Project Information			
Project Title (and acronym)	Principles in Patterns (PiP)		
Start Date	01/09/2008	**End Date**	01/08/2012
Lead Institution	University of Strathclyde		
Partner Institutions	n/a		
Project Director	Dr. Veena O'Halloran		
Project Manager & contact details	Jim Everett (jim.everett@strath.ac.uk, 01415482186)		
Project website	http://www.principlesinpatterns.ac.uk/		
Project blog/Twitter ID	http://www.principlesinpatterns.ac.uk/blog.aspx		
Design Studio home page	http://goo.gl/B4ry4		
Programme Name	Institutional Approaches to Curriculum Design		
Programme Manager	Sarah Knight		

Document Name	
Document Title	Principles in Patterns: Technical Report
Author(s) & Project Role	Dariuz Jabrzyk, Developer; Jim Everett, Project Manager

Principles in Patterns (PiP):

Technical Report

July 2012

Dariusz Jabrzyk & Jim Everett

University of Strathclyde

Contents

1. Background and context

SharePoint was initially chosen as a development platform for the PiP project. This technical paper will provide an overview of SharePoint functionality, the ways in which it can be developed, and it will also describe the development journey undertaken to meet the system requirements as specified within the PiP Project [1].

Work on technical development of the prototype course and class approval system began in November 2009. The first important activities were to gain an understanding of the project aims and to become familiar with analysis work undertaken by the project team to date. As PiP was a research project the requirements for the system were not particularly specific at this stage – rather, they were in the form of a set of ideas and expectations relating to the type of functionality the system could potentially provide.

An analysis of current processes within the University of Strathclyde, relating to course and class approval, identified two main areas of potential functionality for the system:

1. Course and class design
2. Approval workflow

Initial requirements focused on the pedagogical side of the course and class approval process. The main priority was to develop course and class online descriptor forms in a way that would increase the quality of the specifications of new classes and courses (from both formal and pedagogical points of view). These forms would be designed to replace currently used paper-based forms. It was recognised that users should be able to create, edit and store descriptors of new classes and courses within an electronic system. The idea was to create forms that would display dynamic "pedagogical support content", depending on which section of the form the user was working on. This supporting content would be displayed as tooltips / help side bar / links to external sources which would dynamically change while a user was engaged in filling in various sections of the form. The "support content" would encourage course and class designers to create new high quality classes and courses. The system would be designed to provide a central repository of the course and class descriptors created. The system would also hold a central repository of pedagogical support content, which would be easy to maintain by business users. When a user completed the design of a new class or course descriptor they would be permitted to submit it as a "proposal" for approval. Analysis of current processes identified that various steps and tasks within the approval workflow were being conducted at various levels within the institution [2].

1.1 Technical note: InfoPath and C-CAP

In the latter phases of the PiP Project a decision was taken to move towards a SharePoint approach using InfoPath. This approach was part of the PiP Project evolution and would eventually produce the final prototype system, to become known as C-CAP (Class & Course Approval Pilot). The rationale behind this change in technical direction is summarised in the PiP Project "Institutional Story" [3], as well as other Project outputs [4], and is not the subject of this paper.

2. SharePoint 2007 as a development platform

Microsoft Office SharePoint Server 2007 was chosen as the platform for the system. SharePoint is an extensible web application that provides the functionality to store, share and manage various forms of digital information and facilitates collaborative working with information [5]. Functionality is exposed through both "out of the box" components and a .NET API.

Out of the box components enable portal-style web sites to be built, which are accessible through a web browser by authenticated users. SharePoint facilitates rapid development of these web sites whose structure and functionality can be based on one of the templates provided as part of the product. Access to SharePoint sites can be secured by appropriate permissions configuration and various permission levels can be set up (e.g. full control, contribute access, view-only, etc.). Below are examples of the most commonly used SharePoint "out of the box" components:

- ✓ SharePoint sites. A site is a collection of web pages, document libraries, lists and other components used for storing, sharing and managing information. SharePoint sites can be customised in various ways to support various business processes and to achieve related business goals (e.g. project site for tracking project progress, collaboration site for exchanging ideas and sharing information around a given topic, team site for storing and managing various documents centrally, etc.)
- ✓ SharePoint lists. Lists are one of the main types of containers for holding tabular data within SharePoint. Lists may be compared to database tables although they do not provide database typical features (like enforcing relationships, joins, etc.). SharePoint lists may contain any type of data (e.g. simple text, html pages, rich html text, numbers, choice fields, binary files, images). In fact, every type of tabular data container is a list. SharePoint is able to display various types of lists in different ways e.g. a calendar list will render a view similar to Microsoft Outlook's calendar; however underlying data is still stored in the standard list tabular format.
- ✓ Document libraries. They are website-based containers for document based file types like Microsoft Word or Excel. Document libraries allow users to store their files centrally, in one place. SharePoint also provides versioning functionality, allowing users to track changes made to their documents by multiple authors. Check-in / check-out functionality allows users to lock the file they are currently working on to prevent other users from making changes. Since a document library is just a type of a SharePoint list, "metadata" columns may be added to a document library allowing the capture of relevant information about documents in order to facilitate searching.
- ✓ Components supporting collaboration work like calendar lists, task lists, blogs, discussion boards, wiki pages, issue tracking, surveys, announcements, contact lists.
- ✓ Web parts. Web parts are the building blocks of SharePoint sites. A Web part is a type of web module or mini application that enables information to be displayed on a particular location of the page or that performs a specific function. SharePoint provides a rich set of out of the box web parts which users can employ on their sites.

SharePoint allows users to create sites and re-use out of the box components through a process of simple browser-based customisation (no custom development required). In many cases SharePoint's out of the box functionality is sufficient to create solutions that will meet business objectives. However, there may be cases where core SharePoint functionality is not sufficient to achieve business aims. In such cases custom development is required.

SharePoint exposes a rich API (.NET framework) which provides the ability to extend the application programmatically. Knowledge of the API and SharePoint development techniques is necessary to develop custom building blocks and to use them effectively within sites. These building blocks will usually provide functionalities not possible to achieve through simple customisation of "out of the box" functionalities. In this way, SharePoint becomes a development platform.

As SharePoint is both a ready-made product and fully extensible development platform there exist many development opportunities to extend its functionality. The difficulty may be for novice SharePoint developer (with .NET background only) to get familiar with SharePoint specific development techniques and change his/her development "mind set" (e.g. use SharePoint lists

instead of relational databases when appropriate). Thanks to existing "out of the box" functionality, skilled SharePoint developers may quickly focus on delivering actual business functionality instead of spending time writing "plumbing" infrastructure code (e.g. data access layer, custom authentication and authorisation). However, if the software specification requires it, it is still possible to embed a standard .NET layered application within SharePoint. There also exists many new development opportunities which do not exist for standard .NET development. Below is the brief list of what can be achieved within SharePoint using both "out-of-the-box" and custom development:

- Main building blocks of SharePoint sites are web parts. Each web part is a "mini application", self-contained module which can be embedded within a SharePoint site. Such web parts may display data from various sources, being it SharePoint itself or even external systems. Additionally, web parts provide infrastructure to communicate with each other so data displayed within one web part may relate on other web part. SharePoint provides many "out of the box" web parts, however developers are able to develop custom web parts, too. Web parts, once developed may be included and reused on many sites.
- Built in SharePoint "infrastructure" functionalities free developers from writing lots of "infrastructure" code. Such functionalities include authentication and authorisation, advanced search across various scopes (e.g. farm, site collection), navigation components, and so forth.
- Most web pages within SharePoint are so called "site pages". Site pages may be customised by users through web browser or through SharePoint designer. Users may add web parts into their pages (either out of the box or custom developed). This provides a lot of power and flexibility. Users may create business solutions by themselves, without the need of custom development. Site pages have their limitations, too. Due to security it is not allowed to write custom code within site pages directly (it may be used within custom web parts, though). As site pages are stored within SharePoint content database their performance is slower in comparison to standard ASP.NET pages.
- SharePoint allows the creation of so called "application pages". These are just ASP.NET pages embedded within a SharePoint master page. Using application pages developers may create standard, layered .NET applications within SharePoint context if needed. Application pages cannot be customised by users and developers are allowed to write custom code within them. Application pages are not stored within content database but on the front end servers file system. They therefore perform better in comparison to site pages.
- SharePoint allows connection to external Line of Business (LOB) systems (through Business Data Catalog) and display LOB data in the form of a SharePoint list. This data is treated as if it was stored natively in SharePoint, therefore it integrates with other SharePoint functionalities (e.g. SharePoint Search). It is also possible to specify permission rules on external data, i.e. give access to it only for authorised users.

Developing for SharePoint has its drawbacks, too. It has a steep learning curve and due to the specific nature of development techniques, a developer's productivity is lower compared to that of standard .NET development. However, if a developer is able to reuse SharePoint's out of the box functionality, in addition to custom development techniques, then overall productivity may be the same or even better in comparison to standard .NET development. The most important thing is to make an informed decision about which of a particular solution's requirements can be met by out of the box functionality and which require custom development. This could prove difficult, however, especially when developers may not be particularly experienced and/or when requirements are not very clearly defined.

3. A strategic approach to SharePoint development options

3.1 Understanding custom development options

When developing solutions based on SharePoint, it is important to try to reuse out of the box functionalities first. SharePoint exposes rich API to use its features programmatically and code written for SharePoint-based solutions will usually call various SharePoint specific objects. Custom development ideally should cover only those requirements which cannot be met by features provided directly by the SharePoint platform. Custom development should provide additional value and should aim to extend SharePoint features where necessary.

Deciding which requirements may be covered by SharePoint and which need custom development may be difficult, as previously noted. A team designing solutions based on the SharePoint platform should have enough knowledge about available functionalities and development opportunities to avoid re-inventing the wheel by writing custom code only (which would take more time than standard .NET development).

This is an important lesson learned when working on the PiP course and class design and approval prototype system. It was known that some of the requirements could be accommodated by out of the box SharePoint functionality while others clearly required custom development. There were also requirements that were not so easy to identify, positioned somewhere in-between; in this case it was unclear whether they could be fully met by reusing SharePoint features or whether they would require custom development. In addition to this, further complexity comes from the fact that different custom development options or approaches can give the same results – some of them requiring less work but having limitations, others requiring more work but giving more flexibility.

During the development of the course and class design and approval prototype system, various routes of achieving the desired functionality were explored. Some of them were found not to be viable and were abandoned over time due to inherent limitations. Some of them seemed to be a promising approach technically but were also abandoned because system requirements changed due to external factors. Some of them appeared to be the right path but alternatives also existed and had to be considered. This document will try to briefly describe the technical development and research process and useful findings about SharePoint development opportunities and limitations, in relation to the development of a prototype system for a course and class approval system.

3.2 Development rationale for course and class approval prototype

Why was SharePoint chosen as a platform for developing a course and class approval prototype system? The decision was initially influenced by the fact that the University already had SharePoint installed, configured and in use by a small community. Creating a system based on a currently used platform, with a familiar user interface and to which users could log in using their existing Active Directory accounts formed a good basis for using it to develop PiP's solution. The University also had a continuing commitment to SharePoint as an intranet platform, which made the continued sustainability of any solution much more viable than any standalone development.

SharePoint's main strength and selling point is that it is a document management and collaboration application. This functionality would underpin requirements associated with the "design" element of the system. SharePoint also strongly supports workflows (Windows Workflow Foundation) which was a crucial element to establishing the "approval" part of the system. Further research into the use of SharePoint as a development platform was undertaken and this continued as an iterative process throughout the actual development process, during which further decisions were made (e.g.

abandoning some of the development paths because of theirs limitations). Additionally, choosing a particular development path was strongly influenced by problems with requirements specifications, lack of decisions, dependencies on other teams, restructuring within the University and numerous changes within the project team.

The table below summarises the initial high-level requirements and indicates whether they were able to be met by existing SharePoint functionalities or required custom development. As can be seen, many of the course and class approval system features could be covered or partly covered by features already provided by SharePoint, but requirements also existed that required custom development in order to provide optimal functionality and to best meet the requirements identified.

