# Rules:

All business logic resides in the Domain. The Domain contains business logic and nothing else.

Other assemblies may have a dependency on the Domain, but the Domain must not have a dependency on anything (except .Net framework).

In the application layer, there should be one service per entity.

All the application layer should do is load entities, call the domain logic, save entities and raise events.

Each service method should only call one domain method (ubusally named simillarly).

ViewModels should represent the action, not the entity. For example, have UserController.Register(UserRegisterViewModel viewModel)

Rather than UserController.Register(UserViewModel viewModel). Otherwise the entity view model has to do to much, becoming all things to all men, and ineveitable getting shoehorned into various places it does not fit, and having mutally exclusive properties. Also causes validation issues.

Do not attempt to use aggregate roots unless you REALLY understand them.

When doing Ajax, always call a controller that returns a partial and display that partial. In the case of grids, this becomes very important when using paging and sorting.

Ajax and reuse is the only reason to use partials, they should not be used as a way of organising code.

Do not use ajax call which reload the page in their callback - this is pointless!

The user accessing the database must not have delete permissions. Nothing must be deleted from database, only soft deletes.

Takes some work in NHibernate. This paranoia is due to a particular incident in BusinessSafe.

One mapper per ViewModel. Never repeat any of the mapping logic. If you need to map a child ViewModel from a parent ViewModel, delegate to the child ViewModel's mapper.

The system should access only its own database and no other.

For domain, follow one test class per feature pattern (feature is generally a use case/method). Tests should be in a folder named after the entity.

# Decisions:

Scrapped DTOs: we don't need them.

Calling repositories directly from the controllers: 'Get' services in the Application layer we an unnecessary overhead.

Validation: have separate method to validate before the use case, which takes the same parameters. Decided it is better than having a single validation method on the entity which is called before saving, as this does not validate within the specific context.

Validate for raising and editing a Task may look similar and could be refactored out but I have left them like this for clarity.

Returning helpful list of validation errors rather than throwing exceptions.

Having some validation on the view model: validation on the view model is to be helpful to the user, validation on the domain is more secure and for complex business logic it is the only way to do it.

Static classes to map view models: the controller should be tested as ine unit, the output of the viewmodels based on the input of the repositories. We do not need to test controllers and ViewModelMappers separately. In fact, by separating them we overcomplicate the testing process which means developers often do not bother.

This project template uses the 'Session per controller action' pattern, rather than the usual 'session per request' pattern. This is because when we use NServiceBus, we need to wrap the session and transaction in a TransactionScope.

Followed this guys Ajax example: <http://stackoverflow.com/questions/5410055/using-ajax-beginform-with-asp-net-mvc-3-razor> Not using Ajax.Begin Form - not sure why but seems to be the popular consensus.

Discuss whether enumerations are better or worse than lookup tables - enumerations are a problem for reporting.

What actually is the responsibility of a controller? react to user events. So mapping of domain objects to view models is delegated elsewhere. But it must still be testable as one unit (pass entity in, get view model out) otherwise it becomes too cumbersome to test, people don't bother and we fall back on acceptance tests. Need to be pragmatic.

# Other good ideas/to do.

Is there a way of using annotations for validation of ajax call?

Testing of JavaScript.

How to test validation?

Sort CI/Mercurial integration in cluding hgignore file.

Use Cassette as use in BusinessSafeOnline.

Use the auditing as used in BusinessSafe. This is possibly essential as it saved our life at one point.

Auto registration of StructureMap.

Would be good to have a paged list of tasks to further demonstrate why it is important to return partial views in Ajax.

Authorisation.

USe DBDeploy as done in BSusinessSafeOnline.

Set up whole deployment system in this template.

Transforms of config files and accompanying project - see BusinessSafe.

Whole WCF area still to do.

Control/reduce amount of information NHibernate is putting into log4net. Writes SQL command but not parameters?

Add a 'user' colum into log4net output.

Prerequisites:

Needs the intranetuser database user setup with id 'intuspas'

# Discussions:

Validation - throw errors, validate first or special case pattern? Validating before amending entities and validating before persiting entities on view model and entities.

Complete Task is example of validation logic that is too complex not to go in domain.

Should the use cases call the validate methods again and throw an exception if there are any that have got through? probably not bother.

What is the best way to handle the Username? Currently I just get it in the Controller and pass it through. Probably not the best? I even do this for WCF - very insecure.

How big should the list of parameters be before we bother with a request object? should we bother at all?

Discuss the way cancel works - the way we have to serialize and pass in the whole form (so we can return the partial based on search criteria).

Discuss how I have handled to constants for Id for permissions - is this the best idea?

Is the database security done in the best way?