Kistl Guide

The Hitchhiker's guide to the Kistl galaxy

Contents

Chapter 1

Introduction

Kistl is a programming framework to provide the complete process from defining data structures, designing data access and transfer objects, designing servers and GUIs and the necessary parts to make everything work together.

Chapter 2

Programming

This chapter describes the various ways and pieces the Kistl system is programmed and customized.

2.1 Objects

2.1.1 ObjectClass

2.1.2 Relation

A *Relation* defines the relationship between two Objects. Every Object can have zero or more *Relations*.

An example of a *Relation* is the relation between *Project* and *Tasks*. One *Project* can have zero or more *Tasks*. One *Tasks* must have a *Project*.

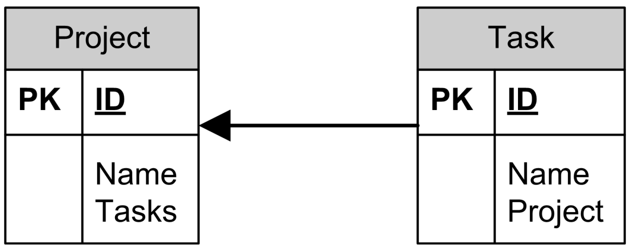

Figure 2.1: Example for a *Relation*

Modeling a relation

A Relation can be defined by creating an object of type *Relation* and two *RelationEnd* objects. This can be done by

- creating an Relation Object.
- invoking the *Create Relation* method on an *ObjectClass* instance.

RelationEnd objects will be created automatically.

Relations are edited in the *Relation Editor*. The *Relation Editor* is a custom *FullObjectView* created by us.

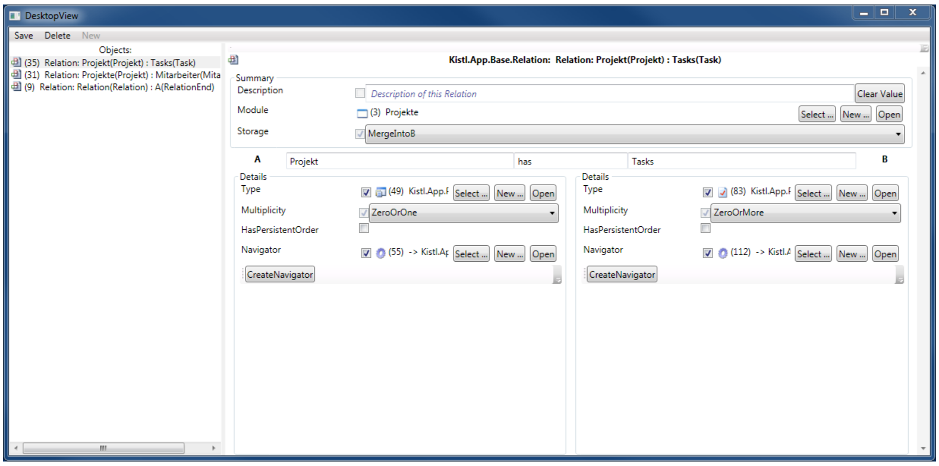

Figure 2.2: Example for editing a *Relation*

Attributes of a relation

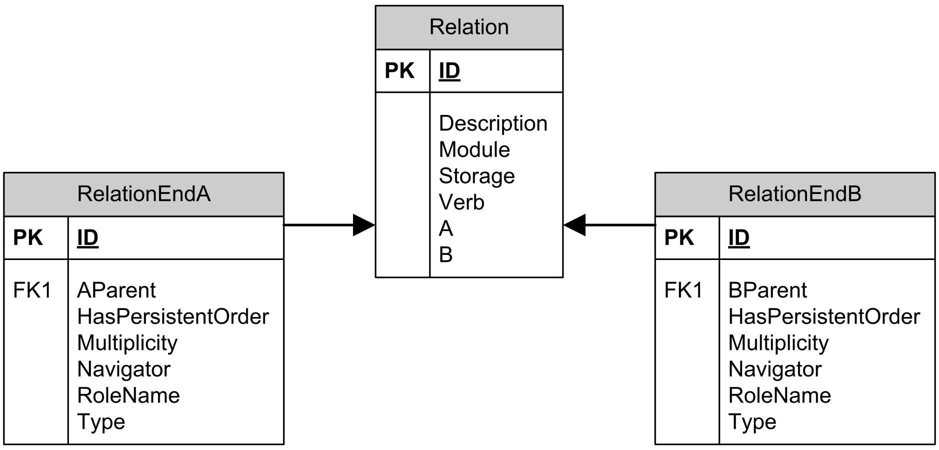

Figure 2.3: Attributes of a relation

A *Relation* has these attributes:

