rubicon informationstechnologie gmbh

The re-motion PhoneBook tutorial

An illustrated primer

Reinhard.Gantar
02.06.2009

Inhalt

What is re-motion?

What is re-motion?

re-motion is a framework for building a special type of web application: applications that revolve around the management of forms and "domain objects" like invoice, customer, receipt, budget, car-pool and the like. re-motion derives forms and how they relate to each other from the descriptions of such domain objects. If you plan to build an enterprise web application for "managing stuff", re-motion might be for you. re-motion has the following characteristics:

- re-motion is based on .NET 3.5 (C#), ASP.NET 2.0 and MS SQL Server 2005
- re-motion is Open Source Software (mainly LGPL 3.0)
- re-motion hides most of the finer points of database programming and web programming from the programmer
- re-motion is focused on applications serving uniform requirements for consistent user interfaces, logging mechanisms or security features ("all reference properties must support X"; "all modifications to confidential domain objects must be logged in Y", etc.)

re-motion is being developed by *rubicon informationstechnologie gmbh*, a Viennese (Austrian) software-company. rubicon uses re-motion for virtually all its projects.

rubicon has a focus on public sector software, but re-motion has not

rubicon's product *Acta Nova* is a workflow and document management system for the public sector. The Acta Nova framework is a very big re-motion application.

Acta Nova, or re-motion, essentially works like many other workflow- or document management applications:

- electronic representations of "domain objects" (people, files, documents, orders, payments, etc.) are stored in a database
- users with certain privileges can create, view and edit domain objects
- the system can not only manage domain objects, but also the collaboration of people who work with them

The only difference between conventional workflow systems and rubicon's product is that the latter has a tradition of deployments in government institutions like ministries, governing authorities and municipal utilities. Traditionally, re-motion has been used as a library for building CRM systems, with the "C" meaning more "citizen" than "customer". However, **re-motion is not limited to the public sector** by principle. The reason that it has been used for e-government applications is that rubicon has virtually all its customers on this market and knows this market very well. Anyone willing to create his or her own specialized domain objects for other domains should be very well able to do so.

re-motion is open source software

re-motion is open source software. Most of it is covered by the Lesser General Public License 3.0 and related licenses. A detailed breakdown on how each component is licensed can be seen in the section **Fehler! Verweisquelle konnte nicht gefunden werden.** at the end of this chapter. Important disclaimers:

- the author is not a lawyer, so all license information in this document is for entertainment purposes only
- rubicon, re-motion's vendor, claims the right to license its products in a proprietary fashion to parties who don't want to share their code, as the LGPL (AGPL) demands. This scheme is called "dual licensing"
- except for people who plan to opt for dual licensing, no other strings are attached than those required by the LGPL (or AGPL, in some cases)

re-motion licenses in detail

Most of re-motion is covered by the LGPL, the GNU "Lesser General Public License", v 3.0. This includes

- re-store
- re-bind
- re-call
- re-strict (API-version)

re-strict, however, is covered by the AGPL, the GNU "Affero General Public License", v 3.0. -- IF RUN AS A WEB-SERVICE. re-strict is covered by the LGPL if (dynamically) linked to a re-motion application.

re-flow and re-form are covered by the AGPL, the GNU "Affero Public License".

Reference:

Dual licensing: http://www.oss-watch.ac.uk/resources/duallicence2.xml

LGPL: http://www.gnu.org/copyleft/lesser.html

GPL 3.0: http://www.gnu.org/licenses/gpl-3.0.txt

AGPL 3.0: http://www.gnu.org/licenses/agpl-3.0.html

re-motion's system requirements

re-motion builds on .NET 3.5, C# 3.0 and ASP.NET 2.0. Currently only Microsoft SQL Server 2005 is supported as database backend. For development it is assumed that you have Microsoft Visual Studio 2008 and an instance of SQL Server 2005 running locally on your machine. re-motion is tried and trusted on both 32-bit- and 64-bit systems. Clearly you will need a largish computer for hosting the development environment. 4 gigabyte RAM work well for 32-bit computers; 8 gigabyte work well for 64-bit computers. On security: re-motion requires *full trust.* This is in part owed to the fact that re-motion depends on liberal reflection permissions.

About this document, programmer requirements

This document is a primer on re-motion. (Acta Nova is not discussed here.) This primer illustrates the most important concepts like domain objects, transactions, data binding, object/relational-mapping, controls and re-motion's "web execution engine". Important parts of re-motion are not covered (workflow) or only briefly discussed (security).

You don't need to be a crack programmer for getting useful results quickly from re-motion. Even if you have only beginner skills in databases, ASP.NET or C# you should be able to get simple applications going with very little effort. However, this primer is not yet equally comprehensive on all aspects of re-motion programming. Virtually no other documentation exists at this time (fall 2008; we're working hard for more every day). The more you know, the quicker and more productive you'll be when you venture into the netherworld of re-motion's (as yet) undocumented depths.

The good news is that re-motion's authors were quite pedantic about sticking to modern established practices and principles of software-engineering (patterns, unit-tests, comments for classes, methods and properties). This tutorial has been written in the hope that it will be useful to programmers of all skill-levels, from neophyte to leet h4x0r. This primer is intended to help with the most challenging task for every programmer: seeing the forest for the trees. As for the reader's expected level: if you have mastered C#, relational databases and ASP.NET to the "... for dummies"-level, you will probably do fine.

Structure of this document

This primer is based on rubicon's introductory training classes on re-motion and roughly follows its pace and structure. In the course of demonstrating how to build a simple phone-book web application, all basic re-motion concepts are introduced and discussed. These practical examples are preceded by

- "backgrounders", introducing common industry concepts (like O/R-mapping) to beginners
- "re-motion theory", discussing aspects of re-motion's design (often trying to motivate certain design desicions)

In addition to the practical phone-book lessons, tips for trouble-shooting and best practices are presented afterwards.

Credits

Many people have contributed to this document. Most of its material came from interviewing re-motion's developers, who were exhibiting a lot of knowledge and patience, namely Michael Ketting, Fabian Schmied and Stefan Wenig. The fascicle **Fehler! Verweisquelle konnte nicht gefunden werden.** is based on a paper by Stefan Wenig, who is also the inventor of the phone-book sample and original lecturer of the long-running re-motion class.

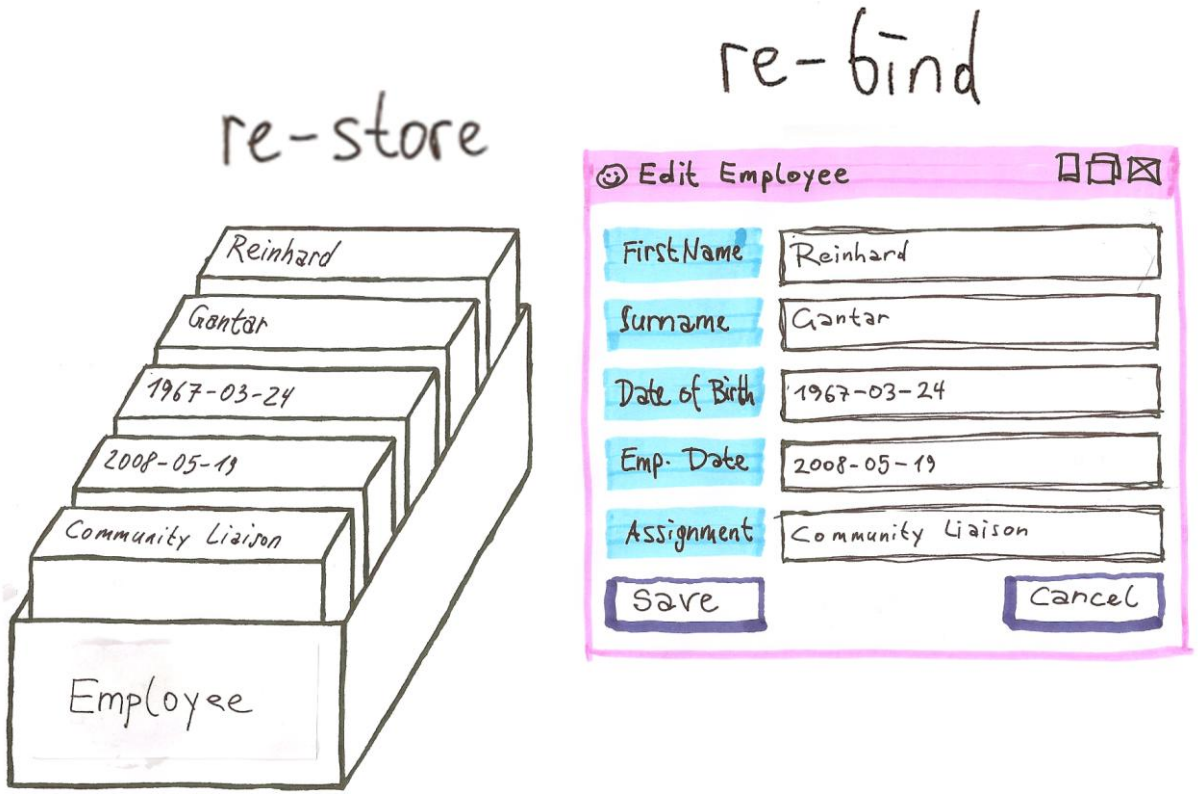
Important re-motion concepts and components -- re-motion for programmers

For a quick and shallow introduction, check out the PhoneBook-Application running on the intarwebs, at http://re-motion.org/PhoneBook .

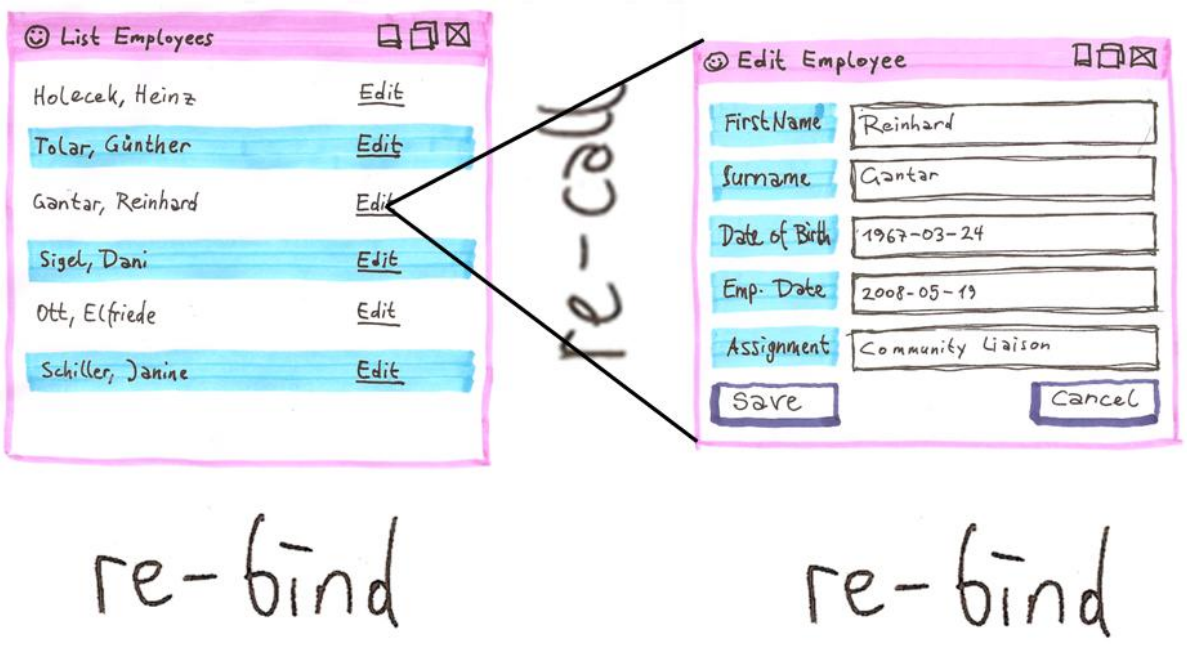
re-motion provides the basic infrastructure and building blocks for constructing CRUD- and workflow applications, and the most important building blocks are:

- re-store -- the object persistence and transaction framework layer (persisting domain objects)
- re-bind -- the Business Object Control layer (binding between business/domain objects and their representation in the browser)
- re-call -- the Web Execution Engine (for building collections of pages the users can navigate)
- re-strict -- the Security Manager (handling user roles and privileges for domain objects and their properties. At this time, re-strict is not covered by this primer)

Here are two illustration how re-store, re-bind and re-call interact and depend on each other. The box on the left shows a "domain object", i.e. a set of properties for an instance of type "Employee": first name, surname, date of birth, assignment. The left window is a browser impression and shows a form for displaying and editing that domain object.



Managing domain objects in memory and in the database is the job of re-store. Mapping from domain object data to controls in the browser is done by re-bind. re-call is concerned whenever the user clicks on something that causes the application to move to a new web-page. In the following illustration, the user clicks on an instance (a "record", if you will) in a list, and re-call moves her to the corresponding form for editing that object. In the drawing, both pages are composed of re-bind controls. Moving from one page to the other is re-call's work:



re-call's work looks simple, but re-call can do a lot for you. There's more to it than meets the eye.

The next few sections give an introduction to and an overview of the most important aspects of re-motion from a technical perspective.

Domain objects vs. business objects

Most ideas and programming in re-motion hinge on one central concept: the "domain object". The first section gave a few examples for domain objects, here are a few more:

- person
- address
- organization
- signer

Whatever the users of a particular system need in their domain of expertise can become a domain object in the system.

In literature and colloquial use, the terms "domain object" and "business object" are fairly interchangeable. What is called a "domain object" in re-motion is often called a "business object" by other authors. Both terms mean software objects representing domain phenomena. However, in re-motion the two terms are related, but each has its own special meaning.

- A *domain object* is persisted in the database
- A *business object* exposes an interface for binding to (ASP.NET) controls

An object that can do both - binding AND persistence - is called a *bindable domain object.* For convenience, we will use "domain object" as the generic term. We will later make extensive use of domain objects and bindable domain objects.

In common practice, "domain object" is not so much a technical term, but also an abstract concept, with each type (class) of domain object having a particular form and face in each layer of the software. In the database, an instance of a domain object is distributed across tables, depending on its type. When transported over a network, the same domain object is represented as XML, in compliance with a type-specific structure. As soon as a domain object is displayed or supposed to be modified by a user, an appropriate control is generated for rendering in a browser. In a word, each "layer" deals with a certain aspect of a given domain object, or translates between aspects. An example for this is the object persistence framework (re-store), which translates domain objects in database tables to .NET objects in a running program, and vice versa.

Domain objects are composed of properties, which in turn can be domain objects themselves, or better: references to existing domain objects. "Address", for example, is a typical domain object. A domain object of type "person" often requires the storage of the person's home address. In this example, an instance of a "person" domain object contains a reference to another domain object instance of type "address". What's more, every property in a domain object has "metadata" tucked to it, for example:

- the given property can be modified or not
- the given property is nullable (optional) or not

Other metadata items depend on the type of the property. A string property like a first name, for example, can have a certain maximum length.

Again, most workflow- and document management systems involve domain objects as understood here, but they are often called "business objects". This term, too, is used for re-motion, but the following criteria restrict what qualifies as a domain object in re-motion:

- domain objects are ultimately rooted in "real" objects from the world of the users and their work (parking ticket, donator, fare dodger, tax report, service of a writ, etc.)

and

- domain objects are persisted in re-motion's database

"Business objects" also exist in re-motion, but their central feature is that they can bind to *business object controls*. This brings us to the next topic.

Business object controls (BOCs, "re-bind layer")

"Business object controls" give (bindable) domain objects faces and handles in a browser, but their use is not limited to domain objects. Business object controls can give faces and handles to anything that supports the interface `IBusinessObject`, and this includes domain objects (which, by defintion, get promoted to *bindable* domain objects if they have this virtue). To illustrate the concept, let's focus on a domain object in particular and a simple example.

For a domain object of type "person", the browser displays a form with text entry fields or other controls for each property of the "person" instance. "First name" and "sure name" are obviously text entry fields. Other possible obvious relationships between properties and controls:

- property is a date -> date picker

- property is a number -> text entry field with a check if the entered string can be parsed as a number
- property is a boolean -> check box
- property is an enumeration -> drop box

Things are not that easy, however, because re-motion's re-bind (binding) layer is more flexible than this. Depending on metadata and configuration, the re-bind layer can display a Boolean property as radio buttons, drop box or checkbox, for example.

So the BOC layer's job is

- to map from domain objects and their properties to user controls for display
- to map from the controls' user input to the corresponding updates of properties

How does the BOC layer know how to display domain objects and their properties? In order to do that, the BOC layer queries the domain object's `IBusinessObject` interface, which in turn looks up important information like size constraints and nullability in the properties' metadata.

As explained earlier, the BOC layer can not only display domain objects. Whatever supports `IBusinessObject` is eligible for rendering in a browser by the BOC layer. Good examples for this are "search objects", i.e. sets of query parameters which support `IBusinessObject`, but are never stored in the database (and thus don't qualify for being domain objects).

Object persistence framework ("re-store layer")

re-motion's object persistence framework is an object/relational mapper like NHibernate, but more specialized and simpler. People familiar with the genre of O-R mapping probably know that various approaches exist, each with its own benefits and shortcomings. The most simplistic approach is to give each type one table, with each property a column in the table and each instance a row in the table. In a world without class inheritance this approach is extremely straightforward and practical. In a world WITH class inheritance, a single table can be used for not only the properties of the base class, but all properties of all subclasses. In this approach, all properties of all derived classes are collected in a single table. An instance of a class that does not support a certain property simply leaves it NULL. This approach works, but it has two essential problems:

- A property in an instance cannot be NULL, because NULL is reserved for "property does not exist in the class in first place"
- If the inheritance tree is very large, the number of columns makes the strategy uneconomical

This simplistic method is called "single table inheritance". For what it is worth, retrieving data from it is extremely efficient because it fits the relational model quite naturally.

O/R mappers differ on how to deal with these issues. The best-known routes around it are

- concrete table inheritance
- class table inheritance

Concrete table inheritance gives each class its own table, be that base class or subclass. In contrast to the previously discussed single table-approach, not all properties are stored in a single table. Each derived class gets its own table. The properties (or better: columns) from all the base classes are

simply copied into the schema for the subclass, and the new properties added as new columns. This approach reduces the number of columns, but there is still overhead across tables along the lines of ancestry.

For (purist) database people, *class table inheritance* is probably the most conventional approach. It solves the problem of inheritance by essentially turning it from the "is a" nature of inheritance into the "has a" nature of composition and then uses tried and trusted normalization techniques for expressing who has what. This approach keeps the number of columns at a minimum, but properties of a given instance of a subclass must be collected from multiple tables.

So what approach does the re-store layer use to persist domain objects?

At the time of this writing, re-motion does not support class table inheritance. The good news is that you can use a combination of single table inheritance and concrete table inheritance for good compromises. The details depend on the application and domain practices. The question "which approach is best for which classes" is decided before the deployment, and based upon analysis of the number of instances expected for various classes and how often they will be used. Such analysis and composition of good compromises is an important part of the expertise of re-motion consultants. The support for class table inheritance is planned for a future release of re-motion. Section *Backgrounder: persisting objects in tables* gives a more detailed introduction to the three types of O/R-mapping, and also how a combination can be used.

Aspects of domain objects in various layers

In its most abstract form, domain objects, paper-forms, C#-objects, rows in a database table and many other things can be seen as typed containers for "attributes" or "properties" or "fields". Depending on the nature (or "type") of the container certain properties are possible or even required. Such typed aggregates of properties can be represented in very different ways. Converting between paper forms, forms in a browser, C#-classes, (single) database tables and XML is usually a fairly straightforward (boring) affair that asks for automation. Doing precisely that essentially is the bread and butter of all content- and document-management frameworks, with each product competing by carefully balancing flexibility and convenience, automation and generalization, and developer effort and execution speed. re-motion provides carefully designed machinery for deriving various "products" from a single description of a domain object:

- the appearance of control elements in forms in a browser
- database schemas
- the structure of and relations between domain objects

The single description of a domain object is a regular abstract C# class, but annotated in compliance with an annotation convention, that is, metadata as C# attributes. From such an annotated abstract class a database schema and a concrete subclass are generated. Why the annotation? On first thought it might seem that a schema can be derived from the data types in the class alone, along the lines of:

- each integer (`int`) in the class becomes an `INTEGER NOT NULL` in the table definition
- each character array becomes a `VARCHAR` of the same length as the character array
- each optional (`nullable`) integer (`int?`) becomes an `INTEGER` in the table definition

- and so on

In accordance with this thinking, the BOC-derivative is formed along analogous lines:

- each integer in the class becomes a text field with a plausibility check whether the entered text is, in fact, an integer
- each character array becomes a text field limited to the same length as the character array
- each optional integer becomes a text field with a plausibility check whether the entered text (if any) is an integer
- and so on

So why the annotation?

This simplistic automation for deriving database schemas and user control elements fails

- for references (because you can't tell from the type whether they are optional or not)
- for read-only properties
- for security information ("access types")

Such information must be tacked on member variables in attributes. By re-motion convention, a handful of attributes exist, not only for nullability and read-only, but also for

- string length
- relationships between properties and other domain objects
- specification of access control privileges (for security)
- type of O/R-mapping

Anything important to say about domain objects and their properties that cannot be expressed in C# must go into annotating attributes.

Transactions

Transactions in re-motion are called "client transactions", or "sub-transactions". They are different from common database transactions, because they are rooted in the realities and requirements of domain objects, not database entities. In re-motion, domain objects have no life outside transactions, because domain object data is loaded from the database into a transaction. Since domain objects can relate to each other in a hierarchical fashion - companies can have employees, and employees can have a company car and a company phone, for example - sub-transactions can begin as soon as a domain object referred by a parent object is needed in memory. For the time of the sub-transaction, the parent object is locked from writing, i.e. read-only. How transactions and sub-transactions are nested depends on how domain objects are used by the user in a "unit of work", and how the used objects relate to each other.

What's more, transactions always come with a *transaction scope*. The transaction scope limits for how long a transaction may last within the code-snippet where the transaction is declared and used. This makes programming with transactions a fairly straight-forward and safe affair and makes sure that no unused domain objects clog resources. The programmer can always infer which code belongs to which transaction simply by looking at where the code is written and which transaction is associated with the given passage.

Relationships

re-motion supports unidirectional and bidirectional relationships between domain objects. A relationship simply is a reference property in one domain object to another domain object. The simpler and less complicated sort is the unidirectional relationship: The domain object `Car`, for example, might contain a property `SteeringWheel` referencing the `SteeringWheel` domain object. This is a 1:1 unidirectional relationship. A slightly more complicated example is that of four `Wheel` domain objects, each with a property `Car` referencing the `Car` domain object to which that wheel belongs. Unidirectional relationships more or less work like plain 1:n relations in database tables. Instead of the foreign key identifying the "owner" of a row (by ID), a domain object identifies its "owner" with a .NET reference.

Bidirectional relationships are more complicated, but also more rewarding, because re-store takes care of many automations. If the relationship between `Car` and `SteeringWheel` is bidirectional, then both domain objects know each other. Car references `SteeringWheel` in a property; `SteeringWheel` references `Car`. As soon as the reference on one side is removed or changed, the other side is updated automatically. This works for bidirectional 1:n relations as well, only that the "n"-side is an `ObjectList<T>` property. As soon as I add another `Wheel` to the `Car's ObjectList`, the added `Wheel's Car` property is set to reference that `Car`.

Web execution engine ("re-call layer")

re-motion's web execution engine builds on ASP.NET and implements basic navigation between web pages. re-call (also called "WXE" in the source code) provides a framework that automates large parts of the programming for session management, integration with transactions, memory management, navigating between hierarchies and more.

The re-call engine typically treats pages as *functions*, complete with a (page) stack, variables local to the "function", parameters and exceptions that can be thrown in a called "function". Just as in ASP.NET, a page is essentially a set of event-handlers. Code in an event-handler might spawn or detour to other pages when controls are used. By providing clever wrappers and program generation, the WXE engine makes this spawning of pages more like regular programming by giving the programmer the illusion that he is invoking functions instead of spawning pages. Just like regular functions, his WXE pages communicate over in-, out- and in/out-parameters and return values. And just like regular functions, WXE pages can have local variables that are invisible to the pages that spawned them, because they are created together with an invocation and deleted when the function/page returns or is cancelled by the user.

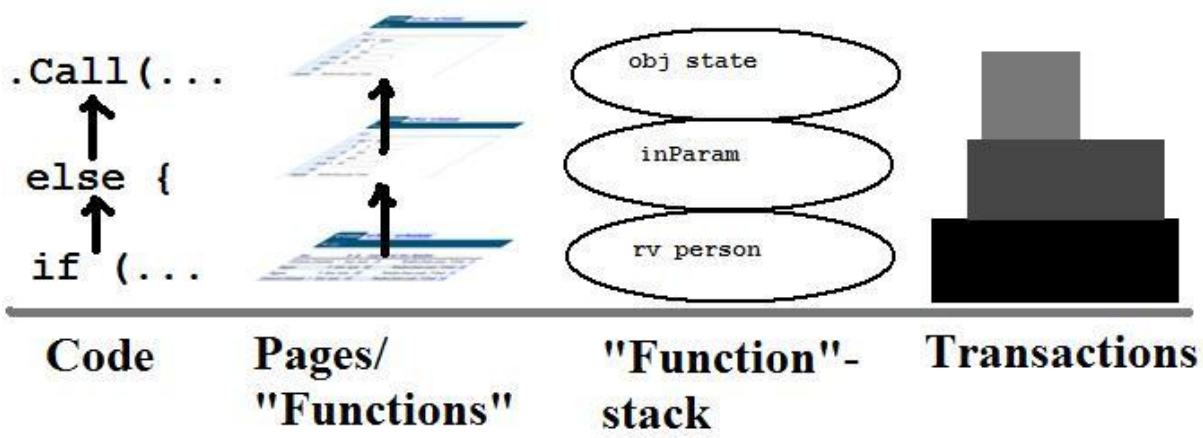
The practice of using a *return stack* for piling return addresses and local variables, parameters and return values on top of each other has been known to programmers for decades. For the WXE, spawning pages means piling those pages and their local data on top of each other on a *page stack*.

If a spawned page throws an exception, it can be catched by any spawning page down the page stack.

All this tried and trusted behavior is supported by the common programming model, i.e. to the programmer it is almost completely transparent that he is programming for pages, not regular C# functions. In other words, the WXE makes the page stack behave if it was a regular return stack.

Transaction-handling can span several web pages and can include sub-transactions. The management of commits upon successful completion of a function (that is, a page) or rollback upon failure or user cancellation is automated by adding appropriate declarations to the page(s).

A very useful consequence of this philosophy is that re-call functions, transactions and web-pages are put on parallel stacks together, so that the top of each stack always corresponds to the "current" domain object -- the domain object the user is browsing or modifying. If the user has open a form for editing company data, then the form (page), the transaction and the re-call function are the top of the stack. If the user clicks on an employee in an employee list of that company, a sub-transaction is created, a new re-call function called and the corresponding form for editing employee data is displayed in the browser. The new top of the stack is that sub-transaction and re-call local data (parameters, local variables). As soon as the employee-form is committed, each of the corresponding element vanishes from the top of each stack and the company-form comes into view again, and with it the previous state of the application. In this fashion, web-programming resembles working with plain functions piling modal dialogs on top of each other. Here is an illustration of the various stacks working in lock-step:



This architecture keeps things simple for typical use-cases, because programmers have little difficulty keeping track of a stack, what aids both coding and debugging.

Code generation in re-motion

Code-generation - "meta-programming" - is an important characteristics of re-motion.

- The command-line tool dbschema.exe generates database schemata for persistence from the C#-declaration of domain objects.
- The command-line tool uigen.exe generates a basic ASP.NET-application from those same C#-declarations
- The command-line tool wxegen.exe generates boilerplate code for re-call-supported navigation between pages
- Behind-the-scenes run-time code-generation (and compilation) manages specialized and flexible sub-classes via "mixins"

The derived code for various representations of domain objects snaps nicely into the corresponding layers for persistence (re-store) and representation (re-bind). Each layer is generic and makes no assumptions on how domain objects are structured or of what type they are. The mapping of domain objects and properties between layers (representations) is entirely driven by the code and the (derived) information in the domain objects themselves. Since the descriptions of domain objects and their relations are fairly concise and good defaults could be found, very little work can get you a long way with re-motion.

The annotated abstract class written by the programmer becomes the basis for the generated concrete class and the corresponding generated database schema. The abstract class with its annotations specifies everything re-motion must know for sensibly translating between storage, runtime representation, display and user editing of domain objects and their relations amongst them.

Security manager ("re-strict layer")

Security (access control) in re-motion is very flexible and fine-grained. So-called access control lists specify what privileges users and groups have. Privileges for users and objects are neither limited to the familiar read-write-execute nor are they global for entire domain objects:

- privileges in access control lists can be assigned to individual properties
- subclasses inherit access control lists from superclasses, but individual properties can be excepted and override inherited access control lists with their own
- domain objects pass their access control lists to each of their individual properties (this inheritance along "has-a" lines is called "acquisition")
- privileges may depend not only on the class of an object, but also on its state in the workflow

Security is provided by two parties:

- a server (decides what a given user may or may not do)
- a client (enforces what the server decides what a given user may or may not do)

Some filtering of data is done by the re-store layer in accordance with the access control attributes and user privileges.

The *security manager* essentially is a service provided by the server's (security manager's) interface `ISecurityProvider`. This service can literally be provided by a server (i.e. a machine communicating via XML) or can be dynamically linked to the client. For the client is transparent which of the approaches has been chosen. For the client only the interface is relevant.

Since requests to the security manager are frequent (at least one for each user action on a domain object) and can be expensive (network communication), a client caches the security manager's replies to requests. People in the know might wonder how the problem of updating the cache is solved. What if information in the cache becomes stale?

Two rules exist:

- a cache is never updated during a transaction
- any modification of server configuration invalidates all caches in all clients

These are simple and practical assumptions. Modifications of user access privileges ("writing" the server) are much less frequent than querying the server ("reading"). re-strict is not covered by this tutorial, by the way. A separate document on re-strict is planned for the near future, however. You can learn a little more about re-strict by reading the fascicle "re-strict Dictionary and Overview" (https://re-motion.org/content/re-strict-dictionary-and-overview.pdf)

Working with domain objects

The phone-book specification

The phone-book is a sample application for storing people's names, their home addresses and their phone-numbers. We assume the following for our phone-book:

1) a person can have multiple phone-numbers, but a phone-number can never be shared by multiple persons
2) addresses (buildings) are called "locations" and an arbitrary number of people can be related to the same location

For locations, only the number of the building is stored, no apartment numbers are given.

As explained in the *What is re-motion?* chapter, there are differences between how re-store manages unidirectional relations and bidirectional relations, and how either is reflected in the database. This primer will elaborate on these differences in detail.

For the first phone-book requirement, we use a bidirectional relation between person objects and phone-number objects. A person object has a list of references to all its phone-number objects. Each phone-number object in turn has a back reference to the person object it belongs to (that's the BI in bidirectional). For the second requirement, the person's address, we use an unidirectional 1:n relation, i.e. a person object has a relation to a location object (in its `Location` property), but `Location` classes have no property for pointing back to the person(s) who live at that address -- that's the meaning of "unidirectional" here. This is in contrast to the relationship between person objects and the phone-number objects, which mutually reference each other. In such a bidirectional relationship, both the phone-number objects and the person objects have a property with references to the referenced object(s) of the other type.

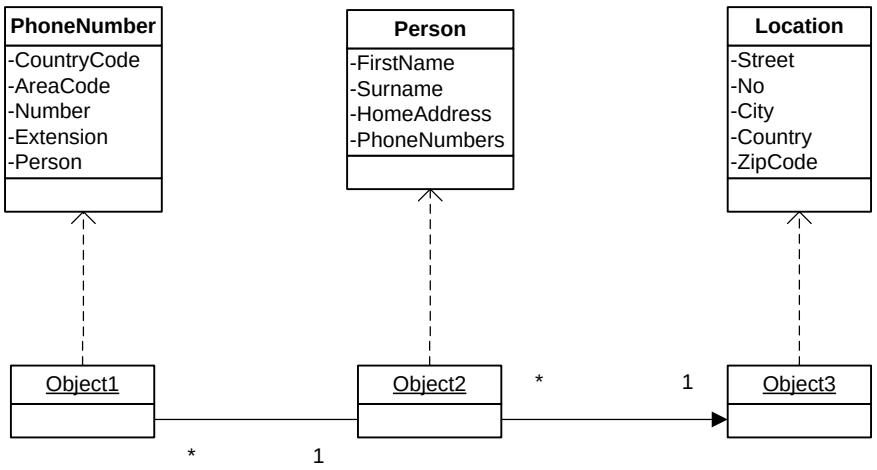
This design is chosen for economical reasons, because bidirectional relations have an interesting side-effect in re-motion: If you access a property referencing another domain object, the referenced domain object gets loaded into memory from the persisting store (i.e. .NET instances are created from database entries). If the property is a list of object references, all the domain objects referenced in the list get loaded into memory. This is okay for people referencing their phone-numbers, because we expect a person to have not more than a handful of phone-numbers. What's more, by implication, if an object in a list in a bidirectional relationship is needed, it loads the parent object on the other side into memory - *and all its siblings in the list*. This is not desired if there are **many** siblings.

A location (building) however, can accommodate many people, potentially hundreds or thousands. If this relation was stored as a bidirectional relation, for, let's say, the Trump World Tower, every reference to a property with an object list with all persons would bring all the corresponding people objects in the object list into RAM as well. For this reason, we use a *uni*directional relationship, because accessing the location property will get only that location object into memory, and nothing

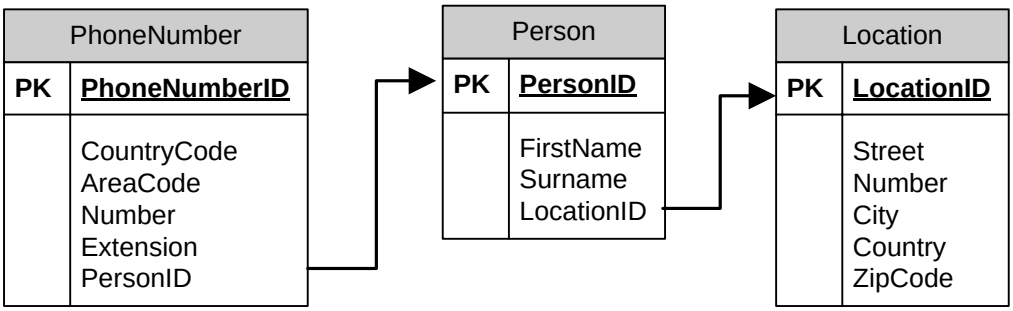
else. The important thing to remember here is this: not all relations are equal in terms of memory efficiency. We will treat lazy loading in more depth in the "Under the hood" section.

Our Phone-Book data as UML and in the database tables

The following UML diagram shows how *Person* objects, *PhoneNumber* objects and *Location* objects can relate to each other as .NET object instances. A *Location* can be populated by multiple *Persons*; *Persons* can have multiple *PhoneNumbers*. Please note that a .NET *Location* has no back reference to its *Person* objects (hence *uni*directional). Contrast this with the association between `Persons` and `PhoneNumbers`, which link back to each other in the corresponding fields.



These subtleties of unidirectional and bidirectional relations are only skin-deep, because the database schema for storing those interconnected instances does not reflect them at all. Each class simply gets its own table; each row stores an instance. The structure in the illustration above is mapped into the following tables and relations for persistence as in this diagram:



As you might have guessed, the relations between domain objects are established with foreign keys. The *LocationID* column in the *Person* table relates to a row in the *Location* table, the *PersonID* column in the *PhoneNumber* table relates to a row in the *Person* table.

As hinted at in the section *What is re-motion?*, the framework's object persistence layer re-store will derive the schema from adequately annotated class declarations automatically. However, as we will see later in this guided tour, for *unidirectional* relations no annotations at all are necessary for re-motion to automate the mapping to the database in a meaningful way. re-motion simply takes a .NET declaration in the `Person` class like

```
Location Location { get; set; }
```

to mean "there is a unidirectional relationship between `Person` and `Location`", and there will be a foreign key in the `Person` table to its `Location` row in the `Location` table. Things are not as easy for bidirectional relationships, as we will see shortly.

If domain objects were not persisted in database tables we'd have difficulty finding all *Person* objects belonging to a given *Location*, because the .NET *Location* object can't be asked for it. Since domain objects are always persisted in a database, however, we can query the *Person* table for a given location and wrap up the query in a member function for the *Location* class. For bidirectional relations, re-motion generates such queries for keeping the .NET properties in sync with the database for you, and makes sure they work correctly at both ends of the relation. A unidirectional relation, however, lacks this automation in the other direction, in this case the query for getting the persons belonging to a location. Section *Querying your objects and classes* will explain how to supplement such a query and integrate it into the *Location* class.

With the basic structure of our phone-book nailed, let's turn our attention to the nuts and bolts and write some code. First, let's set up a project for a library containing the "domain", i.e. the code and data for our three types of domain objects that is independent from any user interface or storage considerations.

Later, we will set up another project in the same solution for the client for this library, i.e. an application that uses the library.

So we will set up a Visual Studio solution and a class library project in that solution first:

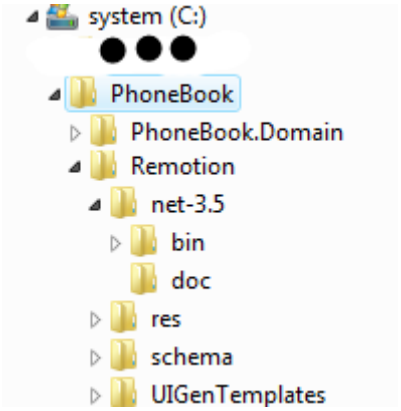
- In Visual Studio, create a new project of type "class library"
- Name the project `PhoneBook.Domain` and the solution `PhoneBook` -- **MAKE SURE THIS NEW FOLDER IS DIRECTLY UNDER C:\, as in C:\PhoneBook**. This is important for running `uigen.exe` later. (`PhoneBook.Sample` will later be the project for the client app in the same solution.)
- Rename the default file `Class1.cs` to `PhoneNumber.cs` and answer the question if all references shall be renamed with "Yes".

At this point, you should have a single `PhoneNumber.cs` file in your `PhoneBook.Domain` project, looking like this:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PhoneBook.Domain
{
  public class PhoneNumber
  {
  }
}
```

Next, we have to add some re-motion assemblies to the project. Again, for running uigen.exe later, it is important that uigen.exe will find the re-motion assemblies where it expects them. For this reason,

copy the entire Remotion folder (the "distribution directory", or "distribution folder") into the C:\PhoneBook folder, so that the entire tree looks like this:



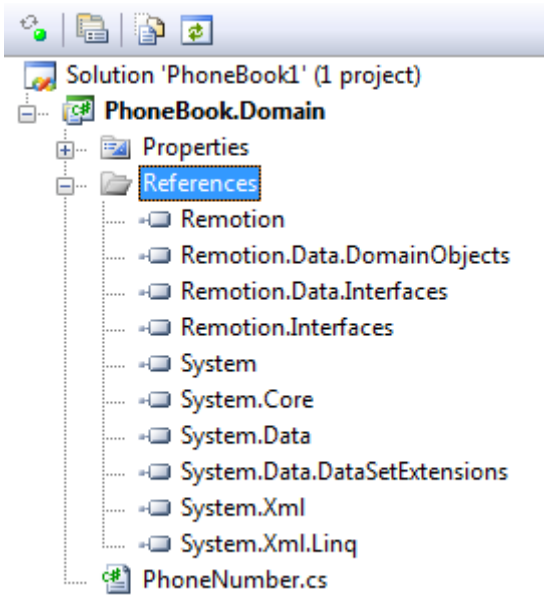
When you are sure your folder structure looks as proposed as in the picture, add the following references to your `PhoneBook.Domain` project:

- `Remotion.dll`
- `Remotion.Interfaces.dll`
- `Remotion.Data.DomainObjects.dll`
- `Remotion.Data.Interfaces.dll`

You'll find these .DLLs in your re-motion binary directory among other re-motion assemblies. For this primer, let's agree on a simple convention here. We assume that the re-motion distribution directory is mapped to "R:". The distribution directory is the one that contains these directories:

- `net-3.5`
- `res`
- `schema`
- `UIGenTemplate`

After adding the assemblies, the references section in your solution explorer should look like this:



Now the `PhoneNumber` class in your `PhoneNumber.cs` file has no ancestor and consequently is not a domain object. Change this by making it a subclass of `DomainObject`. At this point it is time to declare the `usings` for re-motion types. Add to all class files in this library project the declaration

```
using Remotion;
using Remotion.Data.DomainObjects;
```

This being said, flesh out the class by adding the following members to the `PhoneNumber` class:

```
namespace PhoneBook.Domain
{
  public class PhoneNumber : DomainObject
  {
    public virtual string CountryCode { get; set; }
    public virtual string AreaCode { get; set; }
    public virtual string Number { get; set; }
    public virtual string Extension { get; set; }
    public virtual Person Person { get; set; }
  }
}
```

The data type `Person` is not yet known to Visual Studio, but we will fix that in a minute. The meaning of each property should be obvious to most people who are regular users of telephones:

- `CountryCode` is the prefix that dials a country, like 0043 for Austria, 0049 for Germany or 0044 for UK. (Pedantic people in the know will point out that what we are talking about here is the ITU-T recommendation E.146.)
- `AreaCode` is the prefix that dials a city or region in a country, like 89 for Munich, for example, or 212 for Manhattan. In Austria, mobile phone carriers have their own area codes, like 676 for *T-Mobile*.
- `Number` is the local phone number, of course.
- `Extension` is the extension, typically for office phones.

- `Person` is a reference to the person owning that phone-number and a reference to the `Person` object. This does not compile yet, because we have yet to declare the `Person` class.

To people familiar with C# 3.0 the laconic `{ get; set; }` may look unremarkable , because the precise meaning of the laconic `{ get; set; }` has been extended from C# 2.0 to 3.0. Here is a re-cap:

- In C# 2.0 the `{ get; set; }` notation is only permitted for abstract properties and means "will be implemented in a concrete subclass"
- In C# 3.0 the `{ get; set; }` notation is also permitted in concrete classes and in this case means "the compiler shall fill in the boring boilerplate for `get{ return _val; } set { _val = value }` for you.

HOWEVER, in re-motion, where subclasses of `DomainObject` are used for generating code at run-time, a `{ get; set; }` property signals "fill in re-motion-specific boilerplate here, dear run-time code-generator". In other words, re-motion does not care about either 2.0 or 3.0 semantics here, it interprets `{ get; set; }` in its own advanced ways. The "re-motion specific boilerplate" is explained in the section *The secret life of properties* of this chapter.

This brings us to the next steps. In order to declare `Person`, put that class into its own file named `Person.cs`. Make sure that the class `Person` contains the following code:

```
using System;
using System.Collections.Generic;
using System.Text;
using Remotion;
using Remotion.Data.DomainObjects;

namespace PhoneBook.Domain
{
  public class Person : DomainObject
  {
    public virtual string FirstName { get; set; }
    public virtual string Surname { get; set; }
    public virtual ObjectList<PhoneNumber> PhoneNumbers { get; set; }
    public virtual Location Location { get; set; }
  }
}
```

The last property, `Location`, sports an as yet undefined data type, `Location`, but here it comes. Put that class into its own file named `Location.cs`. Make sure that class contains the following code:

```
using System;
using System.Collections.Generic;
using System.Text;
using Remotion;
using Remotion.Data.DomainObjects;

namespace PhoneBook.Domain
{
  public enum Country
  {
```

```
    Austria = 0,
    Australia = 1,
    BurkinaFaso = 2
  }

  public class Location : DomainObject
  {
    public virtual string Street { get; set; }
    public virtual string Number { get; set; }
    public virtual string City { get; set; }
    public virtual Country? Country { get; set; }
    public virtual int ZipCode { get; set; }
  }
}
```

Just as with phone-numbers, these properties should be fairly self-explanatory. For example, the address

"Berggasse 19, 1090 Vienna, Austria"

is stored as

- Street=Berggasse
- Number=19
- City=Vienna
- Country=Austria (or 0, because we use an enumeration here)
- ZipCode=1090

If your phonebook is like the author's (paper) phonebook, the typical addresses in it are local to your hometown, so it makes sense to make the `Country` property optional (that is, `Country?`)

Enums in re-motion

A word, and a warning, on the `enum` declaration for `Country`. Enumerations are stored as plain integers in the database, and they must never change over time, so no creativity is allowed here. In contrast to .NET objects in RAM, this enumeration *does* have a memory in the database (this is, after all, the aim of persistence to begin with). If the enumeration ever changes, your country IDs will be confused in unpredictable ways, so it is important that it never does. For this reason, you should NOT leave it to the compiler to invent them for you. Always specify each enumeration explicitly and start your enumeration with 0 (zero).

The unidirectional relationship

As you can tell from the declaration, `Location` does not contain any references to `Person` (or a `PhoneNumber`, for that matter). This is important, too, for the reasons explained above in *The unidirectional relationship* section. In a nutshell: object instances are pulled into RAM as .NET objects along explicit references, and we don't want that for `Location` objects.

At this point, your project should contain the three class declarations in the files

- `Person.cs`

- `Location.cs`
- `PhoneNumber.cs`

and your `PhoneBook.Domain` project should compile and link. Try it now, good luck. Not much can go wrong in terms of re-motion problems, because there is virtually no re-motion in this project yet.

From declaration to schema

So far, there is not much that makes our `PhoneBook.Domain` library any different from a conventional, non-persisted object design. In this section, we will annotate our class declarations with attributes and have a command-line program named `dbschema.exe` generate the database schema for us.

Annotating attributes give both `dbschema.exe` and the *mapping loader* important hints on the plan the programmer has for various properties. Let's begin with annotating `Location`.

The two properties `Number` (an `int`) and `Country` (a `Country enum`) require no annotation at all for our purposes, because both the `dbschema.exe` and the code generator can infer all they need to know from the .NET data types. The `int` type of the `Number` property is mapped to... an `INT` in the schema, or better: `INT NOT NULL`, for an int is not nullable. Contrast this with the `Country? enum`. It is mapped to an `INT` as well, but of the nullable sort.

Consequently, nothing changes for the `ZipCode` and `Country` declarations, since no annotations are necessary. These are not only perfect .NET declarations, but also perfect re-motion declarations:

```
// not nullable, because int is not a nullable type
public virtual string ZipCode { get; set; }

// nullable, because Country is a nullable type
public virtual Country? Country { get; set; }
```

Contrast this with the properties `Street`, `City` and `Number`. They are all strings and therefore nullable in C#. What's more, `dbschema.exe` doesn't know from the plain-vanilla C#-declaration how long each string may become in the database. This information must be provided in appropriate declarations. Maximum length and nullability is a matter of taste in a toy application like phone-book, but let's not be too fancy and agree on the following: Street names can never be nullable and cannot become longer than 40 characters.

City names and street numbers can be nullable (again, most addresses in most people's phone-books are local to their home-town). City names are limited to 30 characters, zip codes to 6 characters.

The attribute for expressing this is the attribute `StringProperty`. In compliance with our aforementioned agreement, we annotate the three string properties in the `Location` class like this:

```
[StringProperty (MaximumLength = 40, IsNullable = false)]
public virtual string Street { get; set; }
[StringProperty (MaximumLength = 6, IsNullable = true)]
public virtual string Number { get; set; }
[StringProperty (MaximumLength = 30, IsNullable = true)]
```

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

As you can see, the attribute parameter `IsNullable=false` overrides the fact that `string` is a nullable type in C#. `dbschema.exe` gives such a parameter precedence over what it can infer from the raw C# type declaration. In the database, the VARCHAR used for `string` properties attributed with `IsNullable = false` will be declared as `not nullable` in the schema. Since `dbschema.exe` infers nullability in the schema from the nullability of the corresponding C#-type, the spec `IsNullable = true` is redundant here and was listed for illustration only.

Consequently, your class declaration for `Location` should look like this now:

```
public class Location : DomainObject
{
  // for strings, no values are required, because strings are
  // a nullable type. You can override this by setting
  // the "IsNullable" in the attribute to "false"
  [StringProperty (MaximumLength = 40, IsNullable = false)]
  public virtual string Street { get; set; }

  [StringProperty (MaximumLength = 6)]
  public virtual string Number { get; set; }

  [StringProperty (MaximumLength = 30)]
  public virtual string City { get; set; }

  // values not required, because Country is a nullable type
  public virtual Country? Country { get; set; }

  // values required, because int is not a nullable type
  public virtual int ZipCode { get; set; }
}
```

As pointed out before (section *The phone-book specification*), `Location` is seemingly not related to any other domain object. This loneliness is only an illusion, however, because `Locations` are related to `Person` objects in the database, as we will see. Due to its apparent loneliness, `Location` is a fairly boring class. What makes domain object classes interesting is their relationships, as we will see in our next example, `PhoneNumber`.

The maximum lengths for country code, area code, actual phone number and extension are a matter of taste, but, again, since most of your phone-numbers are probably local, it makes sense to make everything optional except for the actual phone number (in the number property). Consequently, you can annotate these properties along the same lines illustrated by the previous `Location` example and make them look like this:

```
    [StringProperty(MaximumLength = 5)]
    public virtual string CountryCode { get; set; }

    [StringProperty(MaximumLength = 6)]
    public virtual string AreaCode { get; set; }

    [StringProperty(MaximumLength = 12, IsNullable = false)]
    public virtual string Number { get; set; }

    [StringProperty(MaximumLength = 6)]
```

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

Things become interesting for the `Person` property, because each `Person` instance points to multiple `PhoneNumber` instances. Since it is a bidirectional relationship, the attribute `DBBidirectionalRelation` applies, and its parameter tells dbschema.exe and the code generator that the relationship has its other end in `Person's` property `PhoneNumbers`:

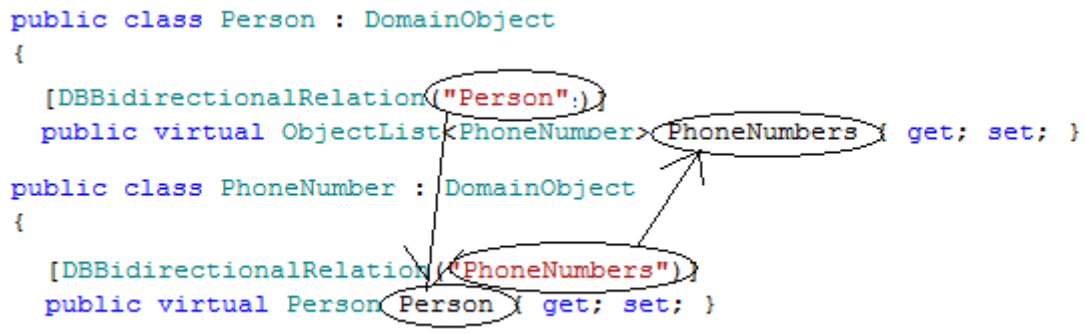
```
[DBBidirectionalRelation("PhoneNumbers")]
public virtual Person Person { get; set; }
```

This brings us to the class in `Person.cs`.

Class `Person` in turn sports *two* relations. One to the seemingly lonely `Location` class, the other with the aforementioned `PhoneNumber` class. Let's turn to `PhoneNumber` first, because it is more straightforward. In class `Person`, annotate the class `PhoneNumbers`, as referenced in `PhoneNumber's` annotation for its relationship with `Person`, with the complement of that declaration. That is, annotate the `PhoneNumber` property with a reference to `Person` (clipping):

```
[DBBidirectionalRelation("Person")]
public virtual ObjectList<PhoneNumber> PhoneNumbers { get; }
```

Here is a picture for the visually literate readers:



This makes the bidirectional declaration complete, but `DBBidirectionalRelation` is also the spot where you can specify a sort order for the phone-numbers in the `PhoneNumbers` object list. You do this by supplementing an SQL fragment for the sorting order, for example:

```
"CountryCode, AreaCode, Number, Extension"
```

or

```
"Number ASC"
```

For practical reasons and from years of experience with telephones and phone-books, we recommend the following declaration:

```
[DBBidirectionalRelation("Person",
                         SortExpression =
                           "CountryCode, " +
                               "AreaCode, " +
                                 "Number, " +
                              "Extension")]
public virtual ObjectList<PhoneNumber> PhoneNumbers { get; set; }
```

Please note that this sort order specification is NOT the one for sorting `Person` objects in a `Person` list. This is the sort order for `PhoneNumber` objects listed as belonging to the `Person` object.

Relating a `Person` object to a `Location` is easier to do here than to explain, because you don't have to do anything here. The hard part is to understand why. The back reference from the unidirectional relationship exists only at the deeper persistence level in the database and is not explicitly stated in any attribute. For this reason, the boring, unannotated declaration

```
public virtual Location Location { get; set; }
```

is totally adequate for our purpose. Lacking other instructions, `dbschema.exe` will conclude that it must reflect the .NET reference in `Location` in some way in the database and simply puts an extra column into the `Person` table for the `Location's` GUID (because that's what re-motion uses as object IDs in the default implementation). This, not by coincidence, is exactly what we want here.

After these proposed modifications, your class declarations should look as in the following listings.

```
public class Person : DomainObject
{
  [StringProperty (MaximumLength = 20, IsNullable = true)]
  public virtual string FirstName { get; set; }

  [StringProperty (MaximumLength = 20, IsNullable = false)]
  public virtual string Surname { get; set; }

  public virtual Location Location { get; set; }

  [DBBidirectionalRelation("Person",
    SortExpression = "CountryCode, AreaCode, Number, Extension")]
  public virtual ObjectList<PhoneNumber> PhoneNumbers { get; }
}


public class Location : DomainObject
{
  [StringProperty (MaximumLength = 40, IsNullable = false)]
  public virtual string Street { get; set; }

  [StringProperty (MaximumLength = 6, IsNullable = true)]
  public virtual string Number { get; set; }

  [StringProperty (MaximumLength = 30, IsNullable = true)]
  public virtual string City { get; set; }

  public virtual Country? Country { get; set; }

  public virtual int ZipCode { get; set; }

}

public class PhoneNumber : DomainObject
{
  [StringProperty(MaximumLength = 5, IsNullable = true)]
  public virtual string CountryCode { get; set; }
```

```
    [StringProperty(MaximumLength = 6, IsNullable = true)]
    public virtual string AreaCode { get; set; }

    [StringProperty(MaximumLength = 12, IsNullable = false)]
    public virtual string Number { get; set; }

    [StringProperty(MaximumLength = 6, IsNullable = true)]
    public virtual string Extension { get; set; }

    [DBBidirectionalRelation("PhoneNumbers")]
    public virtual Person Person { get; set; }
  }
```

After these modifications, you should still be able to build the library without errors, but you have to add a `DBTable` attribute to each of the classes.

The `DBTable` attribute

The classes `PhoneNumber`, `Location` and `Person` must be also be attributed with `DBTable` in this guided tour. This is another important clue for the schema generator `dbschema.exe`, which we will discuss in the next section: `DBTable` tells the schema-generator that the attributed class (and by extension, all its sub-classes) shall get its own table in the database, with a column for each property and a row for each instance. More refined schemes exist (see section *Table inheritance, attributes and views*), but they are beyond the scope of this guided tour of the phone-book toy application.

Putting it all together

Supplementing the `DBTable` attribute is easy. Just put them at the class declaration level, as in this example for Location:

```
  [DBTable] <--- New!
  public class Location : DomainObject
  {
..... --- more code ---
```

Do the same for `Person` and `PhoneBook` as well.

These declarations are sufficient for generating the database tables with `dbschema.exe`. In order to actually do this, you must invoke `dbschema.exe` on the command line. You find `dbschema.exe` in your re-motion binaries directory. In this primer, you need to call the tool only once, but in a real-life project you probably want to integrate `dbschema.exe` into your build-process to run automatically after each modification to one of your declarations. `dbschema.exe`'s most important switches here are:

- `/schema` -- instruction to create a schema
- `/baseDirectory` -- specifies the directory where the domain DLL for processing is located
- `/verbose` -- gives you more precise error messages

(You can inspect the switches with `/?`)

The output file `SetupDB.sql` will be dumped into the working directory where `dbschema.exe` is executed.The executable itself is located in the

```
C:\PhoneBook\Remotion\net-3.5\bin\Debug
```

build directory (i.e. the re-motion directory). If you invoke `dbschema.exe` in the directory `\Phonebook` and have your re-motion directory mapped to R: (recommended), then your invocation looks like this:

```
C:\PhoneBook\Remotion\net-3.5\dbschema /schema /verbose
/baseDirectory:PhoneBook.Domain\bin\Debug
```

`dbschema.exe` will run for a split second and give you a file named `SetupDB.sql` -- this is the file you are supposed to import into your Microsoft SQL Server 2005.

`dbschema.exe` opens each DLL in the `baseDirectory` and inspects them for classes derived from `DomainObject`. Every subclass of `DomainObject` which `dbschema.exe` finds is clearly a candidate for persistence, so the program scours all these classes and generates the schema according to type declarations and attributes. At this time you should create a database in Microsoft Server 2005 to import your database script. **PLEASE NAME YOUR NEW DATABASE "PhoneBook".**

Before you paste the entire script into the query to run, **edit the first line of it, the one that says**

```
USE DataBaseName
```

to

```
USE PhoneBook
```

This benign case of manual correction is not the usual way to operate in a production environment and will be fixed later in this primer. For now it is the quickest way to go. After this modification, simply run the script and step back. The script will run for a few seconds and should complete successfully.

If the `SetupDB.sql` script has run successfully in your database client, you can inspect the tables of the `PhoneBook` database. The three tables

- `dbo.Location`
- `dbo.Person`
- `dbo.PhoneNumber`

should all be there. Inspecting each table's columns, you should find one column for the properties declared in the domain objects, and a few more.

If all this has worked out, you can proceed to the next section, "Enhancing and using the PhoneBook.Domain library". Just in case it has not worked out, the next section gives hints on troubleshooting.

What can go wrong

Wrong `baseDirectory`

A typical mistake here is to give `dbschema.exe` a wrong `baseDirectory`, that is, one with no DLLs or none with `DomainObject` subclasses. This will give you the error message:

```
Execution aborted: Argument classes is empty.
```

If you get that error message, make sure the `baseDirectory` is correct for the spec in the `/baseDirectory` switch.

Typos

If you get the specification for the property at the other side of a relation wrong, the compiler will not warn you, but dbschema.exe will complain with this error message:

```
Execution aborted. Exception stack:

Remotion.Data.DomainObjects.Mapping.MappingException: Opposite
relation property 'PhoneNumber' could not be found on type
'PhoneBook.Domain.Person'.

