child);

}

}

child.Parent = parent;

if (child.Parent != null)

{

if (!child.Parent.Children.Contains(child))

{

child.Parent.Children.Add(child);

}

}

}

The class in which to put the LinkTo methods, should be called LinkToExtensions and it should be put in the LinkTo sub-namespace in your project.

Only if the LinkTo method name is ambiguous, you can suffix it, e.g.:

LinkToParentDocument

Next to LinkTo method, you should have Unlink methods in an UnlinkExtensions class:

public static void UnlinkParent(this Child child)

{

if (child == null) throw new NullException(() => child);

child.LinkTo((Parent)null);

}

If you are linking objects together that you know are new, you may create better-performing variations for LinkTo, called NewLinkTo, that omit all the expensive checks:

public static void NewLinkTo(this Child child, Parent parent)

{

if (child == null) throw new NullException(() => child);

child.Parent = parent;

parent.Children.Add(child);

}

Be aware that executing NewLinkTo onto *existing* objects will result in a corrupted object graph.

#### Cascading Extensions

<TODO: Describe how to organize your DeleteRelatedEntitiesExtensions and UnlinkRelatedEntitiesExtensions. >

#### Manager / Facade

A Manager class combines several related (usually CRUD) operations into one class that also performs additional business logic and validation, side effects, integrity constraints, conversions, etc. A Manager is a ‘CRUD-oriented facade’. It delegates to other classes to do the work. If you do it using the manager you should be able to count on it that the integrity is maintained.

It is a combinator class: a manager combines other (smaller) parts of the business layer into one offering a single entry point for a lot related operations. It is usually about a partial business domain, so manages a set of entity types together. You could also call it a combinator class.

##### Get by ID not in the Manager

Even though Manager classes typically contain CRUD methods and is usually the entry point for all your business logic and data access operations, there is an exception: do not put a Get by ID method in your Manager class. Execute a simple Get by ID onto the repository. The reason is that you would get an explosion of dependency and high coupledness, since a simple operation executed all over the place, would now require a reference to a manager, which is a combinator class, meaning it is dependent on many repositories and other objects. So a simple Get goes through the repository.

#### Visitor

A Visitor class processes a recursive structure that might involve many objects and multiple types of objects. Usually a visitor translates a complex structure into something else. Examples are calculating a total price over a recursive structure, or filtering down a whole object graph by complex criteria. Visitors also give great performance when programmed well.

Whenever a whole recursive structure needs to be processed, the visitor pattern is a good way to go.

A visitor class will have a set of Visit methods, e.g. VisitOrder, VisitProduct, typically one for every type, possibly also one for each collection. A base visitor might simply follow the whole recursive structure, and has a Visit method for each node in the structure. All Visit methods are protected virtual and usually return void. Public methods will only expose the entry points in the recursion. Derived visitors can override any Visit method that they need. If you only want to process objects of a specific type, you only override the Visit method for that specific type. You can optimize performance by overriding Visit methods that would enter a part of the recursive structure that you do not use.

Typically the result of a visitor is not put on the call stack, but stored in fields and used throughout the Visit methods. This is because the result usually does not have a 1-to-1 mapping with the source structure.

By creating a base visitor and multiple specialized visitors, you can create short and powerful code for processing recursive structures. A coding error is easily made, and can break calculations easily. However, it is the best and fastest choice for complicated calculations that involve complex recursive structures.

The classic visitor pattern has a design flaw in it, that we will not use. The classic visitor requires that classes used by the visitor have to be adapted to the visitor. This is adapting the wrong classes. We will not do that and we will not add Accept methods to classes used by a visitor.

A good example of a Visitor class is .NET’s own ExpressionVisitor, however we follow additional rules.

<TODO: Make a good text out of this, covering handling polymorphism in visitors. Merge this with the main text:

- Document that a Visitor that handles polymorphism, should have a Polymorphic visitation that delegates to a concrete visitation, that delegates to a base visitation, and you need all those methods delegating in the right order, for the visitation to happen in the correct order.

- Visitor pattern: mention that you always need to call polymorphic, otherwise you will not get all the objects when you override the polymorphic. >

<TODO: Code example.>

<TODO: Describe this: Patterns, Visitor: Figure out a good way to prevent calling those Polymorphic visit methods if not required.>

#### String Resources

For button texts, translations of model properties in different languages, etc., use resx files in your .NET projects.

If you follow the following naming convention for resources files, .NET will automatically return the translations into the language of the current culture:

Resources.resx

Resources.nl-NL.resx

Resources.de-DE.resx

The culture-inspecific resx has the en-US language.

The key should be representative of the text itself.

<TODO: Mention the resource formatter pattern, e.g. MessageFormatter.>

Resources seem part of the presentation, but they are extensively used in the business layer, so are put in the business assemblies. Especially the display names of model properties should be put in the back-end, so they can be reused in multiple applications.

Framework.Resources contains reusable resource strings for common titles such as ‘Delete’, ‘Edit’, ‘Save’ etcetera.

Extra information in Dutch about how to structure your resource files can be read in Appendix B.

### Presentation Patterns

#### ViewModel

A ViewModel class holds the data shown on screen.

It is purely a data object. It will only have public properties. It should have no methods, no constructor, no member initialization and no list instantiation. (This is to make sure the code creating or handling the viewmodels is fully responsible for it.)

**A ViewModel should say *what* is shown, not *how* or *why*.**

Every screen gets a view model, e.g. ProductDetailsViewModel, ProductListViewModel, ProductEditViewModel, CategorySelectorViewModel.

You can also reuse simple view models that represent a single entity, e.g. ProductViewModel, CategoryViewModel.

ViewModels may only use simple types and references to other view models. A ViewModel should never reference data-store bound entities directly.

Inheritance is *not* allowed, so it is a good plan to make the ViewModel classes sealed.

Do not convert view models to other view models (except for yielding over non-persisted properties). Always convert from business domain to view model and from view model to business domain, never from view model to view model.

A ViewModel should say *what* is shown on screen, not *how*:  
As such it is better to call a property CanDelete, than calling it DeleteButtonVisible. Whether it is a button or a hyperlink or Visible or Enabled property is up to the view.

A ViewModel should say *what* is shown on screen, not *why*:  
For instance: if the business logic tells us that an entity is a very special entity, and it should be displayed read-only, the view model should contain a property IsReadOnly, not a property named ThisIsAVerySpecialEntity. Why it should be displayed read-only should not be part of the view model.

<TODO: Describe the ViewModel pattern more strictly: entity view models, partial view models and screen view models and the words Details, Edit, List, NotFound, Delete, Deleted and Overview. And that those words are there to indicate that it is a screen view model, not an entity or partial view model. LoginViewModel may be an exception. >

##### Considerations

The reason there should be no inheritance is because that would create an unwanted n2 dependency between views and the base view model: *n* views could be dependent on 1 view model and *m* view models could be dependent on 1 base view model, making *n* \* *m* views dependent on the same base view model. This means that if the base view model changes *n* \* *m* views could break, insteaf of just *n*. *m* is even likely to become greater than *n*. If multiple layers of inheritance are used, it gets even worse. That can get out of hand quickly and create a badly maintainable application. By using no inheritance, a view model could only break *n* views (the number of views that use that view model).

#### Lookup Lists

In a stateless environment, lookup lists in views can be expensive. For instance a drop down list in each row of a grid in which you choose from 1000 items may easily bloat your HTML. You might repeat the same list of 1000 items for each grid row. There are multiple ways to combat this problem.

For small lookup lists you might include a copy of the list in each entity view model and repeat the same lookup list in HTML.

Reusing the same list instance in multiple entity view models may seem to save you some memory, but a message formatter may actually repeat the list when sending a view model over the line.

For lookup lists up until say 100 items you might want to have a single list in an edit view model. A central list may save some memory but, but when you still repeat the HTML multiple times, you did not gain much. You may use the HTML5 <datalist> tag to let a <select> / drop down list reuse the same data, but it is not supported by Safari, so it is of not much use. You might use a jQuery trick to populate a drop down just before you slide it open.

For big lookup list the only viable option seems to AJAX the list and show a popup that provides some search functionality, and not retrieve the full list in a single request. Once AJAX’ed you might cache the popup to be reused each time you need to select something from it.

#### ToViewModel

An extension method that convert an entity to a view model. You can make simple ToViewModel methods per entity, converting it to a simple view model that represents the entity. You can also have methods returning more complex view models, such as ToDetailsViewModel() or ToCategoryTreeEditViewModel().

You may pass repositories to the ToViewModel methods if required.

Sometimes you cannot appoint one entity type as the source of a view model. In that case you cannot logically make it an extension method, but you make it a helper method in the static ViewModelHelpers class.

The ToViewModel classes should be put in the sub-folder / sub-namespace ToViewModel in your csproj. For an app with many views a split it up into the following files may be a good plan:

ToIDAndNameExtensions.cs

ToItemViewModelExtensions.cs

ToListItemViewModelExtensions.cs

ToPartialViewModelExtentions.cs

ToScreenViewModelExtensions.cs

ViewModelHelper.cs

ViewModelHelper.EmptyViewModels.cs

ViewModelHelper.Items.cs

ViewModelHelper.ListItems.cs

ViewModelHelper.Lookups.cs

ViewModelHelper.Partials.cs

ViewModelHelper.Screens.cs

To be clear: the ViewModelHelper files are all ViewModelHelper partial classes. The other files have a class that has the same name as the file.

Inside the classes, the methods should be sorted by source entity or application section alphabetically and each section should be headed by a comment line, e.g.:

// Orders

public static OrderListViewModel ToListViewModel(this IList<Order> orders) ...

public static OrderEditPopupViewModel ToEditViewModel(this Order order) ...

public static OrderDeletePopupViewModel ToDeleteViewModel(this IList<Order> orders) ...

Some view models do not take one primary entity as input. So it does not make sense to turn it into an extension method, because you cannot decide which entity is the this argument. In that case we put it in a ViewModelHelper class with static classes without this arguments. ViewModelHelper is also part of the ToViewModel layer.

#### ToEntity

Extension methods that convert a view model to an entity.

You typically pass repositories to the method. A simple ToEntity method might look up an existing entity, if it exists, it will be updated, if it does not, it will be created.

A more complex ToEntity method might also update related entities. In that case related entities might be inserted, updated and deleted, depending on whether the entity still exists in the view model or in the data store.

A ToEntity method takes on much of the resposibility of a Save action.

<TODO: Describe the organization of the ToEntity extensions.>

#### Presenter

Each view gets its own presenter.

Each user action is a method.

A presenter represents what a user can do in a screen.

The methods of the presenter work by a ViewModel-in, ViewModel-out principle.

An action method returns a ViewModel that contains the data to display on screen. Action methods can also receive a view model parameter containing the data the user has edited. Other action method parameters are also things the user chose. An action method can return a different view model than the view the presenter is about. Those are actions that navigate to a different view. That way the presenters are a model for what the user can do with the application.

Sometimes you also pass infra and config parameters to an action method, but it is preferred that the main chunk of the infra and settings is passed to the Presenter’s constructor.

Internally a presenter can use business logic and repositories to access the domain model.

All view model creation should be delegated to the ToViewModel layer (rather than inlining it in the Presenter layer), because then when the ViewModel creation aspect should be adapted, there is but one place in the code to look. It does not make the presenter a needless hatch (‘doorgeefluik’), because the presenter is responsible for more than just view model creation, it is also resposible for retrieving data, calling business logic and converting view models to entities.

#### ToEntity-Business-ToViewModel Round-Trip

A presenter is a combinator class, in that it combines multiple smaller aspects of the presentation logic, by delegating to other classes. It also combines it with calls to the business layer.

A presenter action method should be organized into phases:

* Security
* ViewModel Validation
* ToEntity / GetEntities
* Business
* Commit
* ToViewModel
* Non-Persisted (yield over non-persisted data from old to new view model)
* Redirect

Not all of the phases must be present. ToEntity / Business / ToViewModel are the typical phases. Slight variations in order of the phases are possible. But separate these phases, so that they are not intermixed and entangled.

Comment the phases in the code in the presenter action method:

// ToEntity

Dinner dinner = userInput.ToEntity(\_dinnerRepository);

// Business

\_dinnerManager.Cancel(dinner);

// ToViewModel

DinnerDetailsViewModel viewModel = dinner.ToDetailsViewModel();

Even though the actual call to the business logic might be trivial, it is still necessary to convert from entity to view model and back. This is due to the stateless nature of the web. It requires restoring state from the view to the entity model in between requests. You might save the computer some work by doing partial loads instead of full loads or maybe even do JavaScript or other native code.

<TODO: Consider this: Patterns, Presentation: There is something wrong with the pattern ‘ToEntity-Business-ToViewModel-NonPersisted’ sometimes it is way more efficient to execute the essence of the user action onto the user input view model. Sometimes it is even the only way to execute the essense of the user action onto the user input view model. Examples are removing a row an uncommitted row or collapsing a node in a tree view.>

#### NullCoalesce (ViewModels)

When you user input back as a ViewModel from your presentation framework of choice, for instance MVC, you might encounter null-lists in it, for lists that do not have any items. To prevent other code from doing null-coalescing or instead tripping over the nulls, you can centralize the null-coalescing of pieces of view model and call it in the presenter.

<TODO: Better description. Also incorporate:

* Also add a code example.
* Consider making a separate pattern description for NullCoalesce methods in general and move it to the Other Patterns section to which you then refer from this section NullCoalesce (ViewModels).
* Null-coalesce. Applied to viewmodels that are passed to presenters. The choice is made here to only null-coalesce things that a view / client technology might leave out. Theoretically it might be better to null-coalesce everything in the view model, but this does take full traversal of the tree, which comes with a (small) performance penalty. Also: the null-coalesce procedures take some typing time for the programmer, and requires maintenance when the structure changes. That is why the choice is made to only null-coalesce a select set of things, that is adapted to our specific needs, rather than something that will always work. >

#### Views

A template for rendering the view.

It might be HTML.

In WebForms this would be an aspx.

In MVC it can be an aspx or cshtml.

Any code used in the view should be dumb. That is: most tasks should be done by the presenter, which produces the view model, which is simply shown on screen. The view should not contain business logic.

#### First Full Load – Then Partial Load – Then Native Code

You could also call it: first choice full load.

In web technology you could also call it:

Full postback - AJAX - JavaScript

When programming page navigation, the first choice for showing content is a full page load. Only if you have a very good reason, you might use AJAX to do a partial load. Only if you have a very good reason, you might start programming user interaction in JavaScript.

But it is always the first choice to do full postbacks.

The reason is maintainability: programming the application navigation in C# using presenters is more maintainable than a whole lot of JavaScript. Also: when you do not use AJAX, the Presenter keeps full control over the application navigation, and you do not have to let the web layer be aware of page navigation details.

Furthermore AJAX’ing comes with extra difficulties. For instance that MVC <input> tag ID’s vary depending on the context and must be preserved after an AJAX call, big code blocks of JavaScript for doing AJAX posts, managing when you do a full redirect or just an update of a div. Keeping overview over the multitude of formats with which you can get and post partial content. The added complexity of sometimes returning a row, sometimes returning a partial, sometimes returning a full view. Things like managing the redirection to a full view from a partial action. Info from a parent view model e.g. a lookup list that is passed to the generation of a child view model is not available when you generate a partial view. Request.RawUrl cannot be used as a return URL in links anymore. Related info in other panels is not updated when info from one panel changes. A lot of times the data on screen is so intricately related to eachother, updating one panel just does not cut it. The server just does not get a chance to change the view depending on the outcome of the business logic. Sometimes an ajax call’s result should be put in a different target element, depending on the type you get returned, which adds more complexity.

Some of the difficulties with AJAX have been solved by employing a specific way of working, as described under AJAX in the Aspects section.

#### Temporary ID’s

When you edit a list, and between actions you do not commit you may need to generate ID’s for the rows that are not committed, otherwise you cannot identify them individually to for instance delete a specific uncommitted row. For this you can add a TemporaryID to the view model, that are typically Guids.

The TemporaryID’s can be really temporary and can be regenerated every time you create a new view model.

The TemporaryID concept breaks down, as soon as you need to use it to refer to something from multiple places in the view model.

An alternative is to let a data store generate the ID’s by flushing pendings statements to the data store, which might give you data-store-generated ID’s. But this method fails when the data violates database constraints. Since the data does not have to be valid until we press save, this is usually not a viable option, not to speak of that switching to another persistence technology might not give you data-store-generated ID’s upon flushing at all.

Another alternative is a different ID generation scheme. You may use an SQL Sequence, or use GUID’s, which you assign from your code. Switching from int ID’s to GUID’s is a high impact change though, and does come with performance and storage penalties.

#### Stateless and Stateful

The presentation patterns may differ slightly if used in a stateful environment, but most of it stays in tact. For instance that Presenters have action methods that take a ViewModel and output a new ViewModel is still useful in that setting. In a stateless environment such as web, it is needed, because the input view model only contains the user input, not the data that is only displayed and also not the lookup lists for drop down list boxes, etc. So in a stateless environment a new ViewModel has to be created. You cannot just return the user input ViewModel. You would think that in a stateful environment, such as a Windows application, this would not be necessary anymore, because the read-only view data does not get lost between user actions. However, creating a new view model is still useful, because it creates a kind of transaction, so that when something fails in the action, the original view model remains untouched.

You will be making assumptions in your Presenter code when you program a stateful or stateful application. Some things in a stateful environment environment will not work in a stateless environment and you might make some objects long-lived in a stateful environment, such as Context, Repositories and Presenters. But even if you build code around those assumptions, then when switching to a stateless environment – if that will ever happen – the code is still so close to what’s needed for stateless, that it will not come with any insurmountable problems. I would not beforehand worry about ‘will this work in stateless’, because then you would write a lot of logic and waste a lot of energy programming something that will probably never be used. And programming something for no reason at all, handling edge cases that would never occur, is a really counter-intuitive, unproductive way of working.

#### Considerations

##### ToEntity / ToViewModel

<TODO: Explain the argument that ViewModel, ToEntity and ToViewModel does require programming a lot of conversion code, but gives you complete freedom over your program navigation, but the alternative, a framework prevents writing this conversion code for each application, but has the downside that you are stuck with what the framework offers and loose the complete freedom over your how your program navigation works.>

### Presentation Patterns (MVC)

<TODO: Mention ModelState.ClearErrors.>

<TODO: Mention: Using Request.UrlReferrer in Http Get actions crashes. Use Request.RawUrl. >

#### Controller

In an ASP.NET MVC application a controller has a lot of responsibilities, but in this architecture most of the responsibility is delegated to Presenters. The responsibilities that are left for the MVC controllers are the URL routing, the HTTP verbs, redirections, setting up infrastructural context and miscellaneous MVC quirks.

The controller may use multiple presenters and view models, since it is about multiple screens.

Entity names put in controller should be plural. So Customer**s**Controller not CustomerController.

#### Post-Redirect-Get

This is a quirk intrinsic to ASP.NET MVC. We must conform to the Post-Redirect-Get pattern to make sure the page navigation works as expected.

At the end of a post action, you must call RedirectToAction() to redirect to a Get action.

Before you do so, you must store the view model in the TempData dictionary. In the Get action that you redirect to, you have to check if the view model is in the TempData dictionary. If the view model exist in the TempData, you must use that view model, otherwise you must create a new view model.

Here is simplified pseudo-code in which the pattern is applied.

**public ActionResult Edit(int id)**

**{**

**object viewModel;**

**if (!TempData.TryGetValue(TempDataKeys.ViewModel, out viewModel))**

**{**

**// TODO: Call presenter**

**}**

**return View(viewModel);**

**}**

**[HttpPost]**

**public ActionResult Edit(EditViewModel viewModel)**

**{**

**// TODO: Call presenter**

**TempData[TempDataKeys.ViewModel] = viewModel2;**

**return RedirectToAction(ActionNames.Details);**

**}**

There might be an exception to the rule to always RedirectToAction at the end of a Post. When you would redirect to a page that you can never go to directly, you might return View() instead, because there is no Get method. This may be the case for a NotFoundViewModel or a DeleteConfirmedViewModel.

<TODO:

* Mention that return View in case of validation messages is the way to go, because otherwise MVC will not remember un-mappable wrong input values, like Guids and dates entered as strings. (In one case this lead to the browser asking for resending postdata upon clicking the back button, so check whether this is actually a good idea.)
* Not using return View() in a post action makes old values not be remembered.>

##### Considerations

If you do not conform to the Post-Redirect-Get pattern in MVC, you may get to see ugly URL’s. When you hit the back button, you might go to an unexpected page, or get an error. You may see original values that you changed re-appear in the user interface. You may also see that MVC keeps complaining about validation errors, that you already resolved. So conform to the Post-Redirect-Get pattern to stay out of trouble.

#### ValidationMessages in ModelState

For the architecture to integrate well with MVC, you have to make MVC aware that there are validation messages, after you have gotten a ViewModel from a Presenter. If you do not do this, you will get strange application navigation in case of validation errors.

You do this in an MVC HTTP GET action method.

The way we do it here is as follows:

if (viewModel.ValidationMessages.Any())

{

ModelState.AddModelError(ControllerHelper.DEFAULT\_ERROR\_KEY, ControllerHelper.GENERIC\_ERROR\_MESSAGE);

}

In theory we could communicate all validation messages to MVC instead of just communicating a single generic error message. In theory MVC could be used to color the right input fields red automatically, but in practice this breaks easily without an obvious explanation. So instead we manage it ourselves. If we want a validation summary, we simply render all the validation messages from the view model ourselves and not use the Html.ValidationSummary() method at all. If we want to change the appearance of input fields if they have validation errors, then the view model should give the information that the appearance of the field should be different. Our view’s content is totally managed by the view model.

#### Polymorphic RedirectToAction / View()

A Presenter action method may return different types of view models.

This means that in the MVC Controller action methods, the Presenter returns object and you should do polymorphic type checks to determine which view to go to.

Here is simplified code for how you can do this in a post method:

**var editViewModel = viewModel as EditViewModel;**

**if (editViewModel != null)**

**{**

**return RedirectToAction(ActionNames.Edit,**

**new { id = editViewModel.Question.ID });**

**}**

**var detailsViewModel = viewModel as DetailsViewModel;**

**if (detailsViewModel != null)**

**{**

**return RedirectToAction(ActionNames.Details,**

**new { id = viewModel.Question.ID });**

**}**

At the end throw the following exception (out of the Framework):

throw new UnexpectedTypeException(() => viewModel);

To prevent repeating this code for each controller action, you could program a generalized method that returns the right ActionResult depending on the ViewModel type. Do consider the performance penalty that it may impose and it is worth saying that such a method is not very easy code.

#### Html.BeginCollection

In MVC it is not straightforeward to post a collection of items or nested structures.

This architecture’s framework has HtmlHelper extensions to make that easier: the Html.BeginCollection API. Using this API you can send a view model with arbitrary nestings and collections over the line and restore it to a view model at the server side. In the view code you must wrap each nesting in a using block as follows:

**@using (Html.BeginItem(() => Model.MyItem))**

**{**

**using (Html.BeginCollection(() => Model.MyItem.MyCollection))**

**{**

**foreach (var x in Model.MyItem.MyCollection)**

**{**

**using (Html.BeginCollectionItem())**

**{**

**}**

**}**

**}**

**}**

So each time you enter a level, you need another call to the Html helper again and wrap the code in a using block. You can use as many collections as you like, and use as much nesting as you like. You can spread the nesting around multiple partials.

Input fields in a nested structure must look as follows:

Html.TextBoxFor(x => x.MyProperty)

Or:

Html.TextBoxFor(x => Model.MyProperty)

Not like this:

Html.TextBoxFor(x => myLoopItem.MyItem.MyProperty)

Otherwise the input fields will not bind to the view model. This often forces you to program partial views for separate items. This is good practice anyway, so not that big a trade-off.

An alternative to Html.BeginCollection() is using for-loops.

**@Html.TextBoxFor(x => x.MyItem.MyProperty)**

**@for (int i = 0; i < Model.MyItem.MyCollection.Count; i++)**

**{**

**@Html.TextBoxFor(x => x.MyItem.MyCollection[i].MyProperty)**

**}**

This solution only works if the expressions you pass to the Html helpers contain the full path to a view model property (or hack the HtmlHelper.ViewData.TemplateInfo.HtmlFieldPrefix) and therefore it does not work if you want to split up your view code into partials.

Another alternative to the BeginCollection() is the often-used BeginCollectionItem(string) API. Example:

**@foreach (var child in Model.Children)**

**{**

**using (Html.BeginCollectionItem("Children"))**

**{**

**@\*** ... **\*@**

**}**

**}**

The limitation of that API is that you can only send one collection over the line and no additional nesting is possible.

Beware that currently the different solutions do not mix well and you should only use one solution for each screen of you program.

#### Return URL’s

* Return URL’s indicate what page to go back to when you are done in another page.
* It is used when you are redirected to a login screen, so it knows what page to go back to after you login.
* Return URL’s are encoded into a URL parameter, called ‘ret’ e.g.:  
  http://www.mysite.com/Login?**ret=%2FMenu%2FIndex**

The ret parameter is the following value encoded: /Menu/Index  
That is the URL you will go back to after you log in.

* The Login action can redirect to the ret URL like this:

[HttpPost]

public ActionResult Login(... string ret = null)

{

...

return Redirect(ret);

...

}

ASSIGN DIFFERENT RET FOR FULL PAGE LOAD OR AJAX CALL.

* For full page loads, the ret parameter must be set to:

Request.RawUrl

* For AJAX calls the ret parameter must be set to:

Url.Action(ActionNames.Index)

* The ret parameter is set in a controller action method, when you return the ActionResult. Example:

EXAMPLE WORKS FOR FULL PAGE LOAD ONLY!!!

return RedirectToAction(ActionNames.Login, ControllerNames.Account, new { ret = Request.RawUrl });

* A return URL should always be optional, otherwise you could never serparately debug a view.
* That way you have an easily codeable, well maintainable solution.
* Do not use RefferrerUrl, because that only works for HttpPost, not HttpGet. Use Request.RawUrl instead.
* There is a built-in error proneness in return URLs’. If you pass the same return URL along multiple HTTP requests, only one action has to forget to pass along the return URL and a back or close button is broken and you will find out very late that it is, because it is not an obvious thing to test. The same error-proneness is there for return actions with return actions with return actions, or with bread-crumb like structures with multiple return actions built in.

<TODO: Incorporate this: Ret parameters can be done with new { ret = Request.RawUrl } for full load, and for AJAX this works: { ret = Url.Action(ActionNames.Index) } if you always make sure you have an Index action in your controller, which is advisable.>

#### Back Buttons

There is a pitfall in builing back buttons. If you mix back buttons being handled at the server side, compared to window.history.back() at the client-side, you run the risk that the back button at one point keeps flipping back and foreward between pages.

