

***Programmer's Guidelines for Development of
Software within COBIK & Laboratory for
Multiphase Processes***

Revision 1.4, December 2012.
(Revision 0: Dec. 1010)

Igor Grešovnik

Contents:

1	<i>Introduction.....</i>	<i>1</i>
2	<i>Programming Style.....</i>	<i>2</i>
2.1	<i>General Programming Style.....</i>	<i>2</i>
2.1.1	Prerequisites.....	2
2.1.2	Naming Conventions	3
2.2	<i>Documenting Code by Comments</i>	<i>3</i>
2.2.2	Documentation Comments for Classes, Methods, Properties, Derived Types, etc.	4
2.2.3	Building Code Documentation for C# development projects	7
2.3	<i>Organization of Code.....</i>	<i>8</i>
2.3.1	Individual levels of code	9
3	<i>Team Work</i>	<i>11</i>
3.1	<i>Common Services</i>	<i>12</i>
3.2	<i>Structure of the Code Repository</i>	<i>13</i>
3.2.1	Checking out the Code	14
3.3	<i>Using Subversion Code Repositories.....</i>	<i>15</i>
3.3.1	Precautions to Avoid Obstructing other People's Work	16
3.3.2	Treatment of MS Word Documents.....	22
3.3.3	Troubleshooting Subversion (SVN).....	24
3.3.4	Things You Should not Do or You Should Do with Care.....	24
4	<i>Miscellaneous.....</i>	<i>25</i>
5	<i>Sandbox</i>	<i>Error! Bookmark not defined.</i>

1 INTRODUCTION

The present document contains coding standards and guidelines for development of software in Prof. Božidar Šarler's groups at COBIK & University of Nova Gorica. Beside a set of rules that we will stick with, the document also provides various information relevant for members of the team, such as instructions for use of common services (e.g. the Subversion repository).

In order for team work on software development to run smoothly, some minimum set of standards must usually be set up. The programming style should meet the criteria of good object oriented design, which must be learned from books and acquired through practical work. Since programming style is always an individual thing to some extent, it can not be strictly prescribed. Beside taking care of the quality of your own product, team work also requires a certain level of discipline and compliance with a number of logical rules of conduct. Not to go into details with pretty much obvious stuff, let's just mention the following one:

If you can't help, at least don't obstruct.

The above applies to all forms of not being able to help (lack of knowledge, lack of willingness to help, sluggishness, exhaustion, bad mood, having own problems - just name it). Simply keep in mind that the team is not just you – otherwise it wouldn't be called team.

Reading:

- [Harness the Features of C# to Power Your Scientific Computing Projects](#)

2 PROGRAMMING STYLE

2.1 *General Programming Style*

It is necessary that all people performing development work on the code maintain some minimum standard regarding the programming style. Programming style must be in line with basic rules of object oriented programming (OOP)¹ and each developer is responsible for generating clear, well designed and sufficiently documented object oriented code.

The above request is inevitable in the environment where it is expected that serviceable code is produced. By definition, such code must be modular, maintainable, reusable and scalable. It must be possible for other people to continue or use one's work without unnecessary overhead in introducing to the code. By current modern standards, it should be possible for anyone that is skilled in programming and knows well the subject of the code, to start contributing on the code without undergoing additional training.

2.1.1 Prerequisites

Anybody involved in code development is responsible for acquiring the necessary general programming knowledge (e.g. [3]-[5]), solid knowledge about the programming language of choice (e.g. [6]-[9] for C# and .NET, [10]-[14] for C++) and the development environment used. Knowledge of tools that are used to support work in the team must also be acquired (such as using Subversion, see e.g. [15]). Beside that, programming always requires experience that can only be acquired by practical work. Any individual programmer is therefore also responsible for going into coding problems persistently and on continuous basis in order to improve skills. Checking out and expanding illustrative examples obtained from the internet is a very good practice (even the most experienced programmers do that).

Whenever the course of work allows that, code reviews by more experienced developers should be practiced. Reviewers' remarks should be carefully considered and corrective actions taken if necessary. We want to tend to collaborative organization of work where everybody is open for questions as well as advice from colleagues.

¹ While there exist programming paradigms others than OOP (e.g. functional, logic, imperative, declarative, constraint, concurrent...), many of which are beneficial in certain areas of application, we will usually not need to adopt paradigms that lie much beyond OOP.

2.1.2 Naming Conventions

Names should be chosen to be as descriptive as possible. Often a well written code can be understood without reading any comments if the names are chosen well. Use long enough names in order to be descriptive, but don't exaggerate! Robotic-style names such as "Y_ROL_O" are outdated and must not be used. Use e.g. RollCenterDistanceY instead¹. With the autocomplete function nowadays available in practically all IDEs, using moderately long names will not affect your coding speed but will improve readability a lot.

Otherwise, try to stick with naming conventions that are used as standard for the programming language of choice (see e.g. [1]-[2]). These things are not there to force you into something but to increase mutual readability of people's code, although there is no intention to strictly enforce every detail of universally accepted conventions.

Almost everywhere nowadays there is an agreement that in case of composed names, each constituent word starts with an uppercase letter, which greatly increases readability. The first letter of a name may be an exception. In C#, for example, names of local variables (defined within function bodies) and function arguments should begin with a lowercase letter in order to distinguish them from public class members. Auxiliary class variables (usually defined as private) should begin with an underscore followed by a lower case letter; in this way it is difficult to mix them up with anything that is intended to be less internal. Classes, functions and properties should normally begin with an uppercase letter. You can make exceptions to this rule when you want to warn users of your code that there is something special about a given entity (e.g. that a class or a function is defined only temporarily for testing, which should be documented in comments anyway, but everybody will more easily focus attention if the name is chosen in a non-standard way).

2.2 Documenting Code by Comments

It is important to document any piece of code in such a way that users as well as other developers of this chunk of code can easily establish what the code does, how it is used, which is the meaning of exposed methods, properties and data, etc.

The main way of code documentation will be through comments of classes and methods.

If some method implements complicated algorithms that are not so obvious at a first glance, you should insert some comments between the code in order to make obvious what given portions of code do. Try to be concise but descriptive with such comments. You can put such comments in separate lines above the line or (more often) a group of lines of code that you intend to comment, or you can add short comments at the end of the line to be commented. With the first style comments it is common to form full sentences, capitalized and ended by colon. The second style comments are usually inserted without capitalization and without full sentence structure (such comments are

¹ Also a bit peculiar, since in modern programming languages you'll treat and group vector variables as vectors, and you will seldom define variables for individual components.

common e.g. in commenting variable declarations). Comments (of type `/* */`) can also be inserted in the middle of lines, e.g. to comment actual function parameters in long function calls.