Declaring type: PhoneBook.Domain.PhoneNumber, property: Person
```

This happens if you write, for example, "`PhoneNumber`" instead of "`PhoneNumbers`" in the `DBBidirectonalAttribute` parameter for the opposite property:

```
// Wrong: "PhoneNumber" instead of "PhoneNumbers" -- typo goes
// undetected by compiler
[DBBidirectionalRelation("PhoneNumber")]
public virtual Person Person { get; set; }
```

Missing `DBBidirectionalRelation` *attribute*

If you forget one of the `DBBidirectionalRelation` attributes, you get one of two error messages, depending on which of the Person.cs or Phonenumber.cs file is processed first. It is either

```
Execution aborted. Exception stack:

Remotion.Data.DomainObjects.Mapping.MappingException: The property type
Remotion.Data.DomainObjects.ObjectList`1[PhoneBook.Domain.PhoneNumber] is not
supported.
```

(this is also the message you will get when you forget both `DBBidirectionalRelation` attributes) or

```
Execution aborted. Exception stack:

Remotion.Data.DomainObjects.Mapping.MappingException: Opposite relation property
'Person' declared on type 'PhoneBook.Domain.PhoneNumber' does not define a matching
'Remotion.Data.DomainObjects.DBBidirectionalRelationAttribute'.
```

Forgetting to compile

This evergreen even bites old-timers occasionally: if you fix something in your `PhoneBook.Domain` project and forget to compile, then of course `dbschema.exe` won't notice any of your changes and merrily complain of the same problem until you finally DO rebuild your library.

`dbschema.exe` and remote machines

Please note that `dbschema.exe` will refuse to run from remote machines (.NET policy). If you get a security exception when running `dbschema.exe`, the simple reason probably is that you try to execute `dbschema.exe` from a network share. If you copy dbschema.exe to your local machine,

observe that `dbschema.exe` needs re-motion DLLs just like your application. It should remain in the same directory as the re-motion DLLs.

Mind the SQL!

Even if `dbschema.exe` does not report problems, you might run into problems when trying to run your application later. If you get SQL-exceptions, you might frown upon re-motion and think that it is a bug in re-motion (or re-store, to be more precise). After all, re-store is supposed to hide all those SQL details from you, right? So if there is an SQL-problem, it can't possibly be your problem, right? Almost. The spot where you DO work with SQL is the `SortExpression` parameter in a `DBBidirectionalRelation` attribute. This is an SQL-fragment (for an `ORDER BY` clause) and is copied into the schema on an as-is basis. If you misspell a property name, the database will throw an exception at you as soon as it tries to evaluate that expression. So if you get an "RdbmsProviderException was unhandled/Error while executing SQL command", check the spelling of your `SortExpression` (the same thing applies to the `ContainsKey` parameter, explained in *1:1 bidirectional relationships*).

Enhancing and using the PhoneBook.Domain library

`NewObject<T>`

At this point, all you have is a handful of domain object declarations and a corresponding database schema for persisting those domain objects. No code has been written yet for actually creating domain objects, and some methods must be supplemented to make them useful, even for the modest ambitions of a toy application. In this section, we will discuss how to achieve exactly that and write a simple client application.

As you might guess, you can't simply use default constructors for domain objects. After all, the constructor must provide SOME logic for integration with the persistence framework. What you use instead is a static factory method in your domain object class that should be named `NewObject()`. The baseclass' (`DomainObject`) static method `NewObject` receives the type of the domain object to be created, for example:

```
DomainObject.NewObject<PhoneNumber>()
```

This `NewObject<T>` method is protected, so you can't access it from an application object. The canonical way to declare a specialized `NewObject()` member function for each of your three PhoneBook domain object classes:

```
// in class PhoneNumber in PhoneNumber.cs:
public static PhoneNumber NewObject()
{
  return DomainObject.NewObject<PhoneNumber>();
}

// in class Person in Person.cs:
public static Person NewObject()
{
  return DomainObject.NewObject<Person>();
}

// in class Location in Location.cs:
```

```
public static Location NewObject()
{
  return DomainObject.NewObject<Location>();
}
```

For this toy application, that's all you need to do for domain object creation. Please remember that object instantiation is not persisted in the database before the first `Commit()` in the client transaction where these objects are created (see section *On client transactions and sub-transactions*).

We can now write some client code that creates person objects in client transactions, creates phone numbers and links them to persons (and back -- automatically), creates and manages locations and the persons who live there in meaningful ways -- all by virtue of our declarations. It is easy, for example, to get all phone numbers of a person. Simply look into the Person instance's object list `PhoneNumbers`:

```
myPerson.PhoneNumbers
```

Before we can do this, we must create some objects and link them together appropriately. It is time to start another project, one for the client that uses the `PhoneBook.Domain` library. So:

- Create a new **console application** project named `PhoneBook.Sample` in your Visual Studio `PhoneBook` solution
- Add the three re-motion assembly references you already know from the `PhoneBook.Domain` project:
  - `Remotion.dll`
  - `Remotion.Interfaces.dll`
  - `Remotion.Data.DomainObjects.dll`
  - `Remotion.Data.Interfaces.dll`
- Add an assembly reference to your `PhoneBook.Domain` project

This empty re-motion application should build (but it won't run without exceptions yet). If it builds okay, you are good to go for writing some code that uses our phone-book domain objects.

As mentioned before, creation and modification of domain objects always take place in the context of a client transaction (or sub-transaction). Without this context, you can manipulate domain objects any way you want in memory, without actually enforcing these manipulation. Not before a `Commit()` in the context of a client transaction does anything change in the re-store persistence backend. Setting up a client transaction is easy:

```
using(ClientTransaction.CreateRootTransaction().EnterDiscardingScope())
      {
        // lots of domain object code here
        ClientTransaction.Current.Commit();
      }
```

This `using`-statement creates a *client transaction scope* for your domain object logic. The concluding `Commit()` persists your domain objects or modifications on them in the database. If your code changes its mind about remembering domain object manipulations, it can cancel all modifications by simply not calling `Commit()`. The `EnterDiscardingScope()` specifies that,

as soon as the code leaves the scope, the transaction is discarded and nothing is written to the database. Consequently, if you don't `Commit()` your modifications, nothing happens, database-wise. Client transactions are an interesting topic, and other options exist for transactions. Client transactions are discussed in more depth in section *On client transactions and sub-transactions*.

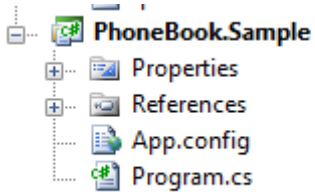
Here is a function you can copy into into your `PhoneBook.Sample.Program` class as a static member function. It creates a `Person` object for Sigmund Freud, complete with his address and phone-number:

```
static void EnterFreud()
{
  using (ClientTransaction.CreateRootTransaction()
                          .EnterDiscardingScope())
  {
    Location loc = Location.NewObject();
    loc.Street = "Berggasse";
    loc.Number = "19";
    loc.City = "Vienna";
    loc.ZipCode = 1090;
    loc.Country = Country.Austria;

    Person person = Person.NewObject();
    person.FirstName = "Sigmund";
    person.Surname = "Freud";
    person.Location = loc;

    PhoneNumber phone = PhoneNumber.NewObject();
    phone.CountryCode = "43";
    phone.AreaCode = "1";
    phone.Number = "3191596";

    person.PhoneNumbers.Add(phone);

    ClientTransaction.Current.Commit();
  }
}
```

If you put a call for `EnterFreud()` into your `PhoneBook.Sample.Program.Main` function, your project should compile and link without problems, but it *won't* run without throwing exceptions. We have to provide some extra information for bolting the components together at run-time.

Using App.config

You must provide a regular Visual Studio configuration file for your application. This file gives important run-time information on how to connect to connect to the database and use it. This *App.config* file belongs to the application, so your ensemble of files should look like this after adding `App.config` as an XML file:



Here is a fragment of a listing of how it is supposed to look like for the phone-book application, with annotations of the critical parts (clipping):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="remotion.data.domainObjects" type="Remotion.Dat
      <section name="storage" type="Remotion.Data.DomainObjects.Persist
    </sectionGroup>
  </configSections>

  <remotion.data.domainObjects xmlns="http://www.rubicon-it.com/Data/Dc
    <storage defaultProviderDefinition="PhoneBookDB">
      <providerDefinitions>
        <add name="PhoneBookDB"                                  1   2
          type="Remotion.Data.DomainObjects::Persistence.Rdbms.RdbmsPrc
          providerType="Remotion.Data.DomainObjects::Persistence.Rdbms.
          connectionString="PhoneBook" />
      </providerDefinitions>                                         3
    </storage>
  </remotion.data.domainObjects>

  <connectionStrings>
    <add name="PhoneBook" connectionString="Integrated Security=SSPI;   4
                                            Initial Catalog=PhoneBook;
                                            Data Source=localhost" />
                                                                  5
  </connectionStrings>
</configuration>
```

Since there is a lot to type for `App.config` (it is XML, after all), we provide a complete ready-made `App.config` file in the directory:

```
<your "instruct" directory>\tutorial\tutorial-files\App.config)
```

(Our lawyers told us we could be liable for repetetive strain injury.)

1.) and 2.) This node gives a list of *provider definitions*, i.e. definitions for finding and using a database storing domain objects, as well as specifying which of the nodes to pick as a default provider definition (2). In this case, the list contains only one provider definition (1), which is also the default. The name given in the name attribute ("`PhoneBookDB`") not only identifies the node within the `App.config` XML file, it is also the identifier used by any application using the config file, which includes our phone-book app.

3.) A provider definition does *not* contain connection strings. A provider definition contains a *reference* to an extra connection string definition native to the .NET configuration. In our

App.config file, the connection string definition has the name PhoneBook, as referenced in the provider definition, what transports us to the next node, aptly named connectionStrings.

4.) Apart from the usual details on how to make the database trust you, the connection string also contains the name of the database where domain object instances are actually stored. In our case that's PhoneBook, of course.

5.) This is the name of the database server. As the listing above reveals, the author runs a complete MS SQL server locally, and assumes that you do so, too. You might or might not enjoy the same luxury, so adapt this value according to your development environment.

However, don't forget to provide the correct name (or IP address) for your database server if it is different from localhost. The cheapest way to get App.config into your project:

- Add a new text file named App.config to your PhoneBook.Sample project

Copy/Paste the content of the provided App.config in

http://re-motion.org/content/PreparedFiles/PhoneBook

- into that new text file
- Modify the connection string (annotation 5 in the listing above) to reflect your credentials and database server name.

You might remember that we asked you to manually fix the name of the database in the SetupDB.sql file after it had been generated (section *Putting it all together*). This was a hak that stemmed from the fact that we had not introduced App.config at that point. If you provide dbschema.exe with a path to App.config, it uses this config-file to configure re-store, thus making the database name available to dbschema.exe. You can try this yourself after adding an App.config file to your project, if you want to, although the step is redundant, because your SetupDB.sql file should already work. You can try this yourself by invoking dbschema.exe again. In your \PhoneBook.Sample-directory, incant this:

```
C:\PhoneBook\Remotion\dbschema.exe /schema /config:Domain.Sample\App.config
/baseDir:PhoneBook.Domain\bin\Debug
```

If you open the generated SetupDB.sql now, you will see that no manual modification of that "USE DatabaseName" in the first line is required. dbschema.exe found "PhoneBook" in the App.config file and correctly generated "USE PhoneBook".

At this time, your application should be ready to run. You can simply call the EnterFreud function in your Main(). Here is a complete listing of Program.cs:

```
using System;
using System.Collections.Generic;
using System.Text;
using Remotion.Data.DomainObjects;
using PhoneBook.Domain;

namespace PhoneBook.Sample
{
  class Program
```

```
  {
    static void EnterFreud()
    {
      using
(ClientTransaction.CreateRootTransaction().EnterDiscardingScope())
      {
        Location loc = Location.NewObject();
        loc.Street = "Berggasse";
        loc.No = 19;
        loc.City = "Vienna";
        loc.ZipCode = "1090";
        loc.Country = Country.Austria;

        Person person = Person.NewObject();
        person.FirstName = "Sigmund";
        person.Surname = "Freud";
        person.Loation = loc;

        PhoneNumber phone = PhoneNumber.NewObject();
        phone.CountryCode = "43";
        phone.AreaCode = "1";
        phone.Number = "3191596";

        person.PhoneNumbers.Add(phone);

        ClientTransaction.Current.Commit();
      }
    }

    static void Main(string[] args)
    {
      EnterFreud();
    }
  }
}
```

(Don't forget to set `PhoneBook.Sample` as you startup project in Visual Studio.)

With every run of your program, new (and virtually identical) Freud-records in all tables should be generated. You can convince yourself that this is actually happening with

- `SELECT * FROM Location;`
- `SELECT * FROM Person;`
- `SELECT * FROM PhoneNumber;`

What if it doesn't? The next section - "Troubleshooting your application" - will discuss what can go wrong. If your Sigmund Freud-app works as advertised, you might want to skip the troubleshooting section. However, remember that it is there if you get stuck in future experiments.(As an aside: "Berggasse 19" really was Dr Freud's address in Vienna. This is where his famous couch was located, and where "Anna O." and "The Wolf Man" went in and out on a daily basis, making Dr Freud rich and famous in the process. Today the couch is still there, but it is serving as a prop in the Sigmund Freud Museum. The phone-number is that of the museum. Dr Freud's phone-numbers in Vienna never exceeded 5 digits. He left Vienna long before 6-digit phonenumbers were introduced in Vienna in 1960.)

What can go wrong

Forgetting to compile

Failing to rebuild the application makes problems, even if you modify `App.config`. Although `App.config` is not compiled during a build, it is copied to your application's `bin\Debug` or `bin\Release` directory, so that your application can find it at run-time. App.config is also renamed to resemble your application's executable's name. For example, for `PhoneBook.Sample`, `App.config` becomes `PhoneBook.Sample.exe.config`.

Querying your objects and classes

Since having only Sigmund Freuds in a phone-book is a little boring, let's add more people from Viennese folklore, namely Habsburg royalities Emperor Franz-Josef and Empress Sisi (she is the one who got murdered by a looney with a double-cut file). Both had their summer residence in the *Schönbrunn* palace, today a popular tourist attraction with a gorgeous, huge park and extremely expensive popsicles and sodas. The phone-numbers are ficticious, just like most of the Sisi-biopic starring Romy Schneider. The next listing is the code for a static member function for `Program` named `EnterHabsburgs`. It uses three new factory methods for instance creation and initialization.

```
static void EnterHabsburgs()
{
  using (ClientTransaction.CreateRootTransaction()
                          .EnterDiscardingScope())
  {
    Location loc = Location.NewObject();
    loc.Street = "Schönbrunner Schloßstraße";
    loc.Number = "1";
    loc.City = "Vienna";
    loc.ZipCode = 1130;
    loc.Country = Country.Austria;

    Person theEmperor = Person.NewObject();
    theEmperor.FirstName = "Franz-Josef";
    theEmperor.Surname = "Habsburg";
    theEmperor.Location = loc;

    PhoneNumber phone = PhoneNumber.NewObject();
    phone.CountryCode = "43";
    phone.AreaCode = "1";
    phone.Number = "555-0001";

    theEmperor.PhoneNumbers.Add(phone);

    Person theEmpress = Person.NewObject();
    theEmpress.FirstName = "Sisi";
    theEmpress.Surname = "Eugenie";
    theEmpress.Location = loc;

    phone = PhoneNumber.NewObject();
    phone.CountryCode = "43";
    phone.AreaCode = "1";
    phone.Number = "555-0002";

    theEmpress.PhoneNumbers.Add(phone);
```

```
        // the empress has two phones -- one for
        // the left ear, one for the right
        phone = PhoneNumber.NewObject();
        phone.CountryCode = "43";
        phone.AreaCode = "676";
        phone.Number = "555-0003";

        theEmpress.PhoneNumbers.Add(phone);

        ClientTransaction.Current.Commit();
      }
    }
```

(If you want to get rid of all those Dr Freuds from the previous steps, simply run the `SetupDB.sql` query file again in your database.)

Modify your `Main()` to call `EnterHabsburg()` instead of `EnterFreud()` and run your application again. If you check the results with `SELECTs`, you should see one `Location` record, two `Person` records and three `PhoneNumber` records (assuming that you purged the Dr Freuds first).

However, the interesting part is how these objects are related. Finding out which phone-numbers Sisi has is easy, just look it up in `theEmpress' PhoneNumbers` property. You could list both of her phone-numbers right away with this snippet:

```
foreach(PhoneNumber p in theEmpress.PhoneNumbers)
{
      System.Console.WriteLine("+{0} {1} {2} {3}",
                               p.CountryCode,
                               p.AreaCode,
                               p.Number,
                               p.Extension);
}
```

You can do it that easily, because re-motion manages the association of the (bidirectional) relation and its mapping between the RDBMS-realm and the .NET-realm automatically. Remember that we basically told re-store that we wish to have the bidirectional relation managed for the database and .NET automatically.

Getting Sisi's location is just as easy, because all you have to do is look into the `Location` property with

```
theEmpress.Location
```

No attribution exists, but re-motion is smart enough to use the mere .NET declaration in the `Person` class as a clue to establish the necessary foreign key column in the `Person` table from this unidirectional (type-only, no attributes) declaration, which is a concise

```
Location Location { get; set; }
```

in the `Person` class.

Contrast the cases with the implicit and explicit relations with getting all `Persons` associated with a `Location`. `Location` objects have no property (object list or otherwise) containing a handy collection of its inhabitants. This information is stored in the database, and a unidirectional relation from `Persons` to their `Locations` exist, but the declaration of the .NET `Location` class gives you no property for going from `Location` to `Person`. However, in the database it is easy to query for a location's persons, of course:

```
SELECT * FROM Person WHERE LocationID = <some Location's GUID>;
```

So in order to have a handy method for asking a `Location` for all of its `Persons`, we simply wrap up that query in a `Location` member function. re-motion provides a mechanism for doing this in a consistent, safe way.

In more technical terms:

- bidirectional relationships offer many advantages for programmers, not the least being that objects referenced point back to their parent object
- a unidirectional relationship is much less entangled. The implicit (i.e. "tacit" or hidden) side has its reality only in the database as a foreign key in a table, not in the .NET-world
- with a query we can discover the implicit, hidden side of the unidirectional relationship

So in order to have a method `GetAllPersons` in a `Location` that works as nicely as the `PhoneNumbers` property does in `Person`, we must connect the function to an SQL query and the result of the query back to the function. Until recently, the only way to do this was re-store's *query manager* (discussed in Appendix A: Using the query manager). However, mapping queries to .NET has become a lot easier with the introduction of Microsoft's LINQ (language-integrated query). This shiny new LINQ thingy makes *some sort of* SQL a *part* of .NET. The way LINQ works can be seen in the listing of the *complete* `GetAllPersons` method in `Location`:

```
public Person[] FindPersons()
{
  var query = from p in QueryFactory.CreateLinqQuery<Person>()
              where p.Location == this
              select p;
  return query.ToArray();

}
```

That's it. No mapping, no XML-superglue, less typos. All you have to learn is how to express your SQL-desires in C#'s LINQish yntax. Your object-relational mapping-blues is OVER!

What the author likes about LINQ is the fact that an introduction is beyond the scope of a PhoneBook-primer, so we can pass the instructional buck to Microsoft, or Wikipedia. Not the worst feature of LINQ is its similarity to SQL's structure and logic, so the above and the following example will probably pose no difficulty, even to readers who have never heard of LINQ but know SQL.

Here is a static method for your `Location` class, giving you all `Location` instances in the database:

```
public static Location[] AllLocations()
{
  var query = from l in QueryFactory.CreateLinqQuery<Location>()
              select l;
  return query.ToArray();

}
```

re-LINQ provides a "query factory" for the given database. It basically serves as your gateway into re-store data.

The old-school way of mapping with the query manager was less convenient and transparent. If nothing else, LINQ made the documentor's (i.e. the author's) query-manager blues go away. If you look into Appendix A, you will find detailed instruction and discussion on how to use the query manager. The query manager is still relevant for complex queries that can't be mapped from LINQ to re-store, but those cases are rare. There will be more documentation on this topic, but for now you must content yourself with the appendix A on the query manager, and appendix B with examples of how re-LINQ queries are mapped to SQL queries.

For a documentor it is much more pleasant to explain how LINQ is integrated into re-store than how the query manager processes its XML mapping files.

re-store includes a so-called *LINQ provider* for re-store objects and interfaces. LINQ queries not only work for databases, but for anything that can be dressed up with an `IEnumerable`, or, even better, `IQueryable`, interface and can be understood as tables and relations – an array, for example, or an array of arrays, or an XML-store. A MUMPS database or http://amazon.com – *whatever*. (To learn how to give amazon.com an `IQueryable` interface:
http://LINQinaction.net/blogs/main/archive/2006/06/28/LINQ_to_Amazon_implementation_fore_steps.aspx

LINQ provides a fairly strange API to its universal mapping machine: it gives you the *parse tree* of a LINQ-expression and let's your LINQ provider do a transformation of that tree into the desired format. For relational database back-ends this is usually SQL-statements; for LINQ-ing
XML it will be be xquery-snippets. You can imagine the LINQ provider as a robot beetle crawling around the parse tree and knitting a scarf from what it sees in the tree. In our case, that's an SQL-scarf that is tailor-made for re-store and the given table inheritance.

Creating re-LINQ (re-store's LINQ-supplement) was hard work, but for you as a user of it this means that you can focus on the .NET existence of your objects and forget how objects are actually spread across tables. re-LINQ maps what you mean for the .NET-world to what have to say to the database. As for the given table inheritance, re-LINQ can find out everything about the class structure and relations in your domain by communicating with re-store (the component doing is called *LINQ2OPF*).

Slightly more information about re-LINQ can be found in appendix B, a grid of how typical LINQ queries are transformed.

LINQ is an interesting subject, way more cool to explain than the query manager. And there is tons of stuff on google (also supporting the `IQueryable` interface: http://langexplr.blogspot.com/2007/05/LINQ-to-google-desktop.html).

The following updated version of the `Main()` function lists all locations, persons and phone-numbers:

```
static void Main(string[] args)
{
  // Freud and Habsburgs removed
  // because they are in the database already
  // EnterFreud();
  // EnterHabsburgs();
  using (ClientTransaction.CreateRootTransaction()
                          .EnterDiscardingScope())
  {
    foreach (Location loc in Location.AllLocations())
    {
      Console.WriteLine(loc.Street);
      foreach (Person p in loc.FindPersons())
      {
        Console.WriteLine("    {0} {1}", p.FirstName, p.Surname);
        foreach (PhoneNumber phone in p.PhoneNumbers)
        {
          Console.WriteLine("   +{0} ({1}) {2}/{3}",
            phone.CountryCode,
            phone.AreaCode,
            phone.Number,
            phone.Extension);
        }
      }
    }
    Console.ReadLine();
  }
}
```

re-store under the hood, best practices

On relationships

The [Mandatory] attribute

You use the [Mandatory] attribute to signal that the attributed reference property must not be null. Here is an example of how to make a Person's Location property mandatory:

```
public class Person : DomainObject
{
  // blah

  [Mandatory] // That's the word
  public virtual Location Location { get; set; }

  // blah
}
```

1:1 bidirectional relationships

The bidirectional relationship between a Person and its PhoneNumbers clearly is an 1:m relationship, as indicated by the PhoneNumbers property's type -- ObjectList<PhoneNumber>. Bidirectional 1:1 relationships are declared more or less in the same way, with one important difference. You must specify which of the two ends , or tables, holds the foreign key. Here is a 1:1 relationship between a company and its president. Note the ContainsKey parameter in the Company's [DBBidirectionalRelation] attribute:

```
public class Company : DomainObject
{
  // The 'Company' table will hold the foreign key
  [DBBidirectionalRelation ("PresidentOf", ContainsKey = true)]
  [Mandatory]
  public virtual Person President { get; set; }
}

public class Person : DomainObject
{
  [DBBidirectionalRelation ("President")]
  [Mandatory]
  public virtual Company PresidentOf { get; set; }
}
```

m:n relationships

Let's say you want to build a movie database in re-motion, modeled after the IMDB (http://imdb.com) . Multiple actors appear in a movie; one actor appears in multiple movies -- this clearly is an m:n relationship. RDBMS-programmers use junction tables for such a task, and in re-

motion you simply use junction objects. In this example, you would have a domain object named Engagement with two properties:

```
[DBBidirectionalRelation("Engagements")]
public virtual Actor Actor { get; set; }
[DBBidirectionalRelation("Engagements")]
public virtual Movie Movie { get; set; }
```

If you are not sure how this is supposed to work, read the extra document *Junction domain objects* (http://re-motion.org/content/junction-domain-objects.pdf). It contains a complete sample with comments and deeper discussion.

Deleting objects

So we have created new objects, but what about deleting old objects? If you want to delete an object, just call... `Delete`:

```
myDomainObject.Delete ();
```

Except that this would not work out of the box, because `Delete` is a `protected` method. If you want to call a domain object's `Delete` method from outside that object, you must redeclare `Delete` as `public` first (just like `NewObject` and `GetObject`. Such a `new` method will work for any domain object class declaration:

```
public new void Delete ()
    {
      base.Delete ();
    }
```

What if you want to delete an object that is enlisted in a bidirectional relationship? Again, bidirectional relationships mean convenience here: re-store does the right thing and updates the remaining end(s) of the bidirectional relationship. For example, if you were to delete a `Person` object, all its {{PhoneNumber}}s get their `Person` property set to `null`. If you delete a `PhoneNumber` object, the reference to that object is removed from the owner's `ObjectList<PhoneNumber>`. Deleting objects referenced by a `[Mandatory]` relationship require some care. re-store will let you remove a domain object that is attributed as `[Mandatory]` in the owned domain objects. If the `Person` property in a `PhoneNumber` were `[Mandatory]`, you must insert another object in its place before `Commit`ing the whole operation.

If you want to delete a domain object that has a *unidirectional* relation with other domain objects, re-store will NOT let you do this. For example, if you want to delete a `Location` object, you must first sever any links from `Person` objects to that `Location` object. You do this by setting the `Location` property in those `Person` objects to `null`. Here is a method `DeleteMakeHomeless` for your `Location` domain object class. It finds all persons living at the given location, sets their `Location` property to `null` and finally deletes the given object:

```
public void DeleteMakeHomeless ()
    {
      var persons = FindPersons ();
      foreach (var p in persons)
      {
        p.Location = null;
```

```
      }
      Delete ();
      ClientTransaction.Current.Commit ();
    }
```

The method uses the `FindPersons` method from section *Querying your objects and classes*. What's more, it assumes that a transaction is provided by the caller. You can try this method for yourself by adding an appropriate facility to the `PhoneBook.Sample` application. Here is a cheap implementation:

```
static void Report (Location[] locations)
{
  int iter = 0;
  foreach (Location loc in locations)
  {
    Console.WriteLine ("{0} {1}", iter, loc.Street);
    iter++;
  }
}

static void ReportWrap ()
{
  using (ClientTransaction.CreateRootTransaction ().EnterDiscardingScope ())
  {
    Report (Location.GetLocations ());
  }
}

static void RunInputMask ()
{
  using (ClientTransaction.CreateRootTransaction ().EnterDiscardingScope ())
  {
    bool goOn = true;
    while (goOn)
    {
      var locations = Location.GetLocations ();
      Report (locations);

      var cmdLine = Console.ReadLine ();
      cmdLine.Trim ();
      if (!String.IsNullOrEmpty (cmdLine))
      {
        switch (cmdLine[0])
        {
          case 'd':
          case 'D':
          try
          {
            var item = int.Parse (cmdLine.Substring (1));
            locations[item].DeleteMakeHomeless ();
          }
          catch (FormatException)
          {
            Console.WriteLine ("Can't parse integer: {0}", cmdLine);
          }

          break;
          case 'q':
          case 'Q':
             goOn = false;
             break;
        }
      }
    }
  }
}
```

On lazy loading

As stated before, re-store uses "lazy loading" for fetching domain objects from the database into their .NET-existence.

The *lazy load* is a well-established design pattern, tried and trusted since the dawn of computing. Martin Fowler has this to say about the lazy load: "An object that doesn't contain all of the data you need but knows how to get it." In essence, lazy loading means just-in-time delivery of persisted bits. In the case of re-motion, being stingy with the domain objects' memory-footprint is not a RAM issue. Loading domain objects is expensive, because

- loaded objects must be schlepped around as state information for ASP.NET (either state server or SQL-server)
- loading an object is not for free in terms of database utilization
- loading an object is expensive, because the framework must build a domain object (and generate code for the type if it hasn't been cached already)

re-store roughly follows this simple strategy: not before domain objects "are needed" they are actually loaded from database records into chips. Objects get purged from memory as soon as the transaction is over.

The remaining question is: what exactly does "a domain object is needed" mean? If you want to display a domain object in its entirety in a form, then you clearly need this domain object, so it will be loaded into RAM. Domain objects are always loaded in their entirety, which means that even if you need just one property (the first name in a person object, for example), then the entire domain object is loaded.

Other cases are not as clear. What if the domain object in the form has a reference property to another domain object? Or a list of references? Are those objects loaded into memory as well?

The rule is: as soon as you actually access a reference property in a domain object, the referenced object(s) get loaded into memory.

If you want to display a domain object in its entirety in a form (a `Person` object, for example), then it is inevitable that you will access every property in the object and cause the loading of each and every object referenced in any of its properties.

This is in contrast to accessing some scalar property in a `Person – FirstName`, for example. Only the `Person` instance itself is needed in RAM (or actually just that `FirstName` property), so no other domain objects needs to be loaded, even if the `Person` object references a `Location` and `PhoneNumbers`. For as long as you don't access the `Person`'s `PhoneNumbers` reference property, the referenced `PhoneNumbers` object won't be loaded.

Less intuitive, but technically reasonable, is the following behavior of objects in bidirectional relationships. To avoid confusion, we use the following convention here:

- we call the side of the bidirectional 1:n relation with the single instance "the parent side" (conventionally called the "n-side")
- we call the side of the bidirectional relation with the multitude of instances "the children side" (conventionally called the "1-side")
- we call the *property* in the parent referencing the children the "children property" (this property is always an object list)
- we call the *property* in a child referencing the parent the "parent property" (this property is never an object list)

If you access the parent property in a child object in a bidirectional relation, then all the other child objects in the children property are loaded into memory as well. `Person` and `PhoneNumbers` provide a good example for this: If you access the `Person` property in `PhoneNumbers`, then not only that Person object is loaded, but also all the `PhoneNumber's` siblings in that `Person` object's `PhoneNumbers` property (an object list). In other words: Since a bidirectional property by definition works both ways, just displaying a `Person` object will cause *all* its child `PhoneNumber` objects to be loaded as well.

For these reasons, unidirectional relationships are often more economical, sometimes even the only viable option. Bidirectional 1:n relationships are *not* a good idea if you expect the n to be very large. After all, the n objects will be aggregated in a single object list, and touching that object list will cause all n objects to be loaded. This has an impact on performance.

Unidirectional relationships are safer for large n, because there is no object list at the n-side to begin with. Only the objects at the 1-side reference the object they belong to (like `Person` objects reference a shared `Location` object). In this sense, an unidirectional relation acts like a firewall against excessive loading/copying/serialization of objects in transactions.

For small numbers of expected children of a domain object choosing bidirectional relationships has important advantages for automation. Since bidirectional relationships are smart and know about each other, their code can cooperate and give you typical UI- and transaction behavior for free. Unidirectional relationships are unsmart and the dumber side does not even know which children it is the parent of.

You will see more examples of the virtues of bidirectional relationships in the next sections. Here are a few hints for judging whether the advantages compensate the disadvantages for each type of relationship:

- bidirectional relationhips:
    - use them for objects that can be expected to be navigated together frequently.
    - use them for 1:1 relationship and 1:n where n can be expected to be not more than "a few". What this means depends on your application, i.e. the size of the loaded objects. Usually less than a hundred items pose no problems for

typical objects, but you probably want to make a few experiments before making that decision.
    - this has the advantage that UI-facilities for easy navigation are added automatically.
    - that related objects are mutually notified on changes to their respecitive parent/children.
    - transactions work across modification of objects belonging to each other.
    - disadvantage: computationally expensive in many aspects

- unidirectional relationships:

    - use them for 1:n relationships where n is... too large for bidirectional relationships.
    - disadvantage: less automation for objects that *could* point to each other.

How large an n can be considered small enough for bidirectional relationships is a matter of priorities and also depends on the complexity of the involved objects. In case of doubt, try alternatives and do benchmarks.

A current problem with unidirectional vs. bidirectional relationships is that they have a slightly different behavior in transactions; this can confuse programmers and users alike. Observe that in bidirectional relationships, both ends know each other *as .NET objects*, what enables quick updates of modifications before they are actually committed to the database. For example, if a user moves a person's phone-number to another person, this change will be visible immediately in both persons and both phone-numbers, even before the commit.

For persons and locations the behavior is different, because (in the PhoneBook application) `Person` has an unidirectional relationship to `Location`. If you change a Person's country from Australia to Burkina Faso, this modification is immediately visible, of course. However, the other way around it is not. You can only detect which persons belong to a location by running a query, i.e. not before it is written to the database, i.e. committed. A user observing this might be confused by this inconsistency.

In a word, for unidirectional relationships you can always be sure that what you see is the actual state of the database. In bidirectional relationships a modification is reflected in memory before being committed to the database.

This behavioral difference might vanish in the future. Some ideas exist on how to make updates in transactions involving unidirectional relationships more like those bidirectional relationships.

On client transactions and sub-transactions

If you are reading this primer you probably know what a database transaction is, but in re-store, *client transactions* have a slightly different meaning and semantics. Transactions, as commonly understood, give you some protected time and space to perform operations on data that make little or no sense without each other. re-store's client transactions mimic this

aspect of a database layer for their in-memory operations. For example, if you have just instantiated a new `Person` object, you don't want other users to find and use your `Person` object in the making before all property values (`FirstName`, `Surname`, etc.) are filled in and good to go. So you open a client transaction for instantiating your person object, populate it with data and commit your modifications for other users to see. If you change your mind before completing the object or decide not to commit it, you simply discard the transaction and can be sure that no unfinished object is left over injecting confusion into your system. These virtues of transactions are well-known in the world of manipulating data and can be applied to re-store very well.

The most important difference of re-store's client transactions in regard to how the concept is usually understood is this: client transactions not only provide a protective umbrella for operations and data, they give domain objects their .NET-existence in first place. A domain object cannot be loaded from the persistence store outside the context of a client transaction, because it is the client transaction that provides the *data container* for the object's property values. As soon as a client transaction is over, the involved domain object's .NET-existence is over as well and it is subsequently removed from memory. A client transaction always comes with a *transaction scope*, essentially a block of code that automatically limits the duration and influence of the transaction.

This design has important advantages:

- the risk of "dangling", "forgotten" transactions is minimized
- domain objects are held in memory only for as long as operations on them are in progress
- since a domain objects require a transaction, and a transaction requires a scope, and the extent of a scope is usually clearly visible, it is easy to remain organized and easy to see which objects belong to what transaction *and which transaction is active for a given scope*.
- it is not possible to modify a domain object in a transaction different from the one it was created or loaded in (an attempt to do so will throw you an exception)

So the benefit of re-motion's transaction model lies in its constraints, because they help keeping things tidy and manageable.

A somewhat unusual consequence of keeping a domain object's data in the client transaction's data container, domain objects also need a client transaction for read(-only) operations.

In the light of this discussion, let's look at some transactional code from this chapter again:

```
static void EnterFreud()
{
  using (ClientTransaction.CreateRootTransaction()
                          .EnterDiscardingScope())
  {
    Location loc = Location.NewObject();
    loc.Street = "Berggasse";
    loc.Number = "19";
    loc.City = "Vienna";
```

```
        loc.ZipCode = 1090;
        loc.Country = Country.Austria;

        Person person = Person.NewObject();
        person.FirstName = "Sigmund";
        person.Surname = "Freud";
        person.Location = loc;

        PhoneNumber phone = PhoneNumber.NewObject();
        phone.CountryCode = "43";
        phone.AreaCode = "1";
        phone.Number = "3191596";

        person.PhoneNumbers.Add(phone);

        ClientTransaction.Current.Commit();
      }

    }
```

`ClientTransaction.CreateRootTransaction()` creates a new transaction, and `.EnterDiscardingScope()` makes the transaction scope active for the scope of the `using` directive. In this fashion it is easy to see where the transaction scope beings and ends. All the object manipulations between the using's braces belong to that scope and that transaction.

The method `EnterDiscardingScope()` simply means that by the end of the scope (i.e. when the using's closing brace is reached), the transaction is discarded, i.e. commanded to the grace of the garbage collector, and the domain objects in the scope go with it. If no `Commit()` is performed on the transaction, it is as if the transaction and its operations had never happened.

Variations to `NewClientTransaction()` and `EnterDiscardingScope()` come into play as soon as we start talking about nested transactions (see next example), but let's look at nested *scopes* here. For a somewhat contrived example, assume that our phone-book application is actually a phone-company application and Dr Freud is a customer who wants an easy to remember phone-number. To that end, you must search for phone-numbers like "888-6666" among the available `PhoneNumber` domain objects. This in turn, requires a separate transaction scouring all available phone-numbers:

```
        PhoneBook rememberEasy = null;
        using (ClientTransaction.CreateRootTransaction()
                                .EnterDiscardingScope())
        {
          PhoneNumber[] phoneNumbers = PhoneNumber
                  .GetAllPhoneNumbers(ViennesePhoneNumbers)
                  .ToArray();
          foreach(int i in phoneNumbers)
          {
             if (EasyToRemember(pn))
             {
               rememberEasy = pn;
               break;
             }
          }
        }
```

Note that this function brings its own transaction and transaction scope. Also note that the code above won't do much good for the variable `rememberEasy`, because the `PhoneNumber` object stored there will lose its data as soon as code execution leaves the scope. Consequently, the following naive implementation can*not* work:

```
static void EnterFreud()
    {
      using (ClientTransaction.CreateRootTransaction()
                              .EnterDiscardingScope())
      {
        Location loc = Location.NewObject();
        loc.Street = "Berggasse";
        loc.Number = 19;
        loc.City = "Vienna";
        loc.ZipCode = "1090";
        loc.Country = Country.Austria;

        Person person = Person.NewObject();
        person.FirstName = "Sigmund";
        person.Surname = "Freud";
        person.Location = loc;

        PhoneBook rememberEasy = null;
        using (ClientTransaction.CreateRootTransaction()
                                .EnterDiscardingScope())
        {
          PhoneNumber[] phoneNumbers = PhoneNumber
              .GetAllPhoneNumbers(ViennesePhoneNumbers)
              .ToArray();
          for(int i in phoneNumbers)
          {
             if (EasyToRemember(pn))
             {
               rememberEasy = pn;
               break;
             }
          }
        }

        // WRONG. /rememeberEasy/'s data is gone at this point.
        person.PhoneNumbers.Add(rememberEasy);

        ClientTransaction.Current.Commit();
      }
    }
```

What's more, it is illegal to use an object created within an transaction for another transaction, so `Person.PhoneNumbers.Add(rememberEasy)` will give you an exception. This exception is your friend, because then at least you know that something is wrong.

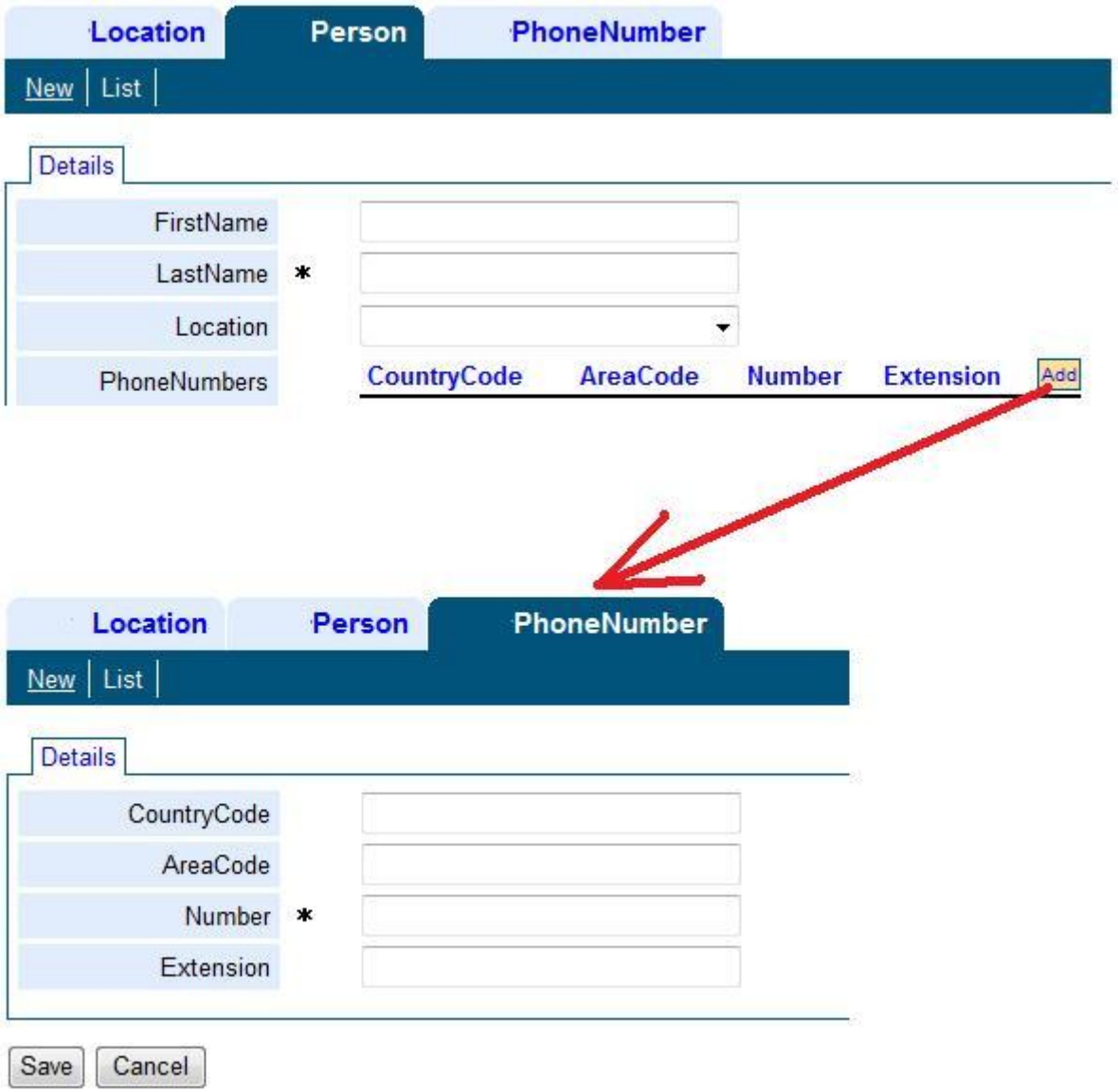
The correct way of implementing our contrived example is to remember the found domain object's ID and use the `RepositoryAccessor` (for example) to re-load that object from the database into the active client transaction:

```
static void EnterFreud()
    {
      using
(ClientTransaction.CreateRootTransaction().EnterDiscardingScope())
```

```
  {
    Location loc = Location.NewObject();
    loc.Street = "Berggasse";
    loc.Number = 19;
    loc.City = "Vienna";
    loc.ZipCode = "1090";
    loc.Country = Country.Austria;

    Person person = Person.NewObject();
    person.FirstName = "Sigmund";
    person.Surname = "Freud";
    person.Location = loc;

    ObjectID rememberEasy = null;
    using (ClientTransaction.CreateRootTransaction()
                            .EnterDiscardingScope())
    {
      PhoneNumber[] phoneNumbers =
        PhoneNumber.GetAllPhoneNumbers(ViennesePhoneNumbers).ToArray();
      for(int i in phoneNumbers)
      {
         if (EasyToRemember(pn))
         {
           rememberEasy = pn.ID;
           break;
         }
      }
    }

    // Correct version. We use the remembered domain
    // object *ID* to re-fetch the
    // found domain object from the database for the active
    // transaction.

   person.PhoneNumbers
         .Add(RepositoryAccessor.GetObject(rememberEasy, false));

    ClientTransaction.Current.Commit();
  }
}
```

In the example, the two transactions are independent from each other. Each of the two transactions is a `RootTransaction`, so commits are written to the database. These transactions are *not* nested, whereas the scope for searching for easy phone-numbers is nested within the scope of creating a new Person and Location object.

Often it is not desired that a related transaction (typically for a referenced object) writes directly to the database before the "parent" transaction does so. An example for this is the creation of an object that has an object that must also be created, for example a new person and its phone-number. Here are screen-shots of how this would look in a PhoneBook web application:



In the illustration, the user is about to create a new phone-book person entry and has opened another window to give the new person a new phone-number. Where is the data supposed to go if the user clicks "Save" in the `PhoneNumber` window? If it goes to the database and the user changes his mind and cancels the parent `Person` window, there will be an orphan `PhoneNumber` in the database. If the user cancels the `PhoneNumber` window, is that supposed to cancel the parent `Person` window as well? Enter sub-transactions.

Sub-transactions

Unlike root-transactions, which read and write from the database, sub-transactions read and write their data from the parent transaction.

For our `Person-PhoneNumber` example this means that

- As soon as the user clicks the "Add" (phone-number) button in the Person form, a sub-transaction of the root-transaction involving the `Person` object is created. The `PhoneNumber` sub-transaction gets its data from the `Person` parent transaction.

- If the user clicks "Save" in the `PhoneNumber` window, the `PhoneNumber` data is committed to the *parent transaction*, not to the database.
- If the user clicks "Cancel" in the `PhoneNumber` window, the `PhoneNumber` transaction is discarded while the parent transaction is still active.
- As soon as the user commits the new `Person` data, modifications (if any) from the `PhoneNumber` sub-transaction will be committed as well.
- If the user instead cancels the root transaction, modifications (if any) from the `PhoneNumber` sub-transaction will be discarded as well.

Sub-transactions can be nested to an arbitrary depth. This is called a "transaction hierarchy", although the term is somewhat misleading, because it suggests a tree, while it is more of a stack, actually. Each sub-transaction gets its data from the parent transaction. A commit in a sub-transaction commits its data to the parent transaction, not the data base. For as long as the sub-transaction is active, its parent transaction data is read-only. In other words: sub-transactions give you commit/rollback behavior, but not before a commit in the root transaction the data is actually written to the database. Apart from the difference in where they get their data from, sub-transactions are more or less like root-transactions:

- they hold the properties for the involved domain objects in a data container
- they are destroyed as soon as their scope is left (closed)

In the `Person-PhoneNumber` example above we have talked about the edit pages/forms for the `Person` root transaction and the `PhoneNumber` sub-transaction as if they were the transactions themselves. This is permissible and not a mere illustration-gimmick. re-motion's web execution engine re-call does the right thing for you and creates sub-transactions for "sub-forms" automatically when the need arises. This scenario of creating or opening (loading) an object referenced by a currently transacted object provides another frame of thinking of transactions here: re-store transactions provide data integrity for objects involved in a "user session", a "unit of work" that involves editing a group related objects in related (= appropriately nested) sessions.

You can see how this works for yourself in a set of simple experiments. In the following examples we will overwrite Sigmund Freud's first name, but play with Commit() in root transactions and sub-transactions. Here is a simple method for the top-level Program class:

Rollback vs. discard

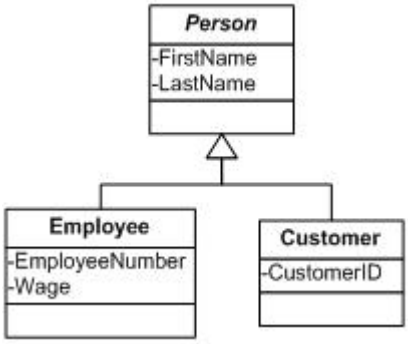
So far, we have used "discard" as the alternative to "commit" – if your program does not like modifications, it simply skips the commit and the changes to domain object data are not permanently stored in the persistence store. However, this assumes that your program wants to give up the transaction together with the data, what is not always so. You can keep the transaction and loose the data with the `Rollback` method.

Optimistic locking

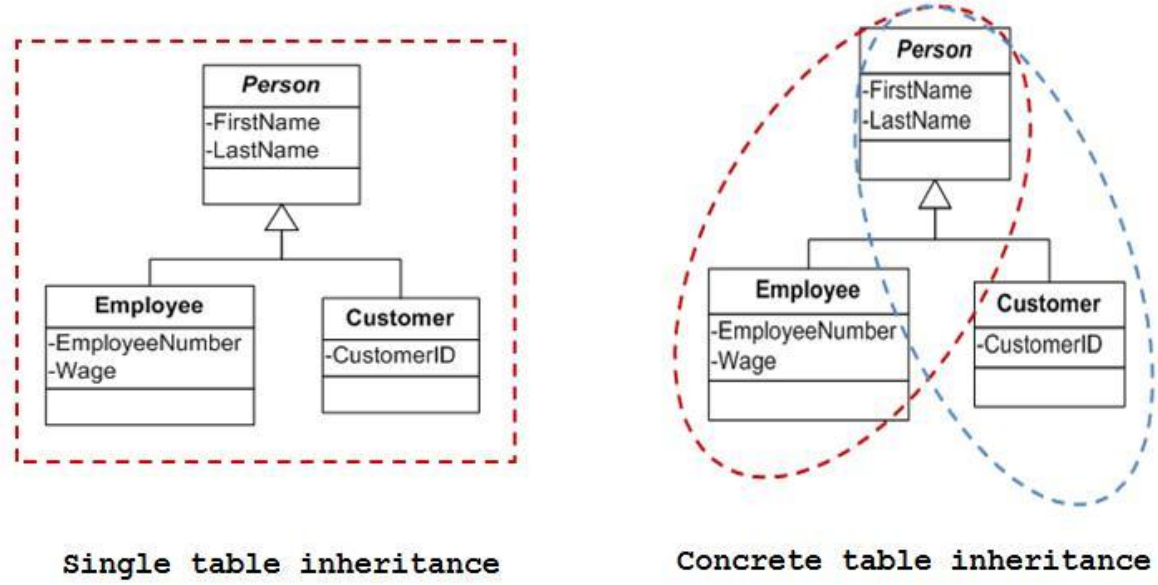
re-store uses *optimistic locking* for locking loaded domain objects. This means that a user CAN use an object that has been loaded already, but he won't be able to write it back if that previously loaded object has been modified AFTER he has loaded it. This mechanism might look odd, or even unsafe at first glance. In practice, however, such collisions rarely happen. The alternative, *pessimistic locking*, has nastier consequences in practice: users lock an object and go to lunch, for example. Or on vacation. Other users are stuck with locked objects.

Table inheritance, attributes and views

The backgrounder to this section is named *Persisting objects in tables* (`persisting-objects-in-tables.pdf` in the *fascicles*). It explains how single table inheritance, concrete table inheritance and class table inheritance can be used when persisting objects in database tables. In this section we will see how to do attribute domain object classes in order to arrive at single table inheritance or concrete table inheritance (class table inheritance is not yet supported by re-store). The key to organizing tables is the attribute `DBTable`. Since creating a meaningful class hierarchy from our three domain object classes `Location`, `Person` and `PhoneNumber` is difficult, we will use the simple example of the `Person-Employee-Customer` class hierarchy from section **Fehler! Verweisquelle konnte nicht gefunden werden.**:



As explained in that section, single table inheritance puts the entire class hierarchy into a single table; concrete table inheritance slices the tables along the specialization of each sub-class:



Single table inheritance **Concrete table inheritance**

In order to find out how properties shall be distributed over tables, re-store (and `dbschema.exe` follow this simple recipe:

- start at the leaves of the inheritance tree (i.e. all classes with no sub-class)
- check if that sub-class has a `DBTable` attribute
- if so, give it its own table
- if not, walk the hierarchy upwards until you find a base-class sporting `DBTable` and put all the classes you found along the way into a single table. However, just one `DBTable` per line of ancestry is allowed.

The following two examples will guide your understanding of the recipe.

Single table inheritance

The base-class `Person` for `Customer` and `Employer` gets the `DBTable` attribute. This means that the properties for all three classes go into the same table. Here is the listing for the three classes:

```
[DBTable]
public class Person : DomainObject
{
  [StringProperty(MaximumLength = 40, IsNullable = false)]
  public virtual string FirstName { get; set; }
  [StringProperty(MaximumLength = 40, IsNullable = false)]
  public virtual Surname { get; set; }
}

public class Employee : Person
{
  [StringProperty(MaximumLength = 3, IsNullable = false)]
  public virtual string EmployeeNumber { get; set; }
  [StringProperty(MaximumLength = 5, IsNullable = false)]
  public virtual string Wage { get; set; }
}

public class Customer : Person
{
  [StringProperty(MaximumLength = 16, IsNullable = false)]
  public virtual string CustomerId { get; set; }
}
```

`dbschema.exe` will give you the following result for your declarations (in the "Create all tables" section of `SetupDB.sql`):

```
-- Create all tables
CREATE TABLE [dbo].[Person]
(
  [ID] uniqueidentifier NOT NULL,
  [ClassID] varchar (100) NOT NULL,
  [Timestamp] rowversion NOT NULL,

  -- Person columns
  [FirstName] nvarchar (40) NOT NULL,
  [Surname] nvarchar (40) NOT NULL,

  -- Customer columns
  [CustomerID] nvarchar (40) NULL,

  -- Employee columns
```

```
  [EmployeeNumber] nvarchar (40) NULL,
  [Wage] nvarchar (40) NULL,

  CONSTRAINT [PK_Person] PRIMARY KEY CLUSTERED ([ID])
)
GO
```

Note that there is just a single table for all three classes' properties – lo and behold, single table inheritance!

Concrete table inheritance

For concrete table inheritance, all you have to do is attribute each sub-class with `DBTable`, as in the following listing.

```
public class Person : DomainObject
  {
    [StringProperty(MaximumLength = 40, IsNullable = false)]
    public virtual string FirstName { get; set; }
    [StringProperty(MaximumLength = 40, IsNullable = false)]
    public virtual string Surname { get; set; }
  }

  [DBTable]
  public class Employee : Person
  {
    [StringProperty(MaximumLength = 3, IsNullable = false)]
    public virtual string EmployeeNumber { get; set; }
    [StringProperty(MaximumLength = 5, IsNullable = false)]
    public virtual string Wage { get; set; }
  }

  [DBTable]
  public abstract class Customer : Person
  {
    [StringProperty(MaximumLength = 16, IsNullable = false)]
    public virtual string CustomerIdee { get; set; }

  }
```

`dbschema.exe` will delight you with a more complex database schema, consisting of TWO tables:

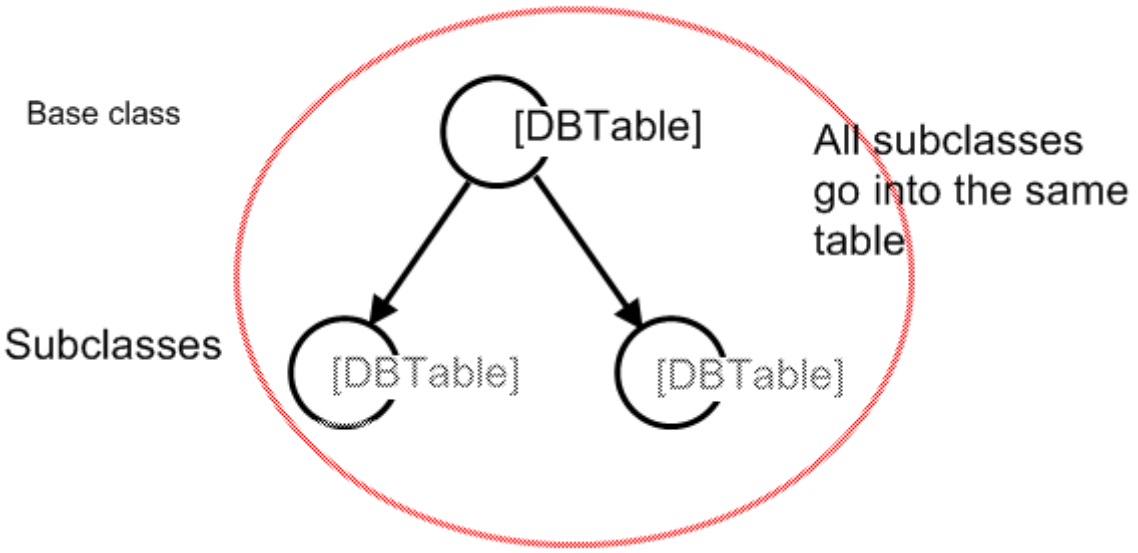
```
-- Create all tables
CREATE TABLE [dbo].[Customer]
(
  [ID] uniqueidentifier NOT NULL,
  [ClassID] varchar (100) NOT NULL,
  [Timestamp] rowversion NOT NULL,

  -- Person columns
  [FirstName] nvarchar (40) NOT NULL,
  [Surname] nvarchar (40) NOT NULL,

  -- Customer columns
  [CustomerIdee] nvarchar (40) NOT NULL,

  CONSTRAINT [PK_Customer] PRIMARY KEY CLUSTERED ([ID])
```

```
)

CREATE TABLE [dbo].[Employee]
(
  [ID] uniqueidentifier NOT NULL,
  [ClassID] varchar (100) NOT NULL,
  [Timestamp] rowversion NOT NULL,

  -- Person columns
  [FirstName] nvarchar (40) NOT NULL,
  [Surname] nvarchar (40) NOT NULL,

  -- Employee columns
  [EmployeeNumber] nvarchar (40) NOT NULL,
  [Wage] nvarchar (40) NOT NULL,

  CONSTRAINT [PK_Employee] PRIMARY KEY CLUSTERED ([ID])
)
GO
```

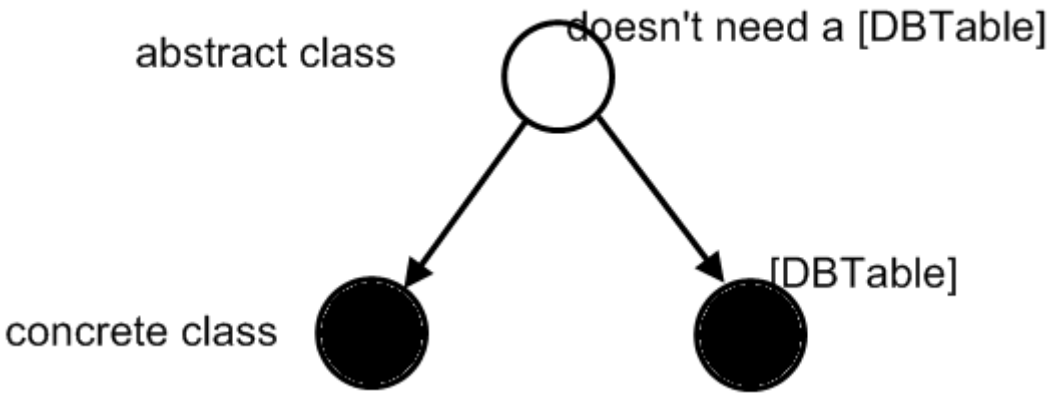
Note that the two tables have each their own set of `FirstName`–`Surname` properties, not by accident the properties that both of them inherit from the `Person` base-class. It's concrete table inheritance!

The guiding principles behind [DBTable] attributes

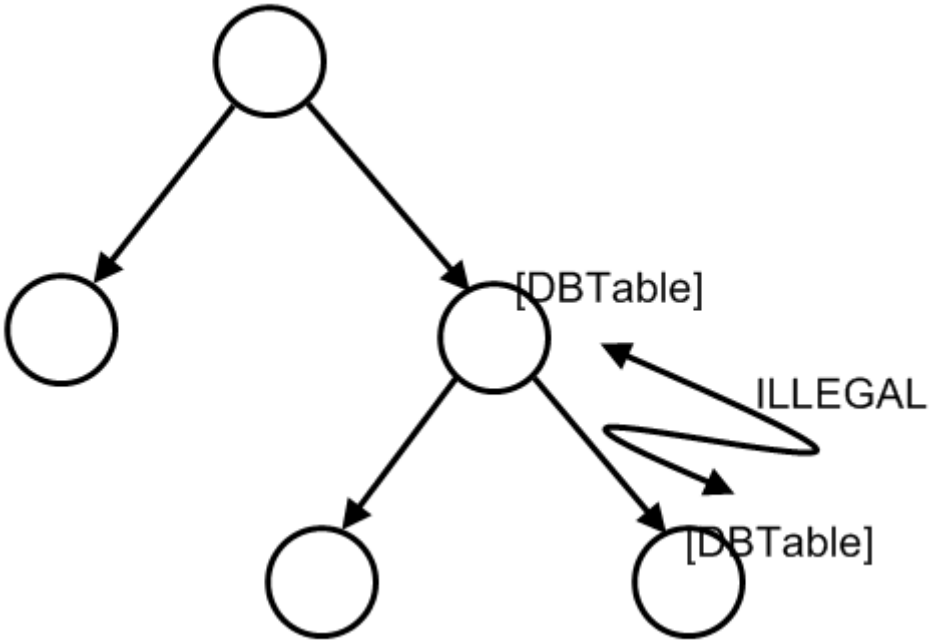
1. If you attribute a domain object class with [DBTable], all those subclasses inherit the attribute, and the attributed class and its subclasses go into the same table:



2. Abstract classes don't need a [DBTable] attribute (neither directly or inherited), but concrete classes do. In the next illustration, the black dots stand for concrete classes, the white dot is an abstract class. The left black concrete class does not have a [DBTable], what is a mistake.



3. Each class must have one and only one [DBTable]. It is an error to give a class a [DBTable] attribute if it already inherits one from its base class. The following image shows such a mistake:



In other words:

1.) From a [DBTable]ed class down, all subclasses go into the same table

2.) Only concrete classes need a DBTable for a table. Abstract classes don't. (Exercise for the reader: WHY?)

3.) Don't attribute subclasses if the base class already has a [DBTable] attribute

Do we use concrete table inheritance for the PhoneBook domain?

Technically yes, but it is a degenerated form of concrete table inheritance. Each of the three PhoneBook domain object classes has its own table, but those classes don't share any properties. They DO have a common base-class, i.e. `DomainObject`, but both re-store and

`dbschema.exe` don't really see it that way, because for them `DomainObject` (and by extension, `DomainObject`) is the rock-bottom of inheritance, just what `System.Object` is for .NET.