### Data Transformation Patterns

#### Converter

A class that converts one data structure to another. Typically more is involved than just converting a single object. A whole object graph might be converted to another, or a flat list or raw data to be parsed might be converted to an object structure or the other way around.

By implementing it as a converter, it simplifies the code. You can then say that the only responsibility of the class is to simply transform one data structure to another: nothing more, nothing less and leave other responsibilities to other classes.

#### TryGet-Insert-Update

When converting one type to another one might use the TryGet-Insert-Update pattern. Especially when converting an entity with related entities from one structure to another this pattern will make the code easier to read.

TryGet first gets a possible existing destination entity.

Insert will create the entity if it did not exist yet, possibly setting some defaults.

Update will update the rest of the properties of either the existing or newly created object.

When you do these actions one by one for one destination entity after another, you will get readable code for complex conversions between data structures.

Note that deletion of destination objects is not managed by the TryGet-Insert-Update pattern.

#### TryGet-Insert-Update-Delete / Full-CRUD Conversion / Collection Conversion

Used for managing complex conversions between data structures, that require insert, update and delete operations. There is no one way of implementing it, but generally it will involve the following steps:

- Loop through the source collection.

- TryGet: look up an item in the destination collection.

- Insert: create a new item in the destination collection if none exists.

- Update: update the newly created or existing destination item.

- Do delete operations after that:

- Generally you can use an Except operation on the collections of existing items and items to keep, to get the collection of items to delete.

- Then you loop through that collection and delete each item.

##### Considerations

Converting one collection to another may involve more than creating a destination object for each source object. What complicates things, is that there may already be a destination collection. That means that insert, update and delete operations are required. There are different ways to handle this depending on the situation. But a general pattern that avoids a lot of complexity, is to do the inserts and updates in one loop, and do the deletes in a second loop. The inserts and updates are done first by looping through the source collection and applying the TryGet-Insert-Update pattern on each item, while the delete operations are done separately after that by comparing collections of entities to figure out which items are obsolete.

In a little more detail:

- Loop through the source collection.

- TryGet: look up an item in the destination collection.

- Insert: create a new item in the destination collection if none exists.

- Update: update the newly created or existing destination item.

- Do delete operations after that:

- Generally you can use an Except operation on the collections of existing items and items to keep, to get the collection of items to delete.

- Then you loop through that collection and delete each item.

Here follows some pseudo code for how to do it:

void ConvertCollection(IList sourceCollection, IList destCollection)

{

foreach (var sourceItem in sourceCollection)

{

var destItem = TryGet(...);

if (destItem == null)

{

destItem = Insert();

}

destItem.Name = sourceItem.Name; // Update

}

var itemsToDelete = destCollection.Except(sourceCollection);

foreach (var itemToDelete in itemsToDelete)

{

Delete(itemToDelete);

}

}

The specific way to implement it, is different in every situation. Reasons that there are many ways to do it are:

- You cannot always count on instance integrity.

- You cannot always count on identity integrity.

- The key to a destination item might be complex, instead of just an ID.

- You do not always have a repository.

- It does not always need to be full-CRUD.

- You might need to report exactly what operation is executed on each entity.

- You might need a separate normalized *singular* form of the conversion, that may conflict with the way of working in the plural form.

- An alternative isNew detection might be needed.

- Some persistence technologies will behave unexpectedly when first retrieving and then writing and then retrieving again. Intermediate redundant retrievals should be avoided. Or not, depending on the situation.

Each variation has either overhead or elegance depending on the situation. If you always pick the same way of doing it, you may end up with unneccesary and unsensical overhead, or with an overly complicated expression of what you are trying to do.

The general forms above is a good starting point. Then it needs to work correctly. The next quality demand is a tie between readability and performance.

##### Alternative: Flagging

An alternative to TryGet-Insert-Update-Delete pattern, which kind of does a full diff of a source and destination structure, is maintaining a kind of flagging in the source structure: Added, Modified, Deleted and Unmodified.

A downside is that when two people try to save a piece of data at the same time, you may end up with a corrupted structure. It depends on the situation whether this would happen at all, since not all data is edited by every user.

Another downside to flagging is that the source structure must be adapted to it, which is not always an option / a good option.

The TryGet-Insert-Update-Delete pattern, though, creates a last-user-wins situation, because not flagging determines whether it is an update or insert, but actual existence of dest object determines it.

#### DocumentModel

An analog of a view model, but then for document generation, rather than view rendering. It is a class that contains all data that should be displayed in the document. It can end with the suffix ‘Model’ instead of ‘DocumentModel’ for brevity, but then it must be clear from the context that we are talking about a document model.

Just as with view models, inheritance structures are not allowed. To prevent inheritance structures it may be wise to make the DocumentClasses classes sealed.

#### Selector-Model-Generator-Result

For data transformations you may want to split up the transformation in two parts:

* A Selector which returns the data as an object graph, or Model.
* A Generator (or Converter) that converts the object graph (or Model) into a specific format.

This is especially useful if there are either multiple input formats or multiple output formats or both, or if in the future either the input format or output format could change.

This basic pattern is present in many architectures and can be applied to many different parts of architecture.

Her follow some examples.

##### Generating a Document

An example of where it is useful, is generating a document in multiple format e.g. XLSX, CSV and PDF. In that case the data selection and basic tranformations are programmed once (a Selector that produces a Model) and exporting three different file formats would require programming three different generators. Reusable generators for specific file formats such as CSV may be programmed. Those will make programming a specialized generators very easy. So then basically exporting a document is mostly reading out a data source and producing an object graph.

##### Data Source Independence

The Selector-Model-Generator-Result pattern is also useful when the same document can have different data sources. Let’s say you want to print an invoice out of the system, but print another invoice out of an ordering system in the same formatting e.g. a PDF. This requires 2 selectors, 1 model and 1 generator, instead of 2 generators with complex code and potentially different-looking PDF’s.

##### Multiple Import Formats

You might want to import similar data out of multiple different data sources or multiple file formats. By splitting the work up into a Selector and a generator you can share must of the code between the two imports, and reduce the complexity of the code.

##### Limiting Complexity

Even if you do not expect multiple input formats or multiple output formats or a change in input or output format, the split up in a Selector and a Generator can be used to make the code less complicated to write, and subsequently also prevent errors and save time programming and maintaining the code.

##### MVC

MVC itself contains a specialized version of this very pattern. The following layering stacks are completely analogus to eachother:

* Selector - Model - Generator – Result
* Controller - ViewModel - view engine – View

### Other Patterns

#### Accessor

An accessor class allows access to non-public members of a class. This can be used for testing or for special access to a class from special places. JJ.Framework.Reflection has an implementation of a reusable Accessor class.

#### Adapter

<TODO: Describe what an adapter is in general and what kind of variations you can think of.>

#### Anti-encapsulation

Encapsulation makes sure a class protects its own data integrity. Anti-encapsulation is the design choice to let a class check none of its data integrity. Then you know that something else is 100% responsible for the integrity of it, and the class itself will guard none of it.

The reason not to use encapsulation is that it can go against the grain of many frameworks, such as ORM’s and data serialization mechanisms.

Anti-encapsulation can also be a solution to prevent spreading of the same responsibility over multiple places. If the class cannot check all the rules itself, it may be better the check all the rules elsewhere, instead of checking half the rules in the class and the other half in another place.

#### Initialization and Finalization

Cleanup code should be symmetric to the set-up code. Build something up in the constuctor then dispose things in the finalizer, start a service at startup then stop a service at shutdown, etc. If in the constructor you bind an event, then in the destructor you unbind it.

You can also choose to implement IDisposable. This is useful if you want to be able to explicitly trigger finalization. Finalizers/destructors only go off when the garbage collector feels like it, and you might want to imperatively tell an object to clean up its stuff.

* If you implement IDisposable, call Dispose from the finalizer/destructor.:

~MyClass

{

Dispose();

}

* Make sure the dispose can successfully run regardless of state, so check any variable you might use for null first and be tollerant towards null.

public void Dispose()

{

\_myConnection**?**.Close();

}

* Also call GC.SuppressFinalize() in the Dispose() method, because then the garbage collector will skip a few unneeded steps in getting rid of the object.

#### Constructor Inheritance

Sort of forces a derived class to have a constructor with specific arguments. Constructors are not inherited, but inheriting from a base class that has specific constructors forces your derived class to call that base constructor, often leading to exposing a similar constructor in the derived class.

#### DebuggerDisplays

<TODO: Describe how I handle DebuggerDisplays. Snippet of text: DebuggerDisplays with private property DebuggerDisplay.>

#### Executor

Executor classes are classes that encapsulate a whole process to run. For processes that involve more than just a single function, for instance downloading a file, transforming it and then importing it, involving infrastructure end-points and possibly multiple back-end libraries.

By giving each of those processes its own executor class, you make the code overviewable, and also make the process more easily runnable from different contexts, e.g. in a scheduler, behind a service method or by means of a button in a UI or in a utility.

#### Inheritance-Helper

A weakness of inheritance in .NET is that there is no multiple inheritance: you can only derive from one base class. This often leads to problems programming a base class, because one base will offer you one set of functionalities and the other base the other functionalities. (See the ‘Cartesian Product of Features Problem’.) To still use inheritance to have behaviors turned on or off, but not have an awkward inheritance structure, and problems picking what feature to put at which layer of inheritance, you could simply program helper classes (static classes with static methods) that implement each feature, and then use inheritance, letting derived classes delegate to the helpers, to give each class a specific set of features and specific versions of the features, to polymorphically have the features either turned on or off. You will still have many derived classes, but no arbitrary spreading of features over the base classes, and no code repetition either.

This allows you to solve what inheritance promises to solve, but does not do a good job at on its own. It basically solves the Cartesian Product of Features problem, the problem that there is no multiple inheritance and the problem with god base classes, all weakneses of inheritance.

#### Factory

A factory class is a class that constructs instances. But it usually means that it creates a concrete type, returning it as an abstract type. The concrete type that is instantiated depends on the input you pass to the factory’s method:

public static class ThingFactory

{

public static IThing CreateThing(int parameter)

{

switch (parameter)

{

case 0:

return new NormalThing();

case 1:

return new SpecialThing();

default:

throw new Exception(String.Format("parameter value '{0}' is not supported.", parameter));

}

}

}

A factory class is used if you want to instantiate an implementation of a base class or interface, but it depends on conditions which implementation it has to be or if you wish to abstract away knowledge of the specific concrete types it produces.

A class that returns instances with various states is also simply called a Factory, even though no polymorphism is involved.

(The classic implementation is not used here, which is a static method in a base class.)

#### Factory-Base-Interface

The Factory-Base-Interface pattern is a common way the factory pattern is applied. Next to a factory, as described above in the ‘Factory’ pattern, you give each concrete implementation that the factory can return a mutual interface, which also becomes the return type of the factory method. To also give each concrete implementation a mutual base class, with common functionality in it, and also to sort of force an implementation to have a specific constructor (see ‘Constructor Inheritance’).

#### TryGet

A combination of a TryGet method and a Get method (e.g. TryGetObject and GetObject) means that TryGet will return null if the object does not exist and Get will throw an exception if the object does not exist.

Call Get if it makes sense that the object should exist.

Call TryGet if the non-existence of the object makes sense.

If you call a TryGet you should handle the null value that could be returned.

TryGet can throw other exceptions, even though it does not throw an exception if the object does not exist.

#### Get-TryGet-GetMany

Often you need a combination of the three methods that either get a list, a single item but allow null or get a single item and insist it is not null. You can implement the plural variation and base the Get and TryGet on it using the same kind of code every time:

public Item GetItem(string searchText)

{

Item item = TryGetItem(searchText);

if (item == null)

{

throw new Exception(String.Format("Item with searchText '{0}' not found.", searchText));

}

return item;

}

public Item TryGetItem(string searchText)

{

IList<Item> items = GetItems(searchText);

switch (items.Count)

{

case 0:

return null;

case 1:

return items[0];

default:

throw new Exception(String.Format(

"Multiple items found for searchText '{0}'.", searchText));

}

}

public IList<Item> GetItems(string searchText)

{

return \_items.Where(x => !String.IsNullOrEmpty(x.Name) &&

x.Name.Contains(searchText))

.ToArray();

}

The GetItem and TryGetItem methods are the same in any situation, except for names and exception messages. Only the plural method is different depending on the situation.

#### Helper

Helper classes are static classes with static methods that help with a particular aspect of programming. They can make other code shorter or prevent repeating of code, for functions that do not require any more structure than a flat list of methods.

#### Info

Info objects are like DTO’s in that they are usually used for yielding over information from one place to another. Info objects can be used in limited scopes, internal or private classes and serve as a temporary place of storing info. But info objects can also have a broader scope, such as in frameworks, and unlike DTO’s they can have constructor parameters, auto-instantiation, encapsulation and other implementation code.

#### IsSupported

A service environment may contain the same interface for accessing multiple systems. But not every system is able to support the same features. You could solve it by creating a lot of different interfaces, but that would make the service layer more difficult to use, because you would not know which interface to use. Instead, you could also add ‘IsSupported’ properties to the interface to make an implementation communicate back if it supports a feature at all, for instance:

OrderStatusEnum IOrderManager.GetOrderStatus();

bool IOrderManager.GetOrderStatusIsSupported { get; }

Then when running price updates for multiple systems, you can simply skip the ones that do not support it. Possible a different mechanism is used for keeping prices up-to-date, possibly there is another reason why price updates are irrelevant. It does not matter. The IsSupported booleans keeps complexity at bay, more than introducing a large number of interfaces that would all need to be handled separately.

#### Mock

A mock object is used in testing as a replacement for a object used in production. This could be an entity model, an alternative repository implementation (that returns mock entities instead of data out of a database). A mock object could even be a database record. Unlike other patterns the convention is to put the word ‘Mock’ at the beginning of the class rather than at the end.

#### Name Constants

To prevent typing in a lot of strings in code, make a static class with constants in it, that become placeholders for the name.

e.g. ViewNames, with constants in it like this:

public static class ViewNames

{

public const string Edit = "Edit";

}

the name of the constant should be exactly the same as the string text.

Everywhere you need to use the name, refer to the constant instead of putting a literal string there.

This prevents typing errors and makes ‘find all references’ possible.

<TODO: Consider not assinging the string value at all, but using nameof(ViewNames.Edit). Consider using nameof() over an existing member to begin with.>

#### NullCoalesce

<TODO: See NullCoalesce (ViewModels) and write some good text here.>

#### Plug-In Model

<TODO: Describe my implementation of a nice plug-in model including the ReflectionHelper.GetImplementation methods.>

#### Progress and Cancel Callbacks

To make a process cancellable and report process without being dependent on the presentation framework, you can simply pass a few callback delegates to a method or class.

public Excute(Action<string> progressCallback, Func<bool> isCanceledCallback

{

progressCallback("Starting.");

if (isCanceledCallback())

{

progressCallback("Cancelled.");

return;

}

// ...

progressCallback("Finished.");

}

It depends on your problem whether those callbacks are nullable and you should do the appropriate null-checks depending on the situation.

<TODO: Add explanations and code examples about the client code.>

Sometimes it is useful to separate Cancel into two: Canceling and Canceled. This is because a process might not cancel immediately. A UI should not immediately enable a Start button again after the user pressed Cancel. A isCancelingCallback then allows the client to signal to the process that cancellation is requested. And an isCanceledCallback will let the process signal the client that cancelation has complete, so it can enable the start button again.

#### Singular, Plural, Non-Recursive, Recursive and WithRelatedEntities

When processing object structures, it is best to split everything up into separate methods.

Every entity type will get a method (the ‘Singular’ variation) that processes a single object. That method will not process any underlying related items, only the one object.

In case of conversions from one object structure to another, every *destination* entity gets a Singular method, not the *source* entity, because that would easily create messy, unmanageable code.

A ‘Plural’ method processes a whole list of items. Plural methods are less useful. Prefer singular methods over plural ones. Plural methods usually do not add anything other than a loop, which is too trivial to create a separate method for. Only when operations must be executed onto a whole list of objects (for instance determining a total price of a list of items or when there are specific conditions), it may be useful to create a separate Plural method.

Singular or Plural methods do not process related entities unless they have the method suffix ‘WithRelatedEntities’ or ‘Recursive’ at the end of the method name. Keep the Recursive and RelatedEntities methods separate from the not-with-RelatedEntities methods. ‘Related entities’ means entities intrinsically part of the entity, not links to reused entities. Also, not the parent.

There is a subtle difference between ‘WithRelatedEntities’ and ‘Recursive’. They are similar, but Recursive processing can pass the same object type again and again, while processing with related entities processes a tree of objects, in which the same object type does not recur at a deeper level.

Finer details about the Singular form:

* They do not process child entities, they can however link to reusable entities, such as enum-like types, or categories.
* They usualy do not assign a parent. The caller of the Singular form does that. That way methods are more independent of context and better reusable and code better rewritable. There are exceptions to that rule.

Here is an example of some Singular, Plural, Non-Recursive and Recursive methods. Note that the words ‘Singular’ and ‘Plural’ are not used in the method names.

private class MyProcess

{

private StringBuilder \_sb = new StringBuilder();

public string ProcessRecipeRecursive(Recipe recipe)

{

if (recipe == null) throw new NullException(() => recipe);

ProcessRecipe(recipe);

ProcessIngredients(recipe.Ingredients);

ProcessRecipesRecursive(recipe.SubRecipes);

return \_sb.ToString();

}

private void ProcessIngredients(IList<Ingredient> ingredients)

{

\_sb.AppendLine("Ingredients:");

foreach (Ingredient ingredient in ingredients)

{

ProcessIngredient(ingredient);

}

}

private void ProcessIngredient(Ingredient ingredient)

{

\_sb.AppendLine(String.Format("{0} {1} ({2})", ingredient.QuantityDescription, ingredient.Name, ingredient.ID));

}

private void ProcessRecipesRecursive(IList<Recipe> recipes)

{

\_sb.AppendLine("Sub-Recipes:");

foreach (Recipe recipe in recipes)

{

ProcessRecipeRecursive(recipe);

}

}

private void ProcessRecipe(Recipe recipe)

{

\_sb.AppendLine(String.Format("Recipe: {0} ({1})", recipe.Name, recipe.ID));

}

}

#### Wrapper

A wrapper class is a class that wraps one or more other objects. This can be useful in various situations. You might give the wrapper additional helper methods that the wrapped object does not have. You might dispose the underlying wrapper object and create a new one, keeping the references to the wrapper object in tact even though the wrapped object does not exist anymore. You may hide a specific object in a wrapper and give it an alternative interface, you might wrap multiple objects in one wrapper to pass them around as a single object for convenience.

### Alternatives

#### Rich Models

<TODO: Write story with pros and cons. Include:

>> Arch: another downside of rich models is the magic of it. You are not sure what happens and all sorts of non-obvious side-effects may go off.

* Arch: anemic models and separation of concerns and no rich models can have the consequence that you loose identity and instance integrity, because derived structures are more common. For controls for instance, with gesture events, the original object needs to be found and raised an event on, and you cannot get away with doing it on a derived object unless you clone everything. Not sure how to descrive this clearly. It is about rich models vs. anemic models and when and how to apply which and what are the pros and cons and I do not have a clear image of that yet.
* Explain the problems with rich models in the business layer comprised of derived classes out of the persistence layer, that extend the model with specific relations, constraints and rules, that you do not enforce in the persistence layer.  
  Problems with putting it in the business layer include that the persistence layer does not know how to instantiate the derived class, so it must instantiate the base class. Also: if rules are strictly enforced in an extended entity model, it is hard to separate creating entities from validating it, so instead of user-generated validation messages, you might get exception messages instead.  
  Problems with putting the specialized classes in the persistence layer include that the entity model must stay as clean as possible: anything you put in the data layer is hard to get rid of.  
  When you do need an business logic interfacing that is comprised of an ‘extended’ entity model, then you cannot really use inheritance. You might be able to create a manager class that creates a wrapper class around the ‘base’ class out of the entity model.  
  Currently the choice is to not make an extended entity model in the business layer.
* Arch: against rich models: You want your entity model to be a direct depiction of what is actually stored, so that you have control over that. If it is obscured, this means less control over what is going on. >

<TODO: Compare rich models with the 2D separation of concerns.>

## Aspects

This section lists some choices concerning system aspects such as security and logging, etc, even though many system aspects are already described under ‘Design Patterns’. Certain aspects already mentioned in the Design Patterns section may be repeated here, so it creates a clearer picture of what is regarded an aspect and what not.

<TODO: Describe the difference between a pattern and an aspect. Something like: aspects are things you are eventually going to have to make decision about, patterns are concrete options to choose from. Or: Patterns are specific solutions to aspects, which are more general. >

<TODO: Do something with this text: Aspects are central to the technical choices. The specific choices about an aspect can be an API that covers it, a pattern that covers it, or both, or multiple API’s and patterns. Aspects represent technical issues that you are going to have to make choice about, API’s and patterns offer a solution for them. And still we are not even talking at all about what functionalities we are going to program. That is totally outside the scope of this reference architecture.>

### Authoring & Reviewing

This aspect covers things such as marking objects with creation dates, modification dates, etcetera, adding an author’s comment to objects and managing multiple versions of objects and logging which user made which change.

<TODO: Describe better: Reviewing: Like rating content. So closely related to authoring I find that it should be grouped together. >

<TODO: Add specific solutions. >

### Caching

<TODO: Write texts. Include: The use of a dictionaries, TryGetValue for performance, locking static dictionaries or other dictionaries used by multiple threads. >

### Calculation

<TODO: Write a text. Include:

Extension methods for small calculations are an option. Putting calculations directly in your entity models (rich models.) I would not recommend it, because it would mix entity modeling with calculation aspect too much. For more complicated calculations you could delegate to specialized business logic objects. You could call those ‘Calculators’. Those could have a recursive structure of objects, if needed. You can also create code generators that compile to more optimal calculation code, if the structure of the calculation does not very too much over time it may be worth it to periodically recompile. Code generators are not easy, though. You could use a visitor to simplify, but isolate complex recursive calculations.>

### Cascading

<TODO: Write texts. Include:

Talk about DeleteWithRelatedEntities and UnlinkRelatedEntities.

Do mention NHibernate cascading.

See ‘LinkTo’ under ‘Patterns’.

Alternatives: In database. might not play well with ORM. Does not give the idea of control to the C# programmer. Alternative: triggers. Downside: Why would you do that if databases already have delete actions?>

### Circular References

<TODO: An example of a check procedure. Mention that it must be checked when writing data, not when reading data, so only writing has a performance penalty, but reading does not. >

### Cloning

<TODO: Describe.>

### Collections / List Processing

<TODO: Write text. Include:

* When to use which, interface types, array / list, KeyValuePair / Tuple / custom tuple class.
* Functions in JJ.Framework.Common.The extension methods and the KeyValuePairHelper.
* ToArray() trick when you adapt the list within the loop.>

#### Specialized Lists

<TODO: Talk about constrained lists and how to program a specialized list if a normal list does not suffice.>

### Concurrency

In a web-based application time elapses between retrieving data to edit and saving changes. Between these actions the data may have been changed by another user.

In this architecture the concurrency strategy is: the last user wins. This is accomplished in code using TryGet-Insert-Update-Delete pattern, that results in readable saving code and restoration of state, regardless of what another user did to it.

#### Alternatives

This paragraph is not that important to read. It is a longer story about the benefits and downsides of several ways to handle concurrency.

One possiblity is locking access to the data. This might not be such a good choice. One reason is that the data is not accessible to other users until the original user saves his changes. A user might leave open an editing window. In fact we cannot be sure that user has even closed the window at all, so we do not know when to unlock the data. Locking can cause a paralizing effect, especially in high-concurrency situations.

Another technique for handling concurrency is to assume that other users probably will not change the data, but if he had, the save procedure fails. Just before saving, the system checks if the data in the data store was changed since the data was retrieved. If the data is different, this will result in an error message. This requires quite some checks by the system, and remembering original values. Furthermore, in high concurrency situations the number of error messages may increases. And an error message basically means that the user needs to discard his changes and start again, which does not sound ideal.

The preferred technique in this architecture works OK in high-concurrency situations, with data shared data by users. The strategy is: the last one to save wins. The data is updated to the state the user wants to save. During the save procedure data will be locked (but not in between the user action of retrieving the data and the later action of saving the data). During the save transaction the data will be update to the state the used wants. In situations where data is hardly shared, this will accomplish the desired effect and just save the user’s changes. In situations where data is more frequently changed by different users, it may result in successive saving of each user’s changes, in case of which the last user wins. In case of even more concurrency, the last one to save will win discarding the losing user’s save action, which may cause confusion with the user that lost. In case of even more concurrency, when many transactions accessing the same data run at the same time, the concurrency problems associated with data stores can occur, in the worst case dead-locks. If save procedures are sufficiently short and fast, this might scarcely occur.

The way this strategy of ‘last user wins’ is accomplished, is by running the save operation in a transaction, adopting a TryGet-Insert-Update pattern for entities. There wll be no checks regarding whether an object is new or already existed. Not-found objects are simple recreated, so that ghost objects (objects read out by one user, deleted by another user) are restored.

Another strategy that will not be used, is trying to keep all users’ additions of related entities. However, this may create more confusion, ambiguity and code complexity. Here is the ambiguity: When a related entity is not in user A’s list and it is in user B’s list, does this mean it is user B’s addition of the related entity, or user A’s removal of the related entity? The ambiguity could be solved by marking which entities are actually new, which are unmodified and which are removed. This will add complexity to the code, but does take away the ambiguity. Now here is the confusion: User A that just saved a data set, gets to see a different data set after saving. User B also gets to see a different data set than what he saved. Neither user seems to have control over the data set. In other strategies at least user B gets to see the data exactly how he saved it, while user A gets to see the data how user B overwrote it. The only thing that you might gain from this strategy, is that users can work on the same data set at the same time, and the result is the accumulation of all these user’s changes. However, it is much harder to manage and the benefits are little.

### Configuration

For configuration we will use our own API: Framework.Configuration. It makes it easier to work with complex configuration files, while using .NET’s System.Configuration directly can be quite a lot of work.

We will use 3 ways of storing configuration settings:

* Custom configuration sections.
* Reading out the appSettings section.
* Reading out the connectionStrings section.

There is another configuration method in .NET: the Settings designer in the project properties. We will not use that, because it is very error-prone. The synchronization between the entered data and the XML does not work very well, and this creates the risk that you might put production settings in a test environment or test settings in a production environment.

