# Olympus Training - PRISM

Cover **basic aspects** on the purpose of PRISM, as well as the main components and patterns involved (*Bootstrapper, Modules, Shell, Regions, DelegateCommand, and EventAggreggator*).

Discuss then **PRISM in Olympus**, adaptation to WinForms being of particular interest.

**What is PRISM?**

Composite Application -> decoupling, reusability, modularity

Patterns Toolbox

**PRISM Components**

|  |  |
| --- | --- |
| **Setup** | 1. Bootstrapper – application startup point  * Initialize Module Catalog + Initialize Modules * Configure Container – **DI** recap? (UnityContainer) * Create and initialize shell * Initialize regions |
| 1. Modules – functional split of the applications (though there are “technical” modules as well) -> UI, Services, Data Access  * Separate DLLs * Communicate strictly via interfaces * Rarely reference each other (exception: Infrastructure & utility modules)   Register (xml)  Discovered by bootstrapper  Initialized (Container) |
| **UI Constructions** | 1. Shell – “main window”  * Root of UI composition * Contains Regions – inject sub screens |
| 1. Regions – standard screen behavior for hosted (injected) screens  * Placeholders – standard & decoupled * Not necessarily just for inner composition (give example with Popup Region) * Behavior injection is also very important (example: turning all text red) |
| **Communication** | 1. DelegateCommand - .NET wrapper – Command Pattern  * Functional programming approach * Used in WPF – give example with buttons and CanExecute/Execute |
| 1. EventAggregator – Publish-Subscribe model  * Register & subscribe to strongly typed events * Can be conditional * Example where to use: view refresh! (Data) |

**PRISM in Olympus**

1. Program – setup Application (see also *SingleInstanceApplication*)

* Hooks on running application
* Initialize Bootstrapper
* Run
* Startup Next Instance
* Startup -> run boostrapper, check db version, login
* Running -> link shell
* Shutdown

(LocalizationService -> used to translate to Dutch)

1. SingleInstanceApplication – ensures only one Olympus Instance
2. Bootstrapper

* Module catalog (see App.Modules.Config)
* Configure container – register shell (notice how shell is set as Main Window for WinForms application) **[connection point WinForms – PRISM]**
* Init Regions

1. Modules – show some implementations to see initialization
2. Shell – base WinForm

* Start application and show Shell both running and in designer

1. Regions – see regions

* All views – user controls (UCs) for composition (UC can contain other UCs)
* WinForms – always requires a form to open user controls in (parent form)
* Start with DialogRegion and PopupRegion -> WindowViewHost important! **[connection point WinForms – PRISM]**
* ContainerRegion -> ViewContainer & ViewContainer with Commands **[connection point WinForms – PRISM]**

**Homework**

Study code and get comfortable with PRISM concepts applied in Olympus.

Take a tour around Olympus and familiarize with the basic screen types (from functional point of view).

# Olympus Training – Infrastructure

Cover basic components of Olympus Infrastructure, with specific focus on the Database common constructs, DAL and UI layer.

**Review basic concepts**

Make sure *PRISM, MVP (passive view vs. supervising presenter), Dependency Injection, Design Patterns (TM, Visitor) and Delegate Command* are all properly understood.

Start with a basic diagram of the system – split into 3 layers, communication with db.

**Database**

Tables - System Tables, regular tables, views

* **System tables** – mapped into enums (T4 templates) – reason and general usage?
* **Regular (entity) tables** – mapped into entities through ORM [dbo schema]
  + Mention the 7 standard columns and their purpose
  + Logical deletion
  + Primary key (never composite, auto-increment etc.) – infrastructure limitation and generalization
* **Views** [Olympus schema]
  + Functional purpose (mostly for projection on list screens)
  + Emphasize logical deletion yet again!

**DAL**

1. Model (Entities)

* Briefly present CommonEntityBase and show common aspects of all models (generalities)
* Entities vs. Presentation Models (what is a Presentation Model? Are entities PMs?)
* ModelName, ModelId etc.