As we will see in the next section, a class named `BindableDomainObject`, which just extends `DomainObject` with the `IBusinessObject` interface, is *not* seen as rock-bottom, bare-bones re-store object, but as a full-fledged abstract base-class, just as a common `PhoneNumberObject` would be.

Mind the name clashes!

Alert readers will have noticed that both the `Location` class and the `PhoneNumber` class have a `Number` property. The natural way of naming the corresponding columns in the database is to give a column for a given property exactly the same name as that column. For example, the `FirstName` property in `Person` will correspond to a `FirstName` column. This works perfectly for both `Number` properties, because (in our implementation) each `Number` column will be in a separate table. This changes if we put `PhoneNumber` and `Location` instances into the same table, for example if we opt for single table inheritance, with all three domain object classes (derived from a common base-class) in the same table. Then it is time to use the `DBColumn` attribute. With `DBColumn` you can rename the `Location` class' `Number` column to `LocationNumber`, for example:

```
// Location's table will have a "LocationNumber" instead a "Number" column
    [DBColumn ("LocationNumber")]
    [StringProperty(MaximumLength=12)]
    public virtual string Number { get; set; }
```

As explained in the trouble-shooting section of *What can go wrong* of the section *From declaration to schema*, dbschema.exe will give you this error message if it detects a name clash:

```
Execution aborted: Property 'PhoneBook.Domain.PhoneNumber.Number' of
class 'PhoneNumber' must not define storage specific name 'Number',
because class 'Location' in same inheritance hierarchy already
defines property 'PhoneBook.Domain.Location.Number' with the same
storage specific name.
```

On views

If you look at `SetupDB.sql` and make a few experiments, you will discover that each class gets its own view, independent from the actual structure of the database. `dbschema.exe` introduces those views into the schema as a form of abstraction – even if the distribution of objects over tables changes, the views will not be affected by the modification. This is useful for making (read-only) queries more convenient and future-proof. For a large part of typical queries, views give the programmer

- a simple one-view-per-class world, even if the actual data model is not so simple
- a future-proof haven for most of his queries, fostering experiments with table inheritance

This view-technology is a simple abstraction method that might look old-fashioned when compared to the abstraction facilities provided by LINQ. However, views have the big advantage of providing this abstraction in the database itself, at a fairly low (and tried and trusted) level.

NewObject and GetObject

`NewObject` is `protected`

So far, we have only programmed public static factory methods for *empty* domain objects, i.e. factory methods. Here is a complete list:

- `static public Location.NewObject();`
- `static public Person.NewObject();`
- `static public PhoneNumber.NewObject();`

Note that these factory methods show very little finesse, which is adequate for a toy application:

- the factory methods are public to the entire application
- the factory methods have no parameters

Both characteristics might be inadequate for larger or some types of applications. Creating objects can be a complicated affair in complex applications. Often objects must be created in a certain order, or not all objects or all parts of the applications are allowed to deliberately instantiate all other objects. Well thought-out instantiation policies can help to keep code neat and maintainable, so it is not necessarily a good idea to make all instantiation methods `public`. For this reason, re-store's `NewObject<>` method is `protected`, i.e. you cannot call it from application code. This means you *must* declare a *public* factory methods for enabling your application code to instantiate your domain objects. So creating a `Freud Person` object by using re-stores `DomainObject.NewObject<Person>()` will NOT work:

```
Person freud = DomainObject.NewObject<Person>().With();
```

will give you an "... is inaccessible due to its protection level"-error.

This protection facilitates some restrictions on which objects may create other objects, what in turn can prevent programmers (including yourself) from making a mess in the code.

Introducing `GetObject<T>` (also protected)

Similar reasoning applies to the protected `GetObject<T>` method, which fetches an object from the database. To fetch a `Person` object with a given `ObjectID`, you chant:

```
DomainObject.GetObject<Person>(objId, false)
```

(The extra `Boolean` parameter specifies whether you want to fetch the object even when it is marked for deletion or not.)

As `NewObject<>`, `GetObject<>` is protected and can't be invoked just anywhere. The safest idea is to provide a static `GetObject` method for those classes that might need it, as in:

```
// in class Person
    public static Person GetObject(ObjectID objId)
    {
      return DomainObject.GetObject<Person>(objId, false);
    }
```

An obvious problem is that you might not know of what type the desired object behind `objId` is. In this case, you have to resort to the most generic type you can safely assume, for example

```
DomainObject.GetObject<DomainObject>(objId, false)
```

This in turn, begs the question: where do you put this method? For fetching objects without knowing their types it is better to use a different `GetObject` altogether. The class `RepositoryAccessor` also has a static `GetObject` method, although it is not a generic method and public. This `GetObject` always returns a `DomainObject`.

You can invoke it anywhere you want. Example:

```
using Remotion.Data.DomainObjects.Infrastructure;
DomainObject someDomainObject = RepositoryAccessor.GetObject(objId, false);
```

`RepositoryAccessor` also has a public `NewObject` method, but its use is strongly discouraged.

`NewObject<>` – mind the constructors!

Our simple toy factory methods do not need any parameters, but your application's factory methods probably will. Our parameter-less factory methods are fun and easy, because they can simply use the class' default constructor, which does not need to be declared for default object instantiation.

However, as soon as you need a constructor that initializes the instance, you must declare a corresponding constructor, for `DomainObject.NewObject<>` to call. A simple example is a constructor that initializes `FirstName` and `Surname` in a freshly minted `Person` object:

```
public Person(string firstName, string surname)
{
  FirstName = firstName;
  Surname = surname;
}
```

This constructor will *automatically* be called by re-store as soon as someone invokes `NewObject`'s `With`–method with two strings, for example:

```
DomainObject.NewObject<Person>().With("Arnold", "Schwarzenegger");
```

This automatic invocation of an appropriate constructor works by the magic of reflection, i.e. deep inside `With` there is some code that inspects the number and types of the arguments and the desired class and tries to find a matching constructor for that class and types. For this reason, both `NewObject<>` and `With<>` are generic methods: they expect type parameters to pass around so that the inner core of `With<>` knows which constructor to invoke for actually instantiating the object.

Generic methods are not a requirement for passing type information, of course. Simpler implementation alternatives are most certainly possible, but they are not as type-safe as generic

methods. Another obvious complication is the indirection over `With`. Why doesn't `NewObject` itself create the instance? The answer to this question is longish, so the author dedicated the next section to it.

The secret life of properties

As stated in the previous section, declaring a class is not the end of the story. re-motion uses class declarations to generate the actual (concrete) class at run-time right before it is instantiated for the first time. The code generated for simple properties follows the pattern

```
public SOMETYPE MyPropertyName
{
  get { return Properties[typeof(SOMETYPE), "MyPropertyName"]; }
  set{ Properties[typeof(SOMETYPE), "MyPropertyName"] = value; }
}
```

for `public virtual SOMETYPE MyPropertyName { get; set; }`

The `virtual` access modifier is required for domain object classes so that the code generator can override it with the re-store boiler-plate listed above. The Properties dictionary stores the actual values, using the type and the name of the property as hash.

Most of the times, if not always, your properties will do no other work except getting and setting the value stored in the database -- `{ get; set; }` will be sufficient. re-motion framework developers discourage putting more code into a domain object property. However, you should be free to make a choice, so here is how to do it.

Your API to a particular property are the methods `SetValue(value)` and `GetValue<>()`. You express this as

```
public virtual int MyMagic
{
  get { return CurrentProperty.GetValue<int> * 3; }
  if (value == 42)
  {
    throw new CosmologicalException();
  }
  set { CurrentProperty.SetValue(value); }
}
```

re-store makes sure that `CurrentProperty` represents the given property, although it is not a real variable, it's more like a placeholder for the code-generator. However, if you ever catch yourself doing this, make sure that you do it for a good reason. Value-mangling code does not belong in a domain object property but into the surrounding application code.

Using get-only properties in domain object classes

You will probably feel the desire to have properties in your domain object classes that do not store values, but compute their return value. Such properties must be attributed with `[StorageClassNone]`. Here is an example:

```
public class Cost : DomainObject
{

  // a get-only property
  [StorageClassNone]
  public virtual int TotalCost
  {
    get
    {
      return PrintingCost + PackagingCost;
    }
  }
  public virtual int PrintingCost { get; set; }
  public virtual int PackagingCost { get; set; }
}
```

This `[StorageClassNone]` attribute signals "Do not give this property a column in the database". This attribute also has implications for the display of computed values. If a get-only property does not have a `[StorageClassNone]`-attribute, re-bind (next chapter) won't show any value for that property.

Backgrounder: re-bind

Introducing `BindableDomainObject`

The word "binding" has many meanings in computing, but for re-motion it means: establishing a mapping between domain objects and their properties on the one side and controls in the browser on the other side. So far, we have looked at domain objects through the prism of persistence and relations among themselves at run-time, but this is not the only (inter)face domain objects can show. The "binding layer" can inspect domain objects via the `IBusinessObject` interface (discussed in more detail in section **Fehler! Verweisquelle konnte nicht gefunden werden.**). This, however, requires a subclass of `DomainObject`, namely `BindableDomainObject`. *Bindable* domain objects are domain objects that support the `IBusinessObject` interface. They can be asked questions about their class, properties and property values by the binding layer which in turn uses the answers to render appropriate controls on the screen. The regular (i.e. non-bindable) domain objects can't expose their guts in this fashion.

Confronted with a *bindable* domain object for which it must render a form or a grid (i.e. a set of controls, one for each property), the re-bind binding layer, or the `uigen.exe`, program generator asks the object via the `IBusinessObject` interface: "What class are you of?" The next question to ask is: "What properties constitute that class?" Finally, the list of properties is scoured by re-bind and each is asked for its type and value. The values are displayed in a meaningful format appropriate for the type – string properties as text boxes, enumerations as radio buttons, for example.

On the surface, a `BindableDomainObject` is not much different from a regular `DomainObject`. A `BindableDomainObject` class is a domain object with

- a `BindableDomainObjectAttribute`
- an extra property named `DisplayName`

This seemingly modest specialization, however, is just in the *declaration* of the `BindableDomainObject`. You might be tempted to think that BindableDomainObject is a subclass of DomainObject, but this is not so. In typical re-motion fashion, the most interesting things happen to the code at run-time.

If `NewObject` (see section *NewObject and GetObject*) finds a `BindableDomainObjectAttribute` on a `DomainObject` class (i.e. the "static type") it inspects the attribute's parameter, which specifies the mixin type that gives a DomainObject its `IBusinessObject` interface (or `IBusinessObjectWithIdentity`, a close relative). `NewObject` generates a proxy class (a.k.a. "concrete type" or "dynamic type") from the static type and the mixin type. What the `NewObject` factory returns is an instance of that dynamic type. In the current implementation the used mixin type is `BindableDomainObjectMixin`. You could write your own mixin to give `DomainObjects` its `IBusinessObject` interface.

Please note that `IBusinessObject` completely decouples the domain (or persistence) layer from the binding layer. You could use your `BindableDomainObjects` not only in re-motion, but in a very different presentation layer (WPF, for example, or even a framework for text-based terminals).

It works the other way around, too: you could write your own domain layer, complete with its own object model entirely from scratch and snap them into the re-motion binding layer if only you get the `IBusinessObject` interface right.

How `IBusinessObject*` works

re-motion needs a way to adapt to all sorts of data (i.e. persisted object) models. Some of them are highly dynamic and enable their objects to change their classes or properties at run-time. Not all re-motion applications expect their domain object data in relational databases. Some want to provide connectivity to object stores or highly dynamic object models. Consequently, re-motion supports run-time *discovery* of classes and their properties. In this context, "discovery" essentially means

- you can ask a given object of what type (class) it is (`IBusinessObject` interface)
- then you can ask that class representation of which properties it is composed of (`IBusinessObjectClass` interface)
- then you can ask each property representation for its "meta-data", i.e. type, string-length (if applicable), nullability, relationships, etc. (`IBusinessObjectProperty` interface)
- finally, you can render the appropriate control for each property, based on its characteristics

This "dialogue" with a given object is conducted via the `IBusinessObject` interface and related interfaces – `IBusinessObjectClass` and `IBusinessObjectProperty` are the most basic ones. All `BindableDomainObjects` must provide these interfaces to snap into re-motion's binding layer.

The feature-set of exposing one's type and properties clearly overlaps with .NET's reflection facilities, but please note that it covers those re-motion-specific extra attributes you use for the properties in your domain objects, like string lengths or relationships (the "meta-data"). Also note that the discovery-scheme is more generic (simpler) than that of .NET and can be integrated with non-.NET entities if only they can be understood as

- classes are typed sets of properties
- objects of that class "have" those properties.

What the `IBusinessObject*` interfaces share with reflection is that they make possible a completely generic binding layer. No compile-time knowledge on which properties or what types thereof is required. All discovery of properties and their types happen at run-time, just pass the object instance. The binding layer must know how to use the `IBusinessObject*` interfaces, that's it.

"Business object" vs. "domain object"

At this point some clarification of the terms "business object" and "domain object" is in order. The term "business object" is well-established in the world of enterprise computing and roughly corresponds to our understanding of re-motion's "domain object". It is an entity living in the database which maps to some entity from the business domain – order, check, employee, invoice, etc. The definition in Wikipedia is "Business objects are objects in an object-oriented computer program that represent the entities in the business domain that the program is designed to support." (http://en.wikipedia.org/wiki/Business_object_(computer_science)). Not by coincidence, this is our understanding of the term "domain object", and not by coincidence, the wikipedia article on "domain object" redirects to... business object.

In re-motion, a business object and its properties typically reflect business domain objects, too, but with one big difference: it is not necessarily persisted. *Search objects*(not covered by this tutorial), for example, are business objects, but they are short-lived – created at run-time, they never make it to the database. In shorter words:

- Business Objects expose their guts to the binding layer via the IBusinessObject interface
- Domain Objects can store their property values in a database

In a re-motion application, however, you need domain object instances that can do both. After all, you want to persist your values in the database, and you want to display values on your web-pages. This is exactly what BindableDomainObject is for.

Using `BindableDomainObject` in the PhoneBook.Domain library

In the next section you will learn how to use the ASP.NET program generator for generating an ASP.NET web application from your domain object classes. This requires that your domain object classes are first-class citizens for the binding layer re-bind.

Consequently, you must switch the base class of the PhoneBook domain classes from `DomainObject` to `BindableDomainObject`.

In `Location.cs`:

```
public class Location : ~~DomainObject~~ BindableDomainObject { ...}}
```

In `Person.cs`:

```
public class Person : ~~DomainObject~~ BindableDomainObject { ... }}
public class PhoneBook : ~~DomainObject~~ BindableDomainObject { ... }}
```

To resolve `BindableDomainObject`, use the namespace `Remotion.Data.DomainObjects.ObjectBinding`.

The `BindableDomainObject` class in turn requires adding two new references to your `PhoneBook.Domain` project:

- `Remotion.ObjectBinding`
- `Remotion.ObjectBinding.Interfaces`

If you went with the re-motion convention, you will find these assemblies in the re-motion binaries directory

```
C:\PhoneBook\Remotion\net-3.5\bin\Debug
```

BindableDomainObject's `DisplayName`

For all that hard work of comprehending `BindableDomainObject` and how to use it, your domain object classes inherit a new property: `DisplayName`. `DisplayName` is a somewhat misleading name, because `DisplayName` gives you an overridable property for customizing "short-hands" or "excerpts" of a domain object's *value*.

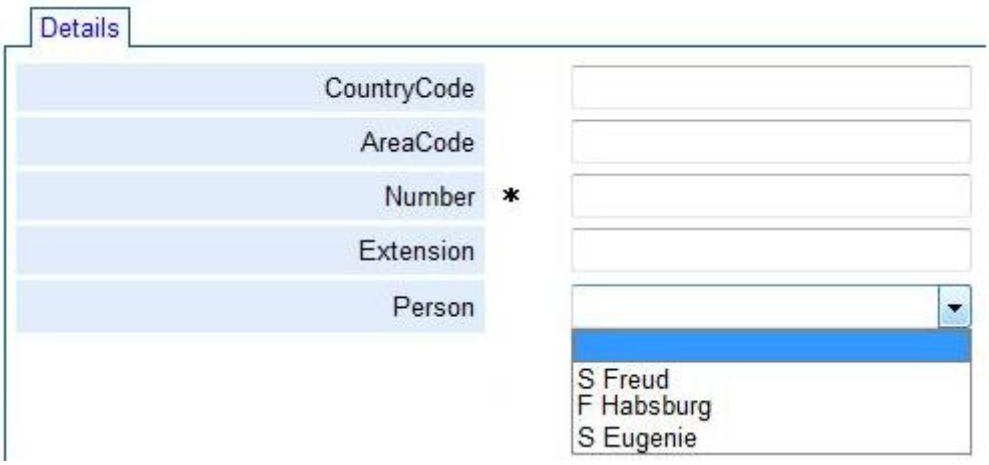
For example, a person's full name might include first name and surname, but we want to give a domain object a facility to obtain a more compact form for more convenient display like showing only the first initial of the first name and the surname. We can program `DisplayName` as

```
public override string DisplayName
{
  get
  {
    if (FirstName != null && FirstName.Length > 0)
    {
      return FirstName[0] + " " + LastName;
    }
    else
    {
      return "? " + LastName;
    }
  }
}
```

It is often handy for the user to display brief versions of the most important properties of an object in the view of the owning parent object. This is an automatic feature of bidirectional relationships: the `PhoneNumber`'s `Person` reference property (`BOCReferenceValue`) is reflected by its `DisplayName` by default. It will display its values in a grid looking like this:

CountryCode	AreaCode	Number	Extension	Person
43	1	3191596		S Freud
43	676	555-0003		S Eugenie
43	1	555-0001		F Habsburg

Or in a menu, looking like this:



In other words: `DisplayName` is the canonical way to derive an excerpt or brief version of all or some of a domain object's properties in a single string -- the domain object instance's elevator pitch, if you will.

Improving our PhoneBook.Sample application with `DisplayName`

Remember that, in the previous chapter, we used listed all objects in the database. To this end, we formatted the properties of each object in the console output:

```
        Console.WriteLine(loc.Street);
        Console.WriteLine("    {0} {1}", p.FirstName, p.Surname);
        Console.WriteLine("   +{0} ({1}) {2}/{3}",
                           phone.CountryCode,
                           phone.AreaCode,
                           phone.Number,
                           phone.Extension);
```

We can improve this code by standardizing how properties are displayed for each object, by adding appropriate formatting in each `DisplayName`:

```
// In class Location we use just the street name,
// for brevity:
public override string DisplayName
{
  get
  {
    return Street;
  }
}

// In class Person we abbreviate the first name,
// if any:
public override string DisplayName
{
  get
  {
    if (FirstName != null && FirstName.Length > 0)
    {
      return FirstName[0] + " " + Surname;
    }
    else
    {
      return "? " + Surname;
    }}
// In class PhoneNumber:
public override string DisplayName
{
  get
  {
    return String.Format("   +{0} ({1}) {2}/{3}",
                         CountryCode,
                         AreaCode,
                         Number,
                         Extension);
  }
}
```

The new and improved code in PhoneBook.Sample to list all objects thus looks like this (old lines in comments):

```
static void Main(string[] args)
{
  // Freud and Habsburgs removed
  // because they are in the database already
```

```
    // EnterFreud();
    // EnterHabsburgs();

    using (ClientTransaction
                .CreateRootTransaction()
                .EnterDiscardingScope())
    {
      foreach (Location loc in Location.AllLocations())
      {
        // Console.WriteLine(loc.Street);
        Console.WriteLine(loc.DisplayName);
        foreach (Person p in loc.FindPersons())
        {
          // Console.WriteLine("    {0} {1}", p.FirstName, p.Surname);
          Console.WriteLine(p.DisplayName);
          foreach (PhoneNumber phone in p.PhoneNumbers)
          {
            // Console.WriteLine("   +{0} ({1}) {2}/{3}",
            //                     phone.CountryCode,
            //                     phone.AreaCode,
            //                     phone.Number,
            //                     phone.Extension);
            Console.WriteLine(phone.DisplayName);
          }
        }
      }
      Console.ReadLine();
    }
  }
```

re-bind and BOC controls

In its original sense, a *control* is a special type of software object, complete with a type, properties and a certain behavior. A control's distinct feature is that it also has a visual appearance, and that its behavior is tied to visual control elements like buttons, text boxes, menus and the like. Controls are objects that give a user a handle on data. For ASP.NET, as we will see in a moment, these observations are an oversimplification, but it will serve us well for the following discussion.

Another feature that controls share with other software objects is that of *containment*, i.e. controls can be nested = controls can be composed of other controls to arbitrary levels in a wheels-within-wheels fashion. A query entry control might be composed of a textbox together with a "submit" button. A login control might be composed of two textboxes; one for the login name, one for the password.

Other controls you probably know already:

- text entry boxes
- list controls
- drop-down list boxes
- buttons

Examples of "compound" controls:

- a text entry box with a "submit" button (for a search engine, for example)
- a set of text-boxes constituting fields in a form

- a date picker composed of three drop-down lists for month, day and year

ASP.NET's achievement is to wrap up the web's mechanics of

- a stateless http protocol
- requests for interacting with the server
- html for rendering controls
- code that runs on the server

in such a fashion that web controls in web applications look and feel like real, seamless software objects. This is more difficult than it might sound, because the web's native mechanics is notoriously hostile to object-oriented programming. (In contrast to "regular" GUI-programming, what is a mainstay of object-oriented programming, of course. GUIs and OOP grew up together in the 1970s and 1980s.)

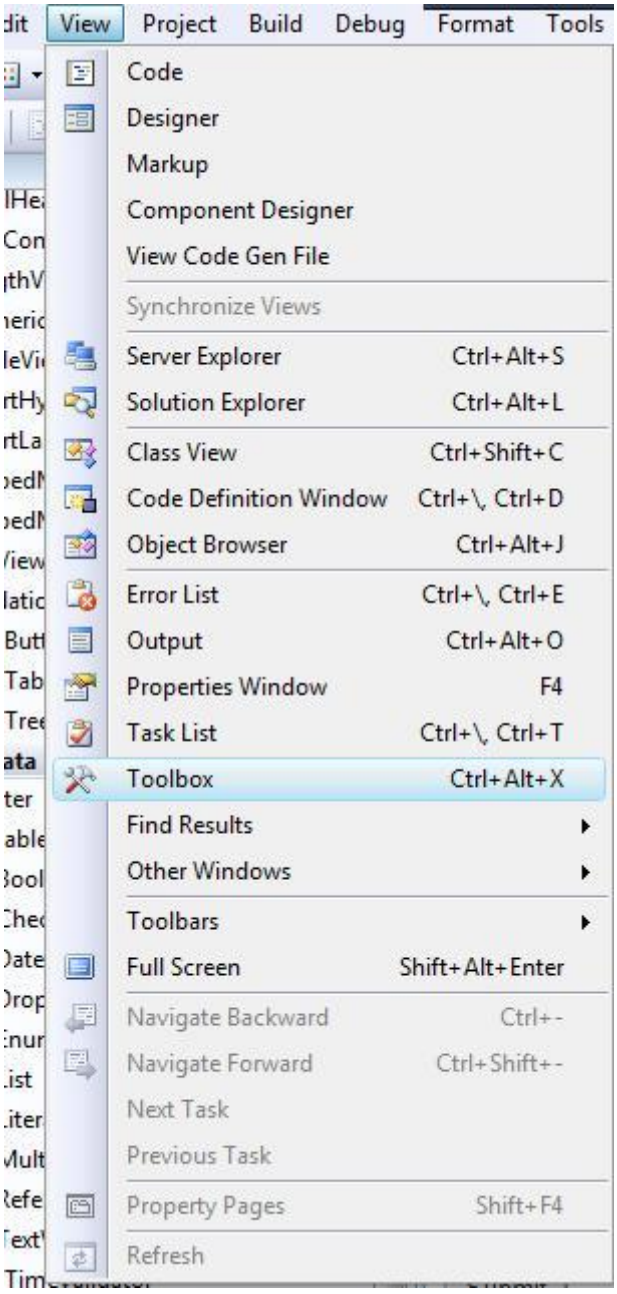
Connecting data to controls for representation in the browser is easier in ASP.NET than starting on the web's bare metal, but still serious work, even for smallish applications. With ASP.NET 2.0 came some facilities for handling the updating of data, i.e. propagating user modifications to the objects that actually hold the data. For ASP.NET 2.0 applications you must design forms, write validators for input data, take care of loading and saving of data at the right time, have to provide and use facilities for localization, have to orchestrate field names, column headings, error messages and the like. Another issue is user privileges – not all data is supposed to be modified or seen by any user, what requires careful design and testing. This is hard work, but flexible and appropriate for all conceivable types of web-applications -- blog, dating, phone-directories, whatever.

re-motion's (or better: re-bind's) object binding is built on top of ASP.NET and provides *two-way* data-binding for re-store domain objects. Without extra code, a re-bind control knows how to update a domain object's data when a control tied to it asks it to do so. This is possible because the rendering and mapping of domain object properties is standardized. Another pillar for re-bind's automation features is that re-bind controls can inspect domain objects to find out how data should be rendered and what operations are permitted:

- which fields (properties) have mandatory input?
- is a property (field) read-only?
- how long are strings that fit into a given string property?
- of what type is a property to begin with?

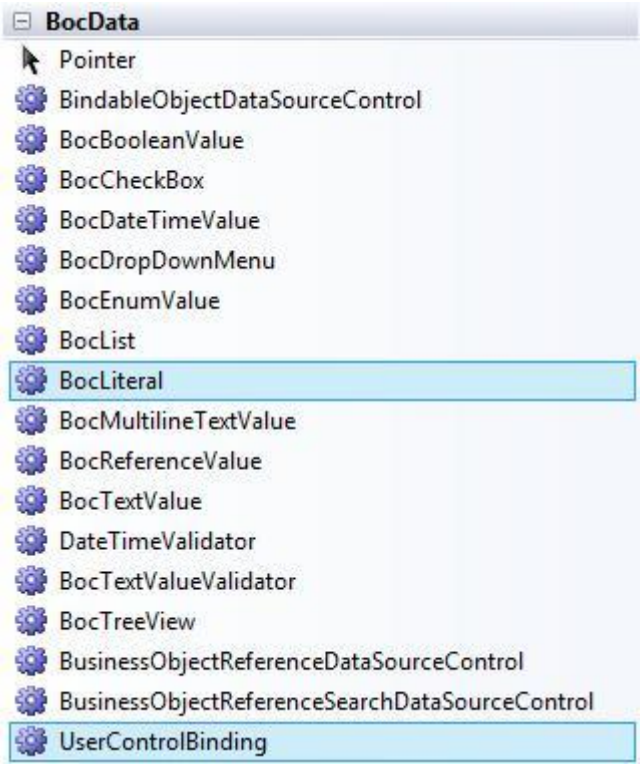
What's more, re-bind knows what input to expect and validates it in compliance with the value's type. If a user enters "five" where an integer is expected, the user is notified about this on the page without requiring any extra work by the programmer. In a word, re-bind provides input-validating controls automatically, together with column headings, field names (= "labels") and the like – for multiple languages, if desired. re-bind controls are integrated into Visual Studio and support rendering and handling in Visual Studio's designer, just like regular ASP.NET controls from Microsoft. You can try this out yourself by adding the re-bind controls to your toolbox in Visual Studio.

Make the toolbox visible with the *View's Toolbox* command:



- right-click for the menu-items in the bottom "General" section of the toolbox, and select *Add Tab*. Name that tab "BocData".
- in that tab, right-click for *Choose items...* and browse for the DLL Remotion.ObjectBinding.Web.dll in your re-motion binaries directory (`C:\PhoneBook\Remotion\net-3.5\bin\Debug`, Visual Studio might be very slow here. You might have to wait for the dialog to show up.)

The new tab should look like this after the loading:



Unfortunately, you can't try the drag and drop features at this point in this primer, because Visual Studio's designer requires re-motion assemblies in the *global assembly cache*. However, we will get there in section *Where this tutorial is going, Getting rid of* `DisplayName`. If you try to use it at this point, VS designer will show you error messages instead of controls.

BOC data controls

Most of the BOC controls you see listed in the toolbox are concerned with simple values, like numbers and strings. The most complex control is `BocList`, a control that gives you spread-sheet like lists for tabular data, i.e. lists of business objects. Another complex control is the `BocTree` for displaying a hierarchy of domain objects.

Controls for simple values:

- `BocBooleanValue`
- `BocDateTimeValue`
- `BocEnumValue`
- `BocTextValue`
- `BocMultiLineTextValue`

These BOC controls let the user interact with the suggested property types. However, in some cases you have various options for which ASP.NET control shall be used for rendering the BOC control. An extreme case is `BocBooleanValue`, which can be rendered as:

- a check box, or a three-state check box, if you want `True/False/null (Boolean?)`
- a set of radio buttons (true/false, male/female, raw/cooked, yin/yang, etc.)
- a drop-down menu

- a list box

`BocEnumValue` runs along similar lines, just with an arbitrary number of items to choose from.

You can specify which control to use for rendering the options by setting the corresponding attribute for the BOC control, or pick it in the property browser in Visual Studio's designer. We will use this feature in the section *Getting rid of `DisplayNames` (introducing column definitions)*.

Complex controls:

- `BocList`
- `BocTreeView`

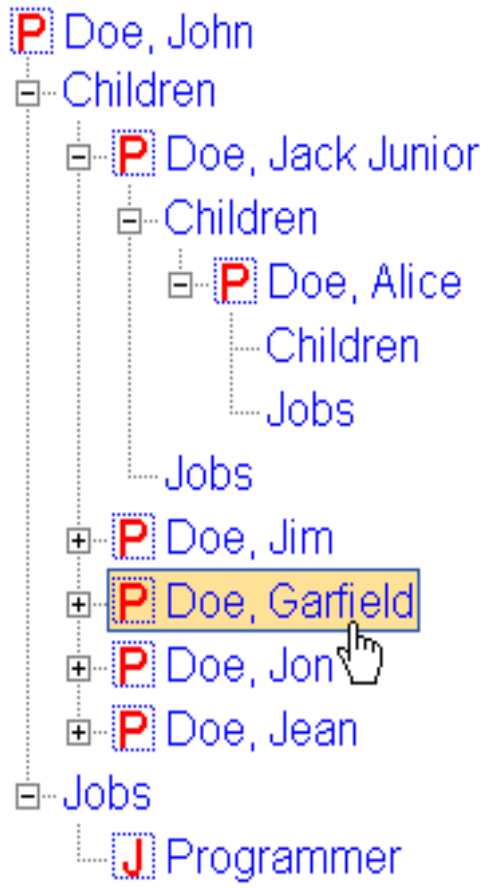
`BocList` and `BocTreeView` are complex controls with a large feature-set, and they provide ample room for customization, both in code and by using Visual Studio's designer. Each of these controls deserves a tutorial as large as the one you are reading, but they are not covered here, or anywhere else at this time, for that matter. The good news is that such tutorials are planned for the near future.

Here is a picture of a typical `BocList`:

Applikation ▲ 1	Ressource ▲ 2	Benutzer/Gruppe ▲ 3	Zugriff	Aktion
Informations- und Arbeitsportal	Bausparen & Finanzen	rubicon\Account Operators	Erlaubt	Löschen
Informations- und Arbeitsportal	Bausparen & Finanzen	rubicon\Administrators	Erlaubt	Löschen
Informations- und Arbeitsportal	Bausparen & Finanzen	rubicon\alex.baumgartner	Erlaubt	Löschen
Informations- und Arbeitsportal	Bausparen & Finanzen	rubicon\alice.harrer	Erlaubt	Löschen
Informations- und Arbeitsportal	Bausparen & Finanzen	rubicon\All	Erlaubt	Löschen
Informations- und Arbeitsportal	Bausparen & Finanzen	rubicon\andreas.schlapsi	Erlaubt	Löschen
Informations- und Arbeitsportal	Bausparen & Finanzen	rubicon\anton.secmantest	Erlaubt	Löschen
Informations- und Arbeitsportal	Bausparen & Finanzen	rubicon\arthur.secmantest	Erlaubt	Löschen
Informations- und Arbeitsportal	Bausparen & Finanzen	rubicon\ASPNET	Erlaubt	Löschen
Informations- und Arbeitsportal	Bausparen & Finanzen	rubicon\ausschreibungen	Erlaubt	Löschen
Informations- und Arbeitsportal	Personelles (Mitarbeiter)	rubicon\krbtgt	Erlaubt	Löschen
Informations- und Arbeitsportal	Vorlagen	rubicon\mailsender_hpm	Erlaubt	Löschen
Informations- und Arbeitsportal	Vorlagen	rubicon\Management Assistant	Erlaubt	Löschen
Informations- und Arbeitsportal	Vorlagen	rubicon\Management AT	Erlaubt	Löschen
Informations- und Arbeitsportal	Vorlagen	rubicon\Management CH	Erlaubt	Löschen

Seite 1 von 2

And here is a picture of a typical `BocTree`:



The BOC data controls owe their name to the fact that they bind to *business objects*, typically domain objects. This binding is mediated by *data source controls*, which require more explanation.

The data source "control"

We started this section with the observation that controls are objects with a visual representation, but the distinct feature of *data source controls* is their invisibility. Like other web-controls, they have a representation *on the web-page*, but they are not rendered. Their sole purpose is to relay data from business objects to the controls that actually have a visual representation on the page. When a page is loaded, its data source control fetches the data and gives "actual" controls (like textboxes or radio buttons) the data to display. When a page is posted back to the server, the data source collects the state from its control(s) and writes them back to the objects the data came from in order to update them. Data source controls are a Microsoft invention and have been introduced with ASP. They are an abstraction mechanism to shield visual controls from the peculiarities of data access as exposed by the objects that actually keep the data.

In its simplest form, an ASP.NET data source control binds to a simple value, like a checkbox to a Boolean. More complex uses involve complete objects with a number of properties that bind to a single data source. This data source in turn is used by a composite control with each sub-control accessing one property.

We leap ahead and presume we have our web application (section *ASP.NET programming in 21 minutes*) already to give you an example on how this works. The following listing shows you a

fragment of a web-form for editing a `Location` object, which looks somewhat like this when rendered in a browser:

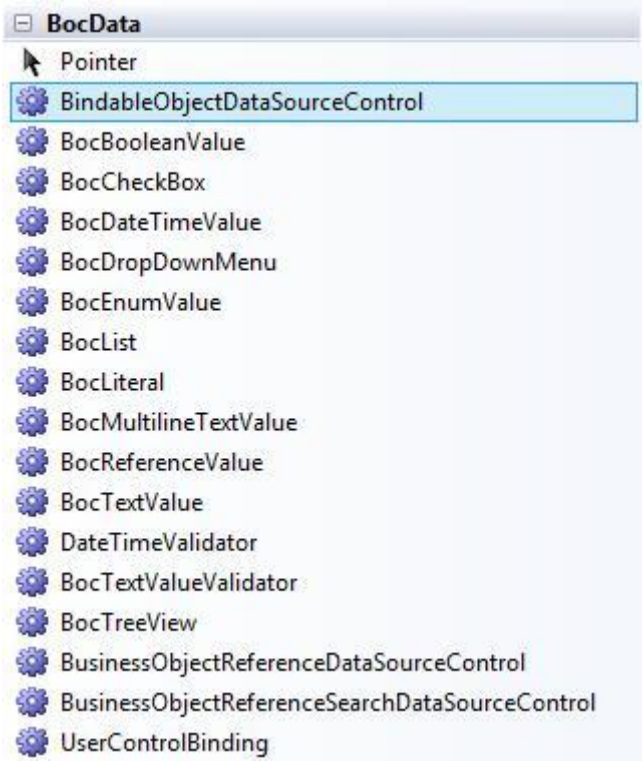
The following is the corresponding listing with comments (you can find this file as `datasource.aspx` in the `instruct\tutorial\tutorial-files` folder in your distribution):

```
<!--
  here we declare the BindableObjectDataSourceControl with the name "ExampleDS".
  We bind the data source control to the "Location" class. At run-time, the "current"
  domain object instance will be used to provide the actual data, or a home for
  the actual data. The loading of this "Location" domain object instance and making
  sure that it actually IS of type "Location" is all handled by re-motion and not
  reflected by this declaration.
 !-->
<remotion:BindableObjectDataSourceControl ID="ExampleDS" runat="server" Type="PhoneBook.Domain.Location,
PhoneBook.Domain" />

<div>
  <tr>
    <td>
      <!-- This "StreetField" knows that its data to display (or update in the case of
           modification by the user) comes from the "ExampleDS" data source control
           declared above. It also knows that the text value it is responsible for stems
           from the "Street" property in the bound "Location" domain object mediated
           by the data source above.
       !-->
      <remotion:BocTextValue ID="StreetField"               <!-- arbitrary ID !-->
                             DataSourceControl="ExampleDS"  <!-- "reference" of the data source !-->
                             PropertyIdentifier="Street" >  <!-- specifies the domain
                                                                 object property !-->
                             runat="server"                 <!-- ASP.NET staple !-->
      </remotion:BocTextValue>
    </td>
  </tr>
  <tr>
    <td>
      <!-- More of the same with the "Location"'s "City" property. !-->
      <remotion:BocTextValue ID="CityField"                 <!-- arbitrary ID !-->
                             DataSourceControl="ExampleDS"  <!-- "link" to the same data source
                                                                 because we use the same
                                                                 "Location" object here. !-->
                             PropertyIdentifier="City"      <!-- the "City" prop in "Location" !-->
                             runat="server" >

      </remotion:BocTextValue>
    </td>
  </tr>
  <tr>
    <td>
      <!-- Still more of the same with the "Location"'s "Country property. !-->
      <remotion:BocEnumValue ID="CountryField"              <!-- el ID !-->
                             DataSourceControl="ExampleDS"  <!-- use the "ExampleDS" data
                                                                 source for providing the
                                                                 object with the property
                                                                 which provides the value
                                                                 to be displayed in the control --
                                                                 just as for the two controls
                                                                 above. !-->
                             PropertyIdentifier="Country"   <!-- Country property here !-->
                             runat="server" >

      </remotion:BocEnumValue>
    </td>
  </tr>
  <tr>
    <td>
      <!-- Time for the "ZipCode"...
      <remotion:BocTextValue ID="ZipCodeField"              <!-- your name shall be "ZipCodeField" !-->
                             DataSourceControl="ExampleDS"  <!-- again, your data comes from
                                                                 "ExampleDS"...
                             PropertyIdentifier="ZipCode"   <!-- ... or, to be more specific, the
                                                                 "ZipCode" property in the domain
                                                                 object behind "ExampleDS" !-->
                             runat="server" >
```

```
      </remotion:BocTextValue>
    </td>
  </tr>
</table>

</div>
```

(This example corresponds *not quite* to the form's screenshot used as illustration, as we will discuss in a minute.)

The central element in this listing is the `BindableObjectDataSourceControl` "`ExampleDS`". In the listing, "`ExampleDS`" is an instance of `BindableObjectDataSourceControl`, which you can see as the first among the *BocData* controls in the toolbox:



This is re-bind's universal data source control, and the only one necessary to relay values from business objects to re-bind's control. It is `BindableObjectDataSourceControl` which knows how to query a business object's (domain object's) `IBusinessControl` interface. The results from that querying is provided to enquiring controls (like `BocTextValue`) who in turn know how to talk to `BindableObjectDataSourceControl`.

The "other" BOC controls

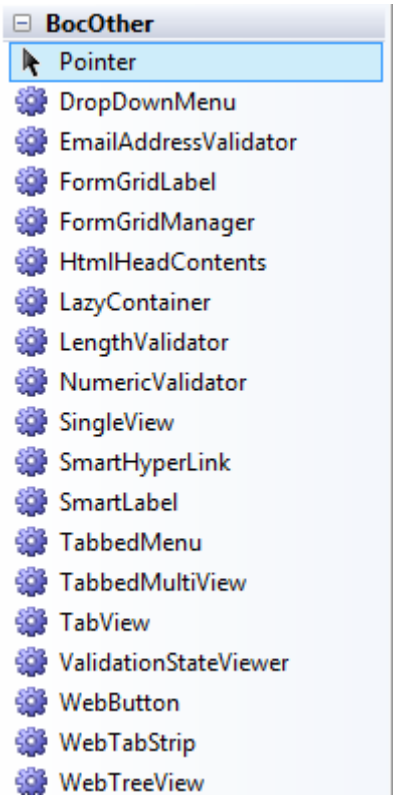
BOC data controls bind single BOC controls to business objects (typically domain objects), but that's only one side of the story. Other controls - called "BOC other" in re-motion parlor, provide validating, embedding and formatting controls.

The *BocOther* controls are located in the assembly `Remotion.Web`. To add them to your toolbox,

- right-click for the menu-items in the bottom "General" section of the toolbox, and select *Add Tab*. Name that tab "BocOther".

- in that tab, right-click for *Choose items...* and browse for the DLL Remotion.Web.dll in your re-motion binaries directory (`C:\PhoneBook\Remotion\net-3.5\bin\Debug`, Visual Studio might be very slow here. You might have to wait for the dialog to show up.)

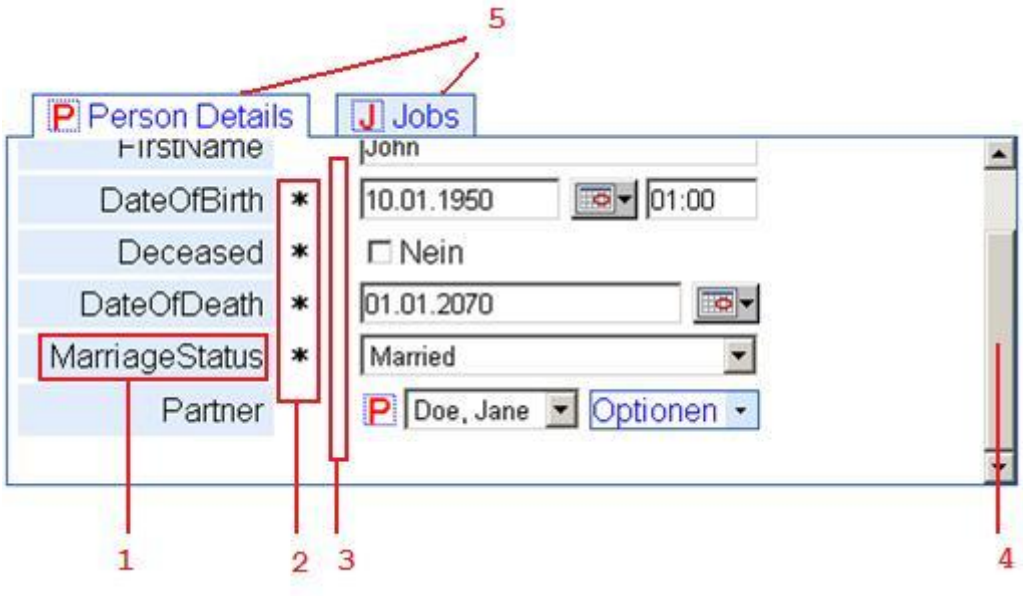
The new tab should look like this after the loading:



The most important *BocOther* controls are:

- **smart labels**
  Smart labels provide a label for other controls. They save form-designing people labor, because the label text is fetched from the control's property itself. What's more, smart labels enable *localization* for labels, i.e. displaying label text for various languages.
- **form grid manager**
  The form grid manager makes controls in a form look neat. It centers multiple controls in a column around a common axis and provides a "*" for required fields. The impression you got from the `Location` form screenshot above is courtesy of the form grid manager.
- **tabbed multiview**
  A large form might not fit on a single page, so multiple "tabs" are required, a method originally invented for modal dialogs. The *tabbed multiview* provides features for displaying domain objects with many properties in limited space on the page.

Here is a screen-shot of a typical edit form in re-motion where you see smart labels, the form grid manager and a tabbed multiview at work, the numbers refer to annotations below:



1. Labels like *MarriageStatus*, *DateOfDeath*, etc. are fetched from the domain object (or resources referenced by it) and displayed by the `SmartLabel` control.
2. The stars for required fields are displayed by the `FormGridManager` control.
3. Observe that labels are left-aligned, controls are right-aligned, giving the whole group a polished look. This, too, is the work of the `FormGridManager`.
4. The scroll bar is provided by the `TabbedMultiview` control embedding all controls on that view.
5. These tabs, too, are the work of the `TabbedMultiview` control, which give the control its name.

Validating BOC controls

Many properties are mapped to text input fields, including numerical properties. For numerical properties, the string the user enters as text must be parsed and validated to make sure that a number actually IS a valid number. This means that you map a string property for arbitrary numbers to `BocTextValue`, and re-bind automatically supplements the numerical validator. Another supported validator is that for email addresses, `BocEmailAddressValidator`. You may want to write your own validator for domain-specific items like serial-numbers or ISBN-numbers, for example.

Values and BOC controls

`BocBooleanValue`

As the name suggests, `BocBooleanValue` is used for two mutually exclusive alternatives – true and false. However, since a Boolean property can be nullable, `BocBooleanValue` can also handle tri-state "Booleans" for true, false and null.

Since users prefer descriptive annotations like "male/female", "yin/yang", "do/don't" over "true/false", it is a good idea to set the `BocBooleanValue's` properties `truedescription` and `falsedescription` accordingly.

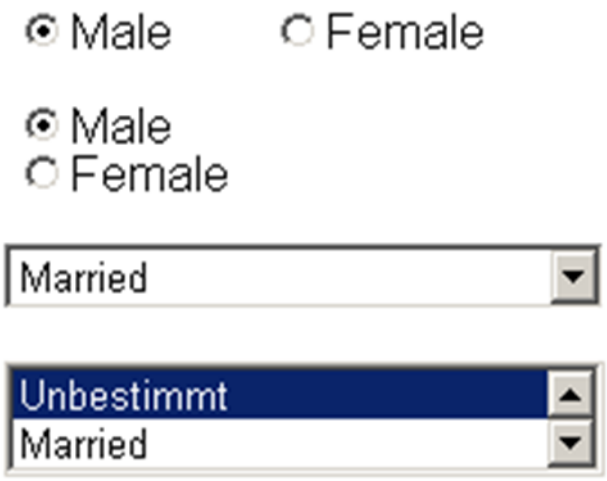
How a Boolean property is rendered is a matter of expected user preferences, shouting matches. A Boolean can be rendered as

- two (or three, for tri-state) radio-buttons
- a drop-down list with two or three items
- a check-box (or two, one extra for "undefined")
- a list-box

Which one to use can be specified in the control's property `ListControlStyle/ControlType`. You can set this property either

- in the `BocBooleanValue`'s property explorer (in designer)
- as HTML declaration, with the sub-node `ListControlStyle`. Example: `<ListControlStyle ControlType="RadioButtonList" />`

In read-only mode the `BocBooleanValue` is displayed as a tri-state checkbox or a label with static text. Here are pictures of the controls for Booleans, in various costumes:



BocEnumValue

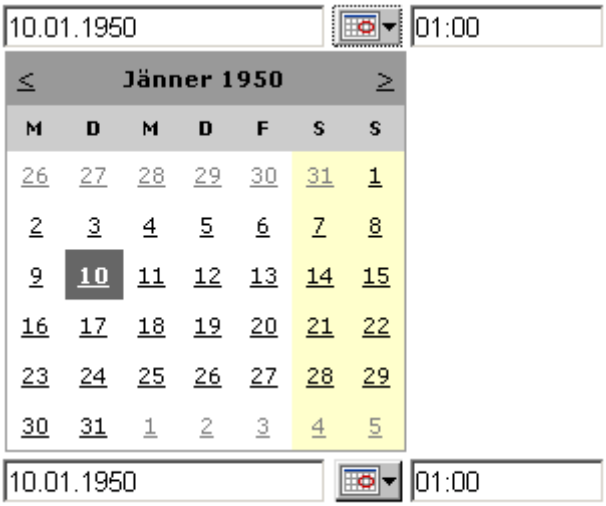
`BocEnumValue` is similar to `BocBooleanValue` in that it offers enumerated alternatives, albeit more than two (or three, if you include null). Consequently, similar options for displaying enum properties exist:

- radio buttons
- drop-down list
- list-box

Which on to use can be specified in the control's property, just like for `BocBooleanValue` (see above). In read-only mode the `BocEnumValue` is always displayed as a label with static text.

BocDateTimeValue

`BocDateTimeValue` displays or modifies date/time-values. The user can enter the date as string (in the format appropriate to his culture) or pick it from a pop-up calendar. This is what the `BocDateTimeValue` control looks like:



Several options exist:

- suppress display/editing of the time value
- suppress display/editing of the date value

In read-only mode, the time and/or date is simply displayed as static text, with or without a label.

BocTextValue

As pointed out in the section *Values and BOC controls*, `BocTextValue` is used for handling various types of value:

- string
- integers (`Int32`)
- double
- date/time (see previous item)

The difference between values is how they are validated by the controls. Strings can be validated for maximum length with the `LengthValidator`.

As you might guess, `BocTextValue` is a fairly rich control with many configurable options. You can specify

- which value type to display
- a format string for rendering the data
- multi-line editing

In read-only mode the text is rendered as a label.

BocMultiLineTextValue

In contrast to `BocTextValue`, `BocMultiLineTextValue` is only used for text, or better: a string-list. It is different from `BocTextValue` in that it handles text as a `List<string>`, with each element being a line. You can specify a maximum length for each line. In read-only mode the text is displayed as a single label.

BocReferenceValue

As the name suggests, `BocReferenceValues` don't hold domain object data themselves, they point to other domain objects holding the data. An example is the Location property in a Person domain object. A reference property is easy for programmers to model in their code; for users not so easy. Confronted with a reference value, typical user desires are:

- display a meaningful abbreviation for the referenced object's values
- change the reference by finding an object to be referenced
- change the reference by creating a new object to fill the reference value

We've seen how the first point in the list can be addressed by overriding `DisplayName`. Using a `DisplayName`, if provided, is in fact the default rendering behavior of `BocReferenceValues`. The other two points require some programming and configuration.

Generating the PhoneBook web application

Introducing the web application generator -- uigen.exe

With a set of real-working domain objects - the *model* - we can proceed to adding a web user interface.

In the age of the internet it has become wide-spread practice to use a web browser as remote control for database applications like our PhoneBook.

For our PhoneBook application, large parts of this user interface can be generated automatically. The console application `uigen.exe` inspects your DLLs with domain objects and its properties and generates a basic ASP.NET application for managing those domain objects. `uigen.exe` is supposed to build a working prototype quickly, or to spare you from the labor of writing a lot of boiler-plate code.

The usual procedure for creating a re-motion application is this:

- write (and test) your domain objects
- let `uigen.exe` generate a basic application
- take it from there and embellish your application

Some of these embellishments require the modification of generated files. Note that `uigen.exe` is *not* a round-trip tool. You are supposed to run `uigen.exe` ONCE, then go with the generated application. There is no easy or automatic way to feed back modifications into the application when `uigen.exe` runs again.

As with `dbschema.exe`, `UIGen.exe`'s output is based on the input domain objects. `UIGen.exe` inspects the domain DLLs and constructs the forms and controls and how they interact from the (annotated, typed) properties in the domain objects.

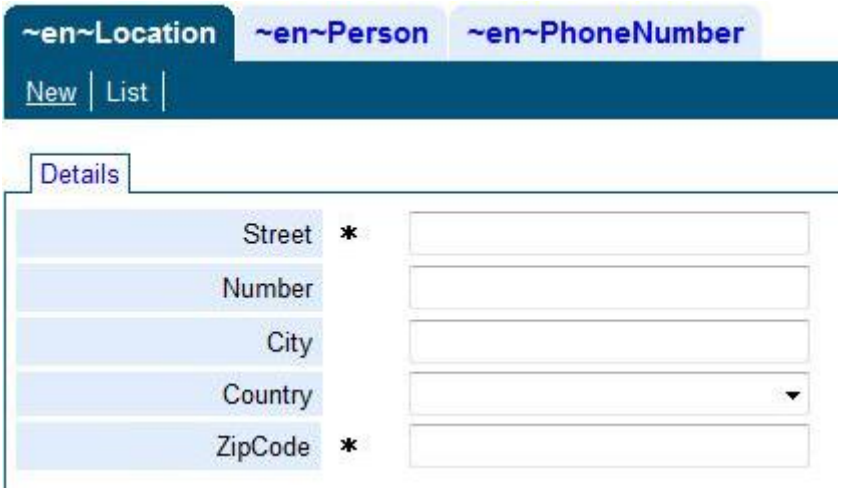
`UIGen.exe` creates these ASP.NET files (and many other) from templates, using a simple text substitution mechanism. Among the generated files is a complete Visual Studio 2008 project file. As soon as the project has been generated, you are good to go and include this project into your PhoneBook solution, compile it and run it. The next section will give step-by-step instructions on how to do this.

The next illustrated section will show you how the freshly generated PhoneBook application will look like, and how it is structured. Afterwards we will turn to the interesting part: adding features and polish to make the `uigen.exe`-generated application look like the one you can see (and test) at `http://re-motion.org/Phonebook`.

PhoneBook web app impressions

The application starts up with a "splash screen" sporting a rubicon logo and a "Start" link for firing up the actual application.

After clicking on "Start" you will see the PhoneBook application in a new browser window. It gives the user three tabs – one for each type of domain object (`Location`, `Person`, `PhoneNumber`). The tab as central element gives this style of web-application its name: "tabbed editor application". Clicking on the "Start" link will move you to an empty "edit form" for `Location`:

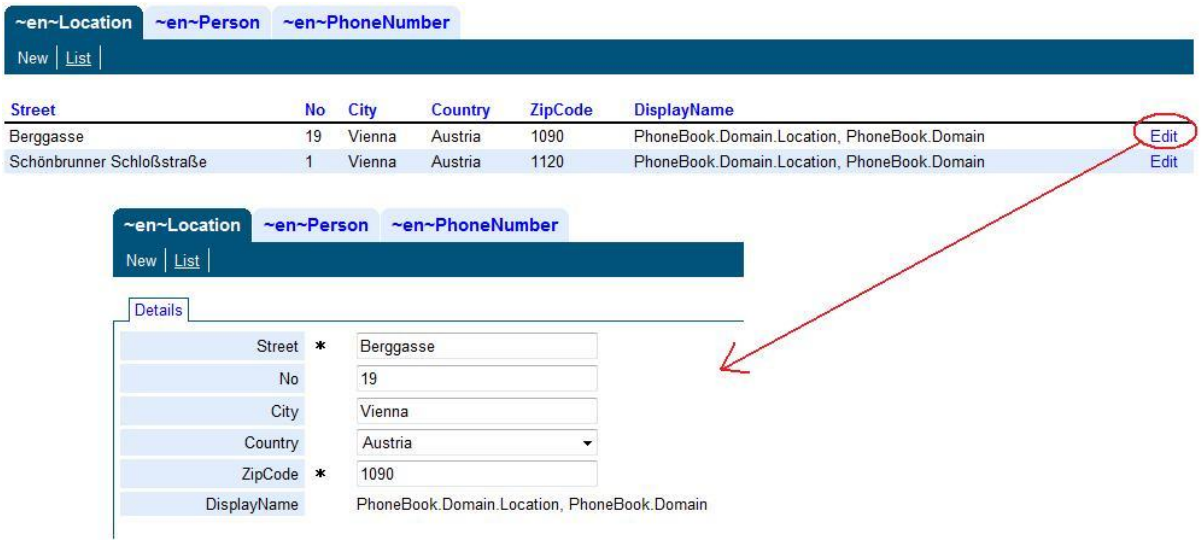


This is the *EditLocationForm* view. The strange ornamental "~"s in the tabs will be generated by uigen.exe, because it does not know yet what "Location" (for example) means in English. "~en~Location" is simply a string meaning "'Location' in English here". "~de~Location" is the dummy string for "'Location' in German here".

On the same tab we find an alternative view, the *SearchResultLocationForm* view. It lists all `Location` objects in the PhoneBook, or those passing a search query. If you kept the Habsburg royals and Sigmund Freund from the

previous chapter in your PhoneBook, clicking on the "List" link will give you the following view on the data:

You can whip up any location in the list in an edit form by clicking its "Edit" link:



To return to the initial empty form for creating a new `Location` object, just click on the "New" link (New).

This logic of a list view and an edit view on the tab works in analogous fashion for `Person` and `PhoneNumber` – a tab, a "new", a "list" for each class of domain object. The default applications includes all sorts of lovingly rendered features already, but we will postpone discussion of those until we have actually built it so you can try them out yourself. The important point here is the rough structure of a tabbed editor application.

The files generated by the `UIGen.exe` program reflect this structure. `UIGen.exe` extrudes a set of ASP.NET files for each domain object class. In the following list "X" is a placeholder for each domain object class name (`Location`, `Person`, `PhoneNumber`):

- **form files for the edit view (ASP.NET pages):**
  - Edit"X"Form.aspx
  - Edit"X"Form.aspx.cs
  - Edit"X"Form.aspx.designer.cs

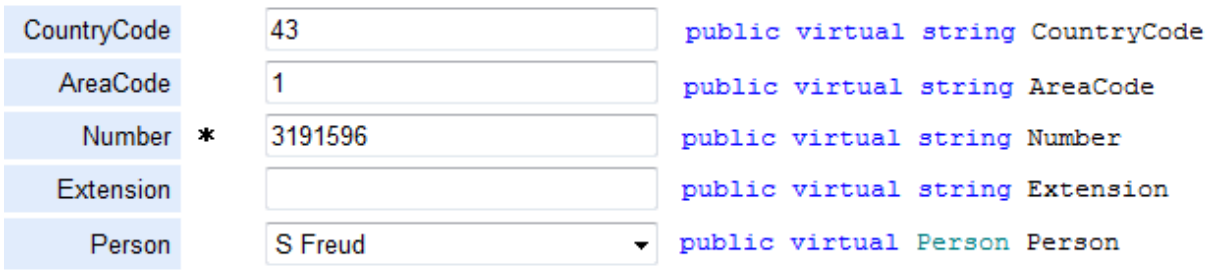
- **ASP.NET control files:**
  - Edit"X"Control.ascx

    - Edit"X"Control.ascx.cs
    - Edit"X"Control.ascx.designer.cs

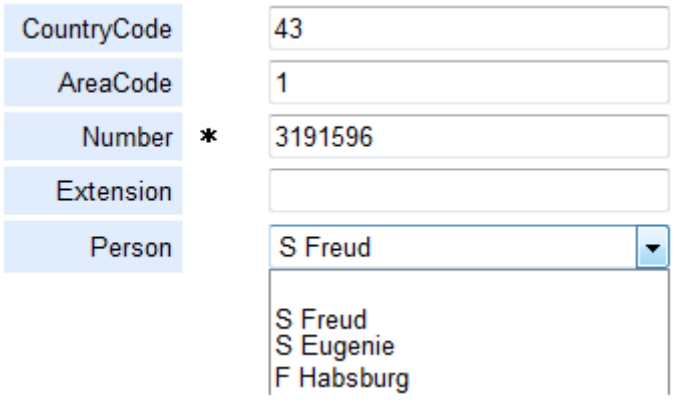
- **search form files for the list view (ASP.NET pages):**
    - SearchResult"X"Form.aspx
    - SearchResult"X"Form.aspx.cs
    - SearchResult"X"Form.aspx.designer.cs

Even if we hadn't told you before these patterns of ".aspx" and ".ascx" name fragments reveal it to the expert: `UIGen.exe` generates an ASP.NET project.

As for re-bind controls - or "BOC controls" - the Edit"X"Control.ascx.* files each contain an "edit control". An edit control is composed of BOC controls, with each control corresponding to a domain object class property. The type of the property is reflected by the type of BOC control, as in this illustration using `PhoneNumber` as example:



The *Person* field clearly is a reference property. It will have its value rendered as a menu so that the user can select a different owner for the phone-number:



`uigen.exe` will give you a working, but unpolished (and unglobalized) application. The next section will discuss how `uigen.exe` is used.

Editing and using the `uigen.exe` configuration file

The console application `uigen.exe` generates a complete *tabbed editor application* from templates, based on the classes and properties it finds in the domain assemblies. For `uigen.exe` to do its work you have to tell it where it can find

- the templates
- the domain assemblies
- re-motion assemblies
- some other tidbits, like the domain project GUID and the first page you want to see in your running web client application

Some of these specs you provide in command-line parameters, some you provide in a `uigen.exe` configuration file. There is a ready-made PhoneBook.xml file at

`http://re-motion.org/content/PreparedFiles/PhoneBook`

Please note that `uigen.exe` is not exactly a polished application, and that it will be rewritten in the near future to make it more robust and less finicky. Now is the time when it will come and haunt you if you don't have your PhoneBook tutorial project in `C:\PhoneBook` and your re-motion assemblies in `C:\PhoneBook\Remotion\net-3.5\bin\debug`, because `uigen.exe` has problems with relative paths.