2.2.1.1 Task Comments

Comments starting with the “TODO” string have a special meaning. They denote tasks that still need to be done. You can locate such comments in the code by selecting “View/Tasks List” from the main menu in Visual Studio.

Whenever you haven't completed something in the code, you should denote this with such comment. This is a notice to other programmers that things in a given portion of code don't function yet completely, that this is known (it is not a bug) and there is intention to return back to it and fix it.

Typical situations when you need to insert TODO comments are when you still don't have solutions to particular problems or when you don't have time to polish things, and you only bring the code into a condition when it can do things you need for your current tasks. You should avoid such situations however, because completed partially finished tasks gets more expensive in the future.

2.2.2 Documentation Comments for Classes, Methods, Properties, Derived Types, etc.

See also:

- [XML Documentation comments](#)

For commenting constituent parts of code structure, use documentation comments. These comments are inserted automatically in C# by inserting three slashes (`///`) and pressing <Enter>. This inserts a template already containing the most common XML tags used for automatic creation of documentation and in code balloons shown in the IDE when hovering a mouse pointer over the appropriate symbol.

2.2.2.1 Denoting Authorship in Documentation Comments

From comments it must be obvious who and when (approximately) designed, modified and updated specific classes, methods, properties, etc.

Authorship tags should be inserted below the comment that documents the specified symbol (class, method, property, etc.).

The general form is

```
/// $A <author_acronym0> <time01>, <time02> ..., <time0N>; <author_acronym1>  
      <time11> <time12> ... <time1M>; ...
```

Author acronyms are agreed within the group of developers. They should be short (to save typing) but distinctive (such that reader of the code can easily figure out who has worked on the specified piece of code). There must be a file in each code repository containing a table of author acronyms and corresponding full names (and possibly other personal data), or a reference to such a table. For example, a root directory of a code repository may contain the file *authors.html*, with approximately the following contents:

Author acronyms used within the code comments:

```
Igor = Igor Grešovnik, ajgor@lvs.com, tel. (+386) 1 3873 879.  
Robert = Robert Vertnik, r.vertnik@lvs.com, tel. . (+386) 1 3873 878.  
GKosec = Gregor Kosec, r.vertnik@lvs.com, tel. . (+386) 1 3873 877.  
Unknown = any unknown author.
```

Time marks are composed of abbreviated three letter month marks and two-digit year written together. Several time marks in a row are separated by spaces. Examples are Jan09 (for January 2009) or Oct10 (for October 2010). The first time mark must tell when the specific author has created or first updated the corresponding item (class, method, etc.), and others denote when subsequent important updates were made by this author. If there are several authors that were working on the specific item then they must be listed subsequently, each one with his or her own time marks, and separated by semicolon (;). Different authors should be listed in chronological order of their first involvement with the specified item.

For time marks, the special mark 'xx' can be used to denote that something was initiated by somebody before time marks were first inserted in the specific comments. This is used e.g. when some programmer has coded a given item but he or she didn't insert any author mark. If you update the code after that you can insert the author mark where you state yourself and the time(s) when you modified the code, but before that you should state the original author with "xx" for time mark (because you don't know at what times the original author has created and updated the specified item). If you don't even know who the original author was, use the "Unknown" keyword in place of the author's acronym. You can even use the xx time mark for yourself if you are not sure when you have created some item.

Examples:

```
/// $A Igor xx; Robert Nov10 Jan11;  
  
/// $A Unknown xx; Igor Jul09;  
  
/// $A Igor xx Feb09 Oct10 Dec10;
```

You can also use author marks more loosely in the middle of the code. There you can use only the \$A tag and your author's acronym followed by a less formal description, e.g.

```
i = i+1; // Corrected, $A Robert on January 12 2010, (orig. i = j+1;)
```

Author should be stated at least for each class.

Author marks can be omitted in comments of methods, properties, etc. In this case it is understood that the given method has been created and updated by the same author as the class containing the method.

It is not necessary to insert time marks for each minor update you make in the code. Be concise but provide some basic information on when and by whom things were created, modified or updated. Two basic purposes of author marks are giving you the credit for your work on the code and to enable users of the code to establish whom they can contact for additional information.

2.2.2.1.1 Examples:

```
/// <summary>Interface for classes that implement blocking until a specified condition  
is met.</summary>  
/// $A Igor Jun09 Feb10 May10; Stanislav Mar10 Sep10;  
public interface IWaitCondition : ILockable  
{  
    ...  
}
```

From the above example it is seen that Igor has first defined the *IWaitCondition* interface in June 2009, and has introduced some substantial modifications in February and May 2010. Stanislav has also worked on this interface in March 2010 and September 2010.

```
/// <summary>Used internally for locking access to internal fields.</summary>  
/// $A Igor xx Apr10 Jun10; Stanislav Mar10 Sep10;  
protected object InternalLock { get { return internallock; } }
```

From the above example it is evident that Igor has created the property called *InternalLock* but didn't denote when (maybe he inserted the author's mark only for the containing class). In March 2010 Stanislav modified the property. He knew that Igor was the original author but didn't know when he created the property, so he inserted xx in place of Igor's time mark, and added his own mark for his March 2010 modification. In June 2010, Igor modified something else in property definition and added the appropriate time mark in his part. Then in September 2010 Stanislav modified another thing and denoted this with an additional time mark in his author's mark. The author's mark evolved as follows:

At unknown time in the past, Igor creates the property but does not insert author's mark:

```
/// <summary>Used internally for locking access to internal fields.</summary>  
protected object InternalLock { get { return internallock; } }
```

In March 2010 by Stanislav (creates the author's mark):

```
/// $A Igor xx; Stanislav Mar10;
```

In April 2010 supplemented by Igor:

```
/// $A Igor xx Apr10; Stanislav Mar10;
```

In June 2010 supplemented by Igor:

```
/// $A Igor xx Apr10 Jun10; Stanislav Mar10;
```

In September 2010 supplemented by Igor:

```
/// $A Igor xx Apr10 Jun10; Stanislav Mar10 Sep10;
```

2.2.2.2 Detailed Guidelines for Author Marks

2.2.3 Building Code Documentation for C# development projects

The documentation is automatically generated from the code and from specially tagged comments. The documentation resides in the

```
<.../workspace>/doc/codedoc
```

directory where <.../workspace> is your workspace directory where you have working copies of the SVN repository directories.

