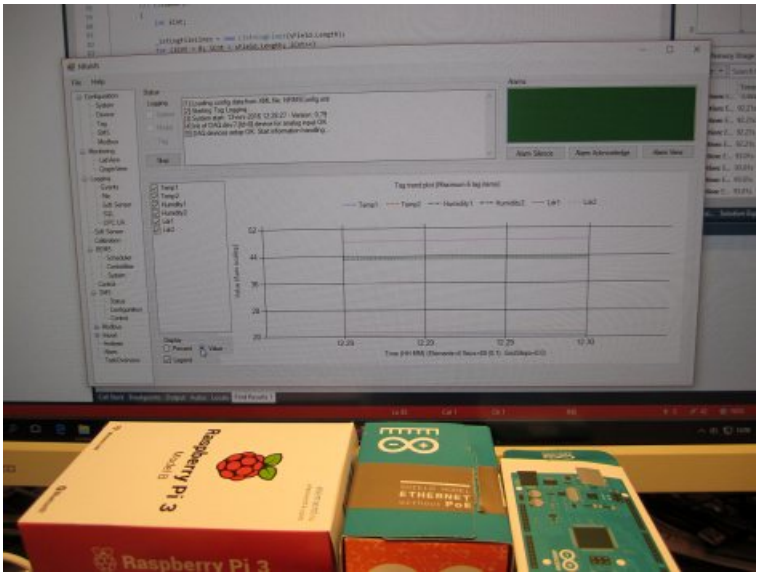
University of South-Eastern Norway

Faculty of Technology, Natural Sciences and Maritime Sciences (TNM) - Campus Porsgrunn

Industrial IT and Automation (IIA) master programme

Course IIA1319 - Software Engineering



Lecture notes for object-oriented analysis, design, and programming using UML and C#

©Nils-Olav Skeie

5th January 2022

Contents

Preface **viii**

Nomenclature **x**

I Elaboration Phase - Iteration #1 (EP-I1) **1**

1 Introduction **2**
1.1 Development process . . . 4
1.2 Software engineering and development . . . 5
1.2.1 Object Oriented Development . . . 5
1.3 Object-Oriented Analysis and Design (OOAD) . . . 8
1.3.1 Why learn OOAD? . . . 8
1.3.2 Requirement . . . 9
1.3.3 Analysis . . . 9
1.3.4 Design . . . 9
1.4 Example . . . 10
1.4.1 Analysis: . . . 11
1.4.2 Design . . . 12
1.4.3 Programming . . . 13
1.5 Case studies . . . 15
1.5.1 Mobile Phone . . . 15
1.5.2 Washing machine . . . 16
1.5.3 I/O Measurement system . . . 16
1.6 Software simulators . . . 16
1.7 Inception phase . . . 17

2 Software development process **18**
2.1 Introduction . . . 18
2.2 Development processes overview (Lifestyle models) . . . 18
2.2.1 Waterfall . . . 19
2.2.2 Spiral . . . 19
2.2.3 Rapid prototypes . . . 19
2.2.4 Agile Development processes . . . 20
2.2.5 Summary . . . 21
2.3 Unified Process (UP) . . . 22
2.3.1 Introduction . . . 22
2.3.2 UP main phases . . . 23
2.3.3 UP summary . . . 25
2.4 Development process summary . . . 25
2.5 UP in this document . . . 26

3 Version Control System **27**
3.1 Introduction . . . 27
3.2 Motivation . . . 27
3.3 Structure of a version control system . . . 28
3.4 Overview of some version control systems . . . 28

3.5 Git . . . 31

4 The C# programming language **33**
4.1 Introduction . . . 33
4.1.1 Type-safety . . . 34
4.1.2 Fully Object-oriented . . . 34
4.1.3 Simplified syntax . . . 34
4.1.4 Standards . . . 34
4.1.5 Components . . . 35
4.1.6 C# versus Visual Basic . . . 35
4.1.7 Naming convention . . . 35
4.2 Integrated development environment . . . 35
4.2.1 Visual Studio IDE Application . . . 36
4.2.2 Windows Forms or Console application . . . 36
4.3 Requirements for an OO programming language . . . 43
4.4 .NET programming languages . . . 43
4.5 C# control statements . . . 43
4.5.1 The if statement . . . 45
4.5.2 The switch statement . . . 45
4.5.3 Iteration statements . . . 46
4.6 Namespace . . . 47
4.6.1 The namespace . . . 47
4.6.2 Assemblies . . . 48
4.6.3 Setting the root namespace . . . 48
4.6.4 Using classes . . . 48
4.7 C# class . . . 48
4.7.1 Attributes . . . 48
4.7.2 Modifiers . . . 48
4.8 C# constants, fields, properties and methods . . . 49
4.8.1 Constants and fields . . . 49
4.8.2 Property . . . 49
4.8.3 Methods . . . 50
4.8.4 Method parameters . . . 50
4.8.5 Method or Property . . . 50
4.8.6 #define . . . 50
4.9 C# coding summary . . . 50
4.9.1 General statements . . . 50
4.9.2 Data types . . . 51
4.9.3 Operators . . . 51
4.9.4 Program control . . . 51
4.9.5 Arrays and strings . . . 53
4.9.6 Classes, objects, and methods . . . 54
4.9.7 C# class example . . . 54
4.9.8 Constructor . . . 56
4.9.9 Method parameters . . . 56
4.9.10 Return value . . . 57
4.9.11 Method overloading . . . 57
4.9.12 Inheritance . . . 57
4.9.13 Partial Class . . . 59
4.10 Testing . . . 59

II Construction Phase - Iteraction #2 (CP-I1) **61**

5 Introduction **62**
5.1 Introduction . . . 62
5.2 Why use UML? . . . 62

6 Analysis **63**
6.1 Introduction . . . 63
6.1.1 Business use cases . . . 63
6.1.2 Specification and requirements . . . 64
6.2 Requirements . . . 64
6.2.1 Challanges in collecting Requirements . . . 64
6.2.2 Collecting the requirements . . . 65
6.2.3 Using FURPS+ for collecting requirements . . . 66
6.2.4 Prototypes . . . 68
6.3 Use case . . . 68
6.3.1 Introduction . . . 68
6.3.2 Use case diagram . . . 68
6.3.3 How to make a use case diagram . . . 69
6.3.4 Exercises . . . 70
6.4 Domain model . . . 72
6.4.1 Introduction . . . 72
6.4.2 Find conceptual/"functional" classes . . . 73
6.4.3 Usage of a domain model . . . 73
6.4.4 Making a domain model . . . 73
6.5 Analysis steps of the development process . . . 74
6.6 Use Case Document . . . 75
6.7 System Sequence Diagram (SSD) . . . 75
6.7.1 Introduction . . . 75
6.7.2 Information in a System Sequence Diagram (SSD) . . . 76
6.7.3 How to make a SSD . . . 77
6.7.4 Some SSD examples . . . 77
6.8 Operation Contracts . . . 80
6.9 OOA Summary . . . 80

7 UML diagrams **82**
7.1 Visual Studio . . . 82
7.2 StarUML . . . 83
7.2.1 Use case diagram . . . 83
7.2.2 Domain model . . . 86
7.2.3 System Sequence Diagram . . . 86
7.2.4 Sequence Diagram . . . 86
7.3 Rational Rose . . . 86

8 Design **91**
8.1 Introduction . . . 91
8.1.1 System Architecture . . . 91
8.2 Object design . . . 93
8.2.1 Design Models . . . 94
8.2.2 Class and object . . . 95
8.2.3 CRC cards . . . 97
8.3 UML Interaction diagrams . . . 97
8.3.1 Sequence diagram . . . 98
8.3.2 Collaboration Diagram (Communication Diagram) . . . 99
8.3.3 Summary . . . 99
8.3.4 Pseudo coding . . . 100
8.4 UML Class Diagrams . . . 100
8.4.1 Special type of classes . . . 101
8.4.2 Relationship between interaction diagrams and class diagrams . . . 102
8.4.3 Example of code . . . 102
8.5 Designing objects with responsibility . . . 104
8.5.1 Responsibilities and Responsibility Driven Design . . . 104
8.5.2 Patterns . . . 104
8.6 GRASP . . . 105

8.6.1 Controller pattern . . . 105
8.6.2 Information Expert pattern . . . 105
8.6.3 High Cohesion pattern . . . 106
8.6.4 Low Coupling pattern . . . 107
8.6.5 Creator pattern . . . 107
8.7 Object design examples with GRASP . . . 108
8.7.1 Grasp Examples . . . 109
8.7.2 Application startup . . . 111
8.8 Modeling with sequence diagram . . . 112
8.9 Mapping designs to code . . . 112
8.9.1 Design the Class Diagram from the Interaction Diagram . . . 112
8.9.2 Implementation model (from UP) . . . 114

9 Programming 115
9.1 Exceptions . . . 115
9.2 Threading . . . 118

10 Data communication in C# 123
10.1 Introduction . . . 123
10.2 Serial port data communication . . . 123
10.2.1 Introduction . . . 123
10.2.2 Physical layer . . . 124
10.2.3 Asynchronous serial communication . . . 124
10.2.4 Communication protocols . . . 125
10.3 Serial port programming in C# . . . 132
10.3.1 Introduction . . . 132
10.3.2 Three tier architecture . . . 133
10.3.3 Serial port library . . . 134
10.3.4 Terminal console application . . . 138
10.4 Network communication programming in C# . . . 142
10.4.1 Introduction . . . 142
10.4.2 OSI model . . . 142
10.4.3 TCP/IP protocol . . . 142
10.4.4 Socket programming in C# . . . 143
10.4.5 Console test application . . . 145
10.4.6 GUI Chat application . . . 151
10.4.7 Security . . . 159

11 Testing 160
11.1 Introduction . . . 160
11.2 Principles . . . 161
11.3 OOAD based testing . . . 161
11.4 Test cases . . . 162

12 Case Studies 163
12.1 Introduction . . . 163
12.2 Mobile Phone . . . 163
12.2.1 Analysis . . . 163
12.2.2 Software development . . . 163
12.3 Washing Machine . . . 164
12.3.1 Introduction . . . 164
12.3.2 Elaboration - Analysis . . . 164
12.3.3 Construction - Analysis . . . 171
12.3.4 Construction - Design . . . 172
12.3.5 Construction - Coding . . . 176
12.3.6 Construction - Testing . . . 176

13 Summary 179

III Construction Phase - Iteration #3 (CP-I2) 180

14 Introduction 181

15 Analysis 183
15.1 Relating Use cases . . . 183
15.1.1 include . . . 183
15.1.2 extend . . . 185
15.2 Domain Model Refinement . . . 186
15.2.1 Guidelines . . . 187
15.2.2 Motivation for superclass . . . 187
15.2.3 Motivation for subclasses . . . 188
15.2.4 Example . . . 188
15.3 Package design . . . 189
15.4 Architectural analysis . . . 190

16 Design 192
16.1 Introduction . . . 192
16.2 GRASP: More objects with responsibilities . . . 192
16.2.1 Polymorphism pattern . . . 192
16.2.2 Pure Fabrication pattern . . . 194
16.2.3 Indirection pattern . . . 194
16.2.4 Protected Variations pattern . . . 195
16.3 Applying GoF Design Patterns . . . 195
16.3.1 Adapter pattern . . . 195
16.3.2 Factory pattern . . . 195
16.3.3 Singleton pattern . . . 195
16.3.4 Strategy pattern . . . 197
16.3.5 Composite pattern . . . 197
16.3.6 Fasade pattern . . . 197
16.3.7 Observer pattern . . . 198
16.3.8 Model View Controller pattern . . . 199
16.3.9 Alarm system . . . 199
16.4 Activity diagrams and modeling . . . 199
16.4.1 Activity diagrams . . . 200
16.4.2 Elements in an activity diagram . . . 200
16.4.3 Guidelines for using an activity diagram . . . 200
16.4.4 Activity diagrams in StarUML . . . 202
16.5 State machine diagrams and modeling . . . 202
16.5.1 State diagram . . . 202
16.5.2 Elements in a state diagram . . . 205
16.5.3 Guidelines . . . 205
16.5.4 State diagrams in StarUML . . . 206
16.5.5 State machine Exercise . . . 206
16.6 Arduino development . . . 207
16.7 Logical Architecture Refinement . . . 212
16.7.1 Deployment diagrams . . . 212
16.7.2 Evaluation of the logical architecture . . . 212
16.8 Summary of using patterns . . . 215
16.8.1 *GRASP patterns overview* . . . 215
16.8.2 *GoF patterns overview* . . . 216

17 Data Structures and Algorithms 218
17.1 Introduction . . . 218
17.2 Arrays . . . 218
17.3 Sorting . . . 220
17.3.1 Bubble Sort . . . 220
17.3.2 Selecting Sort . . . 221
17.3.3 Insertion Sort . . . 222

17.3.4 Sorting comparison . . . 223
17.4 Searching . . . 223
17.4.1 Sequential Searching . . . 223
17.4.2 Find Minimum and Maximum Values . . . 223
17.4.3 Binary Search . . . 224
17.5 Stacks and Queues . . . 225
17.5.1 Stack . . . 226
17.5.2 Queue . . . 226
17.5.3 Ring Buffer . . . 227
17.6 Strings . . . 228
17.7 Lists . . . 228
17.8 Linked Lists . . . 229
17.9 Multitasking . . . 229
17.10Example . . . 230

18 Data Security 232
18.1 Introduction . . . 232
18.2 Software security . . . 232
18.3 Grading of information . . . 233
18.4 General Data Protection Regulation (GDPR) . . . 233
18.5 Encryption / Decryption . . . 233
18.5.1 Character rotation . . . 234
18.5.2 XOR encoding/decoding . . . 234
18.5.3 PGP . . . 236
18.5.4 Block chains . . . 237
18.6 Input fields for information . . . 238
18.7 External access . . . 238
18.8 Passwords . . . 239

19 Programming 240
19.1 Inheritance . . . 240
19.2 Interfaces . . . 244
19.3 API . . . 244
19.4 Robotic Process Automation . . . 244
19.5 Design of a measurement system . . . 245

20 Testing 248
20.1 Introduction . . . 248
20.2 Test Plan . . . 248
20.3 Test Levels . . . 250
20.3.1 Unit or Device Testing . . . 250
20.3.2 Module testing . . . 250
20.3.3 System Testing . . . 251
20.3.4 Examples . . . 251
20.3.5 Control system for train traffic . . . 251

21 Summary 253

IV Managing Software Projects 254

22 Introduction 255

23 Software development 256
23.1 Introduction . . . 256
23.2 Overview structure . . . 256
23.3 Layered design . . . 257
23.4 Component-Based Development . . . 257
23.5 Configuration . . . 257

23.6 Testing and debugging . . . 258
23.7 Comments . . . 259
23.8 Coding standard . . . 259
23.9 Bug tracking system . . . 260
23.10 Procedure based programming languages . . . 260
23.10.1 Analysis . . . 260
23.10.2 Design . . . 260
23.10.3 Programming . . . 261
23.10.4 Testing . . . 261
23.11 Reverse Engineering . . . 261

24 Guidelines for reports and projects 265
24.1 Reports . . . 265
24.1.1 Informal report . . . 265
24.1.2 Formal report . . . 265
24.1.3 References . . . 266
24.1.4 Article or paper . . . 266
24.2 Presentation . . . 266
24.3 Meetings . . . 267
24.3.1 Non formal meetings . . . 267
24.3.2 Formal meetings . . . 267
24.3.3 Effective meetings . . . 268
24.4 Projects . . . 268

V Appendixes 270

A Avoiding software development nightmares? 271

B GiT Command Overview 273

C C# Introduction presentation 276

D Code standard overview example 292

E C# examples 295
E.1 OOADP introduction source . . . 295
E.2 Sensor classes with Inheritance . . . 298
E.3 A Facade class in C# for the NI-USB devices . . . 300
E.4 Facade and Publisher/Subscriber patterns . . . 305

F Documents 309
F.1 Fully dressed use case document . . . 309

G OOAD software examples 310
G.1 Mobile Phone software OOAD . . . 310
G.1.1 Analysis . . . 310
G.2 I/O measuring device software . . . 314
G.2.1 Analysis . . . 314
G.3 Alarm clock . . . 315

H GiT installation 317

Preface

This document is used as lecture notes for the course Software Engineering, based on Object-oriented Analysis, Design, and Programming. This course is an introduction to object oriented analysis, design, and programming using *Unified Process* (UP), *Unified Modeling Language* (UML) and *C#* (C-Sharp) programming language. The main part of the course will be about analysis and designing the objects for the programming part, not the programming part itself. Some experience with programming is expected.

The contents of this course consist of lectures and mandatory assignments. The best way to learn object oriented analysis, design, and programming is by practice, given that the assignments will be an important part of the course. The mandatory assignments will focus on solutions for automation and process control systems (PCS), analyzing, designing and documenting a software system using text documents and UML diagrams. A development process is also important for defining the different steps in the development process, and when to make or use the different types of text documents and UML diagrams. By combining the UML diagrams with an IDE (Integrated Development Environment) tool, the UML diagrams will be used as a framework for making executable software applications. The programming language will be C# and the IDE will be the Microsoft Visual Studio, however UML and the OOAD (Object-Oriented Analysis and Design) concept can be applied to any object oriented programming language.

The syllabus for the course will be this document and the assignments. The textbook "*Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*" will be a good support of the analysis and design part. Both the 2nd edition (Larman 2002) and the 3rd. edition (Larman 2005) of the textbook can be used for this support.

The programming language C# is used for programming, for implementing a system from the analysis and design parts. Parts of the C# section in this document are collected from among others the Microsoft Developer Network[1] (MSDN) web site about the C# programming language, from several books, and the experience of the author. Several books exist regarding the C# programming language, two books that can be recommended with focus for Electronic Engineers are (Allwork 2009) and (Allwork 2010). This document will however cover the main sections of the C# programming language necessary for this course.

The lecturer has been programming since 1980, mainly in C and lately in C#. The first programming languages were Basic (Commodore Vic-20 and 64), Pascal (Turbo Pascal for MS-DOS) and Assembler (for Intel, AMD and Motorola CPUs). He has been the project leader of several software projects with more than 300.000 lines of source code using C, C++, and C# programming languages on operating systems like Concurrent DOS, MS-DOS, Windows, Unix, Linux, and Solaris. The experience after more than 20 years of OOADP is that it is a challenge to make the right OO design the first time, using more time in the OOAD phase, using less time in the programming phase, and a huge savings in the maintenance phase (much better with OO!). OOAD is normally overkill in small projects, but the knowledge is that small projects *always* grow to larger projects. So the advice will be to always use OOAD whenever you want to develop software!

Thanks to several students over the years giving me feedback on this document, a special thank goes to Ole Magnus Brastein for valuable discussions and feedbacks.

[1] Microsoft Developer Network (MSDN): msdn.microsoft.com (JUN-08).

The history of this document:

Revision	**Descriptions / Changes / Extensions**	**Who**	**Date**
0.1	Merged all lecture notes into a single document,	NOS	2006
0.2	Include more information about C#,	NOS	2007
0.3	Include information about using MS-Visual Studio 2008,	NOS	2008
0.4	Extend the information about design patterns,	NOS	2009
0.5	Include information about MS-Visual Studio 2010,	NOS	2010
0.6	Add more examples about measurement systems,	NOS	2011
0.7	Add a new introduction presentation of OOADP,	NOS	2013
0.8	Restructure, and add more information about testing,	NOS	2014
0.9	Add information about Kanban and Scrum, minor fixes.	NOS	2015
1.0	Add sections about measurement system and "Data Structure and Algorithms". Improve the structure of the document.	NOS	2016
1.1	Add Arduino section, test plan section and minor fixes.	NOS	2017
1.2	Extend the case study examples	NOS	2018
1.3	Changed name to "Software Engineering", course code to IIA 1319 Included a chapter about computer security with focus on software.	NOS	2019
1.4	Add section for data communication in C# and minor fixes.	NOS	2020
1.5	Update the version control, software development process section, software security section, and minor fixes.	NOS	2021

© Nils-Olav Skeie: Permission is granted to distribute single copies of this document for non-commercial use, as long as it is distributed as a whole document in its original form, and the name of the author is mentioned.

Nomenclature

This section gives a list of abbreviations used in this document.

Abbreviation	Meaning / Explanation
C#	C "sharp" programming language,
CBD	Component Based Development,
CIL	Common Intermediate Language,
CLI	Common Language Infrastructure,
CLR	Common Language Runtime,
CLS	Common Language Specification,
DLL	Dynamic Link Library,
ECMA	European Computer Manufacturers Association,
EXE	A filename extension denoting an executable file,
GUI	Graphical User Interface (UI - HMI),
HMI	Human Machine Interface (UI - GUI),
IDE	Integrated Development Environment,
IIA	Industrial IT and Automation,
MMI	Man Machine Interface (GUI),
MSIL	Microsoft Intermediate Language,
MS-VS	Microsoft Visual Studio IDE,
OCX	OLE custom controls,
OLE	Object Linking and Embedding,
OO	Object Oriented,
OOA	Object Oriented Analysis,
OOAD	Object Oriented Analysis and Design,
OOADP	Object Oriented Analysis, Design, and Programming,
OOP	Object Oriented Programming,
PCS	Process Control Systems,
RUP	Rational Unified Process,
SCE	System and Control Engineering,
SOA	Service Oriented Architecture,
TBD	Test Based Development,
UI	User Interface (GUI - HMI)
UML	Unified Modeling Language
UP	Unified Process
USB	Universial Serial Bus
VB	Visual Basic
XP	Extreme programming

Part I

Elaboration Phase - Iteration #1 (EP-I1)

Chapter 1

Introduction

Developing software will be more and more important when working with system design as the hardware modules are based more and more on standard modules. Any system will have most of the functionality and "intelligence" encapsulated by the software modules covering the problem domain for the system. The main focus in this document will be modules to be used in Process Control Systems (PCS) (Olsson & Piani 1998), including models, simulators and controller algorithms. Most of the models, simulators and algorithms must be developed using software and will require maintenance of this software. Automation is a consequence of the technological development with the need for more and more process information and control (Dickinson 2011). Modeling of a system and simulation of the behavior of the system are important parts in advanced Automation systems, to be able to predict any results and/or future behavior. Any model, simulator or controller algorithms must be implemented in some sort of software like spreadsheets, ready made packages and/or programming languages.

Spreadsheet gives an easy computation and graphics, but becomes complicated with requirements of several variable output responses. Spreadsheet requires less programming knowledge, but difficult to adapt to controller knowledge. Packages are based on a higher level of "visual programming" using interconnection of predefined blocks. Packages require some programming knowledge, however the predefined block will give additional help for the programming knowledge. Packages adapt easier to control knowledge. Programming languages requires much knowledge about the programming subject, and can give a very flexible system combined with controller knowledge. No matter of the selected implementation, the system must contain some sort of documentation.

The main focus in this document will be to give the readers more knowledge about the concept of developing software, to teach the readers how to become a software designer in addition to being a programmer. It is assumed that the readers have some basic knowledge about programming, knows how to make loops, conditions, and combine data and methods in a class. Developing software is much more then just programming, so the focus will be on software engineering and software development. Software engineering is more about the software development process while software development is using the software development process to develop software applications. As a system designer you should also be able to design software in such a way that a programmer can do the coding based on your design. The software development consists of transforming any specifications and/or requirements into a working application meaning that there must also be a development process describing the sub tasks for this transformation. This development process will be the main purpose of any software development project. Since more and more systems becomes computer based today, knowledge about and experience with software development part will be an important part of system development.

Inception is the startup phase of any software development project and is used to give a short introduction into the problem area, also called the problem domain. The inception phase will normally be a management decision to start a project or not. In this case will the inception phase be your decision about learning more about software engineering and software development. The elaboration phase is the next phase describing the startup of a project with project leader and project members. This introduction is the elaboration phase in this document and consists of the sections 1) Development process (Dybå, Dingsøyr & Moe 2004), (Kruchten 2000), 2) Software Engineering and software Development (Manassis 2004), 3) Object Oriented Analysis and Design (OOAD) (Oestereich 2001), (Page-Jones 2000), (Fowler & Scott 1997) 4) an example of using OOAD and C#, and 5) a description of the case studies used as examples in this document.

The examples for implementing the OO code will be C#, a fully object-oriented based programming

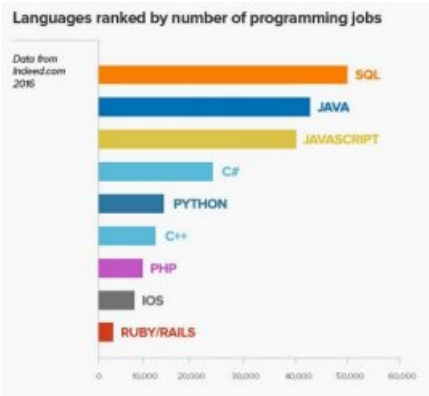


Figure 1.1: The most used programming languages in 2016.

Oct 2020	Oct 2019	Change	Programming Language	Ratings	Change
1	2	^	C	16.95%	+0.77%
2	1	v	Java	12.56%	-4.32%
3	3		Python	11.28%	+2.19%
4	4		C++	6.94%	+0.71%
5	5		C#	4.16%	+0.30%
6	6		Visual Basic	3.97%	+0.23%
7	7		JavaScript	2.14%	+0.06%
8	9	^	PHP	2.09%	+0.18%
9	15	^^	R	1.99%	+0.73%
10	8	v	SQL	1.57%	-0.37%
11	19	^	Perl	1.43%	+0.40%

Figure 1.2: The 10 most popular programming language in October 2020, according to the TIOBE Programming Comminity index (www.tiobe.com: Oct-20).

language. C# (C-Sharp) is a popular programming language, there exists compilers for many operating systems, and Figure 1.1 shows the most used programming languages in 2016.

Equivalent, developer hubs such as Stack Overflow and GitHub release data on the most popular languages on their platforms for 2018. These are 1) JavaScript used for jobs like Web developer, full-stack developer, and front-end developer, 2) Java used for jobs like software engineers, software architects and DevOps engineers, 3) Python used for jobs like Data engineer, full-stack developers, and software engineers, 4) PHP used for jobs like Web developer, 5) C# used for jobs like full-stack developer, .NET developer, and web developer, and 6) C++ used for jobs like embedded engineer, games engine software engineer, and system software developer. Also note that it is more easy to make production code in JavaScript, Java, C++ and C# than in Python, R, and Matlab (Cole 2018).

TIOBE, a company from the Netherlands working with software quality, is making the TIOBE Programming Community index which is an indicator of the popularity of different programming languages. Popular search engines such as Google, Amazon and YouTube, to mention a few, are used to estimate the index and are based on the number of skilled engineers world-wide, courses and third party vendors. The index is updated once a month and the index for October 2020 is shown in Figure 1.2.

The TIOBE Programming Community index has been updated for many years and Figure 1.3 shows the popular programming language world wide for 2002 to 2020. An updated version is available at www.tiobe.com (Oct.-20).

The coding syntax is very similar for C, C++, C# and Java and these figures shows that C# is a popular programming language. However, the important part is to understand programming, how to convert a challenge or a problem into programming code, not the programming language itself.

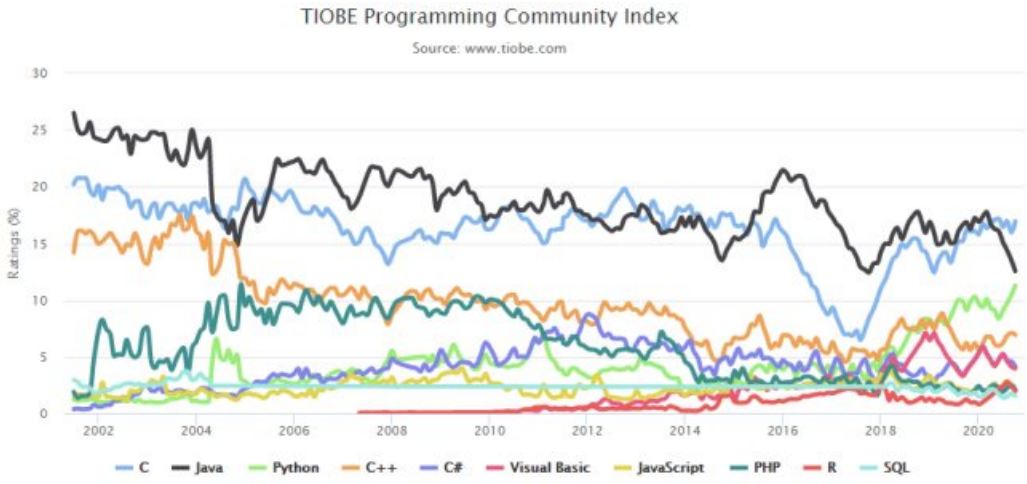


Figure 1.3: The most popular programming langaugae world wide according to the TIOBE Programming Community index for almost the last 20 years (www.tiobe.com: Oct-20).

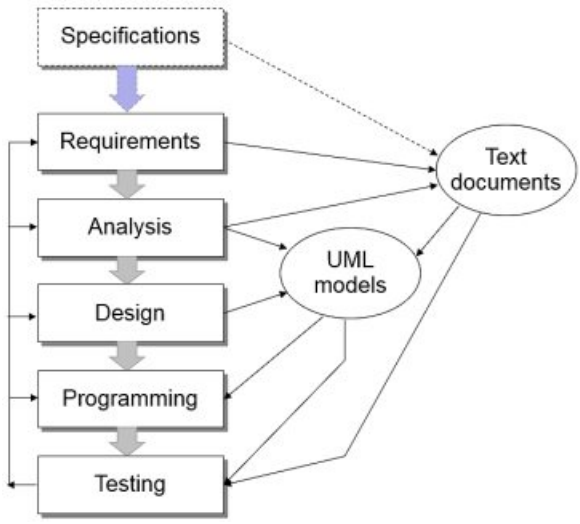


Figure 1.4: The main tasks in a software development process, and the relationships between these tasks.

1.1 Development process

The development process is a structured set of the activities, tasks and decisions for developing a system (Dybå et al. 2004) , (Douglass 1999). A system consists of both hardware and software, and the software development process is a structured set of the activities, tasks and decisions for developing software (Kruchten 2000), (Sommerville 2011), (Robertson & Robertson 2013), (Larman 2002), (Larman 2005). The main set of tasks in the software development process is shown in Figure 1.4.

Any software project starts with a specification, an overview description of what the software is going to do. Based on the specification the requirements must be collected. The requirements is more detailed description of the software functions and also the boundaries for the software application. The requirements will be the basis for the analysis, design, programming and testing of the software. The development of software started as combining several functions, often with global data. However as the size of software applications, the amount of memory and the speed of cpu increased, the usage of object oriented methods grow. In object oriented methods the objects are the main building blocks, with private data and public methods. In object oriented methods the functionality of the software is the goal and each object will be assigned a specific responsibility. An object is a software unit consisting of a set of data, and a set of methods to fulfil a specific responsibility. A software system consists of several (tens, hundreds, or even thousand of) objects cooperating to fulfil the system requirements.

Note that the figure is not including the maintenance of the software application which can be a large

part of a software system. The figure focus on the first version only, however in software development the maintenance is important, and the maintenance can go on for several decades. The is a good reasons for making a good design of the software from the start, and include design documentation of the software.

Object oriented software development also include component oriented development (Wang & Qian 2005) and service oriented architecture (SOA) (Jensen & Billington 2007). A software component consists of several objects to fulfil a specific requirement, more complex than what a single object can handle. A software component consists normally of a set of objects for a specific function or operation and can easily be integrated into a software application to extend the functionality. Service oriented architecture (SOA) is also based on objects, however the focus in SOA is to deliver some sort of service based on available information. SOA will deliver some sort of service based on information (can be both locally and remotely) instead of delivering the information. The service may be located locally on the computer, in the local network (intranet) or on internet as a cloud service, which may be the case for many Internet of Things (IoT) devices.

This document will focus on objects only, how to analyze, design, and implement a software with these objects, known as the OOAD. The programming language used for implementation will be C# (C "sharp"). The workflow of this document is divided into phases with several iterations, just like a software development process. An iterative software process will be used and the document is also based on an iterative workflow. Starting with development process and some basic steps in analysis, design, construction and testing in the first iteration, and allow for extensions in the following iterations.

1.2 Software engineering and development

Software development is an important part of software engineering being an systematic approach for designing, coding, testing and maintain software. Software engineering refers to the application of engineering principles to create software. Software development is mainly the designing, coding and testing phases. A process is necessary to describe the tasks and the sequence of these tasks in developing the software. Many development processes exists that can be used for developing software. The following steps are important in developing a software application and should be part of the software development process:

1. define a set of activities transforming the user requirements into a software application,
2. define WHO is doing WHAT, WHEN and HOW to develop a software application,
3. the type of activities like collection of requirements, analysis, design, programming, test, deployment, project management, configuration control, and documentation control.

This document will use the *unified process* as the software development process. The unified process is also named the Rational Unified Process (RUP). The unified process consists of:

1. using Unified Modeling Language (UML) as a tool for modeling and documenting the static and dynamic behavior of the software,
2. important aspects of this process are use case[1] driven, risk driver, architecture centric, iterative, and incremental.

Unified process is an use case driven process as the focus is use cases or user stories. A use case or user story focus on the functional requirements and define the workflow in the system. Iterative and incremental means that there will be a fixed set of activities (analysis, design, implementation and testing) that will be repeated again and again (iterative) until the software application is finished. The focus in every loop will be one or several use cases. The use cases and the unified process will be independent of the type of software development, in this document the focus will be object oriented development.

1.2.1 Object Oriented Development

Developing software will be to transfer the specification and requirements into a running software application using data and functions or methods. Each function or method has some sort of responsibility

[1] In Norwegian: use case = bruksmønster

to fulfil the requirements. Software development started out with functions and global data. When the size of the software is increasing, usage of global data will complicate the maintenance of the software. In object oriented development the responsibility will be assigned to the objects, not the functions or methods. An object consists of both data and methods, however the data will be private to the object, accessed only by methods. The usage of objects, private data and global methods only have shown to decrease the time for maintaining the software.

Object Oriented background

The object oriented development started with the programming language *Simula* in the 1960's (Simula−67). Simula was developed by Ole-Johan Dahl and Kristen Nygaard at Norwegian Computing Center (Norsk Regnesentral, NR), and was the first object oriented programming language for computers. An object oriented programming language requires higher cpu power from the computers so these were not useful for office computers until late 1990. Books about object oriented analysis and design also turned up in the beginning of the 1990s, and one good book about this subject from 1993 is "Principles of Object-Oriented Analysis and Design" (Martin 1993).

Object Oriented diagrams

One way of documenting Object-oriented software is by using Unified Modeling Language (UML). UML is a set of diagrams that can be used to document both the static and dynamic behavior of the software. UML was developed from a cooperation by "The three amigos"; Grady Booch, James Rumbaugh and Ivar Jacobsen. The contribution were 1) Booch: Commercial and technical: Booch '91, 2) Rumbaugh: Object Modeling Technique: OMT '94, and 3) Jacobsen: Use Cases: Object Oriented Software Engineering: OOSE '94. They published the first version of the Unified Method (UM) by Booch and Rumbaugh in 1995, version 0.8. Jacobsen joined the group in 1996 with use cases (from the telecom industry in Sweden) and they changed the name to Unified Modeling Language (UML). Version 1.0 of UML was taken over by Object Management Group (OMG)[2] for standardization in January 1997, and later several new versions of UML has been released. Version 2.0 is the version of UML used as bases for this document. Some of diagrams in version 2.0 that can be used for modeling the software are listed below:

Diagram	**Operation**	**Type**	**UML**	**Usage**
Activity	Shows behavior with control structure	Behavior	1.x	
Class	Show static structure of concepts, types and classes	Structure	1.x	**X**
Collaboration	Communications between objects	Behavior	1.x	**X**
Component	Structure and connections	Structure	1.x	
Composite	Runtime structure of a class	Structure	2.0	
Deployment	Shows physical layout of hardware components	Structure	1.x	**X**
Interaction	Shows how object collaborate in single use case	Behavior	2.0	
Object	Instance of a class	Structure	1.x	
Package	Shows classes and dependencies among them	Structure	1.x	**X**
Sequence	Sequential interaction between objects	Behavior	1.x	**X**
State	Shows how a single object behaves	Behavior	1.x	**X**
Timing	Time control interaction between objects	Behavior	2.0	
Use case	Shows the requirements from the users	Behavior	1.x	**X**

The Usage column shows the diagram that will be used in this document. The type of diagrams, the information in the diagrams, and when to make the diagrams are decided by the develop process. There exists a lot of different software development process, in this document the Unified Process (UP) will be used. This software development process is also called the "Unified Software Development Process" and is an iterative and incremental development process, with focus on 1) doing the development in steps, 2) feedback from the users while doing the development, 3) doing the steps with the highest risk as early as possible in the project, and 4) defining the usage of UML diagrams. This process is also said to be an use case driven process as the use cases describe how the software system shall work (as a black box) where each use cases are describing part of functionality in the software system. The use cases will be used by both the users, designers, programmers, and test people.