For our PhoneBook web client application, we assume a configuration file named `PhoneBook.xml`. Here is a listing how your `PhoneBook.xml` should look like:

```
<?xml version="1.0" encoding="utf-8" ?>

<applicationGenerator
template="C:\PhoneBook\Remotion\UIGenTemplates\TabbedEditor\TabbedEditor.xml">

  <settings
                      templateRoot="C:\PhoneBook\Remotion\UIGenTemplates\TabbedEditor"
                      targetRoot="C:\PhoneBook"
                      projectNamespaceRoot="PhoneBook.Web"
                      domainNamespaceRoot="PhoneBook.Domain"
                      />

  <projectReplacements>
    <replace from="$ReferencesDir$" to="References" />

    <!-- Domain -->
    <!-- Copy the DomainProjectGuid from your PhoneBook.Domain.csproj file -->
    <replace from="$RemotionResDirectory$" to="C:\PhoneBook\Remotion\res" />
    <replace from="$DomainProjectGuid$" to="{E715FCED-AD7C-4EB7-9E30-EDF67D904A48}" />
    <replace from="$DomainProjectName$" to="PhoneBook.Domain" />
    <replace from="$DomainProjectAssembly$" to="PhoneBook.Domain" />
    <replace from="$RemotionAssembly$" to="C:\PhoneBook\Remotion\net-3.5\bin\Debug" />
    <replace from="$WxeEngine$" to="C:\PhoneBook\Remotion\net-3.5\bin\Debug" />

    <replace from="$WebClientName$" to="PhoneBook.Web" />
    <replace from="$WebClientAssembly$" to="PhoneBook.Web" />
    <replace from="$STRONG_SUPPLEMENT$" to="Version=1.13.6.2, Culture=neutral,
PublicKeyToken=fee00910d6e5f53b" />

    <!-- USER -->
    <replace from="$USER_APPNAME$" to="PhoneBook Web Sample" />
    <replace from="$USER_DEFAULT_ASPX_TOPIC$" to="This application was generated by UIGen.exe"
/>
```

```
    <replace from="$USER_DEFAULT_ASPX_STARTINFO$" to="To start the application just click
'Start'" />
    <replace from="$USER_DEFAULT_STARTPAGE$" to="EditLocation.wxe?WxeReturnToSelf=true" />

    <replace from="$USER_STATUSBAR$" to="yes" />
    <replace from="$USER_CLASSIC_APPSTYLE$" to="false" />

    <replace from="$USER_STORAGEPROVIDER$" to="PhoneBookDB" />
    <replace from="$USER_CATALOGNAME$" to="PhoneBook" />
  </projectReplacements>

</applicationGenerator>
```

Here are two tables on what all that means.

Settings are path- and namespace information:

Setting name	Default value	Substitute if necessary, for...
`applicationGeneratortemplate`	`C:\PhoneBook\Remotion\UIGenTemplates\TabbedEditor\TabbedEditor.xml`	path to the master template file
`templateRoot`	`C:\PhoneBook\Remotion\UIGenTemplates\TabbedEditor`	path to the directory where all the templates are located
`tagetRoot`	`C:\PhoneBook`	your root directory for domain- and web client project
`projectNamespaceRoot`	`PhoneBook.Web`	the *namespace* root for your generated web client project (i.e C#-relevant)
`domainNamespaceRoot`	PhoneBook.Domain	the *namespace* root of your domain library (again, C#-relevant)

projectReplacements are "placeholders" ("macros"). Their values are substituted in the templates:

Placeholder name	Default value	Substitute if necessary, for...
$DomainProjectGuid$	Look it up in the project file	Your domain library project's GUID
$DomainProjectName$	PhoneBook.Domain	name/path of your domain library project (i.e. Windows-relevant)
$RemotionAssembly$	C:\PhoneBook\Remotion\net-3.5\bin\Debug	The directory where your re-motion assemblies are located
$WebClientName$	PhoneBook.Web	The name/path of your (future) web client project (Windows-relevant)
$WebClientAssembly$	PhoneBook.Web	The name of the web client assembly (DLL)
$STRONG_SUPPLEMENT$	Version=1.13.6.2	The extra information required for re-motion assemblies' strong names
$USER_APPNAME$	PhoneBook Web Sample	The title of your web client application
$USER_DEFAULT_ASPX_TOPICS$	This application...	A head-line visible on the web-client's splash screen
$USER_DEFAULT_ASPX_STARTINFO$	To start the application...	Commonly used for a brief help-text
$USER_DEFAULT_STARTPAGE$	EditLocation.wxe...	The relative URL of the page visible upon startup
$USER_STORAGEPROVIDER$	PhoneBookDB	Storage provider to use, as specified in Web.config
$USER_CATALOGNAME$	PhoneBook	name of database to use, as specified in Web.config

Most of these configuration items are self-explaining or don't need to be changed for typical projects. If you are sticking with the recommendations for file- and directory names in this tutorial, the `PhoneBook.xml` file as printed here is good to go, with two exceptions: the placeholder you **should** change (for correctness) is the GUID for `DomainProjectGuid`. You find this GUID in the `PhoneBook.Domain.csproj` file. If you don't change it, then... well, no harm will be done.

Room for confusion is provided by **`projectNamespaceRoot`** and **`domainNamespaceRoot`.** These are the *namespaces* provided in C#-sources, as in "`namespace PhoneBook.Web.UI`", for example. Contrast this with **`$DomainProjectName$`** which is the path snippets used by the file system to locate your project, as in "`PhoneBook.Sample`", for example.

If you were creative and used other names for those than the recommendations in this tutorial, you must take care not confuse those.

The most delicate part is the spec of where the re-motion assemblies can be found – **`$RemotionAssembly$`**. This path must be **E X A C T L Y** the same as the one you used in your *domain library project*, otherwise `uigen.exe` will be confused. `uigen.exe` is extremely sensitive due to the simple reflection method it uses to find its way around the assemblies. *NOT* allowed:

- using identical copies of the assemblies referenced by the domain library project (the used assemblies must not only be identical copies, they must be identical).
- using a physically identical path, but with a different name, e.g. using the full path as references in the domain library project, but a substituted drive letter in the **`$RemotionAssembly$`** placeholder. (Example: I substituted my `C:\Development\Remotion\trunk\build\1.13.6.2` directory for `R:`. For `uigen.exe` `C:\PhoneBook\Remotion\net-3.5\bin\Debug` is not the same as `C:\Development\Remotion\trunk\build\1.13.6.2\net-3.5\bin\Debug`, so those two are ***not*** exchangeable.)

The `$RemotionAssembly$` directory in essence specifies where `uigen.exe` can find the two DLLs

- `Remotion.Web`
- `Remotion.ObjectBinding.Web`
  These are required for your web client project to work.

You can store your `PhoneBook.xml` file wherever you want, because you specify the path to it in a command line parameter, but the natural place for it is right in the `targetRoot`, i.e. the parent-directory of your `PhoneBook.Domain` project directory. The directory with the generated web-client project - `PhoneBook.Web` - will be a peer directory of `PhoneBook.Domain`, located in `PhoneBook`. The recommendation here is: Put `PhoneBook.xml` above the project directory, so that your directory looks like this eventually:

```
15.10.2008  12:24    <DIR>          .
15.10.2008  12:24    <DIR>          ..
15.10.2008  17:16    <DIR>          PhoneBook.Domain
15.10.2008  12:11             1.791 PhoneBook.resharper
15.10.2008  12:11             1.731 PhoneBook.resharper.user
15.10.2008  12:23    <DIR>          PhoneBook.Sample
15.10.2008  12:24             1.933 PhoneBook.sln
15.10.2008  12:49    <DIR>          PhoneBook.Web
```

```
15.10.2008  10:54             1.693 PhoneBook.xml
06.10.2008  12:47             3.907 SetupDB.sql
```

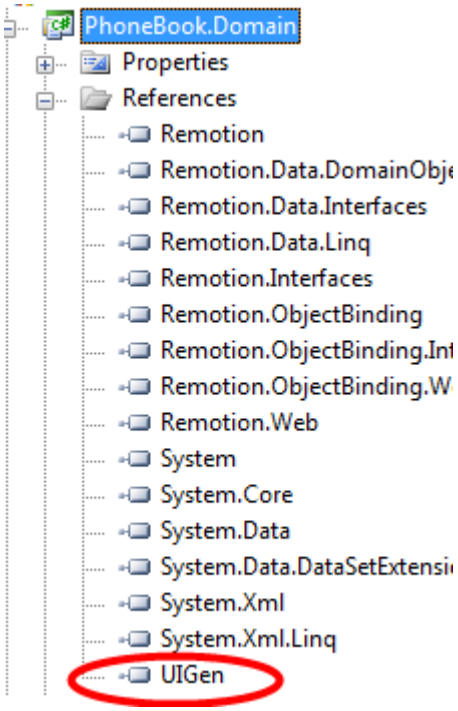
Making sure the DLLs can find each other

The PhoneBook web client project (or application) will need two more re-motion assemblies:

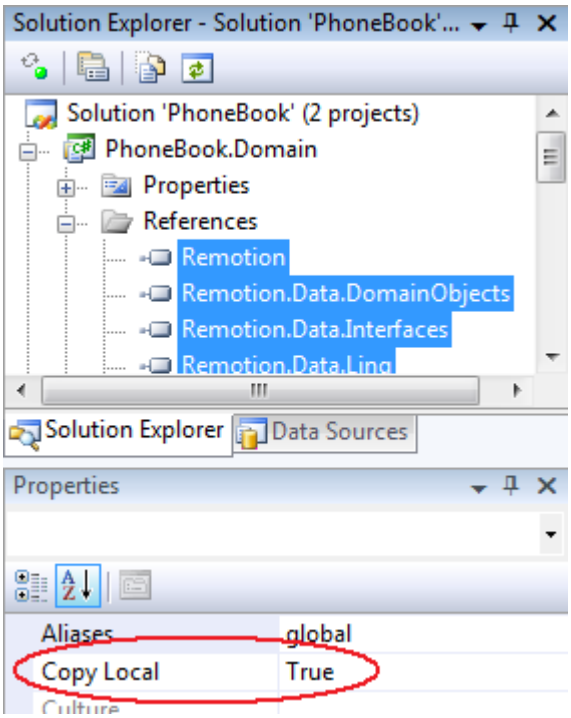
- `Remotion.Web`
- `Remotion.ObjectBinding.Web`

This means that `uigen.exe` needs to know where to find them, but that directory must be identical to the one you used as references in the domain library project. The easiest way to sync up everything is as follows:

- put the `uigen.exe` file into the reference list in your `PhoneBook.Domain` library project



- rebuild your `PhoneBook.Domain` library project – this will pull `uigen.exe` into your `PhoneBook.Domain\bin\Debug` directory -- **given that for the reference the *CopyLocal* property is set to `true` -- as it should be for ALL references**. Here is an picture to clarify this important issue:



- specify the `PhoneBook.Domain\bin\Debug` path as `uigen.exe`'s `/asmdir` parameter
- use the `uigen.exe` copy in `PhoneBook.Domain\bin\Debug` on

If you have those preparations in place, checked and double-checked your `PhoneBook.xml` file, it is time for invoking `uigen.exe` on the command-line. Drum-roll, please:

```
cd c:\PhoneBook
PhoneBook.Domain\bin\debug\uigen.exe /uigen:phonebook.xml
/asmdir:phonebook.domain\bin\debug
```

This takes only a few seconds (on a 3.0 GHz 64-bit computer, that is) and should not give you any output. If you do a `dir` in the `PhoneBook` directory, you should find a new folder `PhoneBook.Web`. Congratulations if `uigen.exe` worked out at this point. You can proceed right to the checkout and continue with the next section. If something did go wrong, you might find the next section on trouble-shooting useful. One error is to be expected: "Missing names", as explained in the next section:

Non-existent names

At the time of this writing, there is a problem with the `wxegen.exe` program generator. If you build the application, you might get these errors:

```
C:\PhoneBook\PhoneBook.Web\UI\SearchResultPhoneNumberForm.aspx.cs(24,13):
error CS0103: The name 'query' does not exist in the current context
C:\PhoneBook\PhoneBook.Web\UI\SearchResultPhoneNumberForm.aspx.cs(25,11):
error CS0103: The name 'query' does not exist in the current context
C:\PhoneBook\PhoneBook.Web\UI\SearchResultPhoneNumberForm.aspx.cs(26,9):
error CS0103: The name 'searchResult' does not exist in the current context
C:\PhoneBook\PhoneBook.Web\UI\SearchResultPhoneNumberForm.aspx.cs(26,78):
error CS0103: The name 'query' does not exist in the current context
C:\PhoneBook\PhoneBook.Web\UI\SearchResultPhoneNumberForm.aspx.cs(28,41):
error CS0103: The name 'searchResult' does not exist in the current context
C:\PhoneBook\PhoneBook.Web\UI\SearchResultPhoneNumberForm.aspx.cs(37,31):
error CS0117: 'PhoneBook.Web.UI.EditPhoneNumberForm' does not contain a
definition for 'Call
```

```
... more of those ...
```
This is a known problem with Visual Studio's build mechanism. These errors should go away if you build the application again. Just ignore this error and rebuild your application.

Here is a round-up of what you should take care of:

- `PhoneBook.xml` in the `PhoneBook` directory, above the project directories
- `uigen.exe` in the `PhoneBook.Domain's` *References* list in Visual Studio
- all assemblies in the *References* list must have the `CopyLocal` property set to `true`
- build the `PhoneBook.Domain` project before running `uigen.exe`, so that `uigen.exe` and all the assemblies it needs are copied the `PhoneBook.Domain\bin\Debug` directory
- run the `uigen.exe` in the `PhoneBook.Domain\bin\Debug` directory
- After running `uigen.exe` and including the newly created PhoneBook.Web project into your PhoneBook solution, rebuild your solution
- Make sure Visual Studio can resolve all the references in the *References* list in the new `PhoneBook.Web` project
- Ignore the missing names error and rebuild the `PhoneBook.Web` application

What can go wrong

In a word: a lot. However, most mistakes you can make don't disclose themselves before actually running the application or looking at the generated project in Visual Studio. Here is a quick guide to the most common `uigen.exe` error messages and what to do about them.

Failure to remove a stale `PhoneBook.Web` project

Message:

```
UIGen error: OBLXE0202: FileAlreadyExists:
C:\PhoneBook\PhoneBook.Web\PhoneBook.Web.csproj
```
This happens if you try to generate your PhoneBook web client application again without removing the existing `PhoneBook.Web` directory first. In this regard, `uigen.exe` is your friend: overwriting your existing project, potentially with weeks or months of invested work by accident is precluded by this simple check. This is admittedly annoying and boring while learning the ropes with `uigen.exe`, but this feature is your friend.

No DLLs (or no files at all) in asmdir

Message:

```
Execution aborted: Argument rootAssemblies is empty.
```
This happens when there are domain library assemblies in the directory specified by the `/asmdir` parameter. The most likely cause (at least in the practice of the author) is to *Clean* the build in Visual Studio and forget to rebuild the domain library project.

Using a `uigen.exe` in a directory different from the domain assembly directory

Message:

```
Execution aborted: Could not load file or assembly
'PhoneBook.Domain, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null'
or one of its dependencies. The system cannot find the file specified.
```

This happens if you forget the advice from the previous section and invoke an `uigen.exe` that can't find `PhoneBook.Domain.dll` in the same directory, as in (for example):

```
C:\PhoneBook\Remotion\net-3.5\bin\debug\uigen.exe /uigen:phonebook.xml
/asmdir:phonebook.domain\bin\debug
```

This can also happen if you migrate to another "instance" (copy) of your project for testing or other experiments.

Domain objects not derived from `BindableDomainObject`

Message:

```
Execution aborted: The type 'PhoneBook.Domain.Location' does not have the
'Remotion.ObjectBinding.BusinessObjectProviderAttribute' applied.
Parameter name: type
```

This message is probably misleading, because the most likely cause is that you forgot to derive your domain objects from `BindableDomainObject` `DomainObject` (as introduced and suggested in the section *re-bind and BOC controls*).

No `PhoneBook.Domain.DLL` in the `/asmdir` directory

Message:

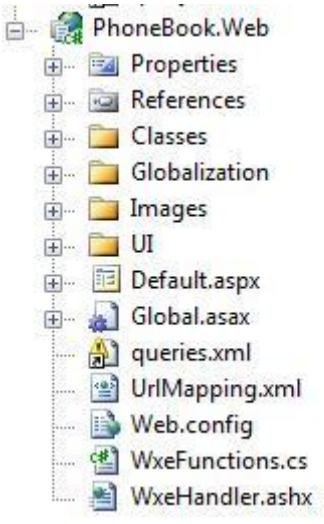
```
Execution aborted: Index was outside the bounds of the array.
```

Look what you have done

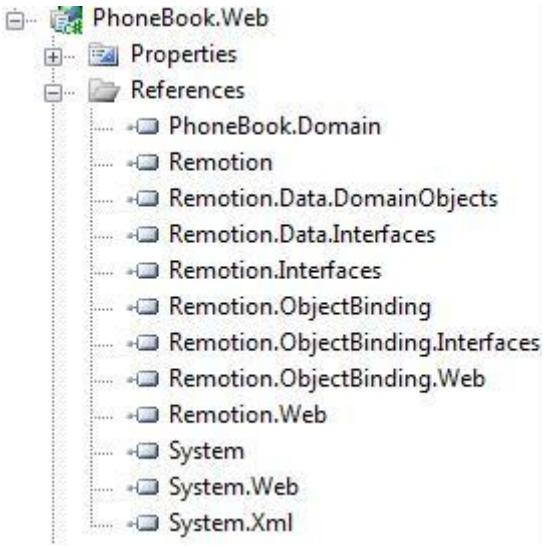
In this section we take a look at the generated web client project and discuss the expected outcome. You find the generated project in `PhoneBook.Web as PhoneBook.Web.csproj`.

As soon as your web-client project has been integrated, you should check that everything has turned out right.

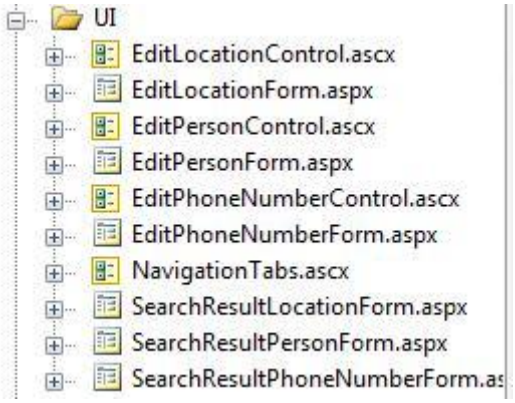
The tree view on your `PhoneBook.Web` project should look like this:



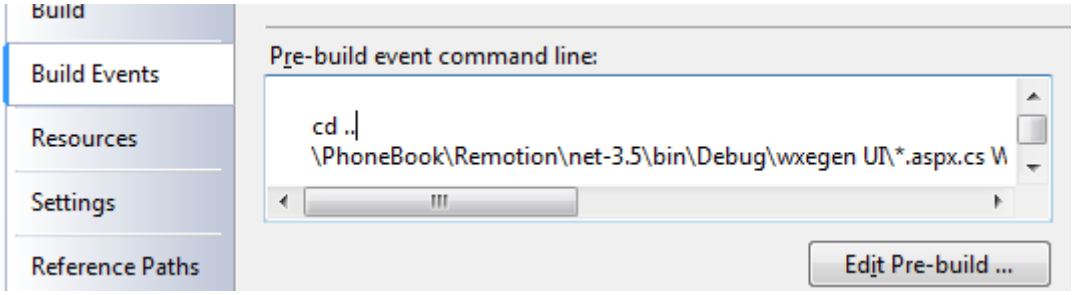
Of particular importance are the generated references. Make sure that Visual Studio can resolve them:



The most interesting part is the set of generated ASP.NET files. You find them under the UI node. These files implement the tabs ("TabbedEditor"), as discussed in the beginning of this section.



As you can see in the `PhoneBook.Web's` properties, `UIgen.exe` has sneaked an invocation of the `wxegen.exe` code generator into the project's pre-build event hook (to be seen in the PhoneBook.Web's project properites):



This invocation causes `wxegen.exe` to generate the file `WxeFunctions.cs`, already part of the project. If you open `WxeFunctions.cs` you will see that the file is virtually empty, because `wxegen.exe` has not run yet.

Build and run your web-client application

Your solution now includes three projects:

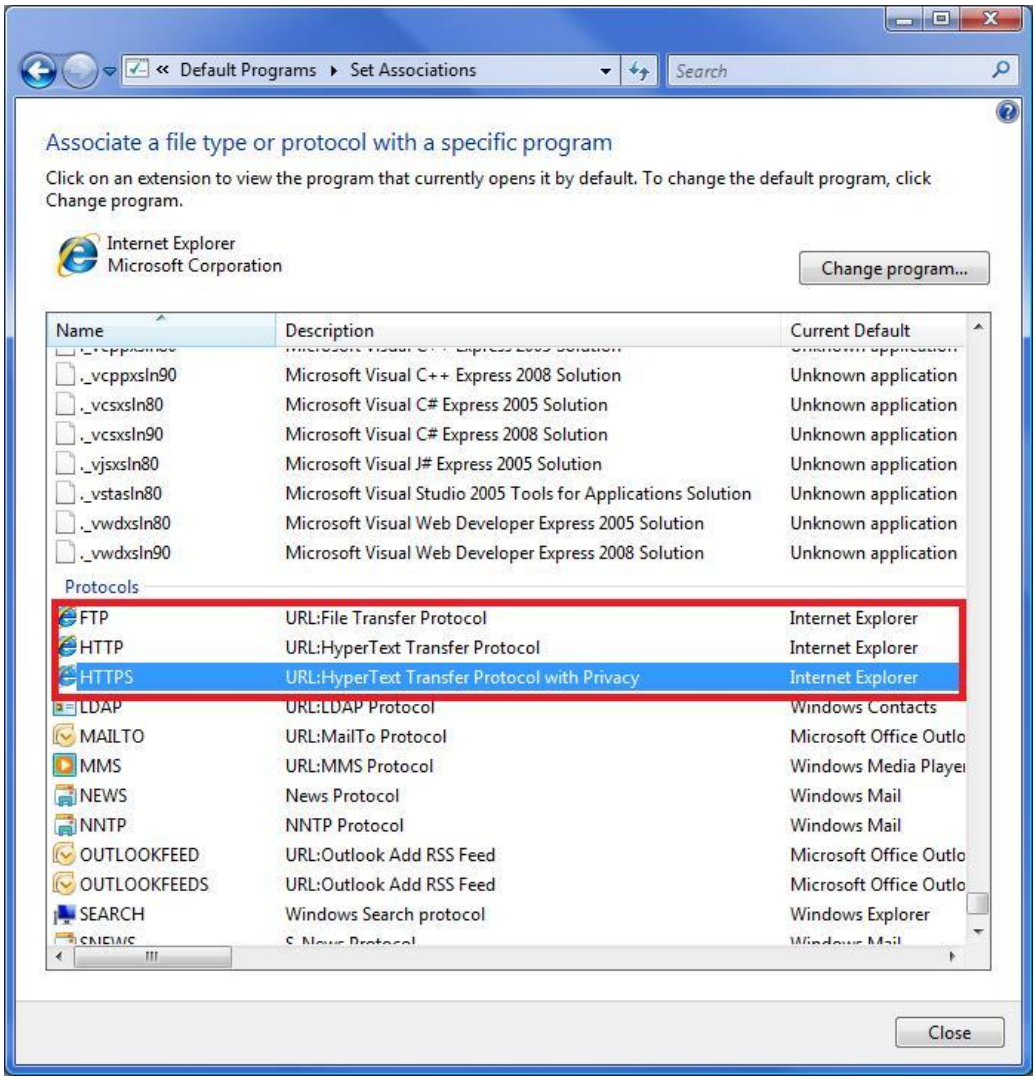
- The `PhoneBook.Domain` library
- The `PhoneBook.Sample` console application
- The `PhoneBook.Web` web-client application

The `PhoneBook.Sample` project still is your startup project, probably, so change this and make the `PhoneBook.Web` application the startup project. The project should build and run. Running means: The ASP.NET web-server is started automatically, and so is your web-application. A random port for http communication is picked at compile time – this means it will change for every new build of your application.

Attention Non-IE drivers: *even if Internet Explorer is not your preferred browser, it is a good idea to make it your default browser for http now. Most things WILL work in Firefox as well, but*

- *Re-motion controls only support IE at this time*
- *Some rendering will look funny in other browsers*
- *Internet Explorer and ASP.NET have been made for each other*

Internet Explorer gives you more support for development of ASP.NET applications. Good news for Firefox-drivers: You may keep Firefox as your default browser for .html and .htm files. In your "Default Applications" control panel, associate the application for http and https in the "Protocols" sections:



What can go wrong

"No current WxeHandler found"

Message (on an HTML-page):

```
Most likely cause of the exception: The page '_<some page name here>_'
has been called directly instead of using a WXE Handler to invoke the
associated WXE Function.
```

The error message says it all: in re-motion you can't call `.aspx` pages directly. You must tell the WXE handler which file you want to fetch in the URL.
Instead of:

```
http://localhost:22809/EditPersonForm.aspx
```

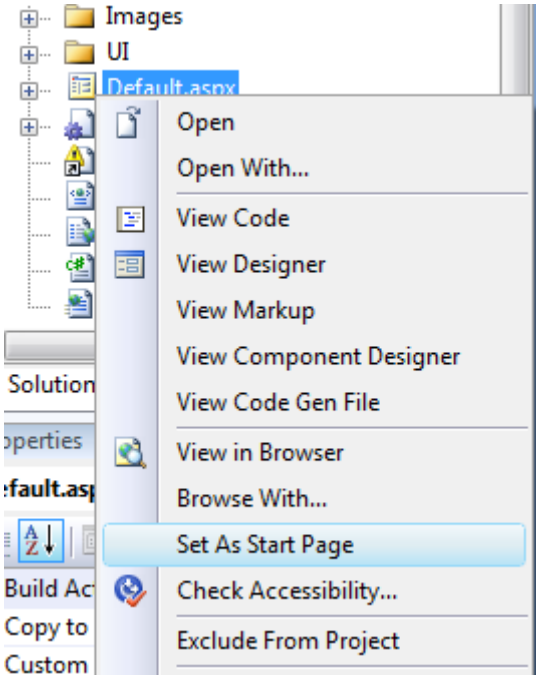
the correct URL is more something like this:

```
http://localhost:22809/EditPerson.wxe?WxeReturnToSelf=True&TabbedMenuSelection=PersonTab%2cEditPersonTab
```

What the error message doesn't tell you is what to do about it. The likely cause is a convenience feature of Visual Studio: If you have an `.aspx` as the top-most document in Visual Studio and start the application directly from Visual Studio, then your Internet Explorer will automatically try to fetch

that `.aspx` page upon startup. This requires Visual Studio to construct the URL and transport it to Internet Explorer. Since neither Visual Studio nor Internet Explorer know anything about the WXE handler, this well-meant scheme breaks down and results in the error.

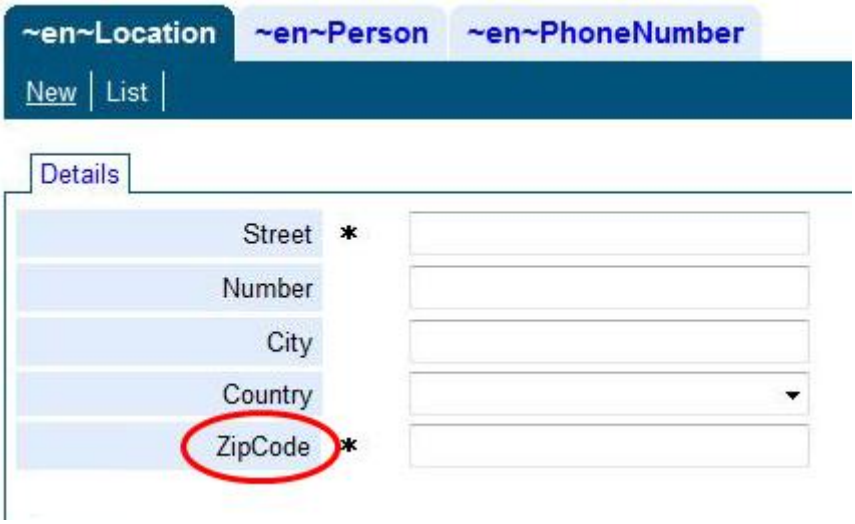
You can fix this once and for good by making your web client project's `Default.aspx` the start page. Right-click on `Default.aspx` and select "Set as start page":



A look at the running application

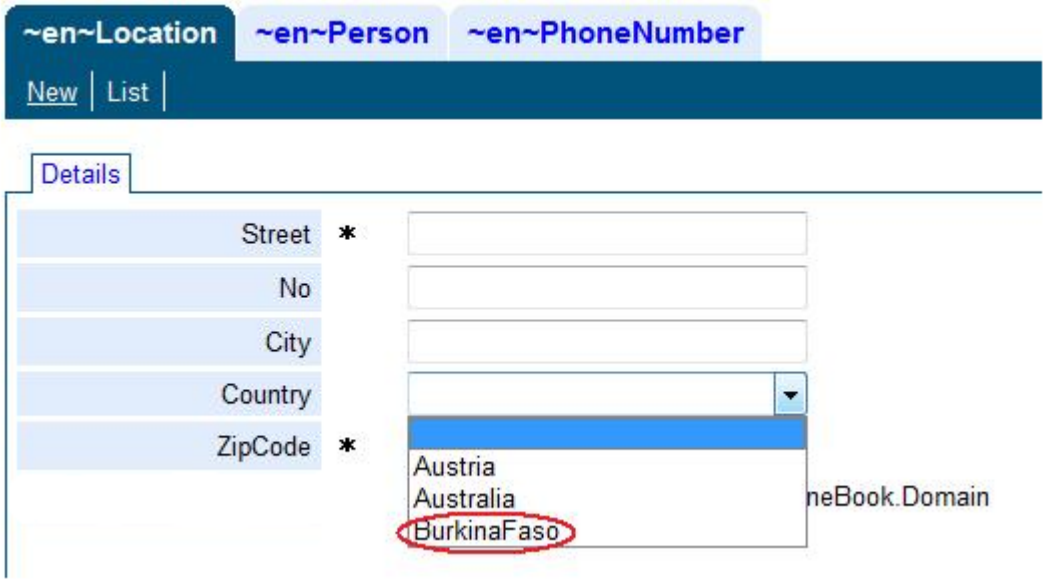
The first page you will see after starting your application is the "splash screen" mentioned in section *PhoneBook web app impressions:*

Clicking on the *Start* link will get you to the edit form for a `Location` object:



(The link behind the "Start" is configured in `PhoneBook.xml`, placeholder `$USER_DEFAULT_STARTPAGE$`.)

Note that raw C# identifiers are displayed as property names, because that's all the application has at this time (the camel-cased "ZipCode" reveals it). C# identifiers can also be seen in the drop box for *Country*. Here the camel-cased "BurkinaFaso" reveals it:



You can try for yourself that the generated web client application works as advertised in section *PhoneBook web app impressions* – list view, edit-link, etc.

Sorting columns

The lists (click on a List)already show sortable columns by default. You can try this out by clicking on the links.

- by default, columns don't sort their values
- the first click on a column heading makes it sort ascending
- the second click on a column heading makes it sort descending

- the third click on a column heading turns off sorting
- the order you turn on sorting is the order in which columns get their sorting priority

FirstName	Surname	Location ▲	PhoneNumbers
Sigmund	Freud	Berggasse	+43 (1) 3191596/
Sisi	Eugenie	Schönbrunner Schloßstraße	+43 (1) 555-0002/ ... [2]
Franz-Josef	Habsburg	Schönbrunner Schloßstraße	+43 (1) 555-0001/

Deploying a re-motion web application

Since the PhoneBook web application is still under construction, it is probably too early to deploy it on the Microsoft Internet Information Server. If you want to try it anyway, be aware of two special measures you must take:

- you must specify a handler for the re-motion's web page's .wxe extension
- you must set the `validateIntegratedModeConfiguration` attribute to `"false"`

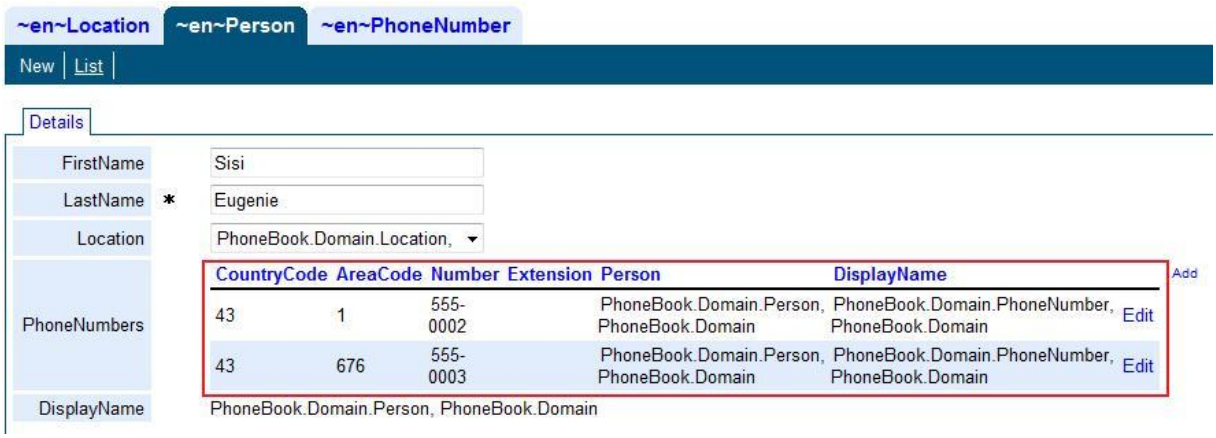
Here it is in one handy snippet for your **`Web.config`**:

```
<system.webServer>
  <validation validateIntegratedModeConfiguration="false" />
      <handlers>
          <add name="re-call"
               path="*.wxe" verb="*"
               type="Remotion.Web.ExecutionEngine.WxeHandler"
               resourceType="Unspecified"
               preCondition="integratedMode" />
      </handlers>
</system.webServer>
```

Rendering `ObjectLists` and reference properties

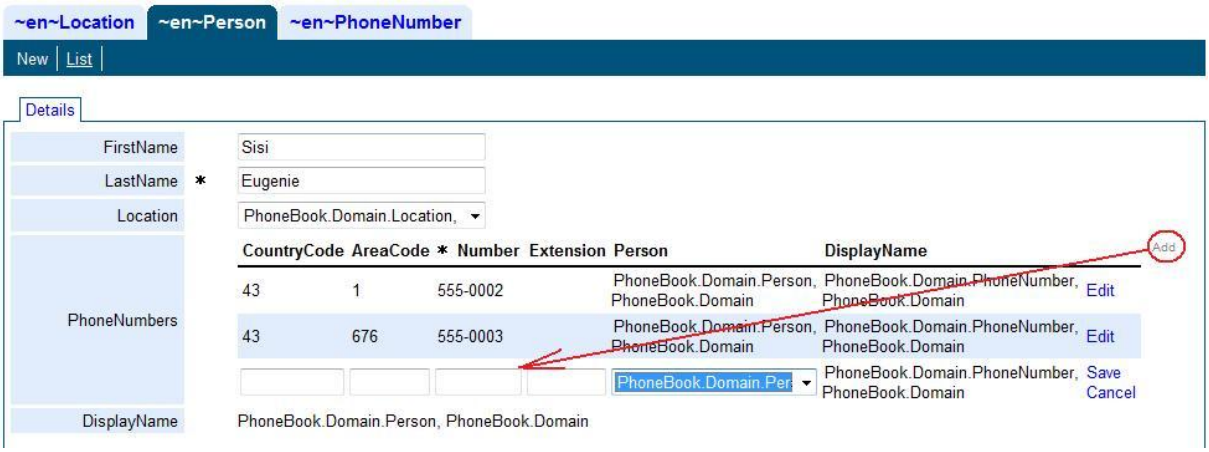
ObjectList

A `Person` instance contains an object list (`ObjectList<T>`) for all `PhoneNumber` instances that person has. Such object lists are rendered as BOCLists:



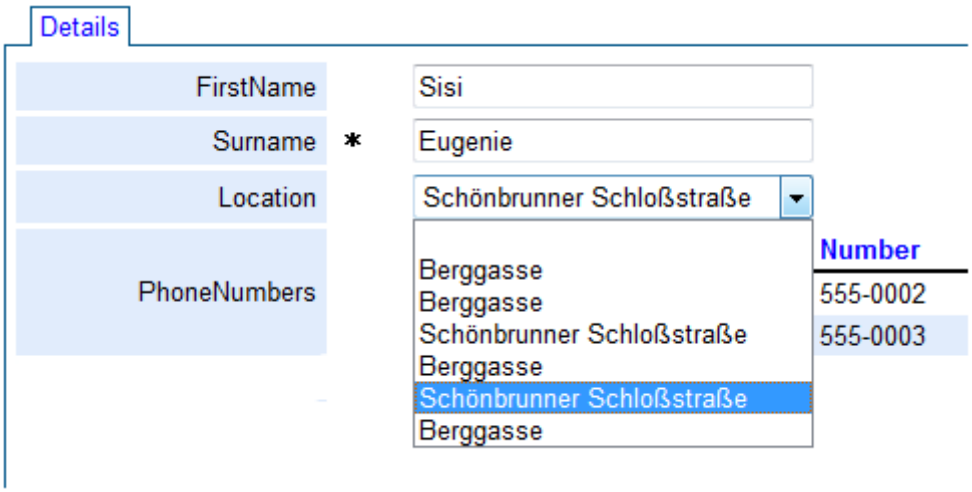
This "sub-list" (for "in-place editing") within an object has sorting columns as well.

What's more, you can treat Sisi with another phone-number (an iPhone, perhaps?) by clicking on the `PhoneNumber` property's *Add* button, what adds some sort of an empty mini-edit-form:



Reference properties

By default, a reference property is rendered as a simple drop list from which you can choose a location for Sisi (for example). At this time your options are somewhat limited, since we have only two locations in the database. Since both are displayed as the same identifier you don't even know which is which, but here it is:

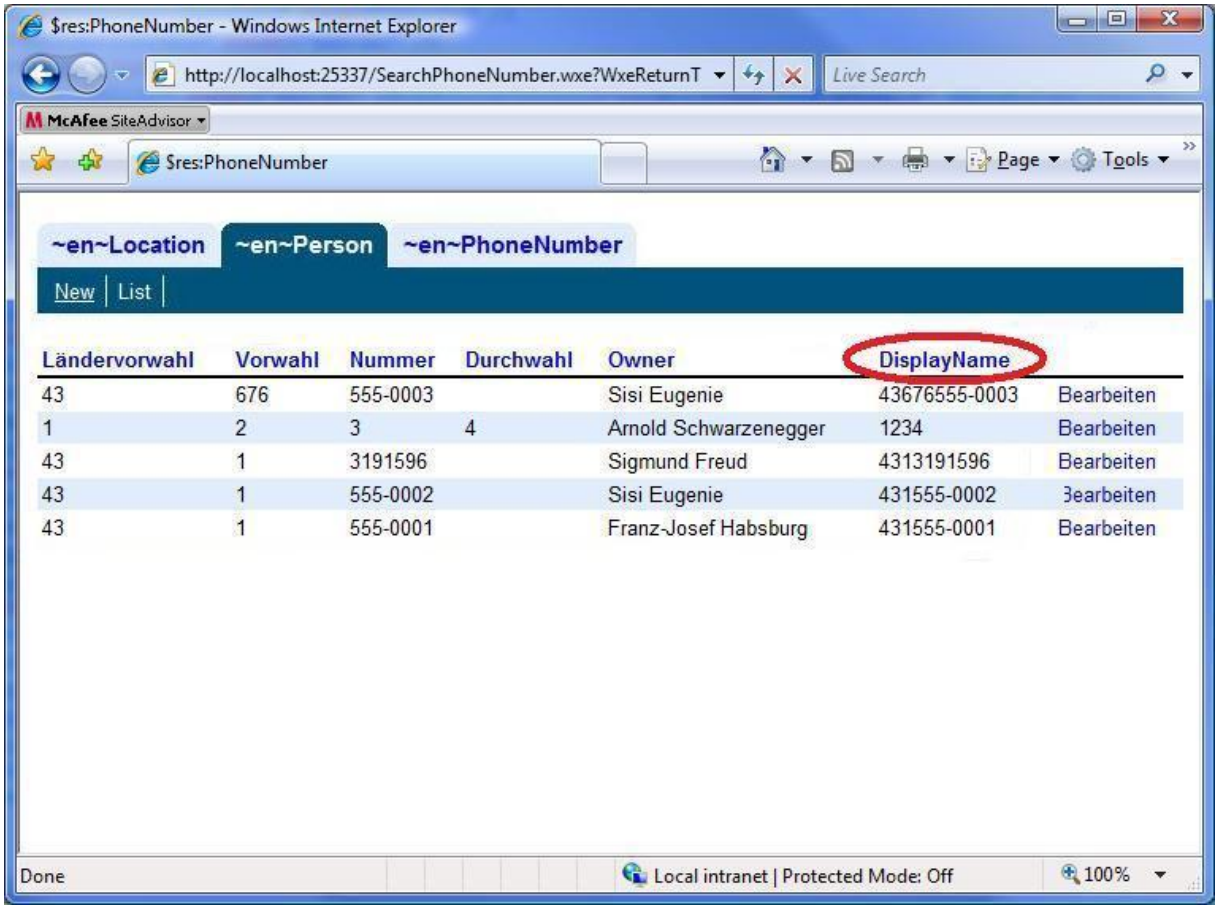


Where this tutorial is going

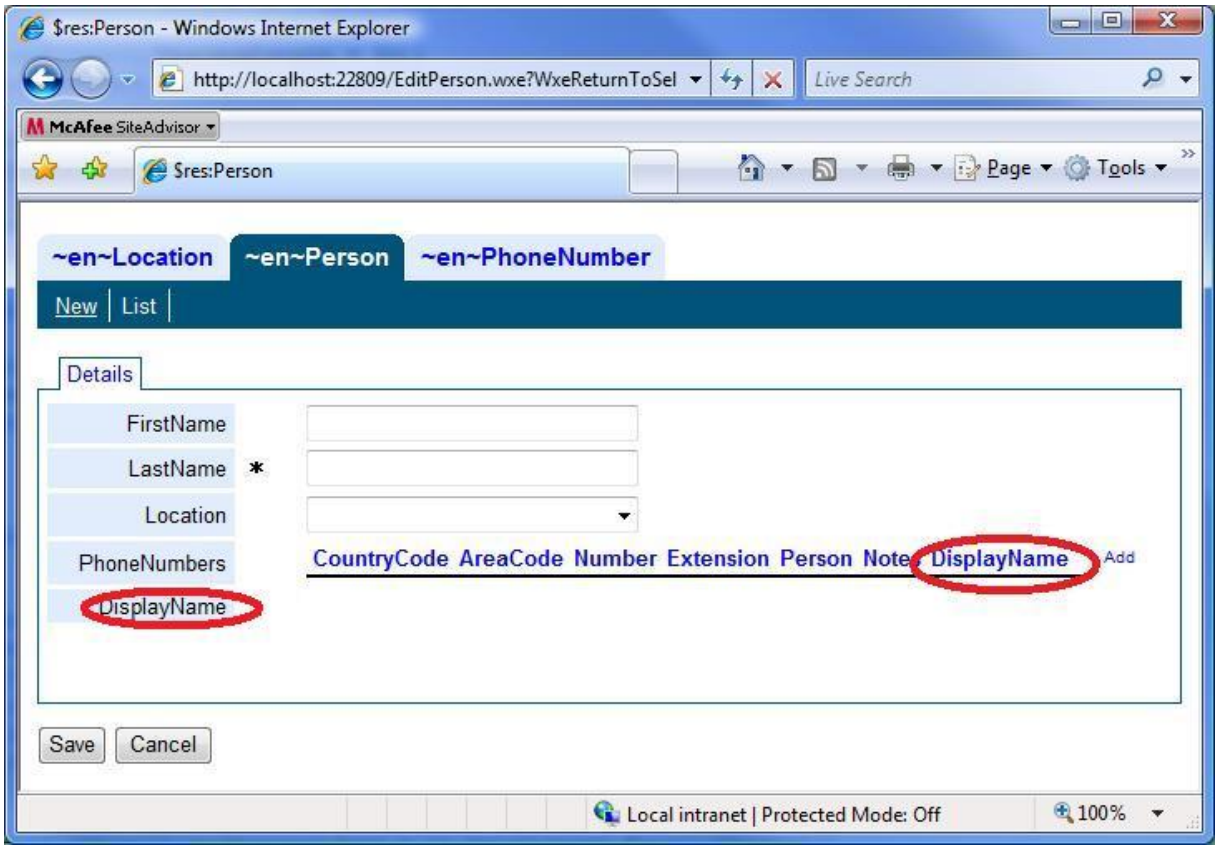
In this section we discuss how the PhoneBook web application will look like after the generated default app got our elaborate make-over. To see the "finished" application, look no further than `http://re-motion.org/PhoneBook`. For brevity, we will assume that Stefan - the author's boss at rubicon - has provided a finished version of the PhoneBook application which we contrast and compare with our state of affairs. Thus the finished application will be called "Stefan's app" in the following, and our application will be referred to as, uhm, "our app".

Getting rid of `DisplayName`

Stefan's app does not show a column `DisplayName` in all BOCLists. Check the *SearchResultPhoneNumberForm*, for example:

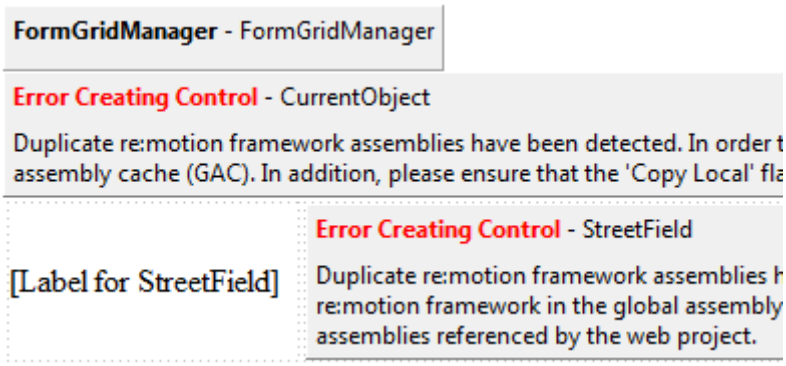


Another example is the BOCList for the PhoneNumbers in the Person edit form:



If you read about `DisplayName` in section *BindableDomainObject's DisplayName* you can probably explain what is happening in our unimproved web-application. Obviously `uigen.exe`, in its robotic stupor, generates a column in the BOCList for each and every property. And since `DisplayName` is a property, `uigen.exe` just does us one too good here. What is the user supposed to think about a property which purpose is completely mysterious? It is clear that we don't wish to display a property ironically named `DisplayName` in this fashion.

How to fix the `DisplayNames` on these pages in both Visual Studio designer and in XML will be explained in the section *Getting rid of `DisplayNames` (introducing column definitions)*, plus a backgrounder on re-bind's BOC (business object controls). You can fix this excess column in Visual Studio's designer or by modifying the page's ASP.NET markup. We will go the VS Designer route first. However, you might feel cheated by Visual Studio's designer, because what it will show you is this:



The problem is that Visual Studio's Designer needs the assemblies for re-motion's BOC assemblies - `Remotion.Web` and `Remotion.ObjectBinding.Web` - in its global assembly cache. By implication, you should put all re-motion assemblies and a few more into the global assembly cache.

The global assembly cache is a bucket where assemblies can be dropped into for .NET to find. If an assembly is referenced by an application, .NET looks into this globally accessible bucket (after scouring the usual suspects: current directory, path, etc.)

The global assembly cache has the advantage that assemblies with identical names but different version can be stored in this bucket, what is unlike a directory that can't store multiple versions of a DLL with identical file names. (The scripting way to *install* DLL in the GAC is to use an utility named `gacutil.exe`.

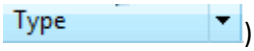
Explorer makes the GAC behave like a regular Windows folder located at `\Windows\assembly`.

So in order to get rid of the designer errors and proceed with this primer copy all .DLL files from the re-motion binaries directory into the `C:\windows\assembly` directory. This includes the non-re-motion DLLs:

- `AjaxControlToolkit.dll`
- `Castle.Core.dll`
- `Castle.DynamicProxy.dll`
- `log4net.dll`

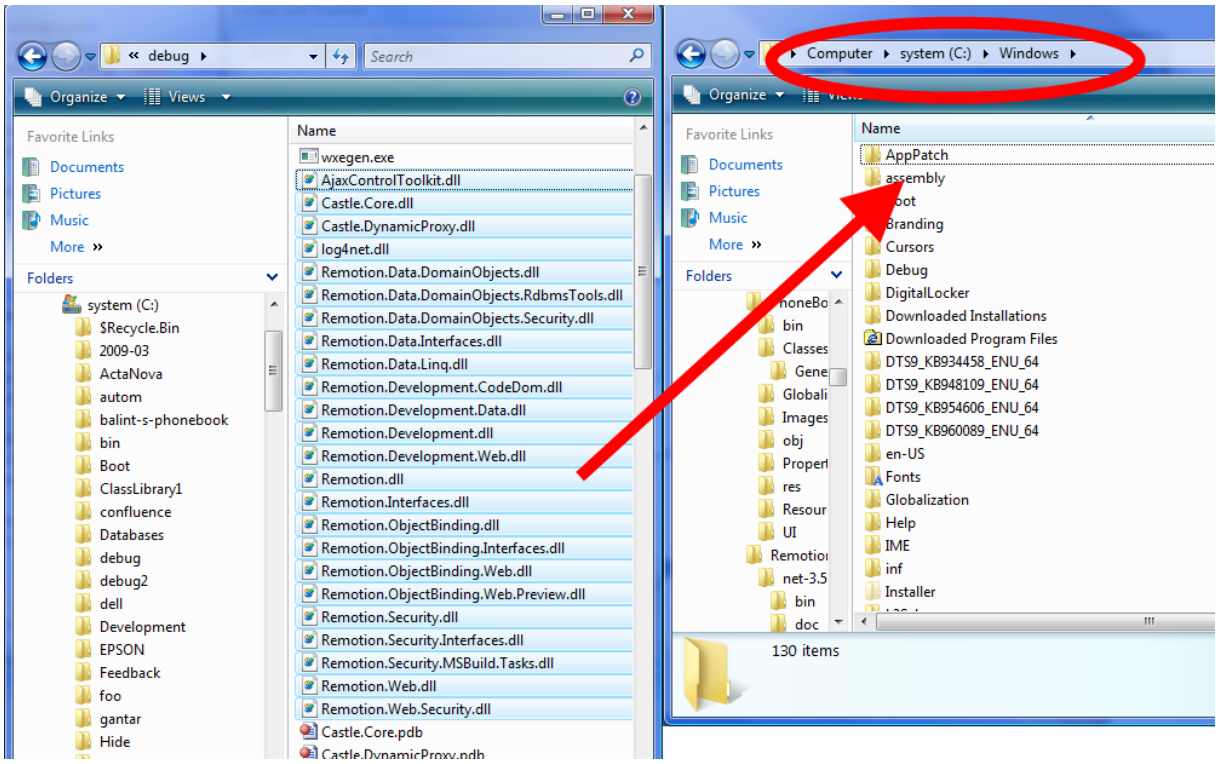
The point-and-click deployment of the re-motion assemblies (and the ones above) works like copying regular files:

In your C:\PhoneBook\Remotion\net-3.5\bin\Debug folder:

- Order the files by type, so that all .DLLs are listed together (

)
- Select those .DLLs
- drag them into the C:\Windows\assembly folder

Here is an illustration



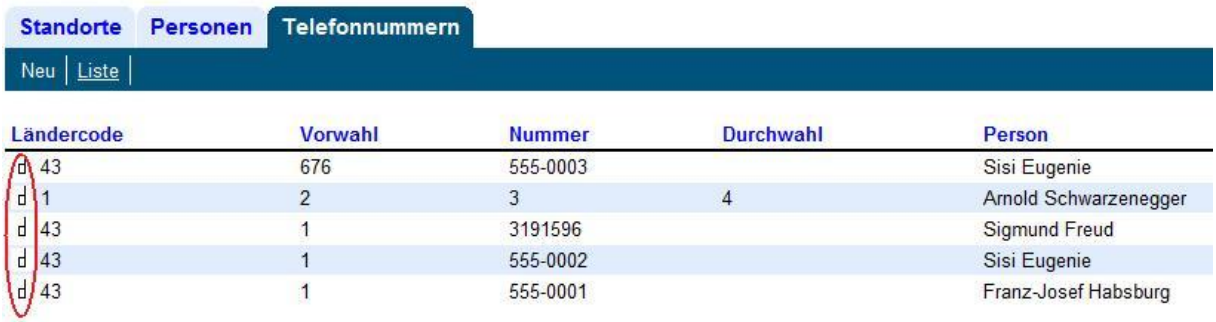
Another wart that Stefan's PhoneBook application has been rid of is the redundant `Person` column in the Person's sub-list, pointing back to the visible parent form. In the `Person`'s edit form, this column, too, has to go, because screen real-estate is is scarce.

German localization

As pointed out before, Stefan's PhoneBook application sports German labels and headings. We will see how such localizations (also called "globalization") work in section *Globalizing the PhoneBook web application*.

Icons for domain objects

An improvement that looks like a minor improvement but can be useful in complex applications with many types of domain objects is icons. On Stefan's List tabs, we find smallish icons for each domain object instance. Since the List tabs show only one type of domain object, this is a little boring. Look at Stefan's "phone-number list":



Note the small icon on the left! It's a mobile phone! (Stefan can program much better than draw.)

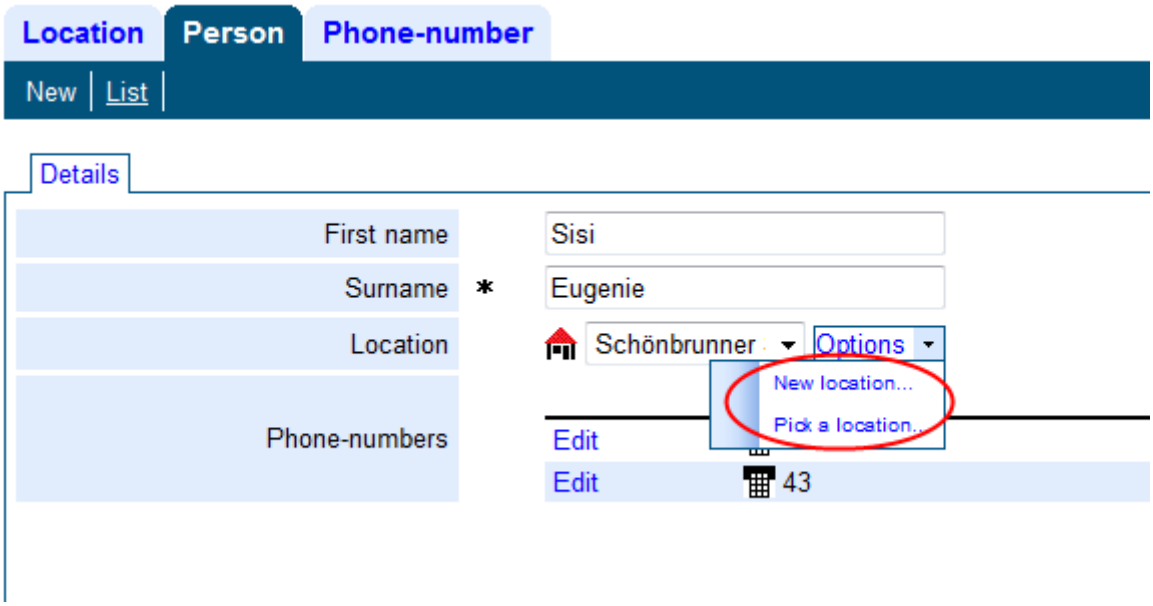
We will add (more beautiful) icons to our phone-book application in section *Creative break -- adding icons*.

Options menu, reference property

Reference properties can provide an *options menu* for creating or manipulating referenced domain objects. An example for this is the `Location` property in Stefan's version of the PhoneBook web app. He has added two option menu items:

New location ("Neuer Standort" in the illustration below)

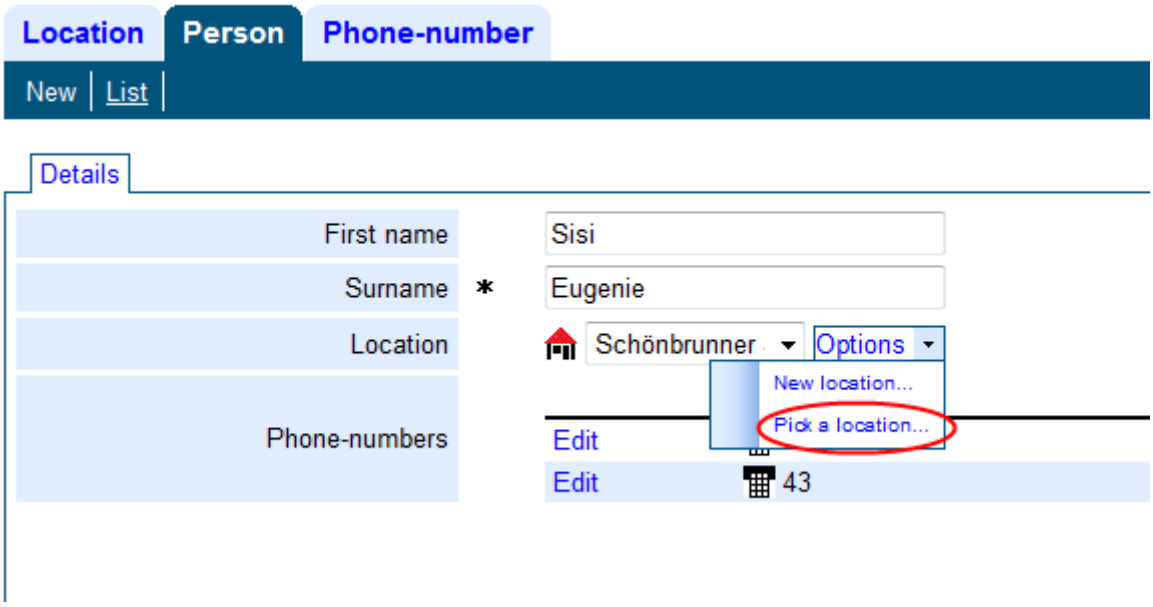
Search for location ("Standort suchen" in the illustration below)



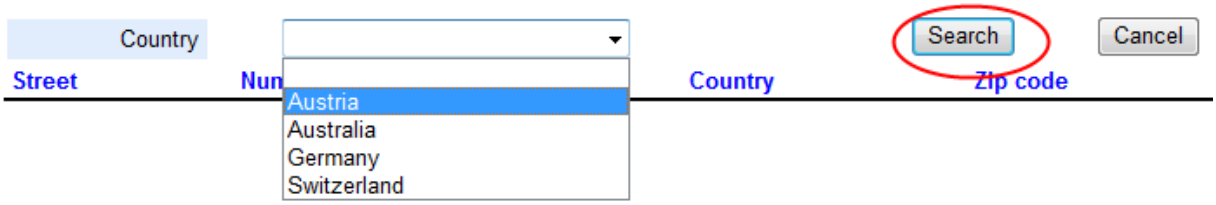
The menu item for "New location..." provides some convenience. It comes in handy when editing a person who lives at a place not in the database yet. If the user selects this menu item, the application gives him an empty edit location form for entering a new location. After clicking "Save" the application returns to the original edit person form, with the newly minted `Location` object in the property. And it makes for a great exercise!

What if the desired `Location` object already exists? Since selecting `Location` objects from a drop list of potentially thousands or (or even a few dozens) is only so much fun, Stefan has added an "options list" to the `Location` property for conjuring up a search result form for browsing locations in a `BocList`. Selecting "Pick a location..." gives you a page where you can filter `Location` s by country.