This directory contains a HTML file with links to documentation and project files and code for generation of the documentation. It also contains configuration files necessary to automatically generate documentation by using the appropriate tools.

The documentation itself is excluded from SVN repository because it contains only automatically generated files, it is large and would load the server unnecessarily.

All instructions for generation of code documentation are included in the html file that contains links to documentation.

2.2.3.1 Some information information about Doxygen

Remark:

This text is not necessary for understanding how to generate your local version of documentation. All necessary instructions are in the HTML file containing documentation links.

Doxygen can be used to automatically generate HTML (or other form of) documentation from the documentation comments in source code. In order to use the tool, you must download the following software:

- [Graphwiz \(Windows installers here\)](#)
- [Doxygen \(download here\)](#)

A general overview of how to use *Doxygen* to generate documentation of your source code can be found in these tutorials:

- [Doxygen - Getting started](#)
- [Doxygen Manual](#)

In order to create a new configuration file for another documentation project, either copy another configuration file and change the relevant entries that must be different for new documentation, or create a new configuration file by typing the following command in the command prompt:

```
doxygen -g <filename>
```

Configuration files can be easily edited by any text editor. Alternatively, the graphical front-end can be used by

```
doxywizard <filename>
```

It is better to edit the configuration files directly by a plain text editor, however, because dialog boxes for choosing files and directories don't work very well in *doxywizard* (e.g. it is not straight forward to choose relative paths or to start with existing path).

After a configuration file is generated, it can be run in order to generate the documentation. This is done by typing the following command:

```
doxygen <filename>
```

2.3 Organization of Code

Code must be organized into a logical and clear hierarchical (tree-like) structure.

There will be several levels of code:

- Testing code that is not yet mature for inclusion in official repositories
- External general purpose libraries
- Internal basic libraries (those for which the group holds complete control over development)
- Domain specific libraries
- Application specific libraries
- Applications
- Customized applications - to meet specific customer requirements

The code in each level will also be structured with respect to platform requirements. For example, C# code that uses libraries available on Windows but not generally available on the Mono platform should be packed to separate libraries that can be easily excluded from the repository when libraries and applications are ported to another platform (such as Linux cluster or a Unix-based supercomputer).

2.3.1 Individual levels of code

2.3.1.1 External general purpose (basic) libraries

These libraries should be included in a special location, which is many times close to the location of internal basic libraries, where all higher level code can access them.

These libraries are considered something you can take and use without thinking too much how it is structured. In order to be treated in this way, these libraries must satisfy certain conditions:

- Their domain is not very specific and one can expect that several applications can benefit from use of the libraries (not necessarily at this moment, but maybe later).
- They must be stable enough (well tested, with few major bugs, with well established and – by expectations - relatively constant interface).
- They must be easily transferrable across different platforms.
- Their license must be such that it does not restrict any intended way of use¹.

2.3.1.2 Internal basic libraries

These are libraries that are developed or co-developed in-house and satisfy similar conditions as the external basic libraries. It is usually beneficial to open development of such libraries (if this would not incur additional housekeeping costs) to attract development potential or additional users that may find more bugs or contribute useful advice for enhancement or addition of features.

2.3.1.3 Domain specific libraries

These are libraries for which is clear that they can only be used (within our group) in narrow domains, such as libraries of CAD tools or finite element libraries or utility libraries for meshless methods. It is advisable to design a branched (but not too much) directory structure for development code in such a way that these libraries are not mixed with more basic libraries, but are close only to the code that will actually use them. In this way developers will not need to download everything from code repositories when working on specific applications and the code will be better structured.

2.3.1.4 Application specific libraries

Some utilities will be used only for specific individual applications. In OOP approach, code should be well structured at every level, therefore it is usually a good practice to separate even individual applications into smaller entities – at least one library module that can be compiled separately, and the top-level manipulation part.

¹ These conditions usually imply free open source license that is not GPL.

2.3.1.5 General applications

These are general applications such as optimization server or a mesh-free analysis application. Things as simulators can often consist of a number of separate applications such as pre-processor, pure analysis module (that just reads the input, performs simulation according to that input, and outputs the required results).

Many times these connected groups of applications will share a large portion of common library code and pure application code will be pretty small. When the code is well organized in such hierarchical way it is easy to produce customized applications (e.g. demonstration software or software tailored to customer or project needs) atop of the systematically improved and extended codebase.

2.3.1.6 Customized applications

These are specialized applications that are created for a given special purpose, such as on order from an industrial customer, for a project, or to widely distribute a demonstration code that popularizes work of the group.

In the case of commercial software for a specific customer, beside using extensive portions of common code base (libraries and sometimes also some general applications), these applications may consist of extensive portions of code that is produced specially for the specific application. Many times such code may contain things that are considered a trade secret of the customer, therefore the top-most part of such code must be clearly separated from other code (sometimes it is required that access to the code is granted in a very restrictive basis).

2.3.2 Organization of Code Projects

Code projects must be organized in such a way that people can easily work in team, projects are adapted to storage in central SVN repositories, code can be easily tested, project data is separated from code, code is easily transferable between different computers and even between different platforms, etc.

In general, the following criteria should be met by code organization:

1. When code is checked out on a different machine, it can be readily compiled, linked and run without any adjustments on the local system.
2. Even when the code is checked out on a computer with different platform than the one the code was created on, it should be easy to make it work. This should not require more than installation of the appropriate development environment and execution of some scripts.
3. Nothing produced by compiling or running the code should be committed to SVN.
4. Tests on the code can readily be performed by anyone.

In order to easily meet the above criteria, we will stick with some simple rules outlined below.

All the code must be put in a single directory named *workspace*. Each code project must have a specified location relative to the *workspace* directory. In this case, relative references

between any two parts of the code will always work. Where to put the *workspace* directory on the local disk is a personal choice of each developer. It must not matter.

If we need some data for testing the code, this data should reside in a directory named *data* somewhere within the code project directory. This data should be put on repository.

If testing (running) the code generates any data on the disk, such data should be put into the directory named *testdata* and located somewhere in the related project directory. This directory should not be put to SVN.