1. Data Service – both non-generic and generic

* Generic role once again very important
* Give an example as how the data service can be used
* Create() method and its purpose

**MVP**

General model in Olympus – MVP with supervising presenter (bindings very important)

1. Presenter class – what’s its purpose?

* Enforces screen lifecycle – combination of Template Method and Visitor. Go through all points and explain them briefly (focus on OnAfterViewCreated, Attach/Detach Events, and Subscribe/Unsubscribe from EA for proper usage!!!)
* Has no model
* Have a short look at View class

1. Hierarchy of presenters & views – presenter List, Edit and Read Only (mention others as well)

* Registration of generic presenters (interesting aspect of UnityContainer resolving on generic types)
  + RegisterType(typeof(IClass<,>), typeof(Class<,>) – special construct -> .NET generics available at runtime
* When do we need a custom presenter? Shortly debate
* When do we need a custom view? (always)
  + Why? UI is not generated dynamically

**Controller & Commands**

What is the controller? – Command container (**not** MVC controller)

Why would we put commands in controller? Central point of access for **UI** interaction commands (opening standard screens), direct interaction with DAL for deletion

When do we need a custom controller? (Always)

* Why? Commands disabled by default

**Navigation Commands**

Specific commands – briefly mention database and show initialization in modules

**Homework**

1. How does F1 functionality work? Find in code.
2. How do lists become open? Click on navigation entry triggers it. How does this happen in code?
3. Consider an open list. What are the steps for opening an overview? (double click functionality)
4. What happens when we want to open an overview for the same entity twice? Why? Explain in code. Same question for edit screen.

# Olympus Training – Basic screen creation

Prepare new members for lab 1. This is more of a show-and-tell/tutorial presentation with focus on the practical aspects on developing basic screens in Olympus.

**Navigation** – cover both database (Navigation tables) and code (Refresh enum, model changes)

* This would be a good point to mention Functionality as well
* Matching between column and code are a bit tricky
  + ResourceKey – Message
  + Code – name of command

**Db Changes** – recap basic columns required

* Quick notes on view creation – always use designer, general format of view code
* Foreign keys very important for navigation
* Always use proper schema (dbo for tables, Olympus for views)

**LLBLGen Refresh** – cover all steps required for refreshing both for list and entities.

* Checkout code – show Olympus Best Practices entry
* Refresh database
* Revert engineer entity & list
* Generate code
* **Always** save LL project after generation complete successfully
* Add files in project & Source Control (Olympus Best Practices)

**Creating lists & read-only/edit screens**

Cover view creation

* Mention proper usages of Layout Control (includes grouping and tab functionality)
* Always use Ol controls (read only for overview, editable for edit)

Cover other elements required for labs

* Filters in row metadata (general ideal)
* Validation Repository for entities (general validation)

Some basic prefetch elements (to be later covered more thoroughly in LL KTS) – how to prefetch simple graph

# Olympus Training – LLBLGen

Describe more thorough aspects on how LLBLGen can be used. The presentation will cover API aspects of LL in tandem with the functionality provided by the infrastructure.

**Basic Data Service Operations**

* *GetById* -> select unique entity (+related graph if needed)
* *Save* -> Add new entity & Update existing entity (+related graph if recurrent)
* *Delete* -> logical delete!
* *GetDbCount* -> does not return select, but instead count only (more efficient)
* *GetEntityCollection* -> selects multiple entries (similar to GetById)

Fetch operations contain extra parameters (to be discussed):

* RelationBucket – GetById
* PrefetchPath – GetById
* RelationPredicateBucket – GetEntityCollection, GetDbCount

**Prefetch Path**

* ORM maps tables into graphs of objects based on FK definitions (1->n)
* When retrieving an entity from db – which related entities do we want?
  + Can’t have them all, especially in case there are cycles
* Prefetch Path – express related entities of the graph to be fetched, starting from the **root**
* Warning: each prefetch is a distinct SELECT (give example on a simple graph with both 1->n and n->1 relations)

**Examples**

|  |  |
| --- | --- |
| LL | Code |
| SELECT \* FROM Sync\_CourseTemplates  WHERE CourseTemplateID = 50 | courseTemplateDataService.GetById(50); |

IPrefetchPath2 prefetch = new PrefetchPath2(EntityType.CourseTemplateEntity);

prefetch.Add(prefetchElement, maxNrOfElements, predicate, relationCollection, ...)