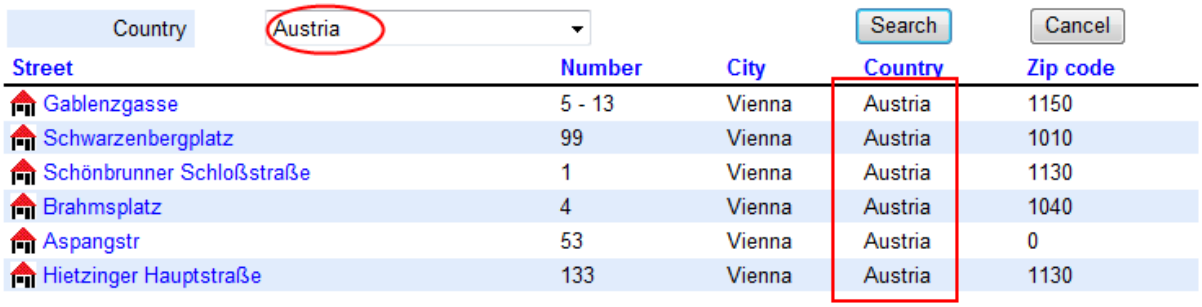
If you want to search for a location for a person, you can do that by selecting "pick location" from the "options menu" (person page).



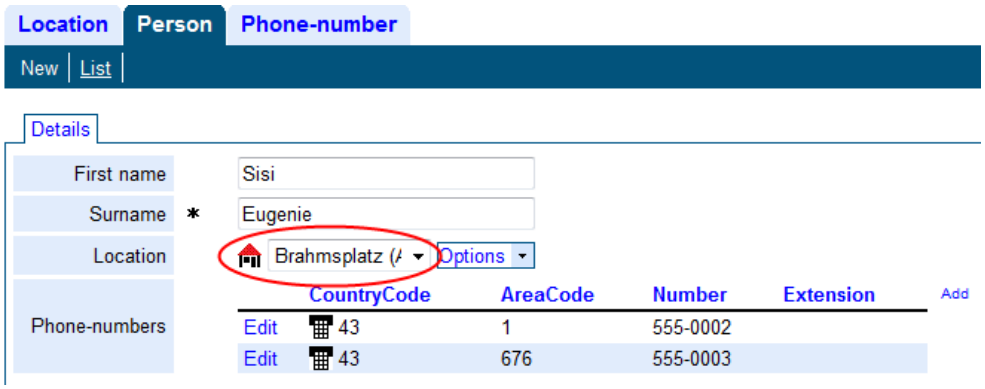
You'll get a dialog for filtering locations by country (filter page).



After doing so you'll get a list of locations in that country and can click on a street name to pick it (picking page).



The picked location shows up in the drop-down list for that person (back to the person page):



You will learn how to create and that options menu in a later section. It will also give you an impression of the power of re-call, the web execution engine.

Phone-numbers as hyperlinks

Often it is practical to have a direct link from a list of domain objects to another referenced domain object. An example is the `PhoneNumber` object in a list of `Persons`. Stefan's advanced application sports hyperlinked `PhoneNumber` s, as seen in the following screenshot:

As you might guess, clicking on one of the `PhoneNumber` links opens an edit form page for that phone-number. So much for the plan, let's turn to execution.

Embellishing the PhoneBook web application

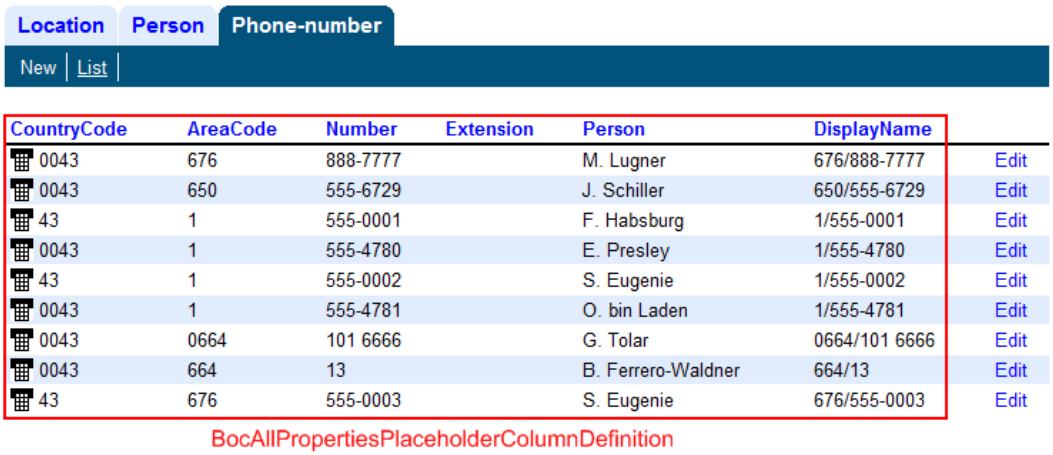
Getting rid of `DisplayNames` (introducing column definitions)

As you've seen in *Getting rid of* `DisplayName`, it is easy to remove the confusing `DisplayName` property from edit forms with designer. Removing them from search result forms is not so easy, for the following reason.

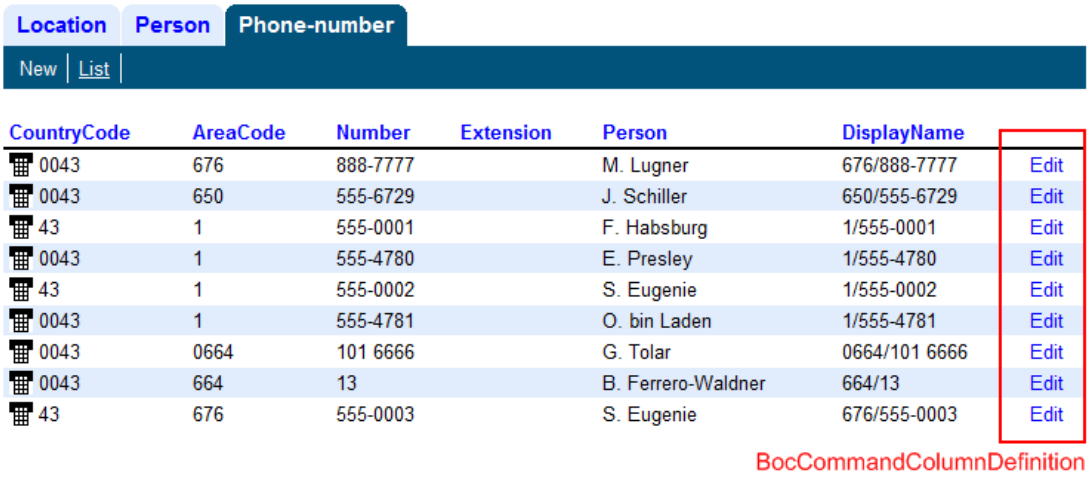
A *column* is a control in a so-called *BOCList*, i.e. a grid containing columns which in turn contain *fields*. Various types of columns exist for different purposes.

By default (i.e. what `uigen.exe` generates) a `SearchResultXForm` contains only two controls:

- `<remotion:BocAllPropertiesPlaceholderColumnDefinition />`, some sort of übercontrol displaying all properties in a domain object as columns. This explains both the name and problem, because "all properties" includes the unsightly `DisplayName`. The following screen-shot illustrates the concept; the red box includes all properties:



- A single `<remotion:BocCommandColumnDefinition>` is used to display the extra column with the Edit links, as the red box in the following screen-shot illustrates:



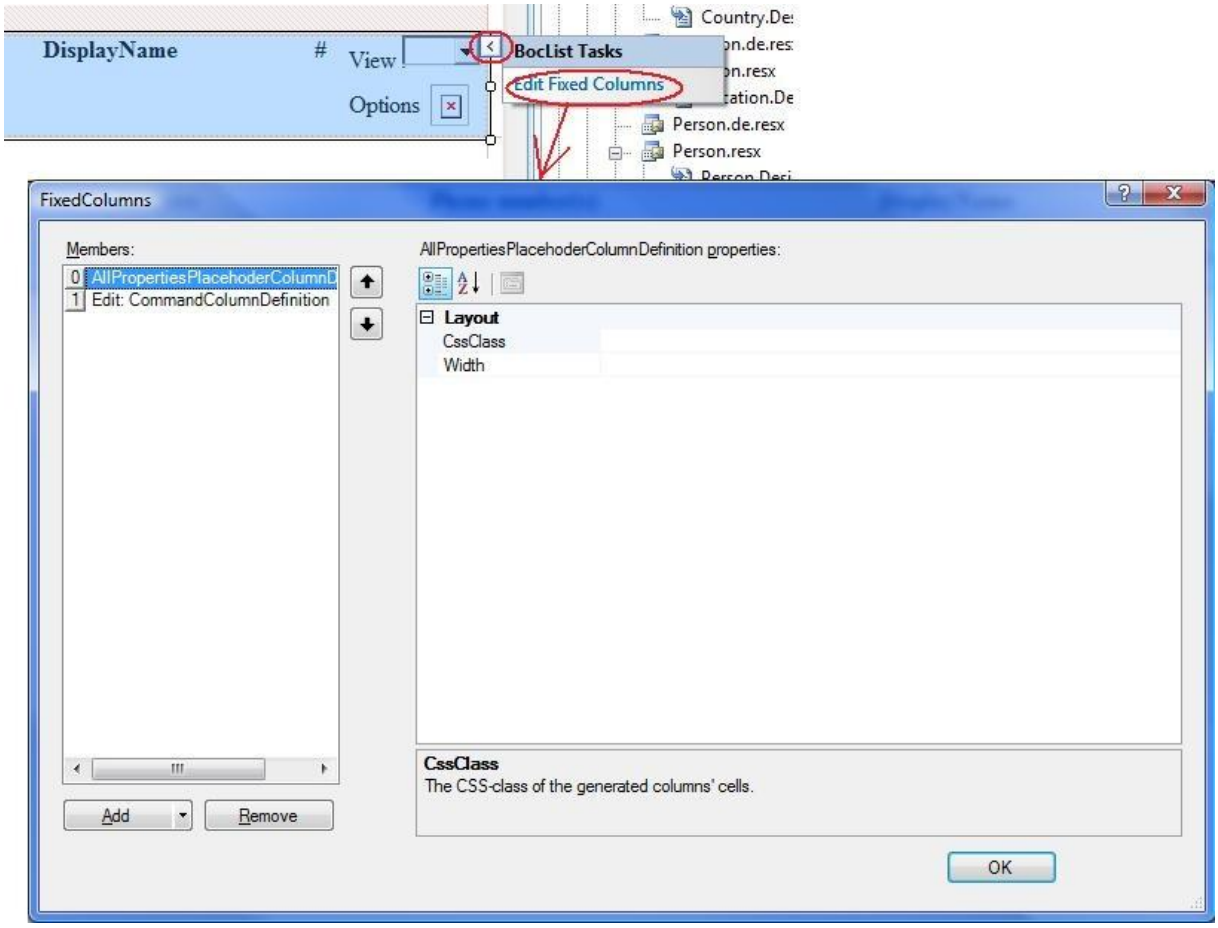
If you look at the other two search result forms (`Location`, `Person`, you will see that the SearchResultXForms for all three of our PhoneBook domain objects have the same `DisplayName`-problem.

If you look at the ASP.NET source of the search result form for the `PhoneNumber`s, you can see `<remotion:BocAllPropertiesPlaceholderColumnDefinition />` and `<remotion:BocCommandColumnDefinition>` in a natural habitat:

```
<form id="Form1" method="post" runat="server">
  <remotion:BocList ID="PhoneNumberList" runat="server" DataSourceControl=
    <FixedColumns>
      <remotion:BocAllPropertiesPlaceholderColumnDefinition />
      <remotion:BocCommandColumnDefinition ItemID="Edit" Text="$res:Edit">
        <PersistedCommand>
          <remotion:BocListItemCommand Type="Event" />
        </PersistedCommand>
      </remotion:BocCommandColumnDefinition>
    </FixedColumns>
  </remotion:BocList>
</form>
```

As you see, these controls fit into ASP.NET-pages just as any ASP.NET-programmer would expect them to. An alternative (and less geeky) view on BOC controls is through re-motion's *FixedColumns editor*, an extension to Visual Studio's designer.

Both `Boc...`-tags have their corresponding entries in the fixed columns' UI. You can see them if you open the FixedColumns editor in Visual Studio's designer:



In order to get rid of the `DisplayName` column, we have to replace the single one-size-fits-all `BocAllPropertiesPlaceholderColumnDefinition` control with a single column control for each property, leaving out only the unwelcome `DisplayName` column. You can hack column definition on one of two ways:

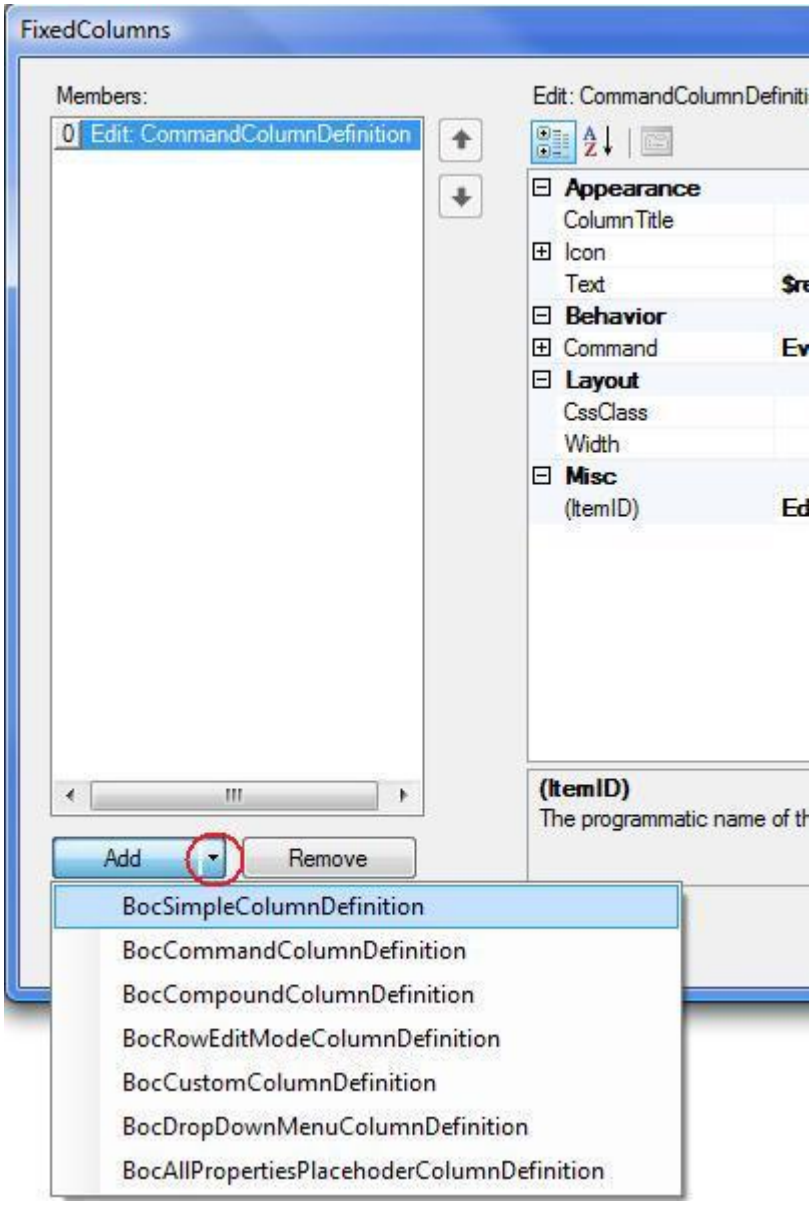
- by directly hacking the ASP.NET HTML code
- by using the fixed column UI in designer

We will do both here. We will remove the `DisplayName` in `SearchResultPersonForm.ascx` with the FixedColumns editor. We will remove the `DisplayName` in `SearchResultPhoneNumberForm.ascx` by modifying the source code directly.

Editing fixed columns in Visual Studio's designer

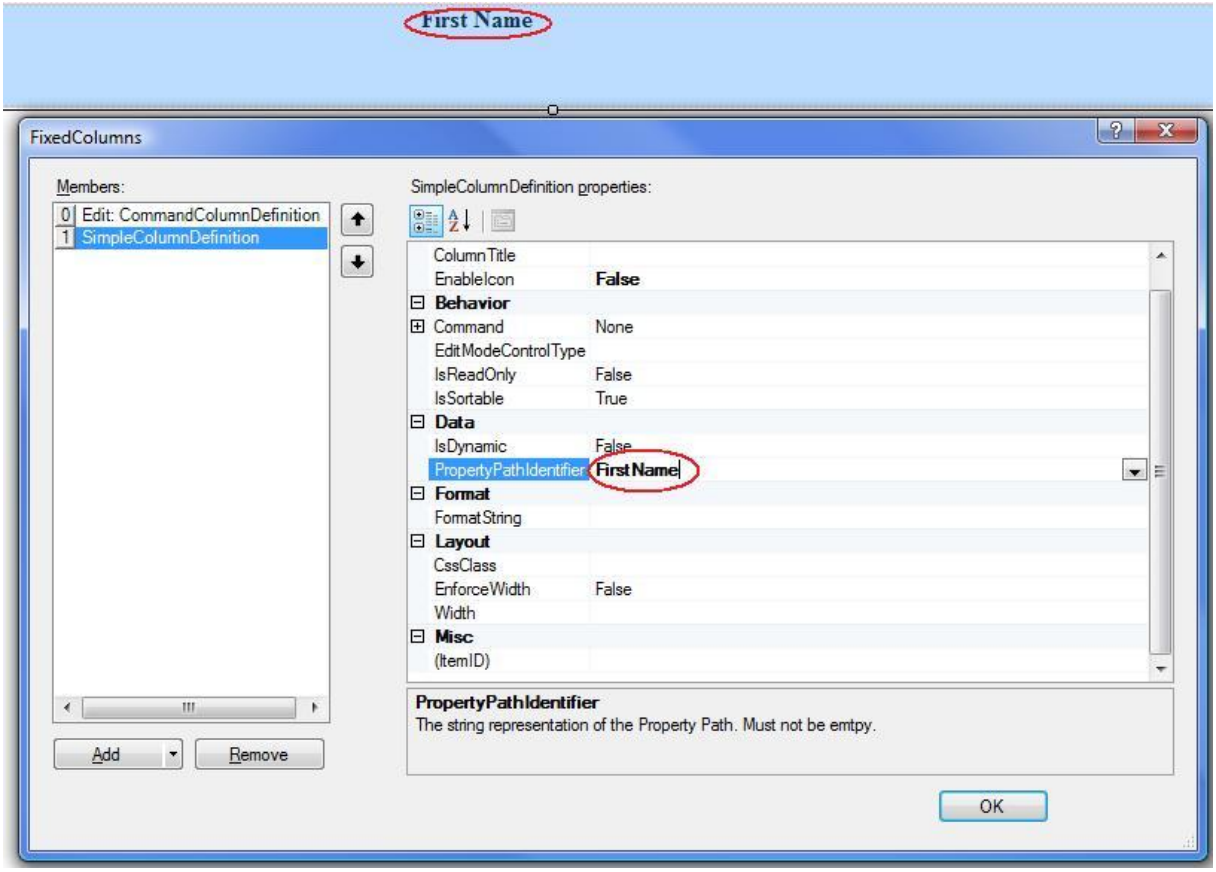
You can use the EditFixedColumns dialog box to add, remove and configure column definitions for a search result form. In this illustrated walk-thru, we get rid of the `DisplayName` column in `SearchResultPersonform.aspx`.

- open the EditFixedColumns dialog box
- in the left view with the column definitions, select "BocAllPropertiesPlaceholderColumnDefinition" and press the "Remove" button. This will rid you of the "BocAllPropertiesPlaceholderColumnDefinition".
- add a "BocSimpleColumnDefinition", as shown in the illustration:

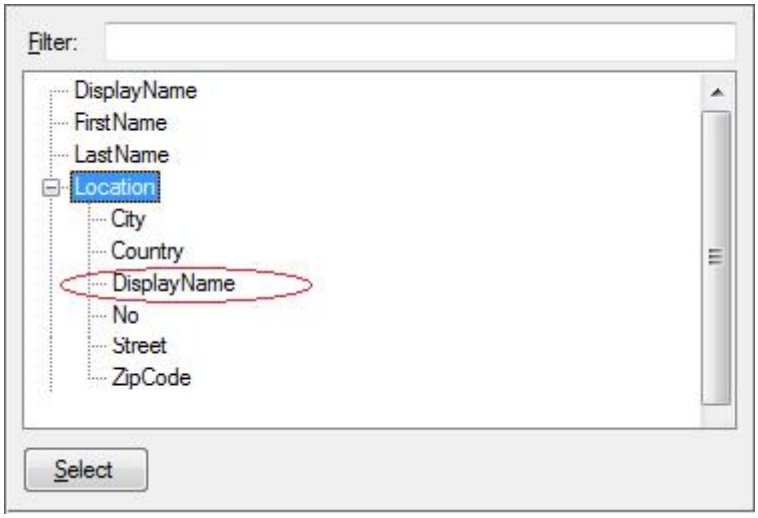


- fill in "`FirstName`" as the property path. If you arrange the dialog in such a fashion that you can see the control with the column headings in designer, you can see your changes (i.e each new `SimpleColumnDefinition` immediately. However, this interactivity comes with a price: since each new column enters the SearchResultPersonForm.aspx file immediately, closing the dialog does NOT return you to the initial state of the file. You have to UNDO unwanted modifications.

You can leave the other configuration properties with their default values for now.



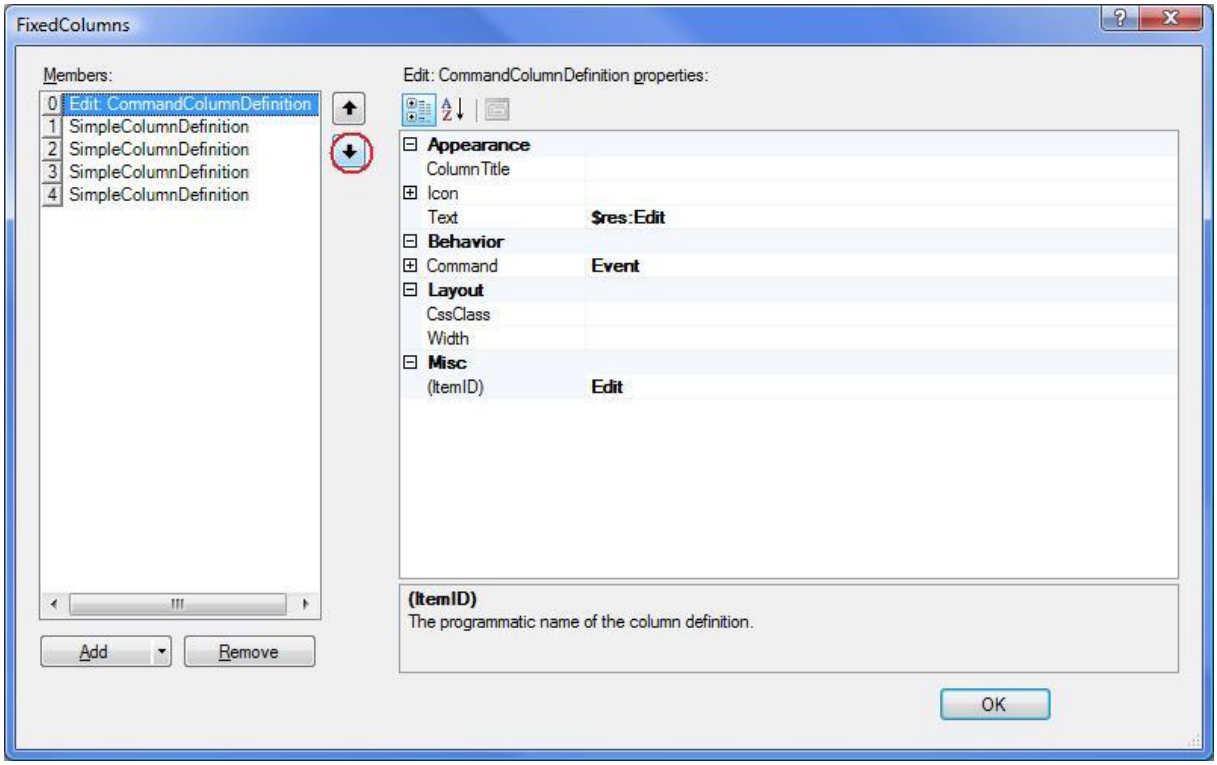
- In analogous fashion, add a simple column definition for the property path `Surname`
- `Location` and `PhoneNumber` are a little more complicated, because unlike the `FirstName` and `Surname` properties, those are reference properties, and the values they point to have themselves properties. This is reflected by the "+/-" tree-control in the FixedColumns editor:



Note that the `Location`'s `DisplayName` shows also up in the list of properties. Displaying a *condensed* or *abbreviated* version of a domain object's value in some other control is precisely what `DisplayName` is made for, so the right choice here is `DisplayName` (the *Location*'s `DisplayName`, mind you). As explained in section *Values and BOC controls*, this is the default behavior of `BocReferenceValues`: unless specified otherwise, use

`DisplayName` to display a value. Another twist is that now you have `DisplayName` as the column's title.

- Repeat the step above for `PhoneNumber` as well: Add a simple column definition for the `PhoneNumber`'s `DisplayName`.
- when you are done with all four simple column definitions, move the `BocCommandColumnDefinition` for "Edit" down, as shown in this screen-shot:



- press "Okay"

At this point, you might want to run your web-client application to see your modification in action.

Editing fixed coloumns in the search result form's ASP.NET HTML code

Visual Studio's designer interface is round-tripping, i.e. the graphical interface adapts to modifications in the HTML-code, the HTML code adapts to modifications in the graphical representation.

Your raw, unedited, generated `FixedColumn` node in the HTML source for the `SearchResultLocationForm` should look like this:

```
<FixedColumns>
        <remotion:BocAllPropertiesPlaceholderColumnDefinition />
        <remotion:BocCommandColumnDefinition ItemID="Edit"
                                             Text="$res:Edit">
          <PersistedCommand>
            <remotion:BocListItemCommand Type="Event" />
          </PersistedCommand>
        </remotion:BocCommandColumnDefinition>
      </FixedColumns>
```

The exercise here is to replace the one-size-fits-all `remotion:BocAllPropertiesPlaceholderColumnDefinition` with a set of

remotion:SimpleColumnDefinition s – one for each of the desired properties (Street, Number, City, Country, ZipCode).

Each of the SimpleColumnDefinition s itself can have many attributes, but we are mostly content with the defaults here, so none of them need to be stated explicitly.

Apart from that, each of the SimpleColumnDefinition s just needs a specification which property belongs to the given column. As you've learned in the previous section, this attribute is the PropertyPathIdentifier.

Consequently, the complete set of proper column definitions looks like this:

```
<remotion:BocSimpleColumnDefinition PropertyPathIdentifier="Street" />
        <remotion:BocSimpleColumnDefinition PropertyPathIdentifier="Number" />
        <remotion:BocSimpleColumnDefinition PropertyPathIdentifier="City" />
        <remotion:BocSimpleColumnDefinition PropertyPathIdentifier="Country" />
        <remotion:BocSimpleColumnDefinition PropertyPathIdentifier="ZipCode" />
```

That said, let the author tell you that Visual Studio's designer is not as smart as you have just become in this exercise. Visual Studio's designer can be told which properties have which default values so that it
does not render attributes and tags that have default values. However, this does not always work. Case in point: PersistedCommand and remotion:BocListItemCommand. For some reason designer insists to render these items, although PersistedCommand's default value is remotion:BocListItemCommand. If you change anything using designer after coding a property/column-list like the one above, designer will sneak

```
<PersistedCommand>
<remotion:BocListItemCommand></remotion:BocListItemCommand>
</PersistedCommand>
```

into each column definition. This means that if you modify BocList columns in designer, the resulting source listing will be more complicated and more verbose than the laconic listing above, *even if you content yourself with default values*. This behavior is inconsistent with the fact that VS designer does not fill in default values of properties.

Exercise for the reader: there is more to get rid of

The DisplayName s in SearchResultPhoneNumber and SearchResultLocation are worse than warts. You are faced with:

- in the PhoneNumbers sub-list of the Person edit form, another DisplayName makes fun of you.
- what's more, the PhoneNumbers sub-list also sports a Person column that is utterly redundant because it points back to the Person object you already see in the Person edit form.
- the Person's SearchResultForm also has a redundant DisplayName.

After the instruction in this chapter you should be able to get remove these redundant columns quite easily.

What can go wrong

The biggest problem you face as a user of the FixedColumns editor in Visual Studio's designer is missing assemblies in GAC (see section *Getting rid of* `DisplayName`). However, another manifestation is not as obvious. If you notice that designer is rendering *fully qualified type names* in the ASP.NET HTML source, make sure that all assemblies are in the GAC. A fully qualified type name is a substitute for the compact `remotion:` prefix (or the other way around). Instead of

```
remotion:BocSimpleColumnDefinition
```

designer will render it as

```
Remotion.ObjectBinding.Web.UI.Controls.BocSimpleColumnDefinition
```

(the substitution is defined in the `PhoneBook.Web's Web.config`).

This brief how-to can't do the features of BOC controls justice. We will work more with BOC controls in the next sections, but even that won't be enough. For the time being, documentation of BOC controls won't be complete, so you'll be left to your own devices (and the generated online-help in `C:\PhoneBook\Remotion\net-3.5\doc` (or

```
http://re-motion.org/content/Remotion.chm
```

respectively). However, the code for BOC controls is well thought-out, so you might get good results quickly.

Globalizing the PhoneBook web application

As you might have noticed, the tab- and property names in the web-client application are identical to the C# property names. More often than not this is undesirable. Most of your users won't be .NET programmers, after all.

If you want to display other names in your application than those found in your source-code, or want to display names or other strings for various languages, you must "globalize" your application. (Note that "globalization" is a very Microsoft-specific term. Unix-people and translators call it "*localization*" or "*internationalization*", or "l10n" and "i18n", respectively.)

In general, globalizing a .NET application means adding and fleshing out resources, and re-motion application are no different. Both parts in our PhoneBook that can benefit from globalization:

- the domain library PhoneBook.Domain (no resources defined yet)
- the application itself (some resources provided by `uigen.exe`)

Even if you don't aim for world-domination, this section will be useful for you, because you might need at least a few resources to override the display of identifier names with the display of something more user-friendly, blank-separated words instead of camel-cased enums with namespaces in them, for example.

For `uigen.exe` (and by extension, re-motion), all foreign languages are equal, but German is more equal, because it is the common language of rubicon's developers and rubicon's customers. For this

reason, `uigen.exe` generates not only resources for the default language but also for German (.de.resx files).

If you happen to know Klingon, please volunteer for translating the PhoneBook resources. The author will be delighted if there are Klingon property names in the next edition of the PhoneBook primer!

Besides that, if you have worked with resources before, this section of the tutorial will pose little difficulty for you. Which of the supported languages to choose at run-time is a matter of programming. uigen.exe generates the program in such a fashion that the culture set in Web.config is used at run-time.

Preparing the domain for globalization

So far, `PhoneBook.Domain` knows nothing about how property names are presented to user, and therefore presents their corresponding .NET identifiers instead. The remedy to this lack is attributing domain object classes with `MultiLingualResources` (or `MultiLingualResourcesAttribute` in languages other than C#.)

The parameter for this attribute is a file-name where resources for their properties can be found.

In the following example we use `Location` to illustrate the use of `MultiLingualResources` with a file name:

```
[MultiLingualResources ("PhoneBook.Domain.Globalization.Location")]
    public abstract class Location : PhoneBookObject
    {
      // code code code
    }
```

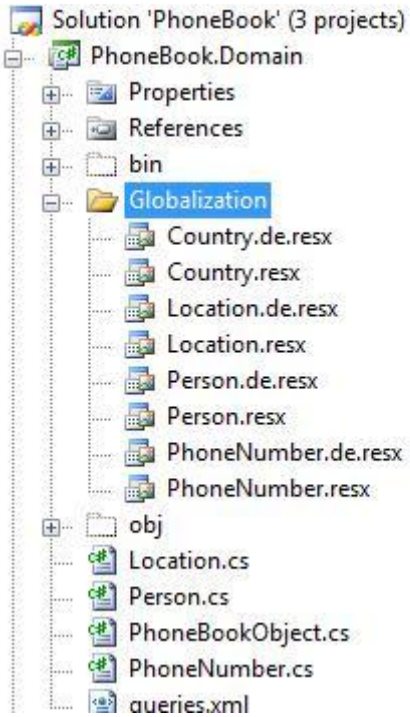
This attribute require a new using:

```
using Remotion.Globalization;
```

The following conventions are followed:

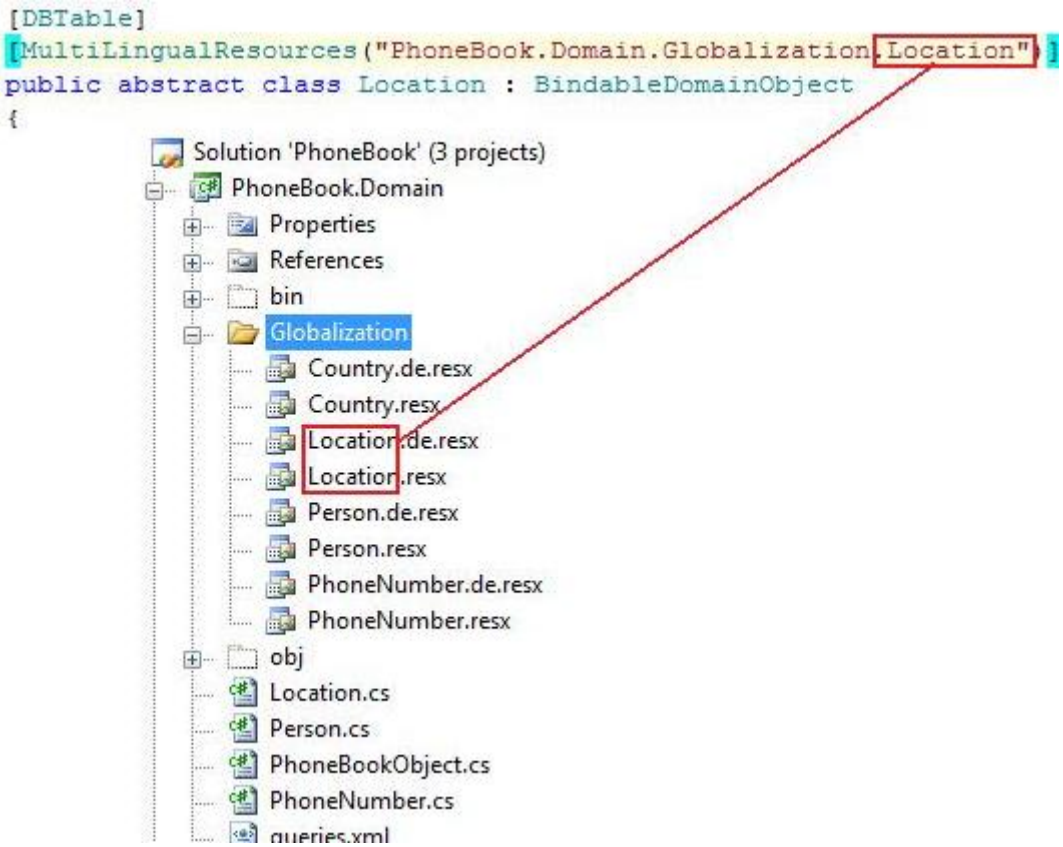
- each domain object class gets its own resource file
- each file with *default* resources has the same file-name as the corresponding class, with an ".resx" extension
- .NET supports extra languages along the following convention: an extra two-character character snippet signals the language. "de", for example, is "Deutsch", what means "German" in German. "fr" is more intuitive. It stands for "français", what means "French" in French.
- these files are stored in a dedicated `Globalization` sub-folder in your domain project

This lengthy explanation is probably not as instructive as the following picture showing a globalized PhoneBook's `Globalization` sub-folder:



Some parts of the convention are fairly arbitrary, but sticking to a convention whenever possible is always a good practice. The base names ("Location", "PhoneNumber", "Person") for the resource files are fairly arbitrary. The only requirement is that they match the name parameters in the `MultiLingualResources` class attribute for each domain object class.

The "Location" in the string parameter must match the file-names `Location.resx`, `Location.de.resx`, `Location.fr.resx`, etc. The `Globalization` in `PhoneBook.Domain.Globalization.Location"` simply is the name of the folder `Globalization`, so you have no control over that one. Here is illustration how the `MultiLingualResources` string parameter must match the resource file-name(s):

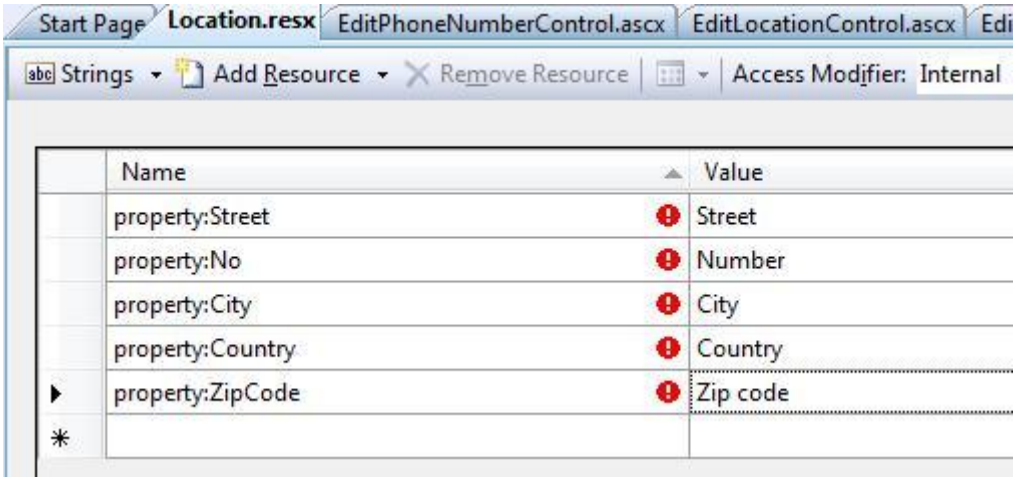


In order to prepare your domain objects for globalization, simply do the following:

- add a `MultiLingualResources` attribute to each domain object class
- create a `Globalization` sub-folder in your domain project folder
- create a new default `.resx` file for each domain object class in that sub-folder

Globalizing the `PhoneBook.Domain`

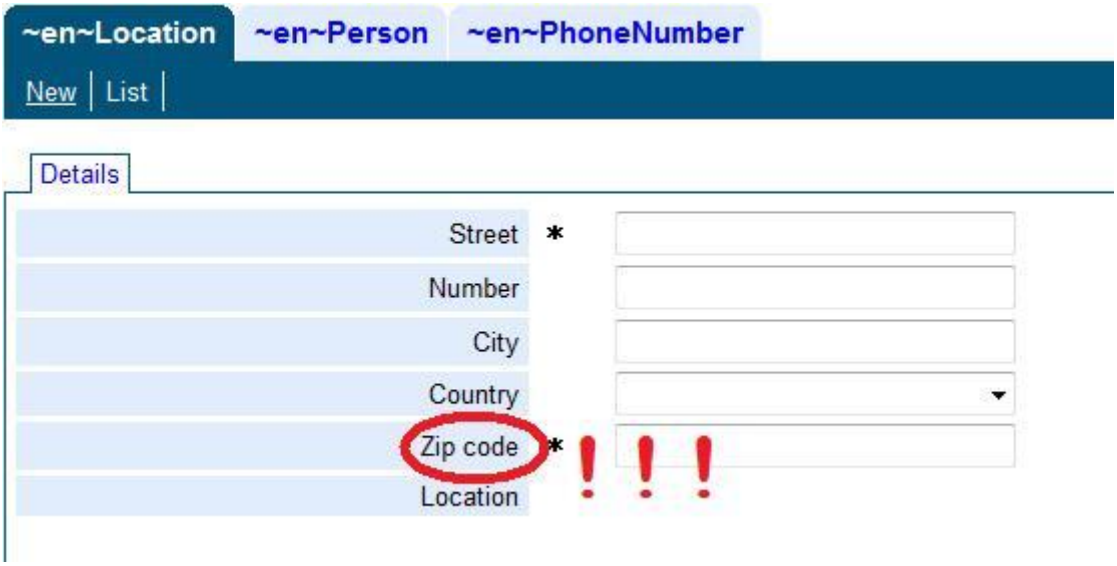
Visual Studio will open the empty resource file for you right after creation, prompting you to fill in name-value pairs into a form. The names are simply property identifier of your Location object here, prefixed with a "property:". In `Location`, this is fairly boring for English: the "localized" mappings are more or less identical to the properties' C# identifiers, the only notable exception being "Zip code", which is not camel-cased as "ZipCode" in non-geek usage. For English, you will end up with a filled-in `Location.resx`-form like this:



What about those red exclamation marks? The author agrees that the red exclamation marks (visible in the screenshot above) are fairly irritating at first, and they stem from the simple fact that ":" (as in "property:") are not allowed in resource identifiers. They are *not* syntax error. They are just warnings and have no influence on the correct operation of your resources or computer program. For this exercise we'd ask you to just play along and tolerate the red exclamation marks. In the section *Don't like red exclamation marks?* you'll learn how to get rid of them and what you'll have to give up for it.

If you run your now globalized application again, it might take a second look to find the reward for your effort. On the `Location` form, the former `ZipCode` is now spelled as a less geeky "Zip code":

Before: your unglobalized application, showing boring C# property identifiers



After: Success! Ladies and gentlemen, we are g-l-o-b-a-l-i-z-e-d!

In this fashion, you can globalize all of your domain object classes with resources.

All this works for German resource files just as well, and the differences from raw C# identifiers is usually much more dramatic than for English "globalization". Here is a snapshot of the German `Location` equivalent:

(If you want to embellish your PhoneBook application entirely in German, there are property names (resource strings) for German PhoneBook properties in the German globalization appendix to this section.)

Globalizing the `Country` enum

Globalizing `enum`s works along the same line as for properties. Add the attribute `EnumDescriptionResource` and the fully qualified identifier as a parameter:

```
[EnumDescriptionResource ("PhoneBook.Domain.Globalization.Country")]
    public enum Country
    {
        Austria = 0,
        Australia = 1,
        BurkinaFaso= 2
```

```
}
```

Next you add your resource-file(s) `Country.resx` and `Country.de.resx` and put the translations in there, just as for the domain object classes:

	Name	Value
▶	PhoneBook.Domain.Country.Australia	Australia
	PhoneBook.Domain.Country.Austria	Austria
	PhoneBook.Domain.Country.BurkinaFaso	Burkina Faso
*		

However, a subtle difference is that these enums are, by definition, not properties. For this reason, they don't get a `property:` prefix. Instead, you must speficy their fully qualified identifier (= including the namespace).

Just for completness, here are the German resource strings:

	Name	Value
▶	PhoneBook.Domain.Country.Australia	Australien
	PhoneBook.Domain.Country.Austria	Österreich
	PhoneBook.Domain.Country.BurkinaFaso	Burkina Faso
*		

What can go wrong

The only gotcha the author sees for globalization is misspelling something, typically mixing up something like "PhoneNumber" and "PhoneNumbers". Typos can bite you

- in property names (so that resource identifiers don't match)
- in file names that don't match the declaration in `MultiLingualResources`.

Globalizing the globals

Modifying global.resx and global.de.resx

In contrast to domain object resources, the resources for the basic framing elements of the web-client application are created for you by `uigen.exe`. The most important of those should be fairly obvious:

- "Cancel", "Save"
- "List", "New"
- "Add"
- "Search"

The home the of the web-client application's resources is `global.resx` (`global.de.resx` for German). Since globalization of the web-client application works just as for any typical ASP.NET application, you find (or place) your `global.*.resx` files in the web-client project's `Globalization` folder.

In addition to the generic resources, `uigen.exe` also generates places for domain object class names, to be displayed as the label for each tab:

Name	Value
Add	Add
Cancel	Cancel
Details	Details
Edit	Edit
List	List
Location	~en~Location
New	New
OK	OK
Person	~en~Person
PhoneNumber	~en~PhoneNumber
Save	Save
Search	Search

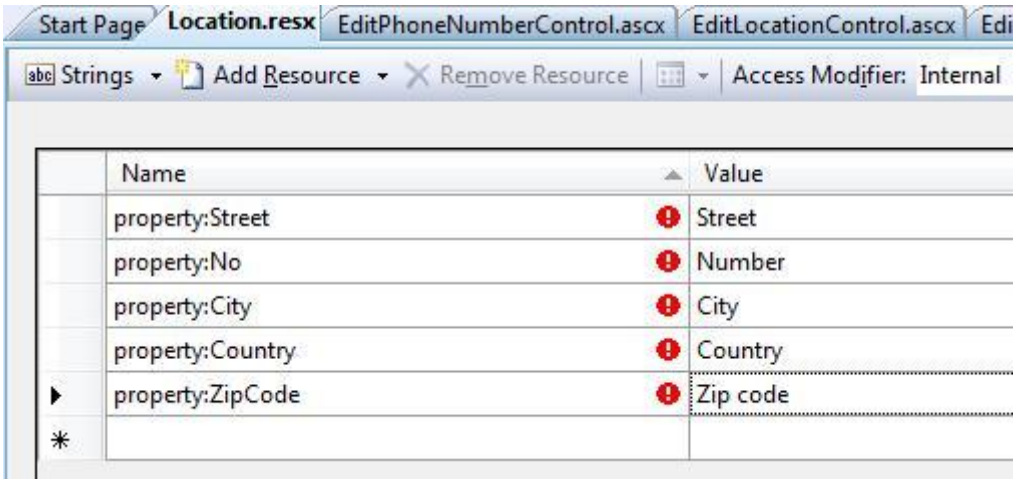
The circled resource strings for domain object class identifiers correspond to what you see here in a tabbed editor application:

If you go with the C# identifiers for globalizing the English property names is a matter of taste. Note that there are two spots for display names for `Person` and `Location`, one each for

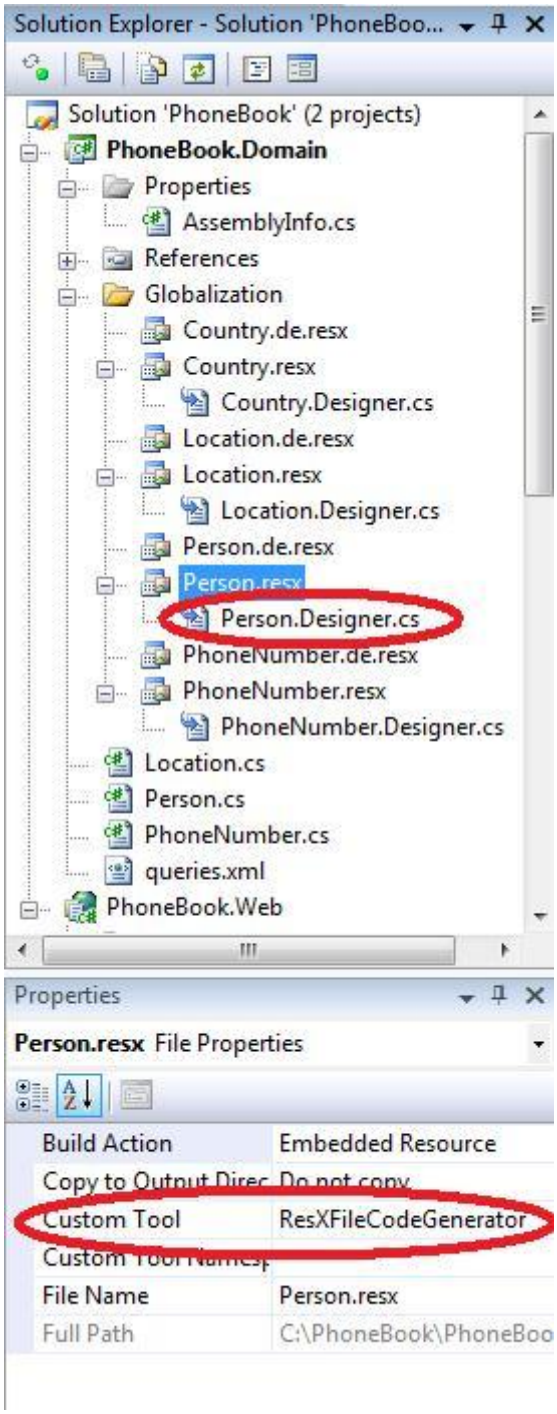
- the tab headings for the object's edit form
- the property referencing the object

Don't like red exclamation marks?

As pointed out in the section above, you will see red exclamation marks when editing the resources for a class. Here is a copy of the illustration above:



The reason for this is that Visual Studio's designer per default generates a .cs file for each resource file you create, and that the ":" in the "property:" prefix is slightly incompatible with the generated code. The tool is configurable in the .resx file's `CustomTool` property:



The `.designer.cs` file generated is wrapper code for accessing resources. re-motion does not *require* such code, so you can turn off code-generation by setting the `.resx` file's `CustomTool` property to `None`.

This will make the exclamation marks go away.

The "benign incompatibility" that causes them is that the wrapper code contains a method for each resource identifier, such as `property_FirstName` for the `Person`'s `property:FirstName` resource identifier. As you might guess, the "property:FirstName"s ":" is an illegal character for C# identifiers, so the code generator changes it to "_" and warns you about this fact by showing you those exclamation marks. That's all there is to it.

Turning off generation of wrapper-code might or might not look like a good idea to you, but wrapper-code has an advantage for resources that are only used by your web-client application, not by the domain library resources. For those, you can use the attribute `WebMultiLingualResources` instead of `MultiLingualResources`. `WebMultiLingualResources` come with the benefit of *typing*: Instead of passing the resource identifier as a string, you can use the generated class' typename in the generated namespace. Instead of using

```
[MultiLingualResources("PhoneBook.Web.SomeClass")]
```

you can use

```
[WebMultiLingualResources(PhoneBook.Web.SomeClass)]
```

German globalization mini-appendix

Location.de.resx	in German
Street	Straße
Number	Hausnummer
City	Ort
Country Land	
ZipCode	Postleitzahl
Person.de.resx	**in German**
First name	Vorname
Surname	Nachname
PhoneNumbers	Telephonnummern
Location	Standort
PhoneNumber.de.resx	**in German**
CountryCode	Ländervorwahl
AreaCode	Vorwahl
Number	Nummer
Extension	Durchwahl
Person	Inhaber
Global.de.resx	**in German**
Person	Person

Location	Standort
PhoneNumber Telephonnummer	

Creative break -- adding icons

As you might remember from the sneak preview of the finished (Stefan's) PhoneBook web application, re-motion supports the display of icons to signal a domain object's type.

You might have guessed that providing an icon is part of a domain object's behavior, but you'd have guessed wrong.

Providing *icon information*, i.e. where to find an icon is part of one service provided by the `BindableObjectProvider`. You can learn more about services and `BindableObjectProvider` in the section `WebUIService` and other services as a *provider*. This section on icons will give you an impression of how services work in principle – and give you opportunity to add your personal touch to your PhoneBook web application.

How exactly an icon bitmap for a particular object (or, object type) is provided to the framework is an implementation detail abstracted by an interface named `IBusinessObjectWebUIService`, which in turn is a specialization of `IBusinessObjectService`. This interface requires three methods:

- GetHelpInfo
- GetToolTip
- GetIcon

Of those, our implementation will flesh out only `GetIcon`. In order to make your implementation accessible to the framework, you must register it with the `BindableObjectProvider`, a central authority managing all sorts of `IBusinessObjectService`s. Other examples for such services include

- `BindableDomainObjectSearchService`
- `BindableObjectGlobalizationService`

and any you can provide.

Our implementation of `IBusinessObjectWebUIService` requires to fetch the icon bitmap for a given object from a file after determining the *static* type of that object. Determining the *static* type of the given object is not as straightforward as you might think, because `typeof` will give you the *dynamic* type, i.e. the class generated at run-time.

To make it work, we must register an instance of our implementation of `IBusinessObjectWebUIService` with the `BindableObjectProvider`.

Draw some icons!

If you don't feel for some creativity, you can use the icon bitmap files provided by the author, to be found in the PhoneBook sample's `tutorial\tutorial-files` folder. The author used the an icon size of 16 x 16 pixel for his cubistic artwork, and the following file-names:

- `Icon-Person.gif`
- `Icon-PhoneNumber.gif`
- `Icon-Location.gif`
- the location of the file-names is (by convention) the `Images` sub-folder in the `PhoneBook.Web`'s project folder, where the re-motion framework's `arrow.gif` and the company logo `rublogo.gif` are located (as generated by `uigen.exe`.
- The sample code in this section assumes these icon sizes and file names and file location. If your implementation is different, don't forget to adapt the sample code accordingly.

Here is the vendor-supplied ensemble of PhoneBook icons (the first is a self-portrait, the last a question mark for "unknown", "null" or "empty"):

If you don't want to be creative, you can use the prepared files in

`http://re-motion.org/content/PreparedFiles/PhoneBook`

Implementing `WebUIService`

`WebUIService` is our class that implements the interface `IBusinessObjectWebUIService`. The file for this class belongs into the sub-folder `Classes` in the `PhoneBook.Web`'s project folder. For our focus on adding icons the most important method is `GetIcon`:

```
using System;
using System.Web;
using Remotion.Data.DomainObjects;
using Remotion.ObjectBinding;
using Remotion.ObjectBinding.Web;
using Remotion.ObjectBinding.Web.UI.Controls;
using Remotion.Web.UI.Controls;

namespace PhoneBook.Web.Classes
{
  public class WebUiService : IBusinessObjectWebUIService
  {
    public IconInfo GetIcon (IBusinessObject obj)
    {
      if (obj == null)
      {
        return new IconInfo ("~/Images/Icon-Null.gif", "16px", "16px");
      }
      else
      {
        // determine the static type of the
        // passed object, i.e. the originally declared domain object
        // class (Person, PhoneNumber or Location).
        // The cast is *not quite* correct
        // but will always work for the PhoneBook web application
        Type staticType = ((DomainObject) obj).GetPublicDomainObjectType ();
        // assemble the path to the icon bitmap based on the class of the
        // object
        return new IconInfo ("~/Images/Icon-" + staticType.Name + ".gif", "16px", "16px");

      }
    }

    public string GetToolTip (IBusinessObject obj)
    {
      return null;
    }

    public HelpInfo GetHelpInfo (IBusinessObjectBoundWebControl businessObjectBoundWebControl,
                                 IBusinessObjectClass businessObjectClass,
                                 IBusinessObjectProperty businessObjectProperty,
                                 IBusinessObject businessObject)
    {

      return null;
    }

  }
}
```

Please note that the cast from `IBusinessObject` to `DomainObject` (which gives us the `GetPublicDomainObjectType`) is not always the right way to get an object's static type. As explained in the section **Fehler! Verweisquelle konnte nicht gefunden werden.**, not every business object is a domain object, so such a code will break as soon as you try to cast an object to a `DomainObject` that implements `IBusinessObject` while not being a domain object. This fine point does not apply to the PhoneBook web-client application, however, so we are safe here.

Implementing is only one leg on the way to having icons in your PhoneBook web application.
The other part is registering it with the `BusinessObjectProvider`. In `Global.asax.cs`, add the following statement:

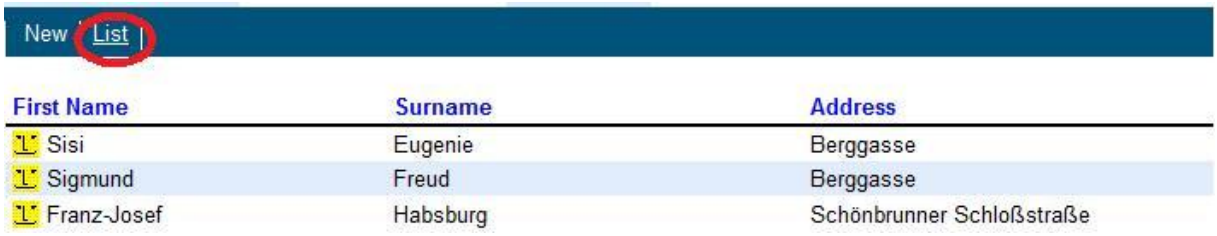
```
BindableObjectProvider.GetProvider<BindableDomainObjectProviderAttribute>().AddService (
          typeof (IBusinessObjectWebUIService), new WebUIService());
```

Two usings are required to resolve this statement:

```
using Remotion.ObjectBinding.Web; // for IBusinessObjectWebUIService
using PhoneBook.Web.Classes;      // for WebUIService
```

One registration should already be there, namely the one for the `GetObjectService`. (This is the service that does the actual work for getting a domain object by ID with `GetObject<T>()`).

After compiling your additions and providing icon bitmaps in the `Images` folder, you should see your icons on the *List* page of any of the three tabs, as in this illustration:



What can go wrong?

In terms of re-motion, not much. Just make sure that you get the expression for composing the file paths to the icon files right (the author failed to do so multiple times).

`WebUIService` and other services as a *provider*

Services, as understood here in the context of

- `WebUIService`
- `GetObjectService`
- `DomainObjectSearchService`

follow the *provider* pattern. They add centrally managed, customizable, behaviour or features to objects and types where adding a method to a class is not desirable for some reason. In the case of getting icon bitmaps for classes (with `GetIcon`, alternatives to a provider (or service) are unattractive. *Of course* it would be easy to implement a `GetIcon` static method for all `DomainObject` classes, ready to be overridden by all derived classes. This brings a few problems into the code, however:

- `DomainObject` s should not know anything about UIs in order to keep the model-view-controller philosophy clean, so `MyDomainObject.GetIcon()` would be inappropriate.
- `BindableDomainObject` is not supposed to know anything about the UI all by itself, it only adds the `IBusinessObject` interface to `DomainObjects`, and all `IBusinessObject` is supposed to know is how to *bind* properties to controls. Therefore, `BindableDomainObject.GetIcon()` would be inappropriate as well. `IBusinessObject.GetIcon()` is even wronger, because, by principle, interfaces can't have static methods.

Consequently, implementing `GetIcon` as part of a provider is a clean way to add specialized behavior (i.e. a deidicated icon) to each type.

In the case of `DomainObjectSearchService` and `GetObjectService`, the service can't even start with a given domain object (bindable or not), because their job is it to get a (bindable) domain object from the repository, given an item to search for or an domain object ID. Whether these services return a retrieved object at all must be flexible for various security scenarios. A provider gives you separate specialization without the need of sub-classing the types that contain the business logic.

The BOCList -- inside column definitions

The BOCList is a complex control with an infinitude of possibilities. A comprehensive tutorial for the BOCList would be a 200-page bouncer all by itself. However, most of its configuration - fonts, colors, etc. - is fairly obvious. ASP.NET programmers in particular should have very little difficulties working with the BOCList, even without comprehensive documentation. This section has a focus on the most important and non-obvious characteristics of the BOCList.

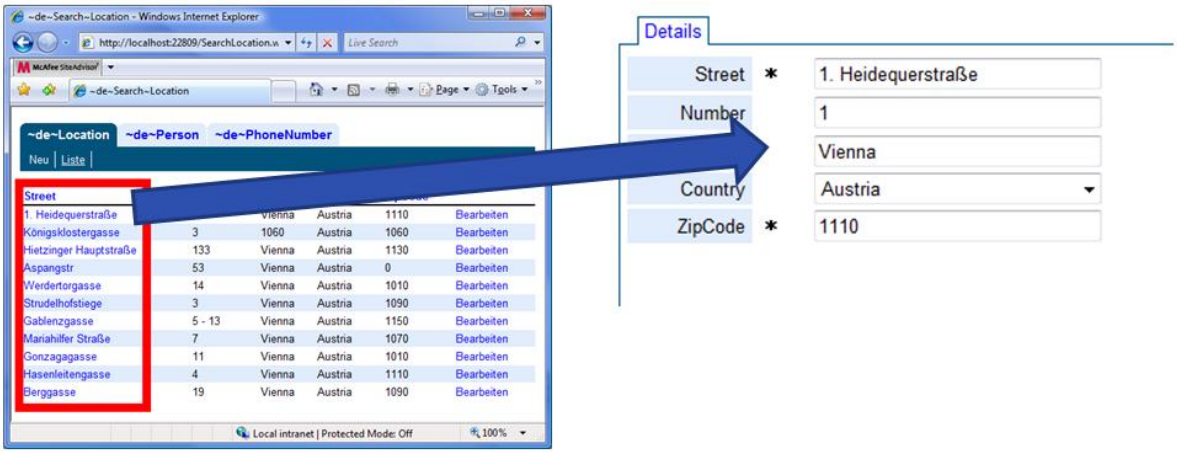
Simple column with a command

A simple column can be rendered as a link that causes a `ListItemCommandClick` event. All you need to do is to add

an item ID to the column, so that it can identify itself with the event handler

add a *persisted command* node to the control's XML

In this primer, we have not covered yet how to hook a certain page to a command – this is re-call territory, and will not get there before chapter *The virtues of re-call, illustrated*. For the next demonstration, we will simply hijack the event-handler and mechanism for the `BocCommandColumnDefintion` for the `SearchResultLocationForm`'s *Edit* command. For clarification, this is what our `BocSimpleColumnDefinition` with a command will do.