All paths for compiler and linker must be relative. When using external libraries, all the necessary files (e.g. .dll, .lib) must be stored in some directory named *bin*. Since these files are platform dependent, there must be some other directory containing these files for all platforms in use, and shell scripts must be provided for each platform that copy the appropriate files to that *bin* directory. In this way, when checking out the code on a different platform, one will only need to check out everything necessary and run the script that copy binary files for the appropriate platform to the bin directory where they are referenced by the code.

When using different integrated development environments, there must be only one source code for all IDEs in use, i.e., the same code will be referenced from different IDEs in different ways (dependent on the IDE). However, auxiliary files specific to different IDEs will be different, and they will be included in SVN repository.

Any production-level code (i.e. code used for anything else than just for testing and demonstration) must be designed in such a way that its data obtained from the disk can be located anywhere on the file system where user has sufficient permissions. It is allowed, however, that data location for a more complex application or set of applications is specified by a designated system variable. In particular, location of the data must be unrelated to location of the application's executable or libraries (failure to comply with this rule points to extremely unprofessional and ignorant attitude of developers and project leads). When code uses more complex but relatively fixed data structure, it is recommended that all data is located in a directory tree with prescribed structure (or with some mechanisms of dynamic pointing to relative locations), such that specifying the location of the data root directory uniquely defines location of all other data obtained by application from the hard disk.

For internal use of the developed software, there should be means of quickly generating test applications and sharing these applications and data between team members. Larger software projects should include standardized subprojects for generating such ugly hard-coded applications. To share data for internal projects, the directory named ***workspaceprojects*** will be used. This directory must always be located beside the *workspace* directory (i.e. contained in the same directory). In this way, relative paths from code to data in *workspaceprojects* will be constant, which will enable easy sharing the project data and working on the same projects by team members. Subdirectories of *workspaceprojects* that are shared between two or more team members can be put to SVN.

3 TEAM WORK

3.1 Common Services

Software development in a team can not be imagined any more without at least two technical services:

- **Revision control system** (“Subversion” in our case)
- **Issue tracking system** (“Bugzilla” in our case)

Revision control systems enable multiple developers to work concurrently on the same code, submit their modifications to a central repository, update their working copies of code with modifications done by the others, revert unintended or adverse changes, restore any file to any previous revision stored in the central repository, etc.

Issue tracking systems enable reporting bugs and other issues related to software or team work in general, defining tasks, tracking status of individual tasks, assigning and re-assigning tasks, creating reports, sending filtered notifications (e.g. via e-mail) about any changes in the status of work, etc.

This section contains basic information for access to common services. Subsequent sections provide more detailed information and describe rules that must be obeyed when using the services. Table 1 contains user data (except passwords) for common services. Below there is detailed information for accessing common services.

Note: Access to the common services is currently possible only from within the local network.

3.1.1.1 Subversion access:

Central code repository in a Subversion server can be accessed by clients such as [TortoiseSVN](#) or [AnkhSVN](#). Details about how to copy software from central repository to your working copy can be found in subsequent Sections, as well as addresses of individual directories. Base access to the repository is through the following address:

- <https://192.168.1.34:8443/svn/>

Type this in the address bar of your browser and press <Enter>. A log in window appears where you can type in your username and password. When you log in, you will first see a list of repositories, which appear as directories that you can browse. Note that actual useful directories can be embedded in several levels of “trunk/” directories. This is so because of the way in which Subversion organizes data, and which enables users to create branches on which they can experiment without disrupting work of the others.

3.1.1.2 Bugzilla access:

Bugzilla can be operated entirely via a web interface that is manipulated in your web browser. In addition to that, notifications via e-mails can be set up for different kinds of events (related usually to changes of status of bugs/tasks). Below is the address for accessing Bugzilla services (currently this does not work):

- <http://192.168.1.34:8080/>

Load this address in a web browser and click "Log In". In the search field you can quickly search for tasks, bugs and other issues of your interest.

Table 1: User data. Username is used for accessing the Subversion repository and should be the same as username used for computer accounts. In Bugzilla, an e-mail address is used as user name. Author's acronym is a mark that is used to denote authorship of classes, methods, etc., within the source code and in some other locations such as discussion forums.

Name	Username	Author's acronym in code	e-mail for Bugzilla	
Božidar Šarler	bozidar	Bozidar	bozidar.sarler@ung.si	
Igor Grešovnik	igor	Igor	igor.gresovnik@cobik.si	
Robert Vertnik	robert	Robert	robert.vertnik@ung.si	
Gregor Kosec	kosec	Kosec	grega.kosec@gmail.com	
Katarina Mramor	katarina	Katarina	kmramor@gmail.com	
Gregor Košak	gkosak	Gkosak	gregor.kosak@gmail.com	
Agnieszka Lorbicka	agnieszka	Aga	zuzanna1981@wp.pl	
Umut Hanoglu	umut	Umut	Umut.Hanoglu@ung.si	
Qingguo Liu	liuqingguo	Qliu	liuqingguo1980@gmail.com	
Tadej Kodelja	tadejk	Tako78	tadej.kodelja@cobik.si	

3.1.1.2.1 See also:

- [Revision control](#)
 - [Subversion](#)
- [Issue tracking system](#)
 - [Bugzilla](#)
 - [Trac](#)

3.2 Structure of the Code Repository

Development code is organized in the following structure:

- **base** – base libraries
 - **iglib** – the basic utility library (covers all basic and general utilities and functionality that could be used among different application)
 - **simulation** simulation framework
-

- **testdevelop** test projects for prototyping and testing of ideas
 - **develop_nafems** – first attempt to design a general simulation framework, based on Robert's NAFEMS¹ example.
 - **example_nafems** – Robert's code for the NAFEMS simple heat conduction example
- **doc** - documentation

You should store these directories with the same relative paths in a separate directory, preferably named *workspace*.

3.2.1 Checking out the Code

All directories from the SVN server should be checked out to standard locations within the *workspace* directory that contains your local working copies of code projects, documents, etc. Table 2 lists addresses of principal directories in the SVN repository and their corresponding locations in the local *workspace* directory. Each principal directory listed in the table must be separately downloaded (checked out) from the repository, and its containing directory must first be created within the *workspace* directory if it does not exist. When checking out a directory, check carefully that you insert the correct URL or repository and, in particular, the correct checkout directory. A frequent error is that one checks out something into a directory that is already under version control (such directories contain a subdirectory named ".svn", which can not be seen in Windows Explorers unless you switch on the option for displaying hidden files). Such errors can lead to complications that are hard to resolve.