Requirement	SharePoint features	Custom development
To develop dynamic course and class descriptor forms which would display additional pedagogical support content dynamically depending on which section the user was working on.	SharePoint lists are not flexible enough to create complex course and class descriptor forms and to additionally display dynamic support content. InfoPath forms can be constraining for this purpose, too (e.g. use of JavaScript within forms is not possible).	Using custom ASP.NET forms seemed to be the most flexible choice. This would enable the creation of complex forms and facilitate the use jQuery to dynamically display pedagogical support content
Single course and class descriptors repository.	SharePoint seemed to be a good choice for this, however going with custom ASP.NET forms raised questions about how data from these forms would be stored within SharePoint only (SharePoint lists are not a replacement for a database structure).	Opting for custom ASP.NET development would require development of custom persistence layer for data gathered within forms. This would obviously require much more development work but would give much more flexibility in terms of reporting, integration with existing data and the ability to respond to changes.
Authentication and authorisation requirements – various groups of users would have access to different parts of the system (e.g. course and class designers could see only their proposals, members of various committees would have rights to approve / reject proposals).	SharePoint provides flexible authentication integrated with Active Directory and also good authorisation configuration opportunities based on SharePoint groups and permission levels.	No need to develop custom authentication and authorisation. Configuring SharePoint permissions appropriately is enough.
The system should provide a repository of additional support content which should be easy to maintain by business users.	Additional support content can be easily maintained within SharePoint lists.	No need to use a custom database for additional support content. Only need to develop some way of hooking support content up to particular sections in the form (e.g. through AJAX calls and jQuery).
Collaboration related functionality facilitating communication, classes and courses design process, approval decision process.	SharePoint provides a lot of collaboration features, e.g. automatic email alerts, discussion boards, wiki pages, task lists, announcements, calendars, document libraries, meeting workspaces, document workspaces. All of these features could be re-used in the context of course and class design and approval.	In cases where SharePoint out of the box features would not be flexible enough then they might be extended by custom development against SharePoint API.
The user interface should be familiar to users, have easy navigation and should preferably be consistent with existing systems.	SharePoint provides out of the box sites with a consistently structured user interface as it is based on standard master pages, themes and CSS style sheets. Navigation is also configurable.	SharePoint out of the box user interface may be enhanced by custom development if needed (e.g. jQuery enhancements, CSS modifications, themes, custom navigation).
Business rules and approval workflow steps should be encoded within the system and automated as much as possible.	SharePoint is built on top of Windows Workflow Foundation and provides a set of out of the box workflows which may be associated with documents and list items.	Specific custom workflow templates with custom workflow activity pages may be developed to supplement SharePoint out of the box workflows.

4. SharePoint development techniques

Custom SharePoint development requires knowledge of SharePoint API and specific development techniques. A significant part of the development process of the course and class approval prototype system was to learn SharePoint API and its development techniques.

Initial research was not encouraging. Conclusions from community-based discussions, gleaned from various articles and blogs, were that SharePoint 2007 development is difficult, development on this platform is less productive, it has a steep learning curve and also has certain limitations. In addition, Microsoft support for SharePoint developers by providing proper IDE for SharePoint was unsatisfactory. This situation has been largely improved since the release of SharePoint 2010 and Visual Studio 2010; but these improvements did not apply to SharePoint 2007. Also, SharePoint is a complex platform with a considerable range of specific approaches to development, which initially seem to be counter-intuitive to developers. For example, list templates are identified by integer numbers and the question arises as to why these weren't made easier to identify by giving meaningful names to templates.

4.1 Configuring the development environment

Development for SharePoint requires a developer to have the SharePoint product installed on his/her development workstation; but SharePoint 2007 can only be installed on Windows Server systems. So the workstation being used should have Windows Server 2003 or 2008 installed. To make the development machine as close as possible to the production environment it should have a dummy domain controller configured or should be joined to the existing domain with several service accounts dedicated only to the SharePoint application. The community suggests creating a development workstation as a virtual machine – this allows a developer to create snapshots in case any testing corrupts the SharePoint installation. This requires a developer to have a powerful host machine with lots of RAM – at least 2GB of memory should be dedicated to the development virtual machine only (preferably 4GB). Ideally, virtualisation should be based on "bare metal" virtualisation technology. Windows Server 2008 has good virtualisation technology – Hyper-V. The development workstation should ideally have fast quad core processor, at least 8 GB of RAM with an additional SSD hard drive to host virtual machines and Windows Server 2008 installed as a host system. It makes SharePoint development expensive. Software licences require additional investment too. One could cope with using trial versions but they usually expire within 3 to 6 months. Ideally, a developer should have an MSDN account which allows the download and use of any software for development, testing and presentation purposes; however, this comes at a cost too. Fortunately, universities receive a discount from Microsoft.

The requirements for a development machine certainly influenced the technical development of the course and class approval system in the early stages. The team working on the project were not part of a well-embedded or well-financed and resourced department so did not have powerful machines to use. Developing for SharePoint on 1.2 GHZ 12 inch Toshiba laptop with 2GB of RAM and 5400 rpm HD is inadequate and frustrating. Fortunately, the need for a more powerful machine was identified quickly but it took some time to procure one and to configure it in an optimal way. Initial configuration of the development virtual machine took some time too – the community was very helpful in this case, providing a detailed guide of steps required to configure SharePoint development Virtual Machine properly, in the absence of any in-house support or extensive experience.

Configuring the development machine was not enough to get to grips with the SharePoint development approach quickly. Deployment of a SharePoint solution is not as simple as only copying files into the IIS server as is standard ASP.NET applications. One needs to create a so-called solution package which then may be installed on the SharePoint farm. A solution package is a file with a *.wsp extension, which is basically a cab file containing all resources (like SharePoint

components definitions, feature definitions assembly libraries, JavaScript files, user controls, web parts definitions, resource files) required by the solution. wsp solution package contains a manifest.xml file which simply tells SharePoint what elements are in the solution package. It is a mark-up code written in Collaborative Mark-up Language (CAML). Originally, developers had to write this manually, which significantly limited productivity – each time some new components were added to the solution the manifest file had to be updated before testing could be performed. Assuming that the average developer switches between coding and testing many times during the day it may impose a significant limitation on their productivity levels. Fortunately, the SharePoint community came up with some solutions. Dedicated developers created several open source tools with the capability of creating manifest.xml (and other CAML-based definition files) automatically. One of the most widely used is WSPBuilder – a Visual Studio extension largely simplifying SharePoint 2007 development by providing several solution templates and generating CAML code common to SharePoint solutions.

Successfully creating a solution package is not enough, though. The solution package is first installed into SharePoint farm. The SharePoint farm administrator then needs to deploy it to specified SharePoint web applications. Again, repeating this process many times during the day for testing purposes limits development productivity. Fortunately, this process may be automated by creating a deployment script to run on each compilation within Visual Studio. This process was improved in SharePoint 2010 development as a response to developers' complaints, but the improvement was not applied to SharePoint 2007.

This is not the end of deployment process. Usually any piece of functionality should be deployed within a given scope within the SharePoint application, whether it is at farm level, web application level, site collection level or specific site level. This is achieved in SharePoint by using so-called features. A feature definition consists of 2 files: feature.xml and elements.xml. The former defines general feature properties (like name, id, scope). The latter defines any SharePoint artefacts to be deployed within a given scope with the feature. These may be custom site columns, content types, list definitions, navigation elements, web parts, user controls and other SharePoint specific components. Unsurprisingly, all these artefacts are defined using CAML language. The WSPBuilder tool creates an initial structure for the most commonly used artefacts but any modifications to them must be done manually. In addition, it is possible to develop "feature receivers", which are just pieces of .NET code which may run on a given feature activation related event (e.g. FeatureActivated). It gives considerable flexibility and facilitates making almost any change to the given scope in the SharePoint application (using SharePoint Object Model API). In this way SharePoint becomes an extensible development platform, with features providing some kind of "plug and play" type deployment.

Once a feature is defined and deployed to the farm using a solution package then an administrator of a given scope (e.g. site collection) must activate the feature to deploy any components defined within the feature and run the activation code developed within the feature receiver. This takes us to the point where a solution may be tested. Each of these steps limits development productivity, but once a developer knows all the steps s/he may automate them by writing deployment scripts.

4.2 Debugging

Debugging in SharePoint 2007 is a bit different to more traditional platforms, too. It requires a developer to manually attach a Visual Studio debugger to the IIS worker process (w3wp.exe) each time code has to be debugged (usually many times during the day). An appropriate version of the assembly must be loaded into the Global Assembly Cache (GAC). W3wp.exe and / or OWSTimer.exe processes may block access to the assembly file. So to update GAC with the appropriate version of assemblies both processes must first be manually killed. It may be useful to create batch files to run before and after each compilation in Visual Studio to perform all the necessary cleaning up.

A Visual Studio SharePoint solution should have an appropriate folder structure created which mirrors folders within the SharePoint root folder. The SharePoint root folder is named with “12”. It contains the “TEMPLATE” folder which may contain the following folders:

- CONTROLTEMPLATES – save all user controls in this folder (*.ascx). In addition, create a folder specific to your application within this folder.
- FEATURES – contains all feature definitions and associated resources. All of these should be located in a separate folder for each feature.
- IMAGES – save all images used by your solution in this folder (create a folder inside to avoid naming conflicts with existing files).
- LAYOUTS – this folder contains all application pages and associated resources (best practice is to create additional folder for your solution inside this).

Each type of file should therefore be placed within an appropriate folder. If a solution was to be deployed manually then all folders with files would need to be copied to the SharePoint root folder. WSPBuilder automates this task.

Deploying a solution in SharePoint sometimes means making changes to the SharePoint’s web.config file e.g. when custom user controls are created you need to add their assembly as a “safe control” within SharePoint’s web.config file. Doing this manually may lead to a mistake which will break the entire SharePoint web application. Fortunately, WSPBuilder will make all required changes to the web.config file when deploying the solution.

This document will now describe other SharePoint development specifics discovered during course of technical development of course and class design and approval prototype system.

5. Development of demonstrator

There were several non-technical difficulties which constrained the development of the system. The main one was the lack of well-defined requirements specification, which was caused by the difficulties encountered when engaging potential users with the project. It is very difficult to identify the right path or approach to development in a short space of time, with only a set of general ideas rather than a detailed specification and formalised requirements. Lack of user engagement was identified as a risk by the team at the beginning of the technical development of the system. To encourage potential users to engage in the requirements gathering and analysis process the team decided to develop a user interface “demonstrator” of the class descriptor form and other views of the system. Development of the demonstrator started at the beginning of December 2009 and finished at the end of January 2010.

By developing the demonstrator the team wanted to illustrate the potential functionality of the system to the PiP steering group and potential users. At this stage strong emphasis was being put on the creation of forms in such a way that they could improve courses and classes design. The idea was to create dynamic forms which would display additional pedagogical support content dynamically, depending on which section of the form the user was working on. The demonstrator was not a working system – it was not saving data in the back end at this time. Its purpose was purely to demonstrate how the user interface of the system might look.

For the purpose of user interface demonstration work entirely focused on developing example online class descriptor form. Class descriptor forms are simpler and shorter in comparison to course descriptor forms. The team gathered more examples of paper based class descriptor forms than course descriptor forms. Class descriptor forms are shorter and simpler than course descriptor forms which are much more complicated. The team gathered a number of class descriptor forms from various faculties and departments. The forms varied across faculties and were not strictly

standardised across the University. However, there were also common sections on every form according to University of Strathclyde guidelines on the design of new classes and courses. For the purpose of demonstrator the team decided to include some of those common sections on the example online class descriptor form which would be familiar to most of the audience of demonstration.

Before the actual development of an example class descriptor form, the initial analysis of SharePoint development opportunities had to be undertaken. As a result, several ways of developing data entry forms in SharePoint were identified.