#### Custom Configuration Sections

If your configuration requires more than a flat list of key value pairs, you might make a custom configuration section. In a configuration section you can add as much hierarchy as you like. You can read out structures like the following:

<jj.demos.configuration>

<items>

<item name="Name1" value="1" />

<item name="Name2" value="2" >

<childItem name="Child" value="3" />

</item>

</items>

</jj.demos.configuration>

You can create any kind of nesting you want. With classic .NET, reading out nested configurations requires about 1 ½ hours of programming for a simple structure. With our own framework, it is easy.

You have to include the following line in the configSections element in your config file:

<section name="jj.demos.configuration" type="JJ.Framework.Configuration.ConfigurationSectionHandler, JJ.Framework.Configuration"/>

You only need to replace the name ‘jj.demos.configuration’ with your assembly name converted to lower case. (You can also use a custom name, which you then have to use explicitly in your code, but that is not advised.)

To read out the config you call:

ConfigurationSection config = CustomConfigurationManager.GetSection<ConfigurationSection>();

MyConfigurationSection is a class you have to program yourself. Its structure of properties corresponds 1-to-1 to the XML structure. The specific behavior of the mapping is documented in the summary of CustomConfigurationManager. In short: properties map to XML elements unless you mark the property with [XmlAttribute]. Array items are expected to have the class name as the XML element name, but the element name of the array items can be specified explicty with [XmlArrayItem]. Here is an example:

internal class ConfigurationSection

{

[XmlArrayItem("item")]

public ItemConfig[] Item { get; set; }

}

internal class ItemConfig

{

[XmlAttribute]

public string Name { get; set; }

[XmlAttribute]

public string Value { get; set; }

public ItemConfig ChildItem { get; set; }

}

Note that C# will follow the convention that property names are pascal case, while this automatically maps to the convention in XML, in which element and attribute names are camel case.

#### appSettings

An appSetting looks as follows in the App.config or Web.config:

<appSettings>

<add key="TestInt32" value="10"/>

</appSettings>

Reading out the appSettings classically, is done as follows:

int testInt32 = Convert.ToInt32(ConfigurationManager.AppSettings["TestInt32"]);

But you get errors if you mistype the string and an invalid cast exception if you convert to the wrong type. The alternative in Framework.Configuration is strongly typed. You first need to define an interface:

internal interface IAppSettings

{

int TestInt32 { get; }

}

This interface defines the names and types of the settings. To retrieve a setting you use:

int testInt32 = AppSettings<IAppSettings>.GetValue(x => x.TestInt32);

It automatically converts to the right data type and allows you to use strongly-typed names.

#### connectionStrings

Reading out connectionStrings is similar to reading out the appSettings. Connection strings in the App.config or Web.config look as follows:

<connectionStrings>

<add name="OrderDB" connectionString="data source=10.40.XX.XX;Initial Catalog=OrderDB..." />

</connectionStrings>

This is the classic way of reading it out:

string connectionString = ConfigurationManager.ConnectionStrings["OrderDB"].ConnectionString;

This is the alternative in Framework.Configuration:

string connectionString = ConnectionStrings<IConnectionStrings>.Get(x => x.OrderDB);

You need to define an interface to be able to use the strongly-typed name:

internal interface IConnectionStrings

{

string OrderDB { get; }

}

### Conversion

See: 'Converter', 'TryGet-Insert-Update', 'TryGet-Insert-Update-Delete / Collection Conversion', 'Singular, Plural, Non-Recursive, Recursive and WithRelatedEntities' under 'Design Patterns'.

### Defaults

Implemented as side-effects that go off in a manager class’s Create methods. See ‘Side Effects’, ‘Manager’ under ‘Design Patterns’.

### Debugging

See DebuggerDisplays under Patterns.

### Entity Model / Data Model

<TODO: Discuss different data modeling options. Among which entity models.>

<TODO: Unintended n -> 1 relationships.>

<TODO: Is this statement true? When there is a 1-to-1 relationship, not to be confused with a 0-to-1 relationship, there is never a technical necessesity to have 2 entities, they may just as well have been 1.>

<TODO: Explain No polymorphism in entity models + the solution.>

### Entity Status Management

Entity status management (or ‘object status management’) is the recording of whether an entity is new, dirty, clean or deleted. Also it is recording if individual properties are dirty or clean. Currently entity status management is done explicitly by using an EntityStatusManager class, that is simply a wrapper for some dictionaries and HashSets that store this information. Then EntityStatusManager is then passed around the presentation and business layer for a particular functional domain.

There is are reusable EntityStatusManager classes in Framework.Business, but you are probably better off custom programming one for every business domain that needs it. That custom-programmed class can then be more specific about exactly which entities and properties get status flagging instead of leaving it up to the entity status writers to guess what entity status reporting is needed and entity status readers to guess of what entities and properties it can expect status to be properly supplied. With a specifically programmed EntityStatusManager you could make members like IsNew(Order) and NameIsDirty(Customer), to be way more specific about what entity status management you need.

#### Alternatives

The consequence of explicit entity status management through the EntityStatusManager class is that if you forget to call it, the entity status may not be correctly reflected by the EntityStatusManager. An alternative is to leave entity status management up to an ORM or other persistence technology. Not all persistence technologies provide this information. To consistently have entity status management through IContext across all platforms, Framework.Persistence should offer its own alternative to entity status management for persistence technologies that do not provide it. This is a difficult task and a project on its own. To lay the responsibility over entity status management at the Persistence side, it would make Framework.Persistence much more complicated, and would require at least a form of property interception to respond to property changes to record IsDirty status for properties. Complicating Framework.Persistence also harms the more or less impartial nature of it, since it should be an interface onto other persistence technologies, rather than a replacement of it.

This is why the explicit status management solution won over the entity status management in the persistence framework.

### Enums

#### General Rules

* Use the ‘Enum’ suffix for enum types e.g. OrderStatus**Enum**.
* Always give an enum the enum member Undefined with value 0:

enum MyEnum

{

**Undefined = 0**

}

This prevents you from accidently forgetting to assign the enum value.

* Prefer not using specific underlying enum types. Enums ‘derive’ from int by default, but you can e.g. do the following, which is not recommended:

enum MyEnum **: long**

{

}

* When taking action depending on an enum value, you might use a switch statement.  
  In that case always specify the default case and throw an exception in the default case:

MyEnum myEnum;

switch (myEnum)

{

case MyEnum.MyEnumMember1:

// Do something

break;

case MyEnum.MyEnumMember2:

// Do something

break;

default:

throw new InvalidValueException(myEnum);

// OR:

throw new ValueNotSupportedException(myEnum);

}

Not only is it informative for the programmer debugging a problem and does it prevent processing invalid or incomplete data, it is also a fail-safe for the fact that an enum is a very weak type. You can assign any int value to it, even ones that are not an enum member!

enum MyEnum

{

Undefined = 0,

MyEnumMember1 = 1,

MyEnumMember2 = 2

}

**var myEnum = (MyEnum)3; // WORKS!**

The difference between throwing an InvalidValueException or a ValueNotSupportedException is that you would use InvalidValueException if all enum members except Undefined were part of the switch, because then it was not a sensible enum value. You would throw ValueNotSupportedException if the switch uses only some of the enum members, but other perfectly sensible members were not relevant in this particular case. But it is not a disaster to use these exception types interchangedly.

* Use enum member *Undefined* in place of *null*, so also *avoid nullable enum* types.

#### Enum-Like Entities

* Entity models often contain enum-like entities:

public class SectionType

{

public virtual int ID { get; set; }

public virtual string Name { get; set; }

}

Often you do not need more than these two properties.

It is common to end the enum-like entity type with the suffix ‘Type’ (not a strict requirement).

The Name property will be filled with the string that is exactly the enum member name:

new SectionType

{

ID = 5,

Name ="SubChapter"

}

* Enum-like entities have an enum-equivalent in the *Business* Layer:

public enum SectionType

{

Undefined = 0,

Book = 1,

Article = 2,

Paragraph = 3,

Chapter = 4,

SubChapter = 5

}

Note that the enums themselves do not belong in the entity model, but in the Business layer.

* It is *not* recommended to give enum-like entities an inverse property to the entities that use it.

public class SectionType

{

// NOT RECOMMENDED!

public virtual IList<Section> Sections { get; set; }

}

The problem with this is that the list is likely to become very large, and maintaining this list (for instance in the LinkTo methods) can result in queries very harmful for performance, while you are not even noticing you are doing anything significant.

* To make assigning an enum-like entity easier, you can put extension methods in your *Business* layer. You can put this in the *Extensions* folder and call the class *EnumExtensions*. They also ensure consistency in the way that enum-like types are handled. The enum extensions allow you to write code as follows to assign enum-like entities:

SectionTypeEnum sectionTypeEnum = section.GetSectionTypeEnum();

section.SetSectionTypeEnum(SectionTypeEnum.Paragraph, \_sectionTypeRepository);

Here is an example implementation of the extension methods:

public static SectionTypeEnum GetSectionTypeEnum(this Section section)

{

if (section == null) throw new NullException(() => section);

if (section.SectionType == null) return SectionTypeEnum.Undefined;

return (SectionTypeEnum)section.SectionType.ID;

}

public static void SetSectionTypeEnum(this Section entity, SectionTypeEnum enumValue, ISectionTypeRepository repository)

{

if (repository == null) throw new NullException(() => repository);

if (enumValue == SectionTypeEnum.Undefined)

{

entity.UnlinkSectionType();

}

else

{

SectionType sectionType = repository.Get((int)enumValue);

entity.LinkTo(sectionType);

}

}

#### Localization

* Localization of the enum member display names is done by means of resources, usually in the Resources.resx in the Business layer. (See the ‘Resources’ pattern and Appendix B for explanations on how to manage resources). The key of the resource should exactly match the enum member name.
* The following code allows you to retrieve an enum member display name:

Resources.ResourceManager.GetString(SectionTypeEnum.Paragraph.ToString())

But a helper extension methods can make the code much more readable. This allows you to for instance use:

string str1 = ResourceHelper.GetSectionTypeDisplayName(section);

string str2 = ResourceHelper.GetPropertyDisplayName(sectionType);

string str3 = ResourceHelper.GetPropertyDisplayName(sectionTypeEnum);

string str4 = ResourceHelper.GetPropertyDisplayName("Paragraph");

Put a class in your Business.Resouces namespace, can it for instance ResourceHelper. These are examples of such ResourceHelper methods:

public static class ResourceHelper

{

public static string GetSectionTypeDisplayName(Section section)

{

if (section == null) throw new NullException(() => section);

string str = GetPropertyDisplayName(section.SectionType);

return str;

}

public static string GetPropertyDisplayName(SectionType sectionType)

{

if (sectionType == null) throw new NullException(() => sectionType);

string str = Resources.ResourceManager.GetString(sectionType.Name);

return str;

}

public static string GetPropertyDisplayName(SectionTypeEnum sectionTypeEnum)

{

string str = Resources.ResourceManager.GetString(sectionTypeEnum.ToString());

return str;

}

public static string GetPropertyDisplayName(string resourceName)

{

string str = Resources.ResourceManager.GetString(resourceName);

return str;

}

}

#### TODO

<TODO: The following things are not yet discussed there:

* If you use Guids as primary keys, you cannot use enums for the ID’s of enum-like entities.
* ID’s columns of enum-like entities are never auto-increment.
* Indexes on enum-like columns are not necessary because they do not have a lot of variation in values, which makes an index not very effective, and also you will not often join or search on an enum-like column.>

### Errors

<TODO: Describe that there are several ways to handle errors: Exceptions, validation messages, status enums and booleans.>

<TODO: Practices or patterns: Introduce returning a bool as an alternative for returning success or failure.

* Sounds like a practice.
* But also sounds like a pattern.
* But it also sounds like the philosophical case of the aspect of validation and error handling, which is quite polymorphic in sofware development these days: string messages, booleans, status enums and exceptions. A lot of different ways of handling similar things, really. It works out OK in practice, but it does not seem very elegant.>

### Exceptions

* Do null-checks on arguments of public methods.
* Sometimes miscellaneous checks need to be performed on public methods.
* Arguments of private methods do not have to be checked if the class already enforces the rule elsewhere.
* For null-checks, use NullException (in Framework.Exceptions).
* Use NotImplementedException for methods you have not finished yet.
* Use NotSupportedException for interface methods that are not supported.
* Use InvalidValueException or ValueNotSupportedException (in Framework.Exceptions) in the default in a switch statement over an enum value.
* Use other exception types out of Framework.Exceptions.
* Otherwise just throw the Exception base class.
* Mention the variable or property name in the exception message.
* Mention the ID of an object in the exception message.
* You can put alternative keys or other data in the exception by using anonymous types:

throw new Exception($"Item with {new { name }} not found");

This will produce a message like: Item with { name = "Item 1" } not found.

* Possibly mention the invalid value in the exception message.
* For most exceptions, use dry formulations such as ‘not found’, ‘cannot be null’ and ‘must be greater than 0’.
* Put exceptions at the beginning of a method if possible.
* Do not use exception filtering (catching specific exception types) unless you absolutely have to:

try

{

// Do something

}

catch (IOException)

{  
}

In fact, prefer not to retrieve information by catching an exception at all.

* To show a full exception message Exception.ToString() does a pretty good job including inner exceptions. If you like you can use ExceptionHelper from Framework.Logging to get a neatly formatted exception text. It also has a GetInnermostException helper method.

### Facades

Facades might sound like a pattern to you, but regardless of whether you use the facade pattern or not, your system has facades whether you know it or not. Most systems have points where multiple responsibilities come together and are combined.

If you have an MVC web app and you think you do not have facades, then it is probably the controllers that are your facades.

If you have a poorly layered Windows app, you probably have a lot of facades: each form probably is one.

You could even say that a very flat, unstructured user app’s facade is the user interface itself, which combines all the underlying code together into a simpler view on it.

In an architecture with many different business logic objects, that each take an arbitrary set of dependencies, all of those business logic objects are facades. You could ask yourself questions about the large gedree of interdependence in your system, and if the responsibilities are separated clearly enough. You could try to make your business logic objects operate as independent from eachother as possble, and only have a few that combine a lot aspects of together by delegating to smaller business logic objects.

These are all options to choose from. You can just mess around and do whatever. Or you can actively think about the choices you make about where you put your facades.

### Inverse Relationship Management / Inverse Property Management

See ‘LinkTo’ under ‘Design Patterns’.

An alternative is the OneToManyRelationship and ManyToOneRelationship classes from the Framework.Business assembly.

<TODO: Consider incorporating these ideas here:

* Idea 2015-04-29: Inverse property management with a List and a HashSet to make operations not n-problems...
* make story about inverse property management in property setters. A general description and maybe later mention the helper classes like OneToManyHandler if they are programmed. (They are.)
* In the software aspects section mention a summary of inverse property management methods, even though they have been individually described in other places in the document.
  + Reasons not to have an inverse property:  
    - Enum-like type  
    - Loosely linked entity  
    - 1-to-1 relationship  
    - The inverse relationship would result in ridiculously enormous lists. >

### IO

<TODO: Write description. Mention CsvReader e.d. The use of .NET serialization API’s XML, etc. Mention the StreamHelper. >

### Logging

Be careful how much you log. Logging unhandled exceptions is usually good enough. If you log a lot, it creates a performance penalty and can impose a serious strain on your infrastructure. Servers have crashed under the pressure of logging. A simple try-catch on a main level and a call to the logger will usually suffice.

For logging we will use our own API: Framework.Logging. It has an easy interface and simple configuration. It allows you to log to a file or debug output and is extensible to support more such logging channels. You can use a log level with your log calls and configure which log levels are included in which logging channel. For instance: you might only log exceptions to a file, but log debug information to the debug output.

Config example:

<configuration>

<configSections>

<section name="jj.framework.logging" type="JJ.Framework.Configuration.ConfigurationSectionHandler, JJ.Framework.Configuration"/>

<section name="jj.framework.logging.file" type="JJ.Framework.Configuration.ConfigurationSectionHandler, JJ.Framework.Configuration"/>

</configSections>

<jj.framework.logging>

<loggers>

<logger type="DebugOutput" level="Debug" />

<logger type="File" level="Exception" />

</loggers>

</jj.framework.logging>

<jj.framework.logging.file

filePathFormat="C:\Log\JJ.Utilities.MyUtility-DEV-{0}.log"

filePathDateFormat="yyyy\_MM\_dd\_HH" />

</configuration>

If you insist on using Log4Net, make a separate ILogger implementation behind which you hide Log4Net. The downside of Log4Net is that its configuration can be quite verbose and complicated. Framework.Logging is simple and can run on all platforms.

### Multi-Language / Translations / Culture

For button texts and other labels in an application: see ‘Resources’ under ‘Other Patterns’. That does not solve multi-lingual user data, for which multiple solutions are possible.

<TODO: Mention: Setting the thread culture rather than custom handling of the current culture. Also: prefer using the culture name as a key in the database directly, rather than a surrogate key. >

#### Separate Content Items

One option to support multi-language, is for a content item to be only available in a specific language. That means a different language gets totally separate content items.

#### Loosely Linked Translation Entities

One possible solution is each possble naming / grammar structure to each have a generic entity type, that can be tied to an arbitrary entity:

NameAndDescription { ID, Name, Description, CultureName, EntityTypeName, EntityID }

SingularAndPlural { ID, Singular, Plural, CultureName, EntityTypeName, EntityID }

The combination { EntityTypeName, EntityID } is a alternative key to the entity. This makes the translation item structure is independent on the model it is applied to, which can be a benefit.

#### Many Foreign Keys

Another alternative is to give the translation item entity a whole bunch of foreign keys: one for each possible translatable entity type.

NameAndDescription { ID, Name, Description, CultureName, **ProductID, DepartmentID** }

A downside of that is that the table structure is dependent on the domain model you applied it to. This can be a problem if you want your translation structure to be very isolated from the other business domains or used by business domains that you do not develop yourself. It really depends on your requirements whether this is a problem at all.

#### Comparison Loosely Linked vs Many Foreign Keys

The foreign key solution does have a big benefit over the generic key solution, because ORM’s will cache the entities in memory and be immediately available throught the object graph, even translation items that have not been committed to the database yet. With generic keys, you cannot query the translation items until they are flushed to the database.

To work with non-flushed loosely linked translation items, you would have to do some sort of caching. You could do the caching in the repositories / data access layer, but that does increase the logic complexity of your possibly so simple and elegant data access layer. You could also opt to make caching a business logic concern and pass around entity cache objects or translation facades around your business layer, as a substitute for getting them from a repository directly, which would not work for non-flushed (‘uncommitted’) entities.

A lot of work to use the loosely linked entities. This is not unique to loosely linked translation entities. It is a problem with any alternative key, that non-flushed entities cannot be retrieved with a (LINQ) query.

### Naming

See ‘Names’ under ‘Coding Style’.

### Paging

All page numbering starts at 1. Even though we usually start counting at 0 as programmers, to the user the first page is still 1 and it is very confusing if you do not carry through the same numbering throughout the whole software layering. Only right before you retrieve something from a data store you may convert the numbers to fit your data store’s needs.

Throughout the software layering we pass through 1-based page numbers and page count. Our data store may need a first index instead, but we only convert to that number as deeply into the layering as possible.

<TODO: Describe programming practices for working with paging in views.>

### Parsing

<TODO: Explain how you could structure your parser code? >

### Performance

<TODO: Give a few pointers to performance issues. Like reflection has a performance penalty compared to literal values, so do expressions. Pre-calculation and caching. >

<TODO: Describe: When you use a dictionary and when to use a list (there is a tipping point) >

<TODO: Aspects, Performance: Some optimizations seem only possible with inheritance and many classes, such as the fastest static dispatch and e.g. preventing dictionary lookups by having only one value stored per object instead of a dictionary for multiple objects. Oh, that is not necessarly inheritance. But why is inheritance the only construct that can do that, because inheritance also comes with downsides.

> It is because delegates are not as fast because the pointer needs to be dereferenced, and delegates may have a more confusing syntax in C# for structures that complex, while inheritance syntax is clearer in C#. Also: the prevention of dictionaries has nothing to do with inheritance, it has to do with graphs, which can be accomplished with objects referencing eachother. The reason you seem bound to a construct for the OperatorCalculator is that for CalculateSample have a reference to a specific SampleCalculator you need to turn the CalculateSample method into a SampleCalculator class, because functions cannot contain have instance members.>

<TODO: Aspects, Performance: Detail about nested loops: just traversing multiple levels is not a bad nested loop. Lookups of 5 to 7 items do not require a dictionary.>

### Persistence

To access a data store (usually a database), Framework.Persistence will be used. Through that framework you can access data using different underlying persistence technologies, such as NHibernate and Entity Framework or even flat files or XML. The framework gives you a single interfacing regardless of the underlying persistence technology, loosely coupling the business logic and front-ends from the way you store your data.

The main interface of the framework is IContext.

See also: ‘ORM’.

### Platform Compatibility

<TODO: Short description of that there are platform compatibility helpers. >

### Reflection

<TODO: You might go into the important utilities we have in the JJ.Framework and basics of System.Reflection including tips of when to use reflection. The use of the ReflectionCache.>

### Scheduling

Various solutions are available for scheduling a process (periodically) in time.

* Windows Task Scheduler
* Immediately running a process in the background in a web application.
* Using a job scheduling API. For instance ‘Quartz’ scheduling.

<TODO: Go into the various options and explain a little further.>

### Security

<TODO: Security needs a lot more topics. IP checking, encrypting information, secure HTTP, preventing various sorts of injection… >

<TODO: Check if these topics are covered, otherwise make a neat description of it: Password hashing, .NET has intrinsic security API’s, but when you use Framework.Security it allows you to interface with a security API through a common interface, which makes it easier to switch to another security API when we want to. Framework.Security keeps us from being strongly dependent on a specific security API. Mention the IPChecker in JJ.Framework.Web?>

<TODO: - Aspects, Security: If content is to be protected with authorization, then for partial presenters you need to do authorization checks if the presenter cass is public, and do not have to do authorization if the presenter class is internal.>

for enum-like tables.

<TODO: Mention: Security? Guids can be safe for security. For instance, for smaller underlying entities you could not guess the ID and sneekily change someone elses data, when only the user-ownership of higher objects are checked.

In other words: If you can enter ID’s of child objects by inspecting HTML, you can screw up another user’s data or another document’s data if you do not check if the original belongs to the right document / user.>

Authentication, authorization and user rights management in the application architecture will be interfaced with using pretty much the same pattern as the way we interface with persistence. Just like we create an IContext and repositories in the top-level project, often an MVC app, and pass it to the layers below that, the security context is also created in the top-level project, and passed to the layers below that. Both persistence and security are infrastructural things, and they will be handled in a symmetric way.

There are the following interfaces:

IAuthenticator

IAuthorizer

IRightsManager

The interfaces might have different implementations, depending on the underlying security technology used.

IAuthenticator will validate if a user’s credentials are correct.

IAuthorizer will verify if that user is permitted to access certain parts of the system.

IRightsManager will allow you to manage and change the users’ rights.

IAuthenticator

{

bool IsAuthentic(string userName, …);

void AssertAuthentication(string userName, …);

}

IAuthorizer

{

bool IsAuthorized(string userName, params string[] securablePathElements);

void AssertAuthorization(string userName, params string[] securablePathElements);

}

IRightsManager

{

bool UserExists(string userName);

bool UserIsLocked(string userName);

void CreateUser(string userName, string password);

void DeleteUser(string userName);

bool ChangePassword(string userName, string oldPassword, string newPassword);

bool ChangeUserName(string oldUserName, string newUserName);

string ResetPassword(string userName);

bool UnlockUser(string userName);

void Grant(string userName, params string[] securablePathElements);

void Revoke(string userName, params string[] securablePathElements);

void CreateSecurable(params string[] securablePathElements);

void DeleteSecurable(params string[] securablePathElements);

bool SecurableExists(params string[] securablePathElements);

IList<string> GetPageOfUserNames(int pageNumber, int pageSize);

int GetUserCount();

}

### Side Effects

See ‘Side Effects’ under ‘Design Patterns’.

### Text Processing

<TODO: Write text. Include: Functions in JJ.Framework.Common, including StringSplit and the StringSplit with quotation.>

### Transactions

<TODO: Write text. Include: Discuss what NHibernate does, explicit and implicit commit, IContext, how to use the SqlExecutor API. Discuss view model transaction and how to work transactionally regardless of how a database does it.

- Describe transactionality: usage of rollback and commit. Document that flush is an exception and dangerous and if used, must be commented with the reason why you use it.

- Framework: The way Commit and Rollback work now, an intermediate commit means you cannot use previously gotten entities. You cannot use entities gotten from a previous transaction, because under the hood the NHibernate session is replaced by a new one, meaning the previously gotten entities are connected to a closed session. I do not know how much of a problem that actually is. Perhaps it is OK. Perhaps I should have opened another transaction on the same session somehow. I’m not sure. >

### Unit Testing

If you would unit test everything, it could cost you 60% of your development time, which is a serious tax to pay. Therefore, unit testing is not mandatory. It only *supports* the goal of testing. However, in certain cases unit testing can be the most efficient way of testing.

Here are a few examples where unit testing could be useful.

Unit testing can be handy to debug a specific procedure in the system, without having to go through a user interface and several layers in between.

Unit testing is also handy for very important functionality that must be guaranteed to work. Price calculations are a good example where unit testing becomes important. Not only must prices always be correct, but also when a price calculation is slightly off, it is easily missed in manual testing.

Another case where unit testing comes in handy is when a calculation has many different variations. Sometimes manual testing might only cover 8%, while 50 unit test cover 99% of the situations and can be run each time you release the software.

But in many cases simply debugging and testing functionally is still a better choice, for efficiency’s sake.

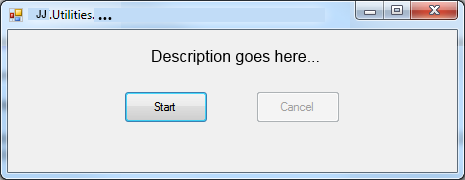
### User Interface

<TODO: Make a final text out of these preliminary texts. User Interfaces have a ton of ways to implement them. Especially due to the large amount of different presentation technologies that exist. But I like to keep a little independence from specific presentation technologies, by at least abstracting my views to view models, which are just simple DTO objects that describe the data that is shown on screen. More such patterns can be found under ‘Presentation Patterns’. ViewModels can then be applied to your UserControls, cshtml, exposed through Web API’s or consumed as json in JavaScript UI’s. Creating a ViewModel can be independent on the specific presentation technology you use. This is just a handful of choices you could make regarding your UI. One could wonder if User Interface is really just one aspect, since it covers about half your code base.>

### Utilities

Utilities are processes that are not run very often. Utilities contains small programs for IT. For example: load translations, things to run for deployment.