Description A text property used to describe the current relation

Module The *Module* which is introducing the current relation

Storage The *StorageType* of the current relation

Verb A verb used to name the current relation. The verb is used in conjunction with the role names of the *RelationEnd* objects to model a unique relation name. This relation name will be used e.g. for the database FK Contraint name.

A The *RelationEnd* A of the current relation

B The *RelationEnd* B of the current relation

A *RelationEnd* has these attributes:

AParent *Relation* object if this *RelationEnd* is the A-Side of the current relation. Otherwise *NULL*

BParent *Relation* object if this *RelationEnd* is the B-Side of the current relation. Otherwise *NULL*

HasPersistentOrder Specifies that the list is ordered. Applies only to lists

Multiplicity The *Multiplicity* of the current *RelationEnd*

Navigator An optional *Navigator*

RoleName Name of the role of the current *RelationEnd*

Type *ObjectClass* to which the current *RelationEnd* points

There are four *StorageTypes* defined:

MergeIntoA The relation information is stored with the A-side database table

MergeIntoB The relation information is stored with the B-side database table

Replicate The relation information is stored on both sides of the relations database tables

Separate The relation information is stored in a separate database table

There are three *Multiplicities* defined:

ZeroOrOne Optional Element (zero or one)

One Required Element (exactly one)

ZeroOrMore Optional List Element (zero or more)

1:n Relation

A *Project* can have zero or more *Tasks*. A *Task* may have one *Project*.

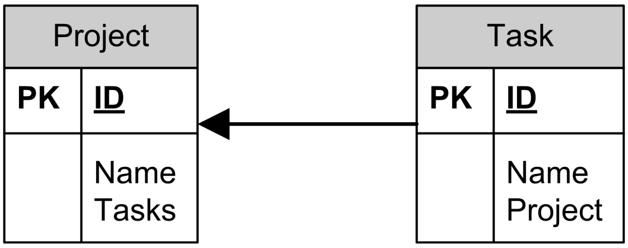

Figure 2.4: Project/Tasks relation

The *Relation* object would be:

Storage = MergeIntoB

Verb = has

The *RelationEnd* A object would be:

AParent = *Relation*

BParent = *NULL*

HasPersistentOrder = false

Multiplicity = ZeroOrOne. If a *Task* must have a *Project* then One.

Navigator = *Navigator* to Tasks. The result would be a collection of *Tasks* (ICollection<Task>)

RoleName = Project

Type = *Task* instance of type *ObjectClass*

The *RelationEnd* B object would be:

AParent = *NULL*

BParent = *Relation*

HasPersistentOrder = false

Multiplicity = ZeroOrMore

Navigator = *Navigator* to the parent *Project*. The result would be a reference to a *Project*