The simplest approach seemed to be to create a SharePoint list to mimic the class descriptor form. However, when creating a SharePoint list one can use only a limited set of field types to gather data (e.g. a text field may be used to capture free text descriptions, a number field may be used to capture numbers, etc.). For many questions on the form it would be sufficient to use standard SharePoint field types but some of the questions on the form had a more complicated structure. In addition, it was decided by the team members that it would be a good approach to move away from "free-text" fields, where appropriate, and instead use more structured and constrained data entry fields in re-designed class and course descriptor forms. This could improve classes and courses design from pedagogical point of view. For example, for the "Learning Outcomes" section there could be a "repeating section" control used. Each section would capture data for one learning outcome only and could contain several data entry fields to capture relevant information about a given learning outcome (e.g. learning outcome name, learning outcome objective, performance criteria, assessment method, etc.). Some of the fields within each section would still be free text fields but others could provide a constrained set of answers to questions (e.g. in the checkbox list or multi-select dropdown list) to improve standardisation and the consistency of descriptors. A repeating section would allow users to add or remove learning outcomes as required. Means of structuring the data gathered via descriptor forms would enable better reporting in comparison to "free text" fields data only. SharePoint lists, however, are primarily intended to capture simple tabular data and the field types provided by SharePoint do not permit the creation of the "repeating sections" model, or other more complex data entry controls. Additionally, the purpose was to develop a class descriptor form that would dynamically display "pedagogical support content" depending on which section of the form the user was filling in. It was determined that this could not be achieved with SharePoint lists. Developing a custom ASP.NET form with JavaScript client code to dynamically display "pedagogical support content" seemed to be an the best choice and the feasibility of this approach was then tested.

Custom ASP.NET forms can be developed and embedded in SharePoint in different ways:

- Within custom user controls
- Within custom web parts
- Within application pages

User controls are a fundamental type of development component within ASP.NET framework. A user control is an ASP.NET mark-up and associated .NET code (developed in C# or VB) which may be re-used within ASP.NET pages or other user controls. Through further research and development it was discovered that a class descriptor form could be developed as a user control and then SharePoint Designer could be used to customise any *.aspx page within SharePoint to register and place custom user control. Before custom user control could be used in SharePoint it must be registered as a safe control within SharePoint's web.config file. When referencing any resource files within user control (e.g. CSS stylesheet, JavaScript files, images) then, as mentioned earlier, folders structures specific to SharePoint must be taken into consideration.

Once a user control is created it may be embed within a custom web part. A custom web part may then be placed within SharePoint web part page by using browser-based customisation, with no need to edit the page in SharePoint Designer). Web parts are the main building blocks of SharePoint

pages. The WSPBuilder tool allows developers to create a Visual Web Part project type which is basically a user control wrapped within a web part. WSPBuilder generates required code and also updates the web.config file during deployment.

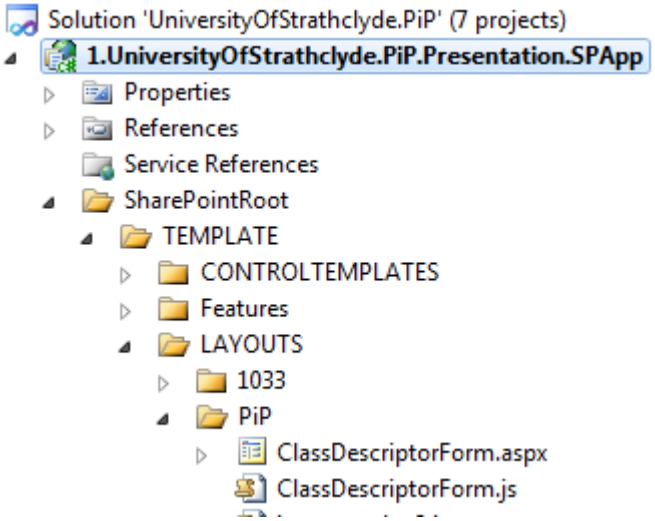
Alternatively, a custom user control may be placed within a custom *.aspx page which then may be deployed as an "application page" within SharePoint. Application pages are stored within the "LAYOUTS" folder under the SharePoint root folder. Application pages may be accessed from any site in SharePoint by adding a "_layouts/Application Page Folder/application_page_file_name.aspx" suffix to the URL address in SharePoint. To maintain the SharePoint user interface look and navigation, the custom application page must reference the SharePoint master page. For application pages the file named application.master, located within "LAYOUTS" folder, should be used.

For the purpose of the demonstrator, a mock class descriptor form was developed as an application page. This was the easiest and quickest way to develop a user interface demonstrator. However, particular sections of the form were developed as user controls. This allowed their reuse within custom web parts later on.

Using ASP.NET to develop mock class descriptor form was the quickest and most flexible approach as it facilitated the creation of more complex data entry fields (which couldn't be achieved with SharePoint lists only) and also to use JavaScript and AJAX to develop "pedagogical support content" side bar which would dynamically change its content.

5.1 How was it achieved?

The first step was to create Visual Studio solution and create SharePoint specific folders structure within it (using WSPBuilder tool). A "mock" class descriptor form was developed as an application page and a ClassDescriptorForm.aspx file was created within the LAYOUTS/PIP directory under SharePopintRoot folder. The "front-end" solution structure is shown in the screen snippet below:



As seen in the snippet, additionally to the ClassDescriptorForm.aspx file there is a ClassDescriptorForm.js file which contains all JavaScript code needed for the purpose of user interface demonstrator. In actual fact, almost all of the user interface logic was written in Java Script and jQuery for the purpose of the demonstrator.

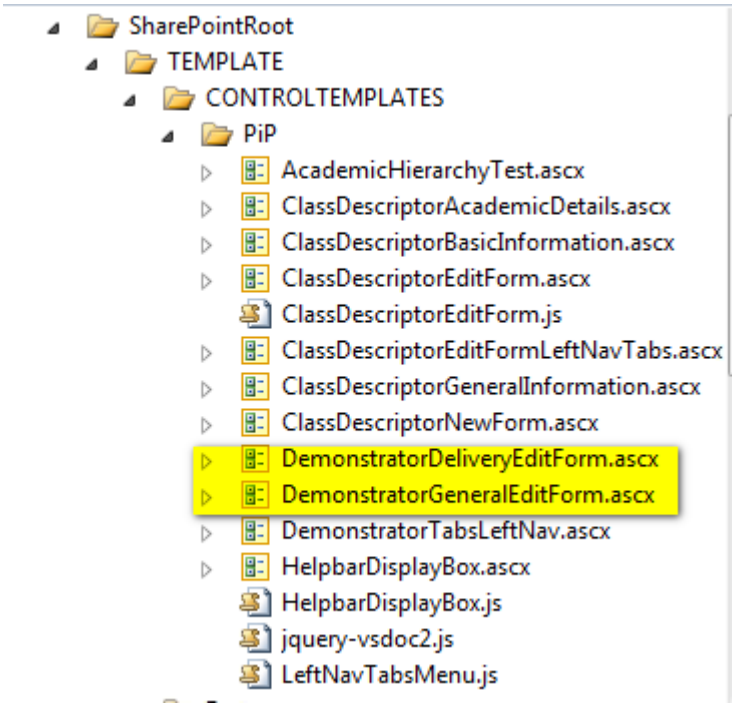
Within ClassDescriptorForm.aspx page appropriate references to various assemblies had to be included (e.g. to Microsoft.SharePoint). As mentioned earlier, the form was split into several user controls, so appropriate user controls had to be registered, too. As the page had to run in a

SharePoint context, the SharePoint's master page file (application.master) had to be used in the Page directive. Using master page file means that all actual content must be placed within proper content placeholders (e.g. `PlaceHolderLeftNavBar` or `PlaceHolderMain`). Pages also need to link to custom PiP.css files, mentioned earlier *.js script file and register ScriptManager ASP.NET AJAX control which references web service developed to return data for dynamically displayed tooltips (later on this).

```
<link href="../1033/styles/PiP.css" rel="stylesheet" type="text/css" />
<script type="text/javascript" src="ClassDescriptorForm.js"></script>
<asp:ScriptManager ID="ScriptManager1" runat="server">
  <Services>
    <asp:ServiceReference                    Path="/_pip_services/TooltipsAjaxService.asmx"
InlineScript="true" />
  </Services>
</asp:ScriptManager>
```

Requirements were to display pedagogical support content dynamically depending on which section of the form the user was working on. To be able to figure out on which section the user is currently working on, the form had to be structured with appropriate html tags. For the purpose of demonstrator two sections were developed in separate user controls:

- DemonstartorGeneralEditForm.ascx
- DemonstratorDeliveryEditForm.ascx



Each user control was a logical section of the form. To indicate logical form section in the markup, each control contains top level html element with appropriate CSS class and id. For example:

```
<div class="form-section tab-container" id="divGeneralInformationSection">
```

In the mark-up above you can see that the section is given a class named "form-section" and id `divGeneralInformationSection`. This means that all content within a given element is a particular form section whose ID is `divGeneralInformationSection`. There is also a "tab-container"

CSS class assigned, which was used to flag that given element (i.e. form section) is a content of the tab (the form was split into several tabs using jQuery).

Additionally each section could contain several “parts” identified by the elements too:

```
<div class="form-part" id="divModuleTitleAndTopicPart">
```

This convention allowed the use of the jQuery “focus” function to identify which section and part of the form the user was working on and dynamically display pedagogical support content (code contained within HelpBarDisplayBox.js file):

```
$('document').ready(function () {
    $('.form-section *').focus(function (e) {
        var partID = $(this).parents('div.form-part').attr('id');
        var sectionID = $(this).parents('div.form-section').attr('id');
        DisplayFormPartTooltips(partID);
        DisplayFormSectionTooltips(sectionID);
    });
    SetAllTooltipsContent();
});
```

5.2 How were data for tooltips stored?

Data for tooltips had to be maintained somewhere and, additionally, hooked into the class descriptor form sections and parts somehow. As it was SharePoint based solution and data for tooltips did not require any relational database features, the choice was made to store this data within a SharePoint list. SharePoint lists allow “Rich Text Box” fields which can have full HTML mark-up stored within them. This “Rich Text Box” field could be edited easily by non-technical users within the browser and displays exactly in the same way as it displays on the class descriptor form later on.

The content of “tooltips” could be retrieved from SharePoint list and used directly within the `HelpBarDisplayForma.ascx` control (purpose of this control is to display pedagogical support content in the “help bar box” which is placed on the right hand side of the page). To hook tooltips to appropriate sections and parts on the form it was enough to create additional columns within the “Tooltips” list which would store section and part IDs. The screen snippet below shows an extract from “Tooltips” list maintained within SharePoint.

General Information	lblModuleTitle	Field Tooltip	Please enter module title as it will appear in class catalogue and on student transcripts
General Information	lblModulePrincipalTopic	Field Tooltip	Please consult *Planning Office* for guidance
General Information	divGeneralInformationSection	Section Tooltip	**General information section** New classes and revisions to existing classes can only be introduced following approval by Ordinanc required for a new class proposal is similar to but less extensive than that required for a new course For more information and resources on class design guidlines use links below: University Procedure and Guidelines on Course and Class Approval
General Information	divLearningInfoPart	Part Tooltip	**Delivery Information** Please provide information related to module teaching and delivery. This includes: - credit value of the module - module length - academic year from which module will start to be taught - academic level - list of modules this module will replace (if any) The class must be assigned a level and credit rating compatible with the Scottish Credit and Qualifica University Procedure and Guidelines on Course and Class Approval (page 12 and Annex1) Scottish Credit and Qualifications Framework

Notice that there are three types of tooltips stored within the tooltips maintenance list:

- Section Tooltip – displayed on the right hand side box and refers to logical section of the form
- Part Tooltip – displayed on the right hand side and refers to particular part of the form section
- Field Tooltip – tooltip displayed below the actual question on the form

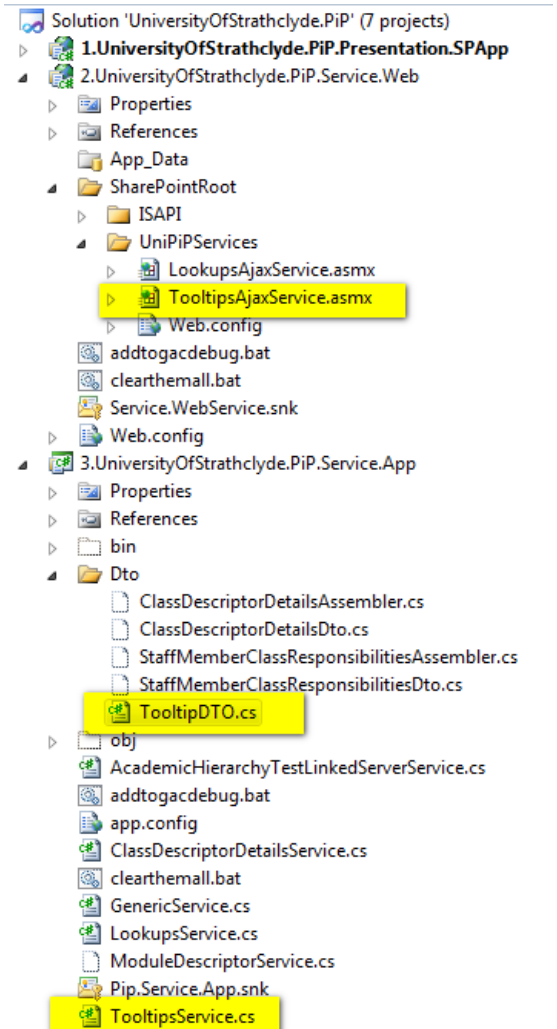
Those tooltips are retrieved and displayed on the mock class descriptor form. As this data is stored in a SharePoint list it allows non-technical users to maintain its content easily.