[2] See also www.omg.org

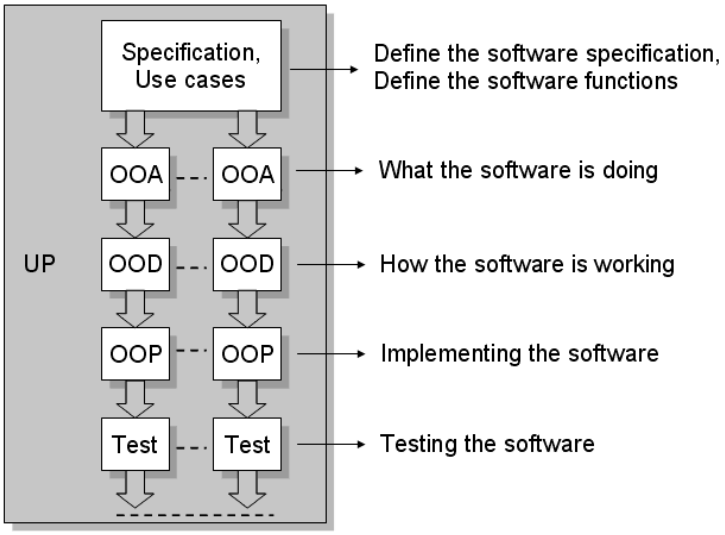


Figure 1.5: The relation between the software development process (UP) and Object Oriented Analysis, Design, and Programming.

For more information about OOAD and UML some of the following books (Manassis 2004), (Fowler & Scott 1997), (Pender 2003), and (Shoemaker 2004) can be of interest.

Object oriented design

Analysis and designing software objects can be a challenging task and there exist several design patterns to help the designer in this task. Several patterns are explained and used in this document as well. A pattern consists of a description of the problem and a description of the solution. Patterns are designed by experienced software developers and describes a general solution for a specific design problem. This document focus on design patterns, but there exists analysis patterns, implementation patterns and test patterns as well.

Refactoring is a methodology for making the internal structure of an existing software system better, without changing the external structure, and without introducing any new errors. Refactoring will not be handled by this document as this document focus on designing new software systems.

Object oriented programming (OOP)

When an use case has been analyzed and designed, it should be implemented in an object oriented based programming language. The important properties for such a programming language:

1. Encapsulation; hiding data inside objects and use methods or properties to manipulate the data, important for better and faster maintenance of the software.

2. Inheritance; an object inherit properties from another object by inheritance, define common codes only one place to avoid duplication of code, important for better maintenance of the software.

3. Polymorphism; different interfaces for the same type of method in an object, avoid duplication of the code, important for the maintenance of the software.

The goal of the OOP is to implement the object oriented design models. See Figure 1.5 for the relation between the software development process, use cases, and the object oriented (OO) steps in developing software described in this document.

As the software development process is an iterative process, the OOA, OOD, OOP and test steps will be performed on use cases defined based on the specification. The incremental part will redo the iterative process for each set of use cases again and again until a final version of the software process is developed.

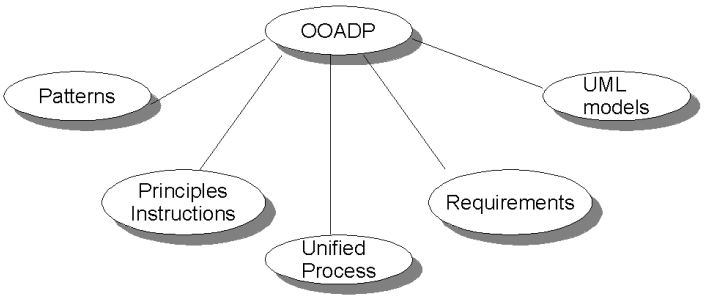


Figure 1.6: Topics and skills in the course.

1.3 Object-Oriented Analysis and Design (OOAD)

Object-Oriented Analysis and Design (OOAD) is the process of transforming any requirements into static and dynamic object models fulfilling the requirements. Software development today is based on objects, most programming languages are based on objects, and many software development processes are based on object-oriented development. OOAD can be done in many ways so the software development process is important as the process describes the phases, the sequence of phases and the set of documents to be produced in every phase. An advantage of using a software development process is the automatic outcome of documentation through the analysis and design phases.

1.3.1 Why learn OOAD?

Some of the reasons for learning OOAD is to:

1. learn and understand the technical skill in object oriented analysis and design to develop robust and maintainable software application (with a good design),
2. learn to think objects, need a maturation period, and be aware that this document will only give an introduction to this subject,
3. give an introduction to dedicate responsibility to objects, be familiar with the most used UML framework, and the usage of some common patterns,
4. make models of the software, both static and dynamic models,
5. documentation of the software, the text documents and UML diagrams will be the documentation,
6. Using more time in the analysis and design steps, and less time in the testing and maintenance steps.

Note that UML is just a set of standard diagrams. UML is not a method, only a tool in doing and documenting the OOAD phases. Some of UML diagrams are very general and can also be useful in system design and in general problem solving. In OOAD there exists many correct answers as the answer is the design of the application. However the answer will depend of the knowledge of the designer(s). There exists good answers, ok answers, less good answers and wrong answers to software design. Figure 1.6 shows the topics and skills that hopefully will extended the knowledge and experience of the reader of this document.

The challenge in every software project is to get to know the problem domain and understand what the customer really want. The time spent in the analyzing steps to try to understand the customer(s) and the problem domain, and in the design steps making a good design will pay off in the testing steps and the maintenance steps. Figure 1.7 shows the usage of time in the "standard" functional software development and the object-oriented software development. An object-oriented development will always use some more time in the analysis and design steps, especially design steps as the classes and interaction between the classes must be defined. However, the maintenance time using an object-oriented development will be much lower.

If the development time is more important than the maintenance time, use the "standard" functional development, if the lifetime of the software is important use the object-oriented development.

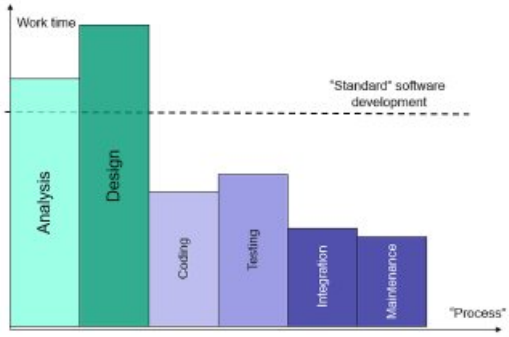


Figure 1.7: An indication of the time spent in the different steps in developing software using a "standard" functional development and the object-oriented development.

1.3.2 Requirement

The development of a software application will always start with a specification. However, the specification will normally be a description of the functionality of the application defined by the "user". A specification often lack a lot of information in order to develop a software application. A requirement analysis is necessary to collect enough information about the functionality to develop the software. The first phase will always be the requirement phase. The requirement phase is independent of the development methods and development tools. There exist different ways of collecting the requirements, one way is described in this book (Robertson & Robertson 2013). The development process will indicate how the requirements should be collected, documented and used. The requirement should also be the input to any test of the system. The next phase will be the analysis phase.

1.3.3 Analysis

The analysis phase will be based on the requirement phase and the software development process. The analysis phase will be a thorough study of the problem area, and the requirements of the system. The analysis phase should focus on understanding the problem area, not finding solutions. In an object-oriented approach this phase will emphasis on finding and describing the objects in the problem area. Analysis is the action of doing an investigation:of the problems/challenges and of the requirements.

The goal of the analysis phase is to understand the problem, why should the software should be developed. The problem domain will be an important part of the analysis phase. The development process should define the type of documentation that should be produced in this phase. The next phase will be the design phase.

1.3.4 Design

The design phase should be based on the analysis phase and the focus is to prepare solutions for the problems, the challenges and the requirements, and make a model of the design. The development process should define the type of documentation that should be produced in this phase as well. The emphasis in this phase, using an object-oriented approach, is on defining the software objects, and how these objects are collaborating to satisfy the requirements. This means that the requirement phase has to be performed prior to the design phase. The design phase is the action of making a conceptual solution, defining the objects, and assigning responsibilities to these objects. The goal of the design phase is to document how the software is working, both the static and dynamic behavior.

In an object-oriented approach the focus in this phase will be how the software can be structured in classes or objects.

Design of classes and object

A class is a collection of methods and encapsulated variables, responsible for a small specific part of the requirements. The methods are the action of the class, called from other classes or system events (like timers, GUI etc.). The variables are the data of the class, normally encapsulated (private). A class will cooperate with other classes for more complex (sub) functions of the software system. An object is a "instance" of a class in memory, the class is the abstract definition and the object is the working

"instance" of the abstract class. A class can result in one or several objects. The class is in your model, the objects will be in your running software (computer memory).

The most important task in OOAD is dedicating responsibility to objects, which object has the responsibility for a specific action. The software is a set of actions, from an input event to an output event or change of state, the design is to select the objects responsibility for these actions. In this document several fundamental principles are presented by dedicating the responsibility to the objects, known as pattern. Patterns is a tool to help the designer in making the type and number of classes, the communication between these classes, and assigning the responsibilities to these classes to perform any actions. The tricky part of software development is transforming the knowledge from the analysis phase into a working software design, and the patterns will be an important tool in this work. The goal of the design phase is to make a model of the software, both for design and documentation. This model should consists of both static and dynamic information. The software development process should define how to document the model. In this document UML diagrams will be used.

Design model

Time critical systems like Process Control Systems (PCS) will be the focus in this document and the design model should reflect both the static and dynamic behavior of such systems. UML will be used as the standard notation to describe various design solutions in diagrams. The most important UML diagrams for time critical systems are :

1. use case diagram. A use case diagram is used mainly in the analysis phase. A use case diagram can be a good starting point for nearly all kind of problems or challenges (projects), not only software challenges,
2. class diagram. A class diagram can be used in both the analysis and design phase, and describe the static information in the software system,
3. interaction diagrams. An interaction diagram is used in the design phase describing the dynamic information in the software system. There exists several interaction diagrams depending on the focus, this document will focus on the sequence diagram where the focus is time,
4. state diagram. A diagram that can be used in both analysis and design phase, focus is state depending systems like real-time systems and communication protocols to mention some.

Speciality versus generally

In the design phase the designer will make a solution that is special for the problem. However, the designer will aim at making a more general solution by making reusable code. Always try to reuse tested code. The programmer needs instructions when to program special code and when to program general code. One common solution to this challenge is to include a configuration section using general code together with the configuration part to perform a special solution. In these cases the configuration part has to be included in both the analysis phase and the design phase. There is a good advice always to include a configuration section in the software.

You will have the same question when developing other type of models like mathematical models. Mathematical models like mechanical models and data driven models, you want to develop models that are more general, not specific for one challenge even if you are using data from one specific case for calibrating the models.

1.4 Example

A portable temperature measurement will be used as an example. Assume one of these small portable devices that you may have at home, measuring the inside and outside temperature, with a display, running on a one or two batteries. Such a device is shown in Figure 1.8. If you are hired for a company to write the software, where will you start? What will be your first steps?

Normally you will jump into writing the code and focus will be on the main functions, without any thoughts about what can go wrong or any extensions. You will end up with a software that will work OK when everything is OK, but will fail when anything goes wrong. You will then start to add the code for the error/fail situations with the results that the structure of the code will be a mess, and very difficult to do any maintenance. This document will take you through the important analysis and design

Figure 1.8: A thermometer showing the inside and outside temperature. The device can also show the minimum and maximum temperatures, but this option is not implemented in this example.

phases before start writing the code. This may be boring, at least in the beginning, but by this way the structure of the code will be better, the used time will decrease, and the maintenance of the software will be much easier and faster.

Let us go through this example with some highlights in each of the phases. The starting point will that your company has bought 1000 of these devices, but the software is missing. So you must develop the software code.

1.4.1 Analysis:

Collect the requirements and understand the problem domain. Start with collecting the requirements. The most important requirements will be:

1. Display the internal and remote temperature,
2. Update the temperatures,
3. A set of buttons for showing the maximum and minimum temperatures, and reset these temperatures. This will be implemented in the second version.
4. A button to select between °C and °F. The first version will support only °C.
5. The device will run on battery. This means that the measurement should not be continuos, and as the temperature process is a "slow" process it will be OK to update the values every 10 seconds to save energy.

The next step will be to make an use case diagram. A use case diagram focus on the main functions of the software application, and the input/output information. An UML use case diagram is shown in Figure 1.9. Note that this is just one solution of the problem domain, there may exist several answers (or solutions) to the problem domain.

The use case diagram shows the main functions as oval circles with text, while the inputs and outputs to the software application are show as small peoples. The text inside oval circles, use cases, is the names of the main functions, what the software is going to do, and should start with a verb. In this case the software has three main functions; 1) Update temperature, 2) Make configuration, and 3) Handle Min/Max Temperatures. A domain model, to visualize the static information, should also be created. One solution for a domain model for this example is shown in Figure 1.10.

The domain model will show the "virtual" classes in the problem area (domain), often the nouns in the specifications and requirements. Note that the analysis phase should focus on the problem domain, what the software application is going to do, not how it is going to do it. At the end of the first analysis

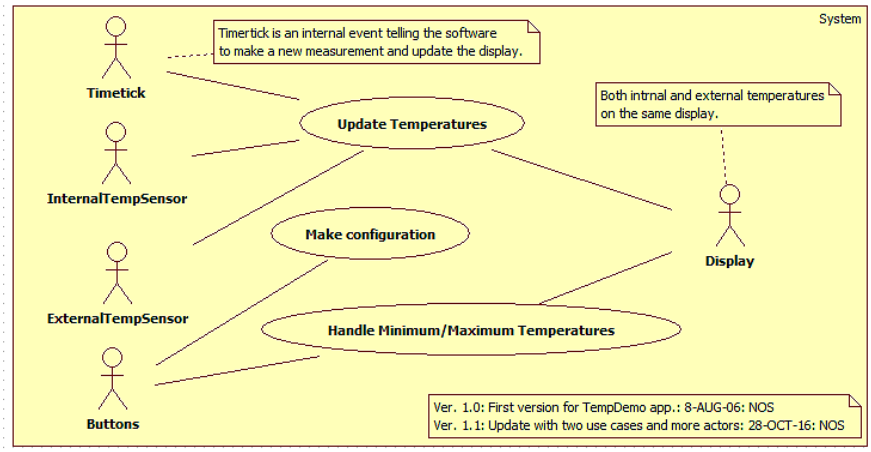


Figure 1.9: The Use Case model for the temperature measurement example.

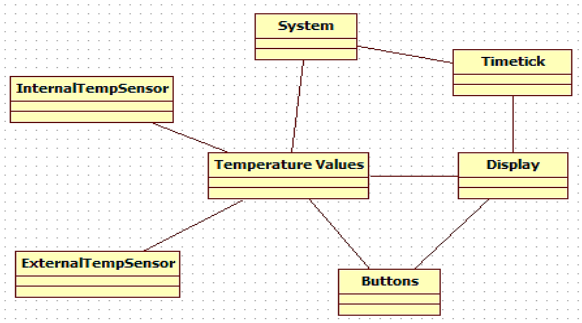


Figure 1.10: Domain model for the temperature measurement example.

phase we will select one or a few use cases for design, implementation, and testing. For each iteration one or a few use cases will be selected for analysis, design, implementation and testing. This is the clue of the UP, the software development process in use in this document. The goal of the analysis phase is to try to understand what problem or challenge the software should solve.

1.4.2 Design

The design phase should be based on the knowledge from the analysis phase, and design the software in such a way this it will fulfill the requirements. Using the UP, one should first select the most important use case (or a few number of use cases), design, implement and test this use case(s) before selecting new use case(s) for analysis, design, implementing and testing. In this example the following design steps will be performed:

1. Select the "Update Temperatures" as the first (and most important) use case. Start with an interaction diagram and select a object to be responsible for the operation.

2. Use the information from the analysis phase, and design patterns, to make an interaction diagram. An interaction diagram shows the objects and how the objects are interconnected, the dynamic information in the design. Design patterns will be a good helping tool in defining the objects. There exists several UML interaction diagrams, we will choose the sequence diagram because these diagrams have focus on time. Time is often important in software design for Process Control Systems (PCS), communication protocols and real-time systems. Figure 1.11 shows one solution of the software for the selected use case. The design has four objects and a set of methods for communication between the objects. Note that names of the methods and the attributes are based on the "CamelCase" coding standard and where the first character define the type (f=floating point and r= object reference). Making an interaction diagram is the most tricky part in this software development process, however this document will show how several design patterns can be used and combined as a tool in this work.

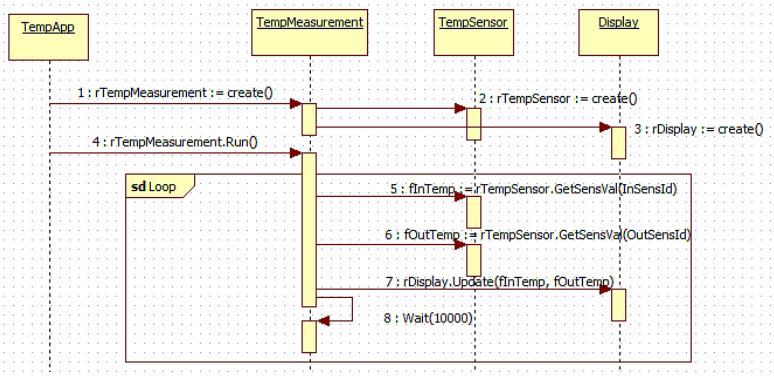


Figure 1.11: A sequence diagram showing the usage of the methods based on the time (vertical axis). Note that the f and r characters in front of the attribute and methods are due to a coding standard (f=foating point number, r=object reference).

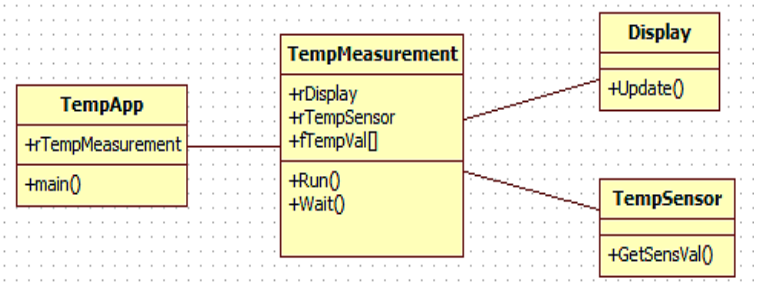


Figure 1.12: The class diagram of the tempearture measurement application.

3. The last step in the design phase is to make a class and object diagram. These diagrams show the static information in the design. The class diagram document the software model, while the object diagram document the class structure in memory when running the application. Figure 1.12 shows the class diagram for this design. The object diagram will not be included in this example, however the object diagram will be the same diagram, with the same classes except that there will be two TempSensor classes as the running software will have two temperature sensor devices.

1.4.3 Programming

Based on the analysis and design phases, the first use case can be implemented. The C# coding of the application is listed in Appendix E.1. The application is developed as a console application just writing the outputs into a console window instead of Windows (GUI) application. The temperature sensor values are based on a random value generator build in C#. The also shows the strength of object oriented development. The TempSensor class is responsible for the sensor values and with no measurement system a value is simulated for testing. When extended for the right hardware platform the only class to be changed is the TempSensor class. The output from this temperature measurement application is shown in Figure 1.13.

This figure shows the loop counter, the time, and the inside and outside temperatures. The output from the application should be an endless loop, but this application is stopped after 16 loops. Without the correct hardware for testing, will the temperature be simulated by adding a random number in the range $[-0.5\,°\mathrm{C}, 0.5\,°\mathrm{C}]$ to the previous temperature.

A Windows application can also be used for testing, one example of such a graphical user interface (GUI) is shown in Figure 1.14. However, the GUI is not implemented in this example. This can be an exercise for the reader.

Note that the user interface (UI) is not in focus here, the display class will be responsible for the correct outputs. Windows UI is used for testing purpose in this stage of the project, however support for the correct display must be added to the display class later on in the project. This is one of the advantages of object oriented development, assigning responsibilities to a class. Changing the output should in this case only affect the display class, so testing on the development platform is a good solution

```
Start of main program
Init the default temperatures ..
Start measuring the temperature ..
 [01] 11.04.58 Temperaures: Inside=22,8 Outside=17,1
 [02] 11.05.08 Temperaures: Inside=22,9 Outside=17,3
 [03] 11.05.18 Temperaures: Inside=22,6 Outside=17,5
 [04] 11.05.28 Temperaures: Inside=22,7 Outside=17,4
 [05] 11.05.38 Temperaures: Inside=23,1 Outside=17,3
 [06] 11.05.48 Temperaures: Inside=23,0 Outside=17,7
 [07] 11.05.58 Temperaures: Inside=23,5 Outside=17,3
 [08] 11.06.08 Temperaures: Inside=23,3 Outside=17,5
 [09] 11.06.18 Temperaures: Inside=23,1 Outside=17,0
 [10] 11.06.28 Temperaures: Inside=23,0 Outside=16,7
 [11] 11.06.38 Temperaures: Inside=23,2 Outside=16,6
 [12] 11.06.48 Temperaures: Inside=23,1 Outside=17,1
 [13] 11.06.58 Temperaures: Inside=23,6 Outside=17,2
 [14] 11.07.08 Temperaures: Inside=23,1 Outside=17,4
 [15] 11.07.18 Temperaures: Inside=23,5 Outside=17,2
 [16] 11.07.28 Temperaures: Inside=24,0 Outside=17,3
End measuring the temperature !
Use any key to quit the application ..
```

Figure 1.13: The console window output from the temperature application, with the APP_DEBUG parameter active.

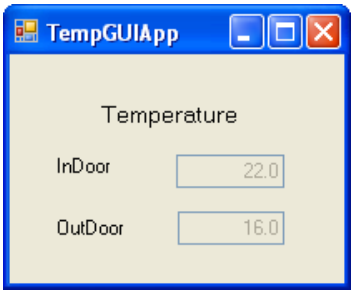


Figure 1.14: The temperature application with a graphical user interface.

Figure 1.15: A set of different elderly mobile phones.

before uploading to the "correct" hardware for final testing.

1.5 Case studies

The focus throughout this document is development of software for time dependent systems like Process Control Systems (PCS), communication protocols and real-time system to mention some. As shown in the example it is important for the reader to have some knowledge about the problem domain to perform the analysis step, so the case studies used in this document will be something that the reader should be familiar with.

1.5.1 Mobile Phone

It is important to use a device that the reader are familiar with when learning the analysis, design, and implementation of software, to better understand the problem domain. Mobile phone is a standard tool today, but the reader should assume that the software is removed and must develop the software (only the upload section is working, for uploading a new version to the phone).

See Figure 1.15 for some types of elder mobile phones. What will be the main functions of a mobile phone? Here is a list of some functions that should be available:

1. Make a SMS,
2. Receive a SMS,
3. Make a call,
4. Receive a call,
5. Make a MMS,
6. Receive a MMS,
7. List of phone numbers and corresponding names,
8. Search on internet,
9. Play games,
10. Setup.

How to start developing the software for the mobile phone?
The mobile phone will used as an example throughout this document.

Figure 1.16: The inner working of a washing machine (howitworksdaily.com:Apr-16).

1.5.2 Washing machine

A simple process system used to rinse, wash and dry clothing. The system must be able to pump water in and out of the machine, must be able to detect when the system is full of water, heat the water to wash the clothing at different temperatures, include detergent, and dry the clothing at the end. The process consists of several sensors and actuators that must be controlled in a specific sequence depending on the operation selected. Figure 1.16 shows the inner workings of a washing machine with front loading of cloths.

What will be the main functions of a washing machine? The main functions should be:

1. Rinse the clothing,
2. Wash the clothing at 40 °C or 60 °C,
3. Dry the clothing.

How to start developing the software for the washing machine?

1.5.3 I/O Measurement system

The measurement system is an input/output (I/O) system handling analog and digital signals. The I/O system is an external device connected to an USB or network port, including some special software driver like a library or a service, and some sort of application programming interface (API). One example is the USB devices from National Instruments using the USB port for hardware interface and the DaqMx library as the software interface (API). Such a I/O measurement system consists of the following parts:

- input channels where the sensors are connected,
- output channels where the actuators are connected,
- the measurement system device (the measurement system device is powered by the USB port),
- the computer, with the measurement software. The main function of this software should 1) filter and display the sensor values, 2) handling the actors, 3) include a configuration part for each channel for the measurement system, and 4) the possibility of logging the any values.

How to start developing the software for the measurement system?

1.6 Software simulators

Simulators contains mathematical models, and running these models by discretization, either in near real-time or in a chosen time scale. Simulators can very easy be implemented in C#. Simulators can

be great tools for testing models, or software systems controlling a process that can be simulated by a model. Examples of simulators can be the washing machine or an oil/water separator as long as we have a mathematical model.

With Industry 4.0 and Internet of Things (IoT) digital twins are also becoming popular. A digital twin is software simulator using recorded data to simulate a process or a device in parallel with the real process or device to optimize the operation or make prediction.

1.7 Inception phase

The startup of the software development process is named inception. Inception is the first iteration where the business is making the decision to start the project, define a vision, business case and/or project main goal, and list the most important use cases. Should be very short, only a couple of days, maximum a week. The vision or business case must be made by the management group including the decision whether to do this project or not. The management group will also select a project leader who will make a project group starting with the main use cases of the project.

This document will be based on the same development process as for the software, containing the same four phases. The first phase is inception , the phase where the decision made by the management group of the university college was include an OOAD course in this master programme, and let the students learn the methodology on object oriented analysis and design of software. The second phase is the elaboration phase where more details about the project is captured. The elaboration phase in this document is this chapter, the Introduction, and the next chapter about software development processes. The third phase is the construction where most of the iterations with collecting requirements, analysis, design, implementation and testing is performed. This document contains three construction iterations, the first one is mainly about programming; using Visual Studio Integrated Development Environment (IDE), the C# programming language, and version control systems. The second construction iteration is mainly about basic OOAD, and the third construction iteration is extending the OOAD knowledge from the second construction iteration. The last phase is transition where a mature version of the software, that satisfies all requirements, is released. The transition phase is not covered by this document.

Chapter 2

Software development process

2.1 Introduction

There must always be a process to describe how to develop the software, the OOAD is just a tool for developing the software, not defining the process. The process will describe the workflow, the orders and sequences on when to 1) do the different steps, 2) make the documentation, and 3) create or use the different OOAD diagrams. Time and resources will be important factors in this process.

Definition 1 *A software development process (a system life cycle model) is a description of the activities carried out in a software project, and the relative order or sequence of these activities.*

Unified software development process , or short Unified Process (UP), is one process describing how to develop the software, and is used as an example of a development process in this document. OOAD is necessary for almost every software development process, and is not used only for UP.

A software development process is not project planning. Project planning is the art of scheduling the necessary activities, in time, space and staff in order to optimize 1) project risk (should be as low as possible), 2) profit (should be as high as possible), 3) customer satisfaction (should be as high as possible), 4) worker satisfaction (should be as high as possible), and 5) long-term company goals. The project planning will be based on the software development process as the project planning will schedule the workflow, based on a specific software development process. A project plan contains much information, but must at least describe 1) resources needed (people, money, equipment, etc.), 2) dependency & timing of work (flow graph, work packages), and 3) rate of delivery (reports, code, etc.).

There are many different software development processes (life cycle models) to choose from, a few of these are 1) waterfall (originally described by Winston W. Royce in 1970 (Royce 1970)), 2) code-and-fix, 3) spiral, 4) rapid prototypes, and 5) agile methods, like Scrum, Kanban, extreme programming (XP) and unified process (UP). Many of these software processes are minor variations on a smaller number of basic processes. None of these development processes are perfect. However, by minor changes of the life cycle model, the improvements and/or trade-offs like 1) development speed (time to market), 2) product quality, 3) project visibility, 4) administrative overhead, 5) risk exposure, and 6) customer relations can be better adjusted to the goals of the projects.

2.2 Development processes overview (Lifestyle models)

Developing software consists of performing several tasks for producing a working software application and a set of documents. These tasks have to be performed in some type of sequence and the outcome of the sequences will be either software or document(s) or both. There exists many different ways of combining these tasks and documents, and type of documents. Since there are different ways of combining and/or mixing these tasks there exist many software process. No perfect software process exists, only a set of processes with a number of advantages and disadvantages. These software development processes, also named software development process models (Scacchi 2001), where the purpose of the process model is defining how to develop and maintain the software during the lifetime of the system. A software process will describe how to combine the analysis, design, implementation and test sections in making and/or maintain a software application, with documentation (Martin 1993), (Leffingwell & Widrig 2000), (Newkirk & Vorontsov 2004), (Viega & McGraw 2002), (Jøsang 2021), (Robertson & Robertson 2013). This section will describe a few of the existing software process models.

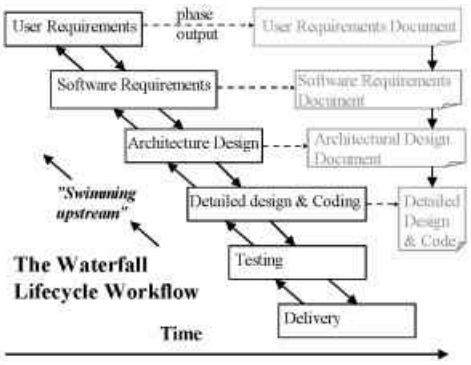


Figure 2.1: The waterfall process.

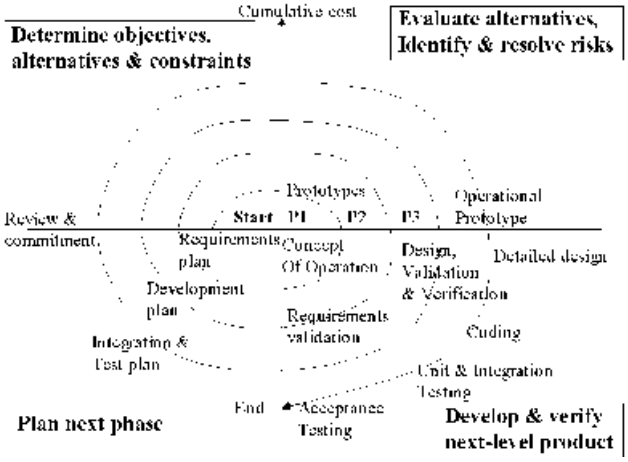


Figure 2.2: The spiral process.

2.2.1 Waterfall

The waterfall model is a sequential software development model in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance. The origin of the term "waterfall" is often cited to be an article published in 1970 by Winston W. Royce (1929–1995) (Royce 1970), although Royce did not use the term "waterfall" in this article. Ironically, Royce was presenting this model as an example of a flawed, non-working model (Mueller 2009). Still many developers used this model for the lack of something better. The waterfall process is shown in Figure 2.1.

2.2.2 Spiral

The spiral model is a software development process combining elements of both design and prototyping-in-stages, in an effort to combine advantages of top-down and bottom-up concepts. This model of development combines the features of the prototype model and the waterfall model. The spiral model is intended for large, expensive and complicated projects (Scacchi 2001). The spiral process is shown in Figure 2.2.

2.2.3 Rapid prototypes

Rapid software prototyping describes a practice in the software development process or application development process and sometimes refers to RAD (rapid application development). Software prototyping, a possible activity during software development, is the creation of prototypes, i.e., incomplete versions of the software program being developed (Pan, Lin, Sui & Tseng 2019).

A prototype typically implements only a small subset of the features of the software application, and the implementation may be completely different from final version of software application (Whitten & Bentley 1998). The rapid prototyping process is shown in Figure 2.3.

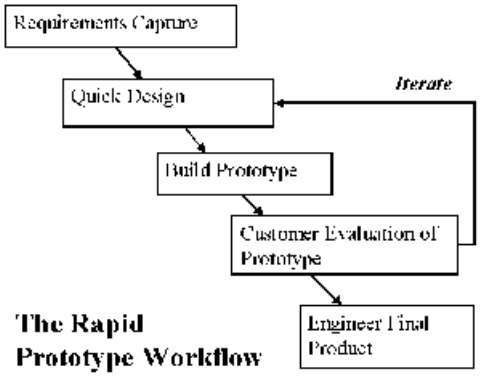


Figure 2.3: The rapid prototype process.

A challenge using the rapid prototypes is that the customers thinks the software is almost finished because of the prototyping. It is important to focus on the missing functionality in the software when discussing with the costumers.

2.2.4 Agile Development processes

Agile Software Development is a conceptual framework for software development that promotes development iterations, open collaboration, and adaptability throughout the life-cycle of the project (Bittner & Spence 2006).

There are many agile development methods; most minimize risk by developing software in short amounts of time. Software developed during one unit of time is referred to as an iteration, which typically lasts from two to four weeks. Each iteration passes through a full software development cycle, including planning, requirements analysis, design, writing unit tests, coding until the unit tests pass and a working product can be demonstrated to stakeholders. Documentation is no different than software design and coding. An iteration may not add enough functionality to warrant releasing the product to market, but the goal is to have a new version of the software at the end of each iteration. At the end of each iteration, stakeholders re-evaluate project priorities with a view to optimizing their return on investment.

Agile methods emphasize working software as the primary measure of progress. Combined with the preference for face-to-face communication, agile methods usually produce less written documentation than other methods. In an agile project, documentation, Gantt charts and other project artifacts all rank equally with working product. However, when stakeholders are asked to prioritize deliverables for demonstration at the end of the current iteration, they generally prefer to see working product. Stakeholders are encouraged to prioritize iteration outcomes based exclusively on business value perceived at the beginning of the iteration. If documentation represents higher business value than working software in any particular iteration then stakeholders give it a higher priority than working software. The (cross-functional) development team will accordingly produce that documentation instead of lower priority software (Bittner & Spence 2006).

Agile teams are often located on the same location or working together online facilitate face to face communication, which is preferred over written documents. However, this type of communication can be a challenge in "lock-down" periods like the covid-19 period. One agile management methodology, Scrum, prefers team sizes of 5 to 9 persons. Larger teams than 9 persons should be split into smaller teams to help make team communication and team collaboration easier (Schwaber & Sutherland 2011). Scrum consists of inspections and sprints. Inspections means checking of the work and the goal is to detect undesirable variances. The heart of Scrum is a Sprint, a time box of a month or less (Schwaber & Sutherland 2011), an iterative process of solving a function in the system. Sprint consists of a planning meeting (short, daily of maximum 15 minutes), a development team, and a review of the solution. Scrum consists of a product owner, the development team, and a Scrum master. The Scrum teams are self organizing in the meaning that they always try to choose the best way to accomplish the work.

Extreme programming (XP) (or pair programming) is another agile software development methodology where two programmers are cooperating on the same computer, often a combination of an experienced developer and a more "fresh" programmer. XP is intended to improve software quality and

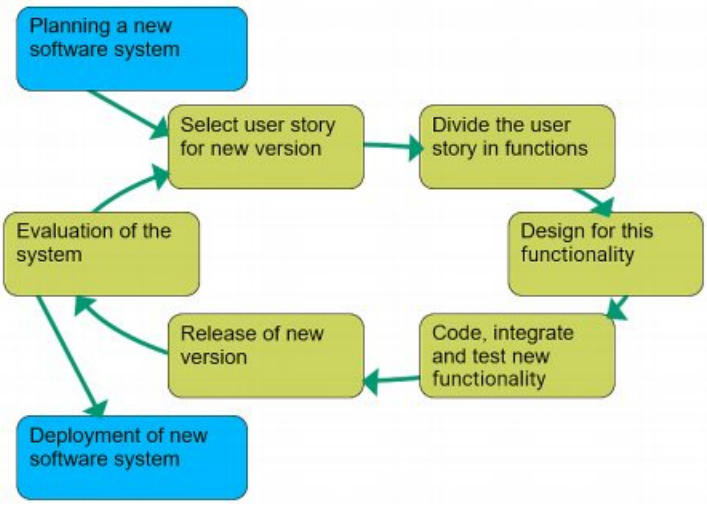


Figure 2.4: The agile software development process, based on iterations (Jøsang 2021)

responsiveness to changing customer requirements by frequent "releases" in short development cycles, intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.