Framework.WinForms contains a reusable window, SimpleProcessForm, in to start and cancel the process and show progress information.



Here is a code example:

public partial class MainForm : SimpleProcessForm

{

public MainForm()

{

InitializeComponent();

}

private void MainForm\_OnRunProcess(object sender, OnRunProcessEventArgs e)

{

var executor = new MyExecutor (x => ShowProgress(x), () => !IsRunning);

executor.Execute();

}

}

The MyExecutor class may look as follows and can call its callbacks at its own discretion:

internal class ExecutorDemo

{

private Action<string> \_progressCallback;

private Func<bool> \_isCancelledCallback;

public ExecutorDemo(Action<string> progressCallback = null, Func<bool> isCancelledCallback = null)

{

\_progressCallback = progressCallback;

\_isCancelledCallback = isCancelledCallback;

}

public void Excecute(IList<MyClass> list)

{

if (list == null) throw new NullException(() => list);

DoProgressCallback("Starting.");

foreach (MyClass item in list)

{

DoProgressCallback("Busy..."); // TODO: include percentage or '3/100' in the text.

if (DoIsCancelledCallback())

{

DoProgressCallback("Cancelled.");

return;

}

}

DoProgressCallback("Finished.");

}

private void DoProgressCallback(string message)

{

if (\_progressCallback != null)

{

\_progressCallback(message);

}

}

private bool DoIsCancelledCallback()

{

if (\_isCancelledCallback != null)

{

return \_isCancelledCallback();

}

return false;

}

}

### Validation

See ‘Validators’ under ‘Design Patterns’.

## API’s

<TODO: API’s: Make sure you mention all the important in-house API’s and external API’s in the documentation.>

<TODO: Mention JJ.Framework.

JJ.Framework documentation: Separate document for overviewing what’s in JJ.Framework. Otherwise (lead) developers that use it would have to have a chrystal ball to know what’s in there.>

### AJAX

We make our own wrapper ajax methods around ones from e.g. jQuery, so we can AJAX with a single code line and handle both partial loads and full reloads.

We choose full loads first, before resorting to AJAX. See ‘First full load – then partial load – then native code’.

### Embedded Resources

<TODO: Write text.>

### Entity Framework 5

*<TODO: Add story about enabling MSDTC and transactionality.>*

### JavaScript / TypeScript

<TODO: Describe Framework.Javascript and why you might avoid JavaScript and why you might not. Also mention TypeScript.>

### NHibernate

<TODO: Describe thgis problem with polymorphism:

API’s, ORM: Arch: when an entity is a proxy or not a proxy, could reference comparison fail?

> When you retrieved a polymorphic type from HNibernate using the base type it returns a Proxy of the base type instead of a proxy of the derived type, which makes reference comparisons between base proxies and derived class proxies fail. You can then unproxy both and it will return the underlying object, which is indeed of the derived class, upon which reference comparison succeeds. But if you can also get failing reference comparisons another way. If you unproxied a derived type, and retrieve another proxy of the derived type, reference comparison should also fail.

>> This I want to test.

So always do ID comparisons, because reference comparisons can fail.

So for polymorphic entities always Unproxy before evaluating their type.>

<TODO: describe which NHibernate methods to use and what features to avoid. Do not map binary and other serialized data fields using NHibernate, because it gives terrible performance. Use separate SQL statements for retrieving blobs. Also: always include bride entities for 1-to-n relationships, never let the two sides of the 1-to-n relationship refer to eachother directly. Always go through a bride entity. Always have surrogate keys in a bridge table, not just the composite key. Otherwise you will get problems with ORM mapping technologies crazy-complicated guarding of integrity of object graphs and passing around composite keys all the time, no-good hashing keys, ugly URL’s, etc.>

<TODO: Describe more of the pitfalls and dos and don’ts around NHibernate and also FluentNHibernate.>

### ORM

Here is a ubiquitous quirk of ORM:

Many methods of IContext work with uncommitted / non-flushed entities: so things that are newly created, and not yet committed to the data store. But IContext.Query usually does the opposite: it only returns committed / flushed entities. This asymmetry is common in ORM’s and doing it any other way would harm performance.

<TODO:

* Reformulate the ‘ubiquitous quirk of ORM’: TryGet and Get MIGHT return uncommitted, non-flushed objects, but properties that lead to related entities usually return the uncommitted, non-flushed objects. IContext.Query or NHibernate’s ISession.QueryOver only return flushed objects. It is still a valid point that one time you get the uncommitted objects and the other time you do not. Only the way it is formulated is not entirely accurate.
* Also find a different word for ‘ubiquitous’.
* NHibernate FlushMode.Never or FlushMode.Commit does not prevent all intermediate automatic flushes. Flushes can be executed upon calling Save() even though the FlushMode.Commit's summary suggests otherwise. This happened when calling Save on a child object before calling it on the parent object. Internally then NHibernate asked itself the question if the child object was Transient and while doing so, it apparently wanted to get its identity, by executing an insert statement onto the data store. This caused a null exception on the 'ParentID' column of the child object.So the solution is either to create entities in a specific order: first the parent object, then the child object, or to choose a completely different identity generation scheme.
* ORM read-write order is relevant.
* Mappings: do not solve n-to-n relationships with (NHibernate) mappings. Always use bridge entities.>

<TODO: A problem with ORM: meet-in-the-middle querties. You have two ends of a graph, you filter both ends and then want what is in the middle.>

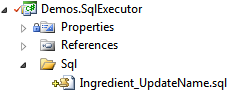
### SQL

<TODO:

* + No SQL strings:
    - Talk about not building up SQL strings in code.
    - No parameter concatination
    - Trick to prevent conditional blocks of sql. (@value is null or Value = @value)
  + No SQL in the code. Use SqlExecutor and .sql files.
  + No string concatination of sql parameters.
  + Hiding the SQL behind a repository.
  + Mention that SQL for upgrading the database structure do not belong in your project and are managed differently as described under Database Conventions.
  + Placeholders >

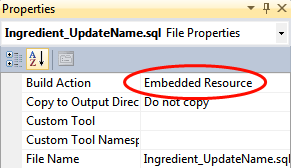
Executing queries onto a database is normally done through ORM, but if performance is an issue, it can be combined with SQL.

We will not use stored procedures or views. Instead we store SQL files directly in our .NET projects. We put the SQL files in a sub-folder named ‘Sql’.



The classic way of executing SQL in .NET is to use System.Data.SqlClient. Instead, we will use our own SqlExecutor API. With that we can execute SQL in a strongly-typed way, often with only a single code line.

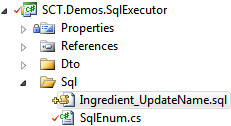
The best method of doing it is to make the SQL file embedded resources:



The SQL may look as follows:

update Ingredient set Name = @name where ID = @id;

Then put an enum in the SQL folder in your .NET project:



Add enum members that exactly correspond to the file names of the sql files:

namespace JJ.Demos.SqlExecutor.Sql

{

internal enum SqlEnum

{

Ingredient\_UpdateName

}

}

You need to create an SqlExecutor as follows:

ISqlExecutor sqlExecutor = SqlExecutorFactory.CreateSqlExecutor(SqlSourceTypeEnum.EmbeddedResource, connection, transaction);

We passed the SqlConnection and SqlTransaction to it.

Then you can call a method that executes the SQL:

sqlExecutor.ExecuteNonQuery(SqlEnum.Ingredient\_UpdateName, new { id, name });

The method names are similar to what you might be used to using SqlCommand. You pass SQL parameters along with the SqlExecutor as an anonymous type:

new { id, name }

The name and type of the variables id and name correspond to the parameters of the SQL. You do not need to use an anonymous type. You can use any object. As long as its properties correspond to the SQL parameters, they will be correctly used:

var ingredient = new IngredientDto

{

ID = 10,

Name = "My ingredient"

};

sqlExecutor.ExecuteNonQuery(SqlEnum.Ingredient\_Update, ingredient);

You can also retrieve records as a collection of strongly typed objects:

IList<IngredientDto> records = sqlExecutor.ExecuteReader<IngredientDto>(SqlEnum.Ingredient\_GetAll).ToArray();

foreach (IngredientDto record in records)

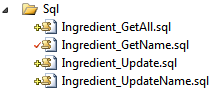
{

// ...

}

The column names in the SQL are *case sensitive!*

It is smart to let the SQL file names begin with the entity type name, so they stay neatly grouped together:



#### With NHibernate

If you use SqlExecutor in combination with NHibernate you have to use the NHibernateSqlExecutorFactory instead of the default SqlExecutorFactory:

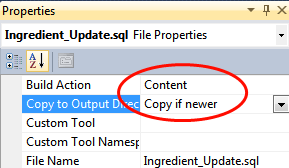
ISession session = ...;

ISqlExecutor sqlExecutor = NHibernateSqlExecutorFactory.CreateSqlExecutor(SqlSourceTypeEnum.EmbeddedResource, session);

This version uses an NHibernate ISession. In order for the SQL to run in the same transaction as the SQL that NHibernate executes, it needs to be aware of the ISession.

It is usually the best choice to include the SQL as an embedded resource, but you can also use files or literal strings.

#### Files instead of Embedded Resources



The is the code to create the SqlExecutor and execute an SQL file:

ISqlExecutor sqlExecutor = NHibernateSqlExecutorFactory.CreateSqlExecutor(SqlSourceTypeEnum.FileName, session);

sqlExecutor.ExecuteNonQuery(@"Sql\Ingredient\_Update.sql", new { id, name });

So the SqlEnum cannot be used anymore. You have to use the (relative) file path.

#### Strings instead of Embedded Resources:

It is not recommended to use SQL strings in your code! But it is possible all the same using the following code:

ISqlExecutor sqlExecutor = NHibernateSqlExecutorFactory.CreateSqlExecutor(SqlSourceTypeEnum.String, session);

sqlExecutor.ExecuteNonQuery("update Ingredient set Name = @name where ID = @id", new { id, name });

In that case no SQL files have to be included in your project.

### XML

Always choose XElement (LINQ to XML) over XmlDocument except when you have to use XPath.

Prefer the XmlHelper methods over using the API’s directly, because the helper will handle nullability and unicity better.

XmlToObjectConverter and ObjectToXmlConverter are also acceptable XML API’s.

### Framework.Business Relationships

The classes ManyToOneRelationship and OneToManyRelationship do inverse property management more or less automatically, which you then use in your models (rich, entity, API or otherwise). More or less: you still have to program classes that derive from ManyToOneRelationship and OneToManyRelationship and use them a certain way, but the result will be in a navigation property and collection property whose ends will be kept in sync.

<TODO: Perhaps clarify a bit more. Code example, perhaps.>

## Practices & Principles

The design concepts mentioned here are abstract principles that you can use in software design. Many of the principes have pros and cons that need to be weighed off in every decision about the software design.

This section also describes good and bad pratices while coding your software. They are concrete technical problems you may face in your work.

There are similar terms for bad practices. *Anti-patterns* are coding techniques that are not recommended. They are not forbidden, but they are an indication that there may be a better way to do it. *Code smells* are vague indications that there might be something wrong with your code and a better option may be available.

Code smells, anti-patterns and design principles can help you review (your own) code.

The topic are organized into groups of related topics.

### Responsibilities

<TODO: Write an intro on the kind of topics you will find here.>

#### Separation of Concerns

(Relatesed SOLID principles Single Responsibility Principle (SRP), Interface Segregation Principle (ISP) and Do Not Repeat Yourself (DRY)).

This is the concept that you split your code into pieces and create separate classes and methods. It is perhaps the single most important design principle of this software architecture.

Separation of concerns can be a split up into functionalities, such as code that handles a whole order and code that handles a separate product. The split up into functional concerns is usually similar to the split up into entities, for instance entities like Order, Product, Customer, but this is not necessarily leading for the split up into functionality.

Separation of concerns can also be applied to technical aspects, such as validation, calculation and security. For instance: you can split up the code to check the validity of an order’s data from the code that calculates the total price of the order. The split up into technical concerns is usually similar to the split up into design patterns.

##### Classes

In this architecture we apply both a split up into functional and technical aspects, creating a 2‑dimensional separation of concerns. This produces a matrix of classes:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Dto** | **Mapping** | **Validator** | **ViewModel** | **Presenter** | **…** |
| **Order** | OrderDto | OrderMapping | OrderValidator | OrderViewModel | OrderPresenter | … |
| **Product** | ProductDto | ProductMapping | ProductValidator | ProductViewModel | ProductPresenter | … |
| **Customer** | CustomerDto | CustomerMapping | CustomerValidator | CustomerViewModel | CustomerPresenter | … |
| **…** | … | … | … | … | … | … |

Plus: you can have specialized variations of these classes, for intance: OrderEditPresenter, SubscriptionProductValidator.

This is a very maintainable structure, because everybody that understands the system of organization, can find the code exactly where he would expect it. A change has low impact, because it can at most impact code that references that specific class. Code is better reusable and recombinable. E.g. you can reuse the same validations in multiple places or use specific validations in specific situations.

Using this split up into classes could impact data integrity, since every class can be independently used, and there is not one thing that guards all the rules. The solution is to use facades that guard (most of) the integrity rules by delegating to smaller business logic objects. The separate concern itself is actually better guarded, since a small class does not get entangled with other code, because it handled totally separately.

##### Assemblies

The separation into technical and functional concerns extends further than the class split-up. It can also be noticed in the assembly subdivision. Those also have a 2-dimensional separation of concerns:

These are the functional concerns:

JJ.Data.**Ordering**.NHibernate

JJ.Business.**Ordering**.Validation

JJ.Presentation.**Cms**

JJ.Presentation.**Cms**.Mvc

And these are the technical concerns:

JJ.**Data**.Ordering.**NHibernate**

JJ.**Business**.Ordering.**Validation**

JJ.**Presentation**.Cms

JJ.**Presentation**.Cms.**Mvc**

JJ.**Presentation**.Cms.**Mobile**

The assemblies are split up by functional domains.

There are separate assemblies for presentation, business and other main layers.

There are separate assemblies for specific protocols and technologies.

It is clear from the assembly name which technique is used, and what the functional domain is and the position within the software layers.

The result of this split up is that we are not stuck with a 1-to-1 relation between an application and its platform.

For instance: if an Ordering back-end was programmed to use a very specific persistence technology, you might only use it with NHibernate and an SQL Server database. You could not use the entity model on a platform that does not support this (e.g. mobile platforms). If a Cms front-end is programmed to specifically use MVC, it can only be deployed as a web site and not as a Windows application or mobile app.

By further splitting up our assemblies we can reuse the Ordering back-end in multiple front-ends. Furthermore: a single front-end could be deployed to either web or mobile platform and we can store entity models differently depending on the infrastructural context. On a mobile platform we might store an entity model in XML, while in a web environment we might store things in SQL Service using NHibernate.

##### Framework Assemblies

* <TODO: Describe: Split up in Framework assemblies: when a framework assembly extends a .NET API, make a different framework assembly for each part of .NET you extend. A framework API should only hook into one specific part of .NET. That way we have control over what parts of .NET we make our application dependent on, which makes it possible to develop for multiple platforms.>

#### Combination of Concerns

After separating all different aspects of both functionality and technique, you can recombine these separated aspects in specific spots in the code: facades or presenters or in very specific classes that are a machine to perform very specific functionalities in a completely controlled way. See ‘Facades’ under ‘Aspects ‘ and ‘Manager / Facade’ under ‘Patterns’.

#### 2 API’s for the Same Thing (bad)

Choose one API and stick with it. It is not recommended to use e.g. two different XML API’s in your application.

#### Copy-Paste Programming (bad)

Also called code duplication, copy-paste programming is the practice of implementing new functionality by copying a old code and then slightly changing it. This is very bad practice and creates a lot of slightly different copies of code, that make it difficult to change their mutual functionality. The alternative is to create one generic piece of code that can be used in multiple different ways and not repeat it (see also: ‘Do not repeat yourself (DRY)’).

Do note that you are allowed to repeat trivial code and two copies of code that must not affect eachother if one of them gets changed.

#### Do Not Repeat Yourself (DRY)

This is the principe that you should not repeat code. It is a general rule that you must then put the shared code in a separate class or method, and reuse the same code in multiple places.

But keep in mind that there are exceptions. Very trivial things can be repeated, and the same code might be repeated in placed where a change to one copy of the code, should not affect other parts of the system.

#### Dump Code Line Here (bad)

<TODO: Strategic mistake: Don’t just put a line of code somewhere that gets rid of the symptom. Ask yourself the question if it is the responsibility of that class or whether it really belongs elsewhere. TODO: Describe that in more detail.>

#### God-Object (bad)

<TODO: Find synonym.>

An object that is used everywhere and can do anything. Consider splitting it up into multiple classes and only using the classes you need where you need them. It is an indication of bad separation of concerns.

#### God Base Class (bad)

<TODO: Find synonym.>

A base that that is a union of the functionality needed in the derived classes, instead of containing just the basic functionality.

Consider moving code to the derived classes that need it, delegating to different helper classes instead of putting everything in the base class, or as a last resort add intermediate inheritance levels, gradually extending functionality.

<TODO: Rephrase this and make it part of the main text: You should use inheritance to share behavior / private implementation, not as a way make methods available from multiple places, or to give a generalized name to a set of types, even though they still have separate unique behavior. You should also not create a base class, that is about a side-issue, because then you have reserved your one inheritance slot with something unimportant. >

#### Granularity

Granularity can be compared to sand. Large pebbles are large granules, while fine sand is made up of a small granules. In code it means that a piece of code might tap into a big object, while it really only needs to depend on a smaller object. A piece of code may use a specialized object, while it can tap into a more generalized form. It can also be expressed as ‘defined at the wrong level’.

#### Helperitis (bad)

Helpers are static methods with static functions that support a specific aspect of programming for which no more than a flat list of methods is required.

Helpers are ofcourse helpful, but sometimes you can end up with code in which everything is delegates to helpers, obscuring what is actually going on.

#### Spread Responsibility (bad)

Keep the responsibility for one thing in one piece of code and do not spread it across multiple pieces of code. E.g. when setting defaults for a new object, try to keep that in one spot in the code.

(Not to be confused with the ‘single responsibility principle’.)

#### Too Many Responsibilities (bad)

When a class does too many different things, you might want to split it up into separate individual classes.

When a method does different things depending on its input, you might want to split it up into different individual methods. Most of the time in the code calling your method, it is already clear which of those different things are needed there, and it does not need to be generic.

#### Two Solutions for the Same Thing (bad)

If you see that a single problem area has two different ways of handling it, this is an indication that your design lacks elegance and you should go look for a single solution to solve the problem. Two solutions is maintenance overhead and it has the same downsides as other types of code duplication.

#### Vague Justification (bad)

<TODO: Describe this. Do not try to justify putting a piece of code in a spot where it does not seem to belong with some vague explanation. The separation of responsibilities in a system should be crystal clear. People have a tendency to look for vague reasoning that justifies what they just did, and will take that very far. It is called rationalization. Be critical. Can you clearly explain why some piece of code belongs there? If not? Where should it belong? Try the stategy of ‘Ideal Solution’. (There is a topic for that in this document.) That could help you figure out a better separation of concerns.>

### Error Checking

<TODO: Write an intro on the kind of topics you will find here.>

#### Be Stict

<TODO: Describe the ‘be strict’ concept.>

#### Error Hiding / Null-Tollerance (bad)

A lot of code contains too much null-tollerance, possibly because of being paranoid about getting exceptions. This is the wrong way to go. You MUST throw an exception. If you build in a lot of null-tollerance you will run into the problem that ‘nothing happened, and we have no error message’. Or ‘the data is corrupted and we have no error message’. What otherwise would have been a clear error message just turned into a horrible problem to solve, in the worst case we will not even be able to solve it at all.

It can also result in a lot of complex code that tries to recover from a faulty situation, that should never occur in the first place and really should result in an error.

If something is null, that should not be null, an exception MUST be thrown. You have to make your code strict when it comes to faulty data and throw an exception when it is encountered. Exceptions are there to tell us what’s wrong.

#### Null-Checks

<TODO: The rule ‘no null checks on lists’ is not true for the canonical model. >

<TODO: If GetByID returns null and you expected something, throw EntityNotFoundException<T>(id).>

<TODO: null-checks on parameters that the class uses directly, not if it is only passed to a method that does a proper null-check itself.>

*The most common programming error is a missing null-check.*

Code should be strict when it comes to nullability. The general message is: check for null if you expect it. But sometimes you can omit the null-checks to keep your code clean. Here are the rules:

* Entity models are anti-encapsulated: none of the data is protected.
* A data store often guards nullability.
* Validators should also guard nullability.
* Those two things determine whether an entity property is not nullable.
* You might mark the entity property with the word ‘not nullable’ in its summary.
* For not nullable properties you never have to do null-checks in your business logic, even though in theory null could be assigned.
* List-properties in the entity models require no null checks at all. They should be created in the constructor of the entity class and we simply assume nobody will assign null to it.
* This means that serious business logic should not be executed on entities that have not been validated yet. When you just retrieved entities from the data store, you may assume the data is valid.
* This saves us a lot of null-checks, which makes code more readable.
* The other entity properties are considered nullable.
* For nullable properties, business logic must have an alternative flow. Some business logic could throw an exception if a null is encountered. Other business logic might have to be null-tollerant and skip certain things. (Reflect this in the code by using words like ‘Try’ and ‘IfNeeded’.)
* Null-checks can be omitted if you know that a variable was verified before. For instance: if you throw an exception in the constructor in case an argument is null, you can leave out null-checks in the rest of your class.

See also: ‘Error hiding’.

Allow nulls as little as possible. Similar rules also apply to other integrity constraints (e.g. "> 0"), but null-checks are the most common.

Here are rules for null-checks for other constructs:

|  |  |
| --- | --- |
| DTO’s | Usually the same rules apply to DTO’s as do for entities. Especially if they just transfer data from SQL statements to application logic. |
| Strings | To check if a string is filled in, use IsNullOrEmpty: if (String.IsNullOrEmpty(str)) throw new NullOrEmptyException(() => str); So this is wrong: if (str == null) throw new NullException(() => str);  Another common mistake is this:  obj.ToString()  This will crash if the object can be null. This is the solution:  Convert.ToString(obj) |
| Value types | Value types, for instance int and decimal, cannot be null unless they are nullable, for instance ‘int?’ and ‘decimal?’, so do not execute null-checks on non-nullable value types. |
| Parameters | Execute null-checks on arguments of public methods. Use NullException (out of Framework.Reflection). You can omit null-checks on arguments of private methods, if the rest of the class already guarantees it is not null.  Avoid allowing null parameters. But if a parameter *is* nullable, you can denote this in several ways.   * Assign null as the default value of the parameter, so it is clear that it can accept null: private void MyMethod(object myParameter = null) * Document with XML tags, that it is nullable:   /// <param name="myParameter ">nullable</param> private void MyMethod(object myParameter)   * Add an overload that does not have the parameter: private void MyMethod() { } private void MyMethod(object myParameter) { } Ideally in the overload with the parameter, do a null-check. |
| ViewModels | ViewModels that are passed to Presenters may contain nulls. You can use the NullCoalesce pattern to resolve the nulls before processing the view model object, so that null-checks can be omitted from the rest of the code. |
| Our own framework API’s | For API’s in our own framework you can count on an object when you call a Get method. You have to take null into consideration when you call TryGet. |
| Your own application code | Conform to that same pattern in your own application code, so you know when you can expect null. |
| Third-party API’s | Some .NET API’s and third party API’s may return null when you call a Get method. Some do not. You have to learn which methods can return null and do null-checks appropriately. |

##### Alternatives

* Microsoft’s C# Language Design team is currently (2017-06-28) working hard at coming up with compile-time analyisis of nullability problems, that may make null-checks a thing of the past. They are not finished with it yet, and we have to wait and see what the outcome will be.

#### Process that Checks Itself (generally bad)

It is an anti-pattern for a process that just ran, to check if the data it wrote was correct. Why would it check what it just did? Shouldn’t the code just be correct? Those are questions you want to ask yourself. Also, perhaps it should just be unit tested, or functionally tested instead.

It is like you are writing the same logical steps twice: once for writing the data, once for checking the data. The effort put into the complexity of the error checking code, would have been better spent making the actual processing code work well. Also: the error checking code might incur an extra maintenance burdon; you just have more code to keep working as the system evolves. It would also have performance overhead. You spread the responsibility of one thing over two different pieces of code. Another problem that could occur is that you might be adapting one piece of code and then you get false error messages, because you forgot that you still had that other piece of code to adapt, that does almost the same thing.

In certain cases with very sensitive, error-prone code you could opt for the process to verify its own data afterwards, but this should be the exception rather than the rule.

#### Reject, Don’t Correct

Do not correct input data, but require that input data is correctly entered. Code that creates tollerance towards user entry errors can quickly get out of hand, while simply rejecting the use input with a validation message would suffice. It also gives the user more control over what happens, instead of the system’s wrongly interpreting the user input.

### Interfacing

<TODO: Write an intro on the kind of topics you will find here.>

#### ‘All’ and ‘Many’

The word ‘All’ is often misused in repository method names. Once just a selection is returned, it is not ‘All’ anymore. Use the word ‘Many’. So not GetAllBySearchText. Instead use GetManyBySearchText.

<TODO: If multiple items are returned by for instance repository methods, remember that the multiplicity should be reflected in the method name. For instance a method GetByCritia could be intended to return a list, but you cannot really see it from the name. One could assume it returns a single item. You could for instance call it GetManyByCriteria in that case, so you can see from the name that can return more than one. There are other ways to express multiplicity, such as the word ‘List’ or ‘Collection’ or a plural name, but as long as it is clear.>

#### Blackboxing and Whiteboxing

Blackboxing means that you put something in a little machine, something comes out, but you cannot see how it was processed exactly.

Blackboxing is the use of encapsulation to hide complexity. It is also a concept of creating an interface in front of an implementation: hiding what is exactly done.

Blackboxing can create a lower degree of dependence between different parts of a system, making things easier to change. It is also present in the concept of interfaces, allowing you to hide multiple implementations behind a similar interface.

Sometimes blackboxing creates problems, because not seeing exactly what is going on is a big downside. Being open about what happens could give a programmer the overview and control he needs. This is the concept of whiteboxing.

Both concepts are important and have their place. Whether blackboxing or whiteboxing is the right way to go, should be evaluated on a case-by-case basis.

Simply giving something a name that reveals its inner workings is a common form of whiteboxing.

#### Clarity over Brevitity

A longer name in code is better than a short, inspecific one. Even through you may think brevity supports readability, if it creates ambiguity, a longer, unambiguous name usually works out better.