In order to perform the SVN checkout, you must install **TortoiseSVN** (available for Windows OS; on other systems you must use some other SVN client such as SmartSVN).

Procedure for checking out project directories from the SVN server is as follows:

- At suitable location, create the directory that will contain your working copy of the code. It should preferably be named **workspace** and should not contain other things.
- Open the SVN repository browser. Open Windows Explorer, right-click on the workspace directory, select SVN Checkout.
- In the Checkout box that appears, URL of the directory location in the SVN repository and the checkout directory (the directory in which contents are saved) must be specified. See Table 2 below for a list of URLs and locations within the *workspace* directory.
 - **Warning:** be very careful when specifying the checkout directory. Wrong paths will make relative paths between referenced projects invalid.
- Click OK. The complete directory structure of the chosen directory will be downloaded from SVN repository to the directory of choice. Wait until transfer completes and click OK again.

¹ NAFEMS - organization that sets and maintains standards in computer-aided engineering analysis (especially the finite element analysis).

Table 2: Repository URLs and corresponding checkout directories.

URL of repository	Checkout directory
https://192.168.1.34:8443/svn/doc/doc/trunk/lab/trunk	...\workspace\doc\lab
https://192.168.1.34:8443/svn/doc/doc/trunk/codedoc/trunk	...\workspace\doc\codedoc
https://192.168.1.34:8443/svn/archive/doc/trunk/literature/trunk	...\workspace\doc\literature
https://192.168.1.34:8443/svn/develop/develop/trunk/lib/trunk	...\workspace\develop\lib
https://192.168.1.34:8443/svn/develop/develop/trunk/shell/trunk	...\workspace\develop\shell
https://192.168.1.34:8443/svn/test/tests/trunk/csharp/trunk/	...\workspace\tests\csharp
https://192.168.1.34:8443/svn/test/tests/trunk/testsvn/trunk/	...\workspace\tests\testsvn

In addition, there is a separate repository in which the C# course material is located. The address is

<https://192.168.1.34:8443/svn/test/tests/trunk/csharp/trunk/>

If you would like to use this material, you can check it out into the workspace directory (e.g. to ...\\workspace\\tests\\csharp), but you can also use some other directory of your choice.

There is also a directory where you can practice use of the Subversion:

<https://192.168.1.34:8443/svn/test/tests/trunk/testsvn/trunk>

Read the readme file before using this directory.

A separate repository named *supplementary_projects* is prepared for project data directories contained in the *workspaceprojects* directory (this directory must be contained in the same directory as the *workspace* directory). Each user and group has its own directory in this repository. Address of these directories are of the form

https://192.168.1.34:8443/svn/supplementary_projects/workspaceprojects/trunk/<user>/trunk,

where <user> is the name of the user or group that owns the directory. Each user can put his/her project directories that are included in the *workspaceprojects* directory to the appropriate directory in the SVN repository. Local relative path within the *workspaceprojects* directory is specified by the creator of project directory. Each of these directories should be checked out separately and not as part of a larger directory structure. For example, Tadej has created and uses a project directory at the location

...\\workspaceprojects\\12_02_paper_neural_process_chain_model

He has imported this directory to SVN repository at the location

https://192.168.1.34:8443/svn/supplementary_projects/workspaceprojects/trunk/tadej/trunk/12_02_paper_neural_process_chain_model

3.3 Using Subversion Code Repositories

Although Subversion is a stable and mature revision control system, there are situations where use of the system is troublesome and where some conflicts are difficult to resolve. This Section describes how the system should be used in order to avoid causing problems to other programmers and to yourself. Use of revision control systems requires some discipline and you should always think whether your actions could cause problems to other developers.

3.3.1 Precautions to Avoid Obstructing other People's Work

You should never check in parts of code unless all dependent projects can be compiled!

When all projects dependent on code that you've modified are compiled without errors, you should check in all modified code at once.

You should also not check in parts of code for which you know that they don't function correctly. If your modifications caused malfunction of some code then you should correct this first and then check in.

When you intend to work on parts of code for which it is likely that others will also work on, you should try to find out if somebody else intends to work on particular parts of code at that time. In such a case try to coordinate with these people in order to avoid conflicts that are difficult to resolve.

When you work on critical parts of code on which a lot of other code depends, you should **inform others** about that. The same is true when you intend to do any major refactoring. In such a case you might consider creating a new branch, work on it and merge it with the trunk when you finish your work.

Avoid locking files! Locking is reserved for really rare situations and you should almost never use it. When a file is locked, nobody else but the owner of the lock can commit changes to that file (others can still read the file from the central repository).

Locking can provide some protection against difficult merge conflicts when a user is making radical changes to many sections of a large file or group of files. However, if the files are left exclusively locked for too long, other developers may be tempted to bypass the revision control software and change the files locally, leading to more serious problems.

3.3.1.1 Ignore Lists for Project Directories

In your working copy (i.e. copy on your local disk that is under version control) of project directories, you should set ignore property for ***.exe, *.dll, *.pdb, *.suo *.bak** files and for **bin**, and **obj** and **testdata** directories. By internal convention, **testdata** is the name used for directories where test data is kept, which is generated programmatically and may change frequently because tests are performed on that data.

The purpose of using ignore patterns is two-fold:

- Other users won't override your personal settings (e.g. those related to your platform specifics or those defining the layout of your development environment).
- Time is saved because the large generated files such as .obj, .exe, .dll, etc., are not transferred over network and copied each time you are performing commits and updates.

This can best be done in TortoiseSVN on Windows. Procedure is as follows (just make sure first that you have TortoiseSvn installed, this is seen in context menu when you right-click on any directory or file):

- Open root directory of the project or solution in Windows Explorer. The directory should be marked by one of the SVN icon overlays, e.g. a green hook or a red exclamation mark.
- **Right-click** on the **root directory** where project is installed, and select **TortoiseSVN/Properties** from the context menu.
- In the *Properties* box that opens, do one of the following:
 - If the box already contains the property named **svn:ignore** then just double-click the corresponding entry.
 - If the box does not contain **svn:ignore** then click **New**, then select **svn:ignore** under Property name, and insert values under Property value.
- In the *Edit Properties* box that opens, add corresponding entries to be added to ignore list under *Property value*. You can separate entries by spaces or newlines.
 - Add your ignore pattern, e.g.: ***.dll *.exe *.pdb *.suo *.bak */bin */obj**
- Check the **Apply properties recursively** box and click the **OK** button.
- Click the **OK** button again (in the *Properties* box).