Mob programming is sort of an extension of XP where the whole team works on the same thing, at the same time, in the same space, and at the same computer. In mob programming, the collaboration is extended to everyone on the team, normally between 5 and 9 persons. The team will still use only a single computer for developing or maintaining the code, and updating the code base.

Another agile management methodology, Kanban, introduced by Toyota, used as a scheduling system for Lean and just-in-time production (Skeie 2014). The success of the Kanban method was noticed by the software development industry. In software development Kanban splits the major problem into many small pieces of problems. These small problems are defined by teams, and put up on the Kanban board for visualization and track what others are working on and see potential bottlenecks during development. In short, Kanban systems focus on 1) continuous work flow, 2) no fixed iterations nor sprints, 3) work is delivered when done, 4) teams only work on few tasks at a specific time, and 5) make constant flow of released features throughout the development.

The agile software development is based on iterations where each iteration consists of tasks like selecting a use story or use case, and analyzing, designing, implementing and testing the specific user story or use case. For each iteration a new version of the software will be available for the customer for evaluation and/or testing. Figure 2.4 shows an overview of the agile software development with the iteration tasks with light green background.

Agile means being able to quickly change direction since early feedback is possible at the end of each iteration, by deployment a new version of the software to the costumer. In software development, it requires strong discipline to code for agility including proper testing at the end of each iteration. Lean is also a way of thinking in industrial process, simplicity (do not waste time on unnecessary tasks). Kanban is such a method transformed to the software industry.

2.2.5 Summary

A software development process is a tool, based on a set of rules, to help you in managing the steps for developing (software) system. The different type of development processes can be compared by the number of rules, and the more rules to follow, the less experience is necessary to use a development process. Experienced developers often combine sections from different development processes to have a processes adapted to the experience, project and knowledge. Agile processes such as UP (RUP), eXtreme Programming (XP), Scrum and Kanban consists of different number of rules, from a prescriptive with many rules to adaptive of less rules. An overview of these processes is shown in Figure 2.5. The numbers indicate the number of roles, activities and artifacts (Kniberg 2009).

Figure 2.5 shows that UP (also known as RUP) is the most prescriptive and Kanban is the most adaptive development process of these agile processes. UP was invented by Rational, the name of the

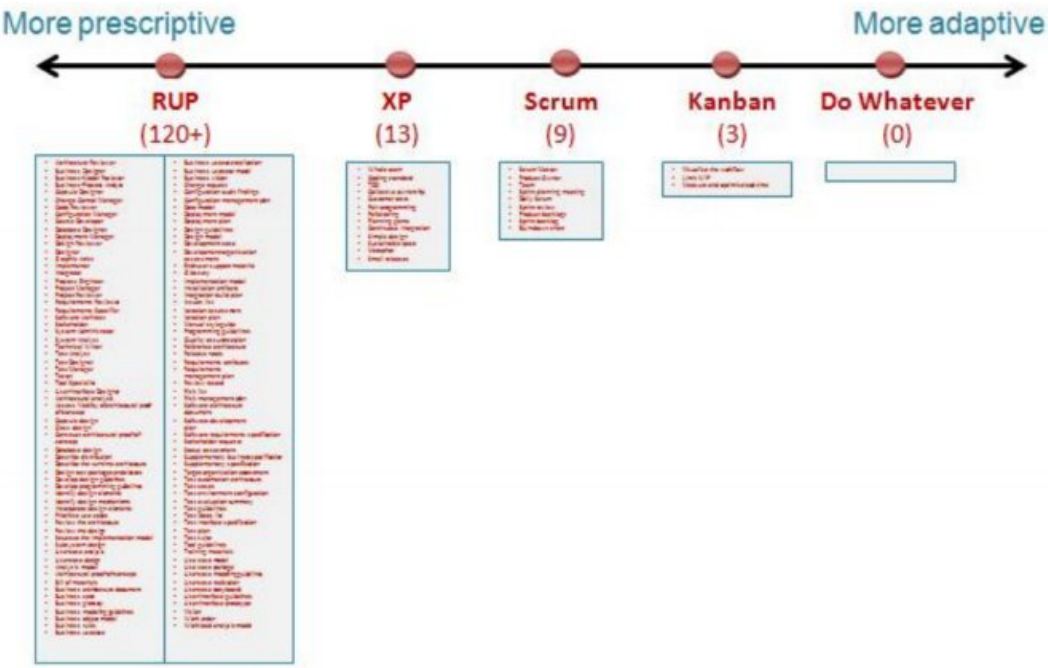


Figure 2.5: Comparing several development processes with focus on the number of rules, scaled in the range between prescriptive or adaptive processes (Kniberg 2009).

firm for the three amigos defining UML (and UP), so UP is also called Rational UP (RUP) previously as in figure 2.5.

UP will be used as the development process for software development in this document, and the document itself is also based on a couple of iterations. The main reason is that this process will help unexperienced programmers in better managing the steps in a development process and it will later be easier to adapt to other processes when getting more experience.

2.3 Unified Process (UP)

2.3.1 Introduction

The Unified Software Development Process or Unified Process (UP) is an iterative and incremental software development process framework. An overview the UP is shown in Figure 2.6 showing a set of iterations collecting the requirements, designing, implementing and testing the software.

The UP rely on an iterative life cycle, a risk driven development and is use case based. This can be explained as:

1. Iterative life cycle; repeating the same action again and again, the development is split in small projects with a locked time interval (Often two to six weeks), in every iteration new software functionality will be tested and integrated with the existing system, every iteration consists of requirements, analysis, design, implementation and testing, at every iteration the functionality of the system will increase, and is named incremental development.

2. risk driven development; easier to discover that the functions are not satisfying the requirements when the system is building in steps, shorter time between getting the requirements and implementation of the functions (functionality), faster feedback from the users, making what the users wants (Notes: Be aware : 1) not making too many changes, 2) the requirements, 3) that the users may not know what they want), faster testing on the right architecture, starts with the area of highest risk (responsibility of the project leader), the developers and programmers avoid long period of analysis.

3. use case based; base the iterations on the use cases (user stories, functionality wanted by the users), solve one or a few use cases in each iteration.

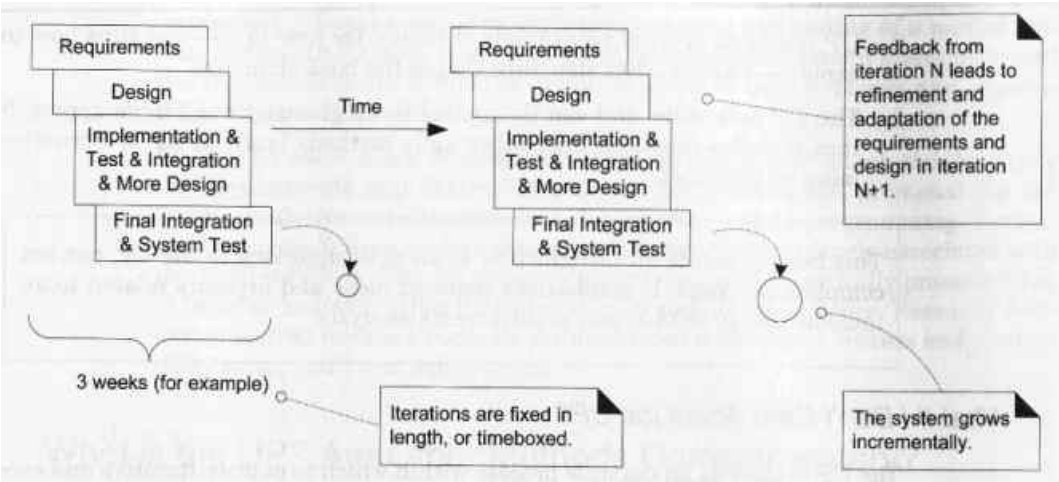


Figure 2.6: Iterate overview in UP (Larman 2005).

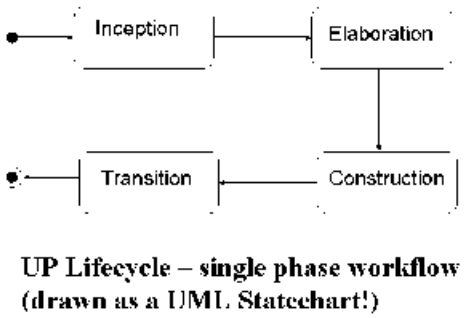


Figure 2.7: The main phases of an UP process.

The important concepts and best practice by using UP are:

1. take care of the most important, and often the most risky areas, in the first iterations,
2. engage the users for evaluation, feedbacks, and requirements,
3. making the "core architecture" in the first iterations,
4. continuous check of quality with a lot of testing (in each iteration),
5. based on use cases (functionality wanted by the users/customers),
6. making models of the software using documentation like UML,
7. administration of the requirements - include a "Change request" document for tracking and documentation of the requirements and changes of these requirements.

2.3.2 UP main phases

The main phases of UP is shown in Figure 2.7, drawing as an UML state chart. Note that for the UP software development process the iterations are more important than these phases. Each phase will consists of one or several iterations depending on the size of the project.

The main contents of these phases are:

1. Inception (initialization): this is the startup phase of the project, decision about running the project or not, project management and business modeling. Estimate and schedule project from derived knowledge based on specification and first part of collecting requirements. Short period of time, normally only one iteration. This phase is normally handled by the management of the company.

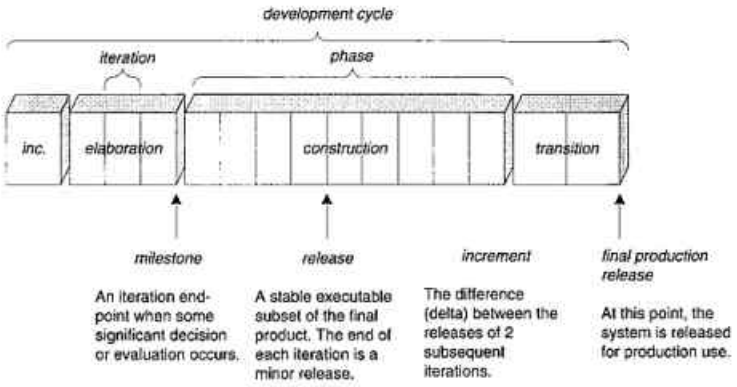


Figure 2.8: The UP development cycle (Larman 2005).

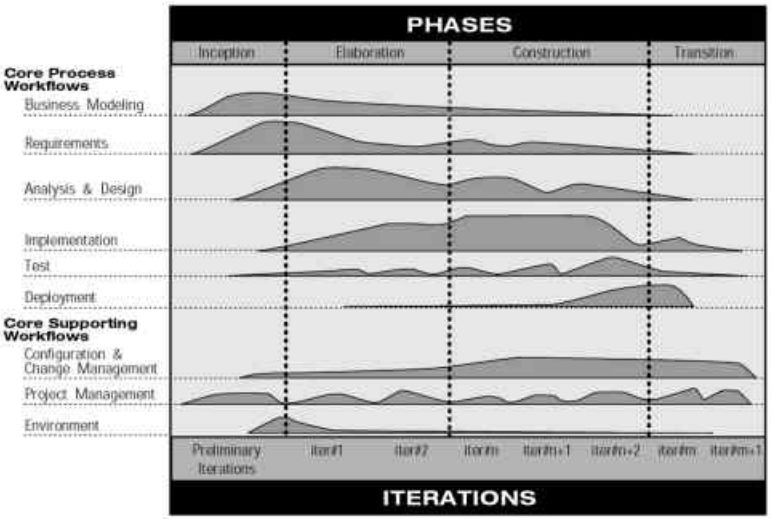


Figure 2.9: An overview of the UP disciplines, phases and iterations (Rational 1998), (Kruchten 2000), (Larman 2005).

2. Elaboration (development): startup of the development in the project, identify use cases, capture requirements and decide on the architecture. Decide on whether to develop modules and systems versus buy them. Decide on development platform, modules and system architecture, few iterations.

3. Construction: main phase of the project, select one or several use cases, collect more specific requirements for these use cases, analysis, design, implement and test these use cases, may also include bought software components that must be interfaced and integrated into the system, includes milestones for release/test versions of the software for the costumer, most of the iterations in the project should be in this phase.

4. Transition: the last phase for releasing a mature version of the software (or system) that satisfies acceptance criteria, including documentations. The software (or system) should be ready for production.

Some of these phases are divided into one or several iterations as shown in Figure 2.8. The time length of these iterations should be constant in a specific project and must be defined at the start of the project.

The relationship between the analysis (business modeling), design, implementation, test, and other disciplines in the software development process is shown in Figure 2.9.

Using milestones for when a new release is installed at the user, for instance every 10 iteration must also be included. Every iteration shall give a new running version of software with more functionality and/or better performance. Each iterations has a fixed time period meaning that the customer or user can plan when a new (test) release will be available for testing and can plan the testing of these releases.

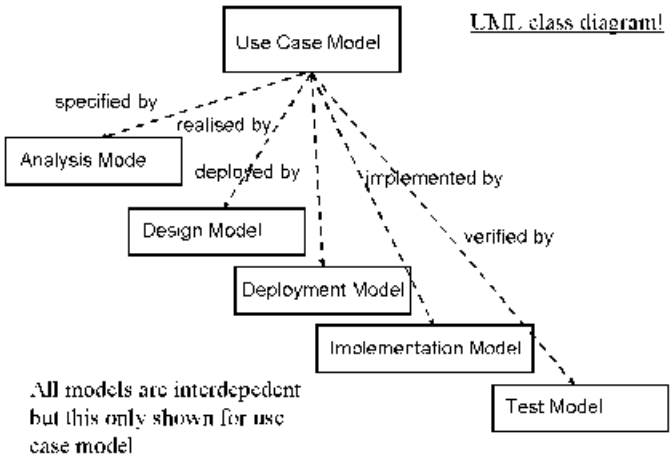


Figure 2.10: Show the UP for the use case model in UML.

Figure 2.10 shows how UP is used for making the analysis, design, deployment, implementation and test model from the use case model. The focus is the use cases, select one or a few use cases for each iteration, collect additional requirements for these use cases, do the analysis, design, implementing and testing. The iteration is finished when the use cases are implemented and tested in the software and continue iterate on the next set of use cases. Each of the requirements, analysis and design states will end up in a set of documentation that will be used for implementation and testing, and for the final documentation of the system.

2.3.3 UP summary

UP is a development process describing the steps and actions when developing software. The UP process summary:

1. use case driven,
2. Do $5-20\%$ of the most important use cases in every iteration, depending on the size of the project. In smaller project only one use case in each iteration. Approximately 5 to 20 iterations to finish the software.
3. short time between analysis and design to coding and testing,
4. Fixed length on each iteration,
5. Iterative lifestyle giving an incremental development,
6. Risk driven, focus on use cases with “challenges”,
7. Testing

2.4 Development process summary

Waterfall traditional development method where the process is sequential and not iterative, with well-defined phases consecutively. Requirements specification, analysis, design, implementation, verification and maintenance. Can take a long time from start to first delivery which limits the possibility of feedback in the process.

Agile a set of principles for software development where the focus is on cross-functional teams that have an evolutionary approach to development. Change of plans, early delivery and continuous improvement, where the focus is on a fast and flexible response to changes.

Scrum development methodology that seeks to realize the agile software development. Uses incremental development sessions (sprints) in a few days to a few weeks, where a variety of functions are implemented. Function list resulting from feedback from previous deliveries.

Kanban flow based methodology for managing the development, with focuses on reducing the number of tasks in parallel and elimination of bottlenecks. Kanban is a technique that visualizes the processes so that decisions about what should be implemented when and how much, is supported.

Devops a philosophy of ensuring that developers and operators of IT systems will work together on the development and operation to reduce the number changes/iterations in development and provide easier operation of the systems.

2.5 UP in this document

The UP in this document consists of four iterations, this elaboration iteration, and three construction iterations. The inception phase is the decision of making this document and the type of information to be included. The last iteration will be the transition, delivering the final results to the customer. The Elaboration phase will introduce the reading to the C# programming language and the usage of classes and objects. The construction iterations will introduce the reader to several documents and tools during the analysis, design, implementation and testing steps. The phases and iterations handled by this document will be (However note that most of the Testing column is outside the scope of this document. This is mainly testing of the students following the course using this compendium):

UP	**Analysis**	**Design**	**Coding**	**Testing**
Elaboration Iteration #1	Introduction UP Tier Architecture	Introduction UP Version Cntrl	Version Control Visual Studio C# / UP	Assignment #1 *(OOP: C#)*
Construction Iteration #1	Requirements Use cases Domain model System Seq. diagram	Patterns (GRASP) Interaction diagram Class diagram Object diagram	Exceptions Threading	Testing Assignment #2 *(OOA)* Assignment #3 *(OOAD)*
Construction Iteration #2	Use Cases Domain model Inheritance	Patterns Activity diagram State diagram	Data Structures Data Algorithms Inheritance Software Development	Testing Assignment #4 *(OOADP: C#)*
Transition Iteration #1				Final test

Chapter 3

Version Control System

3.1 Introduction

Version control is a system that records changes to a file or set of files over time so that it is possible to recall specific versions later. Using a Version Control System (VCS) or a Source Control System (CCS) it will be possible to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. There are several good reasons for using a version control system and they are available for most OS, easy to use and many are free.

3.2 Motivation

Some of the motivations for using a version control system (see Figure 3.1) are:

- safety; a backup of your work with history,
- history; an automatic history solution without making .bak, .old and .oldold versions of your files (or folders),
- collaboration; how to share the source code in a team like:
  - at least two programmers for developing and maintaining the source,
  - at least one programmer for developing and maintaining the source on different OS platforms,
  - at least one programmer for developing and maintaining the source for a long period (several years).

Whenever we have a software challenge, we should try to find the "best practice" to solve this challenge. The "best practice" is often described as patterns in software engineering. The "best practice" for software development is:

- Build the application early, and often (Build Early and Often),
- Use a private source-tree for building the application (Work in private sandboxes),
- Use always updated source when building (Stay in sync),
- Use merging of source code (Prefer merging over locking).

Software developers often like to organize the source code in their own way and not using a complicated system like version control. However, a version control system is easier to use and more safe then most of the "manually structures" made by humans, and provides greater capabilities and features. Some of the disadvantage of sharing the source files manually:

- No control of when or who changed any of the files,
- Difficult to maintain a local building system,
- Difficult to track the changes.

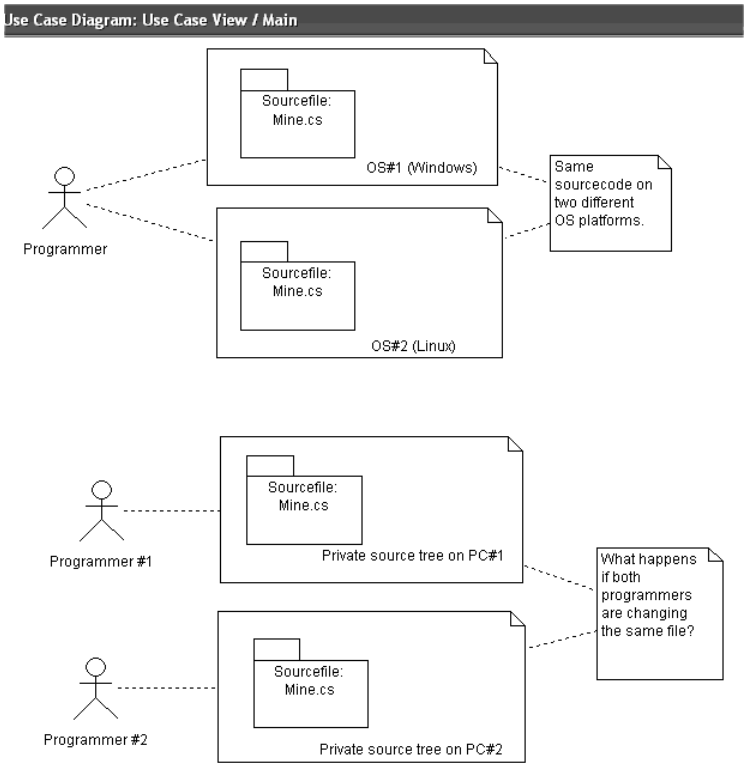


Figure 3.1: Source control of files when the same file is changed at different locations.

3.3 Structure of a version control system

A source control system will require some sort of structure for saving information and a project database is needed, see Figure 3.2. This figure shows the structure of a source control system with the following features:

- Version control (several programmers and several OS platforms),
- Checkout: Get the last version of the source code from the project database,
- Checkin: Save the changes in the project database,
- Locking and/or merging,
- Repository.

3.4 Overview of some version control systems

Several source control systems exist. Some of these systems are:

- SCCS (1972); Source Code Control System, small projects, using "locking", using a central repository,
- RCS (early 1980); Revision Control System, extensions of SCCS, using "locking", using a central repository,
- CVS (1989); Concurrent Version System, using "merging", each client has a private source tree, based on RCS, open source, using a central repository. Available on many platforms like Unix, Linux, Windows, and Solaris. Graphical tools for Windows like the WinCVS applications (www.wincvs.org) or the TortoiseCVS (www.tortoisecvs.org) module integrated into the Windows Explorer. Figure 3.3 shows the integration of the CVS commands in the Windows Explorer menu. Figure 3.4 shows

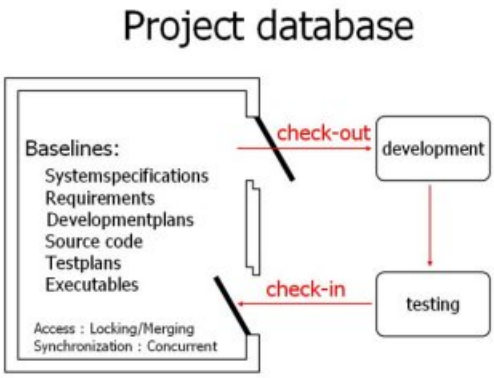


Figure 3.2: A source code project database.

Figure 3.3: The TortoiseCVS module integrated with the Windows Explorer for CVS operations directly on files or directories (www.tortoiseCVS.org; JUN-09).

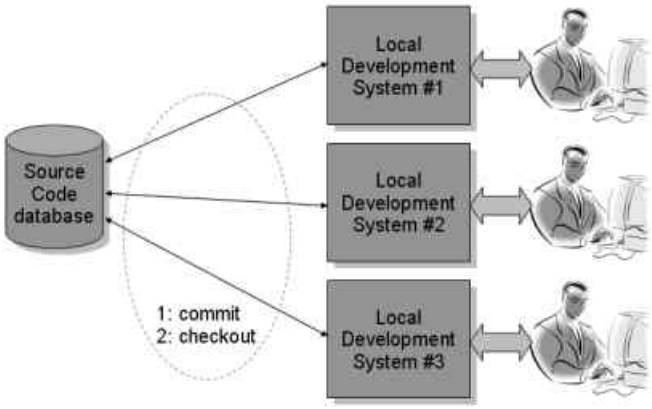


Figure 3.4: Overview of the CVS system.

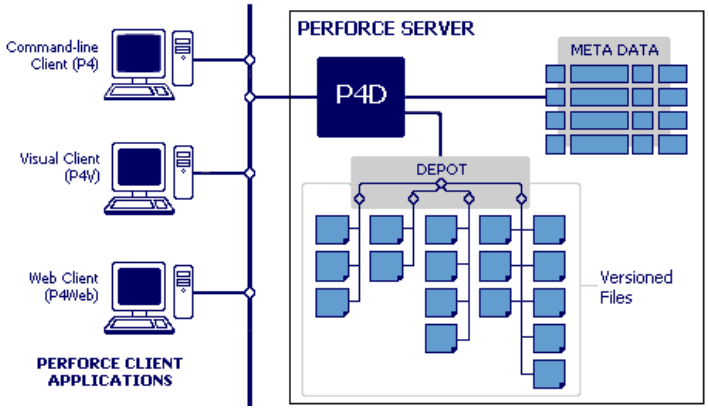


Figure 3.5: The Perforce system with the repository (server with versioned files) and any clients (www.perforce.com; JUN-09).

an overview of a CVS system with 3 developers working on 3 private source trees. The command commit is used for checking the changes into the source code database and the command checkout for getting the last changes from the source code database.

- Visual Sourcesafe (1995), Windows and Visual Studio, Using mostly "locking", but last version is also using "merging", commercial, proprietary revision control system, open source, using a central repository. Available only on Windows.
- Perforce (1985); Fast Software Configuration Management (SCM) System. Perforce SCM versions and manages source code and digital assets for large and small enterprises, based on RCS, offers a hybrid of merge and lock concurrency models, commercial, proprietary revision control system from Perforce Software (www.perforce.com). See Figure 3.5 for an overview of the Perforce system.
- SVN (2000); Subversion, developed to extend the functions of CVS (moving of files and version on folders), developed from "scratch", but based on the experiences by using CVS, private source tree,open source, open source, using a central repository. Available on many platforms like Unix, Linux, Windows, and Solaris.
- Git (1999); , the most important drawback of previous Version Control Systems are the repository, which must be stored on a central computer. Git is a distributed version control system having both local and central repositories. The user will always use the local repository when checking in and out any changes, and can connect to the central repository whenever this repository is available. This way you can work offline and synchronize whenever you are online.

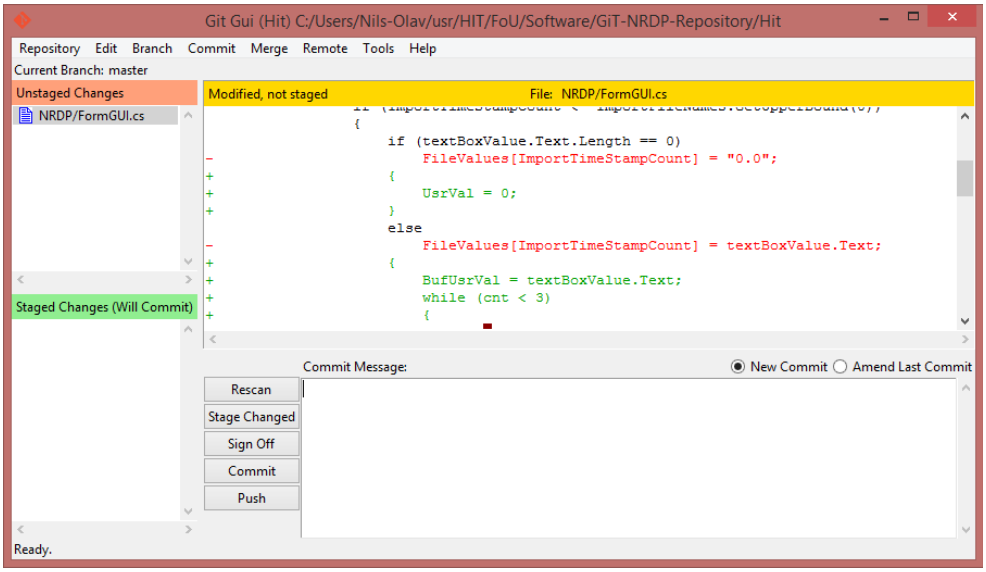


Figure 3.6: GUI based tools for staging, commiting and pushing for the Git version control.

3.5 Git

Git is a distributed version control system with the options of both a local and/or remote repository. There exist different commands for local and remote synchronization (**local**; *Stage changed* and *commit* / **remote**; *push* or *pull*). The manual (Chacon 2009) describe the functions of the Git system in details. Read the "Get Starting" and the "Git Basics" in the manual, if not familiar with the Git system, to get an overview of this system. The system is based on open source and used by many software companies. Available on several platforms like Unix, Linux, Windows, and Solaris. Git has the options for input of commands in both command line and GUI based tools, an example of a GUI tool is shown in Figure 3.6. The commands for showing the history of the software or for a specific file is *a*) *git log*, *b*) *git log -p filename*, or *c*) *gitk filename*. See also an overview of the Git commands in the appendix B.

Git is a very useful tool as it is possible to install a local system on a local computer as a local repository, install it on a server to have a common repository for a company or at home, or use GitHub as a cloud based repository. For GitHub will the repository by in the cloud, delivered as a type of software-as-a-service (SaaS) business model, and can easily by used for open source and equivalent projects. Command line and GUI based tools exists for access to the repository, one GUI tool is shown in Figure 3.6. This figure shows the main menu and four windows. The upper left windows shows the files that have been changed for your development system. You can click on a file and see any changes in the upper right window. By double clicking on the filename, the file state will change from changes to staged, and the filename will be present in the left lower window. Add your comments about the changes in the lower right window and use the commit button to update the local repository. The Remote option in the main menu can be used if a synchronization with any remote repository is needed and/or wanted. This way it is possible to have a local repository so you can work offline developing the software, and you can synchronize the local repository with a common repository at work (or home) when needed. It should also be possible to synchronize with a cloud based repository like GitHub.

The files in a git system can be in several states as shown in Figure 3.7. The files in a Git repository can be either in the tracked or untracked state, tracked files are part of the repository while untracked files are not part of the repository. A tracked file, part of the repository, can be either unmodified, modified or staged. A staged file is a modified (or an untracked) files that has been marked to be updated or included in the repository during the next commit command.

Branching is an important feature in Git and is implemented as a light weight "process" unlike many other version control systems. This means that branching in Git are not using too much resources (like memory and disc space) so it is an useful feature when adding new functionally and/or fixing software bugs. The usage of branching can be a challenge when handling both local and remote repositories but is a very useful feature which should be exploited.

Always remember to keep the repository updated, a special (and funny) note from a company is shown in Figure 3.8 (update both the local and remote repositories before leaving the building).

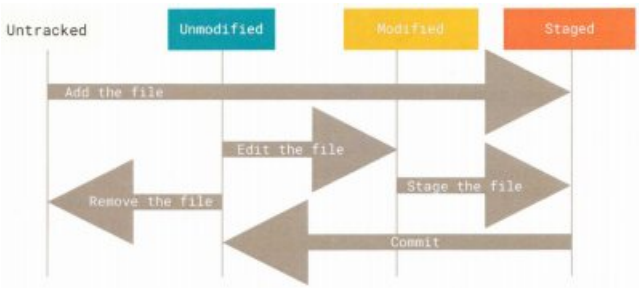


Figure 3.7: The states for a file in a Git system (Chacon 2009).

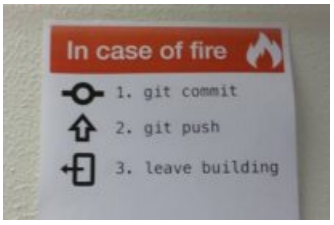


Figure 3.8: Always remember to commit your changes to the repository.

Chapter 4

The C# programming language

4.1 Introduction

This chapter is an introduction to the C# (pronounce C sharp) programming language and the .NET environment. C# is one of the programming languages in Microsoft Visual Studio and several "open source" solutions exist as well for both the Linux and the Windows platform. Mono as C# and Gambas as Visual Basic to meantion a few. C# was developed as a combination of the "best" from the programming languages Delphi, Eiffel, C, C++ and Java, and is fully object orientated like Java. See Figure 4.1 for these combination. Programming in C# will be very much like programming in C, C++, or Java.

C# is selected for this course because it is supported by the Microsoft Visual Studio and it is 100% object oriented. C# is a modernized version of C++. Originally you had the language C, which was widely used. C++ came about to add object-orientation to C, and C++ became the language of building "real" applications for Windows. Several programming languages exist, the important part is to learn to use a programming language, not the programming language itself. As always there exists no perfect programming language for all types of tasks. C is perfect for making interrupt functions, memory critical applications, system code for operating systems, device drivers and utility programs to mention some. Java can be suitable for client server application, at least if the other side is Java. C++ was used for writing the infrastructure and low-level applications, while Visual Basic (VB) developers wrote business applications. Both Java and C# are object oriented languages but requires more memory.

A programming language consists of a set of statements like calling a function or a method and returning an answer:

$$Answer = GetValue(B)$$

or setting a variable:

$$Answer = 5$$

A set of statements that can be called by a name are called different names like funtion, sub routine, procedure and method. The differences are:

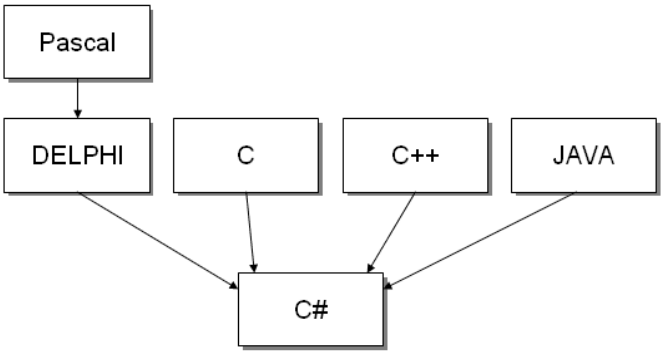


Figure 4.1: C# is a combination (the best from) other programming languages.

- sub routine; not returning any parameter, used in programming languages like Basic,
- function; returning a parameter, used in programming languages like Basic,
- function; not returning a parameter or returning a parameter, used in programming languages like C and C++,
- procedure; not returning a parameter or returning a parameter, used in programming languages like Pascal,
- method; used in classes, can be a sub routine or a function.

Methods are the object-oriented concepts of sub routine, function and procedures.
Some advantages of the C# programming language:

4.1.1 Type-safety

C# is type-safe, which means among others that :

- uninitialized variables can not be used. The C# compiler will notify you if you try to use a variable before it has been initialized it to some valid value.
- access outside an array is not allowed. This is possible in both C and C++, for example declaring an array of three elements, access the fourth element of that array, and get just a random value (whatever is in that memory location).

4.1.2 Fully Object-oriented

Even simple data types can be treated as objects, meaning that an integer has methods associated with it. For example, you can use the ToString method to get a string value for an int, as shown below.

```
int Counter=14;
Console.Write(Counter.ToString());
```

In addition, literal strings can be treated as objects and support a variety of methods, such as Trim, ToUpper, ToLower, and many others, as shown here:

```
Console.Write("hello, world".ToUpper());
```

4.1.3 Simplified syntax

C# attempts to simplify the syntax to be more consistent and more logical while also removing some of the more complex features of for example C++. C# also removes memory management issues from the developer (like Java) by using a garbage collection scheme. Items no longer referenced in memory are marked for garbage collection, and a “background” task will release these memory references and add them to the free memory list. Note that this background task is running on regular basis and using some cpu resources. This task can be an issue in real-time systems.

4.1.4 Standards

Microsoft released C# to European Computer Manufacturers Association (ECMA) and it has been published as a standard. The Mono project[1] is an effort to make an open source version of the .NET Framework (a subset of it called the CLI) and an open source version of C#, all for Linux.

[1] The Mono project: See http://www.mono-project.com/Main_Page (Jun-08).

4.1.5 Components

Creating components is straightforward, as is referencing those components in code. Namespaces in C# replace a lot of the headaches from the COM world in which you had to perform registry lookups and instantiate objects and worry about such things as IUnknown and IDispatch. Using C#, you simply import a namespace and then begin using the classes in that component—no registry lookups or COM plumbing required.

4.1.6 C# versus Visual Basic

When it comes to creating applications for .NET, the choice between VB.NET and C# is typically not important. However, it is quite possible that companies are willing to pay more for C# developers; C++ developers have typically been harder to find than VB developers and have been better compensated. It's likely there will be more VB.NET developers than C# developers, so this trend may well continue into the future.

4.1.7 Naming convention

The naming convention should be part of a coding standard and there exists several standard. This document will try to follow the "Upper Camel Case" and "Lower Camel Case" standards. The "Upper CameCase" is also called "Pascal Style". Only english names should be used on classes, methods and attributes, and the names should be self-explanatory.

"Upper Camel Case" means that the name contains no spaces nor underscore signs, but every word in the name starts with an uppercase letter like "EstimateModelOutput". "Lower Camel Case" means that the first word starts with a lower case letter, the rest of the words in the name starts with an upper case letter.

The project name, the class name, and the method name will be in UpperCamelCase style, while the attribuutes will be in lowerCamelCase style. Constants will be in upper case letter only.

Note that the author of the document has changed from an underscore type of coding standard to this camel case type of coding standard so there may be figures in this document that are not following this standard (yet). Also note that local attributes in a methods may have an underscore as the first sign to separate the it from the attributes for the class. All object references will use the same name as the class with a small r as the the first letter in the object name to indicate a reference to an object.