For entity models, consider the name Order.OrderProducts, not Order.Products if the entity types are Order and OrderProduct, even though the first part of the expression ‘Order.Products’ already seems to imply it that it would be an OrderProduct. Because next to an OrderProduct entity, the model probably also has a Product entity and it would be very confusing that Order.Products would be a list of OrderProducts, as you would sooner think it is a list of Products from the name. Also it makes it harder to ‘guess’ what an entity model property is, if you abbreviate the names. Just use the full entity type name for property names and it will be far less confusing, especially to the ones that did not program your model. Again: yes, even when it seems obvious to *you*.

#### Conceptual Names (bad)

‘Conceptual names’ are bad practice. It is hard to define what that means. But it has to do with the names not being specific enough or only vaguely related to what it is really about.

Often combining a domain term with a design pattern gives you a more specific name:

CustomerViewModel

This instead of calling it just Customer or just ViewModel.

Here are a few more examples:

* CustomerListReload
  + It was a controller action name intended to be a AJAX variation of the Index action. Not only should CustomerList be replaced with Index, but also the word Reload is not clear. It may have something to do with reloading some piece of index, but it really is the AJAX variation of Index, so perhaps a suffix Ajax would be more appropriate. IndexAjax would have been a better name.
* A class name Cooking is also a good example of an conceptual name. Cooking? If you have to ask ‘What about it?’, you got a conceptual name, that should be made more specific.
* A view named \_CollectionListAction.cshtml: The name Action is a conceptual name. It has something to do with an Action. More specifically: multiple actions, and more specifically: it is an ActionBar. The word Action is too general. It can refer to a Controller Action, the .NET Action<T> class, etc. Perhaps \_IndexActionBar.cshtml would have been better.
* Conceptual names are also ones which do not include the pattern name at the end.

#### CRUD

<TODO: Explain.>

#### Delegitis (bad)

This is over-use of delegates, mostly as parameters. Here is an example of a method with too many delegates passed to it. The problem is that it is very unreadable:

public void MyDelegator(

Func<Child, Parent> getParent, Action<Child, Parent> setParent,

string parentPropertyName, string listPropertyName,

Func<Parent> createParent, Func<Child> createChild)

{

// ...

}

Solution: create a base class and a derived class that overrides the methods with specific implementations. This may create more classes, but it might be better overviewable.

abstract class MyBase

{

private string \_propertyName;

private string \_listPropertyName;

public MyBase(string propertyName, string listPropertyName)

{

// ...

}

public void Execute()

{

// ...

}

protected abstract Parent GetParent(Child child);

protected abstract void SetParent(Child child, Parent parent);

protected abstract Parent CreateParent();

protected abstract Child CreateChild();

}

#### Dependency Injection

<TODO: Change your view on dependency injection and rewrite that section. This is because SimpleInjector takes away a lot of drawbacks that Ninject has.>

For dependency injection we will not use frameworks like Ninject anymore. Ninject uses a ‘magic hat’ principle: an object came from somewhere and you have no idea where it came from or if the object is even there. NInject allows you to define a set of implementations of several interfaces centrally and retrieve that implementation from arbitrary places in the code:

// Bind it

Bind<IMyDependency>().To<MyDependency>();

// Use it

MyClass obj = new MyClass();

class MyClass

{

private IMyDependency \_myDependency;

public MyClass()

{

myDependency = ServiceLocator.Resolve<IMyDependency>()

}

}

We will not use this technique anymore, because it has serious disadvantages and the object oriented paradigm completely falls apart:

* You get null-reference exceptions, since there are no guarantees that the dependency is actually there.
* The setup of dependencies is tricky, because you would need to analyse all the code to actually know which dependencies you need to set up, or use an example, which is probably different for your new application.
* You get problems with using multiple instances of the dependency, and it is really hard to figure out where your object came from.
* It takes considerable hours of trouble shooting to set it up or to solve problems and you have a sense of having no control over what is going on.
* Some variations on the technique even require making members public that really need to be private.
* Using constructor arguments in the dependencies is not type safe.
* It is unclear from an class’s members and constructors that it is dependent on something, what it is dependent on.
* It creates spaghetti code where everything is potentially dependent on everything else.
* It gets worse because injected dependencies can be dependent on yet again more injected dependencies.
* Instantiation is slower, because it has to go through a framework.
* Also, dependency injection needs a framework that might not be supported on all platforms.

Here is the proper alternative:

IMyDependency myDependency = new MyDependency(); // Or another means of instantiation.

MyClass obj = new MyClass(myDependency);

class MyClass

{

private IMyDependency \_myDependency;

public MyClass(IMyDependency myDependency)

{

if (myDependency == null) throw new NullException(() => myDependency);

\_myDependency = myDependency;

}

}

It is simply a combination of interfaces and constructor arguments.

Now we have accomplished the same thing, only instantiation is explicit and not magic. You know you need to create the dependency (with dependency injection you did not). The dependency is null-checked (with dependency injection it was not). You can use multiple instances in the right places (with dependency injection multiple instantiation is tricky and has limited capabilities). It is faster, because it does not go through a framework and you do not need to include a framework, that might not work on all platforms.

#### Entity Design

< Entity design:

* (Dutch) Aangeven dat het beperken van tabellen belangrijker is dan constraints bewaken op database niveau omdat alles wat je in de database structuur aanmaakt, daar kom je ‘nooit meer’ vanaf en alles wat je in business logic oplost is makkelijker aan te passen.
* Keep models simple clean and stripped of all accessories, in particular entity models and especially canonical models. >

#### Execution Order Dependence (bad)

If methods only work if you execute them in a particular order, why not have one method that executes them in that specific order?

#### Handy Extras / Ya Ain’t Gonna Need It (bad)

Do not add things to you code (and in particular to interfaces) ‘that might be handy for the future’. The opposite is true. Extra code requires maintenance in case of changes. Also: If programmers use these ‘handy things’, they will be hard to get rid of and then you are stuck with it.

It is better to keep the code minimalistic and add the extras at the time that you actually need them.

Specialized case: Overloads that are never used, should be removed from the code.

#### Hatch / ‘Doorgeefluik’ (generally bad)

A method, that does not do anything but delegate to another method. For example: let’s say there is a method GetImage in both an ImageRepository and an ImageManager. All ImageManager.GetImage does is call ImageRepository.GetImage.

The thinking error might be that you want to consistently call the ImageManager for everything and that it is a good preparation for the future, because the Manager might add extra rules later.

But it usually a better plan to directly call ImageRepository.GetImage and leave out the method ImageManager.GetImage. If you leave in the method that does nothing, then when a deeper layer changes, you’d have to change a lot of pointless layers above it. Also by adding a method to the Manager class, you create the false illusion, that more is done than just retrieving an image, giving you a lessened sense control what is going on.

If you see a method that does nothing but delegate to another method you have to consider removing this method.

However, you could also consider that in this case maybe the hatch is a good thing. If the general rule is to always go through the manager / facade then a method in the manager or facade may be expected. For a simple get by ID you may be better off using the repository directly, otherwise you get depedencies on facades where you do not need them, and this due to the nature of facades, which can do anything, automatically creates a large degree of dependency on many different parts of the code.

#### Hollow Interface

A hollow interface is and interface with many implementations in which many members are not even implemented or do not do anything. This is an indication that there is something wrong with its design. It violates the Liskov subtitution principle from the SOLID principles. You may want to split up into multiple interfaces so that the implementation you are making is not hollow and all interface members are properly implemented. Depending on your system, there could be a downside to having multiple interfaces, because it could harm how accessible your code is to others or how accessible it is to new implementors of your interface. (SOLID supporters may deny this with a vengeance.) You can solve this and make Liskov happy by introducing booleans to your interface saying whether a method is actually supported. See ‘IsSupported’ under ‘Patterns’.

#### Interface Contamination

(Related to the Dependency Inversion Principle from SOLID.)

Interfaces are supposed to be lean. You should keep as much as possible out of an interface. An indication of an interface that is too rich, is for instance that a method has many parameters. You might see a method’s name and expect it does not need all those parameters. That indicates a degree of interdependency that is too high. This method’s responsibilities might have to be redistributed among different parts of the system. An interface might use a type from an API, while you might expect that interface to be API neutral. You might see a data class passed to an interface that you expected to be independent of that data model. These are all indications of interface contamination and the solution is usually to distribute responsibilities differently over different parts of the system.

Interface contamination becomes a big problem, when an interface is used in many different places. Then all those parts have a high degree of dependence on things they really have nothing to do with. It also makes interfaces difficult to implement, and poorly reusable.

#### Interface Neutrality

<Describe in general. Also refer to Interface Contamination and Interface Stability, leaky Abstractions>

#### Interface Stability

(Related to the Stable Abstraction Principle from SOLID.)

You can design the interfaces in such away that they do not tend to change much in the future.

This can be done by thinking about what the core of what you want is, and whether the interface is a dry enough representation of the kind of problems you want to solve with its implementations.

Not making assumptions about its use or its implementation can help.

Input/output transparency can both help and harm. By always passing input as parameters to the method, you take away assumptions about where that data came from, and makes it less hard to adapt when the data comes from elsewhere. But by passing input as parameters, you also increase the interface’s awareness of things that should just be implementation details and makes it harder to make the interface that work when implementations change. Leaky abstractions are the worst example of this. It is an art. You cannot apply a single solution to all problems here.

Interface stability can also be improved by choosing neutral collection types over specific ones, for instance using IList<T> and not Dictionary<Something, Something>. This can have a small performance trade-off, but does improve interface stability. Again: it is a trade off, an art of picking the right tool for the job.

(Stable abstraction principle is also one of the ‘secondary’ SOLID principles. Although, I (Jan-Joost van Zon) personally find anything I can find on the topic vague about whether they mean stability as in ‘does not change much’, or stability as in ‘does not have many bugs’. I also find things poorly explained, often written in a way that makes me feel you already need to understand the topic in order to understand the explanation, as if the author was just writing it for himself.)

#### IO Transparency

Input / ouput transparency.

<TODO: Describe. >

#### Kama-Sutra Pattern (bad)

So many overloads you cannot see which to pick.

#### Leaky Abstractions (bad)

(Related to the Dependency Inversion Principle from SOLID.)

An interface tries to abstract / generalize multiple similar problems into one solution. If an interface exposes the underlying solution, we speak of a leaky abstraction.

For example: this repository interface exposes the underlying NHibernate technology:

interface IMyRepository

{

Entity TryGetByCriterion(NHibernate.Criterion.AbstractCriterion);

}

The repository interface should not have shown NHibernate-related types, because it is supposed to hide the underlying technology. You can also do too much with the interface now. AbstractCriterion allows you to build any query you want. That is also leaky about this interface. It is the repository’s job is to offer a set of optimal queries. With the leaky interface above, a repository cannot do its job anymore.

The following example is better.

interface IMyRepository

{

Entity TryGetByCriteria(string name, DateTime dateCreated);

}

}

You might add extra methods or parameters if more filtering options are needed.

#### Loose Coupling

Loose coupling or a low coupling is the concept of keeping a low degree of dependence between things. Tight coupling or high coupling means many things are tightly dependent on eachother, making it hard to change one of those things without breaking or changing the other things. The benefit of loose coupling is that a change to one thing affects a minimum of other things.

Low coupling does not mean that the total amount of links between things is lower. The amount of links between things may actually go up. The low degree of coupling is about an individual type’s links to other types. That is what gives us the benefit of one change only affecting a limited set of other things.

One technique of limiting the degree of coupling is the use of interfaces. By letting code talk to an interface rather than directly to a specific implementation, this makes the code dependent on that interface without being directly dependent on multiple concrete classes. This makes you able to write one piece of code that handles multiple concrete things, which makes that code dependent on one type rather than the union of concrete implementations.

A symptom of high coupling is what happens if you hit Shift-F12 on a class name (‘Find all references’). If you get a whole lot of results, you have a case of high coupling and you might be in trouble. If you get very little results it is a sign or low coupling and you can make a sigh of relief.

There are cases where high coupling is normal. For instance in case of base classes, combinator classes, framework classes, simple types, canonical models.

#### Lying Names

Names in code should tell the truth. If a method does more than what the name says, then it is a bad name. If a method name only gives an indication what it does, but in fact it does other stuff too, it is a bad name. If a programmer could not come up with a good name, so just typed something vague, then this is bad practice.

#### Magic (bad)

When a call to a member does more than you would expect and has unintuitive side effects that you do not see.

Solution: execute the side effects explicitly, so you see what is going on, instead of letting the side effects go off automatically.

#### Magic Defaults (generally bad)

<TODO: Describe. >

#### Magic Numbers / Magic Strings (bad)

<TODO: Describe. >

#### Method Self-Sufficient

It can be better to repeat the same work in multiple methods, rather than do the shared work once and pass the result to multiple methods. This might harm performance, but this makes the methods self-sufficient.

It is worse for the method to only work if you do very specific work beforehand. It is usually better to then let each of the methods repeat the same work. Performance is the trade-off here, but it makes the methods more reliable. An alternative is to create an instantiatable class that does the shared work in the constructor to not repeat it in the instance methods.

#### Ripple-Effect

This is an analogy to throwing a rock in a pond. If you throw a rock in a pond, a wave is created that goes through the whole pond. In software programming, it means that if you change one piece of code it may require you to adapt many other pieces of code. It can be an indication of a bad design choice where too many things directly dependent on eachother. It is not always a bad design choice, but simply a hard to prevent high-impact change.

#### See from Name, not from Arguments

The name of a method should say what it does. It should not be inferred from the argument list what it does. You may think that if name not clear, see argument list, but having to analyse the arguments it valuable brain time that you loose, that can be prevented by a clearer method name. Closely related to the ‘Toilet-Role Principle’.

#### Spooky Action (at a Distance) / Cause and Effect too Far Apart

<TODO: Describe>

#### Syntactic Sugar

Not always bad practice, but is can be a cause of confusion.

Syntactic sugar is creating a notation that does not add anything, other than a simpler notation.

Object initializers are an example of syntactic sugar. In C# 2.0 you had to initialize an object’s properties as follows:

Cat cat = new Cat();

cat.Age = 7;

cat.Name = "Nala";

In later versions of C# you are able to do the same thing with the following code:

var cat = new Cat

{

Age = 7,

Name = "Nala"

};

These two pieces of code do exactly the same thing. The shorter notation introduced is ‘syntactic sugar’.

You can also program syntactic sugar into your own code. You can do this by introducing shorter names or a shorter notation, helper classes or implicit conversion operators (which can be very confusing). This may create a concise notation, but breaks the rule ‘clarity over brevity’. Often such notations lie a little about what is really going on. So use it sparsely and in most cases go for a more explicit notation.

(The object initializer notation above actually is the recommended notation, not undesirable syntactic sugar.)

#### Unclear Interfaces

Interfaces of classes, methods and other members must clearly show what the member will do. A method’s name should say what it does. Preferrably, if the method needs input, it should be a parameter, if a method has output it should be a return value. If an input parameter has an invalid value assigned to it, you should get an exception. The interface should guide the programmer, so there is really only one way to do it, and not several incorrect ways to do it.

Examples of unclear interfaces are:

* A method’s input parameters make the method do nothing, instead of throwing an exception.
* You can assign null to one of the parameters and the method will change its behavior completely. A better solution is to have two separate methods.
* The input of a method is a property, while the method is the only one using the property. A better solution is to remove the property, and pass it as a parameter.
* The output of a method is a property. It may be better to actually return the output, rather than assign a property. A programmer may not be able to guess that the effect of the method is that a property is assigned.
* A method does nothing unless a property is assigned.
* A property could have been a constructor argument. The class does nothing or throws exceptions unless the property is assigned, while really it could have been made a mandatory constructor parameter, so the object’s state is valid right after construction.

#### The Unwritten Agreement (‘het onderonsje’) (bad)

Two seemingly independent pieces of code only work if one piece of code makes assumptions about the implementation of the other piece of code. Another example of a code smell, that points to an unwritten agreement, is when an interface has elements to it, that are not obviously necessary. For instance a Dictionary parameter where it does not seem to be needed, perhaps because the calling code just so happens to be using a dictionary. You should generally avoid such scenarios. Different pieces of code should be self-sufficient and non-assumptious, but ofcouse it can depend on other trade-offs, such as performance and readability.

<TODO: Consider some of these phrases: ‘Onderonsje’ anti-pattern: polluting the interface of a method or class with members that break a pattern or with non-neutral types such as dictionaries, simply to make something maybe more efficient or another purpose. Usually the pollution can be prevented, because you can just do it higher in the call stack, rather than just passing things on to other methods.>

#### Wrapperitis (bad)

Do not make a class that simply wraps another class with no specific reason at all. For instance wrapping an API into a class that supposedly makes it easier, but really adds nothing new to it. You may be better of directly working with the underlying classes that actually do stuff, instead of having wrapper classes that suggest that they add something, but really do not have any additional value. It is annoying when you have a whole lot of classes and many times you wonder “What does this do?” and the answer turns out to be “nothing really”.

This is closely trelated to the ‘Hatch’ anti-pattern.

There can be a good reason to wrap something though: loose coupling and polymorphism: giving multiple things a mutual interface, while originally they did not have a common interface.

### Variables and Parameters

<TODO: Write an intro on the kind of topics you will find here.>

#### Double Negatives (bad)

If you give a variable name the word ‘not’ in it, then your code is likely to be less readable, since you might get a lot of double negations like "!not" and such. It is usually a better idea to use the ‘positive’ name as the variable name.

#### High-Throughput (bad)

It is not recommended to let a parameter be both input and output. Usually it is a better plan to let the parameter be input, and not write to it, and to return new output. A parameter’s being ‘througput’ should be an exception, rather than the rule. The reason is that it is often confusing to a programmer calling your method. You usually do not expect the data you pass to the method to get deformed.

#### Keep the Names Consistent

Use the same name for something everywhere. For instance if the business domain contains one name, do not reinvent a new name for it. Use the same name everywhere in class names, property names and variable names. Do not come up with a new name for things half way the code.

Here is a code example with a name that changes out of the blue:

int id = 1234;

string sourcePageName = PageService.GetPageName(id);

string destPageName = ConvertPageName(sourcePageName);

private string ConvertPageName(string sourceTitle)

{

// …

}

PageName all of a sudden turned into Title. This confuses everybody. You might not think there is much of a difference in meaning, but for all I know it can be a totally different property and what could also be the case is that someone was mistaken in taking the wrong property. Keep it consistent.

#### Many Properties (generally bad)

It is not a good plan to substitute parameter passing with the use of properties or fields.

It is better to let methods use their own parameters and return values, than to let many methods simply control the same set of properties.

The reason for this is that preferring properties over parameters, you have less control over where the data comes from and goes to. You have no idea what the output of a method actually is, because it may change any of the properties.

#### Methods Instead of Parameters (good)

Sometimes you see a lot of parameters in a method, that require (a lot of) if’s inside the method. This makes the use of the method harder, and the implementation too. Sometimes the solution is simple: make a separate method for every option. This can make the implementation so much simpler and the use of it too. You might thing this is less flexible, because then you have to know in advance which method to call and you cannot just call one method for different situations. But here’s a secret: most of the times that is not a problem. Most of the times the programmer really only needs that one thing in that part of the code, not a single method full of feature switches.

<TODO: A good example would help.>

#### Pass on and Assign (generally bad)

Polluting a whole call stack of methods with variables only used to assign it to an object, that may just as well be assigned much higher in de call stack.

#### Returning a Parameter (bad)

Do not return a parameter again. If you write directly to a passed object, there is not need to return it again. So this is wrong:

A x = Bla(x);

private A Bla(A x)

{

x.Y = 10;

return x;

}

Do this instead:

Bla(x);

private void Bla(A x)

{

x.Y = 10;

}

Returning a parameter suggests that new output is created and the parameter is not written to, while neither is true: the input and output are the same object and the input object is changed. This is very unintuitive.

#### Temporary Variables (good)

To clarify yourself in code, and to make expressions better readable. Use temporary variables just to give things a name in between.

For instance this:

bool isSameControllerAndAction = string.Equals(names.ControllerName, GetControllerName()) &&

string.Equals(names.ActionName, sourceActionName);

if (isSameControllerAndAction)

{

return View(names.ViewName, viewModel);

}

Is more readable than this:

if (string.Equals(names.ControllerName, GetControllerName()) &&

string.Equals(names.ActionName, sourceActionName);)

{

return View(names.ViewName, viewModel);

}

Because it clarifies your reasoning to the next programmer who reads it.

The problem with programming is not writing code, it is reading it.

Most of the time it is a good idea to first put a returned value in a variable before returning it, because this makes debugging easier, because you can inspect the value before returning it, instead of only being able to see the value several frames up the call stack to conclude that a must deeper, now out of sight method did not do its job. Then you are already not in the piece of code you were trying to debug anymore. From a performance point of view it does not matter, because if compiled for release, the compiler will optimize out such temporary variables:

public bool MustExecute(MyClass myParameter)

{

bool **mustExecute** = myParameter.IsSpecial || myParameter.Items.Count > 3;

return mustDoIt;

}

It is not black and white when to use temporary variables just for the sake of giving something a name. It is an art to make your code as readable as possible for the next person.

Do not worry about the performance implications of the extra variable. If a temporary variable is immediately used after it is assigned, it will be optimized by the compiler.

#### Unused Parameter (bad)

Remove parameters from methods if they are no longer used.

#### Variable not Declared Where it is Used (bad)

For instance: when a variable is used in one place and declare in a totally different place, you might want to move the variable closer to where it is used. This may apply to local variables. This may also apply to using local variables instead of fields.

#### Variables that Change Meaning (bad)

When you use a variable for one thing and later overwrite it with semantically something else, it can be confusing to someone reading your code. Consider using a second variable instead.

### Method Bodies

<TODO: Write an intro on the kind of topics you will find here.>

#### Auto-Instatiation

Sometimes auto-instatiation on first use can be replaced by initializing a field in a constructor or type initializer. This performs better becasue it prevents the auto-instatiation ‘if’ and might make the field only initialize once in the lifetime of the app domain.

<TODO: Auto-instantiation variations: benefits and downsides.>

#### Constructor Calls an Overridable (bad)

Calling an overridable member in a base constructor breaks inheritance principles. It creates a chicken and egg problem. Fields in the derived class need to be initialized before running a method, but the field can only be initialized after the base constructor went off, which runs the method! See the following code:

abstract class MyBase

{

public MyBase()

{

Execute();

}

protected abstract Execute();

}

class MyDerivedClass()

{

private int \_value;

public MyDerivedClass(int value)

{

\_value = value;

}

protected override Execute()

{

// PROBLEM: \_value is not initialized yet!

}

}

The solution is to make Execute() public and insist that that it is called explicitly in the code that creates the instance. Ofcourse if the method is not overridable it would be no problem, and if the method was not called in the base constructor it would be no problem, but calling an overridable member from the base class’s constructor could mean trouble.

<TODO: Update the remark below. Another solution is to actually do all ‘the work’ in the constructor, instead of having a separate Execute method, which you could also document in the alternatives above.>

(Validation framework uses this anti-pattern however, because there is too much danger that someone forgets to call Execute. It uses a trick to be able to initialize the members anyway, but it is quite dirty.)

#### Cross-Referencing (generally bad)

<TODO: Practices: Cross-referencing prevention. Do not pass 2 arrays and process them side-by-side, but look for a 'singular form' to process and pass along a tuple. Even better: look for something you can execute onto each tuple element separately.>

#### Empty If-Block (generally bad)

Do not do this:

if (condition)

{

**<<No code>>**

}

else

{

<<Some code>>

}

It looks like you forgot to write code. It looks weird. Either use negation:

if (!condition)

{

// (Some code)

}

Or do an early return:

if (condition)

{

return;

}

// (Some code)

#### Foreach with i

<TODO: Describe: Patterns / code style: for int i met vervolgens het item direct eerst in een variabele.>

#### Last Loop Item

<TODO: Describe different ways of handling the last array item that you can think of, in cases where the last loop item needs to be handled a little bit different from the others.>

#### Method too Long / Class too Long

If a method is long, for instance 25+ code lines, consider if it should be split up into multiple methods.

If a class is long, for instance 800+ code lines, consider if it should be split up into multiple classes.

In both cases this usually means that the method or class has too many responsibilities.

This code smell can also apply to other code than C# classes and methods.

#### Nested Loops (sometimes bad)

This is a nested loop:

foreach (var x in list1)

{

foreach (var y in list2)

{

// ...

}

}

Nested loops usually come with a performance penalty, because compared to a single loop with *n* iterations it might have *n2* iterations. It is not always wrong to have a loop in a loop, but you are only comparing two lists, using a hashset or dictionary might be a better solution, changing the *n2* problem back to a *2n* problem:

Dictionary<int, X> dictionary = list1.ToDictionary(x => x.ID);

foreach (var y in list2)

{

var x = dictionary[y.ID];

// ...

}

#### Nesting too Deep

Too much nesting in code can be confusing. It can be prevented by splitting the code up into multiple methods. It can also be prevented by using early returns. So instead of:

if (condition)

{

// A lot of code…

// …

// …

// …

}

else

{

validationMessages = …;

}

You could do the following:

if (!condition)

{

validationMessages = …;

return;

}

// A lot of code

// …

// …

// …

This keeps cause and effect closer together, making alternative flows in code less confusing.

#### Toilet-Role Principle

Someone reading you code will look at it through the hole of a toilet-role, seeing only a small piece of code at a time. This means smaller pieces of code must make sense on their own. Someone maintaining or correcting your code should not first need to understand 1000’nds of lines of code before being able to correct a minor problem. There are many coding style tricks and design techniques to support this goal, that are talked about in this documentation.

### Strategy

<TODO: Write an intro on the kind of topics you will find here.>

#### Abstract / Concrete

Abstract means that instead of referring to specific items in a set, you refer to a set of items by stating what they have in common. Concrete means talking about a very specific item in the a set.

In software programming abstraction means you can generalize multiple problems and offer a single solution for it.

<TODO: Describe this well: There is a second meaning of abstract, that also has a place is software development: leaving out details. In software development in particular: making something out of smaller building blocks and hide the details behind a simpler view on it.>

It is an art to pick when to abstract problems or when to handle a concrete problem. Both have their benefits. In one case abstraction may prevent complexity, while in another case being specific prevents complexity.

A concrete problem is easier to work out, better to understand, and can tap into specific requirements. On the other hand, abstracting a problem makes you able to write code once and apply it to many different situations. It is harder to write, but might be more maintainable, because it is much less code.

#### Anti-programming

It seems more efficient to reuse many third party software components instead of programming them yourself.