RoleName = Tasks

Type = *Project* instance of type *ObjectClass*

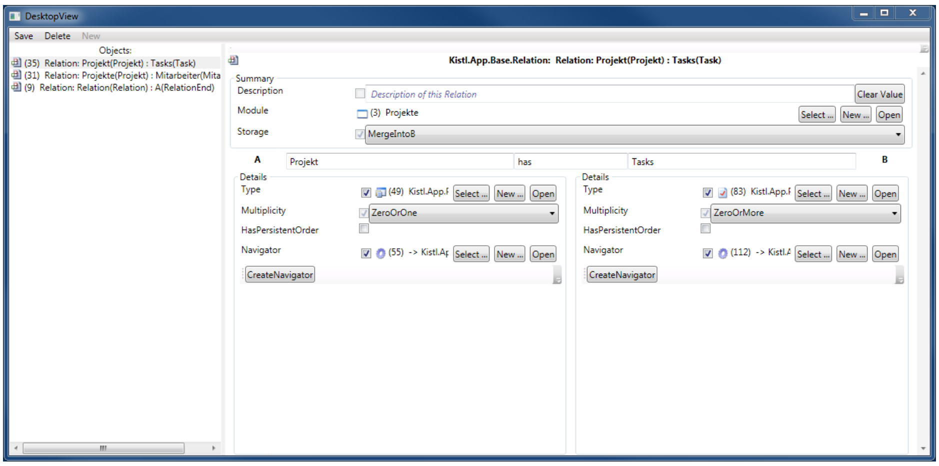

Figure 2.5: Editing the *Project/Tasks* relation

n:m Relation

A *Project* can have zero or more *ProjectMembers*. A *ProjectMember* can be assinged to zero or more *Projects*.

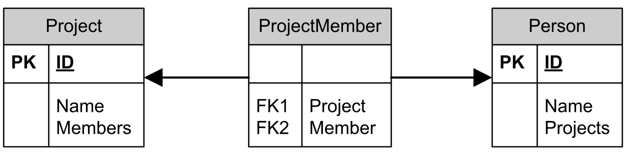

Figure 2.6: Project/Member relation

The *Relation* object would be:

Storage = Seperate

Verb = has

The *RelationEnd* A object would be:

AParent = *Relation*

BParent = *NULL*

HasPersistentOrder = true

Multiplicity = ZeroOrMore.

Navigator = *Navigator* to Persons. The result would be a list of *Persons* (IList<Person>)

RoleName = Projects

Type = *Person* instance of type *ObjectClass*

The *RelationEnd* B object would be:

AParent = *NULL*

BParent = *Relation*

HasPersistentOrder = true

Multiplicity = ZeroOrMore

Navigator = *Navigator* to the assigned *Projects*. The result would be a list of *Projects* (IList<Project>)

RoleName = Member

Type = *Project* instance of type *ObjectClass*

1:1 Relation

A *Relation* must have a *RelationEnd* A. A *RelationEnd* may have a AParent *Relation* if it's a A *ReleationEnd*.

The *Relation* object would be:

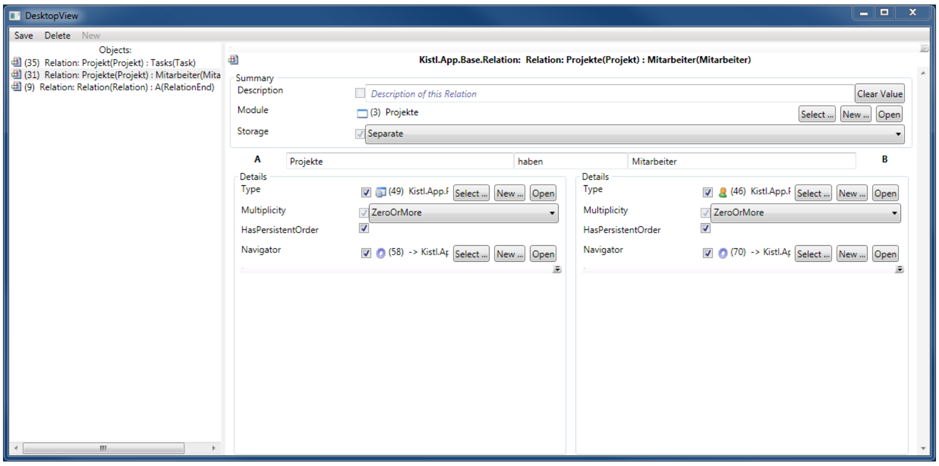

Figure 2.7: Editing the *Project/Member* relation

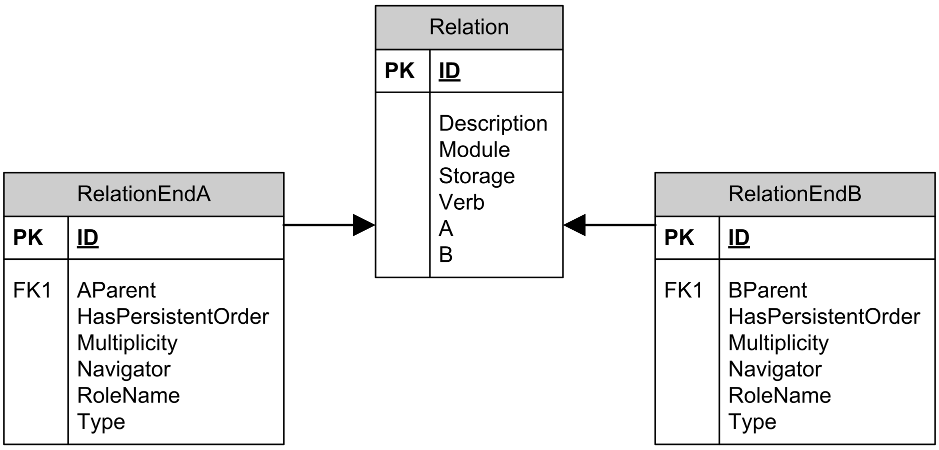

Figure 2.8: Relation/RelationEnd relation

$$\textbf{Storage} = \text{MergeIntoA}$$

$$\textbf{Verb} = \text{hasA}$$

The *RelationEnd* A object would be:

$$\textbf{AParent} = \textit{Relation}$$

$$\textbf{BParent} = \textit{NULL}$$

$$\textbf{HasPersistentOrder} = \text{false}$$

Multiplicity = ZeroOrOne

Navigator = *Navigator* to *RelationEnd*. The result would be a reference to a *RelationEnd*

RoleName = Relation

Type = *RelationEnd* instance of type *ObjectClass*

The *RelationEnd* B object would be:

AParent = *NULL*

BParent = *Relation*

HasPersistentOrder = true

Multiplicity = One

Navigator = *Navigator* to the assigned *Relation*. The result would be a reference to a *Relation*

RoleName = A

Type = *Relation* instance of type *ObjectClass*

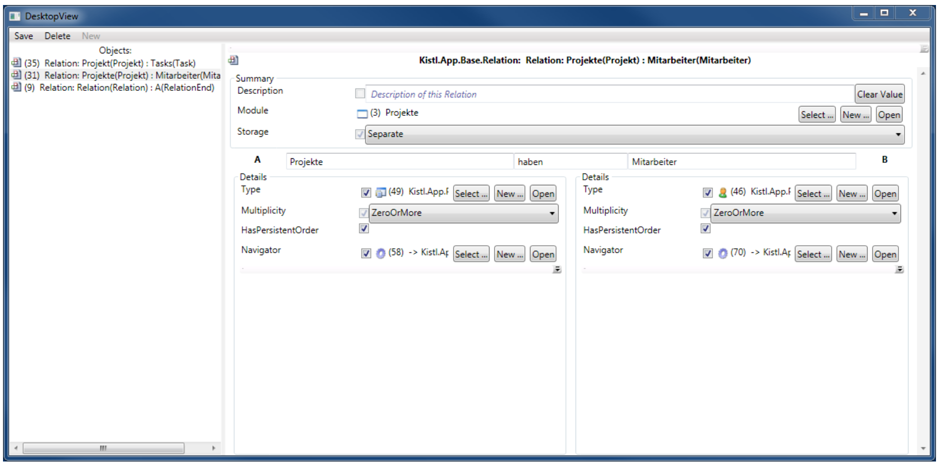

Figure 2.9: Editing the *Relation*/*RelationEnd* relation

Multiplicity and StorageType summary

1:n Storage = MergeIntoB	
A	**B**
A.Nav is a collection Multiplicity = ZeroOrOne	B.Nav is nullable Multiplicity = ZeroOrMore
A.Nav is a collection Multiplicity = One	B.Nav is not nullable Multiplicity = ZeroOrMore

n:1 Storage = MergeIntoA	
A	**B**
A.Nav is nullable Multiplicity = ZeroOrMore	B.Nav is a collection Multiplicity = ZeroOrOne
A.Nav is not nullable Multiplicity = ZeroOrMore	B.Nav is a collection Multiplicity = One

n:m Storage = Seperate	
A	**B**
A.Nav is a collection Multiplicity = ZeroOrMore	B.Nav is a collection Multiplicity = ZeroOrMore