Some guidelines for naming convention:

Divide your software into managable modules like classes (one class in each source file) or several files in non object-oriented program languages. In non object-oriented program languages let the file name indicate the type of functions in the file, and let the first name of the functions be the name of the file to easy recognize in which file the function is defined. Some guidelines for better understanding and maintenance of the code, the headers of these modules should follow some standard format and contain some standard information like 1) the name of the module, 2) the date of module creation, 3) the author of the module and 4) the modification history. One main header for the module, and each function or method in the module should have a short header as well. Some guidelines for the naming conventions for local variables, global variables, constants and methode or functions should be 1) meaningful and understandable variables name always helps anyone to understand the reason of using it, 2) local variables should be named using camel case lettering starting with small letter (e.g. localData) whereas Global variables names should start with a capital letter (e.g. GlobalData), 3) constant names should be formed using capital letters only (e.g. CONSDATA), 4) avoid the use of digits in variable names, 5) the names of method or function should be written in camel case starting with small letters (e.g. methodeName()) and 6) the names of method or function must describe the reason of using the function clearly and briefly (e.g. getConfigDataLogFileName()).

See also Appendix D as an example of the contents of a coding standard.

4.2 Integrated development environment

An integrated development environment (IDE) is a software application that provides facilities to programmers for software development. An IDE normally consists of a source code editor, a compiler and/or interpreter, build automation tools, and (usually) a debugger. Sometimes a version control system and various tools are integrated to simplify the construction of different modules like GUI etc. Many modern

IDEs also have a class browser, an object inspector, and a class hierarchy diagram, for use with object oriented software development. This section will show the usage of the Visual Studio IDE from Microsoft and how to make a GUI or an console application.

4.2.1 Visual Studio IDE Application

The Visual Studio Application from Microsoft Cooperation (MS-VS) is one of the IDE (Integrated Development Environment) that can be used for developing software applications. Other examples of IDE are Eclipse and NetBeans to mention some. In the startup window in MS-VS the project must be selected, either a new or an existing project. Note that depening on the setup of the application, in some settings the programming language can be selected as well. In such a case, use C# only in this coures.

An existing project will just load an already defined C# project. A new project will require to select the framework, in that case select either Windows Form Application or Console Application for this course. In addition should the name and location of the project be defined as well.

A Console Application is easier to program for the novice programmer, the outpout from such an application is shown in Figure 4.19. Note that no graphical elements will be avaialbale for this application, only a terminal window.

A Windows Form Application contains graphical elements and will normally require some more experience in graphical programming to use. The output from a simple Windows Form Application is shown in Figure 1.14.

4.2.2 Windows Forms or Console application

A standard Windows application has a GUI with among others buttons and input fields. These GUI units will use a Windows Form and will require some understanding of the GUI. A console application will have a much simpler UI with printouts (and readins) only and can be used if the GUI part of the application is not wanted.

Console application

The selection of a Console Application will give you a window with the code for a default console application. Figure 4.2 shows the default console application. The code can be compiled and run, and will show you a "DOS window" (black window) for a short while. Use the *Build* menu and the selections *Build Solution* (F6), *Build AppName* (Shift+F6), or *Batch Build* to compile the application. Use the *Debug* menu and the selections *Start Debugging* (F5) or *Start without debugging* (Ctrl+F5) to run the application.

Input and output in a command line application The system namespace contains the Console class for input and output in a console (or command line) application.

Input methods:

- Console.Read()
- Console.ReadLine()

Output methods:

- Console.WriteLine(" .. ") ; => Write a new line with the text, and ends with a newline code,
- Console.Write(" .. ") ; => Write the text, without the newline code,
- Console.WriteLine(" .. " + ivalue1 + " .. " + ivalue2 + " ..") ; => Output of integer values,
- Console.WriteLine(" ..{argnum:Fx} .. ", fvalue) ; => Formatted output, argument displayed as float, with x decimal(s).

Summary of formatting codes:

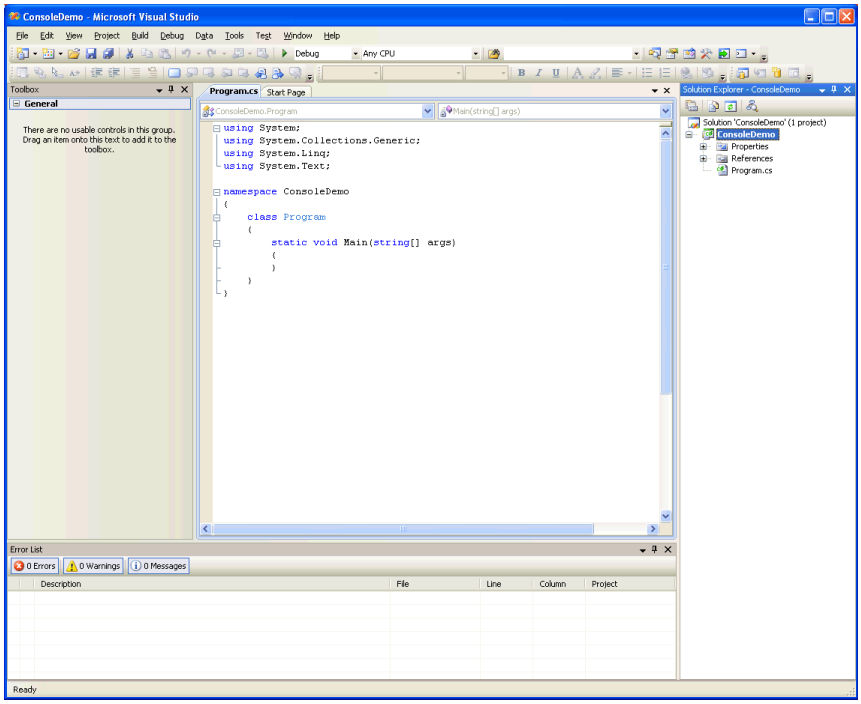


Figure 4.2: The default application for a console application, shown with the Visual Studio 2008 IDE.

Codes		**Meaning**
C	c	Currency
D	d	Decimal (decimal integer)
E	e	Exponent
F	f	Fixed point; Fxx where xx defines the number of decimals.
G	g	General
N	n	Currency
P	p	Percentage
R	r	Round-trip (for floating-point values only); guarantees that a numeric value converted to a string will be parsed back into the same numeric value
X	x	Hex

There are special formatting codes for time and date presentations. An example:

```
Console.WriteLine(" There are {0} students in OOADP, the average grading is {1:F1}.",
                  noOfStudents, fValGrading) ;
```

GUI parts in a Windows Form

Developing a GUI application requires the interfaces for buttons, text fields, lists etc. in your application. All the code for these graphical elements will be generated by the IDE, but you hade to interface these elements into your application. The default GUI application will start as shown in Figure 4.3. The code can be compiled and run, and will show you only an empty graphical window. Use the *Build* menu and the selections *Build Solution* (F6), *Build AppName* (Shift+F6), or *Batch Build* to compile the application. Use the *Debug* menu and the selections *Start Debugging* (F5) or *Start without debugging* (Ctrl+F5) to run the application.

You can add graphical elements to your application by selecting an element from the tool box to the left and place the element into your application form. Figure 4.4 shows a button added to your graphical form. The location and size of the button can be changed using the mouse. The properties of the button is available in the lower right windows, text is the property for the text of the button. The text can be changed by using this property window, together with all the other properties of the element.

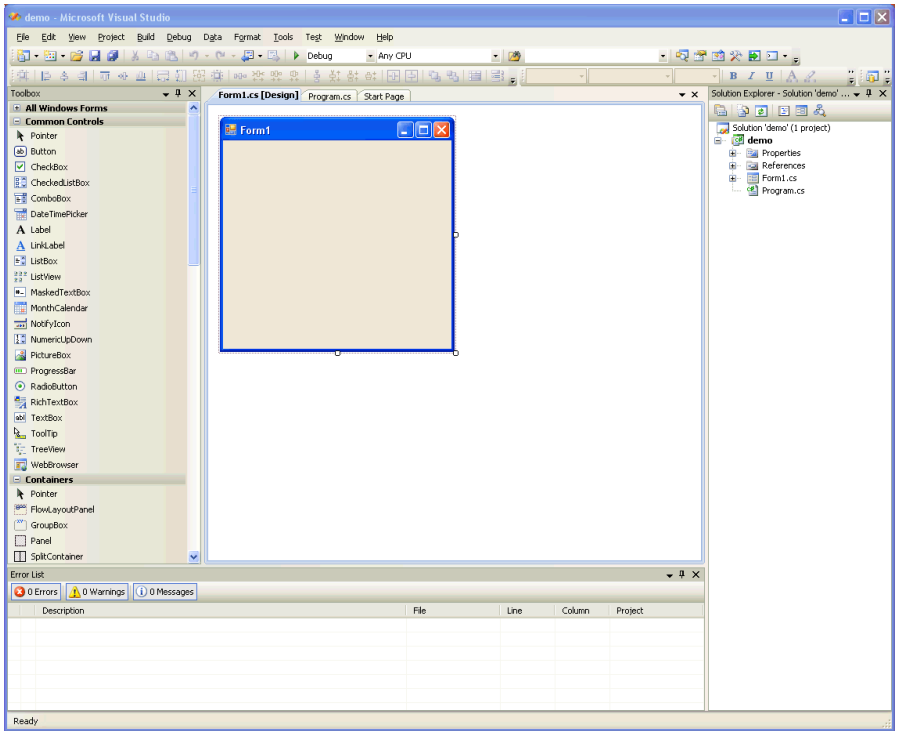


Figure 4.3: The startup window using a Windows Forms application.

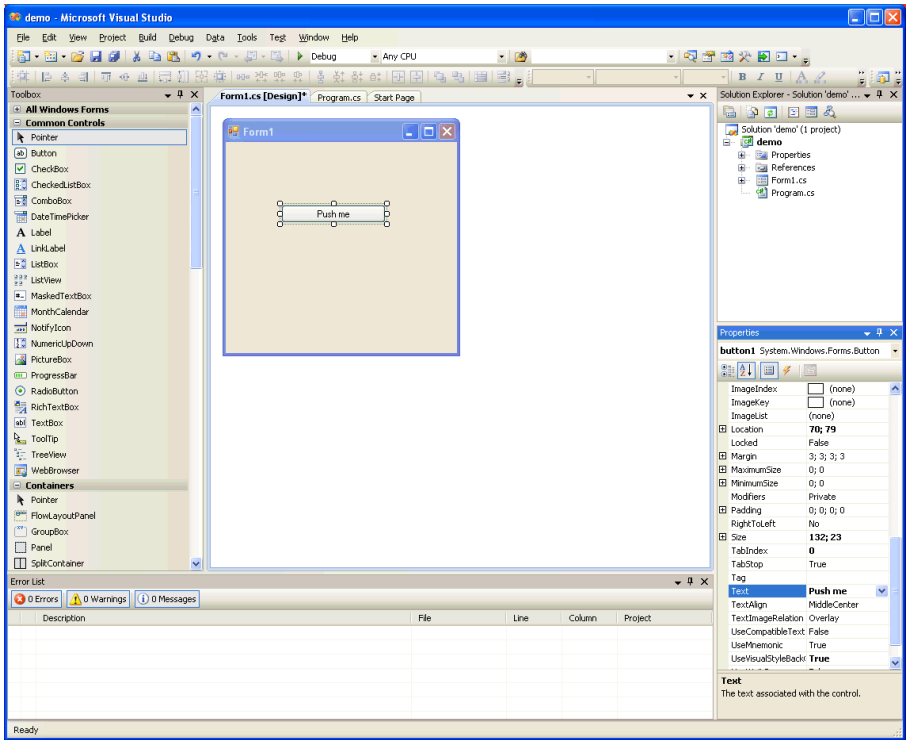


Figure 4.4: A button element is added to your graphical form.

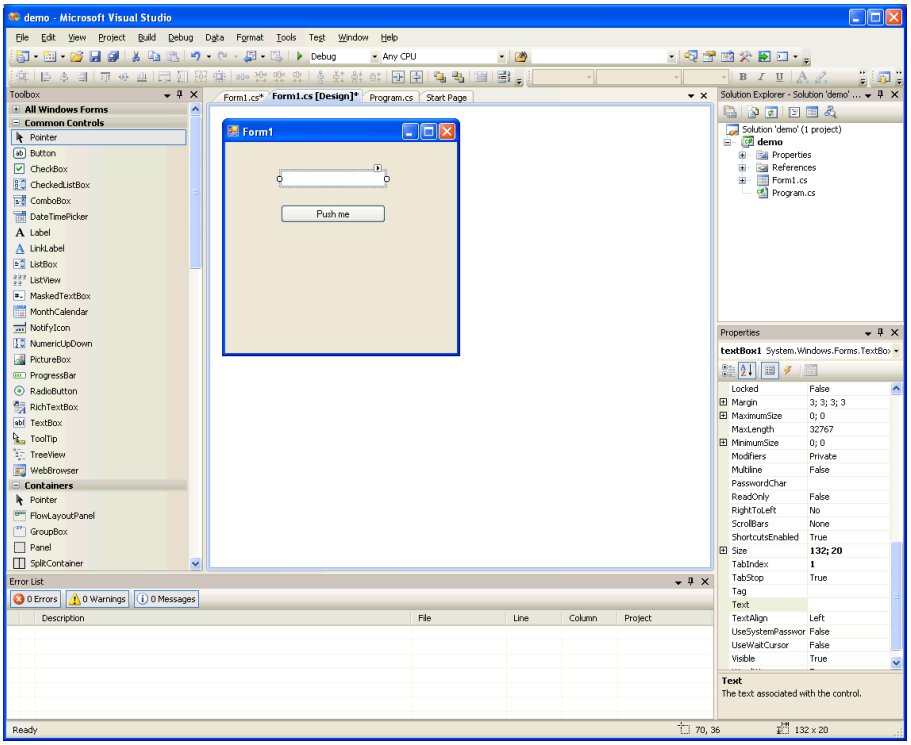


Figure 4.5: A textbox is added to your graphical application for output of text.

The properties of the element can also be changed in the code. Expand the Form1.cs selection in the model explorer (upper right window) and select the Form1.Designer.cs to get the C# code for graphical codes for your application.

A text field is also added as shown in Figure 4.5, the text field is a textbox from the tool box. Now you have 3 graphical elements in your application, the Form, the button, and the textbox. The property of each element can be changed using the property window or changing directly in the code. The advice will be to use the property window. If you want to change the name in the form, click on the Form element and change the name from Form1 to any name you want.

Now you need to add some code for these graphical elements, at least you want some action when you are pushing the button. By pushing the button in your application will the IDE make an empty click function. This is shown in Figure 4.6 where the IDE has generated the function button1_click() with the right number of parameters and the return code. You must define the action inside this function, and you want to write a text in the textbox.

The graphical elements are assign default names when added to a graphical application, the advice is to change these names to more describing element names. This can be done in the property window as well. In this application is the first button called button1 and the first textbox is called textBox1 as shown in Figure 4.6. Better names can for example be button_push_me and textBox_welcome, a good advice will be to still include the type of element in the name.

Compile and run this application will give the application in Figure 4.7. The textbox will be empty or have the text from the property window when starting the application. When clicking on the button, the text should change to whatever you write in the click function.

However, it could be nice to blank the textbox as well. Let us include a new button to clear the textbox. Figure 4.8 shows a new button added to the graphical application.

Click on the button to add code, Figure 4.9 shows the code added to make a blank textfield. The textfield of textbox1 is set to a blank or empty textstring. Compile and run your application.

The name of the Form will default be *Form1*, but you normally will use your name of this form, often the same name as the name of the application. Figure 4.11 shows how the name of the form can be changed to another name, in this example to *SCE Demo*.

In newer versions of Visual Studio will Git be integrated so check-out and check-in can be done within the Visual Studio.

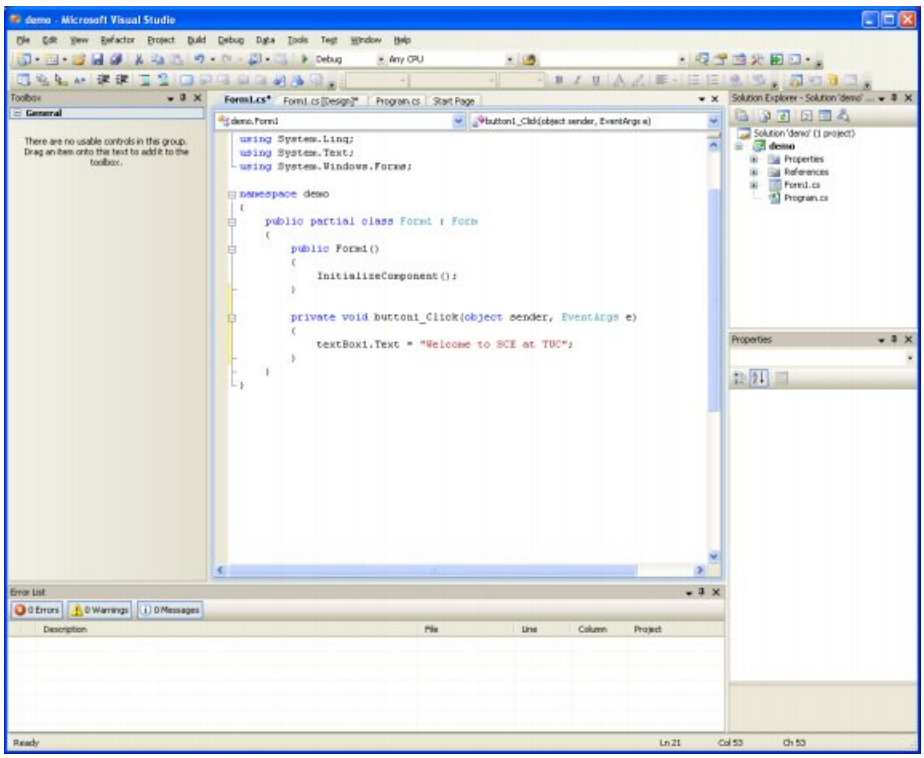


Figure 4.6: The code for the button, the function is created by the IDE when you click on the button for the first time, and you must add the code inside the function.

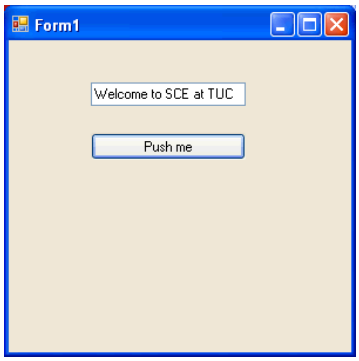


Figure 4.7: Your first graphical application with a button and a textbox, after clikcing on the “Push me" button.

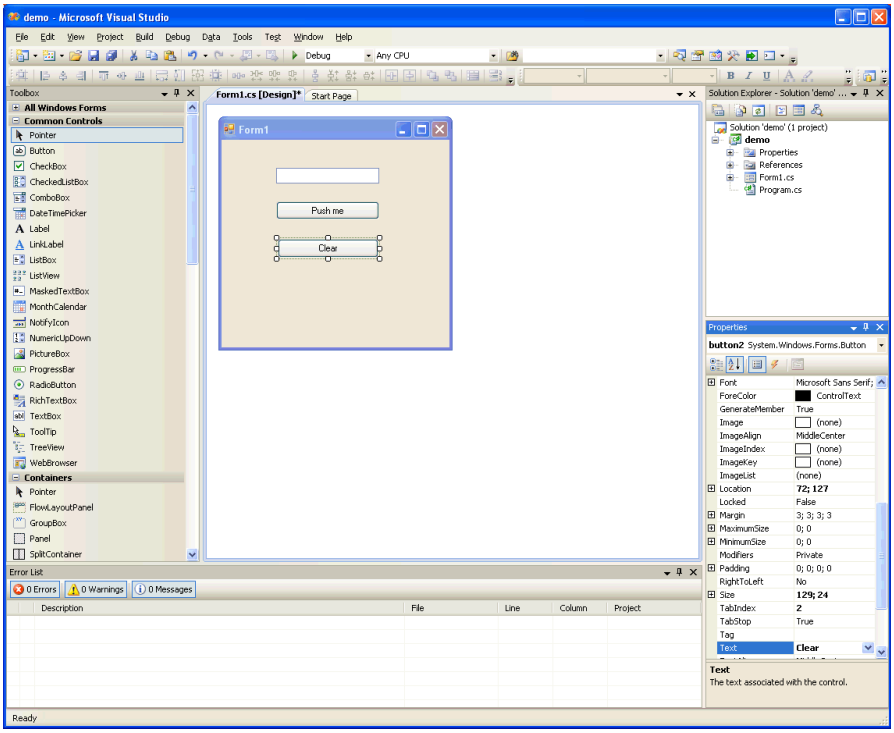


Figure 4.8: A “Clear” button is added the application.

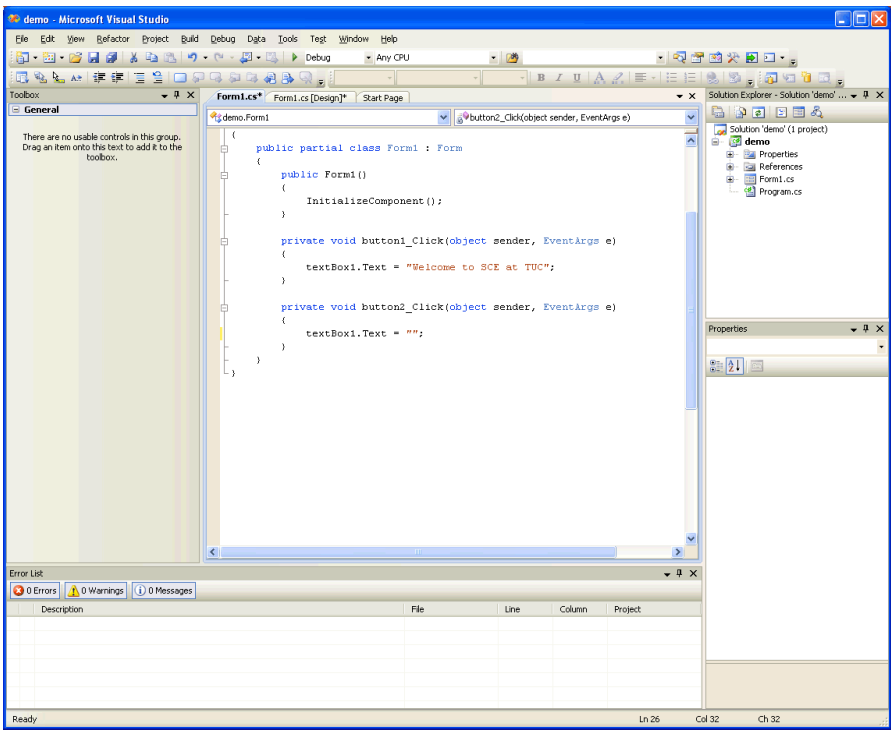


Figure 4.9: The function and code for the “Clear” button. The code is just setting the text field to a blank text.

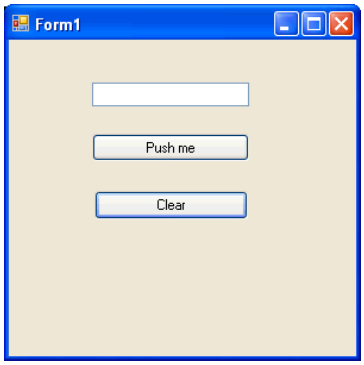


Figure 4.10: Your next graphical application with two buttons and a textbox, after clikcing on the *Clear* button.

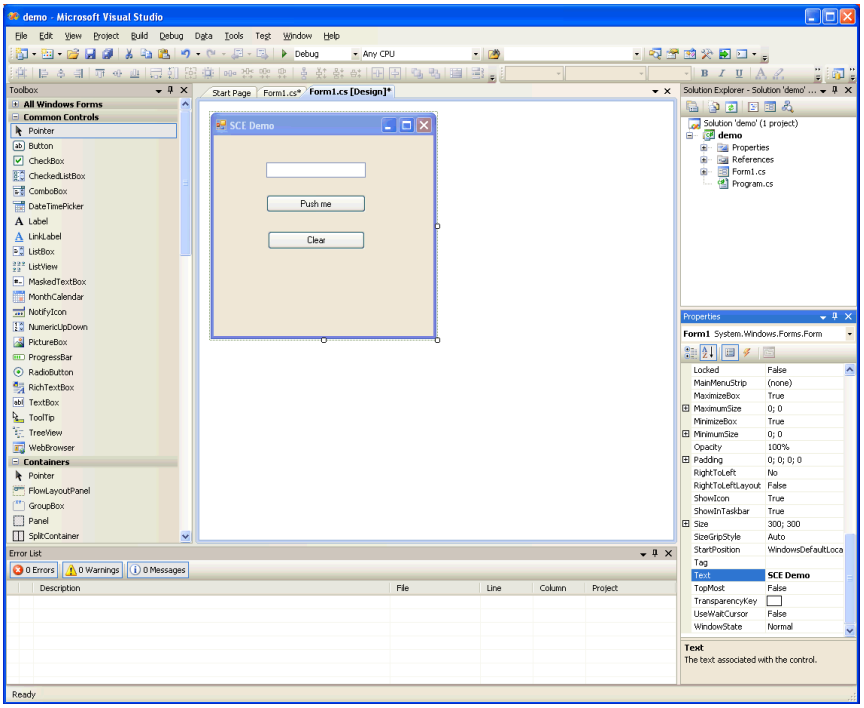


Figure 4.11: Use the property window to change the name of the form as well.

4.3 Requirements for an OO programming language

An OO programming langauge should support three main requirements:

1. Encapsulation
   (a) hidden attributes (data) in a object, using only methods (get.. and set..) for accessing the attributes in an object,
   (b) hidden sub function(s) from other objects,
   (c) important for maintaining of the software.
2. Polymorphism
   (a) different interfaces for the same method,
   (b) define the body of the method only once,
   (c) important for maintaining of the software.
3. Inheritance
   (a) a class can inherit properties from another class,
   (b) define the common part of classes only once,
   (c) important for maintaining of the software.

4.4 .NET programming languages

C# is a standard[2] OO programming language consisting of a "runtime environment" called "Common Language Infrastructure" (CLI). The structure of the CLI is shown in Figure 4.12 defining the compilation to a virtual platform used for the "Common Language Runtime" (CLR). The runtime module is available free of charge for a lot of different hardware platforms.

The Common Language Specification (CLS) defines a set of rules for the languages that can be used with the CLI. The CLI defines how a compiler should convert the CLS specific programming language to a CLR type of code. All compatible languages compile to this Common Intermediate Language (CIL), which is an intermediate language abstracted from the platform hardware. When the code is executed, the platform-specific CLR will compile the CIL to the machine language according to the specific hardware. Each type of hardware architecture must have a specific CLR module.

This way can an application develoved in the .NET environment be able to "run anywhere", just like Java. The difference is however that Java is an intepreter, while CLI is mend to be a "Just-In-Time" (JIT) compiler. An interpreter is a program designed to run other non-executable programs directly, without a compiler. A compiler is a computer program that translates text written in a computer language (the source code) into another computer language (the object code).

An interpreter is interpreting the source code, line by line, when running. A compiler is transforming the source code to an object code for a specific hardware architecture, often in a EXE file. The EXE file can then be used for running on any system with the same hardware architecture. This is shown in Figure 4.13.

A compiled program is running faster then an interpreter, but must be compiled for the right platform. An interpreter will make code for the CPU on the platform it is running. C# is somehow in "between", a Just In Time compiler (or interpreter) being able to be faster then an interpreter, but can run "anywhere" just like Java. C# can run on any platform where the CLR is present.

There exists a lot of books about the C# programming language, some of these books are (Allwork 2009), (Allwork 2010), (Schildt 2002), (Troelsen 2003), (Jeffries 2004) and (Özgül 2012).

4.5 C# control statements

This section will give an overview of the C# control statements and briefly how these statements can be used.

[2] European Computer Manufactoring Association (ECMA): ECMA-334 and ECMA-335.

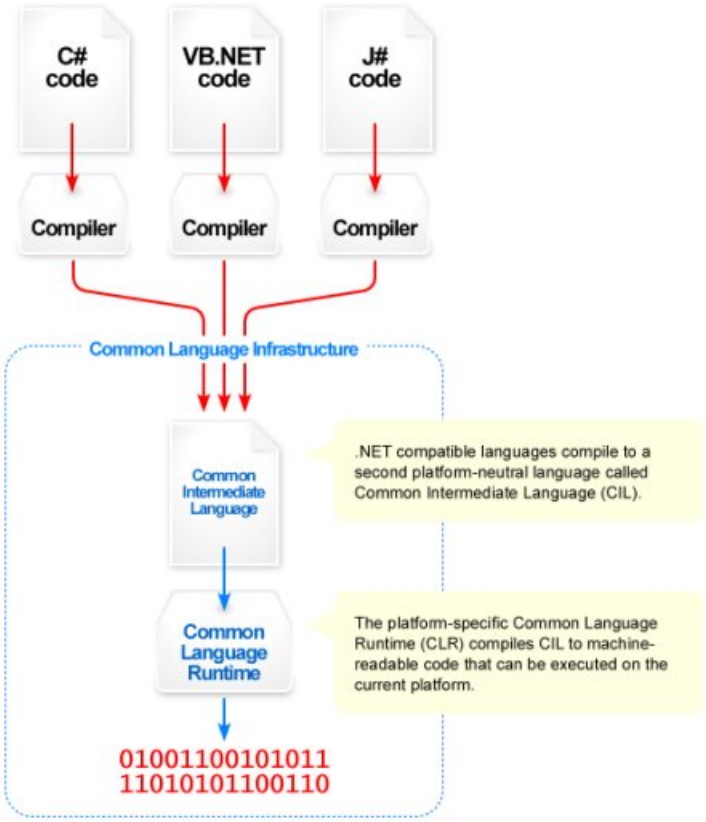


Figure 4.12: Visual overview of the Common Language Infrastructure (CLI) (www.wikipedia.org 2006).

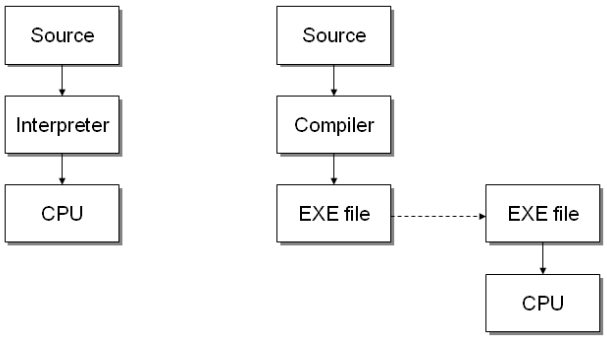


Figure 4.13: Interpreter and Compiler

4.5.1 The if statement

This statement works with an expression that evaluates to a Boolean value. If the Boolean expression evaluates to true, the statement embedded in the if clause will be executed. The if statement can be followed by an else statement that is executed if the Boolean expression used in the if clause evaluates to false. Several else if sections can be combined.

Note: Two equals signs check for equality, whereas one equals sign simply performs an assignment.

The following is an example of an if statement:

```
int i=3;

if (i == 1)
{
    MessageBox.Show("One");
}
else if (i==2)
{
    MessageBox.Show("Two");
}
else
{
    MessageBox.Show("i=" + i.ToString() + "!");
}
```

4.5.2 The switch statement

This statement evaluates an expression and compares the expression's value to a number of cases. Each case is associated with a statement list that is called a switch section. Switch is an easy-to-read programming alternative to having many if/else pairs whenever you are trying to evaluate a value of a particular expression.

The expression used as the driver for the switch statement is displayed in parentheses that follow the switch keyword. In general, the expression used in the switch statement must evaluate to one of the following types: sbyte, byte, short, ushort, int, uint, long, ulong, char, string. You can also utilize an expression that can be implicitly converted to one of the above types.

Switch sections start with the keyword case, which is followed by a constant expression. A colon follows the constant expression, and the statement list follows the colon. The break specifies the end of the statement block.

The default keyword in the switch blocks is utilized to execute a particular section of the code whenever none of the cases listed match the value switch block's constant value.

In general, it's a good practice to utilize the default keyword in order to make sure that all possible scenarios are being handled in the code and can't fall through the cracks in program logic.

The following is an example of a switch statement:

```
int i=3;

switch (i)
{
  case 1:
    MessageBox.Show("One");
    break;
  case 2:
    MessageBox.Show("Two");
    break;
  default:
    MessageBox.Show("i=" + i.ToString() + "!");
    break;
}
```

4.5.3 Iteration statements

While, do, for, and foreach are considered iteration statements because they assist in executing embedded statements multiple times. Here's more information about the four statements.

The while statement

This statement executes a statement list while an expression evaluates to True. The Boolean expression that controls the while statement is enclosed in parentheses that follow the while keyword. The statements that should be executed while the Boolean expression evaluates to True follow the parentheses.

The following is an example of a while statement:

```
int i=3;
while (i < 5)
{
  MessageBox.Show("The variable i=" + i.ToString());
  i++;
}
```

The do statement

This statement executes its embedded statements zero or more times. If the Boolean expression used in the while statement evaluates to false, then none of the embedded statements will be executed. However, in order to ensure that the embedded statements are executed at least once, you can use a do statement.

The do statement is followed by embedded statements, which are followed by a while keyword. The Boolean expression that controls the number of times that the loop executes follows the while keyword. The embedded statements will be executed at least once since the Boolean expression is evaluated after the embedded statements execute.

This is an example of a do statement:

```
int i=1;
do
{
  MessageBox.Show("The variable i=" + i.ToString());
  i++;
}
while (i < 3);
```

The for statement

This statement begins with the for keyword and is followed by parentheses. The parentheses contain an initializer, a condition, and an iterator statement, all separated by semicolons. The embedded statements follow the parentheses.

This is an example of a for statement:

```
for (int i=0; i<4; i++)
{
  MessageBox.Show("The variable i=" + i.ToString());
}
```

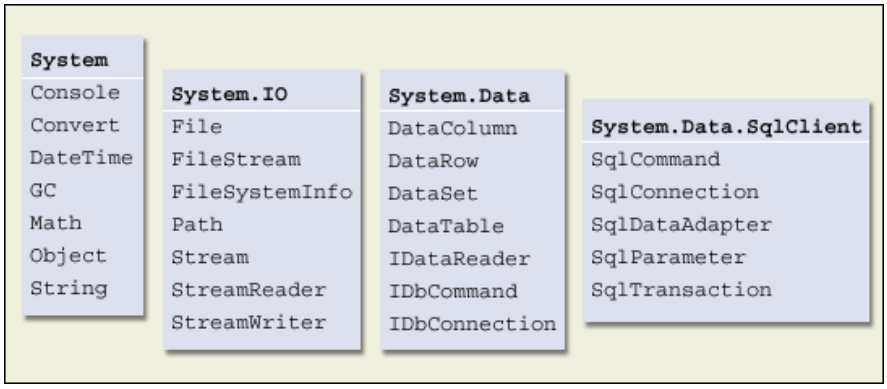


Figure 4.14: The System namespace for the .NET environement. The namespace are divided into a set of sub namespaces consisting of classes.

The foreach statement

This statement is used to iterate over the elements in a collection. Since arrays in C# provide support for the foreach statement, you can use the foreach statement to work with each element in the array.

To use the foreach statement, type the foreach keyword, followed by parentheses. The parentheses must contain the following information: the type of the element in the collection; the identifier name for an element in the collection; the keyword in; and the identifier of the collection. Embedded statements follow the parentheses.

This is an example of a foreach statement:

```
int[] intArray;
intArray = newint[3];
intArray[0] = 0;
intArray[1] = 1;
intArray[2] = 2;

foreach (int ArrayElement in intArray)
{
  MessageBox.Show(ArrayElement.ToString());
}
```

4.6 Namespace

Microsoft's Visual Studio .NET has introduced many new concepts to the Visual Studio developer, including the Microsoft Intermediate Language (MSIL) with runtime compilation, garbage collection, Common Language Runtime (CLR), and, perhaps most misunderstood of all: *namespaces* and *assemblies*. This section will explore namespaces and assemblies and clarify the relationship between them.

4.6.1 The namespace

It's a method of putting classes inside a container so that they can be clearly distinguished from other classes with the same name[3]. A namespace is a logical grouping rather than a physical grouping. The physical grouping is accomplished by an assembly, which equates most directly to a dynamic link library (DLL), COM object, or OCX module.

The .NET CLR consists of multiple namespaces, which are spread across many assemblies. Figure 4.14 shows the System namespaces and how classes are divided up in the namespace that compose the .NET CLR. Each block represents a separate namespace. In the CLR, the classes and structures contained in each of the namespaces represent a common theme of development responsibility.