If you browse your projects' working copy, all the above mentioned files that should be ignored by SVN should not have TortoiseSVN's icon overloads visible in explorer. If this is not true for some files, you should check if the containing directory has the **svn:ignore** property set.

Warning:

Do not add *.exe and *.dll to ignore lists for directories that are intended for third party libraries and applications. Such directories should be separated from root directories of development solutions.

3.3.1.1.1 Setting up Global Ignore

You can set a global ignore pattern In TortoiseSVN. The global ignore pattern will prevent the specified files showing up e.g. in the commit dialog. Files specified here will also be ignored in import. The pattern you set up here will not affect other users because it is not versioned and will exist only in your personal settings for the TortoiseSVN application.

The global ignore pattern should be more restrictive than the ignore patterns that you set up on different directories because it will be used for committing and importing files to the central repository for all different projects, not only specific ones.

The procedure is as follows:

- Open Windows Explorer
- Press Alt-F for the File menu, then choose TortoiseSVN/Settings.
- In the *Settings* box, choose general and then insert the pattern under *Global ignore pattern*.
 - Example of global ignore pattern: *.bak bin obj *.suo. Usually some pattern is already set up when you install TortoiseSvn.

3.3.1.1.2 More on Ignore Patterns for C# Projects

Dependent on the type of the projects, one may use different ignore patterns.

One simple but probably quite effective global ignore pattern I've [found on the internet](#) is the following:

- *\\bin* *\\obj* *.suo *.user *.bak **.ReSharper** **_ReSharper.**. StyleCop.Cache

A more elaborated pattern:

- *.o .lo .la ## .rej .rej .~ ~ .# .DS_Store thumbs.db Thumbs.db *.bak *.class *.exe *.dll *.mine *.obj *.ncb *.lib *.log *.ldb *.pdb *.ilk .msi .res *.pch *.suo *.exp .~ .~ ~. cvs CVS .CVS .cvs release Release debug Debug ignore Ignore bin Bin obj Obj *.csproj.user *.user ReSharper. *.resharper.user

In ignore patterns, different items must be separated by spaces. Patterns use filename globbing where files can be specified by using wildcards. The following characters have special meaning:

- * - matches any string of characters, including the empty string
- ? - matches any single character
- [...] - matches any one of the characters enclosed in the square brackets. A pair of characters separated by '-' matches any character lexically between the two characters (including the characters themselves).

Pattern matching is **case sensitive**.

You should **not include path information in patterns**. Pattern matching is intended to be used against plain file names and directory names. For example, if you want to ignore complete *bin* directories, just add *bin* to the ignore list.

If you want for example exclude *debug* directories included in *bin* directories, but not *debug* directories included elsewhere, you should achieve this by using the *svn:ignore* property on all *bin* directories. According to documentation, there is no reliable way to achieve this using global ignore patterns.

3.3.1.2 Copying Directory Structures to a Location under Version Control

Say you have a working copy of a complex source code, possibly containing several solutions and a large number of projects. Then you want to include existing branched directory structure at the desired location within that working copy and put it under version control.

Rule of thumb is **that you should not (recursively) copy the directory directly into the working copy**. You should instead copy it on the subversion server (e.g. by using a TortoiseSVN repository browser) and get it into the working copy via SVN update.

Recommended steps are the following:

- **Copy** the directory **to** the appropriate location within the **central repository** on the SVN server that manages your working copy.
 - In **Windows Explorer**, **open the directory** that you want to include in the working copy (you must first have TortoiseSVN installed).
 - **Open Tortoise's Repository browser** (in Windows Explorer, right-click on some directory, choose TortoiseSVN/Repo-browser).
 - In repository browser, navigate to the location where you want to include your directory structure.
 - Copy the directory to the server by **dragging it from Windows Explorer to the appropriate location in repository browser**.
- **Update working copy** (In TortoiseSVN, right-click on some directory that will contain the added directory structure, choose TortoiseSVN/). The newly included directory structure should be copied to the corresponding location in the working copy.
- **Recursively add the *svn:ignore* property** on the newly added directory structure.
 - In windows Explorer, right-click the root of the newly included directory, choose TortoiseSVN/Properties, and recursively add the *svn:ignore* list (details are in Section 3.3.1.1).

Important remarks:

If you are copying the directory that was previously under source control, you should create a copy and recursively *remove all .svn subdirectories* from the directory structure.

It may be a good idea to remove user specific and generated files before including the directory, such as *.exe, *.dll, *.pdb, *.suo. Be careful though, since some directories may contain .exe and .dll files that are not generated by the contained projects but are third party libraries and applications referenced by the directory.

When copying a small number of files in a single directory, it may be easier to copy them to the working copy and then add them to SVN's central repository. This is the case only when it is easy to locate and select all new files in Windows explorer. In this case, the procedure is as follows:

- Copy files to the appropriate location within the working copy.
- Select the newly copied files in Windows Explorer.
- Add files to repository (right-click selected files, choose TortoiseSVN/Add, confirm addition).
- Commit addition (right-click selected files, choose SVN Commit).
- You can check in repository browser whether the files were actually added to the central repository (right-click the directory in the working copy where copied files

are located, select TortoiseSVN/Repo-browser). In repository browser, refresh the view first (right-click a directory, choose Refresh), otherwise the changes may not be visible.

3.3.1.3 Modifying Solutions under Version Control in the IDE without SVN Plugin

Sometimes you will work with integrated development environment (IDE) that does not have a SVN client integrated. For example, you can not integrate SVN clients with Visual Studio Express because the express editions do not support plug-ins.

In such a case, changes in solutions and projects will not be automatically reflected in the SVN structure of the working copy, e.g. new files will not be versioned automatically. You will have to put changes under version control manually.

3.3.1.3.1 Adding New Projects or Files within Projects

When a new project is added, do the following:

- Save changes in your IDE in order to make sure that the changes are actually reflected in the file system.
- Check for changes in a directory that for sure contains all newly added files (right-click the directory, choose TortoiseSVN/Check for modifications). If you are not sure, do it on the root directory of your working copy (this will take longer and it may be more difficult for you to filter files that are added as a consequence of your changes).
- In the *Working Copy* box that opens, check the “**Show unversioned files**” option.
- Add and commit new files:
 - Select all files that are related to addition of the new project (they should be listed with *non-versioned* status). Right-click the selected files and choose Add, then click OK.
 - Right-click the files again, select Commit.
- Commit changes on the root directory containing the whole solution (right-click the directory in Windows Explorer, select SVN Commit). You must do this because the solution file also changes when a new project is added.