1:1 Storage = MergeIntoA Storage = MergeIntoB Storage = Replicate (Not supported yet)	
A	**B**
A.Nav is nullable Multiplicity = ZeroOrOne	B.Nav is nullable Multiplicity = ZeroOrOne
A.Nav is nullable Multiplicity = One	B.Nav is not nullable Multiplicity = ZeroOrOne
A.Nav is not nullable Multiplicity = ZeroOrOne	B.Nav is nullable Multiplicity = One
A.Nav is not nullable Multiplicity = One	B.Nav is not nullable Multiplicity = One

2.1.3 Additional Metadata

The object model is intended to be very rich and provide the various subsystems with meta data directly from the *ObjectClass*.

This section describes the various pieces of this meta data.

New related objects

A *CreateRelatedUseCase* describes the use case of creating a new object related to the "current" instance. One such use case would be e. g. "create a new *Relation* from the current ObjectClass."

Such use cases are described with *CreateRelatedUseCase* objects:

```
interface CreateRelatedUseCase
{
        string Label;
        Method Action;
        Relation AffectedRelation; // optional
}
```

The *Action* will be called when the user requests an execution of this use case. This method doesn't take any parameters and returns the newly created object. The infrastructure on the client will cause the returned object to be displayed to the user. The business logic should already have filled out the property values according to the use case. The name of the method should start with "Create".

If the optional *AffectedRelation* is specified, one of its ends it must match the *ObjectClass* of the *Method*. This relation can then be used to identify controls in the UI where the action can be placed.

2.2 Modules

2.3 Enhancing Kistl's inner workings

2.3.1 Database Providers

2.4 Graphical User Interface

Like other subsystems, the GUI core is designed to be platform independent. Therefore only the "outermost" shell contains toolkit specific code.

2.4.1 Architecture

The GUI is modeled after the Model-View-ViewModel architecture. The *Model* represents the underlying data structures and business logic. It is provided by the generated classes from the actual datamodel. *View Models* or *presentable* models provide display specific functionality like formatting, transient state holding and implementing the user's possible actions. They always inherit from *Kistl.Client.Presentables.PresentableModel*. Common implementations reside in the *Kistl.Client.Presentables* namespace. Finally, *Views* (editors and displays) are the actual components taking care of showing content to the user and converting the users keypresses and clicks into calls on the view models interface. Views are toolkit[1] specific and reside in the toolkit's respective assembly.

This architecture decouples the actual functionality of the Model and the View Model completly from the inner workings of a toolkit and thereby maximise the reuse of code between different clients.

2.4.2 Plumbing

The three layers are connected through two sets of descriptors. The *PresentableModelDescriptor*s contain information about the available View Models and their preferred way of being displayed. The *ViewDescriptor*s link *PresentableModelDescriptor*s with the controls capable of displaying them.

Presentable Model Descriptors

Currently there exist three major types of View Models.

***DataObjectModel*s** represent a complete data object; provide standardised access to properties; provide non-standard Views with additional functionality; selected via *ObjectClass.DefaultPresentableModelDescriptor*

***ValueModel*s** represent a specific piece of data; representations of Properties are selected via *Property.ValueModelDescriptor*; method results are currently created directly via *Factory.CreateSpecificModel*

other Presentables represent objects in the View which do not have persistent representations, like dialogs or wizards; always created by calling *Factory.CreateSpecificModel*

View Descriptors

These descriptors list the available Views by Toolkit and which subset of Presentables they are able to work with.

Control Kind

The *ControlKind* specifies the toolkit-independent kind or type of control that should display a given Presentable. While the View specifies the Control Kind it implements the Presentable requests a specific Kind to be displayed via the *PresentableModelDescriptor.DefaultControlKind* value.