5.3 How is data of "tooltip" content is retrieved from server to client form?

A web service called by AJAX code on the form was used to retrieve data from the server side to the client browser.

First, the server-side service code querying "Tooltips" list had to be developed. Then results of this code had to be returned to the client through a JSON-based web service.

In the screen snippet solution (right) named `3.UniversityOfStrathclyde.PiP.Service.App` contains `TooltipsService.cs` file which contains `TooltipsService` class which exposes `GetTooltipsByIDs` method. This method's input parameter is a collection of IDs of sections and parts and also IDs of controls under which "inline" tooltips should be displayed. Logic within the `GetTooltipsByIDs` method queries data from "Tooltips" SharePoint list and return collection of `TooltipDto` objects as a result. `TooltipDto` object contain all data needed to hook "tooltip" to appropriate section or part on the form.



Solution named `2.UniversityOfStrathclyde.PiP.Service.Web` is just a wrapper for the actual service described above. It exposes its GetTooltipsByIDs' method result (TooltipDto objects collection) as a JSON-based web service. This web service may be called from JavaScript client code using AJAX. The web service is designed to be deployed within SharePoint context. There are differences between developing web services which must be deployed within SharePoint context and standard ASP.NET web service, however it is not within the scope of this document to explain them.

Finally, the web service could be called from class descriptor form JavaScript code (using ASP.NET AJAX approach) and further logic could be used to display it in appropriate context.

The function below first gets all IDs of sections, parts and "inline" tooltip controls and saves them within the array. This array is then passed as an argument to the web service method `UniversityOfStrathclyde.PiP.Service.Web.TooltipsAjaxService.GetTooltipsByIDs`. Returned data is stored within the global page variable: "tooltips". This variable is used later to display tooltips data dynamically within an appropriate context.

```
function SetAllTooltipsContent() {
    // get IDs of all controls which have tooltips
    var i = 0;
    var IDs = new Array();

    $('.form-label,.has-tooltip-below,.form-part,.form-section').each(function () {

        IDs[i] = $(this).attr('id');
        i++;

    });


UniversityOfStrathclyde.PiP.Service.Web.TooltipsAjaxService.GetTooltipsByIDs(IDs,function(t
ooltips){
            $(tooltips).each(function () {

            var fields = "#" + this.key + ".has-tooltip-below";

            $(fields).after('<br /><span class="form-field-tooltip" style="display:none">'
+ this.content + '</span>');

        });

        $(document).data('tooltips', tooltips);

    });
```

Additionally, jQuery was used to develop and demonstrate other possible functionalities of "dynamic" class descriptor forms, e.g:

- Content of sections displayed within "tabs"
- Mock "repeating sections"
- Multi-select dropdowns
- Various user interface effects

The demonstrator was presented on the PiP Steering Group meeting on 02 February 2010 and was received favourably, attracting much positive feedback. It initiated an interesting discussion on what

the potential system could achieve and improve within the institution. The aim of stimulating discussion around system specifications was therefore achieved by developing a demonstrator.

6. Investigating other SharePoint development opportunities in context of PiP project

As mentioned earlier, the demonstrator was not a fully functioning system and its purpose was only to attempt to encourage potential users and stakeholders to participate within the design and development process. After a successful presentation of the demonstrator it was decided that further research on SharePoint development opportunities should be performed in the context of the PiP project. As the project team was not particularly experienced in SharePoint development the research mainly consisted of:

1. Reading professional books, articles, documentation and blog posts about SharePoint development and customisation
2. Performing actual development to try various development techniques and paths

This was a research type of development activity and there was no formal software specification provided beforehand. Additionally, "requirements" and ideas of what system could do were not clear and were continually changing. This led to various development paths, some of which were abandoned over the time. Obviously, knowledge gained during this "research development" process was very valuable, demonstrating potential to be used within other University SharePoint-based projects.

At the same time when various development approaches were investigated, the team continued to gather requirements and identify potential uses of the system within the University. Most important and frequently repeated on various meetings requirement was the need to integrate potential class and course design and approval system with existing corporate systems and data from Oracle databases. This non-functional requirement was taken into consideration and design decision allowing easy integration of data in future was made. This will be described in more details within this document later on.

In this section the document we will briefly explain what SharePoint development technologies and approaches were investigated as possible ways to develop class and course design and approval system.

6.1 InfoPath forms

A part of Microsoft Office SharePoint Server 2007 is InfoPath Forms Server. Briefly, this technology allows the publication of InfoPath Forms within SharePoint Server and for these forms to be exposed to users through the browser. Data gathered through web based InfoPath forms may then be easily stored within SharePoint lists. As with every technology there are advantages and drawbacks to using InfoPath Forms versus standard ASP.NET forms. This was investigated in the context of the PiP system needs.

The main advantage of InfoPath Forms is that they can be rapidly designed by non-developers (business power users with an understanding of the SharePoint framework). InfoPath is part of the Microsoft Office suite and designing forms is generally not difficult to power business users lacking development skills. InfoPath Designer provides pre-defined data input controls which can be dragged and dropped into the form. Data gathered within the form is structured and stored as an XML data structure. Additionally, SharePoint stores InfoPath data within SharePoint lists. Lists introduce a limitation on the data complexity as SharePoint lists cannot maintain complex data relationships. A

common approach to overcoming this is to store only core, simple data within SharePoint list and any other, more complex data within the form's XML file. It is worth mentioning that many of the controls provided by the InfoPath Designer application cannot be used on forms which are then intended to be served via a web browser.

Data gathered on InfoPath forms can be pushed directly into relational databases or web services too. But this is something requiring development skill and, depending on the complexity of solution, it may occur to be more difficult and require more work than developing a standard ASP.NET web application. InfoPath provides a visual configuration wizard, allowing the creation of data connections to various data stores, including relational databases. If the solution is simply to deliver a data gathering form and data must be stored within relational database, and tables and columns from the database must directly map to the data gathered on the form, then using InfoPath may be a good approach. The main advantage in such a case in comparison to ASP.NET would be increased productivity. A simple InfoPath form may be developed quicker than a simple ASP.NET form. The difference in productivity decreases when the complexity of the form grows.

However, if the solution requires business logic to be applied before storing data within the database then developing standard ASP.NET applications would probably provide a better approach. The reason is that writing code behind an InfoPath form is more difficult and gives less flexibility in comparison to ASP.NET forms. This is because InfoPath is not optimised to serve as a User Interface layer within an n-layered application (even though it is possible to achieve). InfoPath's principal purpose is rather to provide replacement for various business data forms which would traditionally be gathered within MS Word documents or on paper forms. The main advantage of InfoPath in such cases is that it stores data as an XML data structure, which greatly improves data reporting and search in comparison to Word-based forms. On the other hand, reporting is obviously less powerful than reporting capabilities of relational databases.

In general, the conclusion from the research and testing conducted is that InfoPath forms are best suited as a replacement for paper based or Word forms, and when data does not need to be stored in a normalised and/or constrained way, as within relational databases. Although InfoPath may be used as a user interface for underlying application (which, e.g. stores data within relational database), developers do recommend this. Developers who decide to use InfoPath as a user interface for their underlying application may end up with much more work, effort and frustration than if they went with standard ASP.NET forms approach (even if ASP.NET forms must be used within SharePoint context which imposes other limitations).

The advantage of InfoPath forms are that they may be designed by business users without "hard" development skills. But this may be a disadvantage, too, especially if trying to deploy forms created by various business users as a User Interface layer for your application. Business users usually do not care about proper naming convention of controls and data sources – they just drag and drop controls on the screen. Therefore, the blanket assumption that no development experience is required to maximise productivity or impact with InfoPath is not entirely secure, e.g. a typical business user may encountered difficulties trying to bind InfoPath controls into data objects and database within the application.

If, for any reason, one is required to write managed code against an InfoPath form it occurs to be more difficult than writing managed code behind standard ASP.NET pages. InfoPath forms may occur to be great for quick prototyping of user interfaces, though; however, controls available within InfoPath Designer are limited in comparison to standard ASP.NET controls. There are many more controls available within ASP.NET. Additionally, within ASP.NET it is possible to create your own controls to re-use them or use third party ones. Programmability of InfoPath forms is therefore much more limited in comparison to ASP.NET.

Proponents of InfoPath forms have noted that it is extremely quick to build a User Interface and bind data for simple forms. On the other hand, performance of InfoPath with many Data Sources is extremely poor in comparison to a well-designed ASP.NET application. InfoPath is therefore not recommended for heavily data centric solutions.

InfoPath forms do not permit the creation of "wizard like" forms (it may be overcome by creating views). By contrast, it is extremely easy to create such "wizard" forms in ASP.NET.

One of the selling points of InfoPath is that it integrates well with SharePoint lists, workflows, versioning, and security trimming. This is a real advantage if organisations or companies have SharePoint infrastructure already in place. However, InfoPath forms based solutions tend to require workarounds to get simple things working. In general, working with InfoPath means lots of tinkering-like work, with many options to select or unselect, hidden under various menus and submenus. Documenting of all those various steps and wizard screens is more difficult than documenting written code.

InfoPath's advantage is that publishing it on SharePoint is quite a simple task. But it is true only if publishing / deploying it into single environment manually. Automated deployment and updates into multiple environments can be problematic. This is a common requirement for any larger solution and is fairly simple to achieve in ASP.NET.

Another problem with InfoPath is proper version control. All InfoPath components are packaged within single *.xsn file (which is actually a *cab file with different extension). This could be versioned within some revision control system but it would be impossible to track actual changes within underlying code generated by InfoPath (which is saved in many generated files packaged within *xsn file). To do so, one could extract all individual files from *.xsn file and version control them separately. This would be a manual task and re-packaging back all files into *xsn file has to be manually repeated each time before publishing. But this is not a main problem with versioning of InfoPath forms. InfoPath forms are usually deployed directly on the InfoPath Server. When any change needs to be done to the template (e.g. quick bug fix) it is tempting for the power user or developer to just edit the template and save it back directly on SharePoint without recording changes made within a proper revision control system (which as mentioned earlier would be more difficult in comparison to ASP.NET, anyway). This may easily lead to a so called "maintenance nightmare".

Another drawback of InfoPath forms is that you cannot use JavaScript to develop User Interface logic. This only possible through a VBScript which is more limited in comparison to JavaScript. Where JavaScript cannot be used, neither can existing popular and powerful JavaScript libraries, like jQuery.

After research and testing of InfoPath forms was conducted it was decided to not use InfoPath forms for the PiP prototype system at that point. This decision was made mainly because of various ambitious expectations and requirements existing at that time. There were two main reasons:

- Non-functional requirement to integrate data gathered within the system with existing corporate Oracle databases. Using InfoPath forms and store data either in SharePoint lists or within XML files makes it more difficult (but not impossible) to synchronise with central Oracle databases used as main data store for corporate University systems. This would be especially difficult if XML data files were scattered around many SharePoint sites.
- Functional requirement to create dynamic, rich web based forms (Web 2.0 style which is mainly achieved through JavaScript and jQuery – both technologies can't be used on InfoPath forms). Displaying dynamic support in similar way as it was done within User Interface demonstrator would not be possible to achieve with InfoPath forms.