[3]Programmers skilled in the Java language will recognize namespaces as packages.

In the .NET environment, namespaces represent the logical organization of code but not necessarily the physical location of the code.

4.6.2 Assemblies

In Visual Studio .NET, code is physically organized and structured in assemblies. An assembly consists of one or more DLL or EXEs. Those with Java skills will recognize the assembly as a Jar file.

An assembly is a binary file that contains .NET executable code complied into MSIL. The relationship that exists between assemblies and namespaces can be rather confusing, since one assembly can contain one or more namespaces, and one namespace can be contained by one or more assemblies. Each assembly can be configured with a root namespace. If the root namespace value is left blank, the root namespace value defaults to the assembly name. You can set the root namespace through the project properties dialog box, as shown in Figure B.

Assemblies provide many benefits over traditional DLL, OCX, or COM files. The following are just some of the advantages offered by the assembly design:

Assemblies form .NET's security boundary, which, with the assistance of the computer's security policy, can specify what actions can be performed.

Assemblies introduce a cleaner method of object versioning than that presented by COM.

4.6.3 Setting the root namespace

While standard assemblies can be registered and placed in the global assembly cache, an assembly does not have to be registered to be useful. Unlike COM objects, which required the presence of entries in the NT/9X registry, the presence of an assembly in your application's directory is sufficient for it to be recognized by your application. This permits the development of simple applications that can be executed without an install process.

4.6.4 Using classes

Suppose you have a class called Manager residing in the Employee namespace, which resides in the Company.dll assembly. To use the class, you must first reference the assembly using the References section of the Solution Explorer. For Java programmers, this is analogous to placing the Jar file in the class path. This tells the project where to find the physical file. Once a reference is in place, you can use the *imports Employee* (VB) or *using Employee* (C#) directive to make use of the namespace. Imports can be done on a file basis or on a project basis.

4.7 C# class

A class declaration in C# is composed of attributes, modifiers, the class name, bases, and a body. Attributes, modifiers, and bases are all optional. The body contains class members that can include constants, fields (or variables), methods, properties, indexers, events, operators, and nested types. Nested types are defined by class, interface, delegate, struct, or enum declarations within the class body.

4.7.1 Attributes

The data of the class. Normally the attributes will be private, using modifiers to access the data.

4.7.2 Modifiers

Modifiers allow you to control access to the class and to restrict the ability of the class to be instantiated or to serve as the base of another class. The class modifiers are public, protected, internal, private, abstract, sealed, and new. Using these attributes, the following access levels can be specified:

public	Access is unrestricted,
protected	Access is restricted to the containing class or to classes derived from it,
internal	Access is restricted to the containing assembly (program),
protected internal	Allow protected or internal access. You can also specify this access level as internalprotected,
private	Access is restricted to the containing class.

A class that is not nested within another class may specify only public or internal access, while a nested class or other class member may specify any of the five access levels. If the modifiers are omitted, access defaults to internal for nonnested classes and private for nested classes and other class members.

The abstract modifier indicates that a class is incomplete and must be extended by a derived class in order to be instantiated. Conversely, a sealed class cannot be extended—it cannot serve as the base for another class. These attributes are useful in frameworks to ensure that classes are used as intended. In addition, the sealed attribute enables virtual class methods to be optimized at run time since it isn't necessary to allow for method overriding in derived classes. The abstract and sealed attributes are mutually exclusive.

Modifiers can be methods or properties. The constructor is a special method called only when the object is created in memory.

4.8 C# constants, fields, properties and methods

4.8.1 Constants and fields

A constant provides a way to define a named value that is strongly typed. A field defines a variable within the class. The value of a constant must be specified in the declarion, and can not be reassigned.

```
class A
{
  public const int k = 1000;
  public const string me = "Paul";
}
```

4.8.2 Property

Properties are primary methods to control the access to the data in an object. There exists two kinds of access controls, get and set; 1) reading a value with the get acces; 2) writing the value with the set access. At least one access must be used, the other one is just optional. An example of getting or setting a channel of the DAQ device:

```
public int DaqChannel
{
    get
    {
        return ch;     // ch is an integer
    }
    set
    {
        ch = value;    // value is the input parameter to the property
    }
}
```

4.8.3 Methods

Methods are primary means to define behavior for a class. As with the class itself, the access control modifiers and the new, abstract, and sealed modifiers may be applied to a method. The virtual, override, and extern attributes may be applied as well. A class may declare methods that either require or allow customization by a derived class. An abstract method provides no implementation and requires a nonabstract derived class to provide one. A virtual method provides an implementation but allows a derived class to provide an alternative. In both cases, a derived class is said to override the base class method.

4.8.4 Method parameters

A method specifies a return type (or void), a name, and a formal parameter list. The name and the formal parameter list are known as the signature of the method. A class may contain more than one method with the same name as long as each method has a different signature. Under such circumstances, the method name is said to be overloaded.

By default, a fixed parameter is passed by value. The ref modifier can be specified to pass the parameter by reference. The out modifier can be specified to declare that a fixed parameter is an output parameter. A variable passed as the argument for an output parameter does not require initial assignment before being passed in a method invocation.

4.8.5 Method or Property

Use a method if the object is going to do some action (verb), and use a property if a reference to the object information is wanted (noun).

4.8.6 #define

#define can be used for testing or debugging purposes. The block or section of code can be controlled at compile time. The variable *Test* can be set or not, if set will the test code be included in the application, if not set will the test code not be included. This example shows how the additional code between the lines *#if Test* and *#endif* will be included in the application. To unset the variable Test, one way can be to comment the #define line like *// #define Test.*

```
#define Test
..
#if Test
..
#endif
```

4.9 C# coding summary

4.9.1 General statements

- Comments:
  - Can use both C and C++: // or /* ..*/.
  - Recommendations: Use only //,
- using: Include a name space, a sort of library. System namespace includes among others input and output functions (keyboard and display),
- public: Indicate that the method (or attribute) will be available from other classes,
- static: Indicate that a method will be available when the system starts, before creating any classes,
- semicolon (;): End of a statement (possible with more statements on the same line),
- block of code ({}): Set of statements belonging together, indented with spaces (normally 4),

4.9.2 Data types

- bool: true or false,
- byte: unsigned integer, 8 bits, 0-255,
- char:character, unicode, 16 bits, 0-65535,
- decimal: 128 bits (account systems),
- double: floating-point number, double precision, 64 bits,
- float: floating-point number, single precision, 32 bits,
- int: integer: 32 bits
- long: long integer: 64 bits
- short: short integer: 16 bits

4.9.3 Operators

- Math: +, -, *, /, % (modula), ++, - - (increment and decrement one),
- Relations: ==, !=, <, >, >=, <=
- Logical: && (and), || (or), ^(xor), ! (not)
- Assignments: int x; x=10; x++; x+= 10;
- Bitwise: & (and), | (or), ^(xor), >> (shift right=divide), << (shift left, multiply), ~(one's compliment)

4.9.4 Program control

C# contains several statements for program control like if, for, and while. The use of these statements are:

- if:

```
if (condition)
{
   statements ;
}
```

- Nested if:

```
if (condition #1)
{
   if (condition #2)
   {
      statements ;
   }
   else
   {
      statements ;
   }
}
else
{
   statements ;
}
```

- Else if:

```
if (condition #1)
{
   statements ;
}
else if (condition #2)
{
   statements ;
}
else
{
   statements ;
}
```

- Switch:

```
switch (expression)  // Note: expression must be of type integer!
{
   case constant1 :
      statements ;
      break ;
   case constant2 :
      statements ;
      break ;
   default :
      statements ;
      break ;
}
```

- for loop:

```
for (initialization; condition; iteration)
{
    statements ;
}
```

- while loop:

```
while (condition)
{
    statements ;   // Note that it can loop forever!
}
```

- do-while loop:

```
do
{
    statements ;   // Note that it can loop forever!
} while (condition) ;
```

- foreach loop:

Figure 4.15: A challenge with while loops, how to end: While (out == true) buyMilk;

```
foreach (type variable_name in collection)
{
    statements ;
}
```

- return: end of method (with or without parameter)

```
return parameter ;
```

The challenge with the *while* and *do-while* loops is when to end. Be sure that the condition will change otherwise the loop will run for ever and your application seeems to be "hanging", as shown in Figure 4.15. Add a counter if you are not sure like *while (condition == false && counter < max).*

4.9.5 Arrays and strings

NOTE: First index is **always** 0, and last index is **always** $n - 1$!

- One dimension array:
  - type[] array_name = new type[size] ;
  - Example: int [] sensor_tag = new int[1024] ;
- two dimension array: type[,] array_name = new type[size1, size2] ;
- Multidimension array: type[,,,] arrayname = new type[size1, size2, ..., sizeN] ;
- Array references: sensor_tag[x] = 8 ; or sid = sensor_tag[x] ;
- length property: Array is objects, and contain the Length property => sensor_tag.Length ;
- string: string txt = "Hello world" ;
- string operations: string.Length, string.ToLower(), string.ToUpper, ..

4.9.6 Classes, objects, and methods

Definition of a class:

```
class cname
{
  // declare variables, public or private (default)
  access type var1 ;
  access type var2 ;
  ....
  access type varN ;
  // declare methods
  access return-type methodname1(parameters)
  {
    statements ;
  }
  ....
  access return-type methodnameN(parameters)
  {
    statements ;
  }
}
```

- Make a class: cname object_name = new cname() ;
  - Recommendation: Make one source file for each class in the project, use same name of the source file and the class.
- access: private or public, (*private is default !*)
  - Used for encapsulation.
- return-type: use **void** if no return,
- parameters: List of input parameters, separated by comma if more. Empty list if none.
- Calling a method in an object: ObjectName.method() ;

4.9.7 C# class example

An example of a class for reading temperatures, and the output from the example as an command line application in Figure 4.17. The class and object diagram is shown in Figure 4.16. The left part is the class diagram showing two different classes, the *Tempdemo* and the *TempSensor* classes. The right part is the object diagram shown the that the *Tempdemo* object (class instance when running) will make two objects of the *TempSensor* class when running. The source code for these classes will be:

```
//
// Temperature Sensor readings
//
using system ;
//
class TempSensor
{
   const int max_channel = 64 ;
   int channel_id ;          // Channel number
   int tag_id ;              // Tag number
   float zero_value ;        // Zero adjustment
//
```

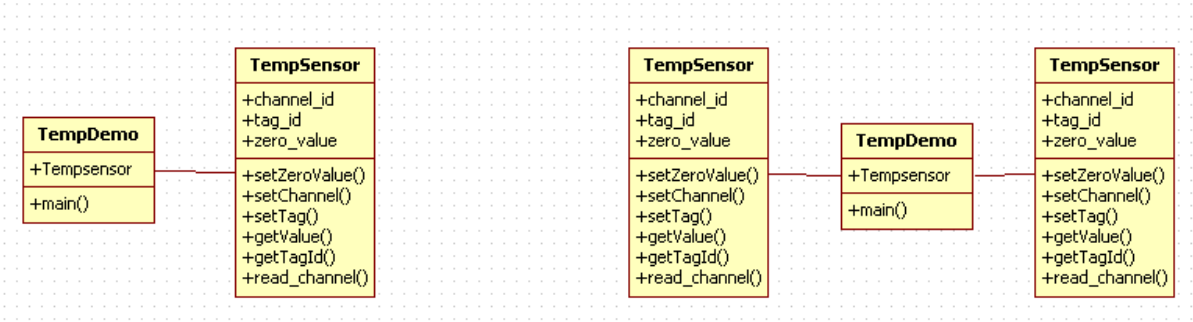


Figure 4.16: The temperature demo application shown as a class diagram to the left and an object diagram to the right. The application is using two objects of the tempsensor class.

```
   public void setZeroValue(float zero)
   {
        if (zero > -5.0F && zero < 5.0F)
            zero_value = zero ;
   }
//
   public void setChannel(int ch)
   {
         if (ch >= 1 && ch <= max_channel)
              channel_id = ch - 1 ;
   }
//
   public void setTag(int tag)
   {
           if (tag > 0)
               tag_id = tag ;
   }
//
   public float getValue()
   {
        float fval = read_channel(channel_id) ;
        fval -= zero_value ;
        return(fval) ;
   }
//
   public int getTagId()
   {
         return(tag_id) ;
   }
//
// Private method for reading temp value
//
   float read_channel(int channel)
   {
          return(15.0F + (float) channel) ;
   }
}
class TempDemo
{
   public static void Main()
   {
       TempSensor bedroom = new TempSensor() ;
       bedroom.setZeroValue(0.0F) ;
```

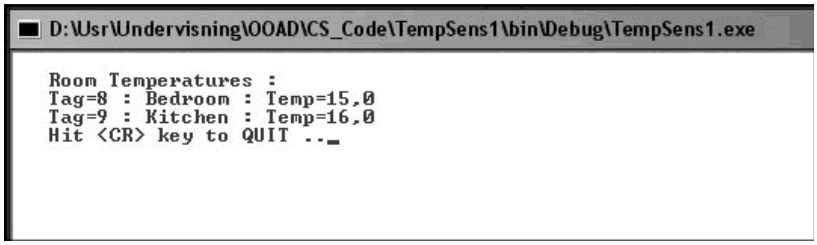


Figure 4.17: The output from the Temperature Sensor Readings example.

```
        bedroom.setChannel(1) ;
        bedroom.setTag(8) ;
        TempSensor kitchen = new TempSensor() ;
        kitchen.setZeroValue(0.0F) ;
        kitchen.setChannel(2) ;
        kitchen.setTag(9) ;
        Console.WriteLine(`` '') ;
        Console.WriteLine(`` Room Temperatures :'') ;
        Console.Write(`` Tag='' + bedroom.getTagId()) ;
        Console.WriteLine(`` : Bedroom : Temp={0:F1}'', bedroom.getValue()) ;
        Console.Write(`` Tag='' + kitchen.getTagId()) ;
        Console.WriteLine(`` : Kitchen : Temp={0:F1}'', kitchen.getValue()) ;
        Console.Write(`` Hit <CR> key to QUIT ..'') ;
        Console.ReadLine() ;
    }
}
```

4.9.8 Constructor

A method with the same name as the class, is called when the object is created. access will normally be public.

access class-name()
{
... ;
}

Making a constructor for the Temperature Sensor example will be:

```
public TempSensor(int cid, int tid, float fval)
{
        setZeroValue(fval) ;
        setChannel(cid) ;
        setTag(tid) ;
}
```

Making the bedroom object can now be:

```
TempSensor bedroom = new TempSensor(1, 8, 0.0F) ;
```

4.9.9 Method parameters

1. Call_by_value: A copy of the value is transferred to the method. Changes in the method will NOT influence the original value.

2. Call_by_reference: A reference to the value is transferred to the method. Changes in the method will influence the original value.

```
noChange(x) ;
// Call-by-value
int noChange(int x)
{
   ...... ;
   if (x < 0)
      x = -x ;
   x++ ;
   .... ;
}
```

```
Change1(ref x) ;
// Call-by-reference
int Change1(ref int x)
{
   ...... ;
   if (x < 0)
      x = -x ;
   x++ ;
   ...... ;
}
```

```
Change2(out x) ;
// Call-by-reference
int Change2(out int x)
{
   x = 0 ;
   if (x < 0)
      x = -x ;
   x++ ;
   ...... ;
}
```

Important:

1. A ref parameter must be set before calling the method,
2. No requirements of setting the out parameter before calling the method.

4.9.10 Return value

A method can return a type of values, but only one value. Use *ref* and/or *out* if more parameters are needed.

4.9.11 Method overloading

One way C# is implementing polymorphism. Different methods in a class can have the same name, as long as the parameter list is different. Example:

```
public void setZeroValue(float zero)
{
   if (zero > -5.0F && zero < 5.0F)
         zero_val = zero ;
}
public void setZeroValue(float MeasVal, float CorrVal)
{
   setZeroValue(MeasVal - CorrVal) ;
}
```

HINT: Use method overloading only on methods with the same function, and define the statements in only one of the methods.

Method overloading can also be used on the constructor.

4.9.12 Inheritance

By using inheritance, a class can inheritance properties (attributes and methods) from another class. The class that is inherited is the *base class*, and the new class is the *derived class*. Using inheritance to avoid copies of the same attributes and methods. Note that the base method use the definition *virtual* and the derived class use the *override* definition. This means that the classes can use the same name of the methods and that the *override* method will be used at run-time instead of the *virtual* method. If no *override* method exists, the *virtual* method will be used.

Procedure:

1. Make the general classes,
2. Make the specific classes, by inheriting the general classes.

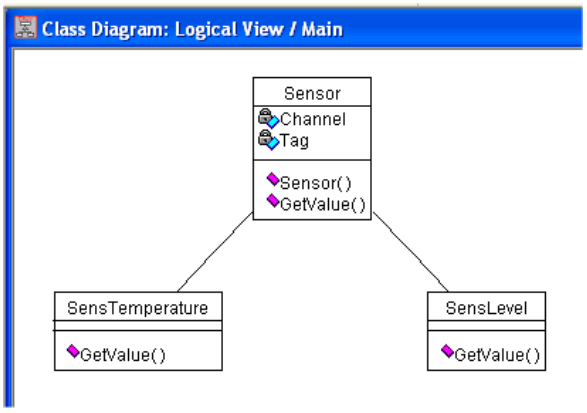


Figure 4.18: UML class diagram with inheritance, using sensor classes.

Example of Inheritance:

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

namespace SensorInheritance
{
    // Sensor base class
    class Sensor
    {
        int Channel;
        int Tag;

        // Constructor
        public Sensor(int ch, int id)
        {
            Channel = ch;
            Tag = id;
        }
        public virtual float GetValue()
        {
            return Tag ;
        }
    }
    // Temperature Sensor
    class SensTemperature : Sensor
    {
        public SensTemperature(int ch, int id) : base(ch, id)
        {
        }
        public override float GetValue()
        {
            float fval = base.GetValue();
            fval *= 8.8F;
            return fval;
        }
    }
    // Level Sensor
    class SensLevel : Sensor
    {
        public SensLevel(int ch, int id) : base(ch, id)
```

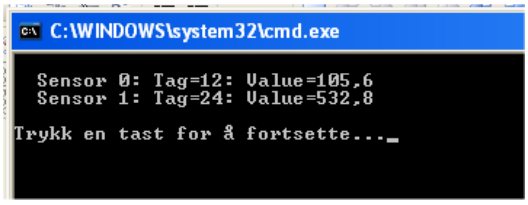


Figure 4.19: The output from the console application (Note: the output of the tag info is removed from the source code).

```
        {
        }
        public override float GetValue()
        {
            float fval = base.GetValue();
            fval *= 22.2F;
            return fval;
        }
    }
    // Program Class
    class Program
    {
        static void Main(string[] args)
        {
            Sensor[] sid = new Sensor[2] ;

            sid[0] = new SensTemperature(1,12) ;
            sid[1] = new SensLevel(2,24) ;