In special situations this default value can be overridden. For example, the metadata of a property contains a *RequestedControlKind* which is used instead of the *DefaultControlKind* when present. If there is no View matching

[1]Toolkits are GUI libraries like WPF, GTK# or Windows Forms but can also be implemented by more complex providers such as ASP.NET.

the requested Kind, the infrastructure may either fall back to the default control kind, or use a similar control kind from higher up in the hierarchy.

Typical kinds of controls:

WorkspaceWindow the top-level control within which all user interaction happens

SelectionTaskDialog a dialog letting the user select something from a longer list of items

ObjectView display the modeled object in full

ObjectListEntry display the modeled object as item in a list

TextEntry lets the user edit a property as text

IntegerSlider lets the user edit a number with a slider

YesNoCheckbox a simple yes/no checkbox

YesNoOtherText radio buttons allowing one to select either "yes", "no" or a TextEntry field

ExtendedYesNoCheckbox a checkbox with additional text as label

The kind of a control is identified by the *ControlKind*'s class. The hierarchy between different kinds of controls is modeled with inheritance.

Control Kinds can also be used to configure the actual control. This possibility should be used sparingly as a control should instead seek to infer its configuration from the underlying Presentable. For example, an integer slider control should lookup the minimal and maximal allowed values in the underlying *IntegerRangeConstraint* while an ExtendedYesNoCheckbox has no other place to retrieve the new label.

2.5 Core Kistl Development Environment

2.5.1 Preparing a clean local build

First, it is necessary to have a clean build environment. Use *subst* to create a drive *P:* where your checkout resides in a directory called *Kistl*.

The *!FullReset.cmd* will bring the database and the bootstrapping code up to the current *Database.xml*'s content.

Now the environment is ready for programming.

2.5.2 Merging local and remote changes

When the subversion repository has changed the *Database.xml* while local changes were made to the schema, it is necessary to merge them before comitting.

After fetching and merging the update from the subversion repository, the local *Database.xml* has changes which are not yet in the database. Running *!DeployAll.cmd* updadates the SQL-schema and produces a new set of generated assemblies in the *CodeGenPath*. After testing that the merge was successful, use *GetCodeGen.cmd* to update the working directory with the newly generated bootstrapping code.

Now the working directory is ready for check in.

Chapter 3

Packaging

This chapter describes the way how modules (packages) are transfered between Kistl installations.

3.1 Content of a package

A package contains schema information, meta data, object instances and code. See 3.1 for details. A package can contain one or more modules. A package is a zip file that contains one or more XML files containing meta information and/or data and binary files like icons or assemblies.

3.1.1 Schema

Schema contains all information needed by the SchemaManager **??** to create or update the database. Schema objects are:

DataType All object classes and structs except enumerations and interfaces. *To be implemented!*

Property All properties

Relation All relations

RelationEnd All relation ends

DefaultPropertyValue All known default property values (IntDefaultValue, ...)

Constraint All known constraints (NotNullable, ...)

Schema is a subset of meta data. See 3.1.2. Currently there is no way to extract schema information without meta information.