However, your own code might actually be better than that of the third party component. Third party components often come with overhead and bugs and problems with integration so that a custom solution might actually be more efficient.

The choice for a third party component might be related to your own inability to program it. That is why it is anti-programming.

You always start out as an anti-programmer. However, as you progress, you might start getting better.

Do not fall into the trap of thinking that using a third party component is always more efficient than programming it yourself.

#### Asymmetry (bad)

When several pieces of code that do similar things are differently structured, this is an indication that you should make these pieces of code consistent. If these pieces of code do not have a similar structure, you should have a good reason for it and understand this reason and be able to explain it.

Even though this may seem a vague point, symmetry in code is very important for good software design.

#### Bug Solving

To solve a bug, first reproduce it.

<TODO: Make more extensive description.>

#### Bottom-Up and Top-Down

Bottom-up design means you first design the lower layers of a system, for instance the data model and gradually work your way up to the front-end. You can also say bottom-up design is starting with the smaller parts and working your way up to creating bigger and bigger parts out of it.

Top-down design means you first desing the higher layers of the system, for instance the front-end and gradually work your way down to the data model the little details. You can also say top-down design is starting to think about the bigger parts first and then gradually working out smaller and smaller details.

No method is best. They are simply two different strategies to attack a problem.

#### Cartesian Product of Features Problem

Say you have some behaviors that you want a class to either have or not have. What if you want some derived classes that either have or do not have that feature in it. Then you would get as many derived classes as 2 to the power of the number of features. If you have 4 features, you would need 24 = 16 derived classes, with each of the features either turned on or turned off. In cases like this it is hard to come up with a good inheritance structure, because neither feature builds on top of eachother. You could make class variations WithFeature1, WithoutFeature1, WithFeature2WithFeature1, WithFeature2WithoutFeature1. All very awkward. Arbitrarily Feature1 was picked to be more basic than Feature2. Also: you would have to repeat the code of Feature2 in two derived classes! Another alternative is also not so good: building a base class that simply has all features in it and derived classes having the feature either turned on or off. This would be called the ‘god base class’ anti-pattern. It would break the way you work with base classes, since base classes should be more basic with less features in it than derived classes; base classes should not have more features than derived classes.

It does not just apply to turning features on and off. If you have 4 variation on a feature and you want to combine it with one out of 4 variations of another feature, then you have 42 base classes and which feature will be in the deeper base class? It is the same situation as the problem as described above.

This is a weaknesses of inheritance, that makes it so that inheritance should not always be your first choice in constructs to solve your problem.

The Inheritance-Helper pattern may solve some of the issues.

#### Chicken and Egg

<TODO: Describe.>

#### Consistent Stupidity

Too much effort into making code consistent can result in nonsensical code when looking at the individual cases. In a worst case scenario it even results in code that works incorrectly.

It is better for each piece of code to make as much sense as possible individually.

“Consistent stupidity is still stupidity.”

#### Core of the Problem

<TODO: Write text. Incorporate:

Solve the root of the problem. Do not work around a problem, because it will bite you in the ass later, very soon.>

#### Delete + Insert != Update

If you want to update a set of records, you can sometimes get away with deleting all of them and then inserting the new ones again, but you have to realize that this is generally bad practice. You are better off checking if the item exists and then choose whether to insert or update the item based on that. The reason for this is that often the existing items are linked to by other items. If you bluntly remove them, those links will be corrupted. Even when items do not seem to have any links to them, it is still a better idea to check for existence, then insert or update, because of various reasons:

* It performs better if those statements will end up executed onto a database.
* You might link to those entities in the future, which makes your code prepared for that, just by following best practice.
* The entity might not be linked to by other entities, but the entity’s ID may very well be present in URL’s someone might send to someone in an e-mail.
* Also there could be links to an entity in the UI, even though there are not links to the entity from other entities.
* The entity might not be linked to through a foreign key but could be linked to externally by a loosely linked key. (It can be said that it is very easy to overlook that there are links to an entity.)

Hopefully this will give you an idea of how soon you run into problems if you pretend an update equals a deletion + an insertion and make you think twice and do it another way.

See also: Patterns, TryGet-Insert-Update.

#### Distortion (usually bad)

When you diverge from a pattern, you are probably not using it right. Find a way to keep the use of a pattern clean. It is an indication that your separation of concerns is not right or another design mistake. Perhaps you are using the wrong pattern, perhaps you are putting the responsibility for something in the wrong spot.

#### Dot It Right, Or Don’t Do It At All

This is a phrase that can help you prevent a mess of half-baked things that do not work well on their own, let along work well together. This strategy can also be applied to whole features, but also separate classes and methods. If you do not have the time to program it right, you have several options. For instance postpone until you do have time or find an alternative that offers limited capabilities, but still does what it says on the tin. Still better than half-baked stuff. Do not litter the code with all sorts of stuff that does not work.

#### Double Stitch

‘A double stitch holds better.’

Combatting a problem by implementing security against it at multiple levels. Sometimes this protects against a problem better, but on the other hand, you introduce spread responsibility and potentially code repetition.

Here is an example. Say there is a process that writes away data. The process is in principle responsible for writing away correct data, but the validation part of the architecture may guard the overall rules. This means the intricacies around the correct data is noticable in both the process and the validation, so if one of these subsystems contains a programming error, the other subsystem acts as a fail safe.

If the correctness of data is described by trivial rules, you might not say there is a spread responsibility and rather call it a double-stitched solution. But if writing away correct data and checking the correctness of that data is very intricate, you may be repeating significant logic, which does lead to spread responsibility and code repetition. So even though that solution is safer because of double-stitchedness, it is unsafe due to excessive code repetition. This goes to show that it all depends on the situation what the best strategy is.

#### First Try Specific, Then Try Generic

It is often a good solution to design something generic, rather than something that only works for one specific situation. But sometimes it is hard to do this. A strategy can be that when you have the feeling a generic solution is appropriate, but you cannot figure it out, to first develop a specific solution, and then refactor it to become more general.

#### Fluff

A lot of code that does not add much functionality is an indication that a more elegant solution might be possible.

#### Ghost Hunt

Not so much programming practice, rather than a practical concern. Seeing problems where they do not exist or investigating a potential problem that might not be a problem at all. Early detection of seeing problems where there are none can prevent such excessive time spenditure of time.

#### GNUID (bad)

An anti-pattern where a GUID (Globally Unique Identifier) is used as an identifier, but is not globally unique: each occurrence of the same thing, for instance spread over different databases, will have a different ID, even though a Globally Unique Identifier has in its name that it should be the global identifier. This can easily result in programming errors, since one might make assumptions, that an object can be referenced by the same ID always, no matter what copy you are working on.

#### Hard-Coding and Soft-Coding

<TODO: Describe.>

#### Hit F5 and See

Hitting F5 and getting rid of symptoms. This is a strategy, where rather than trying to solve all edge cases in your head or on paper, you start debugging the code and tack the problems as you go.

#### Ideal Solution

It’s good to have the ideal situation in your head and then come up with a less than perfect alternative that approaches that ideal more or less.

#### Inheritance not Always Good

<TODO: Point to other pieces of this document that explains this point and perhaps add a few more arguments to it.>

#### It Works, Doesn’t It? (‘Maar het werkt toch?’) (bad)

This is the false conclusion that when the output is OK, it must mean that the program is coded well. This is often paired with the argument that additional coding work is a waste of time.

A senior programmer might tell a junior programmer to change his code, and a junior programmer might think it is a whole lot of unnecessary work, because his program already worked.

Things that could still be wrong with the program are for instance that:

* Poor performance
* Crashes in exceptional cases
* Corrupts data in exceptional cases
* Code not easy to understand. So that another programmer does not easily understand.
* Poor maintainability so that a change takes ages
* or possibly means it has to be rewritten completely
* or a change is error-prone.
* Not adapted to contextual changes such as culture or when variables change. They may be hard-codedly interwoven into the code.
* Assumptions that databases will be available when they are not.
* Security leaks
* Crashes when services are offline.
* Does not give validation or exception messages when something goes wrong.
* Does not integrate well with other technologies.
* Etcetera.

These are all examples of what could still be wrong with the code if a program ‘seems to work’.

Another false argument against doing all this work is the claim that these are just unimportant details.

#### Least Possible Effort Strategy (bad)

<TODO: Describe. Something like laziness may in the end cost you more, then it is not efficiency.>

#### Liskov-Substution Principle (SOLID)

<TODO: Describe my take and my solutions on it.>

#### Unused Functions (bad)

<TODO: Describe:

* (Dutch) Ik heb de policy om niet gebruikte code weg te halen en pas als het echt opgelost moet worden opnieuw te bouwen, omdat de situatie dan weer dusdanig veranderd is en ideeën veranderd zijn, dat er toch niets over blijft van de oude oplossing, die dan in de tussentijd alleen maar in de weg staat en onderhoudslast tot gevolg heeft.
* It also never gets tested.
* Go into how the people might be afraid to throw away something valuable and that’s why they keep unused stuff lying around. But counterintuitively it is more economic to just throw it away and rebuild it later.>

#### Open/Closed Principe (SOLID)

“Open for extension, closed for modification.” is one of the SOLID software design principles.

This is my personal take on it, and I am very sceptical about this principle. And I am not alone. I think it is outdated, like the waterfall method, or only works in theory, but is not practical at all. It sounds like people that believe in this principle are blindly following some authority, without having an opinion of their own.

If you interpret this litterly it says that you cannot ever change code, which is something I am going to completely have to disagree with. That sounds something theoretical there will never work in practice and in its contains the assumption that code that is written, can be made perfect before moving on to something else. Or a mess of an inheritance hierarchy, or protected modifiers that will break encapsulation, just because someone is afraid to change his code. I use design principles so that my code is easy to change to new situations, so that interpretation does not fly with me.

But if interpreted more loosely, it could be something I can agree with:

* That you should not be required to change code, in order to use it.
* Or that a new version of a component should be compatible with existing software, that uses the component. It sounds to me like something you can work around if all of that client code is your own. And backward compatibility can be achieved with other techniques than not changing code.

There is much debate about how to interpret this design principle. It is like people try to give new explanations to this principle, because the original idea does not work in practice.

#### Open Ends (bad)

Or rather: not solving open ends is usually bad and will meet up with you in the near future.

#### Power of Abstraction / Power of Generalization

When you are able to generalize multiple problems into a single solution, you can code something once and solve multiple problems at the same time.

The other side of it is, that it is difficult to abstract multiple problems into a more general problem. Sometimes it is also difficult to understand the solution, because it requires the same abstract thinking.

However, by doing it you can save a lot of work and complexity.

#### Quick and Dirty / Dirty (usually bad)

<TODO: Describe.>

#### Readable, Writable and Rewritable Code

<TODO: Use this phrase: Code should be about expressing your intentions as much as instruct the computer what to do. >

<TODO: Incorporate this phrase: “the difficult part of programming is not writing code, but reading it.”>

Code should be written so clearly that it is easy to read, which also makes it easy to adapt. It is more important that code is easy to read than easy to write. It is not the startup cost, but the maintenance cost that will kill you. Following good design principles, such as loose coupling, good naming, can make code adaptable. You should write code that you should not be afraid to change. You should not be afraid to change code or refactor. If you are afraid to change the code, then it might be poorly written and a digital time bomb. Time to change that. Or if you are afraid to change code, then there might be something wrong with the testing phase of your software development lifecycle, or lack thereof?

#### Subtractive and Additive

Also called ‘inclusive’ or ‘exclusive’.

Subtractive or additive can be a strategy in programming. Subtractive starts with everything and then you start excluding things. Additive starts with nothing and then you start adding things.

An example is security. It is often better to use an additive approach and start with no user rights at all and gradually add rights.

Another example might be storing object structures. You might create a storage mechanism that stores everything unless you exclude something, or you might create a storage mechanism that stores nothing unless you explicitly specify it is included.

#### Testing

Some bad practices regarding testing:

* Not testing something
* No unit testing if favorable.

#### Too Difficult / Disproportional Effort

‘Don’t be too hard on yourself’ principle / ‘It can’t be that hard’ principle / ‘It is not allowed to be hard’.

Allow yourself to admit, that implementing it a certain way is just going to be too difficult (for you). Look at it another way: if it is that difficult, perhaps there is a simpler solution, that you have overlooked. If a potential solution to a problem takes a lot of effort, be it noticed up front, or noticed while trying to implement, take a step back and look for simpler solutions. This may help you keep an open mind for other solutions. Do not let this be an excuse for laziness. Just keep an open mind.

Those simpler solutions may come with limitations from a functional point of view, or require trade offs in other areas, but it may be worth the time you save. Do keep in mind that you do not just create more work in the future. If doing it ‘wrong’ now will give you an overload of work later, it does not fall into the category of ‘less effort’ or ‘simpler solution’ at all. That time in the future where it becomes a problem, is nearer than you think. It is always a gray area.

If things become difficult to implement, think back to the core of the problem and that if the problem sounds simple, the solution might be too. (Large gray area.)

When you code keeps producing errors when you make a change, this could be an indication that your solution is too difficulr.

#### Tooleritis

Too many tools / separate little programs to fix little things, rather than having a coherent system with all the needed capabilities. A lot of single-run, sometimes-run processes, started separately in separate little console and WinForms apps… while one coherent management application might be better. Tooleritis can also be the result of the system’s having too many ifs, ands and buts.

#### Trade-Offs

Every technique in software development has pros and cons. It is the job of the software designer to weigh off all the pros and cons of every possible design choice and come up with a balance best suited to the situations, that will make us run into the least problems in the future.

A striking example is the principe of generalization which is good, and the principe of low coupling which is good.

There are cases where high coupling is normal. For instance in case of base classes, combinator classes, framework classes, simple types and canonical models, a high degree of coupling with these types can be expected.

Basically when you generalize and make something very reusable, you can automatically expect a high degree of coupling with it, because it is reused so often. That is another reason why generalized solutions should be of such high quality.

There are techniques that in general are bad, and techniques that in general are good, but there also good principles that contradict eachother, that need to be weighed off in every design decision.

The danger of such a large gray area, is that people thing they can just do whatever. But you should not do just whatever. You should learn the pros and cons of things and do a careful weigh-off every time. However, some things are generally bad and some things are generally good.

#### Whirlpool Anti-Pattern / Inappropriate Conversions

The architecture contains multiple layers that require converting one type to another, for instance converting a view model to an entity. However, additional conversions such as converting one type of view model to another type of view model are not recommended.

<TODO: Describe that it is also called the Whirlpool anti-pattern. Related to Inappropriate conversions. It is when data get converted in one form to another to another to another with very little need, not even for abstraction layers. You could consider moving more of the conversion logic that is spread into a single place instead and refactor away some of the conversions. You could also consider that instead of converting from source to dest and then reprocessing dest and then reprocessing dest, you just convert source to multiple dest items, not relying on intermediate data transformations.>

### SOLID

<TODO: Move all comparisons to SOLID to here, because I want to cover it once centrally.>

<TODO: write my take on it. Admit that it might be a single sided view and that you are open to different opinions. It will show you know them and thought about them.?

<TODO: Interface seggregation principle from SOLID.>

## Service Architecture

What has been described so far is the *application architecture*. A second part of the software architecture is the *service architecture*, which is mainly about linking systems together. This section is an addition to the documentation with regards to the service architecture. Currently the services are programmed using WCF.

### The ESB Concept

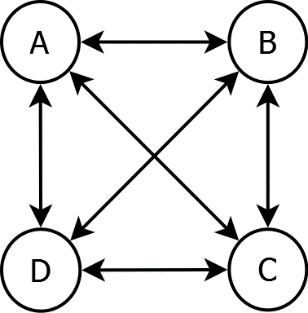
ESB stands for Enterprise Service Bus, which is a system for exchanging data between different systems of different organizations in different formats with different protocols. Central components are used to make integration between these systems more manageable. One important concept is the canonical model.

### Canonical Model

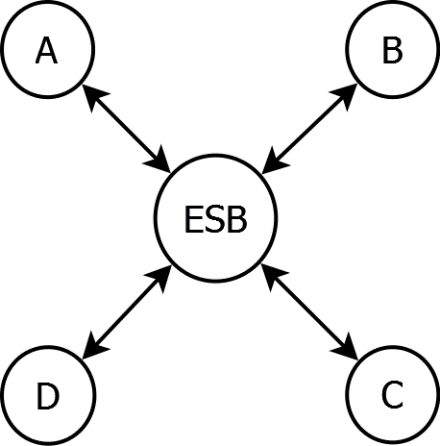
The canonical model helps us exchange data between systems. Data can be retrieved from multiple systems and is converted to a canonical form, so that the same code may be reused for data that comes from various systems. The canonical model should be as pure and general as possible, so indeed information of any system can fit into it with very little modification.

### Less Integration Code

Say you have 4 systems: A, B, C and D and you want to connect all 4 of them together. Theoretically you would have to write 12 different message conversion as you can see from the arrows in the diagram below:

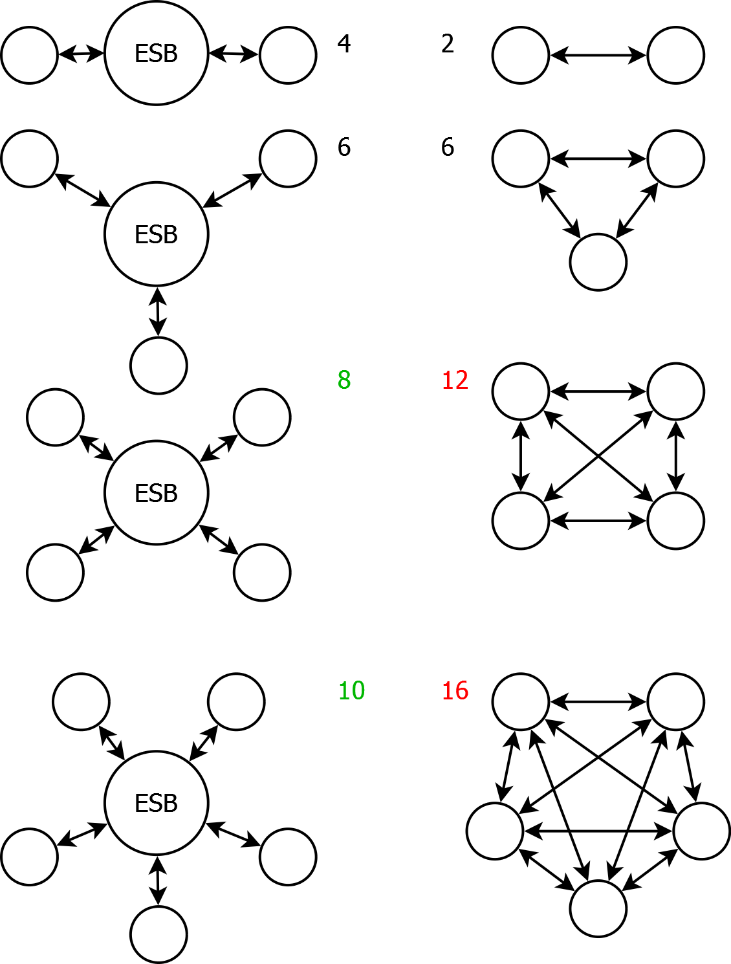


By connecting a system to the ESB, instead of connecting individual systems together, you have to implement only 8 different message conversions as you can see from the arrows below:



You just saved yourself 33% of the work!

With every system you add to your ESB it gets better as you can see from the numbers below that indicate the amount of message conversions.



The first integration between 2 systems you program using your ESB you actually program more message conversions, but with the next system it is already a tie between ESB and no ESB. The 4th integration you introduce you will have saved 33% of the overall work.

It gets better with each system you introduce in your ESB. When messages from a system are converted to and from the canonical model, you can automatically connect it to all the other systems.

### Clearer Integration Code

But it gets better. You save yourself even more work. The conversion code from message to canonical model is often easier than converting from one system’s format to the other system’s format, because instead of converting from one quirky format to another quirky format, which is really difficult to do, you convert from one quirky format to a very clear format, which is much easier to program.

### In Practice

In practice not every system sends every type of message back and forth to every other system. And sometimes the messaging is not bidirectional but one way only. But the benefits of the ESB still hold and you will link systems together with less code and less effort than custom programming every integration between two systems.

### Standard ESB vs Custom ESB

There are standard Enterprise Service Bus software packages available. Yet, we choose to build a custom one ourselves. The concepts are not that hard to implement. And generic ESB’s are really complex and have a steep learning curve, require training, specialists. This all while you are going to have to custom program much of the message conversion code yourself anyway, and design your own canonical model, which is basically all of the work. Therefore we build it ourselves.

### ESB Model

On top of a canonical model, we need more facilities. The ESB model will offer a model for administrating connection settings and register enterprises that can log in to our system to get access to our services.

Next will be listed the main entities of this model.

#### Enterprises

Every enterprise involved in our service architecture is registered in out ESB database. Some of these enterprises will actually log into our system. Those will get an associated User entity with (encrypted) credentials stored in it.

#### ConnectionTypes

Every type of connection between systems is registered in a table of ConnectionTypes. Each ConnectionType is a very specific way of integrating with a system, with a specific messaging protocol, message format and implementation.

#### Connections

Every individual connection between two parties is registered in the Connection table with the connection settings stored with it. Each connection has an associated ConnectionType that indicates what type of integration it is. Note that some connections are not between parties but involve only one party. Connections do not have to be complete messaging implementations. Sometimes they are simply database connection settings or even the path of a network folder.

#### Keys

Often systems have different identifiers for e.g. orders or other objects. There is often a need to map a reference number from one system to the reference number of another system. The ESB model has entities and logic to manage these key mappings.

#### Transmissions

Optionally you can log the transferred messages that went over a connection. Do note that logging all messages can significantly impact performance and storage requirements so use it sparsely.

### Service Implementations

The implementation of a service involves mostly message transformation and transmission. Data is received through some communication protocol, the message format is parsed and then converted to a canonical model. Conversely, canonical models are converted back to a specific message format and the sent over a communication protocol.

### Multi-Dispatch

The content of a canonical model can determine what service to send it to. For instance, one canonical Order might have to be sent to one supplier using their own specific integration protocol, another order might simply be e-mailed to the supplier. The service architecture enables you to retrieve a message from one system, for instance an order, and then send that message to an arbitrary other system. That is part of the power of the canonical model, where multiple systems’ messages being converted to canonical model, enables all those systems to communicate with eachother.

### Namespaces

These namespaces use a hypothetical Ordering system as an example.

|  |  |
| --- | --- |
| JJ.Services | Root namespace for web services / WCF services |
| JJ.LocalServices | Root namespace for windows services. (Not part of the service architecture, but this is where that other type of service goes.) |
| JJ.Data.Canonical | Where are canonical entity models are defined. |
| JJ.Data.Esb | Entity model that stores Enterprises, Users, ConnectionTypes, Connections, etc. Basically the configuration settings of the architecture. |
| JJ.Data.Esb.NHibernate | Stores the Esb entity model using NHibernate. |
| JJ.Data.Esb.SqlClient | SQL queries for working with the stored Esb entity model. |
| JJ.Business.Canonical | Some shared logic that operates on canonical models. |
| JJ.Business.Esb | Business logic for managing the Esb model. |
| JJ.Services.Ordering.Interface | Defines interfaces (the C# kind) that abstract the way messages are sent between different ordering system. These interfaces use the canonical models. |
| JJ.Services.Ordering.Dispatcher | Makes sure messages (orders, price updates) are received from and sent to the right system depending on message content. |
| JJ.Services.Ordering.Email | A specific implementation of an ordering interface, behind which we send the order by e-mail. |
| JJ.Services.Ordering.SuperAwesomeProtocol | A specific implementation of an ordering interface, behind which we implement the hypothetical ‘super awesome protocol’ for sending orders. |
| JJ.Services.Ordering.Wcf | A WCF service that allows you to communicate with the multi-dispatch ordering system. |
| JJ.Services.Ordering.Wcf.Interface | Defines the interface of the WCF service that allows you to communicate with the multi-dispatch ordering system. This service interface can is used by both service and client. |
| JJ.Services.Ordering.Wcf.Client | Allows code to connect to the WCF service using the strongly typed service interface. |
| JJ.Services.Ordering.JsonRest | Exposes the multi-dispatch ordering service using the Json and Rest protocols. |
| JJ.Services.Ordering.WebApi | There is no reason Web API should not be involved in this service architecture, in fact, the idea of WCF being the default for service, might not be a very long-lived. |
| JJ.Presentation.Shop.AppService.Wcf | A special kind of service is an app service, that exposes presentation logic instead of business logic and returns ViewModels. |

### Service-Related Patterns

#### Facade

An interface behind which a lot of other interfaces and classes are used, with the goal of simplifying working with these systems.

This concept is used in this architecture to give a service interface an even simpler interface than the underlying business logic has. It may hide interactions with multiple systems, and hide infrastructural setup.

#### Hidden Infrastructure

Not so much a pattern, but a difference in handling infrastructure setup between the application architecture and the service architecture. Iin the application architecture the infrastructural context is determined by the top-level project and passed down to the deeper layers as for instance repository interfaces or interfaces on security, while in the service architecture the infrastructural context is determined by the bottom-level project. At least in case of multi-dispatch this seems necessary. A bottom-level project, for instance JJ.Services.Ordering.Email does not expose that there will be smpt server setup. You cannot see that from the constructor or interface at all. The service will handle all that internally.

<TODO: Code example.>

### TODO

<TODO: Service Architecture: Three-Stage KeyMapping vs. Custom KeyMapping is something to write about in Architecture Details. It explains why you do not need an even more generic Key mapping, for intance:

* CustomerOrderNumber
* SupplierOrderNumber
* InternalID

Cannot remember how and why it worked exactly.>

## Database Conventions

### Developing a Database

Developing a database generally involves the following steps:

* Create tables
* Add primary keys
  + Create ID column
  + Set primary key
  + Set Identity Yes
* Add columns
  + Give them the right data types (see further down)
* Most columns not nullable
  + Limit the nullable columns as much as possible
  + But keep them nullable where it functionally makes sense.
* Make foreign keys on columns that link to other tables
* Add indexes on foreign keys columns
* Add unique indexes
  + However, sometimes ORM’s will trip over unique keys at which we promptly remove the unique constraint.
  + Note that you do not need an additional index when there is already a unique constriant whose first column is the column you would like to index.
* Add indexes to search columns and alternative keys
* Add indexes for problem queries

Keep in mind to limit the use of ‘exotic’ data types.

For instance: if a number would fit in a tinyint, use an ‘int’ anyway, because this saves the system a lot of casting, and a 32-bit number is better for memory / disk alignment, which is better for performance. Chances are, a system will reserve 32 bit for a 8-bit number anyway to accomplish this memory alignment. But this is just an example.