We want to end up in the `Location`'s edit form when we click on the street

Here is how to fix up the *Street* column's definition, so that it causes a `ListItemCommandClick` event.

In the course of our riddance of `DisplayName`, we have left the *Street* column's XML like this:

```
<remotion:BocSimpleColumnDefinition PropertyPathIdentifier="Street" />
```

That is a plain-vanilla `BocSimpleColumnDefinition` – all it does is to specify from which property it gets its value to display from.

The XML will look a little more verbose if you have used the FixedColumns editor. The FixedColumns editor hacks extra default (i.e. redundant) attributes and an (empty) `<PersistedCommand>` node into the definition:

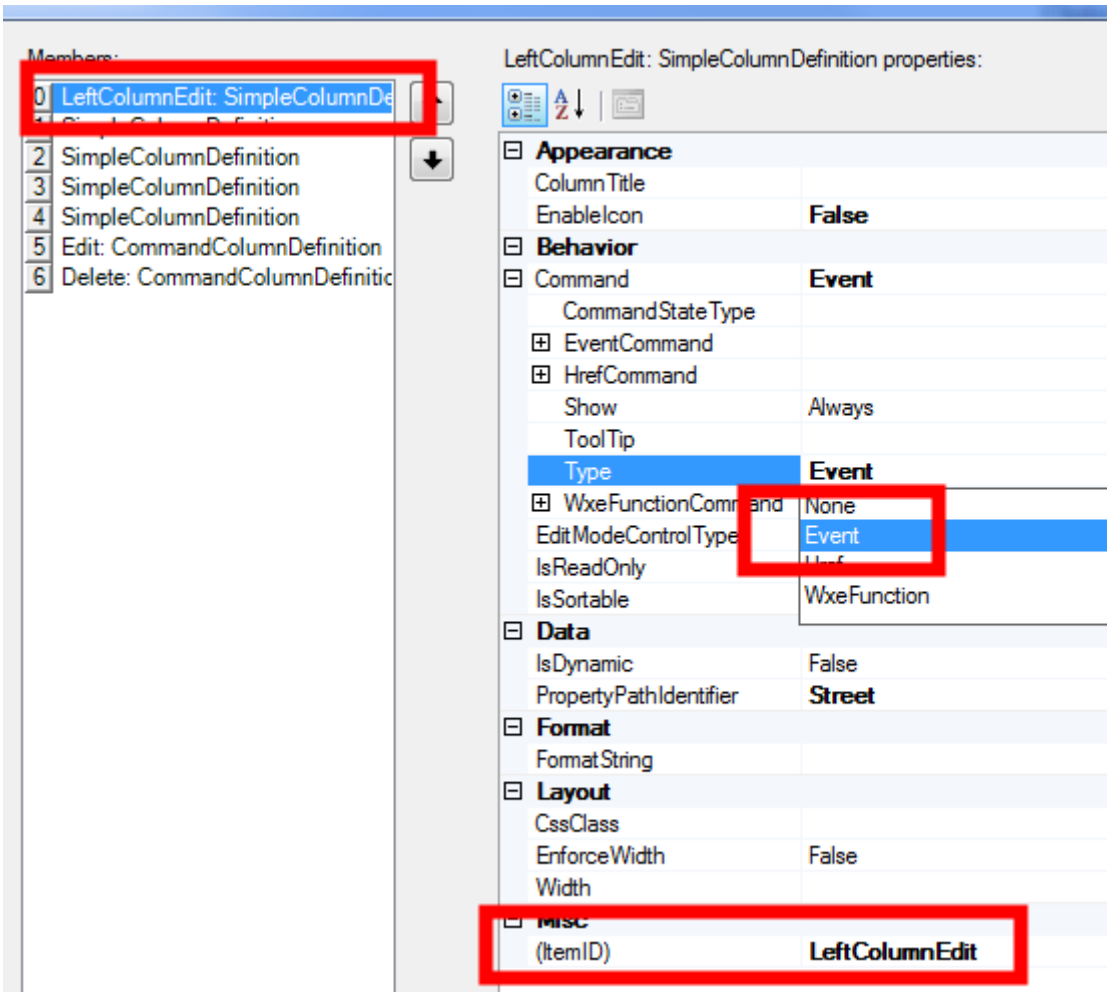
```
<remotion:BocSimpleColumnDefinition PropertyPathIdentifier="Street"
EnableIcon="False">
          <persistedcommand>
            <remotion:BocListItemCommand />
          </persistedcommand>
</remotion:BocSimpleColumnDefinition>
```

The hollow, do-nothing `<PersistedCommand>` node is utterly redundant for plain `BocSimpleColumnDefinition`, but it comes in handy for our purposes, because the new and improved definition shall look like this:

```
<remotion:BocSimpleColumnDefinition PropertyPathIdentifier="Street"
ItemID="StreetColumnEdit">
          <persistedcommand>
            <remotion:BocListItemCommand Type="Event" />
          </persistedcommand>
</remotion:BocSimpleColumnDefinition>
```

For the `ItemID`, we could have used "Edit", but this ItemID is already taken by the `SearchResultPersonForm`'s *Edit* `BocCommandColumnDefinition`, **but duplicate item IDs on the same page are prohibited in re-bind.**

If you don't want to type in the modifications you can use the FixedColumn editor, of course. In VS' designer, open the FixedColumn editor and supplement the item ID and the *Event* type to the first (Street's) definition:



That's it.

No matter whether you hack the definition in the source code or use designer, the `ItemID` **IS** important. **Without an ItemID, your column won't even render the `Street` value as a link.** And you won't get a link either if you forget the all-important "Event"-parameter for the command *Type*.

All that's left to do is to make event-handler do the same thing it does for the *Edit* column's "Edit" command. In the code-behind `SearchResultLocationForm.aspx.cs`, add the new ItemID "StreetColumnEdit" so that looks like this:

```
protected void LocationList_ListItemCommandClick (object sender,
                                BocListItemCommandClickEventArgs e)
    {
      // NEW! '|| e.Column.ItemID == "StreetColumnEdit"' is added:
      if (e.Column.ItemID == "Edit"
                          || e.Column.ItemID == "StreetColumnEdit")
      {
        try
        {
          EditLocationForm.Call (this, (Location)e.BusinessObject);
          ClientTransaction.Current.Commit ();
        }
        catch (WxeUserCancelException)
        {
        }
      }
```

```
}
```

DeleteMakeHomeless revisited for the web

A *BOC command column* provides a link that performs a certain action for the instance the link is in. A typical example for a `BocCommandColumnDefinition` is the *Edit* column in all search result forms:

The `BocCommandColumnDefinition` is closely related to the `BocSimpleColumnDefinition` in that BOTH can be wired to a command. The `BocSimpleColumnDefinition` is different only in that it can display a (clickable) property value, whereas the `BocCommandColumnDefinition` can only show simple text or an icon to click on.

in the re-store chapter, we have added a method `DeleteMakeHomeLess` to the `Location` class, so that it can remove itself from the database and sever all relations from persons referencing that `Location`. Now it is time to bring this feature to our web-application. Wouldn't it be nice to have a "Delete" facility right on our `SearchResultLocationForm`? Of course it would, because it demonstrate how to work with `BocCommandColumnDefitions`. Here is what we want to achieve:

Street	Number	City	Country	ZipCode		
Königsklostergasse	3	1060	Austria	1060	Bearbeiten	Delete
Hietzinger Hauptstraße	133	Vienna	Austria	1130	Bearbeiten	Delete
Aspangstr	53	Vienna	Austria	0	Bearbeiten	Delete
Strudelhofstiege	3	Vienna	Austria	1090	Bearbeiten	Delete
Gablenzgasse	5 - 13	Vienna	Austria	1150	Bearbeiten	Delete
Mariahilfer Straße	7	Vienna	Austria	1070	Bearbeiten	Delete
Gonzagagasse	11	Vienna	Austria	1010	Bearbeiten	Delete
Hasenleitengasse	4	Vienna	Austria	1110	Bearbeiten	Delete
Berggasse	19	Vienna	Austria	1090	Bearbeiten	Delete

The user opts to cleanse his phone-book of the Strudelhofstiege(what is not even a real address, by the way)

Street	Number	City	Country	ZipCode		
Königsklostergasse	3	1060	Austria	1060	Bearbeiten	Delete
Hietzinger Hauptstraße	133	Vienna	Austria	1130	Bearbeiten	Delete
Aspangstr	53	Vienna	Austria	0	Bearbeiten	Delete
Gablenzgasse	5 - 13	Vienna	Austria	1150	Bearbeiten	Delete
Mariahilfer Straße	7	Vienna	Austria	1070	Bearbeiten	Delete
Gonzagagasse	11	Vienna	Austria	1010	Bearbeiten	Delete
Hasenleitengasse	4	Vienna	Austria	1110	Bearbeiten	Delete
Berggasse	19	Vienna	Austria	1090	Bearbeiten	Delete

Done! Gone!

What's comforting about this delete-command is that it does not require re-call (= no transition to another page). We simply create the extra `BocCommandColumnDefinition` and call the `DeleteMakeHomeless` method from section *Deleting objects*.

Taking the `uigen.exe`-generated `BocCommandColumnDefinition` for *Edit* as a model, you could probably do this exercise all by yourself.

Here is a copy of that *Edit* column, as generated by `uigen.exe`:

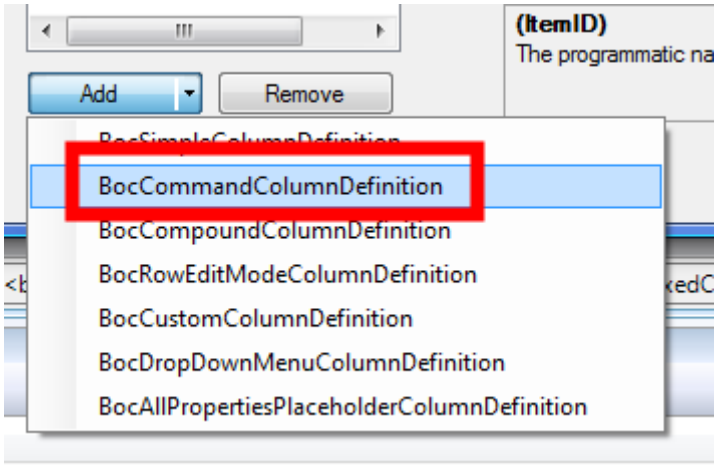
```
<remotion:BocCommandColumnDefinition ItemID="Edit" Text="$res:Edit">
          <PersistedCommand>
            <remotion:BocListItemCommand Type="Event" />
          </PersistedCommand>
        </remotion:BocCommandColumnDefinition>
```

It looks very similiar to a `BocSimpleColumnDefinition` in that it has a command attached to it, the only difference being that a `BocCommandColumnDefinition` has a `PropertyPathIdentifier` attribute for its text to click on, whereas the `BocCommandColumnDefinition` has a `Text` attribute for that. The `PersistedCommand` "Event" causes the `ListItemCommandClick` event; the `ItemID` enables the column to identify itself, so that the event-handler knows who caused the click and what to do about it.

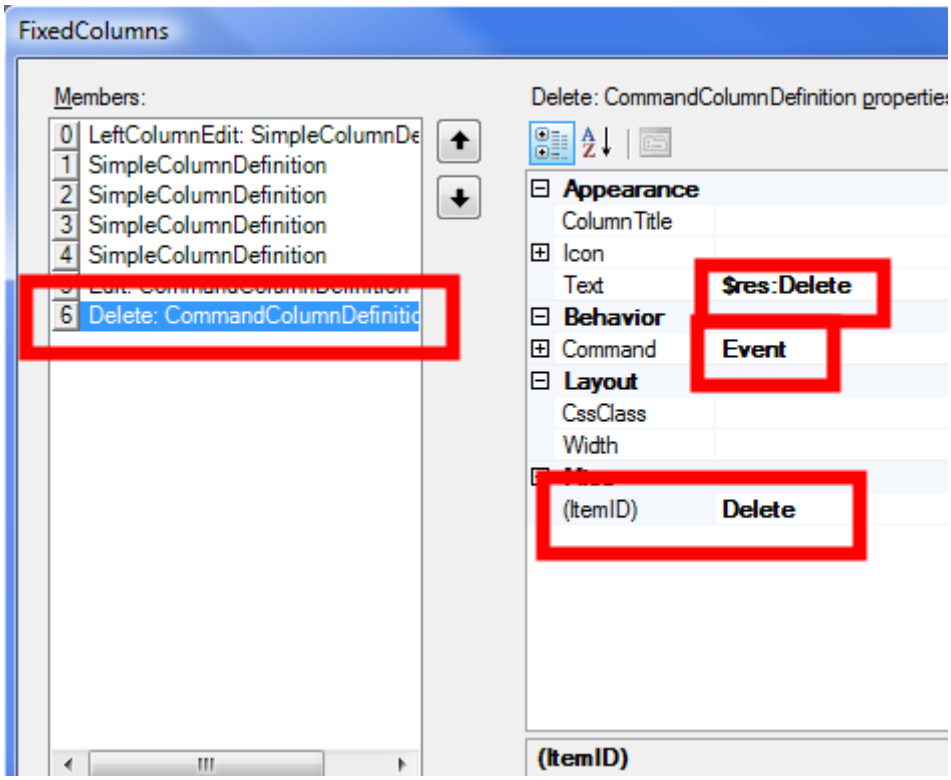
Consequently, our command column for *Delete* s looks very similar:

```
<remotion:BocCommandColumnDefinition ItemID="Delete" Text="$res:Delete">
          <PersistedCommand>
            <remotion:BocListItemCommand Type="Event" />
          </PersistedCommand>
        </remotion:BocCommandColumnDefinition>
```

Add this snippet to the "LocationList" BocList in `SearchResultLocationForm.aspx`, right before the command column for "Edit". Or use the FixedColumns editor instead. If you want to go the designer-route, first create a new `BocCommandColumnDefinition`:



Second, fill in the text, set the command type to *Event* and make sure the column has an ItemID:



Again, both the *Event* parameter and a valid `ItemID` are required, otherwise the column won't even have a link to click on.

The section for *Delete* must be added to the event-handler:

```
protected void LocationList_ListItemCommandClick (object sender,
BocListItemCommandClickEventArgs e)
    {
      if (e.Column.ItemID == "Edit"
                          || e.Column.ItemID == "LeftColumnEdit")
      {
        try
        {
          // EditLocationForm.Call (this, (Location)e.BusinessObject);
          var args = new WxePermaUrlCallArguments (true);
          EditLocationForm.Call (this, args,
                                      ((Location) e.BusinessObject).ID);
          ClientTransaction.Current.Commit ();
        }
        catch (WxeUserCancelException)
        {
        }
      }
      //*** THIS IS THE NEW PART ***/
      else if (e.Column.ItemID == "Delete")
      {
        ((Location) e.BusinessObject).DeleteMakeHomeless ();
        ClientTransaction.Current.Commit ();
        // These three lines reload the BocList items after deletion
        // Never mind how that works for now
        var searchAllService = new BindableDomainObjectSearchAllService ();
        var listLocations = searchAllService.GetAllObjects
                            (ClientTransaction.Current,
                             typeof (Location));
```

```
        LocationList.LoadUnboundValue (listLocations, IsPostBack);
      }
    }
```

This code snippet does two things:

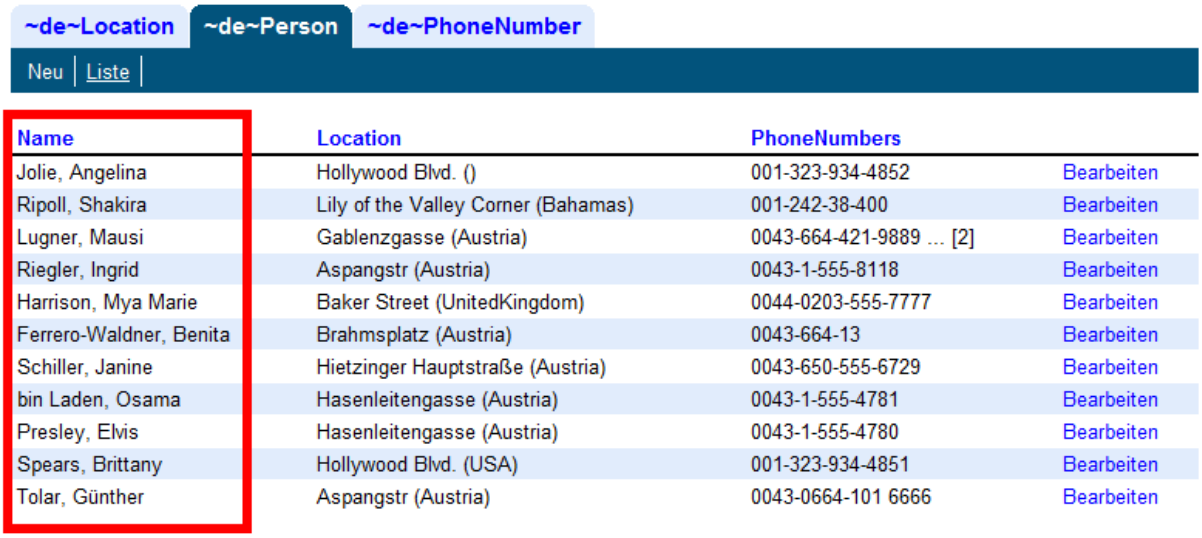
- It calls the `Location` instance's `DeleteMakeHomeless` (discussed in a previous section, *Deleting objects*)
- It calls a `BindableDomainObjectSearchAllService`, so that the `BocList` items get reloaded without the deleted item

The three extra lines for the latter task are exact copies from another `SearchResultLocationForm` event-handler – from `Page_Load`. In `Page_Load` this codes loads the BocList when the page is originally composed.

Compound column definition -- get one column for two

A compound column can display more than one property, merged together in a format string.

We can use a `BocCompoundColumn` definition to display both first name and surname in the same column in the `BocList` of the `SearchResultPersonForm`:



The "Jolie, Angelina" you see in the first line is a result of combining the `Person`'s `Surname` and `FirstName` with the format string "{0}, {1}"

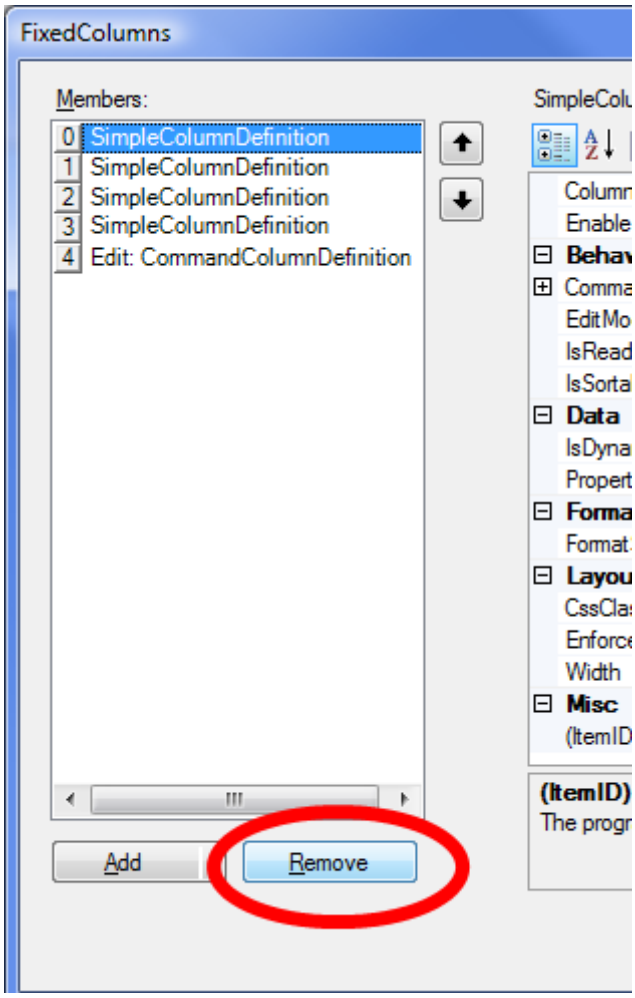
The .aspx declaration for the compound column:

```
<remotion:BocCompoundColumnDefinition ColumnTitle="Name" FormatString="{0}, {1}">
          <propertypathbindings>
            <remotion:PropertyPathBinding PropertyPathIdentifier="Surname" />
            <remotion:PropertyPathBinding PropertyPathIdentifier="FirstName" />
          </propertypathbindings>
        </remotion:BocCompoundColumnDefinition>
```

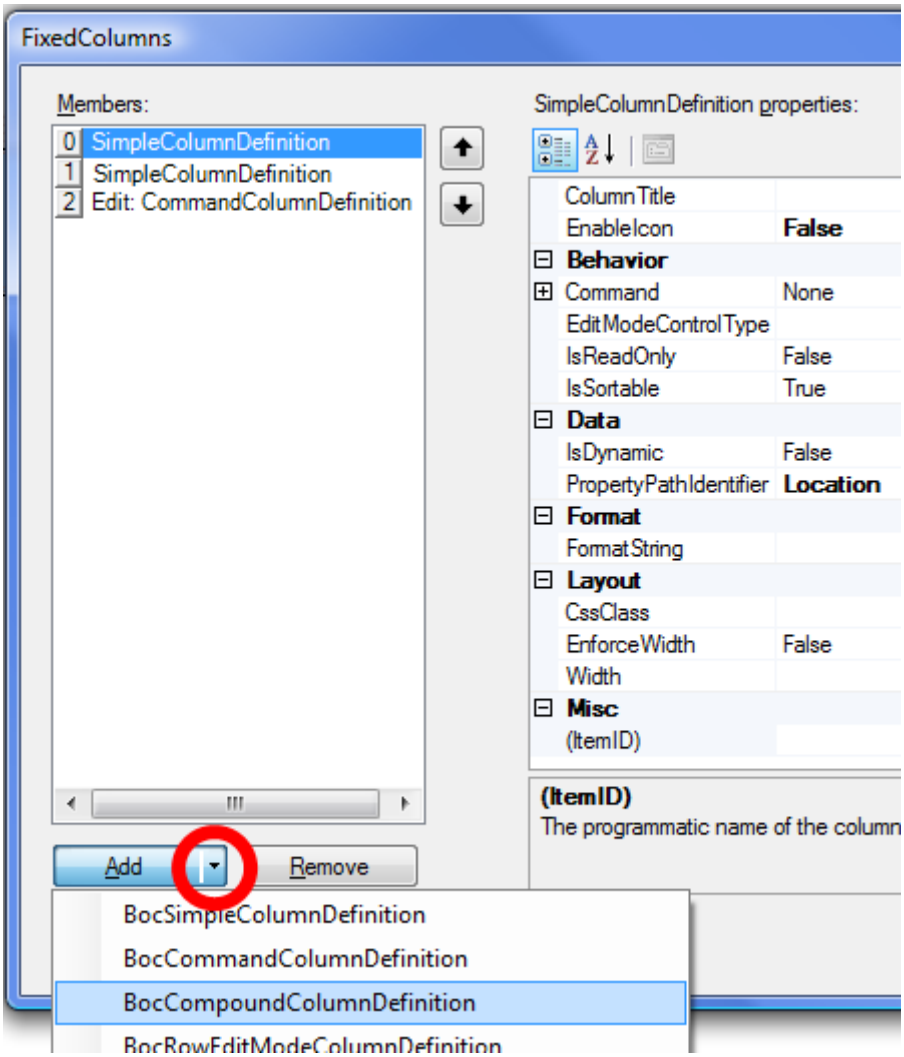
In `SearchResultPersonForm.aspx`, *replace* the two columns for the `Person`'s `FirstName` and `Surname` column definitions with the compound column definition above.

The equivalent procedure in the FixedColumns editor

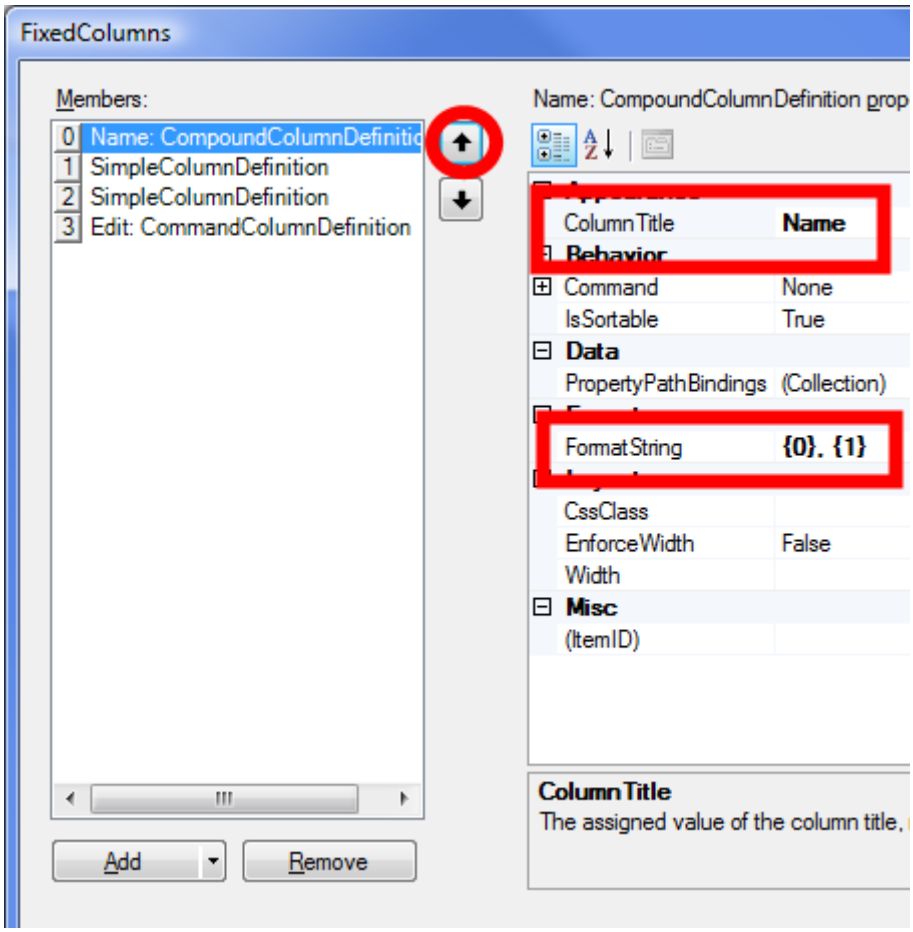
Remove the `FirstName` and `Surname` column definitions:



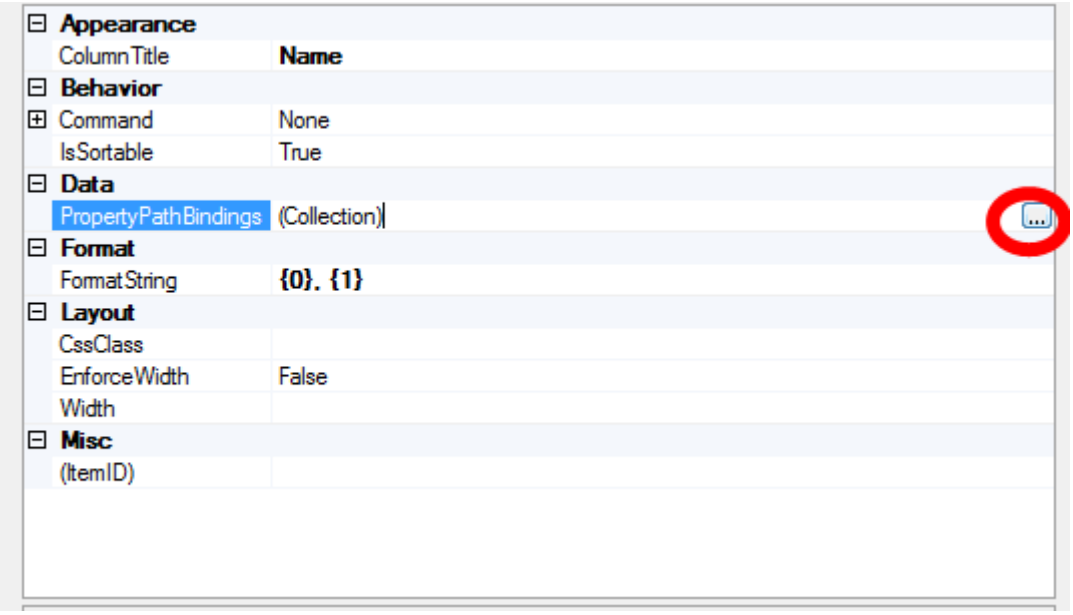
Add a new compound column definition (click on the small arrow instead of the button):



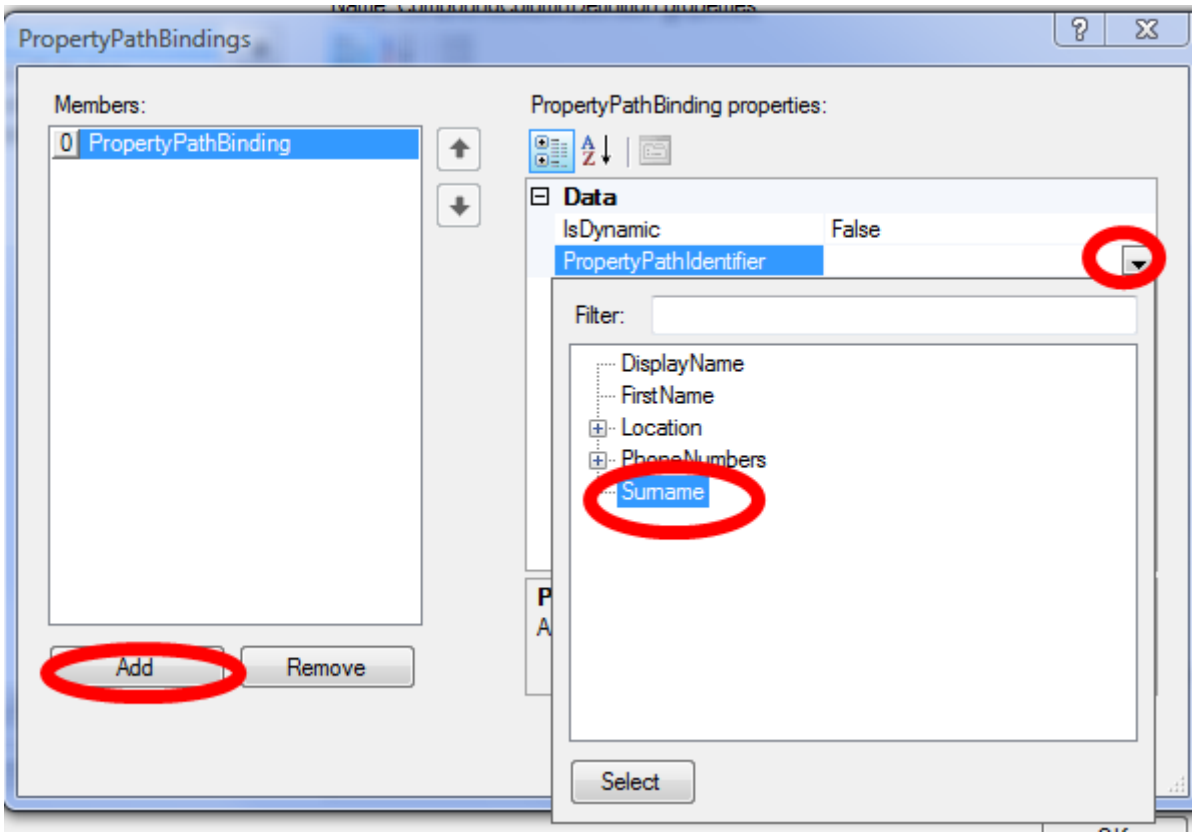
Give the compound column definition the name "Name" and the format string "{0}, {1}"



Now on to the property-paths. This requires editing the *property path collection* in the FixedColumns editor. In the configuration pane, click on the "..."s to open the collection editor:



Add the two property-paths "Surname" and "FirstName" (in that order). This requires to click on the "Add" button for each property path and select it from a list (but you can type it in as well):



Mind the sorting, mind the format string!

Two minor gotchas exist for `BocCompoundColumnDefintions`:

- for sorting columns, the order of columns is important, not the order in which they are rendered by the format string
- no format string means you won't see anything without warning

As for the first point: You might think that ordering the name columns `Surname` - `FirstName` and displaying it with a "{0}, {1}" format string is equivalent to ordering them `FirstName` - `Surname` and displaying it with a "{1}, {0}" format string. And it is. However, when it comes to sorting them, the sorter looks at the order of columns, not on the order of output.

As for the second point: If you forget the format string it will be empty, and this means, no column will be displayed.

Lodging a command in a compound column is possible (of course!)

All you have to do is to provide an item ID and a persistent command and handle the `ItemID` in the event-handler (as demonstrated in section *DeleteMakeHomeless revisited for the web*).

Your compound column definition will look like this:

```
<remotion:BocCompoundColumnDefinition ColumnTitle="Name" ItemID="CompoundEdit"
          FormatString="{0}, {1}">
          <propertypathbindings>
            <remotion:PropertyPathBinding PropertyPathIdentifier="Surname" />
            <remotion:PropertyPathBinding PropertyPathIdentifier="FirstName" />
          </propertypathbindings>
          <persistedcommand>
            <remotion:BocListItemCommand Type="Event"/>
```

```
        </persistedcommand>
      </remotion:BocCompoundColumnDefinition>
```

Your event-handler will look like this:

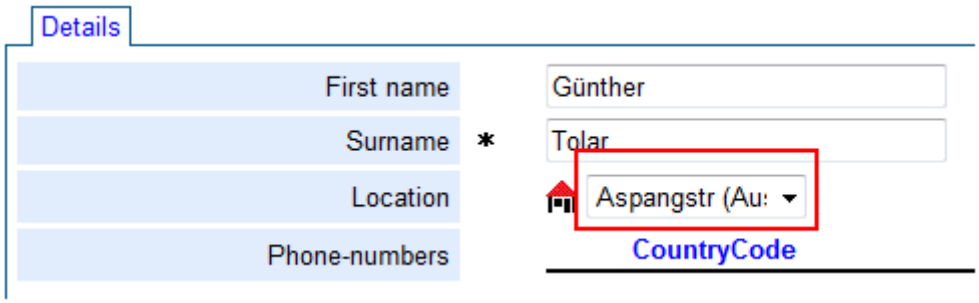
```
protected void LocationList_ListItemCommandClick (object sender,
BocListItemCommandClickEventArgs e)
    {
      if (e.Column.ItemID == "Edit" || e.Column.ItemID == "LeftColumnEdit")
      {
        try
        {
        // ... etc. etc.
```

The options menu for reference properties

`uigen.exe` turns the `Person` class'

```
public virtual Location Location { get; set; }
```

property into a drop-down list for picking one among ALL available properties:

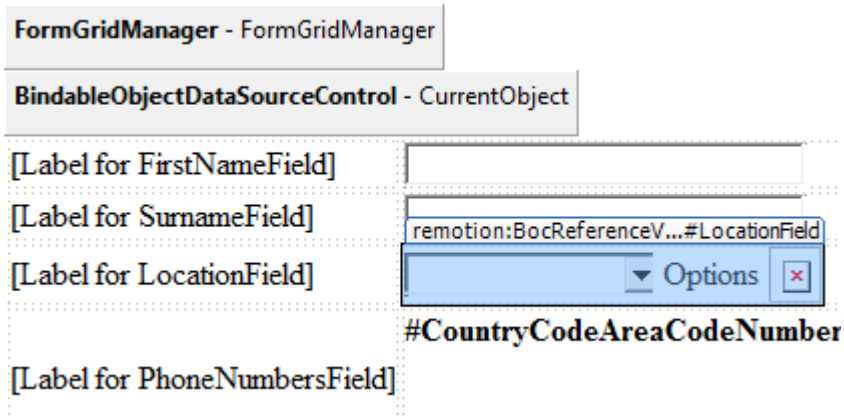


This is useful, but not practical if you

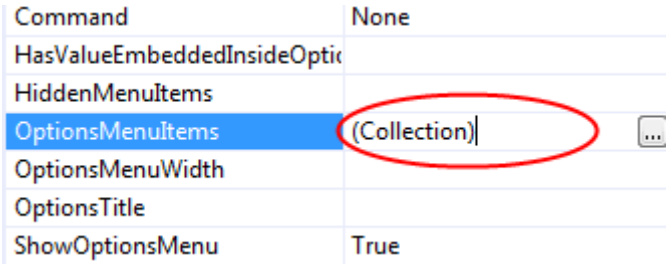
- want to create a new Location object
- have hundreds of Location objects (or billions)

For the first case the user can make a detour to the `EditLocationForm`, create a new `Location` object, return to the `PersonEditForm` and pick the newly minted `Location` from the drop-down list. For the second case there is no substitute, not even an impractical one like for the first case. For these reasons, re-bind provides a mechanism for practical shortcuts for the work with referenced domain objects -- the options menu. In practice it looks somewhat like this:

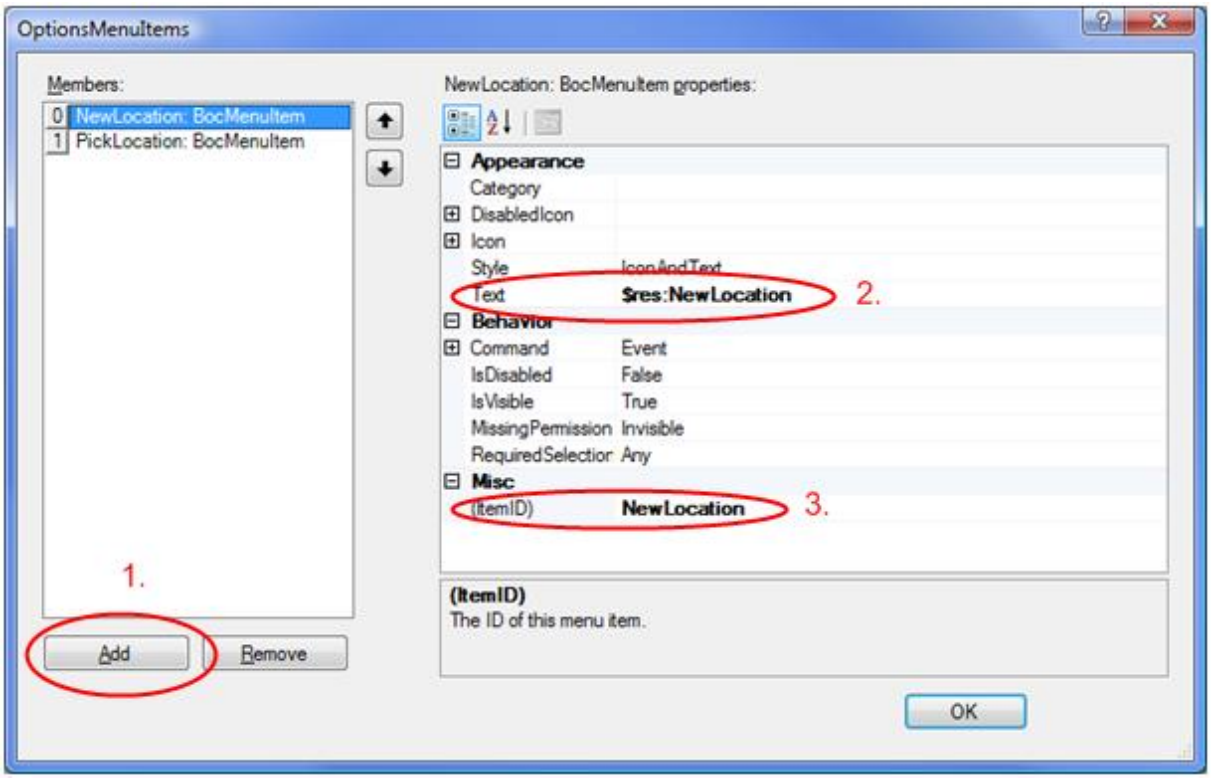
The menu and the menu items are part of the reference property's declaration, and you can create it in Visual Studio's designer:



If you click into the reference property for `Location`, the property explorer will show you the property's properties -- among them the list of options menu items:



As soon as you on the "..." for the collection, the design pane for that options menu opens:



With *Add* (1.) you add new menu items; in this case, our "NewLocation" and "PickLocation".

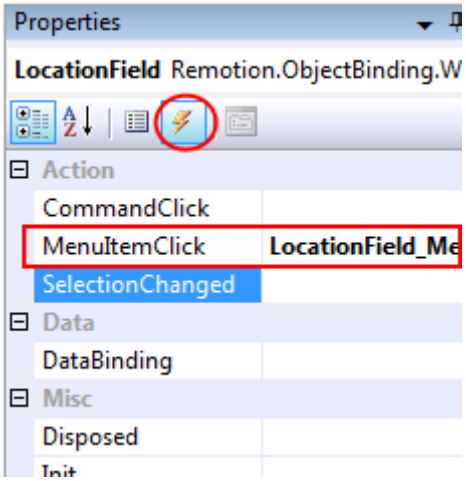
Each menu item should have a label text (2.) that shows up as caption of the menu item. The "$res:" signals to re-bind: "Take the caption from a named resource string". You specify that resource string

(*without* the "$res:", of course) in the `PhoneBook.Web` project's `Globalization\Global.resx` for English and `Global.de.resx` for German, as explained in section *Globalizing the PhoneBook web application*.

"New location..." in German is "Neuer Standort..."; "Pick a location..." in German is "Standort aussuchen...".

The itemID (3.) is the ID for telling the event handler (that we don't have yet) which of the menu items has been clicked. (Note that there is a single event handler for all items in a menu.)

What about the event handler? We don't have one at this time. Double-clicking on the control will give you a default event handler -- `OnSelectionChanged`. However, what we need is a handler for `OnMenuItemClick`. People familiar with ASP.NET will look no further than the  tab in the control's properties explorer, and there it is, the slot for OnMenuItemClick:



The canonical name for the method is assembled from the control and the name of the handler. In this case: `LocationField_MenuItemClick`.

Fleshing out the `LocationField_MenuItemClick` method will remain boring in this chapter, because this requires knowledge about re-call, and we won't get there before the next chapter. So for now, we have to make do with the skeleton:

```
protected void LocationField_MenuItemClick (object sender,
      Remotion.Web.UI.Controls.WebMenuItemClickEventArgs e)
{
  try
  {
    switch (e.Item.ItemID)
    {
      case "NewLocation":
        // Do nothing for now.
        break;
      case "PickLocation":
        // Do nothing for now.
        break;
    }
  }
  catch (WxeIgnorableException)
  {
```

```
      // fall back to here
    }
  }
```

The options menu as declaration

The complete listing for the "*NewLocation/PickLocation*" menu looks like this, and understanding it should pose no difficulty if you if you have grokked the previous re-bind sections:

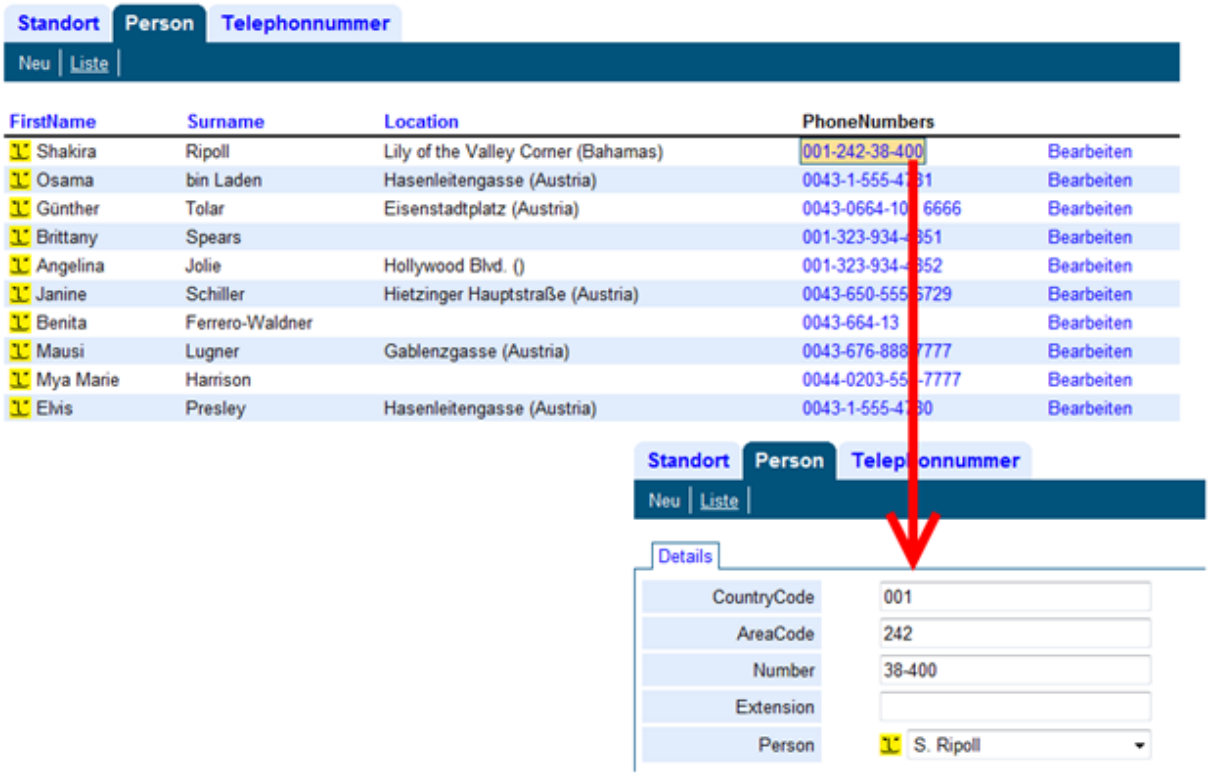
```
<td>
  <remotion:BocReferenceValue ID="LocationField"
                              runat="server"
                              DataSourceControl="CurrentObject"
                              PropertyIdentifier="Location"
                      OnMenuItemClick = "LocationField_MenuItemClick">
    <OptionsMenuItems>
      <remotion:BocMenuItem Text="$res:NewLocation"
                            ItemID="NewLocation">
      </remotion:BocMenuItem>
      <remotion:BocMenuItem Text="$res:PickLocation"
                            ItemID="PickLocation">
      </remotion:BocMenuItem>
    </OptionsMenuItems>
  </remotion:BocReferenceValue>
</td>
```

Not surprisingly, this listing embodies what we have entered in designer: ItemIDs, Text (Caption) and the all-important `OnMenuItemClick` event handler.

Phone-numbers as links, part one

In this exercise we will demonstrate how to use a `BocCustomColumnDefinition`. The "Custom" means "custom *rendering*", and the big advantage here is that we have full programmer control over the appearance of the elements in a custom column, and, by extension, what they do. The bad news is: we *have* to program everything the elements are supposed to do ourselves.

We will develop the new skill along the lines of clickable phone-numbers in the `SearchResultPersonForm`. A click on a phone-number in the (custom) phone-number column takes you right to the `EditPhoneNumberForm`. Here is a cartoon to illustrate what we mean by that:



Why not use a `BocSimpleColumnDefinition` and get this clickability-behavior for free? Keep in mind that a person can have more than one phone-number, but this clickability-behavior in a `BocSimpleColumnDefinition` assumes a single element in each "cell". (A cell is a single box where a given row and a given column intersect.) When using `BocSimpleColumnDefinition`, only the first phone-number is rendered as a link. Consequently, our improvement must render more than one element as a *link* and connect that link to an event-handler.

Since this mechanics requires some knowledge of the next chapter - re-call - we won't fully implement the desired feature at this time. We will limit our ambition to rendering some links to show how to work with `BocCustomColumnDefinition`s for now, and implement the rest of the exercise later, as soon as we have seen how re-call works. In other words: in this chapter we will have links, but those links won't do much at first.

On `BocCustomColumnDefintion`s

If you want to have custom columns in a BocList, you actually implement the behavior of a single `BocCustomColumnDefinitionCell`, or, a subclass of `BocCustomColumnDefinitionCell`. This class comes with a *Render* method that resembles that in regular ASP.NET controls. However, `BocCustomColumnDefintionCell` is not related to any ASP.NET control. It is derived from System.Object, and sports some methods with names borrowed from `System.Web.UI.Control`. So what a programmer must do for customizing is implement the `Render` method and connect control elements (such as links) to event-handlers.

The ASP.NET-declaration of `<remotion:BocCustomColumnDefinition>` understands a special attribute named `CustomCellArgument`. The value of this attribute is a string representing a set of key-value pairs, just like a hash-table or configuration file. This is useful for configuring the `BocCustomColumnDefinitionCell`, and we will make good use of it. What every

<remotion:BocCustomColumnDefinition> node also needs is a specification of the type (class) that implements its behavior. The CustomCellType attribute takes care of that. So here is the declaration that we will use in this exercise:

```
<remotion:BocCustomColumnDefinition
                CustomCellArgument="MaxItems=3,Commit=true"
                CustomCellType="PhoneBook.Web::Classes.PhoneNumberColumn"
                PropertyPathIdentifier="PhoneNumbers" />
```

The CustomCellArgument of "MaxItems=3,Commit=true" is our custom configuration for the BocCustomColumnDefinitionCell

- MaxItems specifies the maximum number of phone-numbers to be displayed in the cell.
- Commit specifies whether the custom column shall perform a Commit() or not. (The reason for this will be explained at the end of this section. It won't contribute much to the story for now.)

Such pairs of keys and values are mapped to properties in the BocCustomColumnDefinitionCell

- "MaxItems" will have its "3" stored in a property named... MaxItems – if declared
- "Commit" will have its "true" stored in a property named... Commit – if declared

The operative word here is "if declared". If you make a typo or forget that declaration, nothing will happen. You will fail without being warned.

That said, let's turn to the actual code, i.e. the Render method for now.

By tradition, the code-behind for any type (class) that supplements behavior is located in the Classes sub-folder, and this application is no different. So: add a new class file named PhoneNumber.cs to PhoneBook.Web\Classes.

You will need the following namespaces:

```
using System;
using System.Collections;
using System.Web;
using System.Web.UI;
using PhoneBook.Domain;
using PhoneBook.Web.UI;
using Remotion.Data.DomainObjects;
using Remotion.Data.DomainObjects.ObjectBinding;
using Remotion.ObjectBinding.Web.UI.Controls;
using Remotion.Web.ExecutionEngine;
```

Visual Studio will fill in the right namespace for the class for you, that is, PhoneBook.Web.Classes. It will also declare the class-name PhoneNumberCell for you. What you will be seeing is this:

```
namespace PhoneBook.Web.Classes
{
  public class PhoneNumberCell
  {
  }
}
```

All you have to do is to see to it that `PhoneNumberCell` is derived from `BocCustomColumnDefinitionCell`:

```
namespace PhoneBook.Web.Classes
{
  public class PhoneNumberCell : BocCustomColumnDefinitionCell
  {
  }
}
```

Next, add the aforementioned properties, so that the values from the `CustomCellArgument` have a home:

```
namespace PhoneBook.Web.Classes
{
  public class PhoneNumberCell : BocCustomColumnDefinitionCell
  {
    public int MaxPhoneNumbers { get; set; }
    public bool Commit { get; set; }
  }
}
```

Again: Make sure you get the spelling exactly as in the `CustomCellArgument` attribute of the `<remotion:BocCustomColumnDefinition>` node. If you don't, your program won't work without the compiler complaining.

This was the easy part. The bread and butter of this class is the `Render` method, of course. Most of it is conventional re-bind programming, and should look familiar from the previous BOC-examples in this section:

- finding out the property path the column is associated with
- getting the *value* of the business object in the cell
- iterating over the resulting list of phone-numbers

Here it comes in code:

```
protected override void Render (HtmlTextWriter writer,
BocCustomCellRenderArguments arguments)
    {
      var propertyPath = arguments.ColumnDefinition.GetPropertyPath();
      var phoneNumbers = (IList) propertyPath.GetValue
(arguments.BusinessObject, false, true);
      for (int i = 0; i < MaxPhoneNumbers && i < phoneNumbers.Count; ++i)
      {
      }
    }
```

ASP.NET-programmers will recognize this as a `Render` method typical for ASP.NET controls. However, don't be misled: `BocCustomColumnDefinitionCell` is *not* an ASP.NET control in the sense that it is *derived* from `System.Web.UI.Control`. The `Render` method is named `Render` to make ASP.NET-programmers feel good about it.

Getting the property path for the column definition is achieved in the first line of the `Render` method. The column definition is passed with the `arguments` argument of the `Render` method. The column definition in `arguments` is asked to which property of the business (domain) object instance in the cell it binds:

```
var propertyPath = arguments.ColumnDefinition.GetPropertyPath();
```

The next line uses this property path to retrieve the actual value of that property in the instance. This instance is also stored in the `arguments`, or in its `BusinessObject` property, to be exact. In this case, the business object is the `ObjectList<PhoneNumber> PhoneNumbers` property of the `Person` object in that particular row in the BocList. Since the value is not typed in the `BusinessObject`, we must cast it to what it actually is: a list (or better: its `IList` interface):

```
var phoneNumbers = (IList) propertyPath.GetValue (arguments.BusinessObject,
false, true);
```

In two steps, we have retrieved the list of phone-numbers to display in the given cell for that `Render` is the method. Now the hard part: we must render each of the phone-numbers in the list as a link. This requires some lengthy explanation. (Just what you were afraid of.)

A naive implementation might assume that we can simply link to the desired `EditPhoneNumberForm`, for example:

`http://localhost/UI/EditPhoneNumberForm.aspx&objid=c0ffee-31337-...`

This *could* work, although it is more effort than you might think. What's more, this bypasses our nifty re-call engine entirely, because re-call won't be able to intercept such a link. Such a mechanism does not post back, so no event-handler can be invoked on the given page.

Instead, the way to go is this: the link causes a postback that tells our program what event-handler to invoke and which phone-number has been clicked. Then the event-handler calls the `EditPhoneNumberForm` as a re-call function and passes the object's id (GUID) as a parameter.

Assembling the link in such a fashion that it encodes the event-handler and the object id is not hard, but good engineering requires that no assumptions are made on how exactly these components shall be rendered. This must be left to the framework and the objects themselves. Let's look at the code first, then at a sample rendered link before a thorough discussion. The rendering code inside the phone-number loop looks like this:

```
for (int i = 0; i < MaxPhoneNumbers && i < phoneNumbers.Count; ++i)
      {
        var phoneNumber = (BindableDomainObject) phoneNumbers[i];
        string renderedLink = String.Format ("<a href=\"#\"
onclick=\"{0}\">{1}</a>",
                                             GetPostBackClientEvent
(phoneNumber.ID.ToString()),
                                             HttpUtility.HtmlEncode
(phoneNumber.DisplayName));
        writer.Write (renderedLink);
        writer.Write ("<br>");
```

```
    }
```

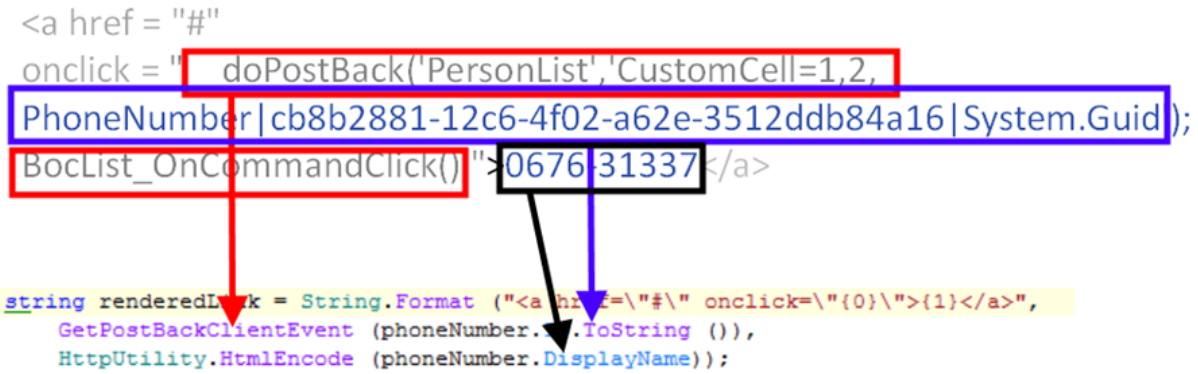
In the first line, we retrieve the next phone-number from the list of phone-numbers. For rendering the link, we must find out how the framework wants its postback-request rendered, i.e. what the javascript-invocation looks like. This could be hardcoded by us, but what if the framework developers change the name of the javascript-function? Or the order of parameters? Asking the framework how to format a postback request is the job of `GetPostBackClientEvent`. This method, too, is old news for ASP.NET programmers. So old, in fact, that the method of this name has run out of fashion in ASP.NET 2.0. `GetPostBackClientEvent` of `Page` used to have the same purpose in ASP.NET 1.0. re-bind, however, still sports the `GetPostBackClientEvent` method.

What `GetPostBackClientEvent` needs is an argument for the event-handler, that is, what shall be passed as `EventArguments`. Consequently, we ask the given `PhoneNumber` object for a string representation of its ID. This ID is more variable than you might think. The structure and format of an object-ID is not fixed, i.e. some classes can have their own scheme of identifying objects and encoding that as a string. For this reason, the `Render` method must leave it to the object entirely how it is identified by a string.

The string resulting from using `GetPostBackClientEvent` is embedded into a regular `<a href>` tag. This tag also has an `OnClick`-attribute specifying what shall happen upon a click – this is the part where we need the complicated assemblage of the link.

Displaying the phone-number in the inner text of the `<a href></a>` is comparably comprehensible. The `HttpUtility.HtmlEncode` invocation for the phone-number's `DisplayName` ("0555-8888") is required for security. Observe that this is text entered by a user, and a malicious user could inject anti-helpful javascript-code into the system if it was rendered unchecked. HTML-encoding prevents such an outcome.

If your head is spinning from this single line of code, don't be sad. That's quite normal at this point, so here is a walk-thru for a single sample link as rendered by `Render`:



The part rendered black is the easy part: 0676-31337 simply is the display name of the given phone-number.

The part rendered in blue -

```
PhoneNumber|cb8b2881-12c6-4f02-a62e-3512ddb84a16|System.Guid -
```

is the phone-number's object-ID rendered as string, using re-store's default implementation of object identification.

The remaining two parts are rendered by `GetPostBackClientEvent` and embrace the phone-number's object-ID. As you can see, the name of the javascript function causing the postback is named `_doPostBack`, and it needs the following parameters here (in that order):

- the name of the BocList that has caused the postback
- the coordinates of the cell that has caused the postback
- the aforementioned object id that has been clicked
- the name of the event-handler that has *received* the click first (`BocList_OnCommandClick`, because the click was processed by the BocList first

Speaking of the event-handler, you must implement at least the skeleton for it, so that you won't get an error. As explained above, we will postpone the actual implementation to the next chapter. This is all we need for now.

```
protected override void OnClick (BocCustomCellClickArguments arguments,
string eventArgument)
    {

    }
```

If you type in this sample code, it should work as advertised and display phone-numbers rendered as links. However, since we don't have an event-handler yet, nothing will happen, except for a brief flash upon a click – that's the postback at work, but since the event-handler does not do anything, we return to the same page as before.

So what about the `Commit=true`?

Good programmers make sure their code can be re-used, and this is also the goal of this `PhoneNumberCell` implementation. It can be used not only in the given BocList in the `SearchResultPersonForm`, but in any form with such a list. However, here is a benign problem.