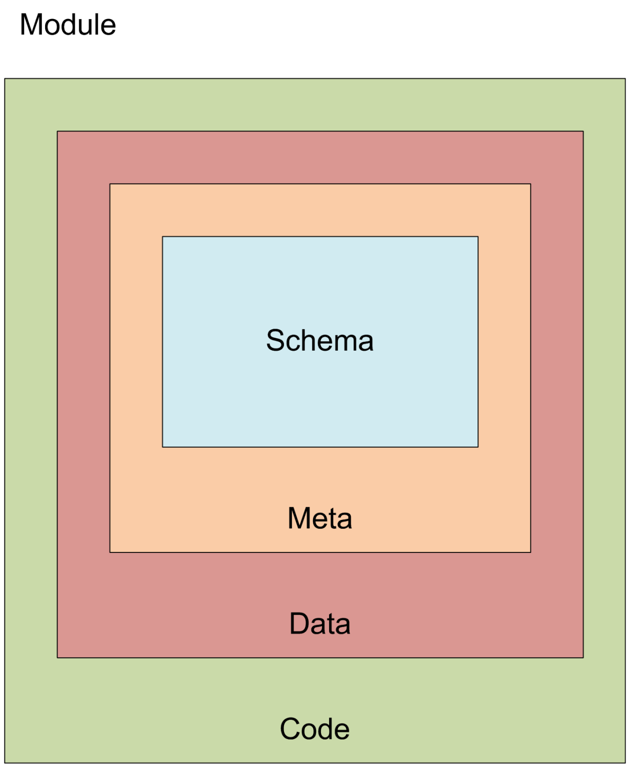

Figure 3.1: Content of a package

3.1.2 Meta

Meta data contains all information needed to describe a Module. That are *ObjectClasses*, *TypeRefs*, *Module* informations, *icons*, *ViewDescriptors* and so on. Meta data is a superset of schema data 3.1.1. These objects are:

Module The module object

ObjectClass_implements_Interface_RelationEntry All object class interface relations

Method All Methods

BaseParameter All parameter of a method

MethodInvocation All method invocations

PropertyInvocation All property invocations (getter/setter)

Assembly All Assemblies (meta information only, not code)

TypeRef All type references

TypeRef_hasGenericArguments_TypeRef_RelationEntry All generic arguments of type references.

Icon All Icons (descriptor only, no binary data)

PresentableModelDescriptor All presentable model descriptors

ViewDescriptor All view descriptors

DAVID: please add more GUI objects here. And please do it also in *PackagingHelper.GetMetaObjects(...)*

Additionally all unknown DefaultPropertyValues and Constraints belongs to meta data. Meta data can be extracted with the publish command 3.2.3.

3.1.3 Data

A package can also contain additional data. Only object classes that implements *IExportable* will be exported. N:M relations between object classes that implements *IExportable* are also exported. Currently all objects of a specific module are exported. Data can be extracted with the export command 3.2.1.

3.1.4 Code

Finally a package consists of code in form of assemblies. These assemblies are referenced by *Assembly* objects.

3.2 Processes

3.2.1 Export

Exporting is the process of saving objects in XML files. Only object classes that implements *IExportable* will be exported. N:M relations between object classes that implements *IExportable* are also exported. Currently all objects of a specific module are exported.

Command line for exporting objects:

```
Kistl.Server <configfile.xml> -export <destfile.xml> ...
    <namespace> [<namespace> ...]
```

The namespace is used to identify a module. *TODO: Work in progress. This is not the best solution!*

This example will export all objects of the project management module:

```
Kistl.Server -export Export.xml Kistl.App.Projekte
```

This example will export *all* meta data of *all* modules:

```
Kistl.Server -export Export.xml Kistl.App.Base Kistl....
    App.GUI
```

This example will export the whole database:

```
Kistl.Server -export Export.xml *
```

3.2.2 Import

Importing is the inverse process of exporting 3.2.1. Objects are imported by the following rules:

1. If an imported object already exists in the target database then the object will be overridden
2. New objects are added
3. No object is deleted if it's not contained in the package

Command line for importing objects:

```
Kistl.Server <configfile.xml> -import <sourcefile.xml...
    >
```

3.2.3 Publish

Publishing is a special case of exporting 3.2.1. Only meta data 3.1.2 of a given module will be exported. Additionally only properties of the Kistl.App.Base and Kistl.App.GUI module are published.

Command line for publishing modules:

```
Kistl.Server <configfile.xml> -publish <destfile.xml>...
     <namespace> [<namespace> ...]
```

The namespace is used to identify a module. *TODO: Work in progress. This is not the best solution!*

This example will publish the project management module:

```
Kistl.Server -publish Meta.xml Kistl.App.Projekte
```

This example will publish all modules:

```
Kistl.Server -publish Meta.xml *
```

3.2.4 Deploy

Deployment is the inverse process of publishing 3.2.3. It also has different rules (see importing 3.2.2). These Rules are:

1. If an imported object already exists in the target database then the object will be overridden
2. Only properties of the the Kistl.App.Base and Kistl.App.GUI module are overriden.
3. New objects are added
4. Any object that is not contained in the packed will be deleted

Command line for importing objects:

```
Kistl.Server <configfile.xml> -deploy <sourcefile.xml...
  >
```

The database schema is not updated. Also no code is generated. This has to be done in an extra step.

List of Figures