3.3.1.3.2 Adding a New File within an Existing Projects

When just adding a file to an existing project that is already under version control, the procedure is similar as when adding a project (Section 3.3.1.3.1) except that you can omit committing the root directory (since all changes are limited to project directory, unless you have re-defined the project directory structure in such a way that it deviates from the default one).

3.3.1.3.3 Renaming Files within an Existing Projects

When renaming a file, the procedure is similar as when adding a file, except that you must now do two things: remove the file with the old name and add the file with the new name. Similar can be carried out when renaming multiple files or when renaming a project.

- Save all in the IDE such that all changes are reflected in the file system (if you use Visual studio, press Ctrl-Shift-S).
- Check for changes in the directory that for sure contains all renamed files (right-click the directory in Windows Explorer, select TortoiseSVN/Check for Modifications).
- Check the “*Show unversioned files*” option. File(s) with original name will appear with the “*missing*” status. . File with new name will appear with the “*non-versioned*” status.
- Right-click the missing file (original name(s)) or select multiple files and right-click selection, choose Delete.
- Right-click the non-versioned file (new name(s)) or select multiple files and right-click selection, choose Add.
- Click the OK button twice.
- Perform Commit on the directory containing all renamed files (right-click the directory in Windows Explorer, choose SVN commit).
- Just for any case, you may check the state of the central repository in a repository browser (right-click the directory in Windows Explorer, choose TortoiseSVN-Repository browser). Don't forget to refresh the directory of interest (right-click, Refresh).

3.3.1.3.4 Deleting a Project or a Directory Structure

When removing a project in Visual Studio, the project is removed from solution but files are not actually deleted. You just need to commit changes in the solution file in this case, and you must manually remove project directory if this is what you want to do.

When you recursively delete a complete directory sub-structure from your working copy, you should carry out the following procedure:

- Check for modifications in the directory that contained the deleted directory (right-click the directory in Windows Explorer, choose TortoiseSVN/Check for modifications).
- If there are too many files listed in the “Working Copy” box that opens, you can uncheck the “Show unversioned files” (since the directory that you have deleted was managed by SVN).
- Right-click the deleted directory (if you have deleted multiple directories, select them first and then right-click the selection; the directory should be listed with the “*missing*” status), choose “Delete”.
- Right-click the directory (or selection in case of multiple directories), choose *Commit*.
- Confirm by clicking the OK button twice.

Remark:


Sometimes it is enough to perform Commit on the directory that contains the deleted directory (or directories).

3.3.2 Treatment of MS Word Documents

When you modify a MS Word document from the SVN repository, you must ensure that modifications are added to the repository document and that modifications added by other users at the time you were editing your local working copy of the document are not overridden by you. Follow this procedure in order to add contents to a Word document:

1. Update your working copy of the document. In Windows Explorer, open the directory containing the working copy of the document. You must have TortoiseSVN installed and the “doc” subdirectory downloaded (checked out) from the central repository to your workspace directory (the local directory on your disk containing working copies of repository files; see remarks below). Right-click the document and choose “SVN Update” from the menu, confirm by clicking the OK button.
2. Open and edit the document (your working copy).
3. Save and then close the document when finished.
4. Commit the changed document back to the central repository. In Windows Explorer, open the directory containing the working copy of the document. Right-click the document and choose “SVN Commit” from the menu. If commit fails then follow the conflict resolution procedure described below.

When somebody else has committed changes to the document in the time between your last update and commit attempt, the commit will fail because the document is in conflicted state (you'll get a descriptive error message). In this case, follow the resolution procedure below:

1. Perform update on your working copy of the document (right-click in Windows Explorer, select “SVN Update”). Update will also fail but you have to perform it in order to proceed.
2. Try to perform commit again (right-click in Windows Explorer, select “SVN Commit”). “Commit” window will open where the document is shown in red and tagged “conflicted” under the “Text status” column.
3. Right-click the document in the “Commit” window and select “Edit conflicts”. This will open a copy of the document where changes between the document in the repository and your working copy are shown, ready for rejecting and accepting individual changes.
4. Use buttons to accept or reject changes () in order to merge your modifications into the working copy, then save the document at the location of your working copy. You can either override (replace) the working copy or (for better safety) use “Merge changes into the existent file”.

- You must accept or reject all changes before overriding the working copy with the open document.
 - Reject all changes that mean deletion of stuff you have entered (changes are marked with respect to the repository document, not your working copy, therefore your changes are marked as deletions, which you must reject).
 - Accept all changes that are not yours, since these are changes that were committed by other users while you have edited the document.
5. Right-click the document in the "Commit" window again, but now select "Resolve conflict using 'mine'". Make sure that the document with accepted and rejected changes has been solved as working copy before you do that (otherwise you can override somebody else's work)!
 6. Click OK in order to commit the document.
- In rare occasions it can happen that this last commit fails in spite of the fact that you have resolved the conflict by using your changes. This happens e.g. if during the (usually short) time between resolution and commit somebody else commits changes to the document. In such a case, just repeat the resolution procedure:
 - a. Perform SVN Update on the document, followed by SVN Commit.
 - b. In the commit window, perform "Edit conflicts" on the document.
 - c. Accept and reject changes as appropriate, and overwrite the working copy with the resulting document.
 - d. In the commit box, perform "Resolve conflict using mine" on the document.
 - e. Commit the document.

Remarks:

In order to download the "doc" directory from the central repository into your *workspace* directory (i.e. to create a working copy of the directory), do the following:

- Open the workspace directory in Windows Explorer. The workspace directory is the location on your local disk where you decide to store working copies of the repositories' files (e.g. d:\userss\workspace).
- In Windows Explorer (you must have TortoiseSVN installed), right-click on an empty space within the workspace directory, and choose "SVN Checkout" from the menu.
- Under "URL of repository", insert "https://192.168.1.34:8443/svn/doc/doc/trunk/lab/trunk/" .
- Under Checkout directory, insert "<workspace_dir>\doc\lab" and press the OK button. You have created a working copy of the "doc" directory from the repository. The document of interest is located in "... \doc\discussions\choice_of_simulation_platform.doc".

With the above procedure, it is recommendable to make an additional local copy of your working copy of the document. If you override your changes in the working copy, you can still merge these changes into it from this additional copy.