            Console.WriteLine(" ") ;
            for (int id = 0; id < sid.Length; id++)
            {
                Console.Write("  Sensor " + id);
                Console.WriteLine(": Value=" + sid[id].GetValue());
            }
            Console.WriteLine(" ");
        }
    }
}
//
/////////////////////////////////////////////////////////////////
```

4.9.13 Partial Class

It is possible to split the definition of a class over two or more source files. The **partial** keyword indicates that other parts of the class (struct or interface) can be defined within the namespace.

4.10 Testing

Testing is used for verifying the functions of the software. The challenge with testing is to know what to test, the functions must have been described in such a way that they are testable. Statement like a user friendly user interface or a fast user response is very difficult to test. Testing must also be done as early as possible in the development process as it will be more and more expensive to correct errors later on in the process. Testing will also be more complicated when the complexity of a system increases so it is important to test at several levels. Start with unit testing, testing each units as methods and classes,

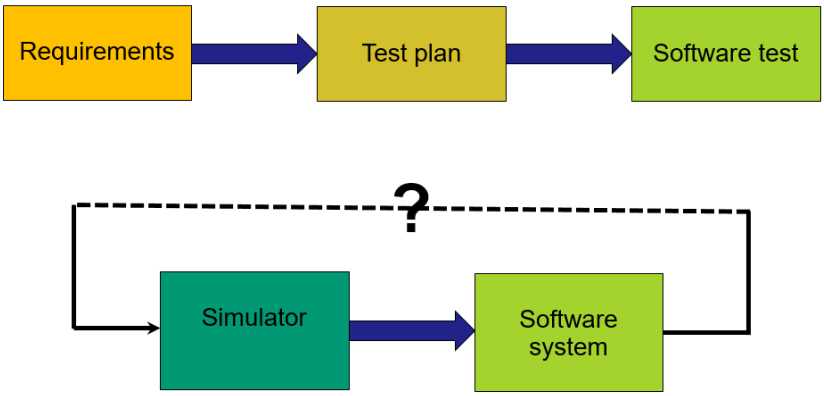


Figure 4.20: A simulator for an automated test of input and output combinations. The testing should be based on a testplan, made from the requirements. The test simulator should be based on the test plan.

before testing modules. The details must be tested in the metods of the classes, the combination and interation must be tested by combining classes in modules and the whole system.

Testing can be boring and should be automated as much as possible. Automated testing can also be used every time a new version of the software is compiled and/or deployed. Automated testing can be solved by adding a method in each class testing the other methods in the class, and returning true or false. A test option can then be to call all the test methods, and check the return parameter. If all of them are true, then is the details of each method working, and the manual testing can focus more on different combinations and interaction in the system. All software application will have one or several inputs, and one or several outputs.

A simulator can be a very useful tool for testing the combinations, by varying the inputs and check the outputs according to the input patterns. Very often will the output depend on the configuration of the system, but this can be solved by letting the simulator using the same configuration data as the application are using. Figure 4.20 shows such an approach where a test plan is made based on the requirements, and the test simulator is made based on the test plan. The simulator will automate the tesing of any combination of input and output information for the software system.

Part II

Construction Phase - Iteraction #2 (CP-I1)

Chapter 5

Introduction

5.1 Introduction

The construction iteraction #1 is based only on programming to get more familiar with programming in C# and using objects. This iteraction will deal with all the steps in developing software, analysis, design, construction, and testing, all steps in an iteration. This iteration will be according to the following list:

Inception	**Elaboration**	**Construction #1**
Analysis	Introduction Version control Dev. process	Requirements Use case diagram Domain model Use case documents System sequence diagrams
Design	Introduction Version control Dev. proces	GRASP patterns Sequence diagrams Class Diagrams Object diagrams
Coding	C# coding - version control - program controls - classes/objects	Exceptions Threading
Testing	Debugging	Testing Test cases

This iteration will also include the usage of an UML tool for making the UML diagrams.

5.2 Why use UML?

Unified Modeling Language (UML) is one way of making models of the software, a way of making requirements and "sketches" of the software. The reasons for using UML is to decrease the cost, the development time and the maintenance time when developing and deploying software. UML will also increase the productivity, the quality and make it easier to maintain the application, but it is important that UML is integrated into the development process. UML is a standard, making it easy for many developers to be familiar with the models, and the diagrams documenting these models. UML will make it easier to focus on the analysis and design steps in each iterations as the models will be developed or extended in these steps, and will be part of the documentation as well. UML will also help in making a better design as the models will be used in the design steps, and make it better to test the applications as the test should be based on the requirements. UML models should give a faster overview of the software design when expanding the functionality of the software.

Chapter 6

Analysis

6.1 Introduction

The basis of the analysis part will be some sort of specification. The specification can be in any form, from an oral description to many pages of written descriptions. The specification very often define some sort of work that should be improved by the software (system). The first step will be to understand the work process. The work process normally consists of a set of busines use cases, where the use cases describe a specific function or operation in the work process (Robertson & Robertson 2013). These business use cases will normally be performed according to some business events, one or several events that may influence when to start a business use case, and the outcome of a business use case can also be the input for a new business use case.

A use case is a list of actions or event steps defining the interations between a system and an external role to achieve a goal. The focus in use cases is the boundary of a system, the "high level" interfaces between the system and the external "world", and the expexted actions in the system based on these interfaces.

6.1.1 Business use cases

The specification will always be about improving the work, either a new business use case or improving an existing business use case. The business use case is part of the business model. The business model is a descriptions to represent the core aspects of a business like the purpose, business processes, target customers, strategies, and infrastructure to mention some. Business models are used to describe and classify businesses, and can be used to explore possibilities for future development in a specific business.

The work will be in response to a business event, and there can be new business events, the combination of business events may have changed or just an improvement of an existing business use case. The work will be a set of business activities. The relationship of the work, business use cases and business events is shown in Figure 6.1. The figure also shows that the work or business area may receive information from adjacent systems that can/should be used in the business use cases.

When analysing software it is important to understand the work or business area, and the business use cases performed. The specification is normally a description of how the work or the business use cases can be improved. The first steps in starting the analysis will be:

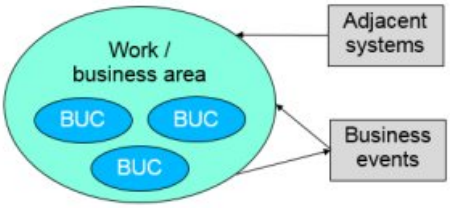


Figure 6.1: How business events and business use cases (BUC) are related to work or business area. The work or business area may also depend on infromation from adjacent systems.

1. The specification; the specification is a description of the idea from the owner/user on how to improve the work,
2. Get an overview of the work or business area; get an overview of the work process, business events and any adjacent systems,
3. Based on the work processes, find the business use cases related to the business events,
4. Describe the business use cases; a use case describe the interaction between a system and the user of that system.

The important part will be to get the specification, get an overview of the work performed, and understand the specification related to the work. In the overview of the work, a set of business use cases are defined. These use cases describe the interaction between the system and the user of that system. Based on the business use cases a set of requirements for the system should be defined.

6.1.2 Specification and requirements

A specification is a document that specifies a system or product. There should be several sections/content in a specification like requirement definitions, reference documents, glossary, and verification information to mention the most important information. Often the main focus in a specification will be the needs and wishes of a system. Anyway, the specification very often end up as an informal document, some times also just an oral description (Robertson & Robertson 2013). This is the main reason why collecting requirement is important. The users are responsible for the specification, while the project leader and/or developers are responsible for the requirements.

A requirement is a statement of something the product or system must do. The requirements must be something that can be tested. A requirement document may have hundreds of requirements in it and it is a good advice to include all the requirements into a requirement document. The basis for the requirement document is the specification, and one part of the specification may end up in many requirements.

6.2 Requirements

The requirements are the specification of a system, in this case will the focus be the specification of the software. Software can solve almost any problem. The challange is that we don't always understand what the problem is. Understanding the real problem is the role of the requirements process. When the real needs are known, it is possible to determine the system that best serves those needs, and to specifiy, completely and innovatively, the requirements to get the right system analysed, developed and built.

The requirements are normally collected in a text document based on the specifications and wishes. These requirements are important as they will be the basis for the analysis, design, development, testing, project management, delivery times, and cost. This means that it is important to make unambiguous requirements that can be tested.

6.2.1 Challanges in collecting Requirements

An overview of the requirements, how to collected them and use them in the software development process.

1. Requirements are the capabilities and conditions to which the system (or project) must confirm,
2. Collection of requirements are 1) the workflow for finding and describing the requirements of the system to be developed, and 2) description of the requirements for the system in a way that the users, the customers and the developers have a common understanding and/or agreement about the system to be developed.
3. A challenge to find and store all the requirements in a way, being obvious for both the users, customer and the developer. An advise is to use a requirement document,
4. Using UP for realizing the changes in the system. fulfil only a small set of the requirements in each iteration.

5. Using UP will let the user to test the system after a number of iteration, allow him to give feedback about the use and functionallity of the system.

6. Important to log the type and number of changes in some sort of documents, an advice is to use an appendix or extend the requirement document.

Requirements are important for defining the project, developing the software and testing the software (system). The collection of requirements is important because it is the bases for: 1) the contract like defining the amount of time to use and the size of project team, 2) the development of the software, like which requirements in which iteration, and which requirements in the releases for the users, and 3) the testing of the system (software).

Challanges in collecting the right type of requirements can be indicated by the survey of Standish Group. This survey dealed with more than 352 companies reporting on more than $8,000$ software projects with the following results:

1. 31% of all software projects are cancelled before they are completed (a waste of $81 billion),

2. 53% of projects cost 189 percent of their original estimate,

3. In large companies, 9% of projects are on time and within budget,

4. In small companies, 16% of projects are on time and within budget.

The same survey also asked respondents to identify the causes of these challanges and failures. The top three reasons why projects fail:

1. Lack of user input / user involvement: 12.8%,

2. Incomplete requirements and specifications: 12.3%,

3. Changing requirements and specifications: 11.8%.

The main reasons are requirements and user involvement, and these reasons are normally connected. This shows the importance of collecting the requirements and involve the users in an early stage of the developmnet. Poor or incomplete requirements are the largest problem when developing software. Without any proper requirements it is impossible to 1) estimate the cost of make the software, 2) create a project plan, 3) assign resources, and 4) test the software.

Some of the reasons why collecting requirements can be a challange are: 1) to understand which requirements the system are suppose to solve, and which requirements shall it not solve, 2) that the developers are not familiar with the problem domain, 3) that there exists many users with different needs and requirements (and remember only a part of it, or rembering only the last problems concerning these needs and requirements), 4) that the description of the requirements in an ordinary text often gives a poorly structure (of relations), 5) that the needs and requirements are changing during the project time, 6) that the system shall satisfy all users and the customer, and 7) the ability to get people to tell what they really need, rather than their perceived solution, or what they think you might be able to deliver.

6.2.2 Collecting the requirements

Collecting the requirements are very important for a system, but it will always be a challange. Some factors that will be important when collecting the requirements are the:

1. starting point of the system - an existing or nonexisting business model,

2. workflow for the system,

3. candidate requirements, ideas from users, customer, designers etc.

4. understanding of the context of the system (relations, problem domain, business model),

5. collections of functional requirements (the functions of the systems),

6. collections of non-function requirements (OS, performance, reliability).

The important factors will for the developer to understand the problem domain, understand the system "boundaries" and the functional and non-functional requirements. The collection of the requirements will be an important helping tool for the developer. Figure 6.2 shows the results of collecting the requirements from a customer. Even on simple systems or processes it may be difficult to collect the right type of requirements, the right combination of requirements, and be able to deliver what the customer really want.

6.2.3 Using FURPS+ for collecting requirements

Use each character in the FURPS (Bruegge & Dutoit 2010), (Larman 2002), (Lieberman 2007) word for remembering collection of the different type of requirements:

1	F	Functional
2	U	Usability
3	R	Reliability
4	P	Performance
5	S	Supportability
6	+	Design challenges/limitations

The Quality parts of the FURPS: The words ending on *..ility* influence most the quality of the system. A more detailed description of FURPS+ will be:

- Functional: Use case diagram and detailed use case documents, **describe with text how the use cases should work**, shows the main functions of the system (in this case, the software only!),
- Usability:How the user can/will interact with the system, Time to learn, online help, documentation, standards (Windows), GUI devices, storage, DAQ devices (Bentley 2005), etc.,
- Reliability: Safety, Mean Time Between Failure (MTBF), 24x7 (running all the time)?, recoverability, predictability, temperature range, dust, etc.,
- Performance: Response times, throughput, accuracy, transactions, capacity, any memory requirement, other resource usage,
- Supportability: change or extend the system, add new functions, add new algorithms, configuration, internationalization, operating systems, maintainability,
- +: Implementation; resource limitations, languages, hardware,
- +: Interface; constraints imposed by interfacing with external systems,
- +: Operations; system administration and management during operation,
- +: Legal; licenses and copyrights.

NOTE: The functional description is describing the main functions of the software, the functions should be verbs, and each function should be followed by a short description of the function (a couple of lines)!

The requirements should stated how the system should work, include specific requirements not general requirements. The requirements must be testable. "Fast user response" is not a requirement, it is a specification or user wish.

Some exercises for practice the FURPS+:

Exercise 2 *Indicate the requirements for the software of your next mobile phone,*

Exercise 3 *Indicate the requirements for the software to the measurement system,*

Exercise 4 *Indicate the requirements for the software of your washing machine (Rinse, wash* 40 °C*, wash* 60 °C*, and dry).*

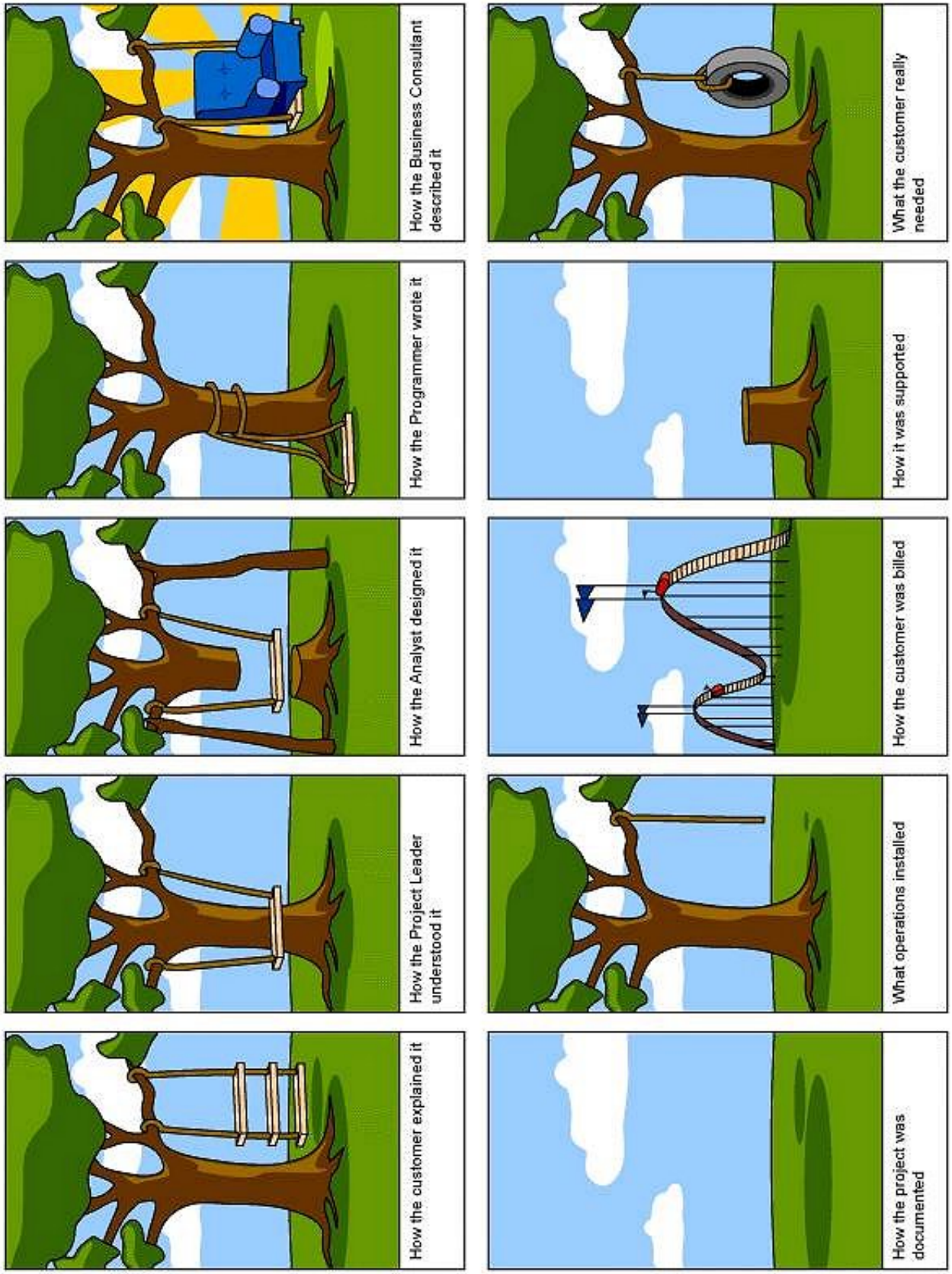


Figure 6.2: The collection of requirements evaluated from different positions and departments. Note especially the difference between what the customer needed and how he was able to explain his need (www.jhouseconsulting.com - 2008).

6.2.4 Prototypes

The analysis phase is for understanding the problem domain and good advice can be to make small protoypes (Whitten & Bentley 1998), especially of the user interface. Making small protoypes of different part of the system is a good way to get more knowledge about the system, and the protoypes can be used in discussions with the customers on how the system should work. A prototype is an early model, sample or software module, often with no or minimum functonality of the product, as a concept for discussions or testing used for learning the problem domain. The user interface is often discussed during the analysis phase by using prototypes, either on paper or as an application. Any selected prototypes of the user interface can be included as part of the requirements. However, it is important to inform the users that this is only a prototype, otherwise they may think that the development is almost finished!

6.3 Use case

6.3.1 Introduction

Use cases[1] (Williams 1995),(Fowler & Scott 1997),(Larman 2002),(Martin 1993),(Oestereich 2001),(Pender 2003), (Holmstedt 2019) are used for describing the user functions of the software. A use case diagram is normally an overview diagram of the user functions for the software, showing the system as a black box. A black box means that we are not going to describe the details inside the system, but just the user functions for the system and how the system is reacting with the "outside" world. A use case document is describing how each use case in the use case diagram is working or reacting with the "outside" world. The use case diagram and the use case documents will then describe all the user functions of the system.

1. A use case:
   (a) describes how the system is going to be used,
   (b) is a good way to understand and describe the requirements,
   (c) is an important part of the UP.
2. A single use case consist of:
   (a) scenario, describing the functionality of the system (subfunctions). Starts with a verb,
   (b) actor, something influence or is influenced by the system (everything in or out of the system), often a person or another computer system.
3. An actor:
   (a) primary actor: the use case fulfil the goal,
   (b) supporting actor, provides a service for the use case,
   (c) offstage actor, just an interest in the behavior of the use case.

6.3.2 Use case diagram

A use case diagram is showing all the use cases of the system. A use case diagram may also be denoted as a use case model in some litterature. In this course will the use case model be the combination of the use case diagram and the use case documents. Note that in large projects sub use case diagram can be used, dividing the use cases in the main use case diagram into new use case diagrams.

A use case diagram is an overview diagram showing the main actors and functions in a system. An example is shown in Figure 6.3. This Figure shows a simple system with one function, the use case for this system. The function is getting data from the input actor and delivering data to the output actor. Remember that all systemes need at least one input actor and one output actor!

Important information for use case diagrams:

1. All the use case's shall describe the main functionality of the system, both the normal functions and the irregular behavior of the system. This is the **use case model,**

[1] In Norwegian: use case = bruksmønster

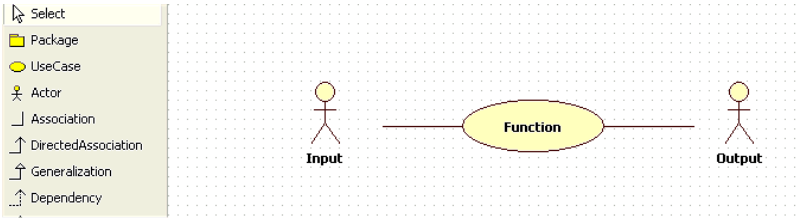


Figure 6.3: An example of a use case diagram showing a use case, the function, with input and output actors.

2. Focus on how the system shall be used to fulfil the requirements of the actor(s),
3. Use cases are describing the behavior of the system, each use case will be based on a sequence of events,
4. Include a verb in the use case name as the use cases are something the system is going to do (responsibility),

Advices about use case diagrams:

1. Use the use case diagram to get an overview of the behavior of the system, or the functionality requirements of the system,
2. Do not overcrowd the diagram with use cases, maximum about 10 use cases (use sub diagrams if more use cases),
3. make a use case document of each behavior element in the use case diagram,
4. Remember UP, select only one or two $(5 - 10\%)$ of the most important use case behavior elements in the use case diagram in each iteration. (This will depend on the size of the project, number of project members, and type of development process).
5. Focus on the business use cases, not how to solve the functions using a programming langauge.

6.3.3 How to make a use case diagram

The use cases and the use case diagram is a visualization of the functional and non-functional requirements, the functional requirements being the use cases, and non-functional requrements can be the actors. The use case diagram is made to gain more knowledge about the problem domain (business model) and better understand the business "processes". Some advices for making a use case diagram:

- The basis will be the requirements, and the functional requirements.
- Make one use case for each of the functional requirements. Remember to focus on the functional requirements for the business, not the software.
- When the use cases are defined, use the non-functional requirements to define the actors,
- Interconnect the use cases and actors.

Very often the number of use cases and/or actors will increase during the process of making the use case diagram, remember to update the requirement document.

Start with making the use cases, based on the functional requirements. Include a verb to show that the functional requirements are something thta the system shall do! Define the system boundary to be able to separate between what is the functions of the system and what are the actors. The actors will be input or output events across the system boundary. Define the goals and/or requests for the actors, this is part of the non-functional requirements for the system. Check that the use cases are covering these goals and requests for all the actors. It may be a challange to find all the actors, this may also affect the number of error conditions for the system. Some extra hints for checking available actors:

1. some primary actors is often obvious, or use the events of the system (system inputs and outputs) to find them.

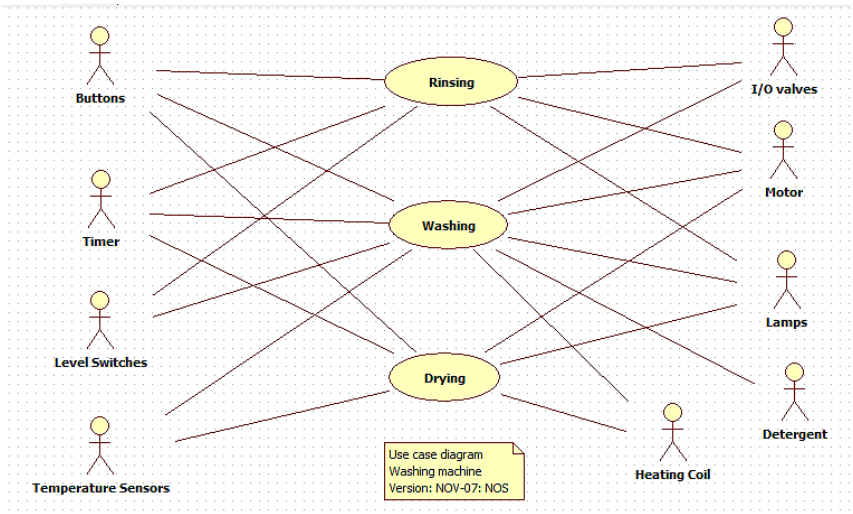


Figure 6.4: One solution for an use case diagram for a washing machine, defined in the previous exercise section.

2. Additonal questions for identifying other actors:

   (a) Administration, start, stop and user management of the system ?

   (b) Security management of the system?

   (c) Is "time" an actor, response of or to an time event ?

   (d) Monitoring of system failure, performance, log or errors ?

   (e) Software update ?

Keep the use case diagram focused on the business use cases. Keep the user interface out of the use cases, focus on the purpose of the actor (business function). Use the "black box" approach, focuc on the events between the actors and the system (system boundary). Do NOT focus on how the system is going to handle these events, that is the purpose of the design phase not in the analysis phase.

An example of a use case diagram can be a washing machine. How to analyse the software for a washing machine and make a use case diagram?

Figure 6.4 shows one solution of a use case diagram for a washing machine, as defined in the previous exercise section. The software has three main functions, and several actors for fulfiling the non-functional requirements. Note that there are many ways of making the use case diagram, there exists no perfect way of doing it. The main reason for the use case diagram is to visualize the main requirements to get an better overview of the problem domian. It is important that the project leader, the developers, the customer, and the users have the same understanding of this diagram.

The use case diagram will be defined in an early phase, in the elaboration phase, and will be the basis for the UP process. Remember that the UP process is said to be use case driven. The next step is to select one or a few use cases, and start to analyzing this use case in more details. The number of use cases depend on the size of the project and the number of members in the project. One or a small group of people will normally only work on one use case in each iteration. The first step in analyzing a specific use case will be to make a use case document describing each use case in more details.

6.3.4 Exercises

Exercise 5 *Make the use case diagram for your next mobile phone.*

One example of a use case diagram for the mobile phone is shown in Figure 6.5. The use case diagram has two actors, the user and the telecom operator, and the main functions of the mobile phone. Some of the functions are only local, used only by the user, the other functions also involve the telecom operator.

Exercise 6 *Make the use case diagram for the software of the measurement system.*

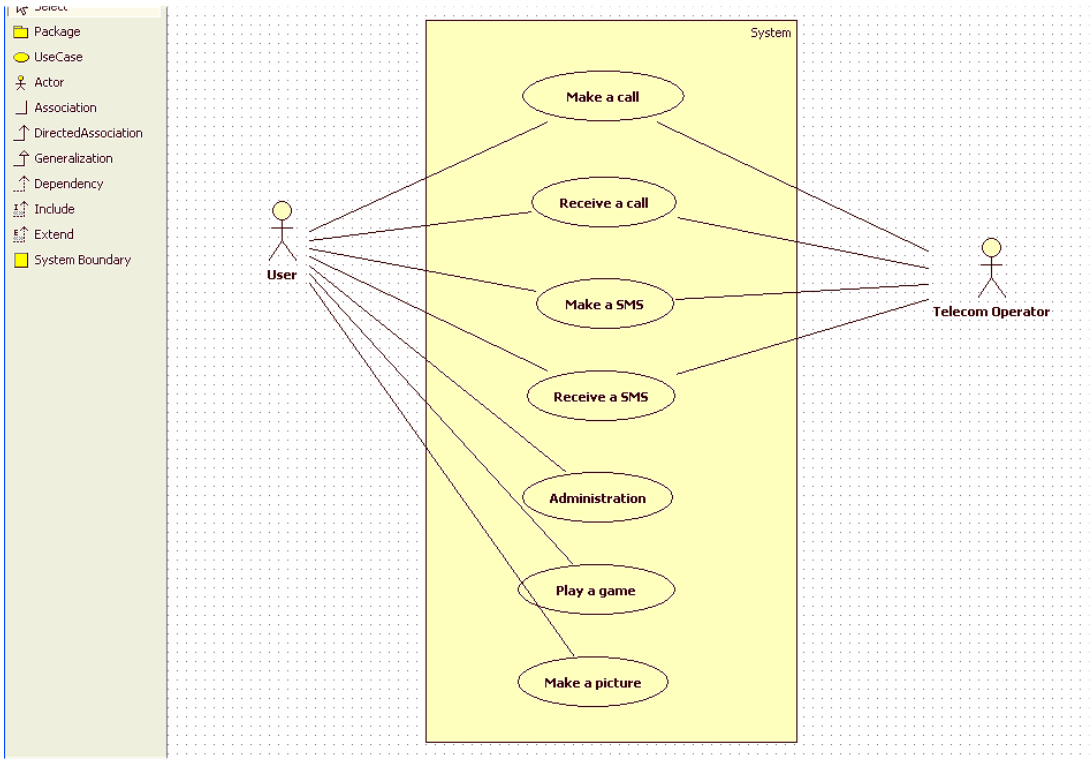


Figure 6.5: The use case diagram for a standard mobile phone.

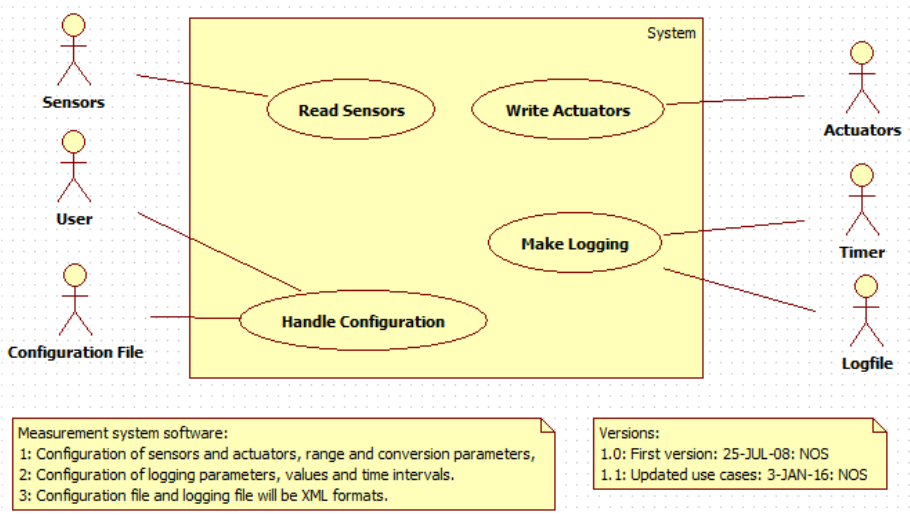


Figure 6.6: The use case diagram for the application using the I/O measurement device.

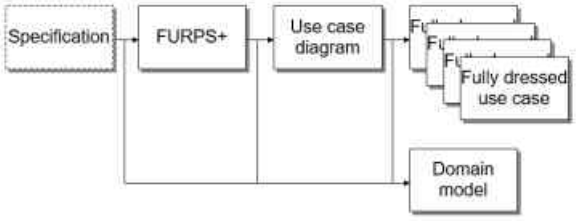


Figure 6.7: An overview of the steps and information needed for a making a domain model.

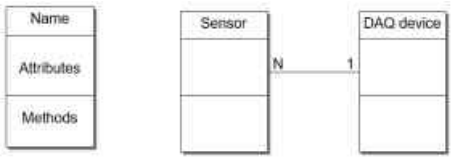


Figure 6.8: UML class diagram, to the left a single class, to the right two classes and an association between the classes.

One example of a use case diagram for the measurement system is shown in Figure 6.6.

Exercise 7 *Make the use case diagram for the software of a washing machine.*

The washing mashine contains two pumps, a motor and a user interface with buttons and a display. The operations could be prewash, washing at 40 °C and 60 °C, rinsing and drying. An example of a use case diagram is shown in Figure 6.4.

Note that there exists no perfect use case diagram, a use case diagram is a visualization of the requirements and your understaning of the system. However, a use case diagram can be wrong, bad or good. It is however important that the use case diagram reflect the requirements, and normally more requirements will be added/included while making the use case diagram because you will gain more knowledge about your system, just remember to update the requirements list or document.

6.4 Domain model

6.4.1 Introduction

The domain model is the first approach towards OOAD, requirements and use cases are not specific for OOAD. Requirements are necessary for all type of software development, and use cases describe the functional requirements in more details. Object-Oriented development will focus on making classes and objects. Objects are modules with responsibilities for fulfiling mainly the functional requirements in the software. In this first step the knowledge from collecting the requirements is used to make a conceptual model, named the domain model. The steps toward the domain model is shown in Figure 6.7.

The conceptual model will be based on the business model with focus on the static information in the system. It is important to notice that the domain model shall visualize the "real" world, NOT the software solution. The domain model is using the UML class diagrams, as shown in Figure 6.8, to visualize the static information in the system, and associations (connection between the classes) to indicate how the static information is linked. The Figure shows a single class to the left, and two classes interconnected with an association to the right. The association can include numerical relationships, in this case that a specific sensor device can only be connected to one specific DAQ device (Bentley 2005), while several a specific DAQ device can have several sensors connected.

Data acquisition (DAQ) device is a measurement device that converts the samples from sensor devices into digital numeric values that can be used by the software. Data acquisition systems (DAQ) includes

signal conditioning modules for handling analog and digital input and outputs. The domain model will represent the conceptual classes in the problem domian with focus on the "real" world, not the software solution. The information that can be used for making the domain model is shown in Figure 6.9, and use as many conceptual classes as possible.

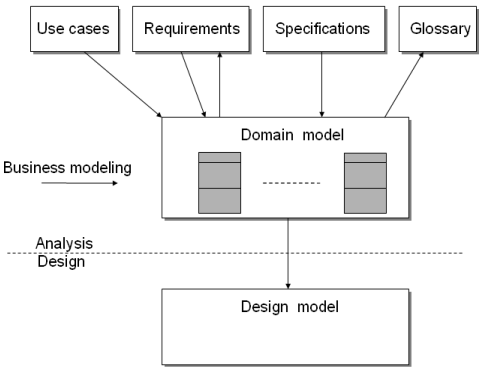


Figure 6.9: The background and reason for making the domain model.

6.4.2 Find conceptual/"functional" classes

The domain model consists of the conceptual classes or the "functional" classes in the system. Do not focus on how the software should solve the problem/challange, focus on the static "information" in the system. The steps for finding these classes are:

1. Identify noun (substantive) phrases ("things"); Check all the nouns in the specification and requirements,
2. Using the category list; physical devices or objects, specification, description, transactions, role, container, a device in a container, other systems, organizations, events, processes, rules, catalog, records, handbooks, manuals, a place to find more static information.

6.4.3 Usage of a domain model

The reasons for making a domain model is to:

1. get a better overview of the problem domain by *a*) using the information from the specification, *b*) using the information from the requirements, *c*) using the information from the use case diagram.
2. start thinking classes and start thinking responsibilities of these classes,
3. start thinking of the inheritance of information (how can the information be reused/combined),
4. start thinking of the interactions between the classes. Use associations to show these interactions. An association in a domain model is defined by;
   (a) a relationship between classes, indicate a meaningful and interesting connection,
   (b) a capitalized association name,
   (c) a multiplicity expression at the ends, indicating the numerical relationship between instances of the classes.
      i. 1, 0..1, *, 1..*, 1..n, n
      ii. see Figure 6.10 where the mobile phone has one DAdrList (AddressList) and many DAdrId (AddressIndex), from 0 to many.

6.4.4 Making a domain model

The steps in making a domain model are:

1. Find the conceptual classes,
2. Include these classes in a domain model diagram using the UML. Domain model is made as a class diagram. Figure 6.10 shows an example of a domain model,
3. Make relationship between the class diagrams (associations; connection and an optional description),

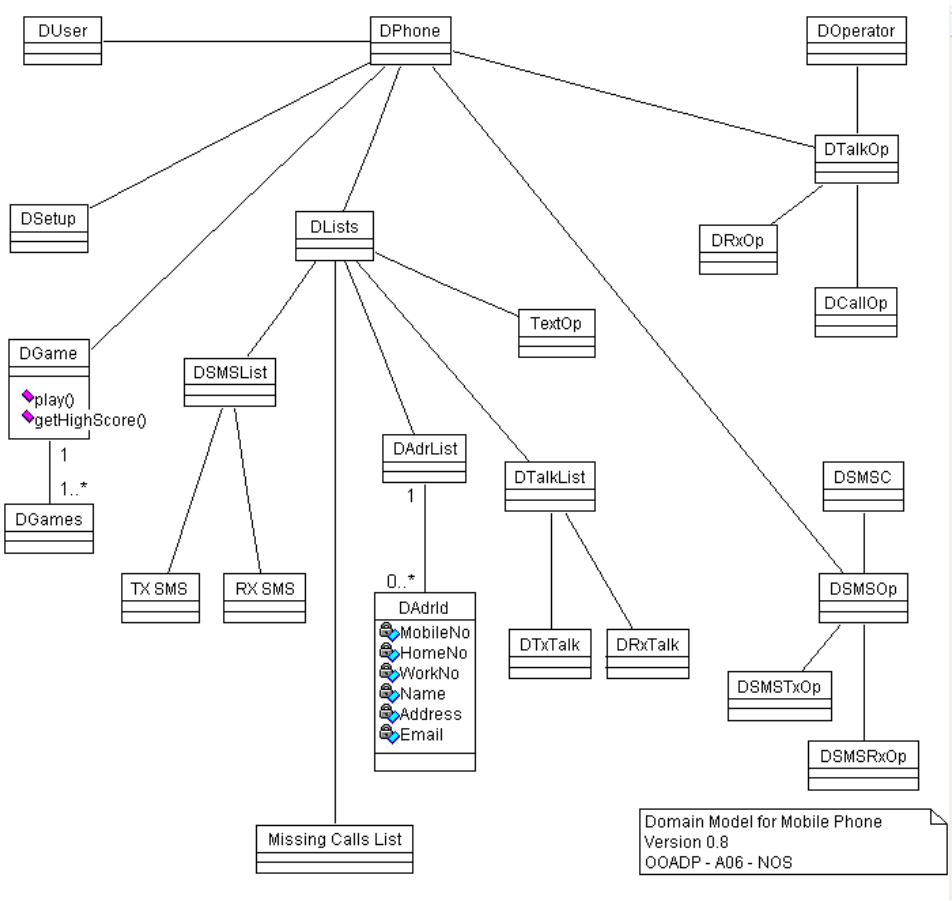


Figure 6.10: An example of a domain model for the mobile phone. All classes starts with the letter D to seperate from the classes in the class diagram model.

4. Add important attributes to the class diagram. NOTE: If the attribute is not a number or a text string, make a new class instead.

IMPORTANT: The domain model is not right or wrong, but can be good or bad. It is however an important tool in the first step of the Object-Oriented Analysis (OOA). Figure 6.10 shows an example of a domain model for a mobile phone, with focus on the static information. The main focus will be on lists holding the information like address lists, list sending and receiving SMS (Short Message Service), lists for calls and so one.

The classes in Figure 6.10 starts with the letter D to indicate a Domain class as the UML tool used (Rational Rose) will not allow classes with the same name along the different software models like the domain model and class diagram model.

Exercise 8 *Make a domain model of the I/O measurement system.*

Exercise 9 *Make a domain model of the washing machine.*

6.5 Analysis steps of the development process

The analysis steps are the first step in each iteration in the unified development process. However, this step is diferent in the Elaboration phase and in the Construction phases. In the first development iteration, in the elaboration phase, the use case diagram and the domain model diagram are created. There will be only one use case diagram and only one domain model diagram for the system, however these diagrams may normally be extended while working trough the first iterations.

The unified development process is use case driven, meaning that one (or a few) use cases are selected for each iteration in the construction phases. Selecting a specific use case, the most important (and/or most difficult) use case in an iteration, an analysis operation must be performed on this use case as well.

The analysis operations are making 1) a use case document to describe the use case in more details, with focus on the events between the actors and the system (use case), and 2) a system sequence diagram (SSD) to show the sequence of these events in time.

6.6 Use Case Document

A use case document is describing the functional and behavioral requirements of the system, but specific for that use case only. This step is to analyize this specific use case in more detail. There exists at least three different formats:

1. brief (often included in the use case diagram, to get the overview),
2. casual (as above, but not used that much),
3. fully dressed (detail description, used as a start of an iteration).

The advice is to use a fully dressed use case document as this will give a much better understanding of the use case. A fully dressed use case document consists of 13 sections. There exists several templates for making these documents. The structure of such a document is:

	Use case section	Comment	Meaning
1	Use case name	Start with a verb	
2	Scope	The system under design	System / Business
3	Level	"user goal" or "sub function"	
4	Primary actor	The main user of the function	
5	Stakeholders and Interests	Who cares, and what do they want	Short description
6	Preconditions	What must be fulfil before starting	
7	Success Guarantee	What must be fulfiled at a successful completion	
8	**Main success scenario**	A typical set of events	Basic flow
9	**Extensions**	Alternative scenarios of success or failure	Conditional/Branch
10	Special requirements	Related non-functional requirements	
11	Technology list	Different I/O methods and data formats	
12	Frequency of occurrence	Investigation, testing and timing of implementation	Trigger
13	Miscellaneous	Such as open issues	