6.2 Custom Field Types

Another SharePoint specific approach investigated to be used for class and course descriptor forms was SharePoint “Custom Field Types”.

As mentioned earlier SharePoint stores its data within lists. Lists are similar to a database table. It is a means of storing tabular data. It has columns and rows, too. Each row represents particular list item and each column represents a particular piece of data on a given list item. In SharePoint list rows are called list items and columns are called fields. You can store various types of data within list item fields. SharePoint provides a set of standard field types which can be easily used when creating new lists.

“Single line of text” standard field type can be used for any simple textual data (e.g. class title in class descriptor form). “Multiple line of text” standard field type can be used to store larger textual information which additionally can be stored as an html enhanced rich text which may contain pictures, tables, hyperlinks and other rich text formatting. “Number” standard field type allows store numerical information. There are also more sophisticated field types, like “Choice” field type which displays choice style control to input data (like dropdown list, checkbox list, radio button list). Or “Picture” field type, or “Hyperlink” field type.

From technical point of view SharePoint field type is data stored within a SharePoint list column and rendering ASP.NET control displays this data within intended format.

SharePoint introduces the ability to work at this technical level, where developers can actually define their own custom field types. By using SharePoint’s API and special development techniques, developers can extend SharePoint by developing and registering their own so-called “custom field types”. Those custom field types may then be re-used at any lists as their columns.

A custom field type represents a new data type for columns. Custom field types need to be developed within .NET managed code, where the developer will provide logic to store the gathered data within the SharePoint list column in an appropriate format. Additionally, a custom field type is also defined in terms of one or more ASP.NET server-side controls that give developers the ability to define how to render custom field type data to the user. Additionally it gives developer an opportunity to leverage standard ASP.NET development techniques (e.g. one could provide logic to push data from custom field types into a database or web service).

How can custom field types be used within the PiP project context? Let us take an example of the “learning outcomes” section of the form. As mentioned earlier, we could develop a “repeating section” style control. Each section would capture data for one learning outcome. Each learning outcome could contain several data entry fields to capture relevant information (e.g. learning outcome name, learning outcome objective, performance criteria, assessment method, etc.). Some of the fields within each learning outcome repeating section could be free text fields but others could provide constrained sets of answers to questions (e.g. in the checkbox list or multi-select dropdown list) to improve standardisation and consistency of descriptors. A repeating section would allow user to add or remove learning outcomes as required.

Such “learning outcomes” repeating section controls would be possible to develop with standard ASP.NET development techniques. When developing custom field types developers uses ASP.NET development techniques to develop rendering control for custom field type. So, from technical point of view, it would be possible to develop a “learning outcomes” custom field type which would render as a repeating section ASP.NET control (JavaScript and jQuery could also be used). This “learning outcomes” custom field type could be re-used in any SharePoint list.

But what about storing data captured within our "learning outcomes" custom field type? Data captured in our repeating section control would be a bit complex and would traditionally be stored within several normalized tables within a relational database. As mentioned earlier SharePoint lists are not designed to store relational data. By default, data captured within list field should be stored within single column of this list corresponding to this field. SharePoint provides API and development techniques to store more complex field data within list columns. When a custom field type is created it is derived from existing parent custom field type class. If storing more complex data structures for a custom field type is required it will usually inherit the custom field type class from the SPFieldMultiColumn class. This class allows data to be stored in the list field containing multiple values. From a technical point of view, the class contains a single string value, in which the values of the various "columns" from the field are separated by special delimiter characters. But what if your custom field type data structure is complex enough that even using SPFieldMultiColumn class is too limiting to use? In such cases using the so called Custom Field Value Class is a solution.

If you create a custom field type that requires a special data structure for the field data, and this structure is not supported by the parent field class from which you are deriving your custom field class, you can create a field value class to contain your custom field data. For example, it is possible to write custom field value class within which its logic would serialise the data into the XML data structure and save it as a string within a corresponding SharePoint list column. Or even logic could be written to push custom field type data into a relational database or web service.

Within our prototype class descriptor form various sections of the form could be developed as custom field types. Then those custom field types could be deployed on the SharePoint farm and could be used, together with standard SharePoint field types, to build class descriptor forms as SharePoint lists. Such an approach at the first sight seemed to have many advantages – e.g. users could build their descriptor forms by themselves by re-using those developed earlier, self-contained components which would correspond to typical sections on the form. This would also allow creation of variations of class descriptors, depending on the requirements of particular faculties and/or department.

However, this approach would also have many disadvantages, outweighing the advantages, and a decision was made to not use "custom field types" approach:

1. Developing custom field types is technically complicated and time consuming task. It would mean huge initial time investment to develop all the possible sections of the descriptor forms as custom field types. Additionally, any changes to custom field types would be a time consuming development task, too. It did not therefore appear to be an ideal approach for a research / development project where requirements and ideas were constantly changing. In essence, the technical approach was not flexible enough to respond quickly to development requirements changes and was simply not worth the effort.
2. Standard data store mechanisms of custom field types has many limitations. Using SPFieldMultiColumn class to store "multi value" data within list column as delimited text is not a stable or flexible approach to storing complex data structures. Especially, when those data structures must maintain relationships and should be easy to query for reporting purposes, as well as being easy to integrate with the corporate Oracle relational database.
3. Developing custom field value type to store data in a database is technically possible to achieve but it is a complex and time consuming task and prone to bugs and errors (at least in comparison to standard .NET development techniques).
4. Additionally, we wanted to treat descriptor forms as single entities, where various sections would be related to each other. Developing separate data persistence logic for each separate section of the form would be counter intuitive and would make maintaining proper data relationships very difficult (if not impossible). Custom field types are really intended to be an extension mechanism for SharePoint lists to build field types which allow to store and render custom data structures. Custom field types should be treated as self-contained components

and any attempts to maintain relationships between them are not worth the effort (and additionally error prone).

7. .NET n-layered web application within SharePoint context

Results of research on available SharePoint development approaches (e.g. InfoPath forms, custom field types) taken together with prototype system requirements at that point of time (both functional and non-functional) led to the conclusion that developing standard .NET n-layered web application would be the most flexible solution. However, we still wanted to use some SharePoint "out-of-the-box" functionalities (e.g. integrated authentication and authorisation, user interface, integration with workflows, document management and collaboration functionalities). The goal was to design a technical solution which would leverage flexibility and extensibility of standard n-layered application and additionally would work within a SharePoint context using SharePoint's "out-of-the-box" functionalities.

7.1 N-layered application architecture

Traditionally, database-driven applications consist of data storage (relational database) and application logic built on top.

Application logic parts of software will usually consist of three main logical parts:

- Data Access
- Business
- Presentation

Data access is responsible for any data-related operations, i.e. data retrieval from and persistence to the database.

Business is the core part of the application, where business domain is modelled and any business operations, transactions, workflows, rules and validations are implemented. All those business operations work around data retrieved through the data access code.

Presentation's responsibility is to present business layer operation results to the user and also to gather users' input through a User Interface (e.g. web site, windows application, mobile application, command line).

This is very simplified model of software architecture. In reality, all of those three core parts of the software application may be built from other sub-components. For example, presentation may be split into User Interface Components (i.e. user interface controls, elements which will be shown to an end user) and presentation logic (which will make sure that output and input data is properly processed and prepared for display by User Interface Components).

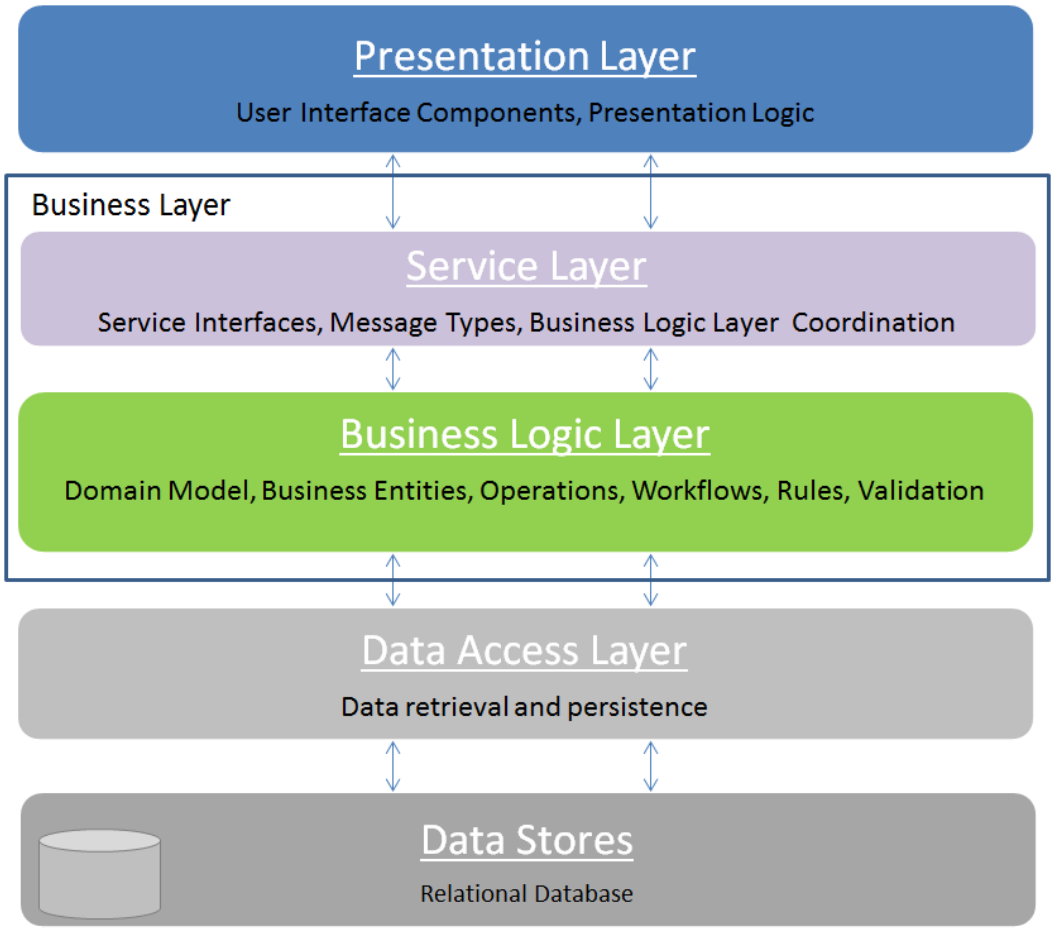
Business parts will usually be built from so-called domain models (which models the business domain in a given object-oriented language using so-called business entities), business logic (which will implement all business operations, workflows, rules and validation) and (optionally) service layer (which would coordinate and expose all underlying business logic operations in a consistent way through well-defined, coarse-grained interfaces)

All those parts of application communicate with each other, usually in the following order: Relational Database -><- Data Access -><- Business -><- Presentation

It is left to the developer / application architect how all those logical parts of the application will be structured within the code. Obviously, it is possible to mix them up all together within a single bunch of code (e.g. data access code written within code behind of ASP.NET *.aspx page). This may work fine for small solutions but mixing up all these logical parts together is not a very scalable and flexible approach, is a bad design practice, and may quickly lead to a "maintenance nightmare" once the application grows and changes arise from stakeholders.

The best practice is therefore to separate those main parts of the application from each other by using so called layering. This means that each logical part is built as a self-contained component, exposing well defined interfaces to communicate with other components. This architecture is called an n-layered architecture. In most cases n will equal three but there may be more layers, too. For example, a service layer may be treated as a separate layer and then n will be equal to four. A business layer will consist of a service layer and a business logic layer then. A business logic layer and service layer should be treated as one logical business layer, though.

From the high level perspective it is worth summarising n-layered architecture by the following diagram (which is still a very simplified version of what one would expect in a real enterprise application):



An ideally designed n-layered architecture each layer should be aware of and communicate only with the layer existing one level below. Additionally, communication should be done through well-defined interface contracts. This can be achieved by using interface-based programming. This is not always the case in practice, obviously - any design decisions depend on various factors like requirements, complexity of the application, time available, etc.