Here are the recommended data types:

* bit
* int
* decimal
* float
* real
* datetime
* nvarchar
* uniqueidentifier
* varbinary

Only if you need a bigger range:

* bigint
* datetime2

### Naming Conventions

|  |  |
| --- | --- |
| **Object Type** | **Example Name** |
| Database name | ShopDB |
| Tables | MyTable |
| Columns | MyColumn |
| ID column | ThingID |
| Indexes | IX\_MyTable\_MyColumn |
| Primary keys | PK\_MyTable |
| Foreign keys | FK\_MyTable\_OtherTable  Or when there are multiple relations between the same tables:  FK\_MyTable\_<*OtherTable\_ColumnName\_MinusID*>  FK\_MyTable\_ThingA  FK\_MyTable\_ThingB |
| Unique keys | When not many columns:  IX\_MyTable\_MyColumn\_Unique  When many columns and only one constraint in the table:  IX\_MyTable\_Unique |
| Stored procedures | SP\_DoSomething / spDoSomething |
| Functions | FN\_DoSomething / fnDoSomething |
| Triggers | TR\_MyTable\_OnInsert / trgMyTable\_Insert / … |

* Avoid using keywords as column names. Think of a different name instead.
* ‘Index’ is an SQL Server keyword! Avoid that name. Think of another one. IndexNumber or SortOrder.

### Rules

Do not use the following object types, because these things are managed in .NET:

* Views
* Stored Procedures
* Functions
* Triggers
* Defaults
* Check constraints
* Computed columns
* Cascade rules
* Synonyms
* Assemblies
* Types
* Rules

For new databases, prefer int’s as primary keys over guids, because guids create performance penalties throughout the software stack.

Only use additional guid columns as an alternative key for entities that need to be unique across multiple systems or databases. Do not forget to put an index on the guid column. Prefer surrogate keys rather than complicated composite keys. Prefer auto-incremented ID’s, except for enum-like tables.

<TODO: Mention: Security? Guids can be safe for security. For instance, for smaller underlying entities you could not guess the ID and sneekily change someone elses data, when only the user-ownership of higher objects are checked.>

For development databases use the “DEV\_” prefix, e.g. DEV\_ShopDB.

For test use the prefix “TEST\_” and for acceptance use the prefix “ACC\_”. For production use no prefix at all.

On development databases add the user dev with password dev. For test add the user test with password test. For acceptance you might use specific user names depending on security demands, otherwise add user name acc with password acc. In production databases use the administrator user’s password with the administrator password for databases or create a separate user name for production with a strong password.

### Upgrade Scripts

Database upgrade scripts are managed as follows.

#### Excel Sheet

Each database structure gets an Excel in which all the upgrade SQL scripts are registered.

The Excel sheet and SQL scripts are put in a Visual Studio project to manage them easily.

Always edit the Excel in the dev branch, because Excels cannot be merged.

The name of an SQL file has a specific format:

2014-08-28 040 ShopDB Supplier.Name not null.sql

So it has the format:

{Date} {Number} {DatabaseStructureName} {DatabaseObject}{SubDatabaseObject} {Change}.sql

|  |  |  |
| --- | --- | --- |
|  | **Description** | **Examples** |
| **Date** | Use the format yyyy-mm-dd | 2014-08-28 |
| **Number** | Use 3 digits and count in 10’s so you might insert one in betweeen | 040 |
| **DatabaseStructureName** |  | ShopDB |
| **DatabaseObject** | A table name or index name or other database object name | Supplier  IX\_Supplier\_Name  FK\_Supplier\_Branch |
| **SubDatabaseObject** | Optional. Usually a column name | .Name |
| **Change** | Optional. Usually left out. You can sometimes mention a specific change, but be brief. | not null |

In the Excel, add a column for each database instance for that database structure. There can be different databases with the same structure for different staging areas (dev, test, acc, prod) or a database for different customers or databases running on different servers. Put ‘TRUE’ (or ‘WAAR’ in Dutch) where the upgrade script has been executed. For instance:



Also include a column saying whether you have scripted it at all (for if you are in a hurry and have no time to script it). A release date column is also handy, to get some sense of when things went live.

##### Exceptional Cases

For upgrades that should only be executed on a specific database, put ‘N/A’ (or ‘N.V.T.’ in Dutch) in the appropriate spread sheet cell.

You can also add something to the SQL file name to indicate this:

2015-01-23 010 OrderDB SHOPDB ONLY Order.DeliveryDateTimeUtc.sql

Some things should be done manually and not with SQL. Those actions should also be mentioned in the Excel:

2015-01-23 020 OrderDB OrderID Identity Yes DO MANUALLY

If a script requires that you be extra careful, you can mention this as follows:

2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql CHECK MANUALLY

2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql EXECUTE SEPARATELY

But be sparse with that, because the person running the script might not actually know what it is he is supposed to check and will feel uneasy executing this script since it is obviously so dangerous, while he has no idea why.

##### Summary

This section covered:

* Visual Studio project
* Excel sheet
* SQL script name format
* Upgrades for specific databases and manual upgrades

#### Scripts

The individual upgrade SQL scripts should not contain GO statements. GO is not an SQL keyword, it is a Management Studio command telling it to execute the script up until that point. What must be separated by GO statements in Management Studio must be split up into multiple SQL files in the database upgrade scripts.

Also get rid of any automatically generated SET ANSI\_NULLS ON and SET QUOTED\_IDENTIFIER ON statements. Those are the default behavior anyway, and it just add unnecessary fluff to your scripts. Also: SET ANSI\_NULLS OFF will generate an error in future versions of SQL Server anyway.

The upgrade scripts should be incremental: DO make assumptions about the previous state of the database structure and script a specific change. Do not write scripts like ‘if not exists’ then add, or ‘drop and create table’ scripts, because you may be throwing away data, or execute things on the wrong database. It is better to make a specific change and *not* be tolerant to differences.

DO NOT script changes from Identity Yes to Identity No or the other way around. Changes in the Identity property of a column require recreating the whole database table. If you script it now, executing it onto a database does not only add the Identity Yes property, it will also restore the whole table structure to the state it had at the time you scripted the Identity.

***Summary***

This section covered:

* No GO statements
* Split up into separate files
* Incremental scripts (no ‘if exists’ checks or drop and recreate).
* DO NOT script Identity Yes and Identity No

#### Deployment

To deploy multiple database structure changes you can use the Excel.

Always edit the Excel in the dev branch, because Excels cannot be merged.



You can easily see which scripts are still to be executed onto the database.

After you have executed them, put TRUE in the appropriate spread sheet cells.

You could execute the scripts one by one, but there is a handier, safer way to do it.

With some creative copying and pasting the SQL file names, you can create a composite upgrade script like this:

begin try

print 'Begin transaction.';

begin transaction;

declare @verbose bit = 0;

declare @folder varchar(255) = 'C:\JJ\Install\SqlScripts';

exec spExecuteSqlFile @folder, '2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql', @verbose;

exec spExecuteSqlFile @folder, '2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc not null.sql', @verbose;

--print 'Rolling back transaction.';

--rollback transaction;

print 'Committing transaction.';

commit transaction;

end try

begin catch

print Error\_Message();

print 'Rolling back transaction.';

rollback transaction;

end catch

This safely executes all changes in a single transaction and shows error information if something goes wrong.

This does require you to add the following stored procedure to the database:

create procedure spExecuteSqlFile(@folderPath varchar(255), @fileName varchar(255), @verbose bit = 0) as

begin

set nocount on;

print 'Executing ''' + @fileName + '''';

declare @filePath varchar(255) = @folderPath + '\' + @fileName;

declare @readFileSql varchar(1024) = 'select BulkColumn from openrowset(bulk ''' + @filePath + ''', single\_blob) x;'

declare @temp table (contents varchar(max));

insert into @temp exec (@readFileSql);

declare @sql varchar(max);

set @sql = (select top 1 contents FROM @temp);

-- Remove BOM from UTF-8.

if (LEFT(@sql, 3) = 'ï»¿') set @sql = RIGHT(@sql, LEN(@sql) - 3);

exec (@sql);

if (@verbose = 1) print @sql;

end

##### Summary

This section covered:

* Composite upgrade scripts
* spExecuteSqlScript

#### SqlScripts table

Consider maintaining a list of executed database upgrade SQL scripts in a table, because it happens too often, that someone has put a database somewhere, without administrating the Excel file, no matter how many times you say it.

### C#-Based Migrations

Some data migrations are easier to program using C# than SQL scripts.

Sometimes the the contrast between how easy it is to do in C# or SQL is so large, that the benefits of programming it in C# outweigh the downsides. It could be a factor 20 difference in development time in some cases.

A benefit of SQL scripts is that it always operates on the right intermediate version of the entity model, while C# code always operates on the latest version of the entity model. This means that earlier C#-based migrations might not compile anymore for a newer version of the entity model, and can only work with an older version of the model.

This problem with C#-based migrations can be mitigated in several ways. Here are a few ideas:

|  |  |
| --- | --- |
| Always rerunnable tool | Replace the one-off C# migration by a tool that does something more general, that can operate on any version of the model.  For instance, in a certain project, resaving most data to the database using newer business logic would set a lot of things right in the data and this procedure was rerunnable at any time, regardless of the version of the model. ‘Run the resaver’ would be the description in the list of data migrations to execute. |
| Get specific version, build, get specific version, build | You can let the C#-based migration operate on a specific version of the model by getting the older version of the software from source control, then building it. Each time you have to do a C#-based migration, you can make a separate executable, that operates on a specific version of the code. As soon as a migration does not compile anymore, you can simply outcomment or remove it. |
| Snapshots of entity model | Storing a snapshot of an entity model in a separate project specifically intended for that migration might be a solution. (Not tried out in practice. Might turn out to be very impractical.) |
| Any other ideas | are welcome. |

## Code Style

This section lists trivial coding rules, that should be followed throughout the code.

Coding standards mostly conform to the Microsoft standard described in the following documents:

<http://msdn.microsoft.com/en-us/library/vstudio/ff926074.aspx>

<http://msdn.microsoft.com/en-us/library/aa260844%28v=vs.60%29.aspx>

Use Resharper. Seriously. Finetune it to automatically check your coding style. Use it to keep code clean as write code or change existing code.

### Casing, Punctuation and Spacing

|  |  |
| --- | --- |
| **Rule** | **Example** |
| Properties, methods, class names and events are in pascal case. | MyProperty  MyMethod |
| Local variables and parameters are in camel case. | myLocalVariable  myParameter |
| Fields are in camel case and start with underscore. | \_myField |
| Constants in capitals with underscores in between words. | MY\_CONSTANT |
| No prefixes, such as “strName”. |  |
| Avoid abbreviations. |  |
| For long identifiers, use underscores to separate ‘the pieces’. | Sine\_OperatorCalculator\_VarFrequency\_WithPhaseTracking |
| Type arguments start with the letter T or are just the letter T | T TEntity TViewModel |
| Abbreviations of 2 letters with capitals. | ID |
| Abbreviations of 3 letters or more in pascal case. | Mvc |
| Start interface names with ‘I’. | IMyInterface |
| Partial view names in MVC should begin with an underscore | \_MyPartialView |

|  |  |  |
| --- | --- | --- |
| **Rule** | **Not Recommended** | **Recommended** |
| Keep Visual Studio’s autoformatting enabled and set to its defaults. |  |  |
| No extra enters between braces. | }  } | }  } |
| Put enters between switch cases. | switch (x)  {  case 1:  break;  case 2:  break;  } | switch (x)  {  case 1:  break;  case 2:  break;  } |
| No braces for single-line if-statements. | if (condition) { Bla(); } | if (condition) Bla(); |
| Loops always on multiple lines. | foreach (var x in list) { Bla(); } | foreach (var x in list)  {  Bla();  } |
| Use braces for multi-line if’s and loops. | foreach (var x in list)  Bla();  if (condition)  Bla(); | foreach (var x in list)  {  Bla();  }  if (condition)  {  Bla();  } |
| Put enters between methods. | void Bla()  { }  void Bla2()  { } | void Bla()  { }  void Bla2()  { } |
| Each property at least its own line. | int A { get; set; } int B { get; set; } | int A { get; set; }  int B { get; set; }  int C  {  get { ... }  set { ... }  }  int D  {  get  {  ...  }  set  {  ...  }  } |
| Put enters inside methods between ‘pieces that do something’ (that is vague, but that is the rule). | void Bla()  {  var x = new X();  x.A = 10;  var y = new Y();  y.B = 20;  y.X = x;  Bla2(x, y);  } | void Bla()  {  var x = new X();  x.A = 10;  var y = new Y();  y.B = 20;  y.X = x;  Bla2(x, y);  } |
| Each variable declaration on its own line. | int i, j; | int i;  int j; |
| Avoid ‘tabular form’. It should only rarely be used. This tabular form will often be undone by auto-formatting. It is non-standard, so it is better to get your eyes used to non-tabular form. | public int ID { get; set; }  public bool IsActive { get; set; }  public string Text { get; set; }  public string Answer { get; set; }  public bool IsManual { get; set; } | public int ID { get; set; }  public bool IsActive { get; set; }  public string Text { get; set; }  public string Answer { get; set; }  public bool IsManual { get; set; } |
| Align the elements of linq queries as follows: | var arr = coll.Where(x => x...).  OrderBy(x => x...).ToArray() | var arr = coll.Where(x => x...)  .OrderBy(x => x...)  .ToArray() |
| Use proper indentation | <TODO: Example.> | <TODO: Example.> |
| Generic constraints on next line.  (So they stand out) | class MyGenericClass<T> where T: MyInterface  {  } | class MyGenericClass<T>  where T: MyInterface  {  } |
| For one-liners, but generic constraints on same line instead. | interface IMyInterface  {  void MyMethod(T param)  where T : ISomething  } | interface IMyInterface  {  void MyMethod(T param) where T : ISomething  } |

### Tri vial Rules

|  |  |  |
| --- | --- | --- |
| **Rule** | **Wrong** | **Right** |
| Give each class (or enum) its own file (except nested classes). | - | - |
| Keep members private as much as possible. |  | private void Bla()  { } |
| Keep types internal as much as possible. |  | internal class MyClass  { } |
| Use explicit access modifiers (except for interface members). | int Bla() { ... } | **public** int Bla() { ... } |
| No public fields. Use properties instead. | public int X; | **public** int X **{ get; set; }** |
| Put nested classes at the top of the parent class’s code. | internal class A  {  public int X { get; set; }  private class B  {  }  } | internal class A  {  private class B  {  }  public int X { get; set; }  } |
| Avoid getting information by catching an exception. Prefer getting your information without using exception handling. | bool FileExists(string path)  {  try  {  File.Open(path, ...);  return true;  }  catch (IOException)  {  return false;  }  } | bool FileExists(string path)  {  return File.Exists(path);  } |
| Do not use type arguments that can be inferred. | References**<Child>**(x => x.Child) | References(x => x.Child) |
| Use interface types as variable types when they are present. | **List**<int> list = new List<int>; | **IList**<int> list = new List<int>; |
| Prefer ToArray over ToList. | IList<int> collection = x.**ToList**() | IList<int> collection = x.**ToArray**() |
| Use object initializers for readability. | var x = new X();  x.A = 10;  x.B = 20; | var x = new X  {  A = 10,  B = 20  } |
| Put comment for members in <summary> tags. | // This is the x-coordinate.  int X { get; set; } | **/// <summary>**  **///** This is the x coordinate.  **/// </summary>**  int X { get; set; } |
| Comment in English. | // Dit is een ding. | // This is a thing. |
| Do not write comment that does not add information | **// This is x**  int x; | int x; |
| Avoid compiler directives  Do not use them unless you absolutely cannot run the code on a platform unless you exclude a piece of code. Otherwise use a boolean variable, a configuration setting, different concrete implementations of classes or, anything. | **#if FEATURE\_X\_ENABLED**  **// ...**  **#endif** | if (config.FeatureXEnabled)  {  // ...  } |
| An internal class should not have internal members.  The members are automatically internal if the class is internal. If you have to make the class public, you do not want to have to correct the access modifiers of the methods. | **internal class A**  **{**  **internal void B**  **{**  **}**  **}** | internal class A  {  public void B  {  }  } |
| Default switch case at the bottom. | **switch (x)**  **{**  **default:**  **break;**  **case 0:**  **break;**  **case 1:**  **break;**  **}** | switch (x)  {  case 0:  break;  case 1:  break;  **default:**  **break;**  } |
| Prefer .Value and .HasValue for nullable types. | **int? number;**  **if (number != null) {**  **string message = String.Format(**  **"Number = {0}", number);**  **}** | int? number;  if (number.HasValue) {  string message = String.Format(  "Number = {0}", number.Value);  } |
| Do not leave unused (outcommented) around. If needed, move it to an Archive folder, or Outtakes.txt, but do not bug your coworkers with out-of-use junk lying around. |  |  |
| it is appreciated when a file stream is opened specifying all three aspects FileMode, FileAccess and FileShare explicitly with the most logical and most limiting values appropriate for the particular situation. |  |  |

### Miscellaneous Rules

|  |  |  |
| --- | --- | --- |
| **Description** | **Not Recommended** | **Recommended** |
| Test class names end with ‘Tests’. | [TestClass]  public class **Tests\_**Validator()  {  } | [TestClass]  public class Validator**Tests**()  {  } |
| Test method names start with Test\_ and use a lot of underscores in the name because they will be long, because they will be very specific. | [TestMethod]  public void **Test** ()  {  ...  } | [TestMethod]  public void **Test\_Validator\_NotNullOrEmpty\_NotValid**()  {  ...  } |
| var should be avoided. The variable type should be visible in the code line instead of ‘var’. Exceptions are: | **var** x = y.X; |  |
| * An anonymous type is used. | **X** q = from x in list select **new { A = x.A }**; | **var** q = from x in list select **new { A = x.A }**; |
| * The code line is a ‘new’ statement. | **X** x = new **X**() | **var** x = new **X**() |
| * The code line is a direct cast. | **X** x = (**X**)y; | **var** x = (**X**)y; |
| * The code line is WAAAY too long and unreadable without ‘var’. | foreach (**KeyValuePair<Canonical.ValidationMessage, Tuple<NonPhysicalOrderProductList, Guid>>** entry in dictionary) | foreach (**var** entry in dictionary) |
| * Use var in your **view** code. | <% foreach (**OrderViewModel** order in Model.Orders) %> | <% foreach (**var** order in Model.Orders) %> |
| Handle null and empty string the same way everywhere. |  |  |
| To check if a string is filled use IsNullOrEmpty. | str == null | String.IsNullOrEmpty(str) |
| To equate string use String.Equals. | str == "bla" | String.Equals(str, "bla") |
| Avoid using Activator.CreateInstance. Prefer using the ‘new’ keyword. Using generics you can avoid some of the Activator.CreateInstance calls. A call to Activator.CreateInstance should be rare and the last choice for instantiating an object. | Activator.CreateInstance(typeof(T)) | T = new T() |
| Entity equality checks are better done by ID than by reference comparison, because persistence frameworks do not always provide instance integrity, so code that compares identities is less likely to break. | if (entity1 == entity2) | if (entity1.ID == entity2.ID)  // (Also do null checks if applicable.) |
| The following data types are not CLR-complient and sould be avoided | Unsigned types such as:  uint  ulong  And also:  sbyte | int  long  byte |
| Parameter order:  When passing infrastructure-related parameters to constructors or methods, first list the entities (or loose values), then the persistence related parameters, then the security related ones, then possibly the culture, then other settings. |  | class MyPresenter  {  public MyPresenter(  MyEntity entity,  IMyRepository repository,  IAuthenticator authenticator, string cultureName, int pageSize)  {  ...  }  } |
| No long code lines <TODO: Describe better.> |  |  |
| When evaluating a range in an ‘if’, mention the limits of the range and mention the start of the range first and the end of the range second. | if (x <= 100 && x >= 10)  if (x >= 11 && x <= 99) | if (x >= 10 && x <= 100)  if (x > 10 && x < 100) |

#### Namespace Tips

Avoid using full namespaces in code, because that makes the code line very hard to read:

NOT RECOMMENDED:

JJ.Business.Cms.RepositoryInterfaces.IUserRepository userRepository = PersistenceHelper.CreatCmsRepository< JJ.Business.Cms.RepositoryInterfaces.IUserRepository>(cmsContext);

Using half a namespace is also not great, because when you need to rename a namespace, you will have a lot of manual work:

NOT RECOMMENDED:

Business.Cms.RepositoryInterfaces.IUserRepository userRepository = PersistenceHelper.CreateCmsRepository<Business.Cms.RepositoryInterfaces.IUserRepository>(cmsContext);

Instead, try giving a class a unique name or use aliases:

using IUserRepository\_Cms = JJ.Business.Cms.RepositoryInterfaces.IUserRepository;

...

IUserRepository\_Cms cmsUserRepository = PersistenceHelper.CreateCmsRepository<IUserRepository\_Cms>(cmsContext);

### Member Order

Try giving the members in your code file a logical order, instead of mixing them all up. Suggested possibilities for organizing your members:

|  |  |
| --- | --- |
| Chronological | When one method delegates to another in a particular order, you might order the methods chronologically. |
| By functional aspect | When your code file contains multiple functionalities, you might keep the members with the same function together, and put a comment line above it. |
| By technical aspect | You may choose to keep your fields together, your properties together, your members together or group them by access modifier (e.g. public or private). |
| By layer | When you can identify layers of delegation in your class you might first list the members of layer 1, then the members of layer 2, etc. |

The preferred ordering of members might be chronological if applicable and otherwise by functional aspect, but there are no rights and wrongs here. Pick the one most appropriate for your code.

### Naming

See also: Casing, Punctuation and Spacing.

#### Boolean Names

Use common boolean variable name prefixes and suffixes:

|  |  |  |
| --- | --- | --- |
| **Prefix / Suffix** | **Example** | **Comment** |
| Is… | IsDeleted | This is the most common prefix. |
| Must… | MustDelete |  |
| Can… | CanDelete | Usually indicates what *user* can do. |
| Has… | HasRecords |  |
| Are… | AreEqual | For plural things. |
| Not… | NotNull | A valid prefix, but be careful with negative names for readability’s sake. See ‘Double Negatives’. |
| Include… | IncludeHidden | Even though it is verb, it makes sense for booleans. |
| Exclude… |  | Even though it is verb, it makes sense for booleans. |
| … Exists | FileExists |  |

If it is ugly to put the prefix at the beginning, you can put it in the middle, e.g.: LinesAreCopied instead of AreLinesCopied.

Some boolean names are so common that they do not get any prefixes:

|  |
| --- |
| Visible |
| Enabled |

#### Class Names

Class names usually end with the pattern name or a verb converted to a noun, e.g.:

Converter

Validator

Calculator

And they start with a term out of the domain:

OrderConverter

ProductValidator

PriceCalculator

A more specialized class can get prefixes or suffixes as follows:

OptimizedPriceCalculator

OrderWithPriorityShippingValidator

Or alternatively:

OrderValidatorWithPriorityShipping

Abstract classes get the preferred suffix ‘Base’:

ProductValidatorBase

This is because it is very important to see in code whether something is a base class. Exceptions to the suffix rule can be made if it would otherwise result in less readable code. For instance, base classes in entity models might not look good with the ‘Base’ suffix.

Keep variable names similar to the class names, and end them with the pattern name.

Common ‘last names’ for classes apart form the pattern names are:

|  |  |
| --- | --- |
| Resolver | A class that does lookups that require complex keys or different ways of looking up depending on the situations, fuzzy lookups, etc. |
| Dispatcher | A class that takes a canonical input, and dispatches it by calling different method depending on the input, or sending a message in a different format to a different infrastructural endpoint depending on the input. |
| Invoker | Something that invokes another method, probably based on input or specific conditions. |
| Provider | A class that provides something. It can be useful to have a separate class that provides something if there are many conditions or contextual dependencies involved in retrieving something. A provider can also be used when something has to be retrieved conditionally or if retrieval has to be postponed until later. |
| Asserter | <TODO: Describe> |
|  | Any method verb could become a class name, by turning it into a verby noun, e.g. Convert 🡪 Converter. |

#### Collection Names

Collection names are plural words, e.g.:

Products

Orders

Variable names for amounts of elements in the collection are named:

Count

So avoid using plural words to denote a count and avoid plural words for things other than collections.

#### DateTime Names

A DateTime property should be suffixed with ‘Utc’ or ‘Local’:

StartDateLocal

OrderDateTimeUtc

#### Enum Names

Use the ‘Enum’ suffix for enum types e.g. OrderStatus**Enum**.  
Another acceptable alternative is the suffix ‘Mode’, e.g. Connection**Mode**, but the first choice should be the suffix ‘Enum’.

#### Event Names / Delegate Names

Event names and delegate names, that indicate what just happened have the following form:

Deleted

TransactionCompleted

Event names and delegate names, that indicate what is about to happen have the following form:

Deleting

TransactionCompleting

UI-related event names do not have to follow that rule:

Click

DoubleClick

KeyPress

Delegate names can also have the suffix Callback or Delegate:

ProgressInfoCallback

AddItemDelegate

Sometimes the word ‘On’ is used:

OnSelectedIndexChanged

OnClick

Or the prefix Handle:

HandleMouseDown

Or the suffix Requested, if your event looks like a method name.

RemoveRequested

Pardon the ambiguity, but the naming above can be used for the names of events, but some of them also serve well as names for methods that fire/emulate or otherwise handle the event. The prefix ‘On’ for instance and the prefix ‘Handle’ may very well be used for the methods that actually raise the event. ‘Fire’ and ‘Do’ are also alternatives.

Avoid event names that indicate that it is an event in two different ways. For instance ‘OnDragging’ can be shortened to just ‘Dragging’, because the suffix -ing is already an indication that it is an event. ‘OnMouseUp’ can be shortened to just ‘MouseUp’, because that is an established event name.

#### Method Names

Method names start with verbs, e.g. CreateOrder.

Names for other constructs should not start with a verb.