3.3.3 Troubleshooting Subversion (SVN)

SVN clients are sometimes pretty capricious. Because it is easy to do something wrong when you are not completely familiar with some (possibly buggy) features, we will here maintain a collection of unusual behavior and possible remedies.

Remember, your first care when using SVN is not to make something that would cause problems to other users!

3.3.3.1 Problems with TortoiseSVN

Sometimes you can not commit some items. It may help if you perform Update on these items first, and try to commit again. It sometimes helps.

If you get a message that you don't have permission to commit changes, contact SVN server administrator to check whether there is actually something wrong with permissions!

3.3.4 Things You Should not Do or You Should Do with Care

Do not delete files directly from repository! Clients might have problems with this. If you really need to do something like this, ask administrator to do it!

See also:

- [Version Control System](#), a Wikipedia article
- [Apache Subversion](#), a Wikipedia article
- [Apache Subversion](#), official home page
- [Version Control with Subversion](#), a book.
- [Comparison of Subversion clients](#), a Wikipedia article
- [An Introduction to Subversion](#) (centered around use of TortoiseSVN)
- [TortoiseSVN Tutorial](#)
- [TortoiseMerge Tutorial](#)

3.3.5 Short Note on Treating Data in Development and Production Projects (in Slovene)

V direktorijh znotraj *workspace* so podatki praviloma v direktorijih z imeni *data* in *testdata*. V *testdata* je lahko karkoli, vendar morajo biti direktoriji s tem imenom v seznamu *svn:ignore* in njihova vsebina ni vključena v SVN (kar pomeni, da se tudi ne avtomatsko bekapira). V direktorijih z imenom *data*, katerih vsebina gre na SVN, ne smejo biti stvari, ki so rezultati programov in jih

načeloma lahko velikokrat generiraš s poganjanjem. Tu so lahko kakšne manjše datoteke, ki vsebujejo nastavitve ali vhodne podatke za teste, ki se izvajajo s programi, predvsem za [unit teste](#). V primeru unit testov so tipično vhodni podatki v direktoriju *data*, rezultati pa v *testdata*, medtem ko je lahko primerjalna verzija rezultatov ročno skopirana v *data*, če ne zavzame veliko prostora in se redko spreminja.

Za ostale namene se za podatke uporabljajo direktoriji izven direktorija *workspace*. Zato, da so fiksirane relativne poti glede na generirano izvršno kodo v *workspace*, se po dogovoru uporablja direktorij ***workspaceprojects***, ki je vsebovan v istem direktoriju kot *workspace*. Ta direktorij se uporablja predvsem za projekte in testne podatke, pa tudi za druge situacije, ko je potreba po deljenju podatkov preko SVN. Direktorije znotraj *workspaceprojects* lahko po želji vključim na SVN, vendar bodo v ločenem repozitoriju in zanje ne morem zagotoviti enake stopnje varnosti kot za razvojno kodo (lahko jih kvečjemu avtomatično kopiram na dva računalnika - .34 in .37, ostalo je prepuščeno tebi).

Datoteke, ki jih generira prevajalnik, ne spadajo na SVN. V posebnih primerih se lahko ročno kopirajo v kak direktorij vključen v SVN, kadar je to potrebno za delovanje kode, ki se razvija na več razvojnih platformah.

Režim je prirejen omejenim možnostim, ki so mi na voljo za vzdrževanje servisa, in se ga je nujno držati, da bo zadeva zadovoljivo delovala na daljši rok. Kar se tiče čiste kode, ni nobenih omejitev glede velikosti (najbrž tudi ni bojazni, da bi kdo spisal več kot 10 MB na leto), pri avtomatično strojno generiranih podatkih pa je ne glede na velikost pravilo, da nikoli ne gredo v iste direktorije na SVN kot koda.

4 MISCELLANEOUS

References:

- [1] Doug Lea: Draft Java Coding Standard. Electronic document, available at <http://g.oswego.edu/dl/html/javaCodingStd.html>.
- [2] C# Coding Standards & Best Practices. Electronic document, available at <http://www.dotnetspider.com/tutorials/CodingStandards.doc>.
- [3] Object-oriented programming on Wikipedia. Read also all pages that are linked from introduction! http://en.wikipedia.org/wiki/Object_oriented_programming.
- [4] Design pattern (computer science) on Wikipedia. http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29.
- [5] Model–view–controller on Wikipedia. An important example of design pattern. <http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>.
- [6] Faraz Rasheed: C# School. An excellent book for learning C#, available at http://www.programmersheaven.com/ebooks/csharp_ebook.pdf.
- [7] MSDN .NET (includes searchable reference documentation for c#). Available at <http://msdn.microsoft.com/en-us/library/w0x726c2.aspx>.
- [8] Srečo Uranič: C# .NET. Slovenian book on C#, available at <http://uranic.tsckr.si/C%23/C%23.pdf>.
- [9] C Sharp (programming language) on Wikipedia. http://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29
- [10] MSDN Visual C++. Available at <http://msdn.microsoft.com/en-us/library/60k1461a.aspx>.
- [11] Bjorn Fahler: An Introduction to C++ programming. Available at http://www.computer-books.us/cpp_1.php.
- [12] Peter Mueller: An Introduction to OOP Using C++. Available at http://www.computer-books.us/cpp_4.php.
- [13] C++ on Wikipedia. <http://en.wikipedia.org/wiki/C%2B%2B>.
- [14] Marshall Cline: C++ FAQ. Available at <http://www.parashift.com/c%2B%2B-faq-lite/>.
- [15] Nikolai Shokhirev: Practical guide to subversion on Windows with TortoiseSVN. Suitable for users of Subversion on Windows, available at <http://www.shokhirev.com/nikolai/programs/SVN/svn.html>.
- [16] Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato: Version Control with Subversion. Suitable for SVN administrators, available at <http://svnbook.red-bean.com/en/1.5/svn-book.html>

References

- [17] Igor Grešovnik: Administrators' rules for COBIK servers, detailed report on use of the COBIK servers, internal report, 2011.
- [18] Igor Grešovnik: Coordination of software development in COBIK and Laboratory for Multiphase Processes. Treatise, COBIK, 2011.
- [19] Igor Grešovnik: *IoptLib*, electronic document at <http://www2.arnes.si/~ljc3m2/igor/ioplib/>.
- [20] Igor Grešovnik: *IGLib.NET Code Documentation*, electronic document at http://dl.dropbox.com/u/12702901/code_documentation/generated/iglib/html/index.html.