A `SearchResultForm` never commits anything (it has no *Save*-button). It is conceivable, however, that the `PhoneNumberCell` is embedded in a BocList in a form that *does* commit. Now let's say we click on a phone-number, travel to the `EditPhoneNumberForm` and make a change (supply an extension, for example). We click *Save* to commit that change – to the parent transaction, not to the database. What happens next depends on where we return to: - if we return to a `SearchResultPersonForm`, the change will never be commited to the database, because a `SearchResultPersonForm` does not do that. If we return to, let's say, some `EditForm`, than we can rest assured that our change to phone-number will be committed as soon as the user commits the data of the parent transaction of that `EditForm`. This question flies in the face of reusability, because our `PhoneNumberCell` cannot know whether it is responsible for committing to the database (when called from a `SearchResultForm` or not (when called from an `EditForm`). However, we, the programmers *do* know whether the `PhoneNumberCell` is responsible for committing or not. So all we need is a mechanism for telling the `PhoneNumberCell` what to do –

a job for a `CustomCellArgument`! For this reason, we have introduced the `Commit` parameter in `CustomCellArgument`. In our case we set it to `true`, so that the event-handler will commit the changes by the invoked `EditPhoneNumberForm`. For the use in an `EditForm` that *does* commit, we simply set the `Commit=false`.

The virtues of re-call

Typical web-sites, even applications, are more content-driven than object- or form-driven. ASP.NET definitely is made for the former type, whereas re-call has evolved to serve as an important component for the latter type. So even on the fairly evolved ASP.NET 2.0 platform, several deficiencies remain for form-driven development. The most important one can be summarized under the head-line "retro-programming": programming transitions and managing data between pages resembles BASIC-programming for 8-bit computers: no sub-routines, no local variables, just GOTO and global variables. What do we mean by that?

ASP.NET programming beyond GOTOs

The typical way of programming a transition to another page works like this in ASP.NET:

- user clicks button (or some other control element)
- button fires event
- event-handler causes `Server.Transfer` or `Response.Redirect` to some specific page (a `GOTO`)

This logic and `GOTOs` CAN be appropriate and work well for both users and programmers, but this simplistic approach is not quite adequate for complex domain object management systems like the ones we build with re-motion. In re-motion applications, the user wants to navigate through piles of objects with many relations among themselves, often in a hierarchical fashion. "A location has many persons, a person can have one or more phone-numbers" is a simple, but typical example for it. Users working with domain objects want to take spontaneous detours, zoom into details, open containers to inspect contained objects – *always with the option of returning to where they branched off*. `GOTOs` are not very helpful with such requirements at hand. The disadvantages are firmly entrenched in programmer folklore and probably don't need any mention, but here they are, just for completeness, and adapted to navigation in form-inspired web-programming:

- `GOTOs` make it hard to re-use code across applications or within an application. If you program a transition to another page, the code returning to the original page must keep track of where it came from. Such a process is automated in procedural programming. The function returns to the caller upon completion.
- `GOTOs` make it hard for the programmer to keep a mental model of how code flows, or how pages are invoked. (That's the essence of the derogatory term "spaghetti code".)
- `GOTOs` are the natural enemy of nested transaction, as used by re-motion. If your code fails to `GOTO` back where it came from, you might end up with a dangling transaction.

Another resemblance to vintage home-computers and retro-programming is ASP.NET's way of keeping state: if you use `ViewState` you have to give up type-checking, and separating *page-local* from *application-global* data is impossible in the `ViewState`, for example. What's worse is that these ASP.NET-provided buckets are hashes of .NET `objects`. Wouldn't it be nice to have page-local *typed* variables, complete with automatic removal after the page has returned? So that data belonging to a page can actually be tucked on the page and is automatically removed as soon as the page has "returned"?

re-call can give you a mechanism for this, too.

re-call wraps one page into a neat package appropriately named a "re-call function", complete with

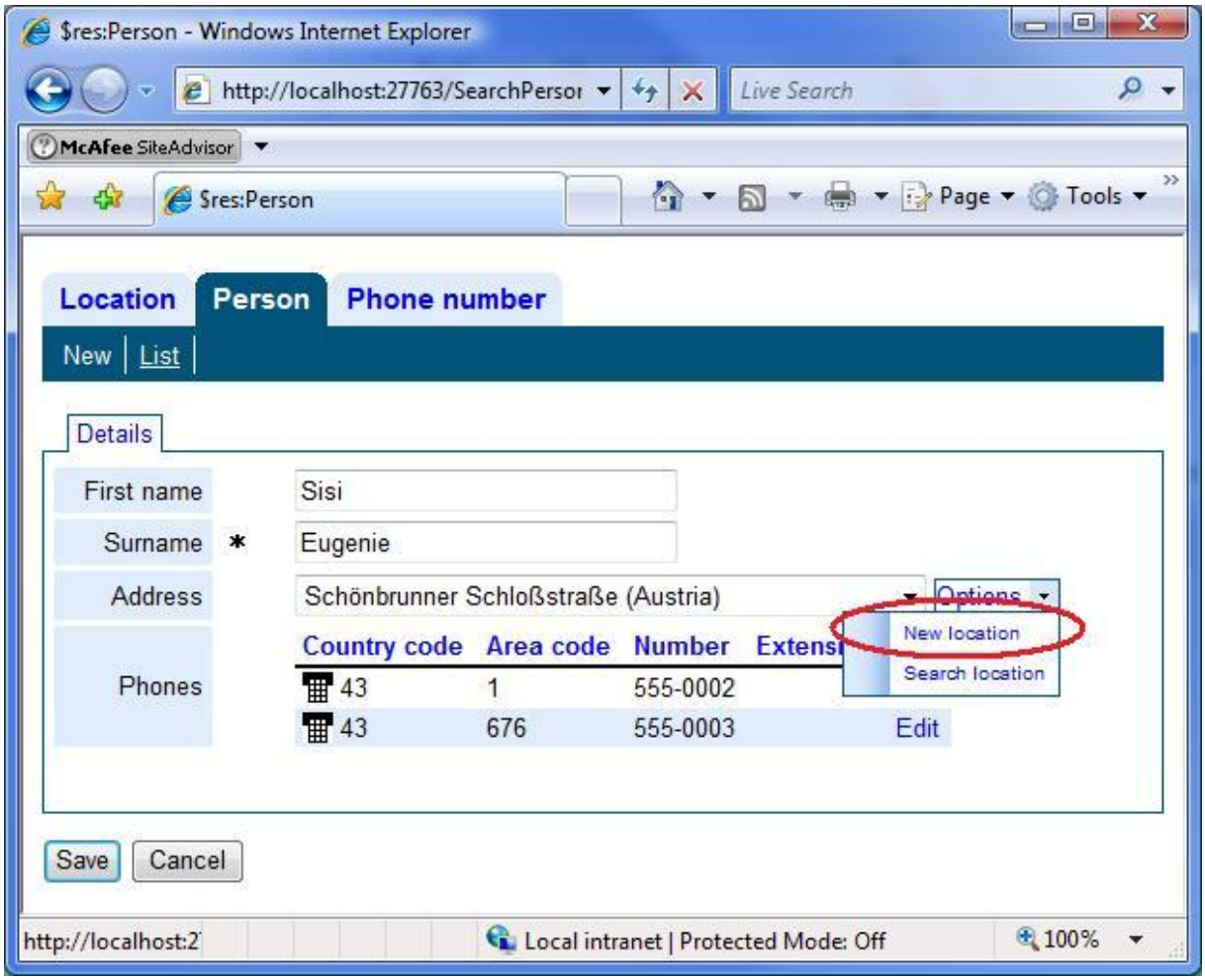
- a re-call page-stack for stacking nested re-call function invocations
- typed local variables
- a typed return value
- the ability to throw exceptions that can be caught in one of the scopes surrounding the function invocation
- nice integration with re-store's transactions

In other words, you can use a re-call page/"function" in an intuitive manner, given that you have evolved your skill beyond the steampunk programming of the Commodore C64 (what the author misses dearly, at times).

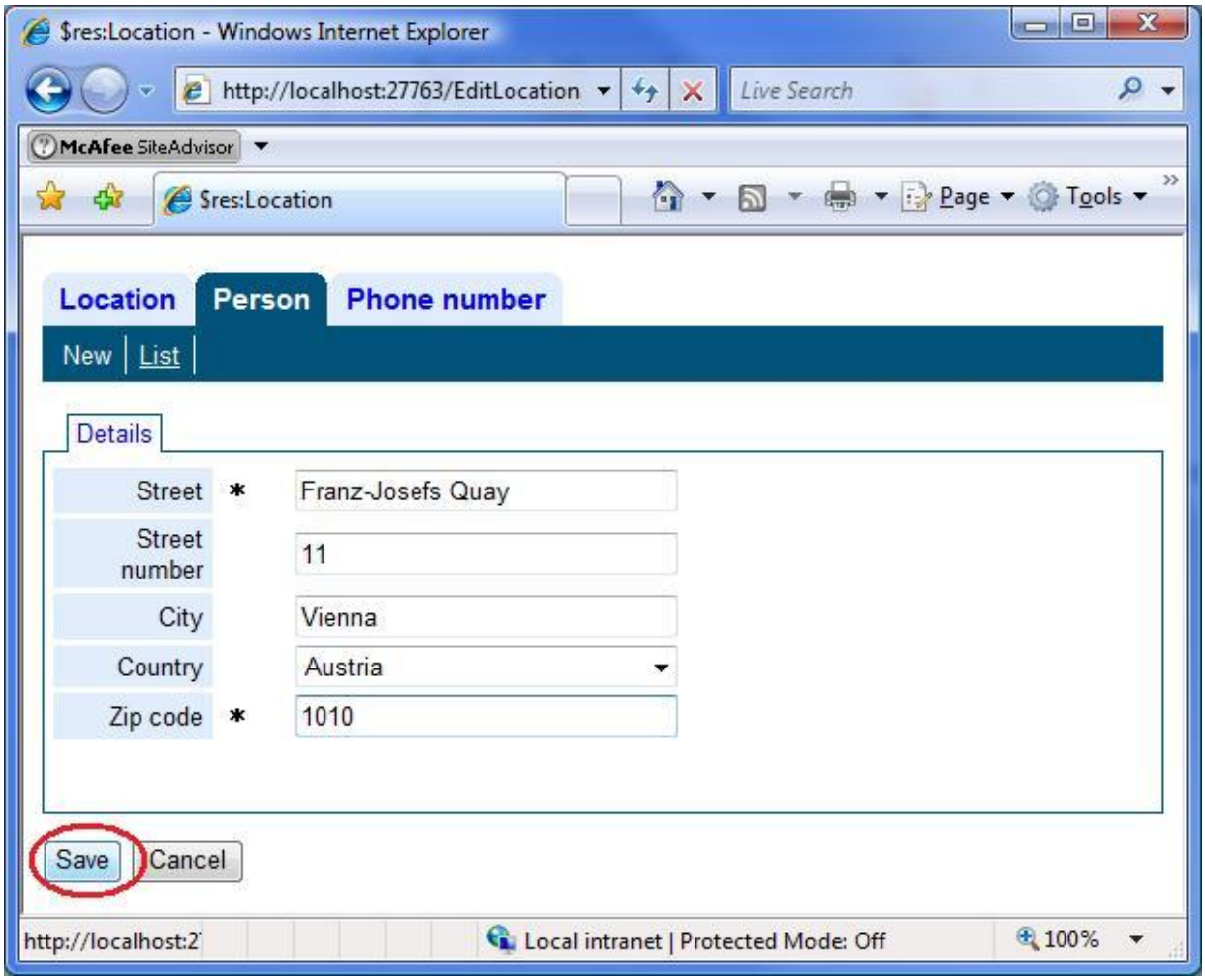
The virtues of re-call, illustrated

Here is an example of how re-call works in practice. We have introduced a menu item for the `Location` reference property in `Person`. So if you edit a `Person` object, you can take a detour to creating a new `Location` object for the `Person`. Here is a simple use-case.

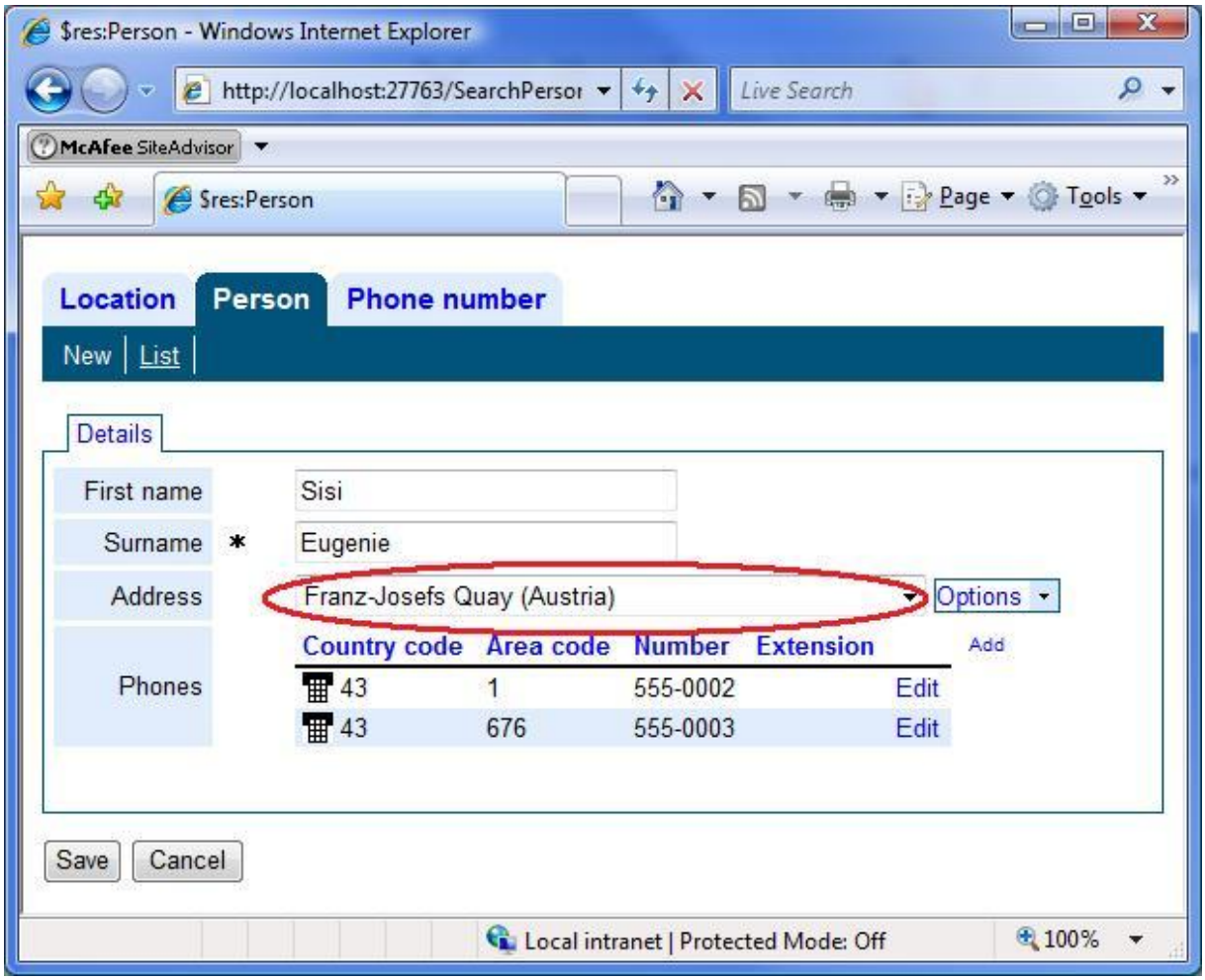
Let's say Sisi Eugenie, ever the party-girl, moves to a flat in the inner city to be closer to downtown night-life, and her new crash-pad is at the Franz-Josefs Quay 11, ironically named after her husband. So the user of the PhoneBook (Sisi's psychotherapist), wants to modify Sisi's home-address. The psychotherapist goes to Sisi's `Person` tab, selects the *New location...* menu item, gets the form/tab for a new location and enters the Franz-Josefs Quay address. After *Save* ing the `Location`, the psychotherapist returns to Sisi's `Person` page. In this example, the `Location` page is invoked as a re-call page-function. In other words, upon completion of this re-call "subroutine", the user (or the application) returns from where he deviated when she selected *New location...*. Here are some screen-shots to clarify the procedure:



Sisi has a new location



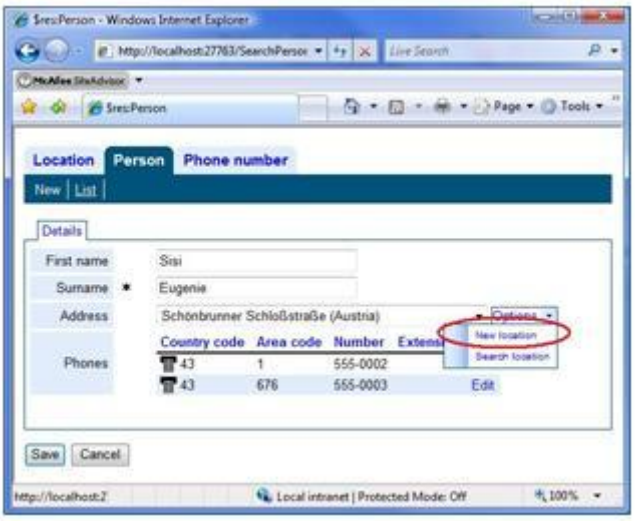
Now we are at the location page (function) -- downtown at the quay. As soon as we commit (click Save)...



... we gave t the empress a new address-- back to the EditPersonForm

The last screen-shot is probably the most interesting, because, as you see, we simply returned to the `EditPersonForm`, with the new location in the `Location` field already.

Another way to visualize what's going on is to think of the `EditLocationForm` as being put on a "form stack", thereby "obscuring" its parent window, `EditPersonForm`. In this way, working with re-call pages/functions becomes like working with modal dialogs. The same sequence again, this time with stacked windows.



Let's give Sisi a new home

The `EditLocationForm` *is "stacked" (obscured), so you can't see the* `EditPersonForm` *anymore*

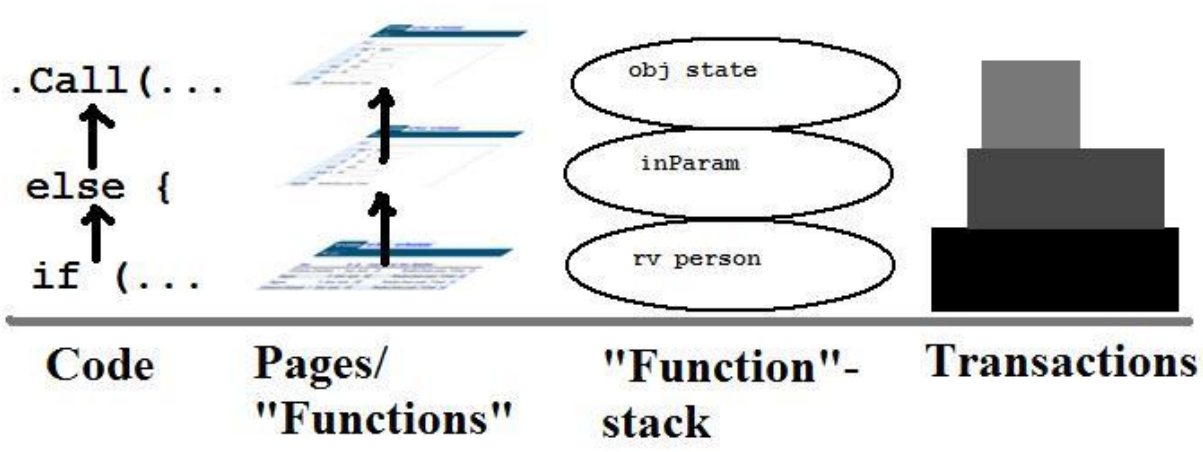
Back to the `EditPersonForm`*, the* `EditLocationForm` *falls prey to the garbage collector*

What can't be seen in the screen-shots is the transaction model at work behind the scenes. The two forms involved span a typical, albeit small unit of work with two domain objects -- a person and a phone-number. When displaying Sisi's `Person` page, re-call tacitly opens a root transaction for reading and writing Sisi's properties. When taking the detour to the page for entering the new `Location`, the application automatically creates a sub-transaction on top of that root transaction. When the psychiatrist clicks *Save* for the `Location` object, the Franz-Josefs Quai address is committed to *the root transaction*, not the database. Not before the user clicks the *Save* button on the underlying `Person` page the data is actually committed. If Sisi's shrink would instead *Cancel* her `Person` page, the new `Location` object would vanish without a trace (as explained in section FIXME).

The code to do all this is simple: Just invoke the `EditLocationForm` in the event-handler for the clicked *New location* menu item. Since the `EditLocationForm` page is also a function, it *returns the newly minted location object as a value*. Here is the (simplified) code fragment, a fleshed out event-handler for the *New location...* menu item from the last chapter:

```
// The event-handler
    public void LocationField_MenuItemClick(object sender,
                                            WebMenuItemClickEventArgs e)
    {
      try
      {
        // Do we handle the "New location" menu item?
        if (e.Item.ItemID == "NewLocation")
        {
          // That's the code. That's it: We CALL the EditLocationForm...
          var result = EditLocationForm.Call(WxePage, null);
          // ... and assign its TYPED return value to the "Person"'s
          // "Location" property.
          LocationField.Value = result;
        }
      }
      // We can even handle exceptions thrown by the EditLocationForm
      // function in a natural way!
      catch (WxeUserCancelException) { }
    }
```

The `EditLocationForm` does NOT contain any hard-coded logic to return to Sisi's `EditPersonForm` after the user is done with it. The `EditLocationForm` automatically returns to where it came from. This magic is achieved with a "return stack", and you probably know what a return stack is. What's more, you can mentally map this stack to the equivalent stack of "pages" illustrated above: not only is the spot where `EditLoctionForm` will return to kept on a stack, the call to it "obscures" the `EditPersonForm`, whose state is frozen underneath the newly opened `EditLocationForm`. Along similar lines, re-call opens a new sub-transaction for the `EditLocationForm` which has the transaction for `EditPersonForm` as parent transaction. Here is the illustration from the FIXME chapter again to show this concept of corresponding stack frames, EditXForms and sub-transactions:



What can't be seen in the simple example with Sisi's psychotherapist: Both `EditPersonForm` and `EditLocationForm` can also contain typed local variables that keep their values across requests of the same page.

Since local variables are also kept on the stack, just like return addresses, both regular functions (Pascal-functions, for example) and re-call functions come with the added benefit that it is always clear what is not needed anymore and vanishes automatically.

We will return to the topic of page-local variables in section FIXME.

This means you don't have to explicitly throw away page-local data or commit modifications upon completion of a page/function like `EditLocationForm`. As soon as the entire stack has been carried off by returning from the bottom-most "function", the entire root transaction is committed automatically (if so-called auto-commit is enabled, which is what uigen.exe generates).

More re-call features

ASP.NET gives you many options for parameter-passing or transitions from page to page. You can opt for passing parameters in URLs, for example, and you can use `Server.Transfer` or `Response.Redirect` for moving to another page. However, each of the choices comes with advantages and disadvantages. re-call also gives you control over parameter-passing, time-outs and whether a new page opens in another browser window or in the same browser window.

Timeouts and aborts

In terms of timeouts, ASP.NET is rather poor -- only session timeouts are supported. re-call supports those as well, but in addition it also gives you

- timeouts for a single page before the return to the page that spawned it (configurable, default is 20 minutes)
- a refresh interval for no timeouts (i.e. the timeout clock is reset every 10 minutes)

What's more, if a page/function is called in an external window, and a user goes to another page altogether or closes that window, re-call sees that as a *Cancel* of the corresponding function.

Out-of-sequence postbacks

If you have played with your browser's back-, forward- and refresh-buttons in your phone-book application you will have noticed that re-motion (or re-call, to be more precise here) gives you a slap

on the fingers for that, in the form of a message "Navigating this page by using the Forward and Back buttons is not supported". This has been explained already in section FIXME, but here is a reminder in the context of re-call: re-call DOES support out-of-sequence postbacks, but they are currently not discussed in this tutorial. For a deeper discussion of this issue, see Appendix D.

About edit-forms and wxegen.exe

Understanding edit forms and edit controls

If you look at the `EditLocationForm.aspx*` family of files (in the `UI` sub-folder of the `PhoneBook.Web` project), you will discover a fairly straightforward set of ASP.NET files for displaying and managing a `Location` domain object's properties.

First and foremost, an edit form and its related edit control file is an ASP.NET page with controls, complete with code-behind files.

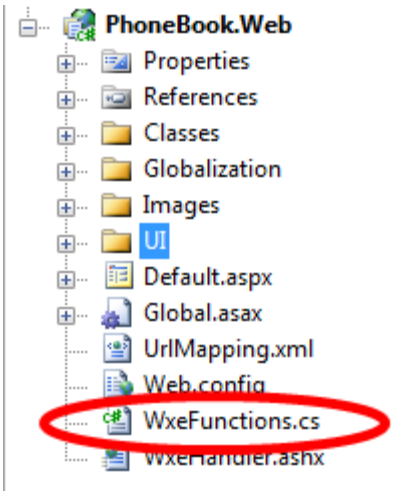
However, these forms are closely related to re-call and re-call's code-generator `wxegen.exe`. Every edit form also represents a re-call function, and its functional characteristics - parameters, local variables and return value - are declared as an XML-fragment on top of the page.

For the unedited (= default = as generated by `uigen.exe`) `EditLocationForm` this XML-fragment looks like this (see the top of `EditLocationForm.aspx.cs`):

```
// <WxePageFunction>
//   <Parameter name="obj" type="Location" />
// </WxePageFunction>
```

This snippet means: the annotated class (`EditLocationForm`) has one parameter named `obj` of type `Location`. (This object is the `Location` domain object to be edited in the form, of course.)

XML-comments like these are the input for the command-line tool `wxegen.exe`. It generates extensions to the classes annotated in such a fashion. The resulting code in a file named `wxefunctions.cs` (by default) is responsible for actually giving you the declared parameters as first-class .NET citizens. You might have noticed the file WxeFunctions.cs before in your project, because it is clearly visible in your PhoneBook.Web project:



This `wxegen.exe`-part (in `WxeFunctions.cs`) *extends* the annotated `EditLocationForm` class, which is declared `partial`. Besides the code for easy access of declared re-call-variables like parameters, local variables and return value (if any), `wxegen.exe` includes the following methods (and properties) in the classes' `partial` extensions:

- the all-important `Call` method(s) for invoking the page as function. This is a fairly sophisticated method in both use and structure.
- the `CurrentFunction` property contains the current `EditLocationFormFunction`, itself derived from `BaseFunction`. This `BaseFunction` class implements the code that gives the page its functional meat. For each edit form and search result form, a specialized sub-class of `BaseFunction` is generated and stored in `wxefunctions.cs`.

If you inspect the `Call` method(s) `wxegen.exe` has generated for this declaration, you'll see that `Call` is `void`, because the XML-annotation specifies no return value. There are two overloaded methods for `EditLocationForm` in `WxeFunctions.cs`:

```
public static void Call(Remotion.Web.ExecutionEngine.IWxePage currentPage,
Remotion.Web.ExecutionEngine.IWxeCallArguments arguments, Location obj)
```

and

```
public static void Call(Remotion.Web.ExecutionEngine.IWxePage currentPage,
Location obj)
```

The first one, the one with the extra `IWxeCallArguments arguments` argument, is the more sophisticated one, allowing control of how exactly the transition to the form/page/function shall work. We will turn to this `arguments` argument in section FIXME. For now, we will make do with the simpler method. The `currentPage` argument is the page from which the re-call function is invoked, so that it knows what its parent page is. The `obj` argument is the aforementioned `Location` domain object.

Both `Call` methods are `void`, because `uigen.exe` generates them this way. Be aware that many "defaults" or "conventions" explained here are not really re-call's defaults but uigen.exe's implementation details.

A deeper discussion on how to write your own XML-annotations for re-call-functions/pages/classes is given in section FIXME. Practical examples are presented in the next sections. As promised in section FIXME, we will flesh out the event handlers for the menu items *New location...* and *Search location...*.

Fleshing out *NewLocation...*

In a previous section, FIXME, we introduced how to use menu items and how to connect them to an event handler. Since actually doing something in the event handler required the material of this chapter, we postponed it to "a later exercise". Now this exercise is here. In this section, we will call `EditLocationForm.aspx` as a re-call function.

If we want to flesh out the handler for *New location...* as illustrated in the section FIXME, we have to do the following.

- modify the `EditLocationForm`'s `Call` methods in such a fashion that its return type is `Location` instead of `void` (which is generated by `uigen.exe`)
- supplement the call to the `EditLocationForm`/function in the `LocationField_MenuItemClick` handler: FIXME
- supplement some logic to the `EditLocationForm`'s `SaveButton_Click` handler so that it causes the form/functions return of the fresh object

Adapting the return type

Remember what's being said in section FIXME: The only parameter is `Location` obj – this is the Location domain object to be edited when the form is called. If `obj` is `null`, then a new object will be created. This modest interface is declared in the *re-call header*, an XML-snippet that annotates the `EditLocationForm` class in a comment. As generated by uigen.exe, this header looks like this:

This is the header for `EditLocationForm`:

```
// <WxeFunction>
//   <Parameter name="obj" type="Location" />
// </WxeFunction>
```

Only one *parameter* - `obj` - is specified. The `EditLocationForm` re-call function has no return value and no local variables.

For changing the `Call` method's return type, we have to add a `ReturnValue` node to the declaration in the XML-annotation, which is

```
// <ReturnValue type="Location" />
```

Consequently, the new XML-annotation in the comment to the EditLocationForm class shall look like this:

```
// <WxePageFunction>
//   <Parameter name="obj" type="Location" />
//   <ReturnValue type="Location" />
// </WxePageFunction>
public partial class EditLocationForm : EditFormPage
{
... etc ...
```

[Just for completeness: local variables are defined in `Variable` nodes, but for this exercise we don't need any.]

After adding the `<ReturnValue type="Location" />`, rebuild the `PhoneBook.Web` project.

`wxegen.exe`, as invoked in the `BuildEvents` property of the project, will automatically detect that `EditLocationForm.aspx.cs` has changed and regenerate `WxeFunctions.cs`. As pointed out in section FIXME, this method of generating code in the build-events has a problem. You will have to re-build the project TWICE, because the first time will give you errors. (We have met this glitch in section FIXME already.)

If you inspect the two `Call` methods in the `EditLocationForm` class (`UI\EditLocationForm.aspx.cs`) after the build, you will see that they are not `void` anymore. The new signatures are

```
public static
Location Call(Remotion.Web.ExecutionEngine.IWxePage currentPage,
       Remotion.Web.ExecutionEngine.IWxeCallArguments arguments,
                                                      Location obj)

public static
Location Call(Remotion.Web.ExecutionEngine.IWxePage currentPage,
                                                      Location obj)
```

What's more, you can see that `wxegen.exe` added another property for the return value to the `EditLocationForm` class:

```
public Location ReturnValue
{
  set
  {
    this.Variables["ReturnValue"] = value;
  }
}
```

The `Variables` member of `EditLocationForm` is a dictionary of .NET `objects`, just like ASP.NET's `ViewState`, for example. It stores all the page-local data, like parameters, local variables and, as in the snippet above, the page-function's return value. This is the technology mentioned in the section *ASP.NET programming beyond GOTOs:* TYPED local variables and TYPED return values.

We can use the `ReturnValue` property in the handler for `SaveButton_Click`. The unaltered `SaveButton_Click` handler, as generated by `uigen.exe`, looks like this (in `UI\EditLocationForm.aspx.cs`):

```
protected void SaveButton_Click (object sender, EventArgs e)
    {
      if (SaveObject())
      {
        Return();
      }
    }
```

However, what we want is that the `obj` is stored in the `ReturnValue` property, so modify it in the following fashion:

```
protected void SaveButton_Click (object sender, EventArgs e)
    {
      if (SaveObject())
      {
        ReturnValue = obj; // NEW!
        Return();
      }
    }
```

After supplementing the `ReturnValue = obj;` line, your `Call` function and the `EditLocationForm` "function" is good to go.

Actually calling the page/function

After fixing the return type of the `EditLocationForm`'s `Call` method, we have to use it for invoking the `EditLocationForm` as a re-call function.

To the `LocationField_MenuItemClick` handler in `UI\EditPersonControl.ascx.cs`, add the line

```
LocationField.Value = EditLocationForm.Call(WxePage, null);:
```

The augmented handler should look like this:

```
public void LocationField_MenuItemClick(object sender,
                                        WebMenuItemClickEventArgs e)
    {
      if (e.Item.ItemID == "NewLocation")
      {
        // NEW!
        LocationField.Value = EditLocationForm.Call(WxePage, null);
      }
      else if (e.Item.ItemID == "SearchLocation")
      {
      }
    }
```

The `WxePage` property in all `EditXControls` stores the page for the use in `Call` methods, so that the callee knows the parent page from which it was invoked.

If you rebuild the `PhoneBook.Web` project now, you should be able to actually use the new feature as advertised – unless you get the idea to cancel the `EditLocationForm`. We haven't told you about `WxeUserCancelException` and `WxeExecuteNextStepException` yet.

A problem re-call must deal with is the decision how far in the stack we want to drop if the user cancels the top-most form. The typical plan is to drop down to the form immediately below the canceled form – the canceled form is removed, together with its data and client transaction and the operation resumes with the parent.

Some situations might require more and deeper cleanup, so some flexibility is in order. A simple rule keeps things simple for programmers: A *Cancel* operation causes the canceled form to throw a `WxeUserCancelException`, and the form where this exception is caught remains on the stack, all forms above it are discarded.

In our case, *New location...*, the natural thing to do is to have the the `EditLocationForm`'s parent, `EditPersonForm`, catch the exception, so that only the canceled `EditLocationForm` is discarded.

To this end, wrap the `LocationField_MenuItemClick` handler code in a try-catch-block:

```
public void LocationField_MenuItemClick(object sender,
                                        WebMenuItemClickEventArgs e)
    {

        if (e.Item.ItemID == "NewLocation")
        {
```

```
          try
          {
            LocationField.Value = EditLocationForm.Call(WxePage, null);
          }
          catch (WxeUserCancelException) { }

        }
        else if (e.Item.ItemID == "SearchLocation")
        {
        }
    }
```

The `WxeUserCancelException` identifier requires another `using` in `UI\EditLocationControl.aspx.cs`: `using Remotion.Web.ExecutionEngine;`

After this minor improvement, you have a real-working invocation of an edit form as a re-call function. You can satisfy yourself that this works by trying it out.

If you *Save* some newly created `Location` object in `EditLocationForm`, it's display name should show up in the `Location` `BocReferenceValue` in `EditPersonForm`.

If you *Cancel* your modifications, nothing should happen, i.e. the `BocReferenceValue` should keep its original value.

Some observations and experiments

`Location` vs. `LocationField`

Note that we don't actually store the object in the `Person` object's `Location` property. We use `LocationField` instead.

Why is this so?

The reason is that controls reflect what's stored in the properties, but modifications by the user travel in the other direction.

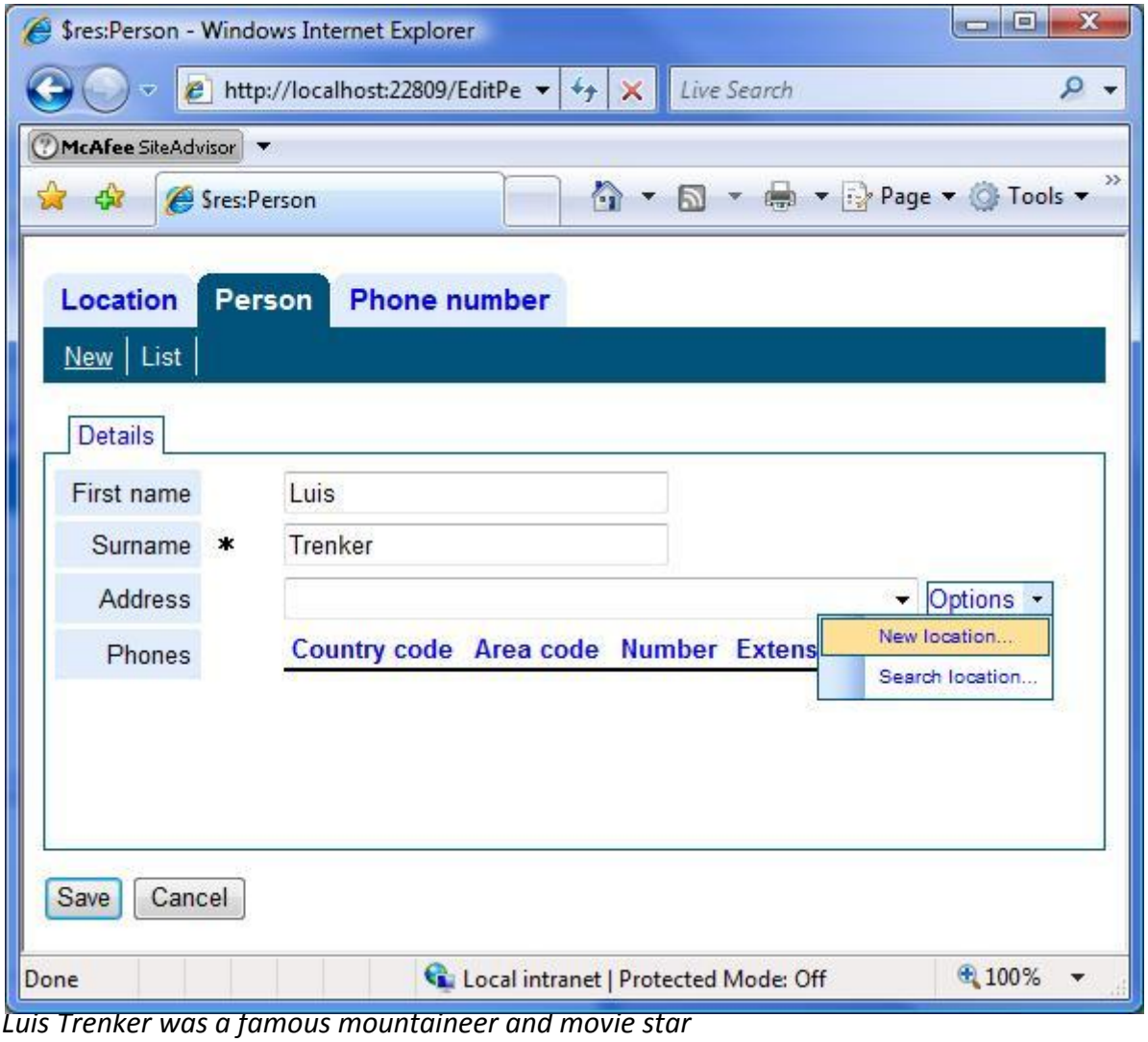
The control has no way of knowing that you sneaked data past it into the domain object. Without calling a function named `LoadValue`, the modification of a value in the domain object will not be displayed by the control. That's why we go the route in the opposite direction, because if a value is changed in a BOC control, the control takes care of updating the corresponding property in the domain object.

You CAN put the new `Location` object into the `Person`'s `Location` property directly, of course. All you have to do is walk your way from the `BocReferenceValue` to the actual property, via its `DataSourceControl` and `BusinessObject`:

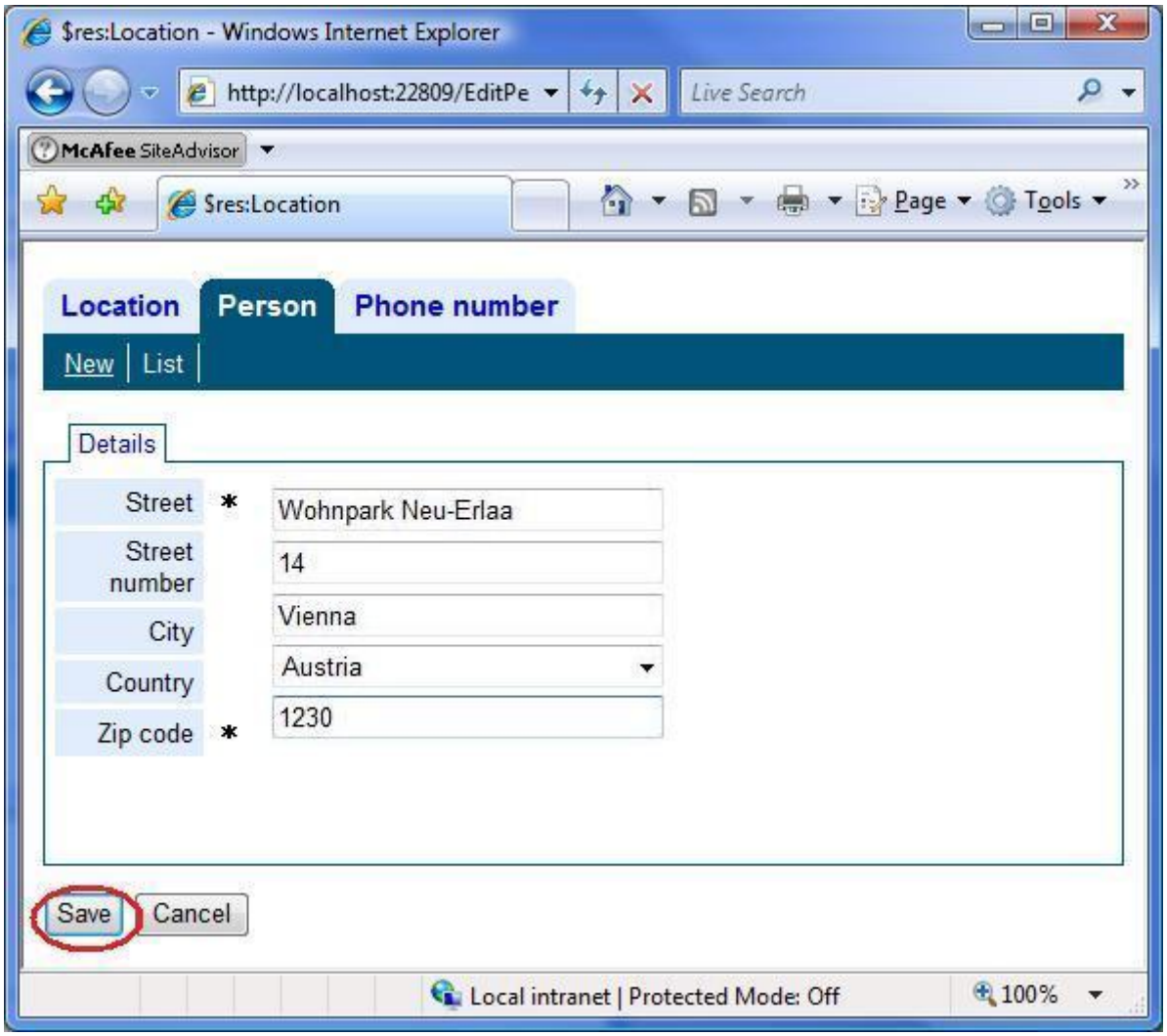
```
Person p = (Person)LocationField.DataSource.BusinessObject;
p.Location = EditLocationForm.Call(WxePage, null);
```

This even works. However, your modification will NOT show up immediately in the `LocationField` in your browser, although it's already in the property.

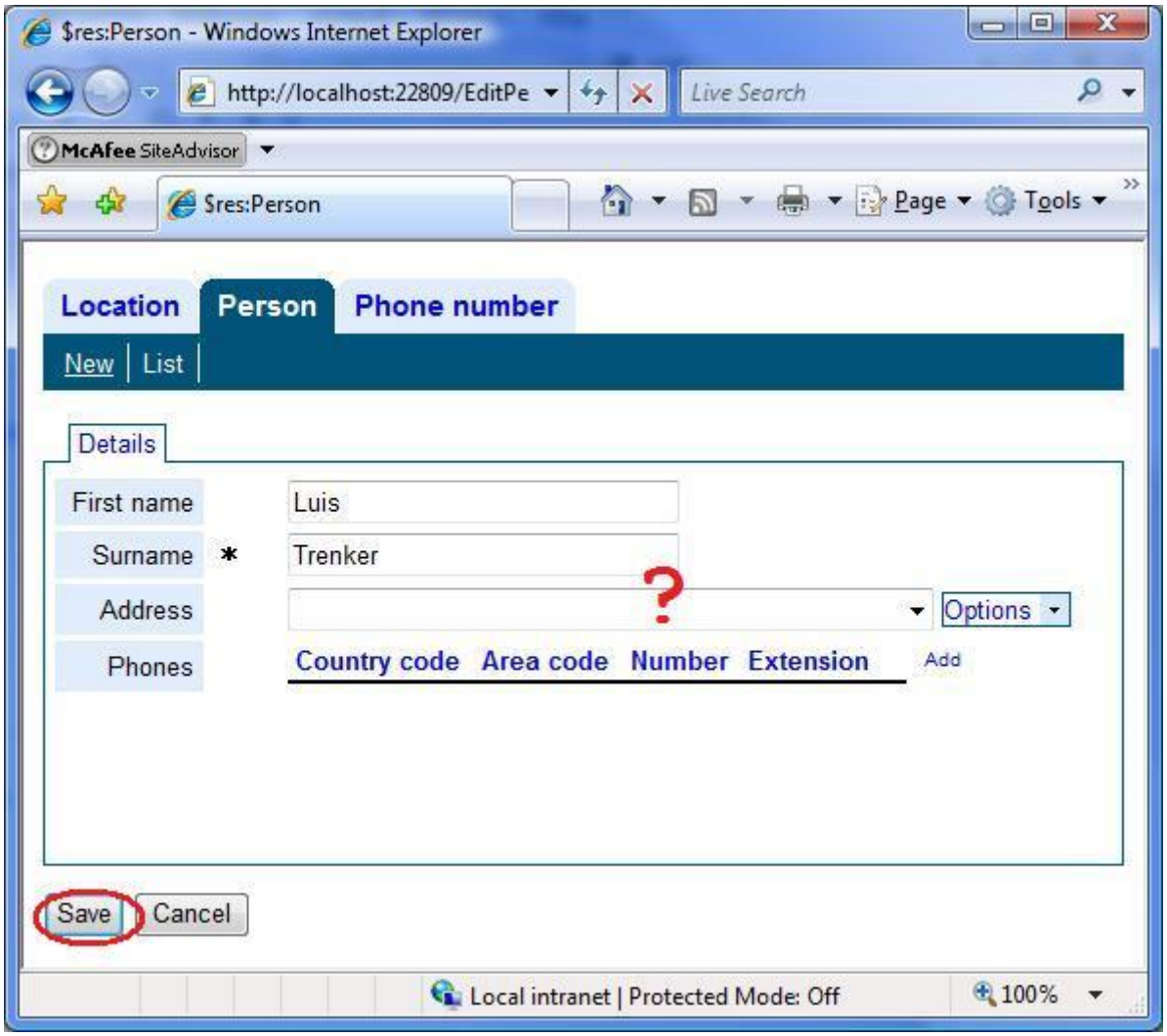
If you try out this code you WON'T see the `Location`'s `DisplayName` in the `BocReferenceControl` when you are done. This takes another post-back.



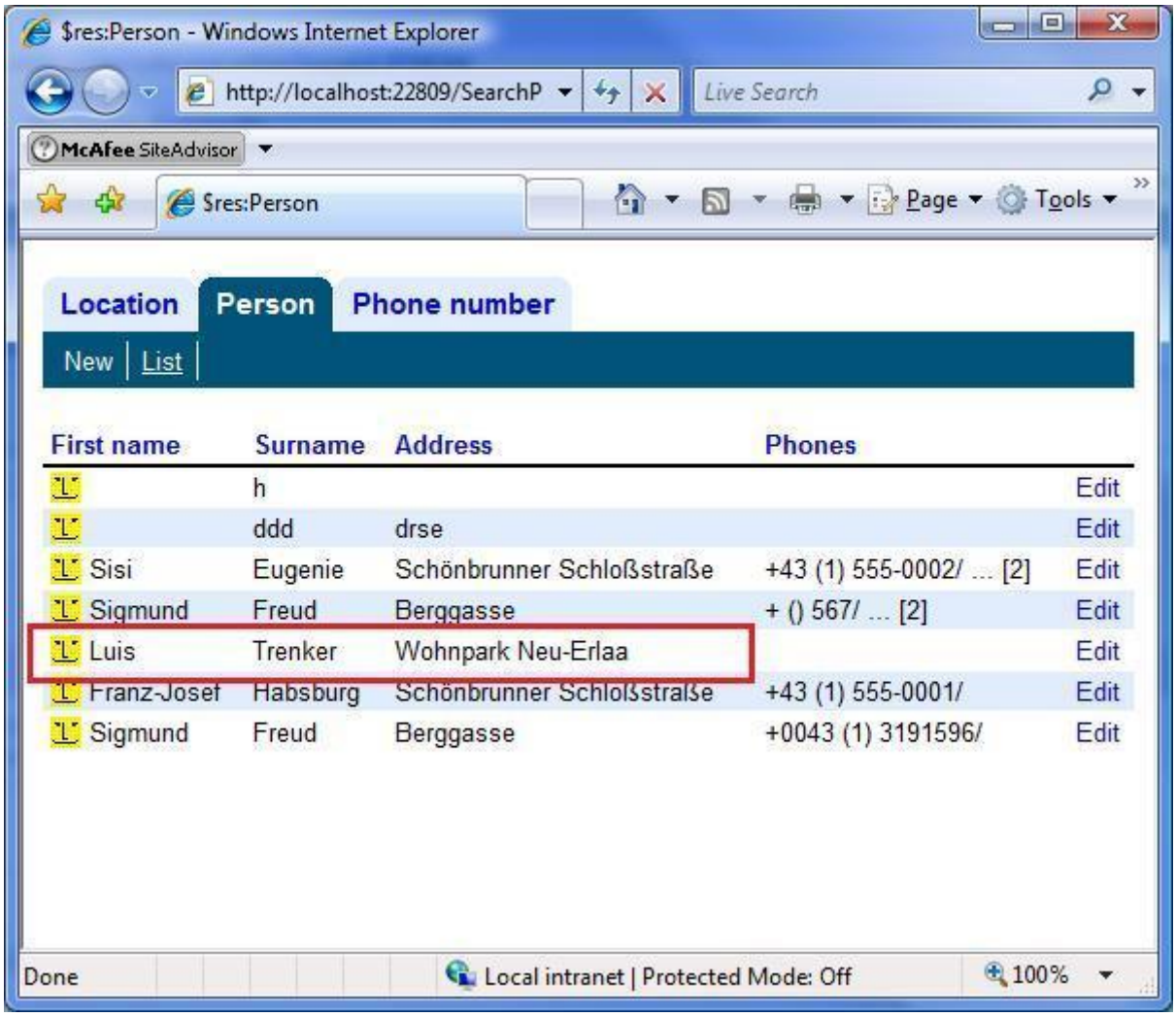
Luis Trenker was a famous mountaineer and movie star



This is probably the worst address in Vienna for a mountaineer, even worse for a movie star



What's worse: the address does not show up in the `BocReferenceValue`. Click SAVE anyway...



However: *the address has been committed*

Don't believe your lying eyes – introducing returning postbacks

People with some understanding of ASP.NET will notice something strange about the invocation of a page function. Here is the fleshed-out event-handler again:

```
public void LocationField_MenuItemClick(object sender,
WebMenuItemClickEventArgs e)
    {

        if (e.Item.ItemID == "NewLocation")
        {
          try
          {
            LocationField.Value = EditLocationForm.Call(WxePage, null);
          }
          catch (WxeUserCancelException) { }

        }
        else if (e.Item.ItemID == "SearchLocation")
        {
        }
    }
```

Note that this code requires that some is executed before the `EditLocationForm.Call` and some code that is executed *after* the call. Also observe that the called `EditLocationForm` is

another page, complete with its own `Server.Transfer` or `Response.Redirect`. As ASP.NET programmers are probably aware, no code written after a Server.Transfer or Response.Redirect is ever executed, because these page-transitions kills the thread of the request altogether. Consequently, the code to be executed *after* the call will be gone when the `EditLocationForm` has performed its duty, because it resides on a different page – the `EditPersonForm`. So if the `catch(WxeUserCancelException)` is supposed to actually catch the exception, the `EditPersonForm` must be executed *again* upon completion of the `EditLocationForm`, and it must resume at precisely the spot *after* the invocation of `EditLocationForm.Call`.

Otherwise the code for the catch-block would not even run. This observation applies to all code supposed to run after the invocation of `Call`.

In other words: the seamless resuming after the invocation is an illusion. In actuality, the page which embeds the event-handler is invoked (posted back to) twice. Once for running the event-handler code up to the page/functions invocation and once for throwing exceptions (if any) and running the code after the invocation (called a *returning postback*).

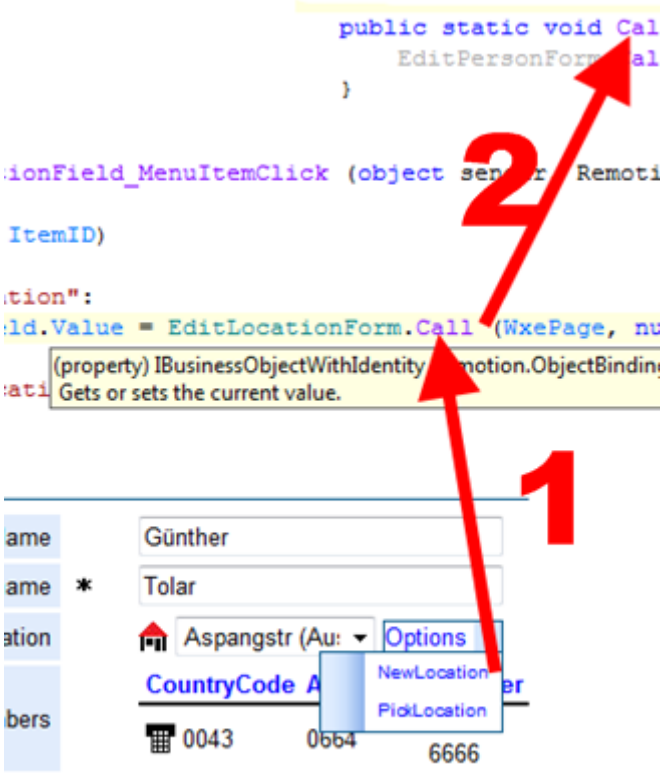
The good news is that this second invocation and resuming of the calling page is *almost* completely automated. The bad news is that it is *not quite* automated. What re-call does for you is to take care that the event-handler around the calling function is invoked twice.
The first time around, the `Call` method)

- determines which event-handler caused its invocation
- does what it is supposed to do (run the actual form, for example)
- catches any exception and remembers it for the second invocation of the event-handler
- invokes the event-handler from the calling page again, so that
  the code *after* the invocation can run

The second time around, the `Call` method

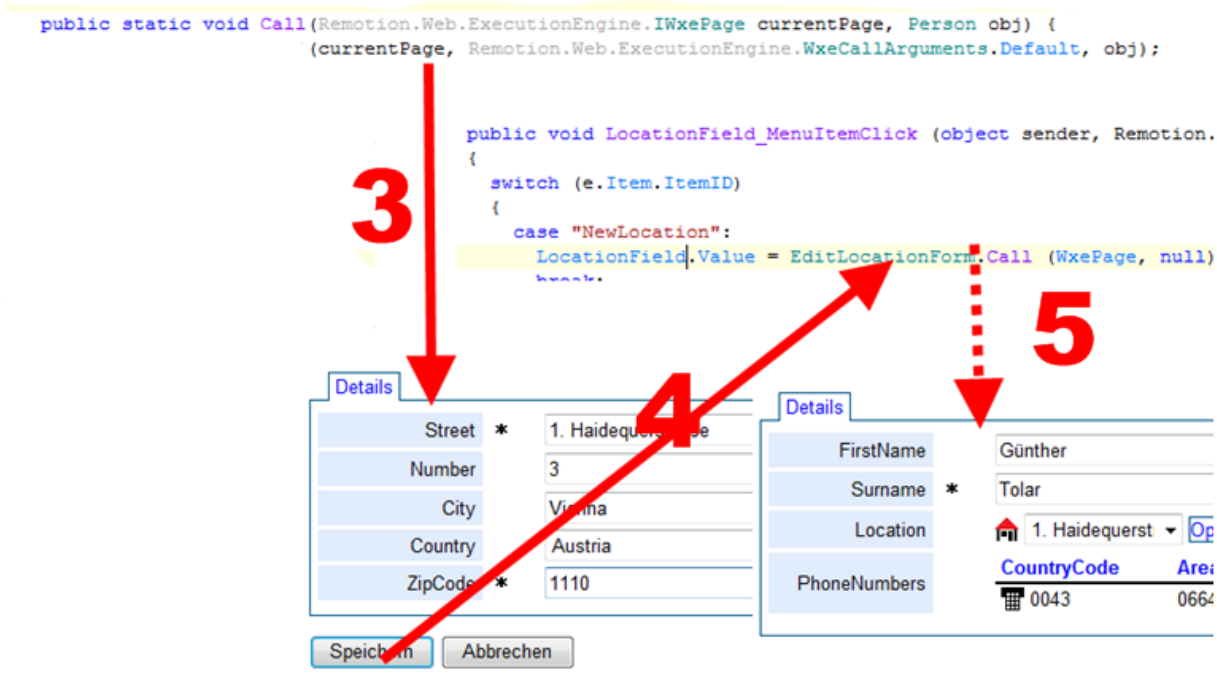
- returns the return value coming from the called page (if any) if there were no exceptions
- rethrows exceptions if there were any, so that the event-handler can catch them

Since this bullet-list might be confusing, here is a slow-motion cartoon of what happens when the user clicks on *New Location…:*



1.) The user clicks "New Location", which invokes the `Call`-method

2.) This overridden `Call`-method does the real work. It calls the actual function. In this case, this causes the EditLocationForm to be built and run.



3.) An exception that is thrown here is caught and stored in a member named ExceptionHandler.Exception. It will be needed later, when the page is called again in the "returning postback" (see below). If everything goes okay, the called page/function is displayed so that the user can do her work.

4.) As soon as the user presses "Save", we want to commit the result and return a value (a Location object, in this case). This requires that the event-handler of the calling page is invoked again, where the

```
LocationField.Value = EditLocationForm.Call (WxePage, null));
```

happens. This requires a postback. In re-call, this is called a "returning postback". re-call gives you an extra flag - `IsReturningPostback` - to check whether you are in a returning postback or not, just as ASP.NET gives you an `IsPostback` flag.

In the returning postback, the entire event-handler where the Call is located is executed again, so that the ReturnValue (if any) or an exception thrown in the previous step are retrieved by Call and actually returned to the caller.

5.) The caller can use the return value or catch the exception. In our example, it assigns the Location object to the `LocationField` control.

A variation of step 4 is that the user clicks Cancel, which throws a `WxeUserCancelException`. In our example, that exception is caught by the caller, but this is not always so. If the caller does not catch the exception, the caller of the caller will, or some other caller down the stack. In any case, the catching caller is the one where the flow of control will return to after the cancel.

The author recommends to step thru this process in the Visual Studio debugger -- that really helps to clarify this well-thought-out process, even if you believe you have grokked it.