Common verbs:

|  |  |
| --- | --- |
| **Verb** | **Description** |
| Add | E.g.  List.Add(item)  ListManager.Add(list, item)  In cases such as the last example, it is best to make the list the first parameter. |
| Assert | A method that throws **exceptions** if input is invalid. |
| Calculate |  |
| Clear |  |
| Convert |  |
| ConvertTo |  |
| Create | When a method returns a new object. |
| Delete |  |
| Ensure | Sets up a state if it is not set up yet. If Ensure means throw an exception if a state is not there, then consider using the verb ‘Assert’ instead. |
| Execute |  |
| Generate |  |
| Get |  |
| Invoke |  |
| Parse |  |
| Process |  |
| Remove |  |
| Save |  |
| Set |  |
| Try |  |
| TryGet |  |
| Validate | A method that generates **validation messages** for user-input errors |

#### File-Related Variable Names

Variable names that indicate parts of file paths can easily become ambiguous. Here is a list of names that can be used to disambiguate it all:

|  |  |
| --- | --- |
| **Name** | **Value** |
| FileName | "MyFile.txt" |
| FilePath | "C:\MyFolder\MyFile.txt" |
| FolderPath | "C:\MyFolder" |
| SubFolder | "MyFolder" |
| RelativeFolderPath (sometimes also called ‘SubFolder’ or ‘SubFolderPath’) | "MyFolder\MyFolder2" |
| RelativeFilePath | "MyFolder\MyFile.txt" |
| FileNameWithoutExtension | "MyFile" |
| FileExtension | ".txt" |
| AbsoluteFilePath | "C:\MyFolder\MyFile.txt" |
| AbsoluteFolderPath | "C:\MyFolder" |
| AbsoluteFileName | DOES NOT EXIST |
| FileName**Pattern**, FilePath**Pattern**, etc. | **\***.xml  C:\temp\BLA\_**????**.csv |
| FileName**Format**, FilePath**Format**, etc. | order-**{0}**.txt  orders-**{0:dd-MM-yyyy}**\\*.\* |

**Prefixes and Suffixes**

|  |  |
| --- | --- |
| **Suffix** | **Description** |
| source..  dest… | In code that converts one structure to the other, it is often clear to use the prefixes ‘source’ and ‘dest’ in the variable names to keep track of where data comes from and goes to. |
| existing... | Denotes that something already existed (in the data store) before starting this transaction. |
| new… | Denotes that the object was just newly created. |
| original… | Denotes that this is an original value that was (temporarily) replaced. |
| …WithRelatedEntities  …WithRelatedObjects | Indicates that not only a single object is handled, but the object including the underlying related objects. |
| Versatile… | A class that handles a multitude of types or situations. |
| …With… | When tou make a specialized class that works well for a specific situation, you could use the word ‘With’ in the class name like this:   * CostCalculator * CostWithTaxCalculator |
| ...Polymorphic | Handles a multitude of differrent derived types, possibly each in a different way. |
| …IfNeeded | If something is executed conditionally. This is a nice alternative for the less pretty suffixes ‘Conditionnally’ or a prefix ‘Conditional’, which obscures the name that comes after. |
| …Unsafe | When it lacks e.g. thread-safety or executes unmanaged code, or lacks a lot of checks. |
| …Recursive | (Some people tend to use ‘Recursively’ instead, probably insisting it is better grammer, but Recursive is shorter and not grammatically incorrect either. It is a characteristic, as in ‘Is it *recursive*?’.) |
| To… | For conversion from one to another thing. Usually ‘this’ is source of the conversion, for example:  array.ToHashSet()  Less commonly the ‘To’ prefix is used when the ‘this’ is not the source, for instance:  MyConverter.ToHashSet(object[] array)  The Convert or ConvertTo verbs might be more appropriate there:  MyConverter.ConvertToHashSet(object[] array) |
| From… | For conversion from one to another thing. A lot like ‘To…’ executed on the dest object instead:  dest.FromSource(source)  The ‘To…’ prefix is more common, and usually more readable. |

#### Miscellaneous Names

* For number sequences you can use names like: ListIndex, IndexNumber, SortOrder. (Avoid Index because it is an SQL keyword.)

## Software Lifecycle

### Branching, Versioning & Release Management

<TODO: Describe a versioning methodology. >

<TODO: Assembly versions.>

## Team Management

[Under construction. A personal brainstorm of possible ways to go.]

This documentation section describes techniques you can employ for the hands-on technical leading of a software development team.

### Focus Points

It is still debateable, whether everything falls into these key areas. I am trying to structure this…

* Dialog
* Support
* Direction
* Planning
* Strategy
* Keep your cool
* Be open to other views

### Dialog

* Is the one I might forget about, but if I put extra focus on it, it would help a lot.
* Exchange ideas between team lead and team member.
* Exchange ideas between team members.
* Get to know eachother to make optimal ‘use’ of skills but also to enhance morale en good feeling, mutual understanding and promote openness.
* Assess what the team member’s problems are, regarding the way he feels about the work.
* Dialog should sometimes be limited too, because this job requires focus and quiet often, in order to do it well.

#### Periodic Individual Meetings

* Periodically talk to each developer individually about work done. Ask:
  + Currently working on
  + Done this period
  + Problems encountered
  + Large time-consumers
  + Positive notes
  + Other points of attention
  + Solutions to the problems
  + Next work to do
  + Good question: ‘What way of working, would make you enthusiastic? Not looking at the way we work here.’
  + Good question: ‘Can you tell me some interesting new things you found out?’
* The team lead also asks himself the same questions.

#### Reporting to Management

* Lead makes a report of issues discussed, preferably not grouped by employee, but grouped by development area. Employees can be mentioned with tasks.
* Lead discusses report with management.

### Limiting Dialog

* Prevent people from getting distracted by too much dialog.
* Promote sympathy for people that have ear plugs with music on.
* Only bother people when useful.
* Many people need to focus to program.
* You should be considerate towards people that need undistracted focus, ask more carefully if they can be disturbed and make them know, that is OK to say no to that.
* There are people that seem to handle distractions better, although I doubt their productivity does not get harmed.
* But perhaps I am wrong and these are the people that you can bother more often to vent about am issue.
* Consider moving a conversation to another room, so others can focus.
* Not speaking too loudly also helps a lot.

### Support

Support from the team lead is mostly inward directed toward the team members. But support is also has an outward direction from the team towards the rest of the organization or to the customer.

* A strategy of giving team members more freedom.
* Give people the responsibility, that their code works well and that others can still read it and adapt it.
* Support with solutions and techniques.
* Give examples
* Lead by example.
* Formulate suggestions with prefixes like “I would do it like this.” or “In my experience this works well.”
* So do not talk in definites, allowing the other person to form their own opinion and understand why or why not to take a certain route.
* You are going to have to accept that things will be programmed in a less than perfect way, or not precisely the way you would do it, and allow programmers to grow into making things better on their own.
* Look for a compliment to go along with it.
* When someone does not asks enough questions, as a lead you should inquire yourself.

#### Question Rotation

* Questions from outside the development department are handled on a rotating basis.
* Everyone takes turns in a day of being the one to ask questions.
* This will make sure most of the time developers can continue undisturbed with their programming work.

### Support & Direction

* Support and Direction are very closely related.
* Support is more oriented towards the coworker: what can you do to make him do a better job and feel better doing it and advance in his job.
* Direction is more about how can I make the work be beneficial to the client or rest of the organization. It is more outward directed.
* Both are about how can we do the work optimally to serve the goals.

#### Developer Skills

You can test someone’s skills in something by giving them a specific task, that requires one skill in particular.

Meta-skills:

* Research
* Documentation
* Bug fixing
* Process flow programming
* API programming
* UI design (not graphical, just the screen layout and program flow)
* Programming for reusability
* The worst-practices and almost worst practices lists from the architecture documentation.

Tools & techniques:

* Database development
* SQL

<TODO: Add more… Could be a pretty extensive list.>

### Direction

#### Miscellaneous Pointers

* Guard the main outlines
* Accept that people work outside the borders.
* Be involved at the beginning of a task at the end of a task and during the execution of a task too.
* If your involvement starts in the middle, do not try to to stear it too much into another direction.
* You could find the biggest problem area and then suggest that as a point of improvement.
* Meetings: (borders the topic of Dialog)
  + Try keep focus in meetings, so things do not go off topic too much.
  + On one hand I dislike it, when a meeting about a topic that could have been 30 minutes turns into 5 hours of a whole range of topics with only loose ends and inconclusives at the end.
  + On the other hand I dislike it, when the response to a proposal is: that is not what this meeting is about.
  + The intermediate version would be brief attention to off-topics and main focus on the topic at hand.
  + It also depends on whether it is a general meeting or a conversation about a specific topic.
* API choice: Medior developers and juniors are *not* free to choose any API without collaborating with seniors, leads and architects. It just has too high an impact. Not that we should not be open to new API’s.
* Framework development: Same point as API choice: juniors and mediors only adapt framework in colaboration with seniors, leads and architects.

#### Vary a Developer’s Tasks

What I often see, is that each developer is assigned to his application and never changes focus, because everyone feels, that this person is the ‘go-to guy’ for this topic. The motivation behind this is that it would be more economical to let the guy that knows most about it, do the work. That makes sense intuitively, but what’s intuitive, isn’t always what’s right. It is an over-simplified view on the issue that will cost you money. You lose something when you do not vary a programmer’s work.

* There is the risk that comes a developer becomes ill or falling away for another reason.
* It excludes other developer’s ideas from the subject.
* It will limit the developer’s personal development, because he never gets to do something else.
* It will encourage developers to not think about: “What if someone else had to work with my code?”
* The lack of critical peer-review could make the code quirky and hard to maintain.
* There is no opportunity for developers to learn from eachother.
* There is less incentive to review eachother’s code, which can translate to bugs and increased development time.
* It will be more difficult to scale up the amount of developers on a job.
* It will be more difficult to move a developer from one project to another, for instance when there is not much work in one area anymore, or higher priority work in another area.
* It does not promote a joint way of working, that makes it easier for others to pick up eachother’s work.

I cannot stress enough that it is a bad idea to let a only one developer work on a specific thing all the time. Did I mention that the the software developer’s personal development is slowed down? I have never been in a situation where this costs more and I have seen productivity go up doing it.

Here are suggestions of how to spread and vary the work over multiple team members. Variations can be mixed and mingled at will. That is the point. Be flexible.

* Variation 1:
  + Junior front-end
  + Medior back-end
  + Senior framework
* Variation 2:
  + Senior coaches medior
  + Medior coaches junior
* Variation 3:
  + Senior reviews code of mediors
  + Medior reviews code of juniors
* Variation 4:
  + 1 developer does rework of previous iteration
  + Another developer does new features.
  + Every now and then they switch
  + That way rework is not in the way of new developments.
* Variation 5:
  + One developer answers questions coming from outside the department and each day a different developer does it.
  + If needed other tedious task go to that developer.
  + That way the other days you know you can keep working on actual software development uninteruptedly and you at most only lose that one day that it is your turn.
* Variation 6:
  + Each developer or a set of developers works on a different part of the application or major feature.
  + Perhaps a junion/medior pair can work on a feature together.
* Variation:
  + A developer gets a similar task as before because he is now proficient at it.
* Variation 7:
  + A developer gets intentionally different tasks than before, to spread knowledge over the team and enriching the developer’s knowledge.
* Variation 8:
  + Some developers work on high-risk experiments.
  + Other developers work on low-risk productive feature development.
  + That way you create room for doing experiments that you do not know the outcome of yet or do not even know how long it is going to take or if it gives any useable results, without it actually stalling productivity completely.
* Variation 9: (very specific)
  + Seniors do:
    - Task management
    - Code review
    - Architecture
    - Framework development
    - Data model design for the most part
    - Software design
    - Rsearch / orientation of possible solutions
  + Mediors program:
    - Business logic
    - Data model implementation
    - *Some* framework and data model design but in tight collaboration with seniors.
  + Juniors do:
    - Front-end development, but under supervision so that patterns are followed.
    - Repetitive tasks, but only ones the junior can handle.
    - This may sound a bit deprecating, but sometimes a junior developer, when asked to do something in a very specific way, apply it the same all over the place, do it totally differently when you turn your back. That says something about the need for guidance. You have to monitor junior developers.

#### Upward Management

* Lead does ask critical questions to management about strategy and decisions.
* Lead advises management about (future) employees to hire.

#### Budgeting

* Lead investigates software and hardware requirements and opportunities to use newer software, offer better quality, quantity, service or reduce costs.

<TODO: Write a little more about it.>

### Planning

Borders the topic of ‘Direction’.

#### Tasks

* Split up tasks into multiple distinct pieces of work.

If the task is to broad, some people that think they can add another 200% of work to it, that they made up themselves. It will be unclear where a task starts and ends.

* As the lead, always give an expected estimated time it will take.

This will make someone aware of expectations, and of how deep to go.

* Lead must control task subdivision
* Lead must be asked for new task and discuss how to solve
* Lead must be informed of completed task
* Lead must be asked in case of unclarities
* Lead communicate the following rules
  + Begin Task -> Go to lead
  + End Task -> Go to lead
  + No Task -> Go to lead
  + Questions/unclarities -> Go to lead.

Fill in the leads name and basically hang up the wall.

* Data data model extension task should mention:

‘+ Add the basics around this field to all the architectural layers.’ Discuss verbally which layers. Use the Layering Checklist from the Software Architecture document to make a selection.

* A task should involve all the required work, but what is not asked should be left out

But this comes with problems.  
Ideally a developer knows which things to involve in a task, and which things to not involve.  
But in practice this is often not the case.  
One developer may not know what a task implies, another developer will involve too many side-issues.  
If you need to mention everything in a task, this is too much of a burdon on the one administrating the tasks.  
> TODO: Brainstorm and reformulate later.

### Strategy

* Strategy is broad. It takes anything you could do and molds it to fit your current goal.

<TODO: Write something about it.>

### Keep Your Cool, Be Open to Other Views

<TODO: Write some more about it.>

<TODO: Use this phrase? All this heated discussion about subjective things. Can't we just keep calm and discuss pros and cons and try to solve the problem together? Nothing is black and white in the choice of tools and techniques. It is always a trade off.>

<TODO: Use this phrase? Be less sarcastic.>

<TODO: Use this phrase? Someone with a way of working that does not promote maintainability, might not just be stupid.>

## Server Architecture

[Under construction]

The server subdivision is subject to the needs of the organization, so this overview is just a suggestion. The main concerns are safeguarding and keeping things optimal. Economics might force you to look for alternatives, but from a technical point of view the full set of servers with recommended configuration is advised. Cutting corners might make your IT run less efficiently, which would translate to cost overhead too.

<TODO: Mention the split-up into a C: and D: drive.>

<TODO: Reconsider the sizes of the development workstation drives after some cleaning up and counting disk space and considering extra dev tool requirements.>

<TODO: Consider the machine configuration needs in more detail.>

|  |  |  |  |
| --- | --- | --- | --- |
| **Stage** | **Name** | **Remarks** | **Configuration Focus Points** |
| Development | Database server | Stores a development copy of all the databases we use. | SQL Server, decent performance, particular focus on having enough RAM. |
| Development | App server | Where development can use a shared FTP server if needed, run long processes to aleviate the development workstations. Can also host shared web services, be it third party, be it internally developed ones, even though for that last thing it is usually better to run it on the development workstations. | IIS, preferrably many-core. SQL Server installation is advised, for delegating number crunching from the main development database server to another server. RAM is also important, since heavy number crunching processes may use a lot of memory. |
| Development | Source control server | For storing the source control database, running the source control services, running builds, unit tests and code analysis upon each check-in. | TFS. Must be decent configuration for each checking requires a heavy process to run, and the development team has to be able to work efficiently. |
| Development | Workstations | Each software developer’s own machine. | Two 21” monitors. No laptops, those run 2x slower. Core i5 for junior and medior developmers. Core i7 for senior developers and sofware architects, since they will more commonly work with larger solutions.  At least 8 GB RAM, so you can run large-cache applications and services. SSD for of 256 GB. Split up into a C: drive for windows <TODO: specify size> and a D: drive for data, main source code, but also aother frequenty accessed things. An extra ‘spinning disk’ drive with at least 256 GB storage for amont other things the ability to hold large database backup files. |
| Development | Laptop | One laptop for a whole team, just to connect to your workstation when you are in a meeting or being on the road to a customer | Relatively low specs. Core i3, a moderate amount of RAM. |
| Production | Database server for number cruching | For doing the heavy processing, like datawarehouse imports and processes, heavy reporting, heavy pre-calculation processes, to aleviate high-traffic production servers. | Many cores |
| Production | App server for number crunching | Same as above. | Many cores |
| Production | Database server for high traffic | For all the databases involved in high-traffic, storing data of user applications and services. |  |
| Production | App server for high traffic | For all the production web sites and services. | Note that RAM is very relevant, to meet in-memory caching needs. |
| Test | Database server |  |  |
| Test | App server |  |  |

### DTAP

<TODO: Write something about this. Include:

* Explain DEV, TEST and PROD. And ACC. And then also the INTERNAL and EXTERNAL production environments and their benefits, mostly with regards to publishing. Use the prefix ACC, instead of ACCEPTANCE, because the prefix goes all over the place e.g. ACC\_BuildlingBlocksDB instead of ACCEPTANCE\_BuildlingBlocksDB. But then again I like this better: acceptance.electroluxservice.JJ-bv.nl >

### Folders

If your servers and all your development workstations have a D: drive for data, then put the folders on the D: drive, otherwise put the the folders on the C: drive. Make a folder in the root of the drive with your company name:

D:\**JJ**

In this folder, each environment gets a sub-folder written in all capitals.

D:\JJ\**TEST**

D:\JJ\**PROD**

D:\JJ\**DEV**

Even machines with only one environment on it, should get a sub folder with the environment. That makes it better visible what environment you are working on, you can easily emulate the situation on a development workstation without additional configuration and it allows you to move environments from one server to the other.

Also add the following folders:

D:\JJ\**Install**

D:\JJ\**Backup**

<TODO: Add descriptions to each folder above.>

Those are not put in the environment folder.

The environment folder can contain the following sub folders:

D:\JJ\PROD\**Images**

D:\JJ\PROD\**IO Files**

D:\JJ\PROD\**Log**

D:\JJ\PROD\**Utilities**

D:\JJ\PROD\**Web**

<TODO: Add descriptions to each folder above.>

The Images folder contains images e.g. uploaded from an application, not images that are content of the application, so not icons or basic content of the web site, but user-uploaded images or images uploaded by content management systems. The Images folder can contain sub-folders to keep images apart from eachother, that belong to a different application or set of applications.

D:\JJ\PROD\Images\**QuestionAndAnswer**

D:\JJ\PROD\Images\**Synthesizer**

An application image sub-folder can contain again sub-folders, for resized images with particular dimensions:

|  |  |
| --- | --- |
| D:\JJ\PROD\Images\QuestionAndAnswer | Contains original images. |
| D:\JJ\PROD\Images\QuestionAndAnswer\**100x100** | Images resized to 100x100 pixels |
| D:\JJ\PROD\Images\QuestionAndAnswer\**320x280** | Images resized to 320x280 pixels |

The resolutions above are simply examples. You can have different sizes per application.

<TODO: Describe the Utilities folder, that you use fully qualified application names and version sub folders. Same for the Web folder, and add to that that it contains both web services as well as web applications.>

#### Development Workstation

Development workstations should have the same kind of folder subdivision, also put on the same drives as on the servers. You might not put web sites in these folders, but Imags, IO Files and Logs should be present in the same folder structure as the servers.

Put the source code folders in the same spot as all your coworkers. Then things keep cooperating with eachother. Most of the times relative paths work, but sometimes they don’t so it is a good plan to all have out copies of the source code in the same location. In case of TFS it should be D:\TFS that is mapped to the outermost root of the source control system. Not to a branch, not to a Collection, but to the server name or IP address of the source control server.

<TODO: Use this phrasing? - TODO: If servers store data files on C, then development worktations had better store it on C too, because it is incredibly confusing to have it in a different spot in development, for instance when you try to emulate a TEST or PROD environment on your development workstation and the letter C or D can be overlooked so easily in configuration files that you keep on making mistakes.>

### Backups

<TODO: Write something.>

## Appendices

### Appendix A: Layering Checklist

This checklist might be used if you want to bulk-program the architecture for an application by going through all the layers one by one:

* Data: Database structure
* Data: Data migration
* Data: Entity classes
* Data: Repository interfaces
* Data: Default repositories
* Data: NHibernate mappings
* Data: SQL queries
* Data: NHibernate repositories (optional)
* Data: Other repositories (optional)
* Data: Other mappings (optional)
* Business: LinkTo
* Business: Unlink
* Business: Cascading: DeleteRelatedEntitiesExtensions
* Business: Cascading: UnlinkRelatedEntitiesExtensions
* Business: Enums
* Business: String Resources
* Business: EnumExtensions
* Business: Validators
  + Delete Validators too
  + Warning Validators (optional)
* Business: SideEffects
  + SetDefaults SideEffects too
* Business: Managers
* Business: Extensions
* Business: RepositoryWrappers
* Business: Dtos (optional)
* Business: Calculations
* Business: Visitors
* Business: Factories (optional)
* Business: Api (optional)
* Business: EntityWrappers (optional)
* Business: Other helpers (optional)
* Presentation: ViewModels
  + Item ViewModels
  + List item ViewModels (some may only need IDNameDto, no ListItem view model)
  + List ViewModels
  + Detail ViewModels
  + DocumentViewModel (optional)
* Presentation: ToViewModel
  + Singular forms
  + WithRelatedEntities forms
  + ToListItemViewModel
  + ToScreenViewModel
  + CreateEmptyViewModel (not every view model needs one)
* Presentation: ToEntity
  + Singular forms
  + WithRelatedEntities forms
  + From screen view model
* Presentation: Presenters
  + List Presenters
  + Detail Presenters
  + (Edit Presenters)
  + Save methods in Detail (or Edit) Presenters.
* Presentation: Views (Mvc / UserControls…)
  + List Views
  + Detail Views
  + Main View (optional)

### Appendix B: Knopteksten en berichtteksten in applicaties (resource strings) (Dutch)

Er is een bepaalde structuur waar binnen we werken voor knopteksten en meldingen in onze applicaties. De hele bedoeling is maximale herbruikbaarheid, minimaal vertaal werk en correcte teksten. Dat doen we door hele algemene teksten op plaats X te zetten, zo veel mogelijk domein termen op plaats Y, en alleen wat er dan over is, komt in specifieke projecten te staan. Dit kan het verschil betekenen tussen 100’en of 10000 teksten.

Resources worden op dit moment overal neergezet waar ze niet thuis horen, met verkeerd hoofdlettergebruik en verkeerde interpunctie. En op andere plekken worden resources gewoonweg niet gebruikt en staat alles hard op 1 taal.

Hier moet secuurder mee om worden gegaan. Van ontwikkelaars wordt verwacht zowel de Nederlandse taal als de Engelse taal in de resource files te zetten. Bij twijfel over Engels, vraag het een collega.

#### Hoofdletters, interpunctie, spelling

Hoofdlettergebruik etc. is conform de taalregels van de betreffende taal.

1. Resources kunnen hele zinnen zijn. Die moeten correct geschreven zijn: In het Nederlands begint dat met een hoofdletter en eindigt het met een punt, tenzij het een vraag is, dan met een vraagteken. Blijkbaar is het nodig om dit aan te geven, want het gebeurt vaak niet goed.
2. Voor Engels gebruiken we de Amerikaanse spelling.
3. Namen van properties, classes en andere titels zoals knopteksten zijn in het Nederlandse zijn als volgt: "Links in artikel", dus alleen beginnen met een hoofdletter. En dus geen punt erachter.
4. Namen van properties, classes en andere titels zoals knopteksten in Engelse titels doen we als volgt: "Table of Contents", dus alle woorden beginnen met een hoofdletter, alleen onbelangrijke woorden zoals 'in', 'and', etc. in kleine letters.
5. Er is dus een verschil in hoofdlettergebruik tussen volzinnen en losse titels.

#### Assemblies

Resources worden met de assemblies meegecompileerd\*.

Termen worden zo veel mogelijk hergebruikt. Daarom zijn er plekken bedacht waar de termen thuis horen. Je moet in deze volgorde op zoek naar een resource die misschien al bestaat:

(Update: De hoeveelheid verschillende plekken waar resources staan is een zwakte van dit ordeningssysteem, omdat het verwarrend kan zijn. In toekomstige oplossingen is het wellicht een idee om resource teksten mmer op één centrale plek te zetten. Dubbelzinnigheid van termen in meerdere domeinmodellen is daarbij wellicht meer een uitzondering dan een regel, waar omheen gewerkt kan worden.)

1. 'Save', 'Close', 'Edit', etc. staan in Framework.Resources, toegankelijk via de *CommonResourceFormatter* class.
2. Validatiemeldingen uit Framework.Validation, toegankelijk via de *ValidationResourceFormatter* class.
3. CanonicalModel: een tussenmodel voor uitwisseling van gegevens tussen verschillende systemen, toegankelijk via de *CanonicalResourceFormatter* class.
4. Business layers bevatten alleen vertalingen voor de overige teksten die niet in het canonical model staan.
5. Ook teksten die niet direct domeintermen zijn, maar wel in applicaties worden gebruikt op plekken waar het gaat over een bepaald business domain, mogen in de business layer, zijn resources gezet worden.
6. Presentation layer bevat over het algemeen geen teksten. Die zetten we in de business layer: we hebben al genoeg plekken waar we resources neerzetten.

#### Tips

1. Gebruik van placeholders zoals {0} is toegestaan, maar dan moet je wel een class erbij maken, die de placeholders vervangt. Zie Framework.Resources voor een voorbeeld. Het is dan verstandig om de resources zelf internal te maken en alleen de class die de placeholders vervangt public te maken. Kijk echter uit dat je het daarbij geschikt houdt voor meerdere talen, want een creatief met placeholder opgebouwde resource string werkt al gauw niet voor een andere taal.
2. Negeer dat de beschreven werkwijze kan resulteren in berichtteksten met hoofdlettergebruik zoals: 'Het **Ordernummer** is niet ingevuld bij de **Bestelling**.' Als we dit aanpakken, doen we dat met een algoritme, niet met nog meer resources.
3. Het is verstandig om teksten in de applicatie algemeen te houden. Dus bijv. 'Artikelen', i.p.v. 'Artikelen in dit boek'. Dit scheelt vertaalwerk. Ook dit is verstandig: 'Artikel 1: Naam is verplicht.' Daarbij zijn de teksten 'Artikel', 'Naam' en '{0} is verplicht.' waarschijnlijk allang vertaald. Door de ‘dubbele punt’ notatie aan te houden ('Artikel 1:'), voorkom je het verwerken van de term in een zin, wat voor iedere taal een compleet andere grammatica kan zijn, wat voorkomt dat er complete volzinnen vertaald moeten worden.

*\* Resources staan niet in een database, omdat het applicaties trager maakt en kun je de code niet draaien op omgevingen waarbij je geen toegang hebt tot de database. In sommige situaties kun je niet eens compileren zonder toegang te hebben tot een specifieke database. Bovendien geeft mee compileren van resources in specifieke projecten ons de mogelijkheid dubbelzinnige termen anders te vertalen per business domein.*