Using all of the layers in each application is not compulsory. For simple applications, such as websites, serving as input-output interface for underlying database only, it may be more appropriate for presentation layer to communicate directly with data access layer or even database without the overhead needed to design and develop middle business layer (business logic layer plus service layer).

However, if the application is complex enough, with business rules, workflows, validations and operations which need to be coded with logic properly, it makes much more sense to invest initial design and development time to build a proper n-layered architecture of the application. It pays back in the longer term as n-layered architecture has many advantages:

- Separation of concerns – separating each logical part of the application allows one to replace them if needed without touching the other layers. For example, if there is a need to replace an SQL Server database with an Oracle database it would require only (apart from migrating databases) a change to the data access layer code, without touching other layers at all. Additionally, if specific patterns and tools were used within the data access layer (e.g. Object-Relational-Mapping) it would enable changes to several configuration settings without the need to re-develop the data access layer code. If the application was not properly designed, with connection strings and other database vendor specific settings scattered around the code, with column names from database tables weakly referenced from any part of the code, it would be a much more difficult (and frustrating) task.
- Main business logic of the application is included in the business layer (business logic layer plus service layer) only. That means that there is no business logic in either presentation (like in case of InfoPath forms) or data access layers (like in applications written purely in database languages, e.g. T-SQL, PL/SQL). This makes the application design cleaner and maintenance far easier. In the business layer there are no technical details relevant for specific presentation or data access and database technology. There is also no data access logic embedded within presentation layer (as it is in the case of InfoPath forms).
- N-layered architecture allows having many presentation layers communicating with the same underlying business layer (through interfaces defined within a services layer). So, for example, it is possible to develop a single business logic and data access layer on top of a relational database and then present data in various ways, depending on specific needs (e.g. website, windows application, mobile application, command line tool or … SharePoint web parts).
- N-layered applications are much more flexible if it comes to responding to change requests. It makes implementation of future changes easier and quicker.

Each layer may be built from other smaller components. All those components will play some role within the given layer and will be built by using design and architectural patterns. It is not within the scope of this document to write about all these patterns but it is worth briefly mentioning two of them:

7.2 Domain Model pattern

Applications are being developed to solve specific problems within specific business domain. A business domain will consist of a set of business entities. For example, a class and course approval system business domain might include following business entities: class descriptor entity, course descriptor entity, lecturer entity, faculty entity, department entity, university entity, etc. All those entities are related to each other. A course will be built from a collection of classes, a class or course may be delivered by given department which will be a part of given faculty which belongs to given university (or other higher education institution). This is obviously a very simplified model of a business domain but it already shows quite complex dependencies and associations which may exist between business entities.

If it comes to software development, those business entities are best modelled with object-oriented design using one of the object-oriented programming languages. This entails modelling a business domain by designing business entities (objects) class definitions which correspond to real business domain entities. So, for example, one will design CourseDescriptor class which will model a real course descriptor business entity. Such a CourseDescriptor business entity class will contain properties to store data describing given business entity (course descriptor in this case). Those properties may be simple data structures (e.g. CourseTitle, DescriptionForCourseCatalog, etc.) or may actually refer to other business entities modelled in the domain model (e.g. CourseDescriptor business entity may contain a collection of ClassDescriptor entities. And the ClassDescriptor entity may contain a collection of LearningOutcome entities, and so on).

Additionally, business entities may contain methods which will correspond to real life operations related to a given business entity. Object oriented design guidelines say that a given business entity should contain only methods which will operate on data belonging to this entity and entities referred in a given business entity (to maintain encapsulation). If there are operations which work across several business entities they should be implemented outside the core domain model entities in separate business logic classes or service layer classes. Business entities will contain business validation methods, too.

All those business entities, theirs properties, methods, validations, dependencies and associations are best modelled using object-oriented design. Obviously, complex business domains are not easy to model and any implementation should be preceded with thorough business analysis and design processes. It is worth pointing out that for simple applications, choosing a domain model approach may be an unnecessary design overhead and simpler patterns could be used (e.g. Active Record pattern, Transaction Script pattern, Table Module pattern).

7.3 Data Mapper pattern and Object Relational Mapping tools

As mentioned before, an object-oriented design paradigm is best suited to model complex business domains. However, a standard relational database is still the best and most commonly used way to store data generated by an application. An object-oriented world is different to a relational one. Objects and relational databases have different mechanisms for structuring data. Many features of objects, such as collections are not available in relational databases. An object schema of a business domain model and a relational schema of database do not match up. However, there is still the need to transfer data between a database and a domain model, and this data transfer mechanism becomes a complex logic on its own. Additionally it is purely technical logic, so-called plumbing code, not solving any business problems. It would be ideal to minimise the amount of code needed to write here and additionally make sure that this code is written only once and then reused across an application. It is here where Data mapper pattern and object relational mapping tools are helpful.

Data mapper is a component of software, which resides in data access layer and separates domain model objects from the database. The data mapper's responsibility is to transfer and map data between relational databases and domain model business entities. Writing data mappers manually is quite difficult and a laborious task. Fortunately, the pattern was recognised by developers and there exist many implementations of it (open source or commercial) which can be reused in applications with much less effort than writing a data mapper. These implementations are called "object relational mapping tools".

7.4 N-layered architecture, domain model and data mapper pattern in the context of class and course design and approval prototype system

How did all these patterns fit into initial class and course design and approval prototype system architecture and what were benefits of using them?

As mentioned earlier PiP project prototype system did not have a detailed requirements specification and, to complicate the scenario, what requirements or ideas were specified were changing continuously. For example, the project team gathered various paper based class descriptor forms from departments across the University, each of which varied significantly. The team tried to pick up common sections and question from those forms to build the unified form for the purpose of the prototype system. The team were aware, though, that it was very likely that the form (and corresponding data structures) would be changing frequently during the development lifecycle and if a system was to be used within the University, the forms (both class and course descriptors) sections and questions would very likely change after initial deployment. The therefore team identified that the prototype system architecture should be flexible enough to allow easy and quick implementation of possible changes to the system in the future. Choosing a flexible n-layered architecture with associated patterns seemed to be the best approach. If properly designed at the beginning of development, such architecture would allow implementing changes fairly easily and quickly. Flexibility of the system would be much higher, which was particularly important in a research-type development project like PiP, with continuously changing requirements and development paths.

Additionally, a requirement to maintain department / faculty specific variations of forms (and in the same time gather core information relevant to all forms across the University) was likely to appear in the further stages of development. Having flexible n-layered architecture in place would allow the team to add such variations to the initial forms structure and the domain model by using object-oriented design principles and design patterns.

Other non-functional requirements which were being frequently repeated at many project meetings, was to build the system in a way which would make it easy to integrate with existing corporate data structures held in Oracle databases. As mentioned earlier, using data mapper pattern and an appropriate object-relational mapping tool would make it easy to switch between SQL Server to Oracle database if needed. This would require only replicating data tables' structure within new a database, migrating data (if needed) and making several minor changes to the object-relational mapping tool settings within data access layer. Business and presentation layers would remain untouched if nothing changed in the domain model itself. Alternatively, proper interfaces could be exposed in the service layer (e.g. as web services) which would allow the PiP system to be queried for relevant data from external systems (e.g. corporate Oracle-based systems) in the Service Oriented Architecture (SOA) fashion.

Another important benefit of n-layered architecture is that it would allow the implementation of any identified business operations, workflows, rules and validations using object-oriented paradigm, which is much more powerful and flexible in comparison to procedural programming (e.g. in PL/SQL language). All these operations, workflows, rules and validations would be encapsulated within the business layer and operations relevant for the presentation layer (or other external systems) would be exposed through service layer interfaces.

Generally, building the system with flexible architecture and using appropriate patterns and practices would make the system flexible enough to implement changes in the future and would make it easy to integrate with other systems.

It is also important to not "over-engineer" a system's architecture at the very beginning of the development cycle. Development teams should not try to build "ideal" domain models at the very

beginning. Development should rather start from a simplified version of the domain model which will be re-factored later on. In practice, re-factoring of the domain model (and database structures, data access, business logic, service and presentation layers) is a continuous process during development cycle.

Therefore, in the class and course approval prototype system case, the team decided to start with a simplified class descriptor form and build a domain model and all other layers for this class descriptor form only.

7.5 Embedding n-layered architecture within SharePoint context

The question which arises is how to implement n-layered architecture within the context of SharePoint. Is it simple?

If we decide to use SharePoint as a presentation layer for underlying custom layers only and additionally use SharePoint out-of-the-box functionality for the purposes of the application, then it is not a very difficult task. User Interface components and logic for the underlying custom business layer may be simply embedded within SharePoint web parts. Then custom developed web parts could be used together with SharePoint out-of-the-box functionality (by manual customisation or by creating sites and lists definitions). We still could reuse custom developed underlying layers with different user interfaces (e.g. standard web site, windows or mobile application) if needed. But we wouldn't be able to use SharePoint-specific functionality in other contexts then.

The task becomes more difficult if we decide to use SharePoint-specific functionality within our business layer (so that we could expose it through a service layer to various User Interfaces and external systems). Hooking this SharePoint-specific functionality into the business layer somehow would be required. Using the SharePoint Object Model directly within our business layer would be required, thus introducing a dependence on SharePoint, obviously. To minimise this dependence one could separate SharePoint-specific logic into separate layers / components. For example, we could create a so-called "productivity layer" (name recommended by Microsoft) which would contain all SharePoint-specific logic. Or, we could treat SharePoint as an external system which exposes its functionality and data through its interfaces (SharePoint Object Model or SharePoint Web Services). Then we could create an additional component within the data access layer to have single point of communication with SharePoint.

All in all, if we create an application using SharePoint-specific functionality in its logic, it must be dependent on SharePoint in some way.

8. Prototype system technical framework based on n-layered architecture, domain model and data mapper patterns using Fluent NHibernate Object-Relational Mapping tool

Project team initially decided to develop a prototype technical design of the system using n-layered architecture with simplified domain model, service and data access layers and database tables' structure based on class descriptor form. Within the data access layer the object-relational mapping tool – Fluent NHibernate was explored and used as a data mapper.

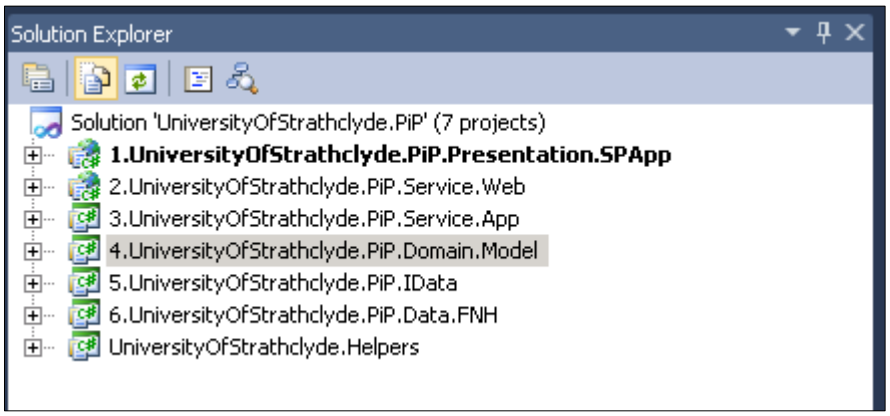
It is important to mention that Fluent NHibernate (http://fluentnhibernate.org/) is an extension to a popular ORM tool for .NET – NHibernate (http://nhforge.org). It still uses NHibernate assemblies inside, but extends it in the way that simplifies the mapping process. Traditionally, mapping between

the domain model classes and database tables in NHibernate is configured using XML configuration files. This approach has some disadvantages. Writing mappings in XML is error prone. Developers may easily make some errors within XML mapping file and it is not possible to discover this error during compilation. The error will appear only during runtime (which makes it more difficult to discover and fix it). Within Fluent NHibernate all mappings are done within .NET code (using C# 3.0 feature – Lambda Expressions) and the code will not compile if there are mapping errors. This is a big advantage. Additionally, Fluent NHibernate allows for convention-based mappings, i.e. mappings based on names of classes, their properties and names of database tables and their columns. This is particularly useful when developing new applications. Developer will make classes and their property names the same as the database tables and their column names. Then Fluent NHibernate will make mappings automatically based on naming conventions only. This is really powerful and makes the data access layer very thin and quick to develop, and very flexible and quick to respond to change requests. Additionally auto-mapping naming convention patterns may also be configured. If auto-mappings cannot be used for some reason (e.g for many-to-many relationships or when building domain model on top of legacy database) then manual mappings or even traditional XML-based mappings can be used and mixed together. There are other useful and advanced functionalities of Fluent NHibernate and NHibernate but this is not within the scope of this document.