However, there is a problem. The code before the `Call`-method invocation will run twice, once for each post-back. This does not matter for some code (if it has no side-effects). For some code, however, this is a bad idea. You can protect this code by embracing it in an `if (!IsReturningPostback)`. Here is an example. Let's say you want to create the new `Location` object in the calling `EditPersonForm` rather than the called `EditLocationForm`. Consequently, a naive *New location...* event-handler could look like this (which is WRONG):

```
public void LocationField_MenuItemClick(object sender,
                                        WebMenuItemClickEventArgs e)
    {

        if (e.Item.ItemID == "NewLocation")
        {
          try
          {
            // WRONG
            var newLocation = Location.NewObject();
            LocationField.Value = EditLocationForm.Call(WxePage,
                                                        newLocation);
          }
          catch (WxeUserCancelException) { }

        }
```

```
        else if (e.Item.ItemID == "SearchLocation")
        {
        }
    }
```

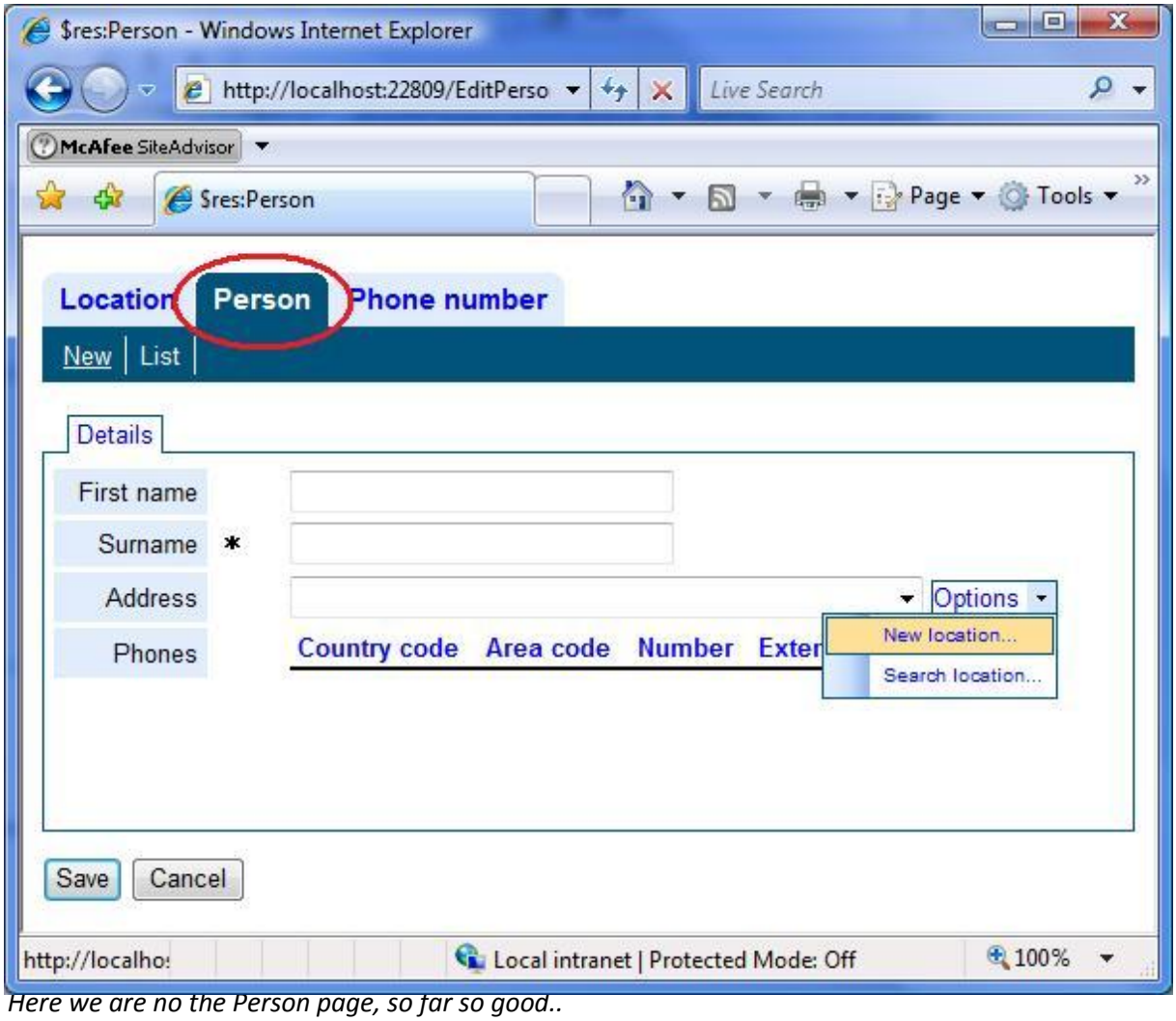
The creation of a new domain object clearly is a side-effect and should not occur twice. Only the first time around, during the execution of the event-handler up to the call should the object be created, not during the returning postback. The correct way to achieve this is

```
public void LocationField_MenuItemClick(object sender,
WebMenuItemClickEventArgs e)
    {

        if (e.Item.ItemID == "NewLocation")
        {
          try
          {
            Location newLocation = null;
            if (!IsReturningPostBack)
            {
              newLocation = Location.NewObject();
            }
            LocationField.Value = EditLocationForm.Call(WxePage,
                                                        newLocation);
          }
          catch (WxeUserCancelException) { }

        }
        else if (e.Item.ItemID == "SearchLocation")
        {
        }
    }
```

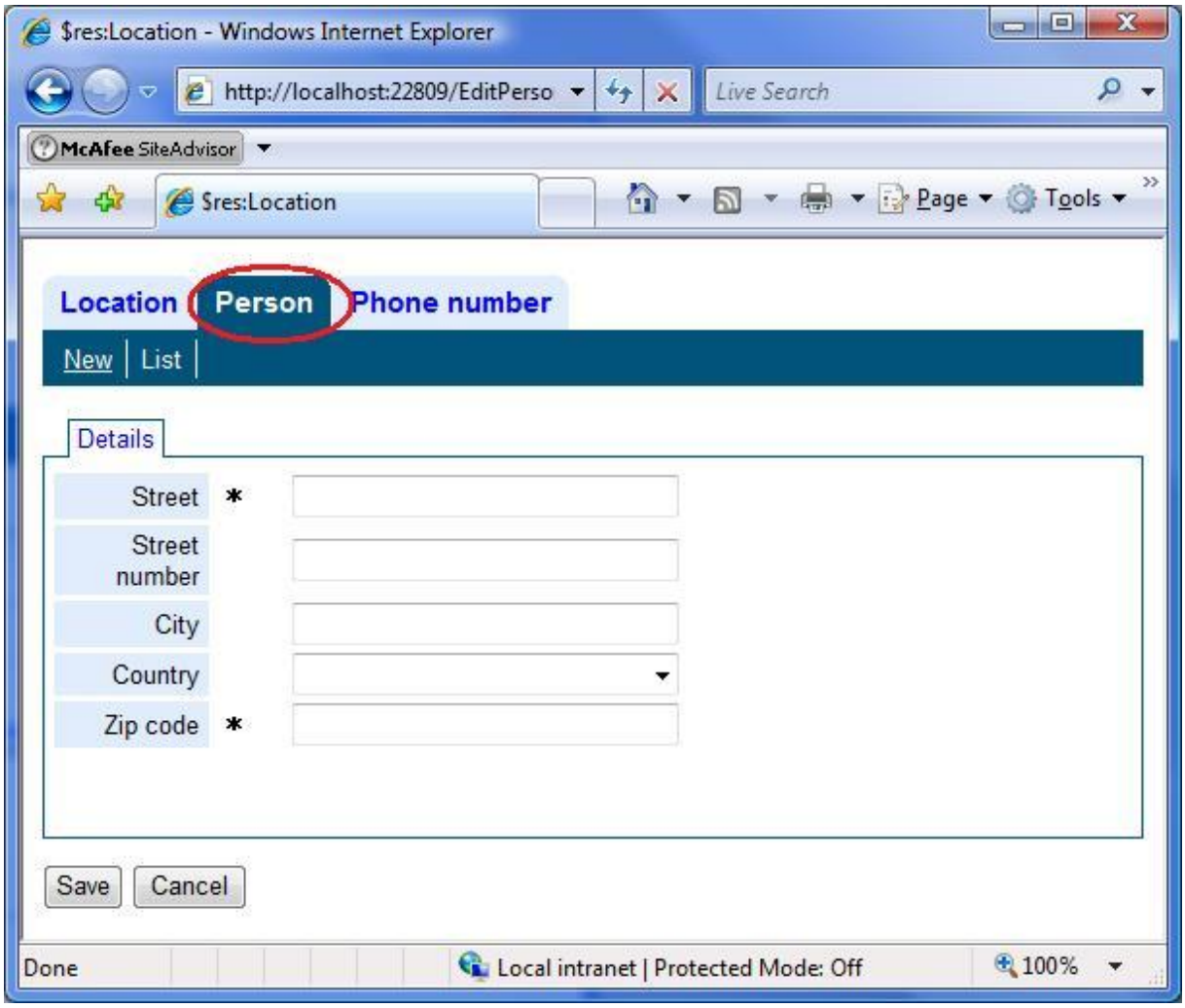
This works despite the `newLocation` parameter being null for the second invocation. For the invocation in a returning postback arguments to `Call` are ignored. As always: Keep your event handler code free of side-effects, if possible. In this case, we keep the creation of the `Location` object (clearly a side-effect) out of the event-handler by putting it into the `EditLocationForm`.

Notice anything about the *Person* and the *Location* tab?

Note that, if you go from the `EditPersonForm` to the `EditLocationForm`, the page-tab does *not* switch from *Person* to *Location*. Is this a bug? Here is an illustration, you can try this out yourself:

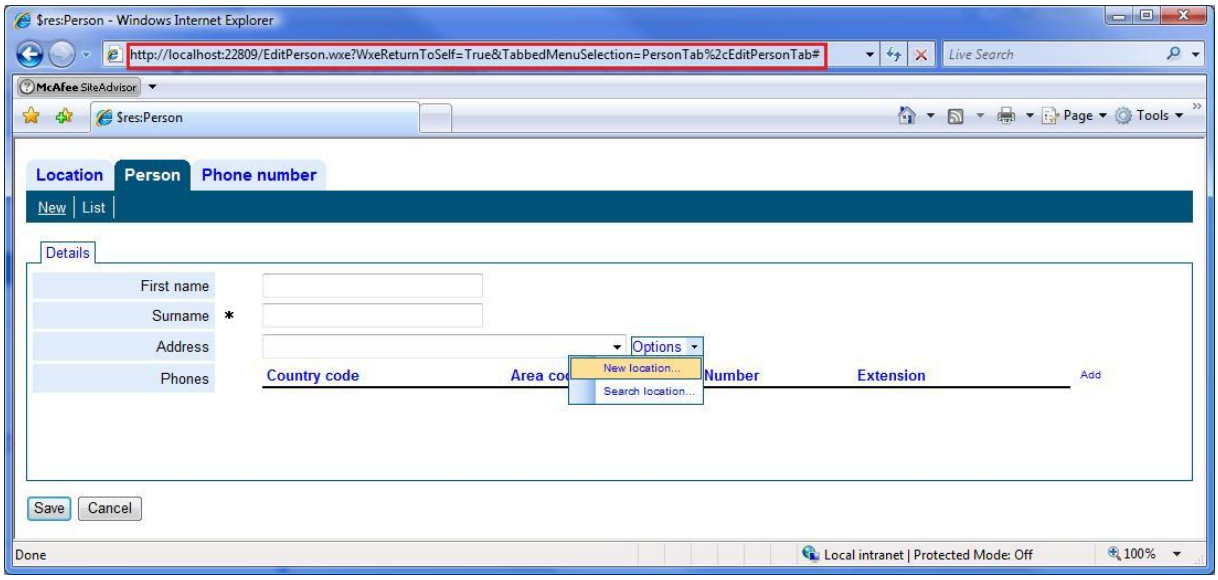


Here we are no the Person page, so far so good..

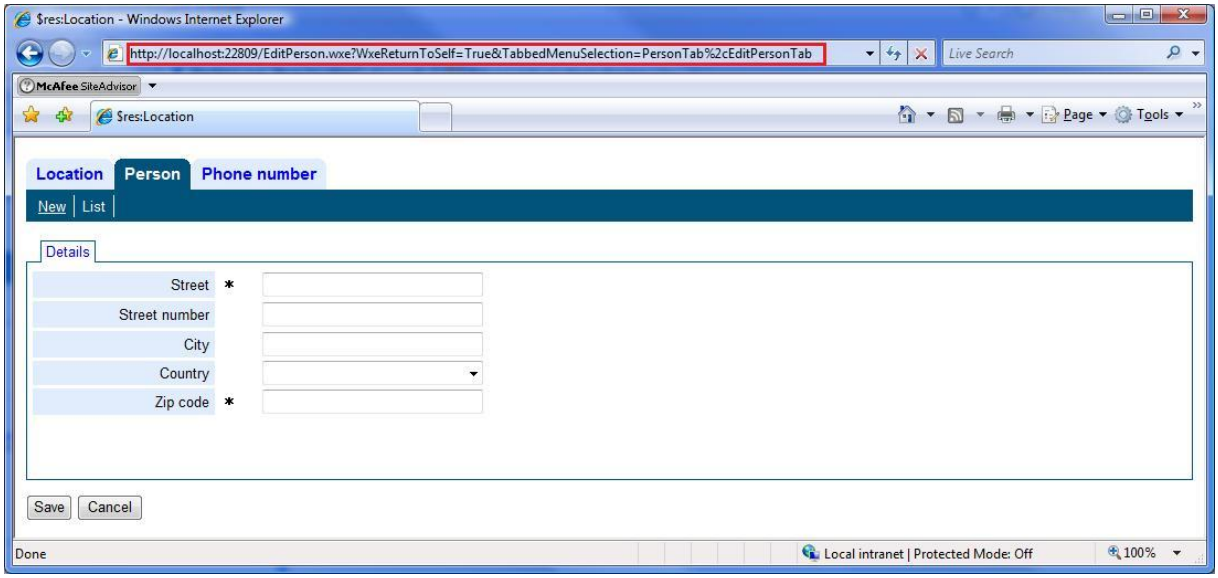


... but here we have arrived at the Location page, and still have a Person tab. What's wrong?

This is *not* a bug. It is a natural consequence of a *Server.Transfer* (section FIXME), one of two methods ASP.NET uses for showing the user another page. In this *Person->Location* example, a Server.Transfer gets you the transition from the `EditPersonForm` to the `EditLocationForm`. Remember from the chapter FIXME that a Server.Transfer keeps the URL intact when sending you a different .aspx-page. You can see this for yourself by looking at the URL before and after the transition. Here is an experiment: In `SearchPersonForm.aspx`, we click on "Sigmund Freud"s *Edit* menu item, which takes us to the corresponding `EditPersonForm`. Note that the URLs are the same in both screen-shots:



From the person...



... to the location

Never mind how the URL is composed and why. Suffice to say that it reflects which of the tabs is displayed as being selected. Since Server.Transfer does not
change the URL, the selection of tabs does not change, either. URLs and URL parameters deserve more discussion. We will return to URLs and URL parameters in section FIXME.

Understanding and using Wxe-Headers

The typcial use of the XML re-call headers is type information for local data – parameters, return value and local variables. A typical declaration looks like this:

```
<WxeFunction>
  <Parameter name="obj" type="Customer" />
  <ReturnValue type="Invoice" />
  <Variable name="fooBar" type="FooBar" />
</WxeFunction>
```

(The order is significant here.)

The meaning of this example WXE-function header roughly corresponds to a C# declaration like

```
public Invoice MyFunc(Customer obj)
{
  FooBar fooBar;
  ...
}
```

Since angle brackets (<>) must be escaped in XML, a declaration like

```
<ReturnValue type="ObjectList<Customer>" />
```

is invalid and must be re-phrased as

```
<ReturnValue type="ObjectList&lt;Customer&gt;" />
```

or, more readably (and writably):

```
<ReturnValue type="ObjectList{Customer}" />.
```

(Note the curly brackets substituting angle brackets. The mechanism is the same as in C#'s DocComments. Instead of substituting the "<" and ">" for XML's "<" and ">" you substitute "{" and "}".)

More features exist, but they are rarely used. You can find the xml-schema for WXE-headers at https://svn.re-motion.org/svn/Remotion/trunk/Remotion/Web/ExecutionEngine.CodeGenerator/Schema/FunctionDeclaration.xsd

What follows is a brief discussion of those extra elements.

Understanding `WxePageFunction` attributes

The attributes of `WxePageFunction` are optional and override defaults.

Attribute	Meaning	Example
`pageType`	The ASP.NET-type behind the page	`EditPersonForm`
`aspxFile`	The .aspx page file	`EditLocationForm.aspx`
`functionName`	The name of the function class that will be generated. Default is name of the page class + "Function"	`EditLocationFormFunction`
`functionBaseType`	The base class of the function class that will be generated. Default is the name passed to `wxegen.exe` as `/baseFunction` parameter	`BaseFunction`

Important note: wxegen.exe uses the path specified in its file mask (first) parameter to find the file specified by aspxFile. The path is relativ to the directory where wxegen.exe is run, i.e. the PhoneBook.Web project directory.

```
wxegen UI\*.aspx.cs WxeFunctions.cs /prjfile:PhoneBook.Web.csproj
/functionbase:...
```

`uigen.exe` builds the project with the assumption that the generated files (specified in `wxegen.exe`'s file mask parameter) shall be stored in a project sub-folder named `UI`.

For a complete discussion on `wxegen.exe`'s command line arguments, see section FIXME.

In/out parameters for the `Parameter` element

The optional `direction` attribute for `Parameter` can specify whether the parameter is an in or out parameter or both:

```
<Parameter param1 direction="In" />
<Parameter param2 direction="Out" />
<Parameter param3 direction="InOut" />
```

The default is `In`.

Mind the usings !

`wxegen.exe` copies the `using` s from the file with the WXE function header to `wxefunctions.cs`, so that all types can be resolved. However, if a type is used *only* in a function header, then a tool like Resharper might deem the `using` statement covering the type redundant and remove it.
Workaround: Use fully qualified type names for all types.

Fleshing out clickable phone-numbers

In the section FIXME we have introduced "clickable phone-numbers". However, at that point we left it open how a click actually conjures up the `EditPhoneNumberForm`. In this section we will complete our quest for clickable phone-numbers. Not much is left to do, and after the previous section you could probably figure out yourself how to connect a click to the event-handler.

Calling the `EditPhoneNumberForm`

In contrast to *New location...*, the called `EditPhoneNumberForm` won't return a value. And in contrast to *New location...* the `EditPhoneNumberForm` will receive an *existing* object as parameter,, namely, theinstance representing the clicked phone-number. Therefore we are all set in terms of the called page/function's signature. No changes to the re-call header required!

What we do have to prepare in code is retrieving the `PhoneNumber` instance we want to pass to the `EditPhoneNumberForm`. As explained in section FIXME, we have gone to great length in order to funnel the clicked `PhoneNumber`'s ID thru the link to the event-handler. Now it's time to use it.

In the event-handler, add the following code :

```
protected override void OnClick (BocCustomCellClickArguments arguments,
                                 string eventArgument)
    {

      var id = ObjectID.Parse (eventArgument);
```

```
      PhoneNumber number = PhoneNumber.GetObject (id);
      var page = (IWxePage) arguments.List.Page;
      try
      {
        EditPhoneNumberForm.Call (page, number);
      }
      catch (WxeIgnorableException) { }

      if (Commit)
      {
        ClientTransaction.Current.Commit();
      }

    }
```

As you see, we get the phone-number's `ObjectID` as a parsible string in the `eventArgument`. The snippet `ObjectID.Parse (eventArgument)` is the opposite companion of the code in the `Render` method that rendered the `ObjectID` as a string:

```
phoneNumber.ID.ToString()
```

As soon as we have that ID, we can retrieve the `PhoneNumber` instance:

```
var id = ObjectID.Parse (eventArgument);
PhoneNumber number = PhoneNumber.GetObject (id);
```

`eventArgument` is not all there is to it in this event-handler, however. The `eventArgument` is just a spot where the event-handler receives the string that was passed to `GetPostBackClientEvent()` (see section FIXME). More argument data is passed in the `BocCustomCellClickArguments`, namely:

- the business object (`IBusinessObject`) of the row where the click occurred
- the column definition of the column where the click occurred in the property `ColumnDefinition`
- the BocList instance containing that column (and, by extension, our `PhoneNumberCell`)

The clicked business object in `BusinessObject` is *not* the clicked phone-number. Since we rendered the link ourselves, re-bind does not know that it was the phone-number that was clicked. For re-bind the click occurred in a list of `Person` objects. Therefore the `BusinessObject` property in the `BocCustomCellClickArguments` is the corresponding `Person` domain object.

In this code, we are only interested in the `Page` that invoked the event-handler, because this page is what we pass to the re-call function invocation as the calling page. You might think that simply passing this `arguments.List.Page` value to `EditPhoneNumberForm.Call` would be enough, but this is not always so. Remember that we aim for re-usability here, so we can't make any assumptions on the type of the calling page. We might publish this `PhoneNumberCell.cs` as an open source module, and some people might have their own implementations of `BasePage`. For this reason, the only assumption we can make here is that any page invoking the event-handler must support
the interface for re-call page/functions, i.e. `IWxePage`.

Always cast to the interface here, never to a class!

So there you go with the lines of code we have explained here so verbosely:

```
var page = (IWxePage) arguments.List.Page; // assume re-call page
  PhoneNumber number = PhoneNumber.GetObject (id);
  try
  {
     // use that page in the call to EditPhoneNumberForm

    EditPhoneNumberForm.Call (page, number);  }
  catch (WxeIgnorableException)
```

The `try/catch`-block is for telling re-motion that we want to return to the `SearchResultPersonForm` if the user clicks the *Cancel* button (as explained in section FIXME).

The last statement in the event-handler handles the `Commit` parameter discussed in section FIXME. When we return from the called `EditPhoneNumberForm`, we want to make sure that modifications are not lost.

Since our `SearchResultPersonForm` never commits anything (`SearchResultXForm` s never do), we must commit the transaction ourselves here. However, in our re-usable `PhoneNumberCell.cs`, we can't be sure that we are always called from a `SearchResultXForm`. We must provide a facility for *not* committing if that need arises. To this end, we have given our `PhoneNumberCell` an extra boolean property named `Commit` in the `BocCustomCellArguments`.
This property is checked here:

```
if (Commit)
      {
        ClientTransaction.Current.Commit();
      }
```

All together now: Fleshing out *Pick location...*

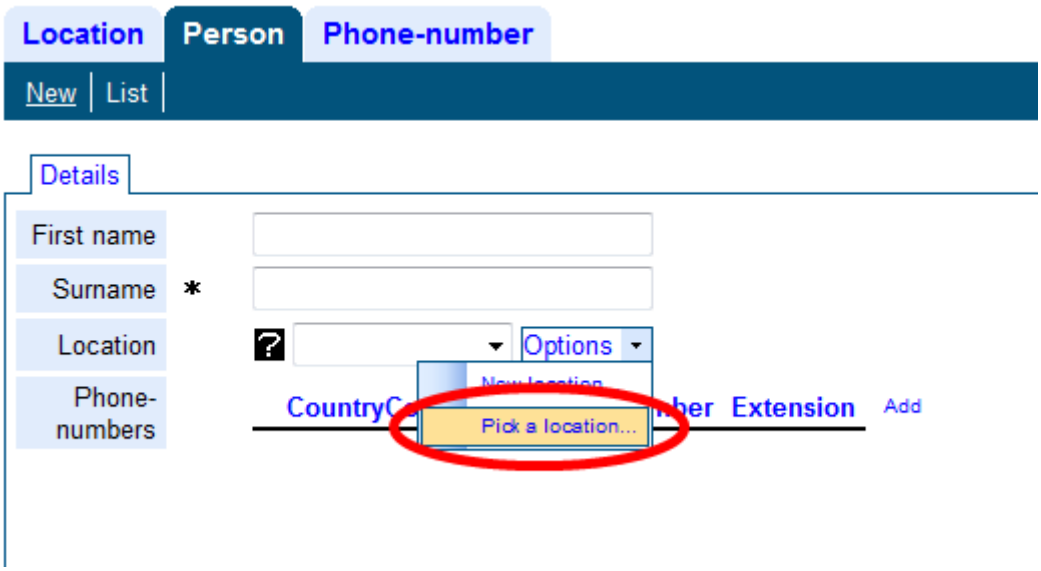
The exercise: Pick a location

After minor improvements to various generated pages/functions we can now turn our attention to a more ambitious exercise. We will cap our PhoneBook-adventure with a self-made re-call page made of various re-bind controls and program some mechanics. And we will use a local variable!

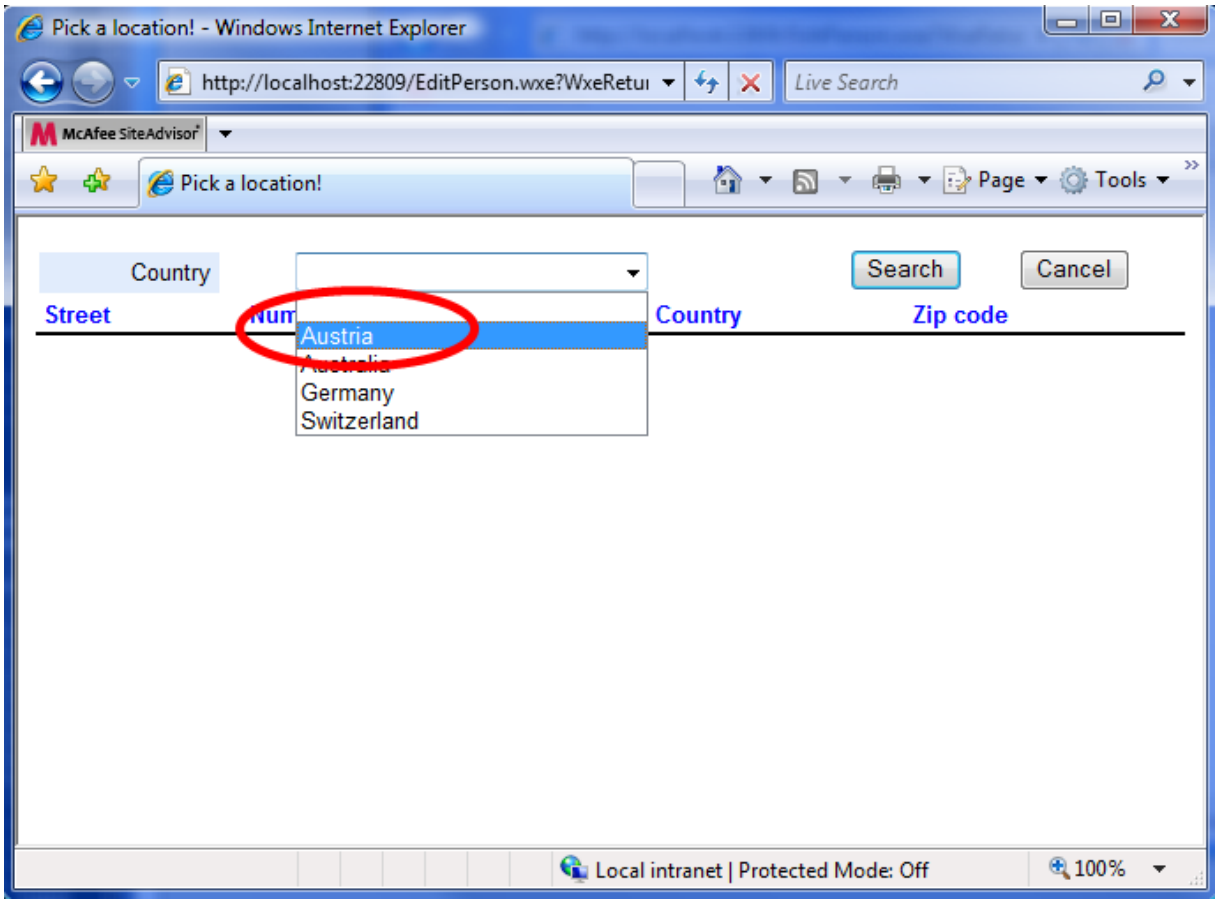
In section FIXME we have added the menu item *Pick location...* to the `Location` reference property in `Person`. We postponed the actual implementation to some later exercise. This later exercise is here!As explained in section FIXME, a drop-down box is not very practical if you have many `Location` objects to choose from – potentially a lot of them. If you are a programmer like me, you probably know millions of people, most of them young and beautiful human females. This can become a real drag if you have to pick a location from a list of thousands.

It is much more practical to choose from a list of a few dozens, filtered by a certain parameter. We will use a given country as a filter parameter in order to limit the number of locations to something manageable.

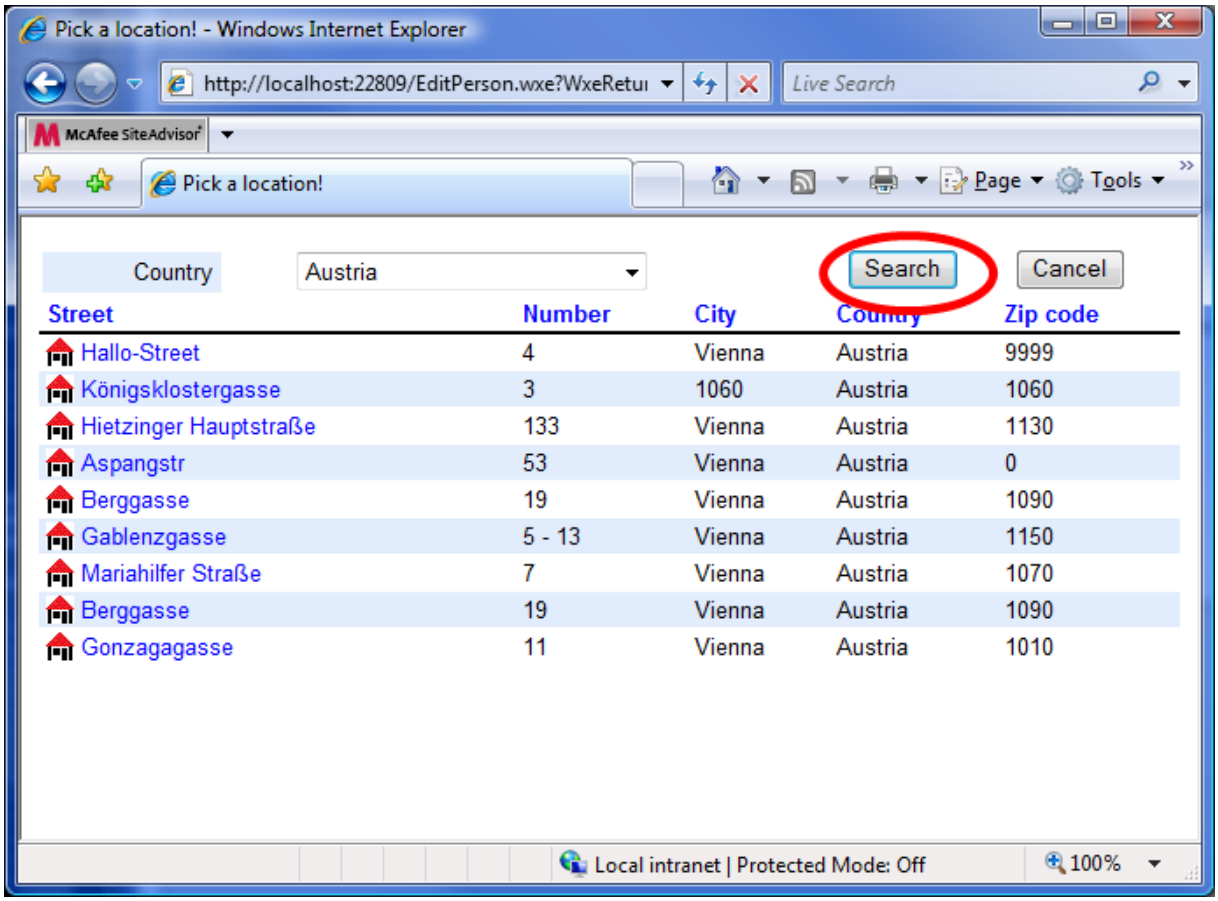
Here is a cartoon specification for the endeavor.



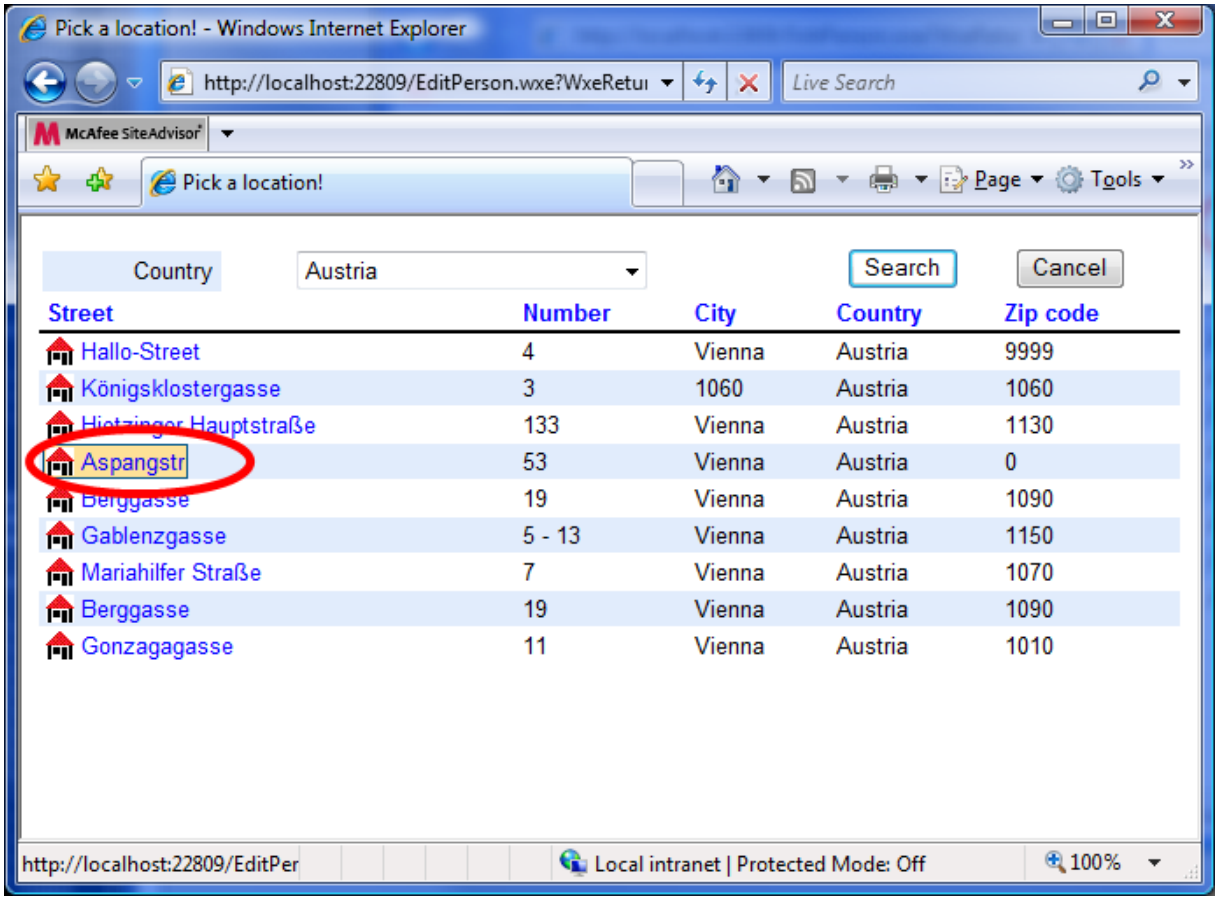
Pick a location...



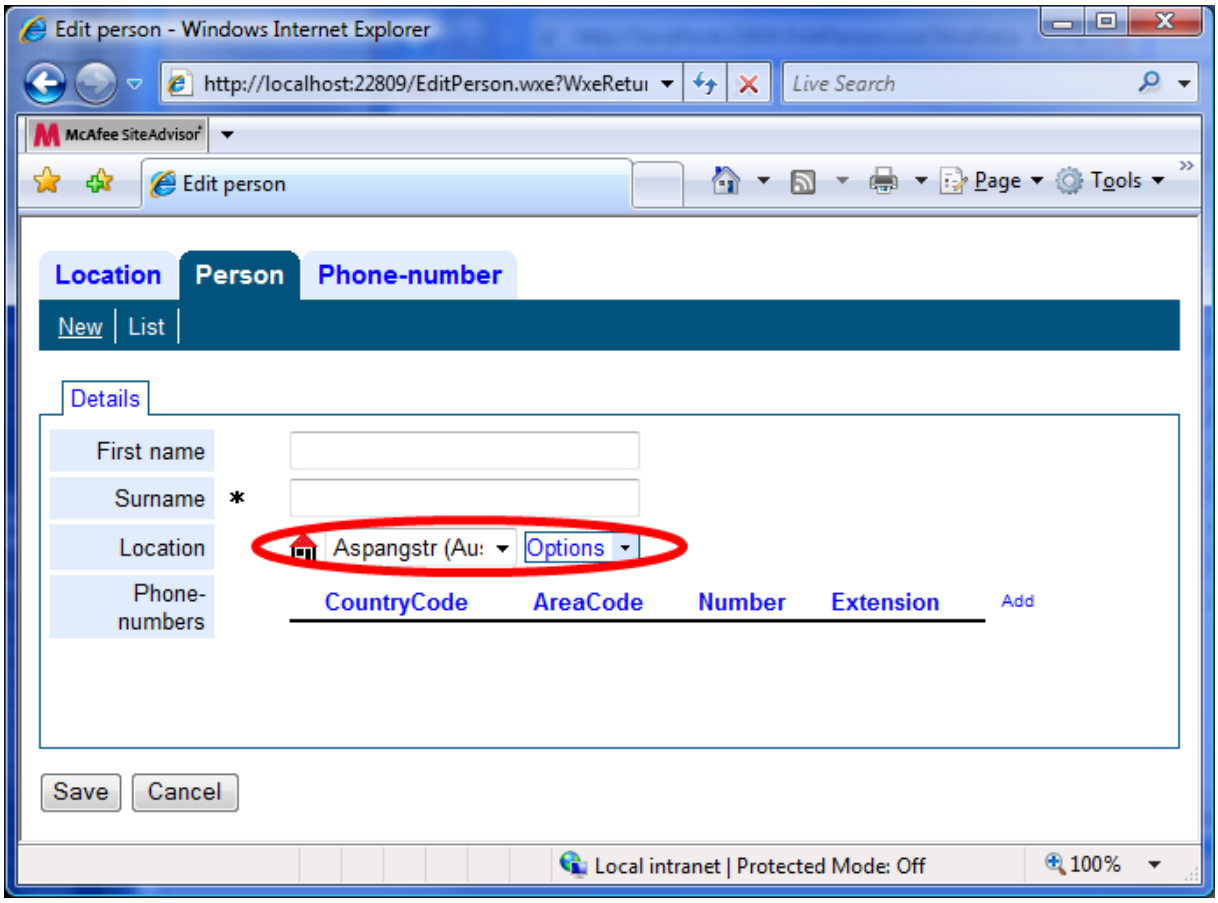
Choose a country...



Voila! Filtered locations...



We pick , for example



Location picked.

A more technical description

We need a re-call page/function with no parameters that returns a `Location` object. We assemble this page/function from controls, as seen in the following illustration:

Here is a list of who is who:

1. A smart label for telling the user that the next control - a BocEnumValue - contains countries. (Name: `CountryLabel`.)
2. A BocEnumValue for picking one of the available countries (from our `Country` enumeration). (Name: `CountryField`.)

3. A web-button (not a BOC control) for submitting the selected country. (Name: `SearchButton`.)
4. A web-button for cancelling the operation. (Name: `CancelButton`.)
5. The BocList for displaying the list of filtered countries (empty in the illustration). (Name: `FilteredLocationList`.)
   Not visible in the illustration are two more controls we need:

- The BocBindableObjectDataSourceControl for getting the `Location` data in the first place (Name: `PickLocationDataSource`).
- The FormGridManager for rendering the smart label in light blue and arranging the smart label, the BocEnumValue and the two web-buttons in a decorative fashion. (Name: `FormGridManager.`)

These controls are not much good without event-handlers. The easier one is `CancelButton_Click` – it just throws a `WxeUserCancelException`.

The event handler for the *Search* button (`SearchButton_Click`) must use the selected country for a LINQ-query to get the filtered `Location` objects.

The set of filtered objects must get into the BocList. This is achieved by loading them unbound (as explained in section FIXME).

As proposed in this outline, the *Pick location...* feature would even work. However, a problem occurs as soon as the dear user clicks on one of the column headers to sort the `Location` s in the BocList – instead of being sorted, the `Location` objects vanish entirely. Why is this so?

The reason is that a click on a column for sorting causes a postback – after all, the `Location` objects get sorted on the server and the page must be reconstructed with a sorted BocList. The BocList, however, can't remember its list of unbound values (the `Location` objects) when a postback occurs. Therefore, a local variable belonging to the page/function is used to remember them for the BocList between postbacks.

In this case, we can use a simple array to store the `Location` objects. Consequently, we have to supplement a declaration for such a local variable ("`items`") to our `PickLocation` page/function. SearchButton_Click stores the filtered `Location` objects in the `items` variable and loads them into the BocList from there. As for the postback: upon page-reconstruction, the `Location` objects in `items` are loaded unbound into the page's BocList again.

Another embellishment requires an extra declaration of an event-handler: `Page_Load`. In the previous section FIXME we have seen that this is the spot where parameters for the page/function can be initialized. We don't have any for the `PickLocation` page/function, but we need `Page_Load` for setting the title of the page to a localized string resource.

Declaring `PickLocation.aspx`

In this exercise, we will go all the way and craft everything by hand, in source-code. No Visual Studio designer will be used, and no pre-fabricated .aspx. The first trap here is to have Visual Studio generate the skeleton for the web-form. When doing so, the header will sport

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```
If you want to cheat and go with the Visual Studio-skeleton, at least replace this declaration with the old-schoolish

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

This is necessary, because the re-bind controls render HTML 4.0/Transitional-code. If you go with the Visual Studio-generated boilerplate, the program will still work, but your controls will look weird.

Create a the `PickLocation.aspx` file under the `PhoneBook.Web\UI` folder and add it to the project. Delete everything in the newly minted `PickLocation.aspx` file.

The first lines of our `PickLocation.aspx` shall be

```
<%@ Page Language="C#" CodeBehind="PickLocation.aspx.cs"
Inherits="PhoneBook.Web.UI.PickLocation" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
```
Apart from the replacing the XHTML for HTML, this should be little surprises:

- our code-behind is in the yet-to-be-written `PickLocation.aspx.cs` – that's where our event-handlers will go
- the new page gets its meat from the yet-to-be-written `PhoneBook.Web.UI.PickLocation`

The next lines are fairly boring as well:

```
<head id="PickALocationHeader" runat="server">
  <title><!-- Page title set in Page_Load !--></title>
</head>
```

This is the header with a title. The title is commented out, because we don't want static text here. We will use a globalized string for this in the `Page_Load` event-handler in `PickLocation.aspx.cs`.

The next lines are more interesting, because we move from plain ASP.NET to re-motion:

```
<form id="PickLocationForm" runat="server">
<remotion:FormGridManager ID="FormGridManager" runat="server" />
<remotion:BindableObjectDataSourceControl ID="PickLocationDataSource"
  runat="server" Mode="Search" Type="PhoneBook.Domain.Location,
PhoneBook.Domain">
```

The first line, `<form id="PickLocationForm" runat="server">`, simply is the opening to our form. Next:

```
<remotion:FormGridManager ID="FormGridManager" runat="server" />
```

declares the form-grid manager we need for our lonely `Country` smart label.

The declaration for the `PickLocationDataSource` (next line) is the all-important data-source for our `Location` objects. This data source is needed by

- the BocEnumControl with the set of available countries (Quite an overkill here: we just want to make sure that the enum type is correct. For that we could tie it to any `Country` instance.)
- the BocList. As explained in section FIXME, the BocList does not need the data-source for *values* (because the values get into the BocList unbound). The data-source is needed because it knows of what properties/column names a `Location` object is composed, and what types those properties have.

The next table serves no other purpose than to give the `FormGridManager` clues on how to format the controls embraced by the table. It contains a single row:

```
<table id="CountryFormGrid" runat="server">
  <tr>
    <td>
      <remotion:SmartLabel runat="server"
                           id="CountryLabel"
                           ForControl="CountryField" />
    </td>
```

Important: Don't be too creative with the table's name. It MUST end with "`...FormGrid`", otherwise the form-grid manager won't know that it is this table it is supposed to render.

Here we meet the aforementioned smart-label. It is named "`CountryLabel`". The `ForControl` field specifies which control the SmartLabel describes. The label uses this control to determine what test to display: this simply is the property name the field is associated with: `Country` in `Location`.

The next column in the `<table>` is for the BocEnumValue:

```
<td>
      <remotion:BocEnumValue ID="CountryField" runat="server"
        DataSourceControl="PickLocationDataSource"
PropertyIdentifier="Country" />
    </td>
```

This is the `BocEnumValue` for selecting the country. By default, it will be rendered as a drop-down list, but this can be changed. This `BocEnumValue` is named "`CountryField`". The name must match the `ForControl`-attribute in the smart-label, of course. As explained above, the `CountryField` BocEnumValue gets its data - type and enum-values - from the `PickLocationDataSource` data-source, as specified in the `DataSourceControl` attribute. Here the `PropertyIdentifier` attribute is not as usual a place for getting a value from a concrete instance, but a place for getting enum-values.

The next two columns in the `<table>` are for the two web-buttons, and we complete both the single row and the entire table:

```
<td>
      <remotion:WebButton ID="SearchButton"
                          runat="server"
                          onclick="SearchButton_Click"
                          Text="$res:Search" />
    </td>
    <td>
      <remotion:WebButton ID="CancelButton"
                          runat="server"
                          onclick="CancelButton_Click"
```

```
                            Text="$res:Cancel" />
    </td>
  </tr>
</table>
```

The web-buttons have globalized text, as denoted by the "$res:"-prefix. And each has its own event-handler, of course.

So much for the easy part. Now on to the advanced level – the BocList. We begin the declaration with a typical BocList-header. The BocList has the name "`FilteredLocationsList`" (mind the plural "s"!). Just as the `LocationField` above, it is tied to the `PickLocationDataSource`. The event-handler for a click into the BocList is "`FilteredLocationsList_ListItemCommandClick`" (again: Mind the plural "s").

```
<remotion:BocList ID="FilteredLocationsList" runat="server"

  DataSourceControl="PickLocationDataSource"

  OnListItemCommandClick="FilteredLocationsList_ListItemCommandClick">
```

After the header, we find the declaration of the fixed columns – the columns/properties of the `Location`s we want in our `BocList`. The first column is the most sophisticated one, because it is supposed to fire an event when clicked – the `FilteredLocationsList_ListItemCommandClick` command from the declaration above, to be exact. This is the `BocSimpleColumnDefintion` for the `Street` property. Since `Street` is a required field, we can be sure to always have text to click on. This is important for usability.

```
<FixedColumns>
    <remotion:BocSimpleColumnDefinition
      PropertyPathIdentifier="Street" ItemID="LocationPick">
      <PersistedCommand>
        <remotion:BocListItemCommand Type="Event" />
      </PersistedCommand>
    </remotion:BocSimpleColumnDefinition>
```

Since the `BocSimpleColumnDefintion` for `Street` can fire the event, it needs a `PersistedCommand` declaration to learn which type of event to fire. We go for the simplest case here: XQuestion. Unlike the other `BocSimpleColumnDefinition` s, this column also needs an ItemID to identify itself in the event-handler. You might think that this is not quite logical, because no other sources of events exist in this BocList, but fact is, that without an ItemID, our `BocSimpleColumnDefinition` won't fire events. This is a safety-feature:
FIXME.

The remaining `BocSimpleColumnDefinition`s are fairly straightforward. We include *all* properties/columns for `Location` objects:

```
<remotion:BocSimpleColumnDefinition PropertyPathIdentifier="Number" />
    <remotion:BocSimpleColumnDefinition PropertyPathIdentifier="City" />
    <remotion:BocSimpleColumnDefinition PropertyPathIdentifier="Country" />
    <remotion:BocSimpleColumnDefinition PropertyPathIdentifier="ZipCode" />
```

Since those `BocSimpleColumnDefinition`s don't do anything beyond displaying the values for their columns, not more than a `PropertyPathIdentifier` is needed. This

`PropertyPathIdentifier` specifies where each `BocSimpleColumnDefinition` finds the value to display.

The rest of the code is just tying the loose ends together:

```
</FixedColumns>
</remotion:BocList>
</form>
</html>
```

At this point you have a hand-made, whole-cloth, on-the-metal `PickLocation.aspx`. Treat yourself with a soda!

Programming `PickLocation.aspx.cs`

We got the .aspx nailed, now we need a corresponding code-behind to handle all those events. If Visual Studio has not done that for you already, create a `PickLocation.aspx.cs` file in `PhoneBook.Web\UI`. Just as for `PickLocation.aspx`, delete all the Visual Studio-generated boilerplate – we will write our own boilerplate.

This program constitutes the re-call page/function we have talked about so much already. The opening code is the required namespaces:

```
using System;                                          // .NET core
using System.Linq;                                     // yes, there will be
                                                       // a Linq-query
using PhoneBook.Web.Classes;                           // that's where
                                                       // BasePage is from
using Remotion.Data.DomainObjects.Queries;             // yes, there will be
                                                       // queries (re-store)
using Remotion.ObjectBinding.Web.UI.Controls;          // re-bind
using Remotion.Web.ExecutionEngine;                    // re-call

using PhoneBook.Domain;                                // our PhoneBook
                                                       // domain
```

Like all the other ASP.NET-code in our application, this code also belongs to the `PhoneBook.Web.UI` namespace:

```
namespace PhoneBook.Web.UI {
```

The next lines are of central interest to this chapter, because they constitute the re-call header for our page:

```
// <WxeFunction>
  //   <ReturnValue type="Location" />
  //   <Variable name="items" type="Location[]" />
  // </WxeFunction>
```

As outlined in FIXME, this page/function returns an object of type `Location`. The local variable `items` is the only real innovation in this exercise. It is the `Location`-array where we will store our filtered `Location` objects between postbacks. Such a local variable is declared in a `<Variable>` node. Mind the spelling! In contrast to .aspx, this XML is case-sensitive. Also be careful to get the `<WxeFunction>` right. If you don't, the bogus header will be ignored without warning.

The annotated class is named `PickLocation` and derived from `BasePage`, like all re-call pages in this application:

```
public partial class PickLocation : BasePage {
```

The first event-handler we implement is `Page_Load`. Here we set the title to a globalized string named "`Pick~Location`" (to be globalized later). The more important task is to repopulate the BocList with the `Location` objects stored in the page-local variable `items`.

```
protected void Page_Load (object sender, EventArgs e)
    {
      Title = ResourceManagerUtility.GetResourceManager (this).GetString
("Pick~Location");
      if (IsPostBack)
      {
        FilteredLocationsList.LoadUnboundValue (items, true);
      }
    }
```

Note that `item` is unknown to Visual Studio at this time, because the declaration for it is generated by `wxegen.exe`, and it has not run yet. Also note that the `FilteredLocationsList` (BocList) is only populated after a postback. If the page is loaded originally (i.e. not a postback) there are no `items` yet to fill the BocList with.

A more interesting event-handler is `FilteredLocationsList_ListItemCommandClick`. This is the handler for the click on one of the filtered `Location` s in the BocList. As you can see, this event-handler checks whether the `ItemID` from the `Street SimpleColumnDefinition` really is "`LocationPick`", and throws an exception if it isn't. This should never happen and might look anal-retentive, but such measures are usually your friend. If you make extensions to such code and the exception is thrown, then you know something is wrong. As always in re-call/re-bind, the `ItemID` comes in the event arguments.

```
protected void FilteredLocationsList_ListItemCommandClick (object sender,
BocListItemCommandClickEventArgs e)
    {
      if (e.Column.ItemID == "LocationPick")
      {
        ReturnValue = (Location) e.BusinessObject;
        Return ();
      }
      else
      {
        throw new ArgumentException ();
      }
    }
```

The next event-handler is `SearchButton_Click`. It triggers the query for filtering `Location` objects in the database, fetching only those that happen to be in the same country as the one selected in the `CountryField`:

```
protected void SearchButton_Click (object sender, EventArgs e)
    {
      var locations = from loc in QueryFactory.CreateLinqQuery<Location> ()
                      where loc.Country == (Country?) CountryField.Value
                      select loc;
```

```
      items = locations.ToArray ();

      FilteredLocationsList.LoadUnboundValue (items, false);
    }
```

The found `Locations` are stored in the `items`, and the BocList `FilteredLocationsList` is loaded with those `items`. Where is the transaction, you might wonder? Again: re-call has set up a transaction for us, so we don't have to.

The last event-handler is the most boring one. We do what we always do when faced with an event that means *Cancel*: we throw a `WxeUserCancelException` and let re-call do the rest:

```
protected void CancelButton_Click (object sender, EventArgs e)
    {
      throw new WxeUserCancelException ();
    }
```

At this point you might think it is time for another soda, but we are not quite done yet. We have yet to globalize "`Pick~Location`". Simply add it to the `PhoneBook.Web\Globalization\Global.resx` and `PhoneBook.Web\Globalization\Global.de.resx` files:

Global.de.resx	Global.resx
Standort aussuchen	Pick location

Listing PickLocation.aspx.cs

```
using System;
using System.Linq;
using PhoneBook.Web.Classes;
using Remotion.Data.DomainObjects.Queries;
using Remotion.ObjectBinding.Web.UI.Controls;
using Remotion.Web.ExecutionEngine;
using Remotion.Web.UI.Globalization;

using PhoneBook.Domain;


namespace PhoneBook.Web.UI
{
  // <WxeFunction>
  //   <ReturnValue type="Location" />
  //   <Variable name="items" type="Location[]" />
  // </WxeFunction>
  public partial class PickLocation : BasePage
  {
    protected void Page_Load (object sender, EventArgs e)
    {
      Title = ResourceManagerUtility.GetResourceManager (this).GetString
("Pick~Location");
      if (IsPostBack)
      {
        FilteredLocationsList.LoadUnboundValue (items, true);
      }
    }

    protected void FilteredLocationsList_ListItemCommandClick (object sender,
BocListItemCommandClickEventArgs e)
    {
      if (e.Column.ItemID == "LocationPick")
      {
        ReturnValue = (Location) e.BusinessObject;
        Return ();
      }
      else
      {
        throw new ArgumentException ();
      }
    }

    protected void SearchButton_Click (object sender, EventArgs e)
    {
      var locations = from loc in QueryFactory.CreateLinqQuery<Location> ()
                      where loc.Country == (Country?) CountryField.Value
                      select loc;

      items = locations.ToArray ();

      FilteredLocationsList.LoadUnboundValue (items, false);
  }



    protected void CancelButton_Click (object sender, EventArgs e)
    {
      throw new WxeUserCancelException ();
    }
  }
}
```

Listing `PickLocation.aspx`

```
<%@ Page Language="C#" CodeBehind="PickLocation.aspx.cs"
Inherits="PhoneBook.Web.UI.PickLocation" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head id="PickALocationHeader" runat="server">
  <title><!-- Page title set in Page_Load !--></title>
</head>
<form id="PickLocationForm" runat="server">
<remotion:FormGridManager ID="FormGridManager" runat="server" />
<remotion:BindableObjectDataSourceControl ID="PickLocationDataSource"
  runat="server" Mode="Search" Type="PhoneBook.Domain.Location,PhoneBook.Domain" />
<table id="CountryFormGrid" runat="server">
  <tr>
    <td>
      <remotion:SmartLabel runat="server" id="CountryLabel"
ForControl="CountryField" />
    </td>
    <td>
      <remotion:BocEnumValue ID="CountryField" runat="server"
        DataSourceControl="PickLocationDataSource" PropertyIdentifier="Country" />
    </td>
    <td>
      <remotion:WebButton ID="SearchButton" runat="server"
onclick="SearchButton_Click" Text="$res:Search" />
    </td>
    <td>
      <remotion:WebButton ID="CancelButton" runat="server"
onclick="CancelButton_Click" Text="$res:Cancel" />
    </td>
  </tr>
</table>

<remotion:BocList ID="FilteredLocationsList" runat="server"
  DataSourceControl="PickLocationDataSource"
OnListItemCommandClick="FilteredLocationsList_ListItemCommandClick">
  <FixedColumns>
    <remotion:BocSimpleColumnDefinition
      PropertyPathIdentifier="Street" ItemID="LocationPick">
      <persistedcommand>
        <remotion:BocListItemCommand Type="Event" />
      </persistedcommand>
    </remotion:BocSimpleColumnDefinition>
    <remotion:BocSimpleColumnDefinition PropertyPathIdentifier="Number" />
    <remotion:BocSimpleColumnDefinition PropertyPathIdentifier="City" />
    <remotion:BocSimpleColumnDefinition PropertyPathIdentifier="Country" />
    <remotion:BocSimpleColumnDefinition PropertyPathIdentifier="ZipCode" />
  </FixedColumns>
</remotion:BocList>
</form>
</html>
```

re-call options

After the discussion how to prepare and call re-call pages/functions, let's see how to control the terms on which such pages/functions are called. re-call gives you influence over the manner such a transition happens, namely:

- shall the page/function open in a new window?
- shall the page/function be transmitted via `Response.Redirect` instead of `Server.Transfer`?
- shall the page be specified by a "perma-URL" (section FIXME)?

As mentioned briefly in the overview of this chapter, such modes of operation are controlled via the `arguments` argument of `Call`, the override we have not used yet.

The following sections discuss these topics in detail.

Calling pages/functions as "external"

An "external" window is a new browser window that opens for a called page/function. If you want this to happen, simply pass an instance of an `WxeCallArgument` that has a `WxeCallOptionsExternal` wrapped into it to `Call`. It is easy to see how this works by modifying our clickable phone-numbers in such a fashion. In your `PhoneNumberCell.cs` `OnClick()` event-handler, replace the re-call invocation

```
EditPhoneNumberForm.Call (page, number);
```

by

```
var externalOption = new WxeCallOptionsExternal ("_blank");
var externalOptionArgument = new WxeCallArguments ((Control) page,
                                                    externalOption);
// Note the extra "externalOptionArgument" argument here
EditPhoneNumberForm.Call (page, externalOptionArgument, number);
```

In this listing, we create a new `WxeCallOption`, or, more precisely, a `WxeCallOptionsExternal` (a sub-class thereof). This new object will signal that we wish to open the `PhoneNumber` form in a new browser window. The parameter "_blank" specifies which window shall be the target ("_blank" means a new window, "_parent" a parent window. "_self" defeats the purpose of external somewhat, because it means the target is the calling window -- and the calling window will be closed when you are done with it. Check your HTML reference if all of this is new for you.)

That single option is not any good without a wrapping argument. That's what the statement

```
var externalOptionArgument = new WxeCallArguments ((Control) page,
                                                    externalOption);
```

is for. It wraps the `externalOption` created before in a `WxeCallArguments` instance. This instance is then passed to the invocation of `EditPhoneNumberForm.Call`.

If you type in this code, your `EditPhoneNumberForm` should open in a new window.

When we try the same modification for our `EditLocationForm` in `NewLocation...`, we meet some resistance, because, unlike the `EditPhoneNumberForm`, `EditLocationForm` returns a `Location` object.

Returning complete objects from an external page/function has become problematic since re-motion 1.11.14, because internal improvements prohibit this practice since then. This shortcoming will be fixed in the near future.The next example demonstrates how to use external windows for page/functions that return complete domain objects.

For this to work,

- change the type of the `EditLocationForm's` return value from `Location` to `ObjectID`
- store the object-id in the `ReturnValue` instead of the `Location`
- retrieve the `Location` instance with `GetObject()` to store it in the `LocationField`
- and, of course, pass the `WxeCallOptionsExternal` to `EditLocationForm.Call()`

Changing the return type of the EditLocationForm

As always, this must be done in the in the re-call header. The item that must be changed here is `<ReturnValue type="Location" />`, i.e. to

```
<ReturnValue type="ObjectID" />
```

New and improved header:

```
// <WxeFunction>
// <Parameter name="obj" type="Location" />
// <ReturnValue type="ObjectID" />
// </WxeFunction>
```

Storing the object-id in ReturnValue

The ReturnValue shall read like this in order to return the object's ID:

```
protected void SaveButton_Click (object sender, EventArgs e)
{
  if (SaveObject ())
  {
    ReturnValue = obj.ID;
    Return();
  }
}
```

Retrieving the Location instance with GetObject()

The new `EditLocationForm.Call()` (in `EditPersonControl.cs`) currently looks like this:

```
LocationField.Value = EditLocationForm.Call (WxePage, null));
```

but it is supposed to look like this:

```
LocationField.Value = Location.GetObject(EditLocationForm.Call (WxePage,
                                                                null));
```

Now this invocation needs an extra parameter with the `WxeExternalCallOptions`.

Instantiating and passing WxeExternalCallOptions

This works exactly as the example in the previous section for clickable phone-numbers in an external window. You can copy/paste the code if you are too impatient to type. Instantiation:

```
var externalOption = new WxeCallOptionsExternal ("_blank");
var externalOptionArgument = new WxeCallArguments ((Control) page,
                                                    externalOption);
```

Now you have to supplement that `externalOptionArgument` to the aforementioned `Call`, so that the entire line looks like this:

```
LocationField.Value = Location.GetObject(EditLocationForm.Call (WxePage,
                                                   externalOptionArgument,
                                                                null));
```

This completes our detour to external windows. The following sections assume that you have changed the code back to its original form (i.e. without call-options arguments, and with the `Location` return type back in place).