* prefetchElement - static property on entity
* maxNrOfElements - 0 = infinity
* predicate ⬄ WHERE condition
* relationCollection ⬄ JOIN

**Predicates** – conditions for filtering, joining etc.

1. Simple predicates

* Equality, Inequality, >, <, >= etc.
* [Entity]Fields.Field == value (DBValue.Null example and explanation)

Note: For deletion check – always use CommonEntityBase.GetNotDeletedPredicate<T>();

* Custom predicates – see options from VS after writing new Predicate

1. Complex predicates – use predicate expressions

* AND and OR between conditions
* Expressions derive from predicates (classes)

**Relations** – setup joins in expressions for more complex filtering

* This enables adding fields to predicates that are on related entities specified in relations
* IRelationCollections – holds multiple relations
  + EntityRelation (pk, fk, type etc.)
  + .Add(relation, alias (optional), join hint (optional) – INNER, LEFT, RIGHT, CROSS)
  + Relation obtained from Entity.Relations property

**How are complete queries generated?** (start from an example)

* 1 prefetch node = 1 SELECT
* JOINS = Relations
* WHERE = Predicate
* TOP X = maxNrOfItems
* FROM = prefetch element

**RelationPredicateBucket** – on multiple select/count (GetEntityCollection, GetDbCount)

* Combines a *PredicateExpression* with *Relations*

**Another example to emphasize difference between prefetch and relation+predicate**

|  |  |
| --- | --- |
| **Get CT by Id -> (prefetch) PG (not deleted)** | **Get PGs -> with non-deleted CTs** |
| SELECT \* FROM Sync\_CourseTemplates  WHERE CourseTemplateID = 50 | SELECT \* FROM ProgramGroup pg  INNER JOIN Sync\_CourseTemplates ct ON pg.CourseTemplateID = ct.CourseTemplateID  WHERE ct.Deletion\_Date IS NULL |

**Complex prefetch with branching**

CT -> PG -> CI -> SA (1) -> Student -> Person -> Employee & Teacher

SA (2) -> SAI -> MI (with Event) -> Program (with Season) -> MT -> KA & EL

**Note**: For update & Add keep in mind that the entire graph will be persisted!

* Connections are automatically handled both ways (FK vs. PK)

**Aliases** – used for circular references in joins or prefetches

**How to run scripts in Olympus for Lab2?** Button or dump in OnAfterViewCreated()

# Olympus Training – Commands & Services

Cover various alternatives to implementing services in Olympus and their triggering from UI buttons.

**Recap ICommand vs. DelegateCommand**

* ICommand – pure OOP approach, write separate classes for each different functionality
  + Verbose but easier reusability in OOP contexts
  + Problem of type safety: object passed instead of generic parameter T (can be fixed)
* DelegateCommand – functional approach, wrapper over ICommand
  + Easy to create contextually based on delegates
  + Not reusable outside context

**Link commands to screens**

Programmatically added in *AddCommandsInToolbarCatalog* Template Method

* ToolbarCommandDescriptor – ICommand, name, caption, imageIndex (explain each)

Behind the scenes – Presenter

* Creates a command catalog
* Populated via AddCommandsInToolbarCatalog and AddAdditionalCommands
* RegisterCommandCatalogue in host (*ViewContainerWithCommands*, *Shell*, *WindowViewHost*)
  + Concretely materialized as a BarButtonItem

Resulting model is simple:

Controller Presenter Infrastructure

**Problems with the model**

* Very fat controller – too much code there. Should handle UI flows, not business!
* No automatic error handling.
* No extensibility.
* Very poor composition of commands (no interaction mechanics).
* Cannot be unit tested.

**Service<T>** - building block for business services

Combines the advantages of ICommand’s reusability and context isolation with DelegateCommand’s type safety.

* Interface: Execute, CanExecute, ServiceBehaviours (discuss ExecutionResult and Behaviours)
* Extra protected hooks: CheckPreconditions, CheckPreconditionsOnExecuting, InnerExecute
* Discuss difference between the two “CheckPreconditions” methods

Typical service

* CheckPreconditions – immediate CanExecute (light conditions) – activate/deactivate button
* CheckPreconditionsOnExecuting - failure On Execution (button always enabled)
* InnerExecute – business logic goes here!

Generic parameter T usually wrapped around ServiceArgs

* ItemChangedEvent functionality – better notification mechanics instead of manual RaiseCanExecute()

**Rules when developing services**

* Never fetch data or save data from service (composition problem)
  + Can expose GetPrefetch() method to hint on possible prefetch
  + Behaviours can perform save or refresh before execution
* Never access UI (can use service input to interact with UI layer)
* Authorization is possible – use behavior

**Problem**: Services are not commands. So how can we link them in the schema above?

**Solution**: Introduce an adapter (command adapter)

**ArgumentServiceCommand<T>**

* Pass service via construction
* Argument changeable – triggers CanExecute
* Error handling can be managed depending on service result via **behaviors**

**ArgumentServiceSaveCommand<T>**

* Saves Model on successful execution
* Raises Model Saved event (refresh)
* Performs compensation on behavior (in case of behavior crash)

**Final Picture**