8.1 Domain model

During this stage of the project the team focused on the technical and architectural aspects of the system to build an application framework which could be easily extended once proper requirements were gathered. The domain model shown below is therefore a simplified version of what could be designed and achieved in a real-life system. It is based on A simplified class descriptor form only.

The picture below shows the structure of the PiP solution (in Visual Studio 2010) which uses n-layered architecture and Domain-Driven Design approach.

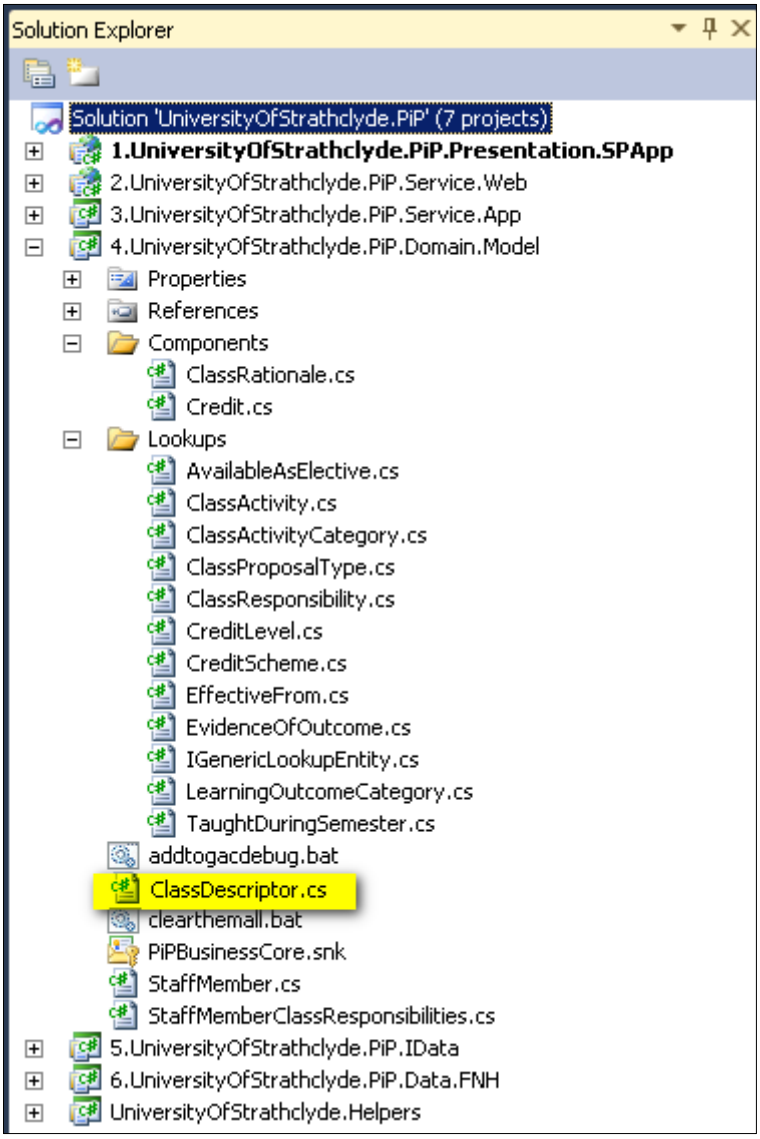


The solution consists of 7 sub-projects. Each sub-project corresponds to given layer / component of the application (solutions' names start with numbers to order them from top to bottom layers).

When developing A new application using domain-driven design approaches one starts building the domain model business entities within an object oriented language by building classes with properties which will hold data of entities. Project named `4.UniversityOfStrathclyde.PiP.Domain.Model` is where the domain model entities reside. The top level class descriptor object is defined within `ClassDescriptor` class (in `ClassDescriptor.cs` code file). In domain-driven design terms such top level object is called aggregate. Aggregate means that this is a cluster of associated objects. It usually references other smaller objects within itself. `ClassDescriptor` class will reference, e.g., all objects defined within code files in the "Lookups" and "Components" folders on the picture below.

Any external references to objects within aggregate object should be done through aggregate object itself, e.g. `ClassDescriptor.Credit.CreditScheme`.

Pictures below show code files structure of the domain model layer of the PiP prototype solution and also a definition of `ClassDescriptor` class. Please note that this class contains only properties to hold data. It could contain various methods corresponding to business operations, rules and validations relevant to class descriptor. To maintain encapsulation (object-oriented principle) those methods should operate only on ClassDescriptor object data only, like simple types of data (e.g. ClassTitle string) and referenced objects (e.g. ClassMemberClassResponsibilties object). Within domain model architectural patterns business entities should not contain any methods calling the data access layer directly. It should be executed outside those objects within separate classes, usually within the service layer.




```
public class ClassDescriptor
{

        public virtual int Id { get; set; }
        public virtual Guid ClassDescriptor_Guid { get; set; }
```

```
        public virtual string ClassCode { get; set; }

        public virtual DateTime Created { get; set; }
        public virtual string CreatedBy { get; set; }
        public virtual DateTime Modified { get; set; }
        public virtual string ModifiedBy { get; set; }
        public virtual bool IsDeleted { get; set; }

        public virtual string ClassTitle { get; set; }

        public virtual int VersionId { get; set; }
        public virtual string VersionLabel { get; set; }
        public virtual string VersionLevel { get; set; }

        public virtual string Faculty { get; set; }
        public virtual string Department { get; set; }
        public virtual ClassProposalType ClassProposalType { get; set; }
        public virtual string OldVersionClassDetails { get; set;}
        public virtual string SummaryDescription { get; set; }
        public virtual Credit Credit { get; set; }
        public virtual TaughtDuringSemester TaughtDuringSemester { get; set; }
        public virtual EffectiveFrom EffectiveFrom { get; set; }
        public virtual AvailableAsElective AvailableAsElective { get; set; }
        public virtual string PrerequisitesClasses { get; set; }
        public virtual IList<StaffMemberClassResponsibilities> StaffResponsibilities { get;
set; }
        public virtual string Rationale { get; set; }
        public virtual ClassRationale ClassRationale { get; set; }
        public virtual string ClassAims { get; set; }
        public virtual string TeachingAndLearningMethods { get; set; }
        public virtual string ClassContentStructure { get; set; }
        public virtual string ResourceImplications { get; set; }
        public virtual string Assessment { get; set; }

    }
```

8.2 Service Layer

The service layer resides on top of business logic layer and both layers build one business layer (as their logic is mostly related to real-life business processes). The main role of the service layer is to coordinate underlying business logic layer (i.e. business entities, operations, validations, workflows) and expose this coordinated logic to the presentation layer (or other external systems) through well-defined interfaces. The service layer communicates with the data access layer to get or save business entities from entities repositories.

Communication with service layer is based on defined messages. For example, the presentation layer will collect data from end user and then send this data through request message to the service layer. The service layer will perform relevant operations on the underlying business logic layer (which in turn will usually cause some calls to the database through the data access layer) and will return response messages to the presentation layer. Request messages carry all data necessary to perform the service layer operation (exposed through an interface) and response messages carry all the data which must be displayed to an end user.

Request and response messages are designed using Data Transfer Object pattern (DTO). In most cases DTOs are "flattened" versions of business entities from a domain model. Associations between domain model business entities may be quite complex. In such cases we may want to simplify and flatten those associations before sending data to the presentation layer. To achieve this we create data transfer objects (DTOs) and additional logic which performs mappings between the domain model business entities and DTOs. This requires an additional development overhead and for simpler applications it is acceptable to send and use domain model business entities within the presentation layer directly. Alternatively, if we were sure that most of the business layer operations and entities will be used within our application context only (e.g. only within the context of ASP.NET application) then it is acceptable to use domain model business entities directly in the presentation layer, too.

However, if we want to expose some of our service layer methods remotely (e.g. through web services), then we need to create DTOs at least only for those operations we want to expose. Because DTOs have a flattened, tree-like structure, they can be serialised and sent through the network (e.g. as an XML SOAP message). In such cases we need to build an additional thin layer of web services on top of the service layer which would forward calls between web service caller (it may be, for example, an AJAX call from the presentation layer) and the actual service layer method.

Service Layer methods will usually correspond to actual business use case operations. For example, in the PiP system we could have SubmitClassDescriptorForApproval(ClassDescriptorDTO dto) or ApproveCourseDescriptor(CourseDescriptorDTO dto) methods, etc. If necessary, we could create separate data transfer object for each service layer method. Such a DTO would carry all data necessary to perform given operation (e.g. for ApproveCourseDescriptor() method we would create data transfer object carrying not only data of course descriptor but also any additional relevant data needed to approve course descriptor at given stage).

During the development of class and course design and approval prototype system, the team mainly focused on the architectural and technical aspects of the system to build a technical framework for further extension. Therefore, the service layer was initially developed contains several basic CRUD operation methods (which are the most common operations used in real-life scenarios – Insert, Update, Delete and Get).

```
public int InsertClassDescriptor(ClassDescriptor classDescriptor)

public int UpdateClassDescriptor(ClassDescriptor classDescriptor)

public void DeleteClassDescriptor(ClassDescriptor classDescriptor)

public ClassDescriptor GetClassDescriptorById(int id)
```

The methods above are contained within the `ClassDescriptorService` class within the service layer of the solution (project named `3.UniversityOfStrathclyde.PiP.Service.App`). Please note that domain model business entities are used as parameters within service layer methods – this approach was chosen for the sake of simplicity at that stage. However, if needed, the code could be refactored and additional logic developed to use DTOs.

Service layer method logic calls the data access layer to get relevant business entities from the database. The service layer operates in an object-oriented world so it is not aware of underlying relational database schemas. All it knows is that within the data access layer there is a place which may be queried to get any needed business entities. This place in the data access layer is called Repository and is built based on Repository architectural pattern.

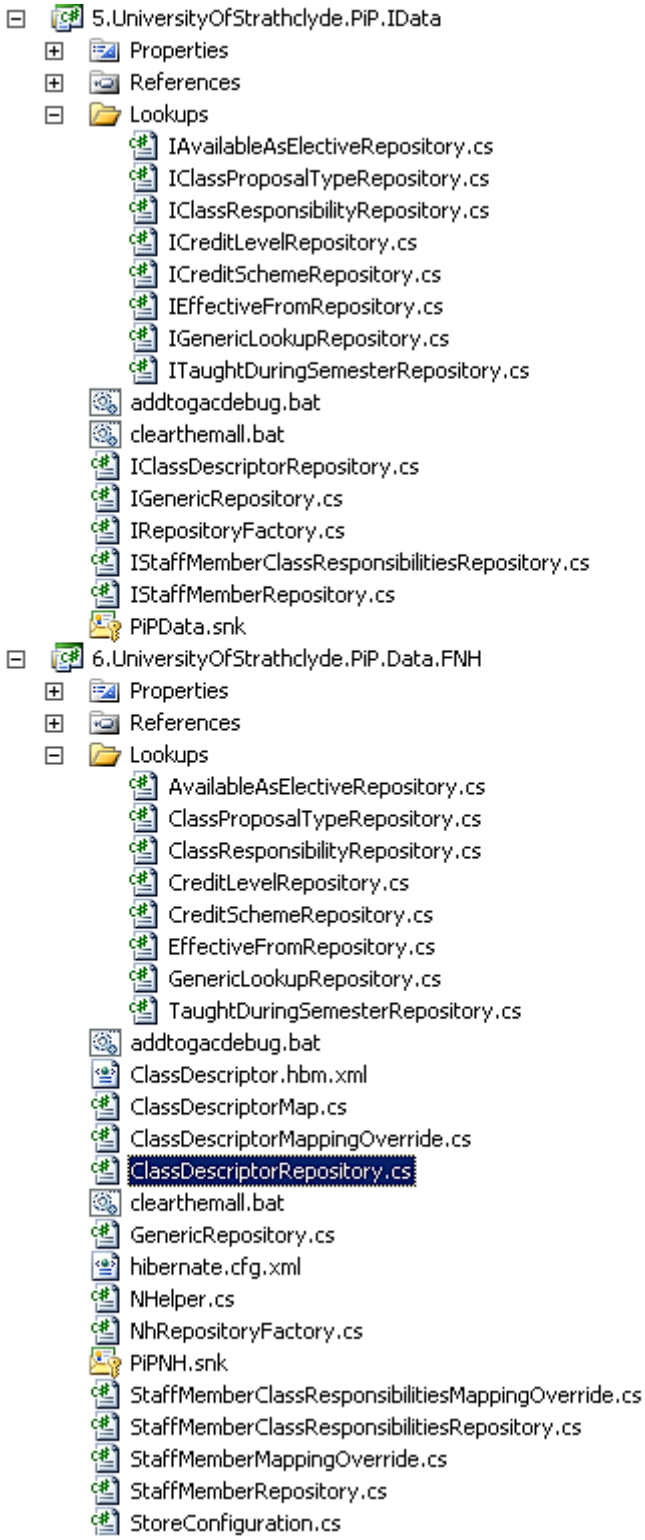
The following sections of this report will describe the technical framework of the data access layer implemented with the data mapper and repository patterns and using Fluent NHibernate Object-Relational Mapping tool. It will also show how to call data access layer repositories from the service layer.

8.3 Data Access Layer

The technical framework for data access layer (DAL) was developed using data mapper pattern implemented with the Fluent NHibernate Object-Relational Mapping tool. The repository pattern was used to provide a single point within the DAL to get and save business entities from and to the database. Interface-based programming was used to define DAL interfaces.

The project named `5.UniversityOfStrathclyde.PiP.IData` defines DAL interfaces. The project named `6.UniversityOfStrathclyde.PiP.Data.FNH` implements those interfaces and uses Fluent NHibernate tool for object-relational mapping and querying underlying SQL Server database. Thanks to this approach other layers calling DAL (e.g. service layer) are aware only of DAL interfaces - implementation details are hidden behind those interfaces. This allows replacing actual implementation of the DAL with new versions if needed (e.g. using different ORM tool or implementing Data Mapper pattern manually) without touching other layers at all.

Interface names start with a letter “I” (e.g. `IClassDescriptorRepository`) and their implementations have the letter “I” removed from class names: `ClassDescriptorRepository`. All classes names ending with “Repository” suffix are where the repository pattern is implemented for given business entities. `ClassDescriptorRepository` is a repository class for class descriptor business entities.



`GenericRepository` is an abstract base repository class which implements CRUD operations common for all business entities. It uses .NET generics so implemented logic may be re-used by concrete repositories inheriting from this base generic repository. As seen within the code snippet below `GenericRepository` uses `NHibernate Session` object to get and save entities from and to the database. All data store related logic (including SQL statements generation) is delivered by NHibernate ORM tool. No T-SQL stored procedures are needed to be written by developer as all T-SQL code is generated by NHibernate. All that is needed within a database are tables to store data.

This saves a lot of development and maintenance time and makes solution less prone to bugs. This is especially useful for large applications with dozens of business entities.

```
public abstract class GenericRepository<T,ID>:IGenericRepository<T,ID>
{
        private ISession _session;

        //NHibernate Session object
        public ISession Session
        {
            get
            {
                if (_session == null)
                {
                    _session = NHelper.GetCurrentSession();
                }
                return _session;
            }
            set
            {
                _session = value;
            }
        }

        // gets entity (of type T) by given id from the database
        public T GetById(ID id)
        {
            return Session.Get<T>(id);
        }

        //gets all entities (of type T) from the database
        public IList<T> GetAll()
        {
            return Session.CreateCriteria(typeof(T)).List<T>();
        }

        //saves entity (of type T) within the database
        public T MakePersistent(T entity)
        {
            Session.SaveOrUpdate(entity);
            return entity;
        }

        //deletes entity (of type T) form the database
        public void MakeTransient(T entity)
        {
            Session.Delete(entity);
        }
}
```

Concrete implementations of repository classes (e.g. `ClassDescriptorRepository`) inherit from `GenericRepository` class. T generic parameter should be replaced with concrete business entity type (e.g. `ClassDescriptor`). ID generic parameter should be replaced with type of identifier of given business entity (e.g. integer). Within concrete implementations the developer may add additional, more complicated queries to the database, based on specific criteria needed for specific scenarios. However, if only basic CRUD operations are needed then repository class definitions will simply be

empty as it will inherit logic from `GenericRepository` class. Imagine having dozens or hundreds of such classes where all basic CRUD operations are implemented with just class definition with no code inside.

```
public class ClassDescriptorRepository : GenericRepository<ClassDescriptor, int>,
IClassDescriptorRepository
    {

    }
```

Database connections and mapping configurations are implemented within the NHelper class (code snippet below). Methods of this class are called from the service layer to open new NHibernate Session and from `GenericRepository` to get current session.

```
public class NHelper
{
        private static readonly ISessionFactory SessionFactory;
        private static readonly FluentConfiguration cfg;

        static NHelper()
         {
             try
             {
                 //sets database configuration context to be for SQL Server 2005
                 var mssqlConf = MsSqlConfiguration.MsSql2005;
                 cfg = Fluently.Configure();

                 //gets connection string from web.config
                 mssqlConf.ConnectionString(c => c.FromAppSetting("PiPDbConnection_2"));

                //sets proxy factory assembly (needed for lazy loading of associated
                collections of objects)
                mssqlConf.ProxyFactoryFactory(("NHibernate.ByteCode.Castle.ProxyFactoryFacto
                ry, NHibernate.ByteCode.Castle"));

                 //set context for web applications (httpContext)
                 mssqlConf.CurrentSessionContext("web");

                 cfg.Database(mssqlConf);
                 var storeCfg = new StoreConfiguration();

                 //sets auto-mapping (Fluent NHibernate convention based mapping feature)
                 cfg.Mappings(m =>
m.AutoMappings.Add(AutoMap.AssemblyOf<ClassDescriptor>(storeCfg)
.Conventions.Add(FluentNHibernate.Conventions.Helpers.DefaultLazy.Never())
.UseOverridesFromAssemblyOf<ClassDescriptorRepository>));

//build NHibernate session factory
SessionFactory = cfg.BuildSessionFactory();

             }
             catch (Exception ex)
             {

             }
         }
```

```
        //opens new NHibernate session
        public static ISession OpenSession()
        {
            return SessionFactory.OpenSession();
        }

        //gets current NHibernate session
        public static ISession GetCurrentSession()
        {
            return SessionFactory.GetCurrentSession();
        }
    }
```

The session is Nhibernate's implementation of Unit of Work pattern. It is not within the scope of this document to explain this pattern in detail. Most useful NHibernates Session's functionalities are taking care of transactions and concurrency – developers do not need to take care of those difficult aspects of applications. This is another advantage of using advanced Object-Relational Mapping tool.

Mapping between the domain model business entities and database tables is done by Fluent NHibernate tool with its powerful feature of auto-mapping. Basically, it is possible to agree on and configure naming conventions for domain model classes and their property names and also database tables and their column names to be the same (or match configured pattern). If done properly, Fluent NHibernate will take care of object-relational mapping and SQL statements generation. This is an additional huge saving to development time and costs. Imagine you required a new business entity and a corresponding table in the database; you would just create new class in your domain model with appropriate properties, create database tables with appropriate columns, develop Repository for your new business entity (which will inherit from `GenericRepository` class) and that is all – basing CRUD operations for new business entity are already implemented for you. Obviously, there will be more work if there are bigger changes to your application, but this approach is still flexible enough to respond to change requests in a timely fashion.

There are other technical aspects of DAL related to Fluent Nhibernate ORM tool. However, this is not within the scope of this document to describe them. The purpose of this section was to show that using appropriate architectural and design patterns together with Object Relational Mapping tools may save a lot of development time and future maintenance costs, especially for large solutions with dozens or hundreds of business entities. Such architecture is very flexible and allows responding to unavoidable change requests in a timely fashion. A developer may focus on business logic to solve real-life business problems. With DAL implemented in the way described above, there is significantly less time needed for developers to develop and maintain DAL logic.

The disadvantage of this approach is that it requires initial significant investment in development time and developer skill sets to build a proper architecture of the application. But it pays back in the longer term – change requests may be delivered in a timely fashion as the architecture is flexible enough and less prone to regression (breaking some piece of code by doing some changes in other piece of code).

8.4 Calling DAL repositories from the service layer

The code snippet below shows and explains how to call DAL repositories from the service layer. It shows `SaveOrUpdateClassDescriptor(ClassDescriptor classDescriptor)` method which either adds new entity (and any other associated entities) into the repository (if entity does not exist there yet) or update existing entity with changes. `GetClassDescriptorById(int id)` methods returns

`ClassDescriptor` entity with given id. The code below is simplified – in the real-life application there would be probably much more business logic between the begin and commit of the transaction.

NHibernate takes care of cascade inserts, updates and deletes, too. That means that if there are associated entities within top level aggregate entity then calling method from this top level entity will automatically propagate changes for all associated entities appropriately.

```
SaveOrUpdateClassDescriptor(ClassDescriptor classDescriptor)
{
   //opens NHibernate session
   using (ISession session = NHelper.OpenSession())
   {
       int savedId = 0;
       //begins transaction
       using (ITransaction transaction = session.BeginTransaction())
       {
           try
           {
             //binds to current session context
             CurrentSessionContext.Bind(session);

            //gets instance of ClassDescriptor repository from Repository factory
             IClassDescriptorRepository             classDescriptorRepository             =
       RepositoryFactory.GetClassDescriptorDao();

            //Saves or Updates ClassDescriptor within repository
            classDescriptorRepository.MakePersistent(classDescriptor);

            //commits transaction
            transaction.Commit();

            savedId = classDescriptor.Id;
          }
          catch (Exception e)
          {
            //if error - rollback changes
            transaction.Rollback();
            throw;
          }

      }

      return savedId;
  }
}
```

```
public ClassDescriptor GetClassDescriptorById(int id)
{
  //creates new empty instance of ClassDescriptor
  var entity = new ClassDescriptor();

  //gets instance of ClassDescriptor repository from Repository factory
  IClassDescriptorRepository repository = RepositoryFactory.GetClassDescriptorDao();

  //opens NHibernate session
  using (ISession session = NHelper.OpenSession())
  {
     //begins transaction
     using (ITransaction transaction = session.BeginTransaction())
     {
```

```
        try
        {
           //binds to current session context
           CurrentSessionContext.Bind(session);

          //gets ClassDescriptor entity from repository
          entity = repository.GetById(id);

          //commits transaction
          transaction.Commit();
        }
        catch (Exception e)
        {
            transaction.Rollback();
        }
    }
  }
  return entity;
}
```

9. References

[1] "Principles In Patterns," 2012. [Online]. Available: http://www.principlesinpatterns.ac.uk/. [Accessed: 29-Feb-2012].

[2] D. Cullen, J. Everett, and C. Owen, "The curriculum design and approval process at the University of Strathclyde: baseline of process and curriculum design activities," 2009. [Online]. Available: http://www.principlesinpatterns.ac.uk/Portals/70/pip%20document%20library/ProjectReports/PiP%20Baseline%20October%202009.doc.

[3] PiP Project Team, "Principles in Patterns (PiP) Institutional Story," 2013. [Online]. Available: http://www.principlesinpatterns.ac.uk/Resources/Documentsanddissemination.aspx. [Accessed: 29-Apr-2013].

[4] J. Everett, G. Macgregor, and R. Mohamed, "An incremental approach to technology-supported curriculum design and approval," in *Procedings of the IADIS International Conference WWW/Internet 2012*, Madrid, 2012, pp. 453–457.

[5] Microsoft, "Microsoft SharePoint – collaboration software," 2013. [Online]. Available: http://office.microsoft.com/en-us/microsoft-sharepoint-collaboration-software-FX103479517.aspx. [Accessed: 24-Apr-2013].