The advice is to use a document with two columns, one column for the name of the use case section and one column containing the description. The appendix shows an example of a template for this document. You can make your own tempate if this docuemnt do not give you a good understanding of the information.

The important sections are 8, 9 and 13. Section 8 contains a description of the basic flow for this use case. The basic flow means the normal operation, without any errors. Focus on the events between the actors and the use case, do NOT start to think how to program the use case. In section 9, take every step in section 8 and check if there can be an error and indicate how to solve the error. Section 13 indicates how often the use case should be triggered. Do not write continous or as often as possible, use the non-function requirement to indicate the timing. Section 8 should be as detailed as possible, include enough information that it is possible to understand how to solve any calculation or algorithms. Section 8 can be from halv a page to a couple of pages. Focus on the system having responsibilities, and how to get the information to be able to perform the responsibilities.

6.7 System Sequence Diagram (SSD)

6.7.1 Introduction

A business model is a method of describing the purpose, strategies, infrastructure, organization, and/or the operation for a business. In software engineering will the business model shows the core functions of the software, the knowledge, experience, and practice build into the software. In analysing a software

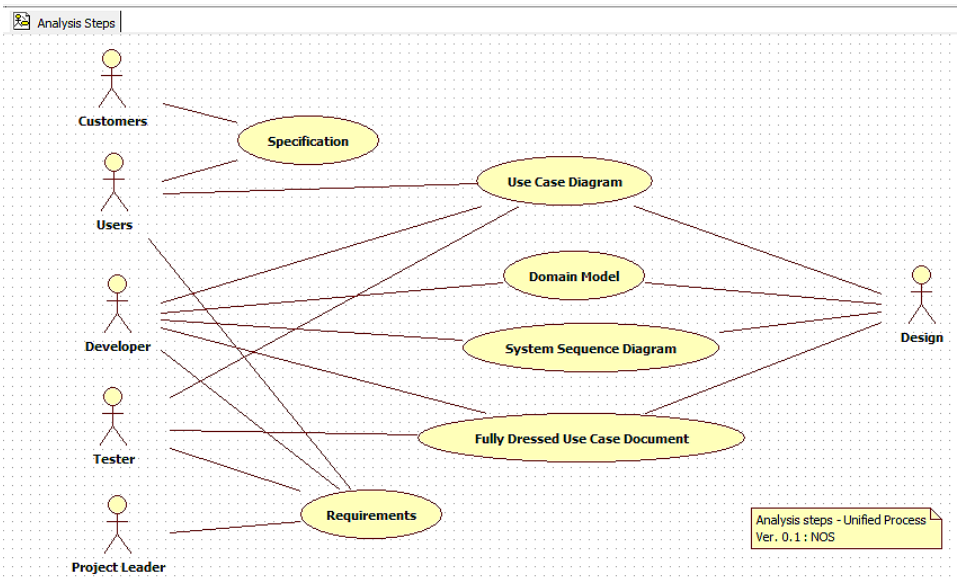


Figure 6.11: An use case diagram showing the the activities for the analysis steps. Several actors as input events to make the documents and knowledge for the design steps.

system can the domain model diagram and the system sequence diagram be used for the static and dynamic information for the business model as:

1. Domain model describes the static functions and informations of the business model,
2. System Sequence Diagram describes the dynamic functions of the business model.

The structure of analyzing the software is shown as a use case diagram in Figure 6.11
where the first steps are collecting the requirements based on the users and the specifications. The specification can be more informal and contains wishes, while the requirements should be testable. The use case diagram shows the behavior of the analysis steps, but no information about the correct sequence for making the document.

The informations in the fully dressed use case document (FDUCD) should be the starting point for making a SSD. Based on the FDUCD one SSD should be made for each of the use cases in the use case diagram.

6.7.2 Information in a System Sequence Diagram (SSD)

The FDUCD is a text document, while the SSD is an UML diagram. A SSD will focus on:

1. the behavior of the system as a “black-box”, and focus on only one use case at the time,
2. describing what the system shall do, not how to do it !
3. describing the events from the actors to the system, and the events from the system to the actors.

The SSD will focus on the time sequences for the behavior of the system and shows any operations as an actor is expecting from the system. The naming of the SSD should be a verb as the SSD is modeling the behavior of the system. SSD will also be part of the use case model. Some hints in making the SSD:

1. Add the main points from the use case as notes in SSD,
2. Use the same name as the use case,
3. Focus on section #8 and #9 from the FDUCD,
4. Combine the sections #8 and #9 in the same SSD.

The SSD will contain information about how the “Main Success Scenario” and ’Extensions” are combined in time. In practice this means that the developer should be able to think of all possible scenarios before start coding, both valid and invalid scenarios. This means that any error scenarios also will be analyzed before start designing the system.

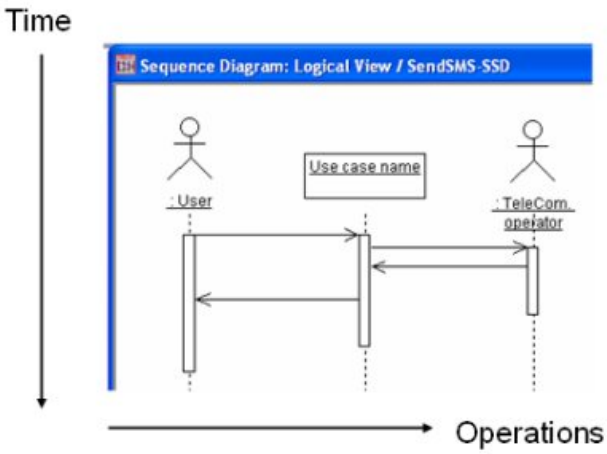


Figure 6.12: A general System Sequence Diagram (SSD).

6.7.3 How to make a SSD

The SSD is an UML diagram, select a sequence diagram to make a SSD. The sequence diagram must contain both actors and objects, and try to minimize the number of objects as this is a System Sequence Diagram. The focus should be the system, use only one or a few number of objects to describe your system. The SSD should contain the sequence of events (messages) between the system and the actors.

The contents of the SSD should be:

1. An object module being the use case, use the name of the uses case,
2. The actors as input and output to the use case,
3. Messages as described in the Fully Dressed Use Case Document (FDUCD), section 8 and 9,
4. Use the Combined Fragment block to indicate any alternative message blocks as defined in section 9,
5. Use the Combined Fragment block to indicate any loops defined with a goto structure in the FDUCD.

The Combined Fragment block can describe variations in the sequence of messages, and is defined as:

- Opt; Optional message sequence, use the guard to define the condition,
- Alt; Alternative message sequence, only one sequence occurs on any occasion,
- Loop; The fragment repeats a specific number of times,
- Break; abandone the rest of sequence if this fragment is executed, use the guard to indicate the break condition,
- Par; parallel sequence of messages,

to mention some of the fragment block types.

6.7.4 Some SSD examples

Figure 6.12 shows a general SSD with a use case with one input actor and one output actor. The input actor sens an event to the use case, the use case sends an event to the output actor, the output actor respons with an event back to the use case and the use case respons with an event back to the input actor.

Figure 6.13 shows the SSD for the SendSMS use case. The SSD shows the events between the user and the system when the use case SendSMS is active. In this case is the system the SendSMS use case.

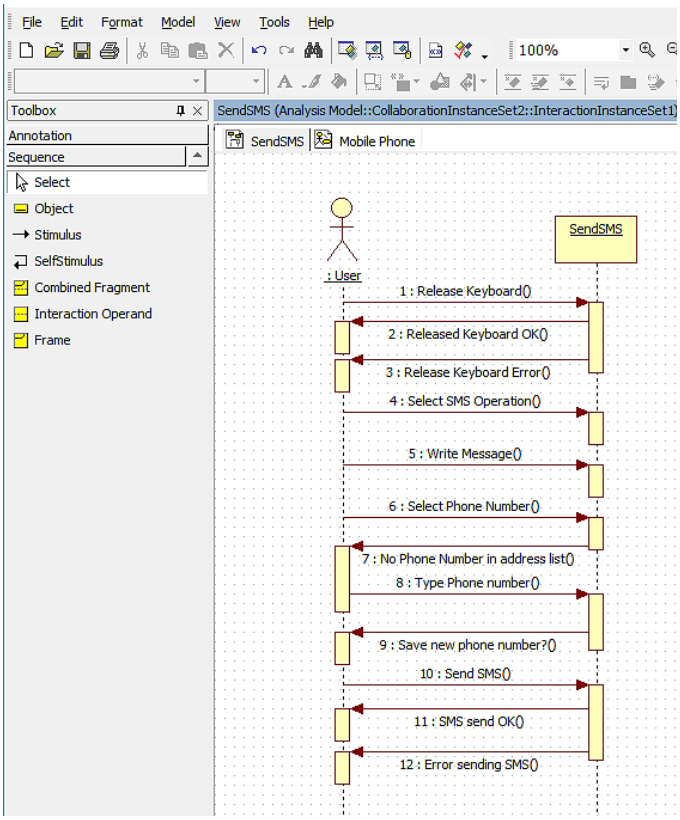


Figure 6.13: A System Sequence Diagram for sending a SMS on the mobile phone.

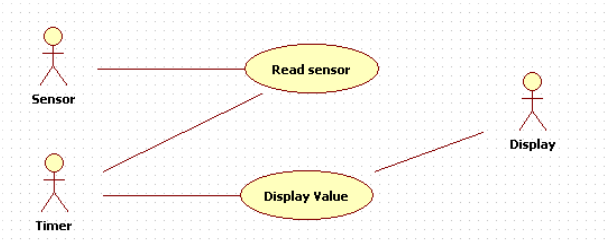


Figure 6.14: A simple monitoring system consisting of sensors in a process and a display showing the values from this process.

After finishing the SSD you should have a better overview of the function for the use case, the sequence of operations and possible error situation. This is the analysis phase of the system, you should now know enough about this use case that you can start designing the software for handling this use case. Also think about the development process, UP, focusing on one or a few use cases in each iteration.

The use case diagram for a simple monitoring system is shown in Figure 6.14. The corresponding SSD diagram for the *ReadSensor* use case is shown in Figure 6.15.

This Figure shows the function of the use in time, read the sensor value, wait, and then read the sensor again. The wait operation is used to allow the computer system to other tasks as well, no need to read the sensor continuous. The wait operation also controls the sampling time of the sensor system. Note that the wait operation is a service of the operation system either the scheduler or a timer event. The loop frame indicates a continuous loop, meaning that this use case will only read the sensor value, wait for some time (from milliseconds to seconds or minutes), and then read the sensor again.

In measurement systems there will normally be many sensors, and also an interest for a service option having the raw values from the sensors available in a service or debug window. The SSD can be extended with calculation of the raw value of the sensor and selection of the next id. The sensor read value will be an electrical signal, normally volt $(0 - 5V)$ or current $(4 - 20mA)$, and must be converted to the right unit for each sensor. This unit conversion is the calculation of the raw sensor value. Any low pass

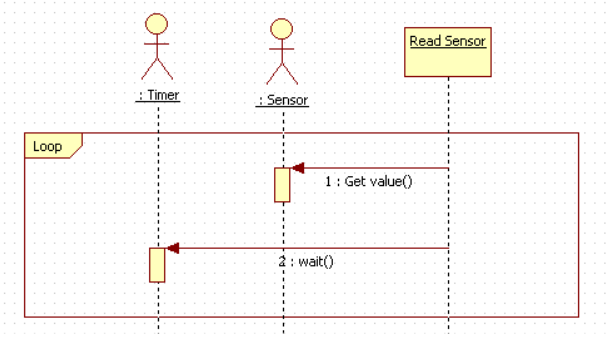


Figure 6.15: the SSD for the ReadSensor use case of the monitoring system.

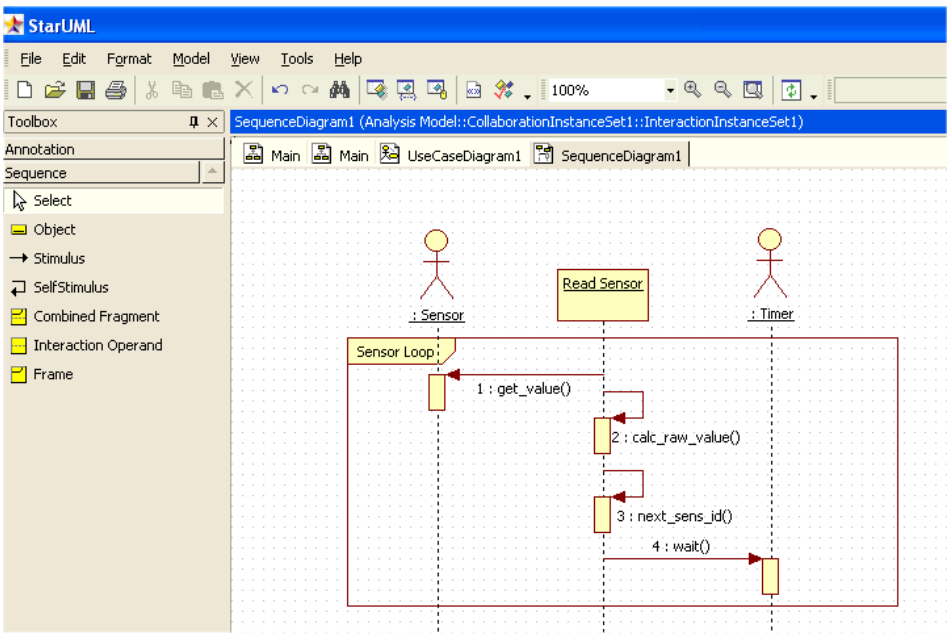


Figure 6.16: The SSD for reading a set of sensor devices, and calculating the raw unit value of the sensor devices. Is this a good SSD?

filtering should be performed using the unit converted signal from the sensor device, and all input signals into a computer system should be filtered in some way.

Figure 6.16 shows the SSD with an internal calulation and selection of several sensor devices. This should be a SSD, System Sequence Diagram, do NOT include too much of the internal logic of the use case. In this case there will be some programming details, however will the focus be to show the handling of several sensor devices without any much details. It may be a better solution to avoid the calc_raw_value() activity as this will be more a design part. The SSD shows that the use case "Read Sensor" will handle all the sensors in the system, or connected to one DAQ device, will read one and one sensor, and wait some time between reading each sensor devices. Some DAQ devices let you read all the sensors in one operation, however this will not be the solution here. This means that the type of DAQ device(s) must also be defined in the analysis phase so you know how to analyse the system. The DAQ device must be selected in the first iteration when deciding the system architecture. The calculation of the unit conversion and any filtering should be part of the FDUCD, state any equations for conversion or estimation in the FDUCD. Figure 6.16 is also missing any error handling, what should the Read Sensor use case do in case of no sensor connected, or if the sensor is broken? Based on information from the DAQ device and the valid range of the sensor is should be possible to verify the signal from the sensor, and an invalid signal from the sensor should be an alarm in the system. As reading sensor values include noise one single reading can be wrong, which should be handled by a filter, but if two or three following readings are wrong, indicate an error.

6.8 Operation Contracts

An operation contract can be useful in the analysis phase when there is a complex system behavior and a FDUCD is not used. An operation contract will focus only on section #7, #8 and #9 from the FDUCD and is used as an appendix to the SSD if none, a brief or casual use case document is used. An operation contracts is:

1. part of the use case model,
2. is used when a more detailed or precise description of the system behavior is necessary (in case of no FDUCD),
3. basis is the SSD, and define the effect on an object in the domain model,
4. look upon your system as a "black box",
5. describes the responsibility of the operation,
6. write in past tense,
7. normally no need for an operation contract!

An operation contract consists of several text fields with reference to a use case, a more simplified FDUCD.. The contents are:

1. Operation: name of the operations, and the parameters (if any),
2. Cross References: Name of the use case
3. Preconditions: State of the system before the operation,
4. Post conditions: State of the system after the operation.

An example of an operation contract:

Operation	SendSMS
Cross References	Make a SMS
Preconditions	The phone is turned ON
	Keyboard lock is ON
	Contact with SMS server
Post conditions	The message was sent
	The SMS sent list was updated
	Keyboard was locked ?

6.9 OOA Summary

The object oriented analysis is to get an overview of the problem domain for which the software system is going to work. The analysis will be to:

1. collect the requirements for the software system,
2. make an use case diagram; defining the main functions of the software,
3. make a domain model describing the static information in the problem domain,
4. make a text document for each use case; in this course a fully dressed used case document (FDUCD),
5. make a system sequence diagram for each of the use case; describing the dynamic information between the software system and the actors.

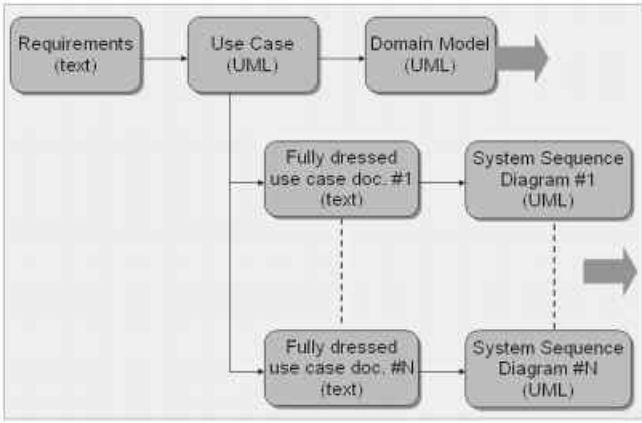


Figure 6.17: The documents and the diagrams used in the analysis phase of developing software.

Note that the software development process will define the sequence of making these documents. In the UP will the requirements and the use case diagram be the first task. The next task will be to make the domain model, based on the specification, requirements and the use case diagram. In each iteration a few number of the use cases will be analysed. The inception iteration will contain the startup of the project and making some sort of specifications. Elaboration iteration(s) will be the project timeschedule, collect the requirements, make the use case diagram, and the domain model. The construction phases will be any use cases, the most important first (highest risk?). A use case analysis includes making the fully dressed use case document and the system sequence diagram for this use case. Figure 6.17 shows the relationship between the documents and diagrams in the analyses phase.

The main goal of the analysis phase is to understand the problem domain, understand behavior of the system, understand the type and amount of information/events between the environment and the system that must be handled by the software. A goal is also to understand the main type of behaviors, any deviations of these behaviors, and any error situations.

The chapter have looked into the process of doing the analysis, based on a set of documents. Some or these documents are pure text documents, some other are UML documents. Let us look into some the UML tools that can be used to create these UML diagrams.

Chapter 7

UML diagrams

The UML diagrams can be drawn by hand, or using any graphical software package. Using an UML tool, all the symbols are already present, the type of symbols in each diagram is present only for the specific diagram, and the relationship between the symbols are easy to include. In this document the following tools are used:

1. Visual Studio, the IDE from Microsoft can be used for designing some of the UML diagrams. Note that only a few versions contain the Architecture menu that can be used to create UML diagrams. Newer versions (from 2019) do not contain the Architecture menu. The focus in Visual Studio is design, not analysis.

2. StarUML, an open source tool that can be downloaded from internet for free. This tools also contains an option to generate C# code from your UML models. StarUML contains of two versions, the first version was open source and free, the newer version is still open source but a license fee is requested. Most of the diagrams in this compendium is created with the free version of StarUML. See staruml.sourceforge.net/en/,

3. Rational Rose, the tools made by Rational, the firm of the "three amigos". The Rational Rose tool has for years been the reference of the UML tools. This version is an old version of this tool, free, but with a set of limitations.

This chapter will give an introduction to the usage of these UML tools showing the creation of an use case diagram, a class diagram, and a sequence diagram. Creating a domain model is using the class diagram, and creating an object model is also using the class diagram.

7.1 Visual Studio

Some versions of Visual Studio can also be used for making some of the UML diagrams used in this document. Only Visual Studio with the Architecture module can be used, containing an *Architecture* button as shown in Figure 7.1, however this option was removed from Visual Studio 2017.

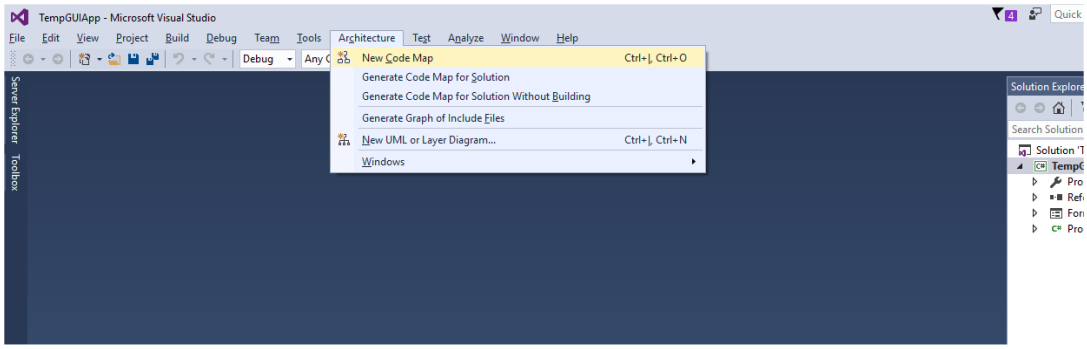


Figure 7.1: The main menu bar in Visual Studio 2015.

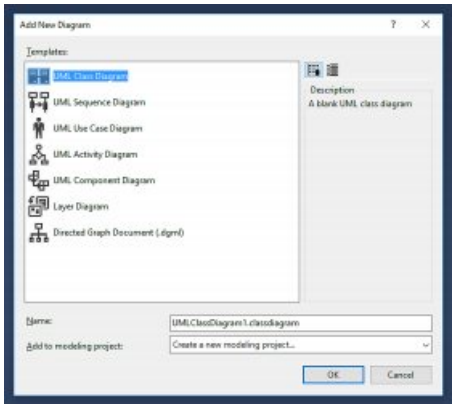


Figure 7.2: The available UML diagrams in Visual Studio Enterprise 2015.

Select *New UML or Layer diagrams* to get available UML diagrams in your version. The menu for selecting the UML digram can be like Figure 7.2.

Adding a loop or an extension in the sequence diagram, select a message, right-click and select a "Surround with" type.

7.2 StarUML

The startup window of the StarUML application is shown in Figure 7.3. Choose the Default Approach to start with, or select the Recent Files tab folder to select an already created project.

After selecting the startup approach the main windows of the application is shown in Figure 7.4. The Model Explorer window is to the upper right, use this explorer to select the type of operation and diagram. The operations are use case model, analysis, design, implementation, and deployment. We will focus on the use case model, analysis, and design in this course.

The use case diagram is part of the use case model, the domain model and the SSD is part of the analysis model.

Use the main diagram in the use case model to make your use case diagram. At the left side will now the toolbox for the use case diagram be present, separated in the UseCase and the Annotation. On the lower right side is the property window, showing the property of the selected component in the model.

A new diagram can be created by rightclick on the operations in the Model explorer, a list of possible diagrams should be present where you can select the type of diagram you want. You can also change the name of diagram to fit your model.

A complete use case model for a software application is shown in Figure 7.5. This figure shows a combination of use cases, actors, and annotations.

The use case diagram should contain about 3 to 9 use cases, if you need more use cases you should separate the use case model in different use case diagrams using a hierarchical structure of the diagrams.

7.2.1 Use case diagram

Select the use case model in the Model Explorer menu, shown in Figure 7.7. Use the Main diagram or make a new Use Case diagram if you want to use antoher name.

A new digram is made by right clicking on the model in the Model explorer, the menues are as shown in Figure 7.7. Select the type of diagram that you want to add, and fill in the name fo the diagram.

When selecting an use case diagram, will the toolbox on the left side change to show the items that can bu used in such a diagram. Figure 7.5 shows the items for the use case diagram like the use case, the actor, and the association. Click on the item and click in the diagram to plase the items. Add a suitable name if necessary. The items can be dragged to any position in the diagram. An association can be placed by cliking on the association in the toolbox, click on the first item and keep the button pressed. Drag the association to the second item and release the button.

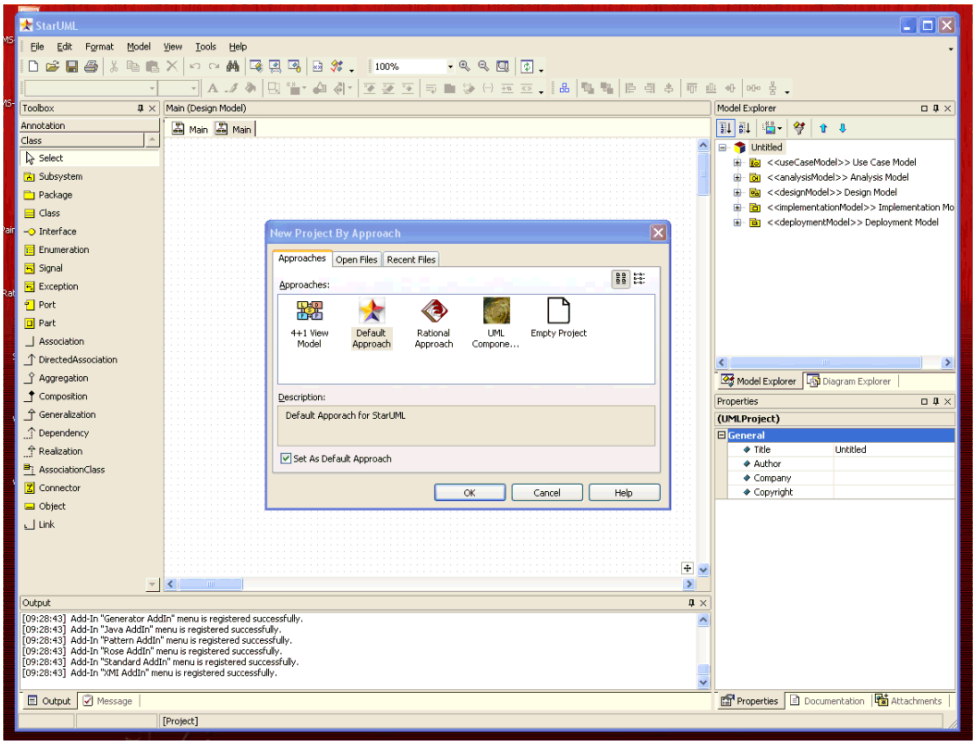


Figure 7.3: The startup window of the StarUML application.

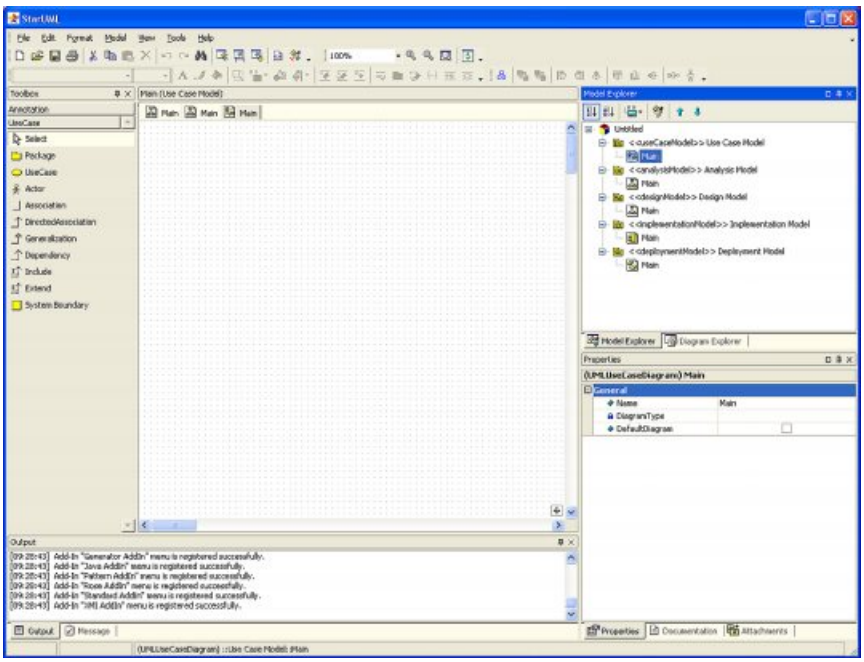


Figure 7.4: The StarUML application with the toolboxes to the left, the working area in the middle, status information at the bottom, a tree structure of the data in the right upper section (model explorer), and the property of the current data in the right lower section.

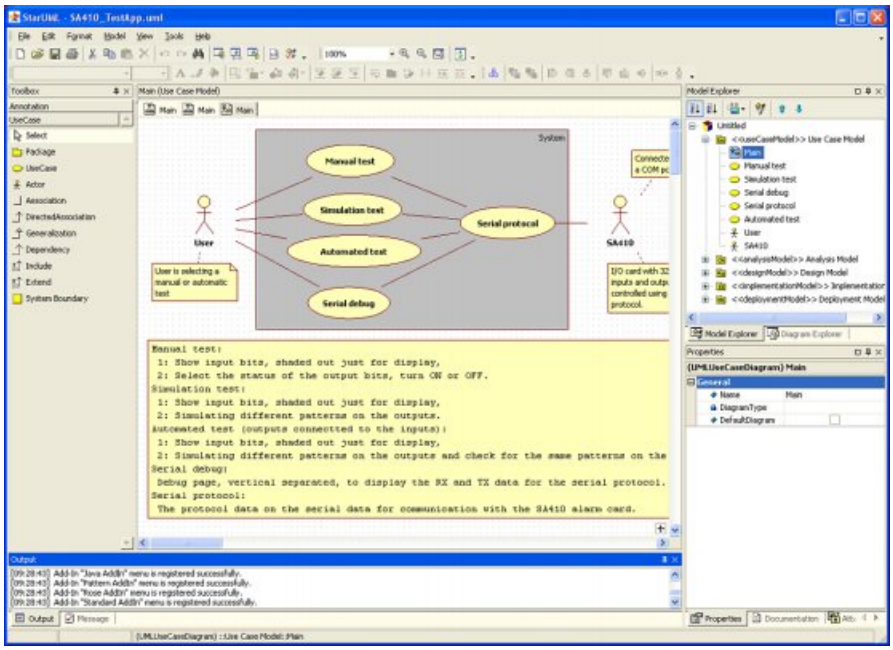


Figure 7.5: A use case diagram with use cases, actors and annotations.

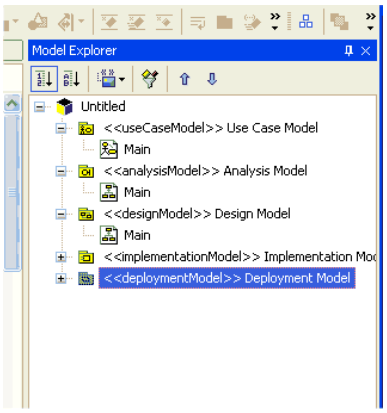


Figure 7.6: Model explorer menu on the right side of the StarUML application.

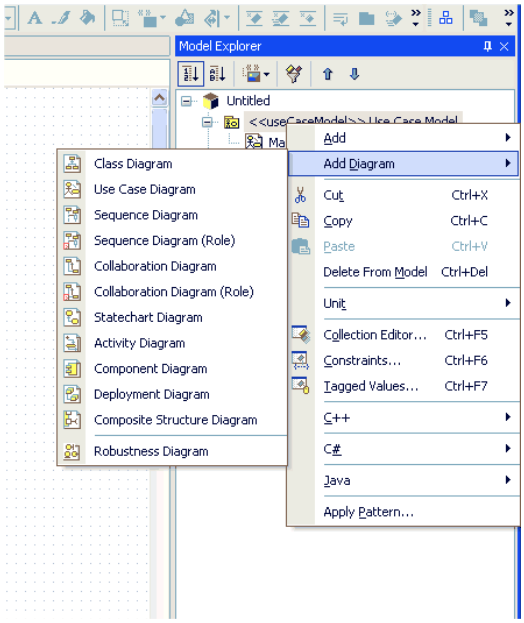


Figure 7.7: The menus for selecting a new diagram on a specific model. Right click on the model in the Model Explorer to get these menues.

7.2.2 Domain model

Select a class diagram from the menu in Figure 7.7 to make a domain model. Domain diagram will normally belong to the analysis model.

7.2.3 System Sequence Diagram

Select a sequence diagram from the menu in Figure 7.7 to make a system sequence diagram. System sequence diagram will normally belong to the analysis model.

7.2.4 Sequence Diagram

Select a sequence diagram from the menu in Figure 7.7 to make a sequence diagram. Sequence diagram will normally belong to the design model. In StarUML is normally the variable or variables in a method hidden, these variables can be visible in two ways. One general way for all sequence diagrams and one way for a specific sequence diagram.

The general way is to use the *Tools - Options* meny as shown in Figure 7.8. Set the *Message Signature* to *show name and type*. The StarUML application has to be restarted before this option is activated.

If you want to change this option for a specific diagram, click at the background of the diagram and use the Property window to change the *Message Signature*. This shown in Figure 7.9.

For a loop or any extension a combined fragment can be used, indicating the type of operation. In Visual Studio click on the message and select "Surround with" to include a combined fragment.

7.3 Rational Rose

Rational Rose has an old demo version that can be downloaded for testing, but it has the following limitations of maximum 30 classes and actors, 10 use cases, 10 states, 3 packages, and 10 modules. Starting the demo version will give a main screen like Figure 7.10. If a model is already saved, use the File menu to reload the model.

Starting from scratch, right click on the UseCaseView folder to get a menu for making a new use case, see Figure 7.11. Select a name of the use case in the folder list, and then double click on the name to load the use case.

Figure 7.12 shows a use case diagram. Click on the symbol, and the click inside the use case diagram to place the symbol.

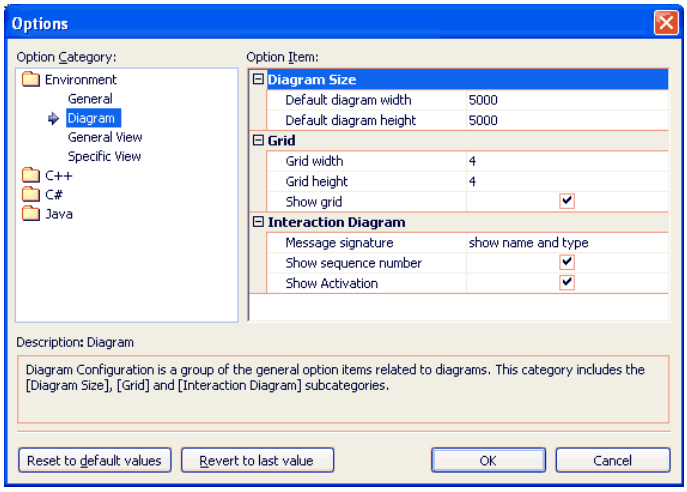


Figure 7.8: The TOOLS - OPTIONS menu to add the display of the input variable(s) to a method in the sequence diagram.

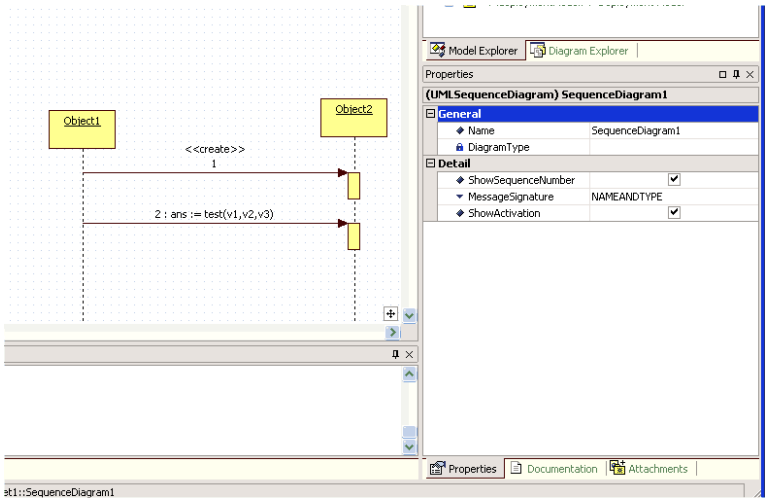


Figure 7.9: Change the MessageSignature for a specific Sequence Diagram.

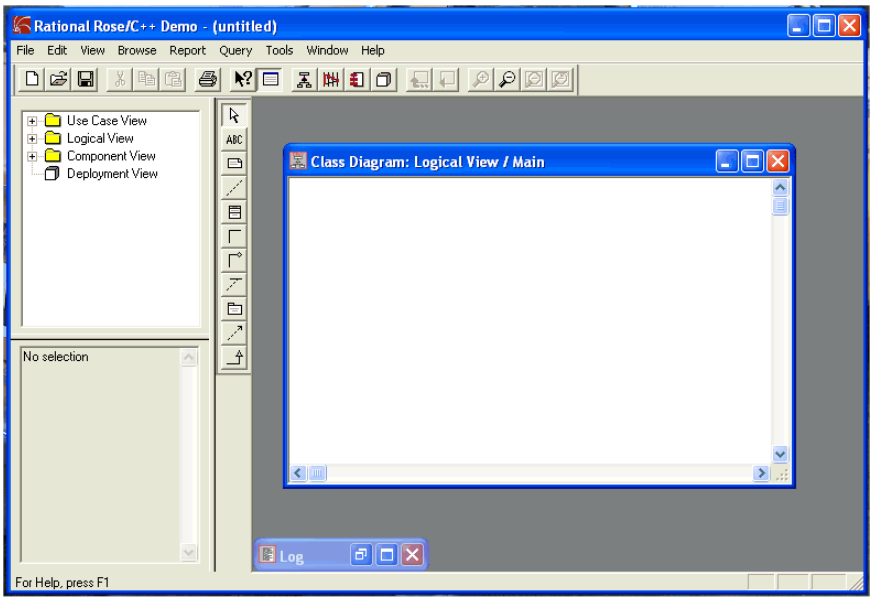


Figure 7.10: Main window after starting Rational Rose demo.

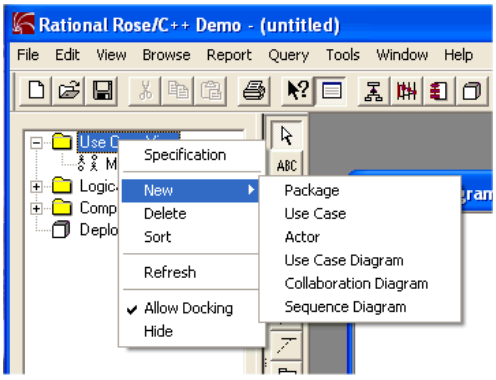


Figure 7.11: Right click the UseCase folder to get the menu making a new Use case Diagram.

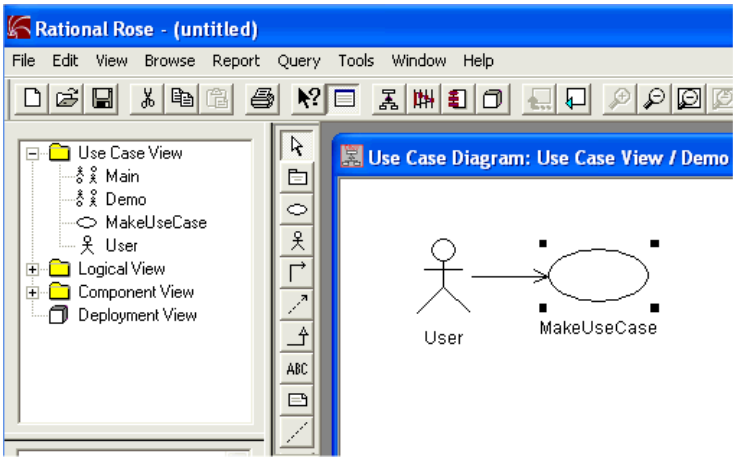


Figure 7.12: Making a use case diagram.

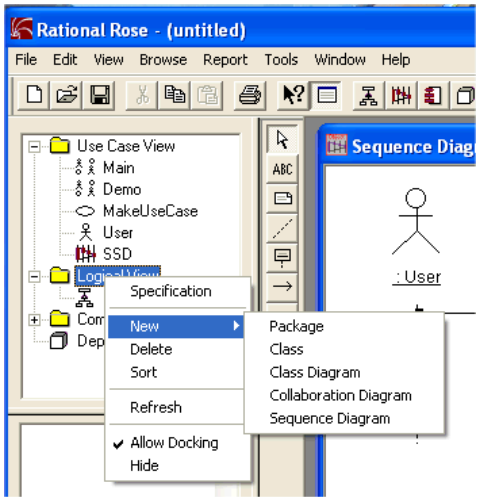


Figure 7.13: Use the Logical View folder to get the menu for making the Domain diagram (a class diagram).

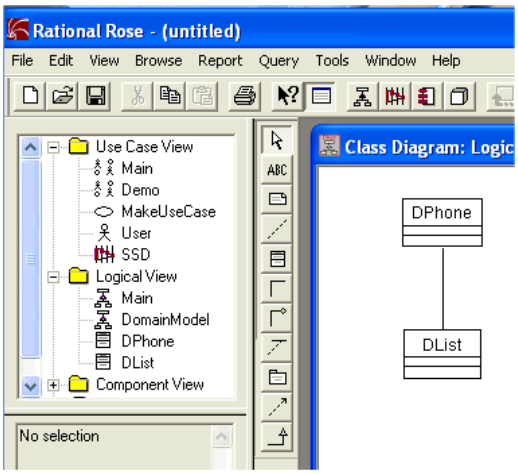


Figure 7.14: Use the class diagram for making the domain model.

As explained in class, the domain model was not used before, the name domain model is not present in this demo version. But the domain model is just a class diagram, use the LogicalView folder to make a class diagram as shown in Figure 7.13. Set the name of the domain model, and double click on the name to load the domain model.

The domain model is shown in Figure 7.14, and the class diagram and the association can be selected using the tool bar.

The System Sequence Diagram is a sequence diagram and can be made the same way as the use case diagram. Select the Sequence Diagram in Figure 7.11, and set the name. Then double click to load the SSD as shown in Figure 7.15. The symbols are present on the tool bar.

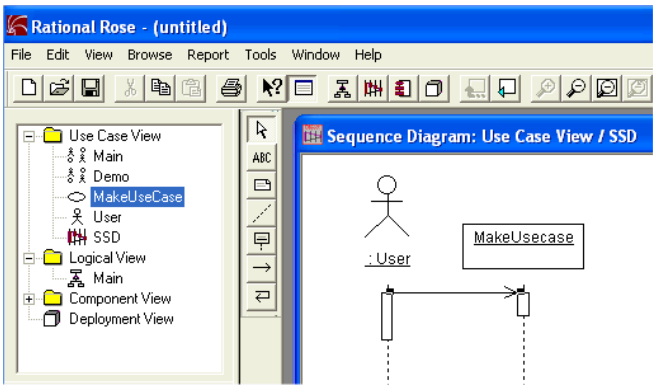


Figure 7.15: Use the UseCaseView folder to make the SSD.

Chapter 8

Design

8.1 Introduction

This iteration has focus on the analysis and design phases. The analysis phase has focused on collecting requirements, making a use case diagram, a use case document, a domain model and a system sequence diagram. The software development process, unified process (UP) has defined in which order these steps should be done and also the type and number of documents. It is important to make some documents for the analysis step of the software application. The knowledge, experience and documentation from the analysis step will be the basis for the the next step, the design step. The design step will focus on:

1. transfering the requirements into the design, but in several iterations. This is defined by the software development process,
2. designing the system (doing the "things" right - the design). In the analysis step the focus was on understanding the system (doing the right "things" - the work/process).

The result of the design step will be a design model with interaction diagrams, class diagram and object diagram. The interaction diagrams document the dynamic "behaviour" of the system, while the class and object diagrams document the static "behaviour" of the system. There will be one interaction diagram for each of the use cases, while there will be only one common class diagram and one common object diagram. The class diagram will show the abstract design model, while the the object diagram will show the physical implementation. NOTE that the important part of the design step is the object design of the software application, not how to make the UML diagrams!

Figure 8.1 shows an UML use case diagram for the functions of the analysis and design steps, and the diagrams used. The *Use case* section is to make the use case documents, one text document for each of the use cases in the use case diagram. The *System Sequence Diagram* section will contain one SSD for each of the use case.

HINT: This shows that an UML use case diagram can be an useful tool for describing any problem or challange in a more understandable way!

8.1.1 System Architecture

A software application is normally designed as a "layered" application consisting of several layers (or tiers) to separate the user interface part, the logic part, and the data handling part. Each part is separated by layers to make the application more easy to maintain, extend and/or modify. By using tiers or layers a separate layer can more easily be modified or replaced by another equivalent layer, like changing one of the layers, with minor changes of the other layers. Normally two, three, or four layers are used, Figure 8.2 shows the type and number of layers in a two, three, or four layer application.

The focus is object oriented design and each layer will contain a set of objects with functionality and responsibility regarding that layer. Figure 8.3 shows a three tier architecture application with several objects in each tier. In this application most of the functionality is in the business layer and the presentation layer, not so much in the data layer. If storing the data in a comma separated values (CSV) file, changing the storing into a SQL database will affect mainly the data layer.

In a distributed system can the different tiers be running on several computers affecting the server side and client side of the applications. In these cases thick or thin clients affect on which computer the

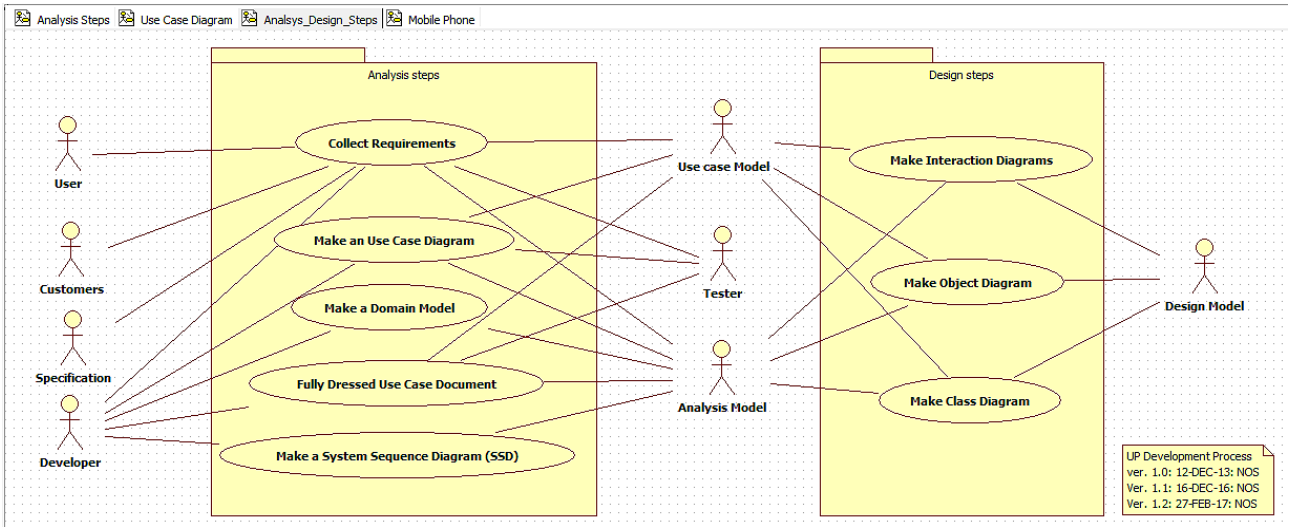


Figure 8.1: Using a UML use case diagram to describe the functions of the analysis and design steps.

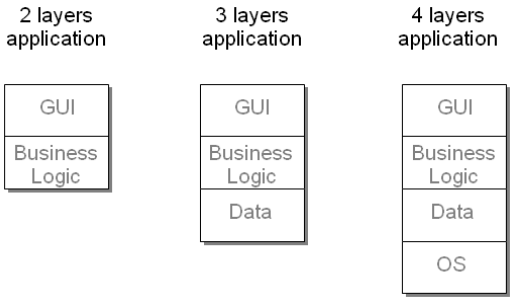


Figure 8.2: Application based on two, three or four layers (or tiers) using a tier architecture.

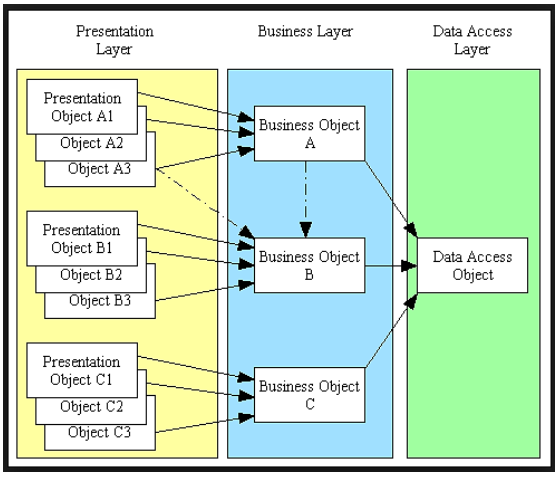


Figure 8.3: A three tier architecture with a set of objects in each tier. In this case most objects in the presentation layer, but the number of objects with depend on the type of functonality in each layer.

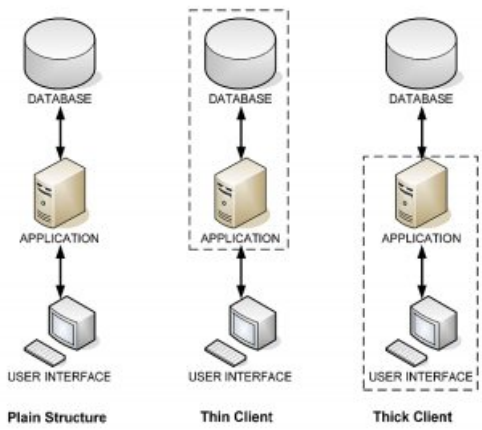


Figure 8.4: A three tier architecture application as a distributed server/client structure (Santos 2009).

tier will run. Figure 8.4 shows a distributed three tier architecture applications, based on server/client technology, with the different tiers running on different computers (platforms). The figure shows the difference of thin and thick clients indicating the number of tiers running on the client.

In this document the focus is the business layer (tier), also called the logic layer. This layer contains all logic that is specific for one type of business, and these logic should neither be in the GUI nor data layer. The first step in the object design should always be to define the architecture design, define the number of layers (or tiers) to use and start with this design. The number of layers should be part of the logical architecture analysis .

A system architect needs to have knowledge in several categories like hardware platforms (mainframes, distributed platforms, desktops and mobile devises), skills and understanding of a variety of operating systems, knowledge about several programming languages and a good understanding of databases. There exists several BOK like Software Engineering Body of Knowledge (SWEBoK), Enterprise Information Technology Body of Knowledge (EITBoK) and Systems Engineering Body of Knowledge (SEBoK) that can be useful knowledge for system architects. Body of knowledge (BOK) refers to the core teachings and skills required to work in a particular field or industry, and is normally defined by professional associations or societies.

8.2 Object design

The object design involves transformation of the knowledge from the analysis steps into software objects. Some questions will arise:

1. How much time on analysis? Using 2 to 4 weeks for the time boxed iteration, use only a few days on analysis and UML diagrams. Focus on design, implementation and testing,

2. Use paper or UML tools / IDE (Integrated Development Environment)? It is up to you, figure out was is working best for you, however very often a combination is good!

3. How detailed should the design be? The design should focus on the class and object structures, and the responsibilities for these classes. In the interaction diagrams the classes and the interaction between the classes will be defined. The interaction will be defined as messages, given the main methods in each class. The detailing level should be the classes and the main methods. An advice will be not to design the inner of methods or private methods as this will be very difficult to synchronize when start the implementing step. A detailed level of design will require an UML tool supporting reverse engineering, create or update UML diagrams from the code.

4. What about reverse engineering? This will depend on the detailing level. Including design of methods will normally require an UML tool that can support reverse-engineering. This means to be able to create or update the UML diagrams based on the coding, to keep the diagrams (documentation) and coding synchronized. The advice is to focus on the classes and the main methods in the design phase to avoid reverse engineering. **However, it is important that the**

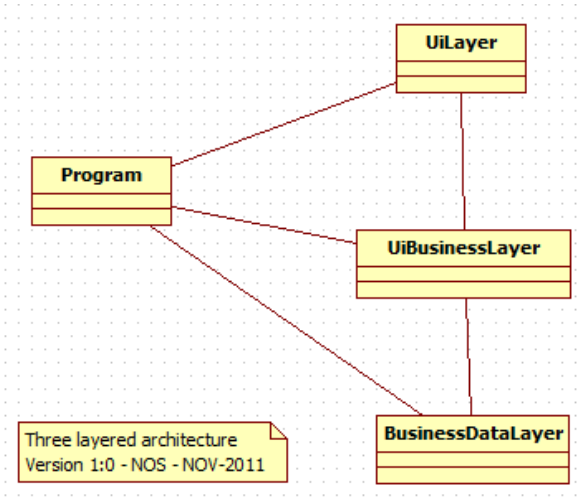


Figure 8.5: The first class diagran in the design model should be for the architecture design.

documentation and the code stays synchronized no matter of the type of methods that are used.

Designing a system should start with making a layered structure, to separate the main functions of the system. The selection of number of layers should be handled in the analyzing part. In software design a layer design architecture is used to separate the user functions, the core function of the application, often called the business layer and the handling of the data. In web design or multi-operating system (OS) design a separate layer for the functions of the operating systems can also be useful. This layer separation is named layer design or tier architecture, see Figure 8.2. The type of layer design or tier architecture can be:

1. 2-tier architecture: separate the user interface (GUI) and the business layers,
2. 3-tier architecture: separate the GUI, business and the data layers,
3. 4-tier architecture: separate the GUI, business, data and OS layers.

The most used layer design is the 3-tier architecture. The most important layer for any company will be the business layer (tier) as this layer will contain the logic and structure of any processes or functions for the business. Start with selecting the number of layers (tiers) for the application, and start with evaluating two layers. The data access will decide two or three layers, if using a database or different type of text files like csv files and log files, choose three layers. Supporting different OS evalute if three or four layers should be used, and choose four layers if supporting different type of OS like web and mobile platforms. Figure 8.5 shows a class diagram for a three layer application.

This figure shows the first class diagram with four classes, the program class (generated by Visual Studio) and three classes for the layers. The Program class is normally created by the IDE as the starting point for your application. In the startup method (main()) create the BusinessDataLayer object, then the UiBusinessLayer object including a reference to the BusinessDataLayer object, and finally the UILayer object including a reference to the UiBusinessLayer object. In graphical applications will the UiLayer object be the "running" object as this one is receiving and handling all the events. The UiLayer object will have an object reference to the UiBusinessLayer object and can access methods in the business layer by using this object. The UiBusinessLayer object has a reference to the BusinessDataLayer object and can access methods in the data layer by using this object.

8.2.1 Design Models

Designing the software application is named making design models of the application. The model is described by a set of documents. The development process defined the type of documents and the sequence of making the documents. The design step consists of several models, and two main type of design models exists - dynamic and static models. Some develpment processes starts with static model first, our UP development process starts with the dynamic model. The development process is use case driven, so the first step is to model the dynamic behaviour of a specific use case.

Dynamic design models

The dynamic design models focus on the behaviour of the system. In the design step the models should describe how the software can handle the system. The dynamic design model will be based on UML diagrams, UML interaction diagrams. These diagrams define and/or design the logic of the system and the behavior of the software code, the messages between the objects, the class methods.

Static design models

The static design models focus on the static information in the system, the "data". The static design models will also be based on UML diagram, the UML class diagram. This will be like the domain model diagram, however the class diagram will focus on the static structure of the software application. The static design model will consists of both the class and object diagrams, showing both the model and physical implementation. A class diagram consists of the following parts: 1) class names, 2) attributes, 3) methods, and 4) the associations between the classes. See Figure 6.8 for an example of an UML class diagram.

Advice: Develop the dynamic and static models in parallel, but spent most time on the dynamic models.

8.2.2 Class and object

The class contains an abstract definition of an object. The class is part of your model, the object is an instance of a class in memory. However, both the class diagram and the object diagram is designed using the UML class diagram, the class diagram will show the design model while the object diagram will show your running model in the computer memory.

A class consists of (optional):

- name,
- attributes;
  - private data (variables) that hold information about the object,
  - encapsulation means to have private data,
  - private data means that the data is not visible (accessable) outside the object. Use properties/methods to access the data.
- methods;
  - public methods that can be called by other objects,
  - private methods that can only be called by this object,
  - properties used to change or access the properties of the object (consists of get and set statements),
  - constructor, a method with the same name as the class, only called when the object is created.

Methods and properties are used to access the private data of the object, as it is a very bad practice to use public (globale) data. By using private data will any change of the data always be controlled by the inside of each object. The difference of a method and a property is that a method is used when the object should carry out some sort of action, while a property is used when accessing some type of information associated with the object. Figure 8.6 shows a class with private information and a method that are suppose to share that information. What kind of information is the method sharing?

An example of a class containing a method and a property:

```
//////////////////////////////////////////////////////////
public class SensorData
{
    //
    const int          MAX_IO_CH = 8 ;
    private short      iChannel ;
```

```
public class Girl {
   private int age = 28;
      public int getAge() {
         return 20;
      }
}
```

Figure 8.6: An example on how to keep information private and use a method for accessing the information. However in this case is the method is sharing the "wrong" information!

```
private short      iType ;
private int        iId ;
//
public SensorData(int id, short ch, short type)
//
// Purpose: constructor
// Version: 1.0: 12-DEC-13: NOS
{
  iChannel = ch ;     // Channel id
  iType = type ;      // Type of sensor unit
  iId = id ;
}
// Properties
//
public string Channel
//
// Purpose: get or set the channel number, as string
// Version: 1.0: 12-DEC-13: NOS
{
  get
  {
    return iChannel.ToString() ;
  }
  set
  {
    int    iVal ;

    iVal = Convert.ToInt(value) ;
    if (iVal >= 0 && iVal < MAX_IO_CH)
    {
        iChannel = (short) iVal ;
    }
  }
}
// Methods
//
public double GetValue()
//
// Purpose: get a sensor value from the daq device
// Version: 1.0: 12-DEC-13: NOS
{
    double dVal ;
```

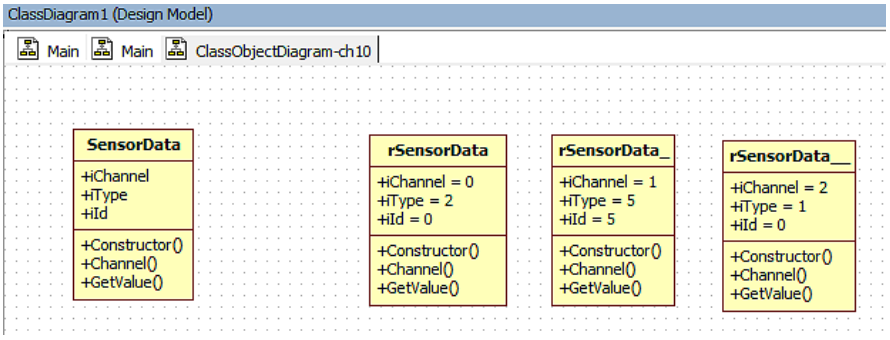


Figure 8.7: A class diagram to the left and an object diagram to the right. A class diagram contains only the abstract definitions, while the object diagram contains the physical implementation indicated with real values for the attributes. Note that the object names should be equal, only the object references are different. However, this is not possible in the StarUML tool.

```
        dVal = GetDaqValue(iChannel) ;              // DAQ signal
        dVal = ConvertValueType(dVal, iType) ;      // Unit value
        return dVal ;
    }
}
//////////////////////////// EOC ////////////////////////////
```

The class is the abstract model, while the object is the physical implementation (instance) of the class in memory when the application is running. In UML both class and object diagrams can be made, both using the class diagram as the starting point. Figure 8.7 shows the class diagram to the left, for C# code listed above. Note that the property is indicated as a method in this diagram. To the right is an object diagram with three objects of the same class, with different values of the attributes. The name of the objects should have been the same, but the UML tool restrict the use of the same name of classes.

8.2.3 CRC cards

A text-oriented modeling technique for assigning responsibility to an object, called Class Responsibility Collaboration cards (CRC). Define one document for each class, and the structure should be:

Class name	
Responsibility #1	Collaborator #1
Responsibility #2	
....	Collaborator #2
Responsibility #N	

where the "Responsibility #x" defines each "sub"responsiblities for a specific class. The collaborator column indicates other classes that will be used for fullfilling this "sub"responibility. An example for the responsibility of the SendSMS class for the mobile phone, where the class will get the text for the SMS message, the pohne number for the receiver of the SMS mesage, and then send the message.

SendSMS	
Get the text for the message	TextInputClass
Get the PhoneNumber	AddressClass
Send the message	SendClass

8.3 UML Interaction diagrams

UML interaction diagrams are part of the dynamic design model. This model will illustrate how objects interact via messages. UML interaction diagrams consist of sequence diagrams and collaboration

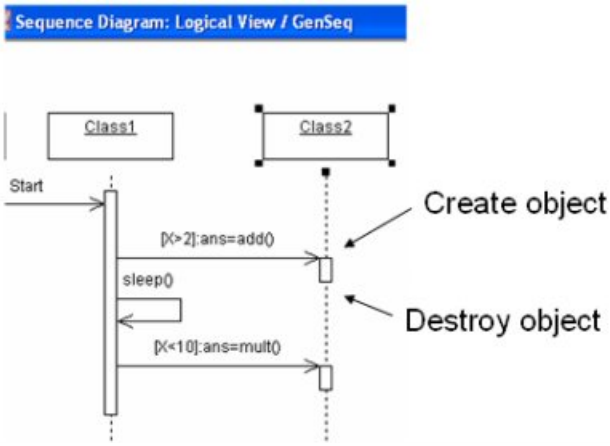


Figure 8.8: A general sequence diagram with two classes.

(communication) diagrams. A sequence diagram will, like the System Sequence Diagram (SSD), focus on time, while a collaboration diagram will focus on the information (data) in the system. A process control system application should be modeled by using a design sequence diagram while an office application should be modeled by a collaboration diagram. When making an interaction diagram you must select either a sequence diagram or a collaboration diagram. The UP process define one interaction diagram for each use case.

8.3.1 Sequence diagram

The sequence diagram is one type of interaction diagram and:

1. shows the sequence of messages in time (focus on time),
2. contains simple notation,
3. takes a lot of horizontal space,
4. shows no differences between class and object,
5. has the following type of messages:
   (a) Message with return parameter: $ans = message()$
   (b) Condition (1): $[x < 10] : ans = message()$ $[else] :$ $ans = message2()$
   (c) Condition (2): $[ok = true] :$ $ans = message()$
   (d) Loop: $*[i = 1..n] : ans = message()$

Figure 8.8 shows a sequence diagram with two classes. Class1 is created once, while class2 is created twice.

The StarUML application will as default not show the input variables of a message. See the Chapter about creating UML diagrams how to show the input variables in sequence diagram in StarUML.

Several fragments can be combined in a sequence diagram as several statements may belong to a condition block or a loop block. Use the *Combined Fragment* block and define the fragment to be alternatives (alt), options (opt) or loops. Use alternatives if the code consists of an if - else block, use options if the code consists of an if block, and use loops if the code consists of a loop. The combined fragment is used to group together the set of messages within the condition or loop, to show the conditional flow in the sequence diagram. Figure 8.9 shows the usage of the alternative fragment blocks for an *if - else* condition block.

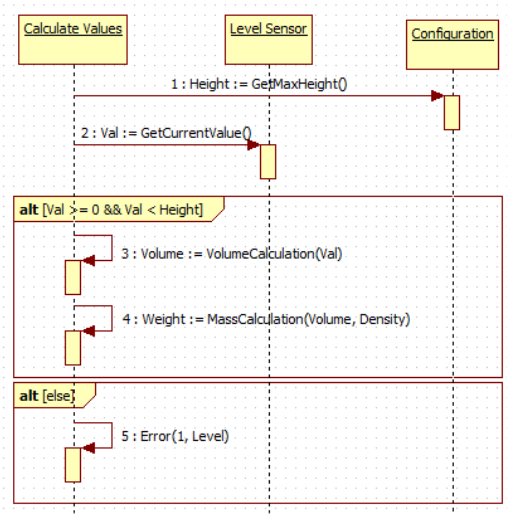


Figure 8.9: Design with combined fragments as alternative fragments for the *if* and *else* fragments.

8.3.2 Collaboration Diagram (Communication Diagram)

Collaboration diagram is one type of interaction diagram and:

1. shows the collaboration (focus on information),
2. it is easy to show parallel behavior and complex branches,
3. it is difficult to see the sequence of messages,
4. requires less space then the sequence diagram,
5. has complex notation,
6. shows different notation for class and objects,
   (a) Class
   (b) *:Object* (or underlined in older versions)
7. has the following type of messages:
   (a) Message with return parameter: $ans = message()$
   (b) Condition: $[x < 10] : ans = message()$ $[else] : ans = message2()$
   (c) Condition (2): $[ok = true] : ans = message()$
   (d) Loop: $*[i = 1..n] : ans = message()$

Figure 8.10 shows a collaboration diagram.

8.3.3 Summary

Sequence Diagram (SD) focus on the messages between the classes, no need to separate classes and objects. Collaboration Diagram (CD) focus on data and messages between classes and as only the object contains data CD must separate on classes and objects. An interaction diagram will then be either a Sequence Diagram or a Collaboration Diagram depending on if the time or the data is most important for the application.

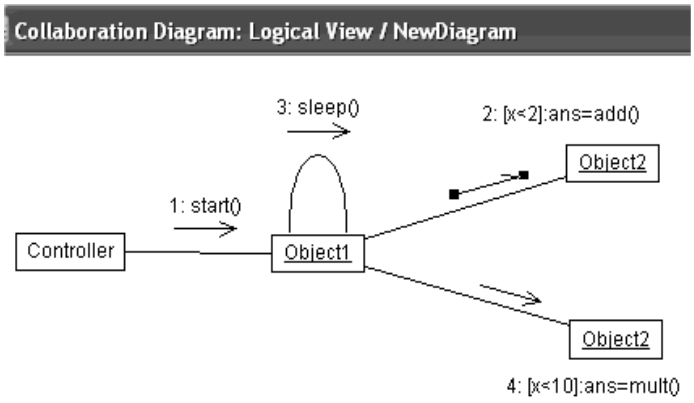


Figure 8.10: A general collaboration diagram (communication diagram).

8.3.4 Pseudo coding

Pseudo coding is an informal description of the basic steps that a computer program must perform to accomplish the tasks. Pseudo code (Olsson & Piani 1998) can be used in addition to the interaction diagrams to describe the algorithms better. The interaction diagrams are better on showing the structure and the data flow in the system, while the pseudo code can be better to explain any algorithms. The pseudo coding should be performed in a text document, or can be added as a text box in the interaction digrams. Pseudo coding focus on the dynamic behaviour of the system, not the static behaviour. An example of pseudo coding:

```
GetMovingAverageFilter(FilterBuf[])
{
    if (FilterBufLength > 0)
    {
        sum = 0;
        for (i = 0; i < FilterBufLength; i++)
            sum += FilterBuf[i];
        avg = sum / FilterBufLength;
    }
    else
        avg = 0;
    return avg;
}
```

No need to define variables or follow a strict syntax, but keep a structure logic and use self-explanatory names on variables and methods.

8.4 UML Class Diagrams

UML class diagrams are part of the static design model. A class is a model element that describes behavioral and structure features and is a static model containing the 1) class name, 2) attributes, and 3) messages. A class diagram contains one or several classes, and it will also contain any associations between the classes. An association is a "connection" between two or several classes and is necessary for the classes to cooperate. A diagram for one class is shown in Figure 8.11 and a class diagram with several classes is shown in Figure 8.12.

The class diagram in Figure 8.12 contains 3 classes, each class with one attribute and one message (or method or operation). Class1 can cooperate with both class2 and class3, while class2 and class3 can only cooperate with class1.

An UML class diagram is used for making the following diagrams:

1. The domain model in the analysis part. The focus is the information or "things" in the system,

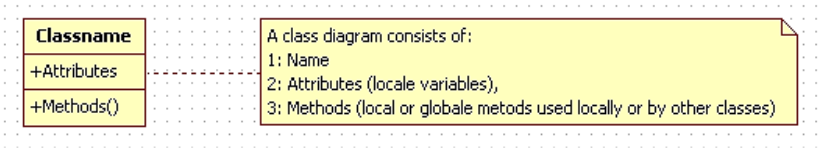


Figure 8.11: A class diagram with only one class with one attribute and one method.

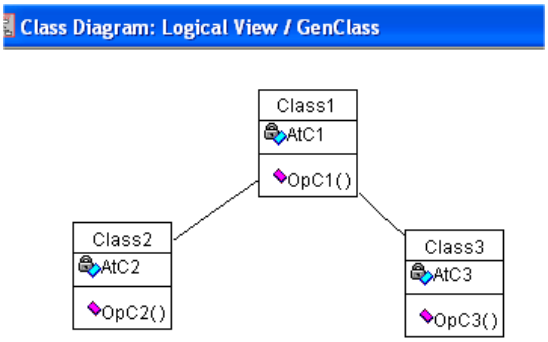


Figure 8.12: A general class diagram containing 3 classes with a attribute and a message.

2. The class diagram in the design part. The focus is the software model,
3. The object diagram in the design part. The focus is the physical implementation of the software in computer memory.

The difference of a class and an object is the number of items. If there exists only one physical item there will be one class and one object. If there exists several physical items of the same type, like a sensor, there will still be only one class but several objects. Having 24 temperature sensors in a system, there will be only one temperature sensor class but 24 temperature sensor objects.

8.4.1 Special type of classes

Most of the classes will be normal classes, however there exist some special type of classes. These classes are:

1. Association class: treat the association as a class,
   (a) many to one and many to many relationships, see Figure 8.14.
2. Singleton class: only one instance of the class (only one object),
   (a) an example can be an algorithme used to convert the physical value of a sensor device to the value of interest, with for example 1000 sensors you will have 1000 copies of the same algorithm. Use a singleton class for the algorithm to have only one object of the algorithme if the algorithme is complex (large).
   (b) Another example can be the PT-1000 converson using a table of 250 doubles for interpolation between the resistance and the actual temperature. Such a table will consists of 4000 bytes (two dimensional table, each of $250 * 8$ bytes). If your system has 1000 sensors you will have 1000 copies of the same table occuping about 4 $MBytes$ of the memory. Using a singleton class you will occupy only 4 $KBytes$ of the memory.
3. Active class: an instance of the class runs and controls its own thread of execution. Used when making threads.

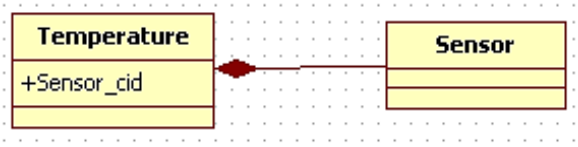


Figure 8.13: A composition class where the temperature class is the front end class and the sensor class is the back end class. Notice the dark diamond on the association indicating the composition class.

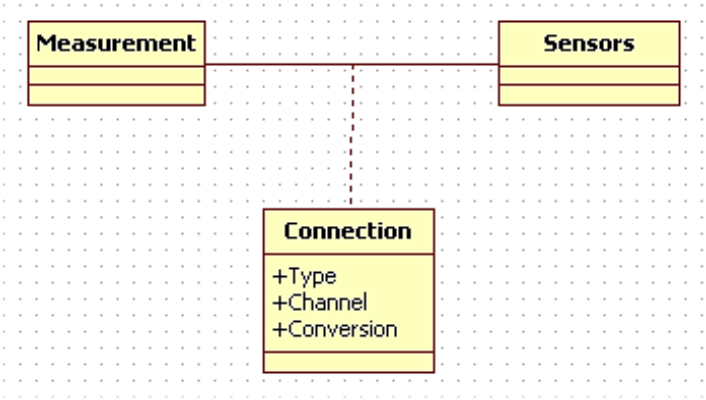


Figure 8.14: An association class used for information about the connection between the measurement class and the sensors class.

8.4.2 Relationship between interaction diagrams and class diagrams

The UP defines the work process. Based on the analysis step one use case has been selected. The design step starts with the interaction diagram defining the classes and the messages between the classes, the messages being the operations (methods) of the classes. Based on the interaction diagram make or extend the class diagram. There will be one interaction diagram for each use case, while there will be only one class diagram and only one object diagram.

8.4.3 Example of code

The class diagram of a temperature measurement system is shown in Figure 8.15 and 8.16.

The C# source code for this system is:

```
/////////////////////////////////////////////////////////////////
//
class TempMeasurement
//
// Purpose: class for temp. measurements
//
// Version: 1.0: 12-DEC-13: NOS
//
class TempMeasurement
 {
```

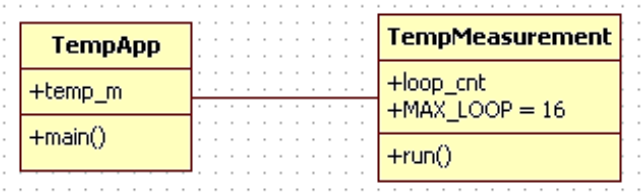


Figure 8.15: The class diagram for a temperature measurement system.

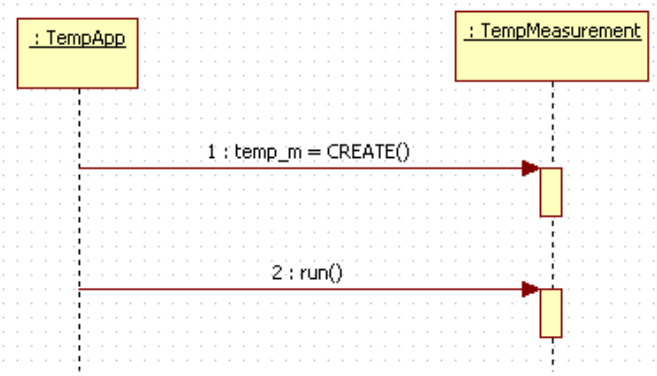


Figure 8.16: The sequence diagram for the temperature measurement system.

```
    const int MAX_LOOP = 16;
    ////////////////////////////////////////////////////////////
    //
    public void run()
    //
    // Purpose: performing the measurement loop
    //
    // Version: 1.0: 12-DEC-13: NOS
    {
        int loopCnt ;

        for (loopCnt = 0; loopCnt < MAX_LOOP; loopCnt++)
        {
            ....
        }
    }
}
/////////////////////////////////////////////////////////////
//
class TempApp
//
// Purpose: The main class of the application
//
// Version: 1.0: 12-DEC-13: NOS
{
    static void Main(string[] args)
    {
        TempMeasurement tempM;

        tempM = new TempMeasurement();
        tempM.run() ;
    }
}
//
/////////////////////////////////////////////////////////////
```

The starting point in any C or C# application will be the Main() method. Based on the sequence diagram and the class diagram should the TempApp class contain the Main() method, creating the TempMeasurement object and start the run method in this object.

8.5 Designing objects with responsibility

The main challange in software design is to transform the information from the analysis step into a software application in the design phase. There will NOT be only one solution, there will be a lot of good and a lot of bad solutions. The important factor here is the designer and the critical design tool for software development is a *MIND* well educated in design principles!

Note that UML is just a standard visual modeling language used for making the design model, UML will not help you in making a good design. UML can however help you in strukturing and documenting the design.

In designing software, several classes must cooperate to fullfill the requirement of the software. In order to fullfill these requirements the classes must have some sort of responsibility, and the focus in the design phase is to be aware of these responsibilities and define the classes to fullfill these responsibilites. In designing classes and objects, the focus should be:

- to assign responsibilities to collaborating objects,
- use patterns.

leading to responsibility driven design (RDD). Patterns are a tool that will help the designer to assign responsibilities to classes.

8.5.1 Responsibilities and Responsibility Driven Design

The starting point in the design phase will be to make an interaction diagram. An interaction diagram should design the behaviour of a use case, from the use case diagram. The use case is described in the fully dressed use case document (FDUCD). However in the FDUCD the focus was on the functonality of the use case, the messages between the actors and the system. Now the focus will be on the design, how to design one or a set of classes to fulfill the functonality. The functonality must be fulfilled by a set of objects communicating via mesages. The classes must have some sort of responsibilities to fulfill this functonality. The responsibilites will be *doing* and/or *knowing*:

Doing

1. Doing something itself, like creating an object or doing a calculation,
2. initiating action in other objects,
3. controlling and coordinating activities in other objects.

Knowing

1. knowing about private encapsulated data,
2. knowing about related objects,
3. knowing about things it can derivate or calculate.

A helpful tool for assigning responsibilities to objects is design patterns.

8.5.2 Patterns

Assigning responsibilities to classes in a structured way is difficult and design patterns are available to help the designer. Patterns (Bruegge & Dutoit 2010), (Larman 2002) are tools that are made by experienced OO developers/designers and describe general principles and solutions for creating OO software in a structured format. A pattern consists of a 1) name, 2) description of the challange or problem, and 3) description of a solution.

Pattern example:

Pattern name	Information expert
Problem	Basic principles for assigning responsibility to objects?
Solution	Assign responsibility to a class that has the information to fulfil it.

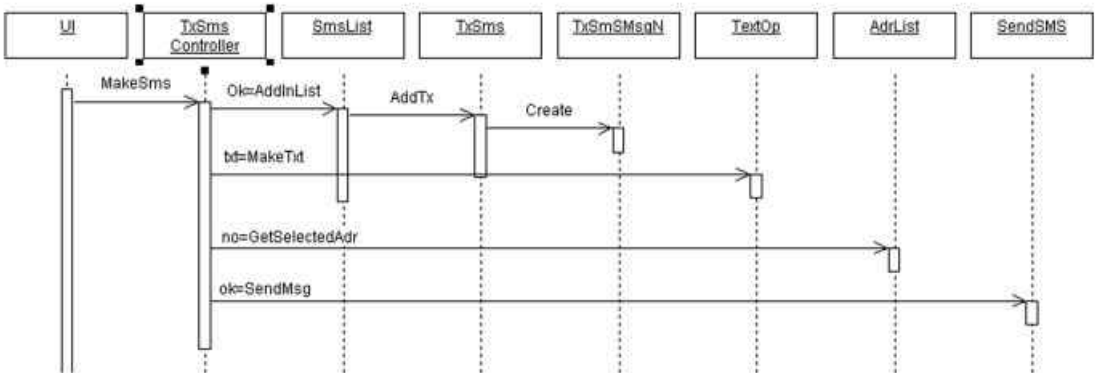


Figure 8.17: Designing the TxSms use case by using an controller to handle the behaviour of the use case.

8.6 GRASP

GRASP (General Responsibility Assignment Software Patterns) is a set of design patterns that can help us in making an object design. Design patterns like GRASP are used in object-oriented design (OOD) as a tool for making a better object design. This section will focus on the following five GRASP (of total nine GRASP):

1. Controller,
2. Information Expert,
3. High Cohesion,
4. Low coupling,
5. Creator.

8.6.1 Controller pattern

According to the UP an use case has been selected, analyised, and now it is time to make the design. An object should be responsible for this use case or user operation, which object to control this use case or user operation (handle the sequence of system events) ?

Pattern name	Controller
Problem	What first object beyond the UI layer reveives and coordinates a system operation ?
Solution	Assign the responsibility to an object representing one of the choices: 1: Represent the overall system, a root object. 2: Represent a use case scenario within which the system operation occurs.

Example 10 *Which object to be in control of sending a SMS message (See the domain model, Figure 6.10) ?*

Use the controller pattern to select an object (class) that is responsible for the use case, and an advice can be to use the same name on the object (class) as the use case. Later on you just need to find the class with the same name as the use case to know which class is responsble for the use case. The method in the class is activated by a message from the user interfase (UI) See the Figure 8.17.

8.6.2 Information Expert pattern

Normally all software applications will need to take some sort of decision, but in order to make any decisions it need some type of information to base these decisions on. Which object should be responsible to provide this information?

Pattern name	Information expert
Problem	Basic principles for assigning responsibility to objects?
Solution	Assign responsibility to class that has the information to fulfil it.

NOTE: A specific information should only be stored in ONE location in your software. (*Never ever store a specific information in different locations !)*

Assign the responsibility to a specific object thaht should have this information in your system, and use the domain model to help you to decide which object to choose. Let the name of the object indicate the type of responsibility of the object and the type of infomration that the object should contain. At this stage you don't care about how the object can get the information, just start with assigning the responsiblity first.

Example 11 *Which object to have the responsibility for knowing the number of SMS stored in the mobile phone ?*

In the Figure 8.17 the object AdrList has been assigned this responsibility.

Example 12 *Which object to have the responsibility for knowing the time and date for the last SMS received by the mobile phone ?*

An object that is responsible for the list of received SMS should have this information, just find the last received SMS in the list.

HINT: Find the object that has most of the information required for a specific responsibility, and assign responsibility there.

8.6.3 High Cohesion pattern

A use case will consists of several objects with different type of responsibility, how many objects should be created for the use case?

Pattern name	High Cohesion
Problem	How to keep objects focused, understandable, manageable, and support low coupling?
Solution	Assign responsibilities so that cohesion remains high.

1. High cohesion means that an object has well defined responsibility, but can use other objects to fulfil this responsibilities. Like a firm where the boss is delegating the responsibility to the staff members, and the ratio of delegation normally tells how good the boss is.
2. Low cohesion means that an object is doing everything itself not giving a very focused object,
3. Important to assign responsibility to the object with the right functionallity,
   (a) make objects with a reasonable number of methods and methods that have some sort of relation,
   (b) each method in an object shall solve only ONE functionallity, and methods solving the same type of functionallity should be kept in the same class,
   (c) the name of the class should indicate the type of responsibilities for that class.
4. Avoid saving the same information at different locations (in different objects). Think syncronizations, and with syncronization only one object should be responsible for providing an updated information.
5. Cohesion indicates the degree to which a class has a single and well-focused purpose. Higher cohesiveness of the classes often gives a better object-oriented design.
6. The benefits of higher cohesion are classes that are 1) easier to maintain, 2) less frequently changed, and 3) easier to (re)use as designed with a well-focused purpose.

Example 13 *Check Figure 8.18 and Figure 8.19, which one to select ?*

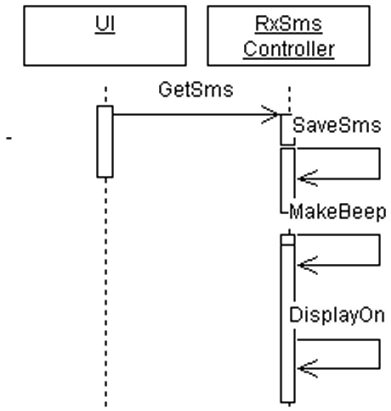


Figure 8.18: Is this a good or bad design ?

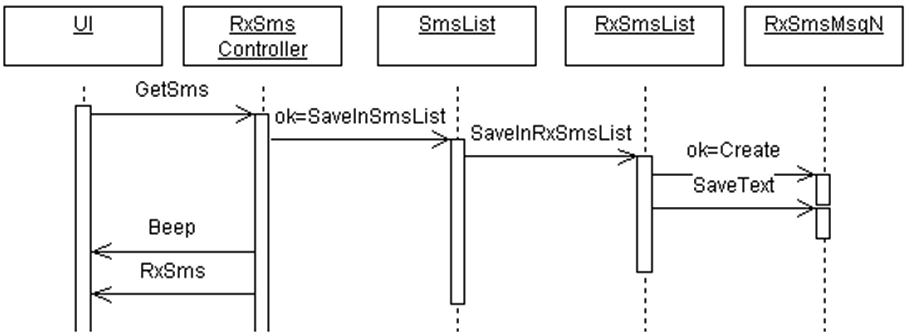


Figure 8.19: Is this design better or worse ?

8.6.4 Low Coupling pattern

A use case will consists of several objects with different type of responsibilities, how should two or several objects be interconnected?

Pattern name	Low Coupling
Problem	How to reduce the impact of change?
Solution	Assign responsibilities so that coupling remains low, evaluate alternatives.

Coupling is a sort of measurement of how strongly one element is connected to, has knowledge of, or depends on other elements. In software design will coupling indicate how strongly an object is connect to other objects, and due to maintenance the coupling should be as low as possible. High cohesion often gives low coupling so focusing on high cohesion will normally also give low coupling.

The reasons for low coupling is what happens when changing one of the elements ?

1. In software design these elements are objects, and objects need to be changed when fixing errors and/or extending the functionality of the software,

2. Important in building software is to keep low coupling as low coupling tends to reduce the time, effort and defects in modifying the software.

Coupling can also be evaluated by looking at the interaction diagrams, the direction of the interactions between the objects. If two objects have interactions both ways they will have a higher then if the objects have interactions one one way. Look at Figures 8.17 and 8.19 showing interactions only one way between the objects giving low coupling.

8.6.5 Creator pattern

An use case will consists of several objects, which object should create a new object?

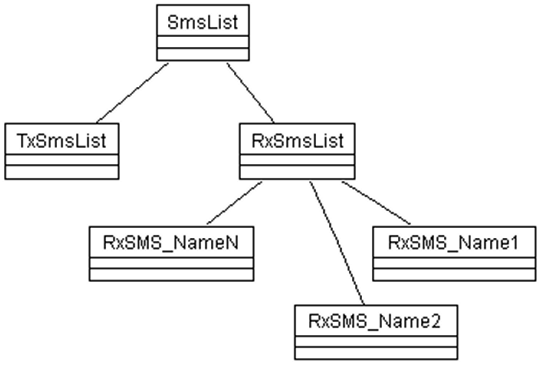


Figure 8.20: A list of software objects for the Rx SMS list.

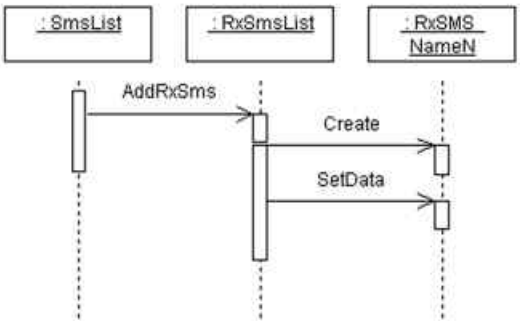


Figure 8.21: Make a new object when receiving a SMS.

Pattern name	Creator
Problem	Who creates the object B ?
Solution	Assign the class A the responsibility to create object B if: 1: A "contains" or compositely aggregates B, 2: A records B, 3: A closely uses B, 4: A has the initializing data for B.

An object must have an association with another object in order to use a method in that object. The association is the reference when creating an object, and if the object has not created the object the reference must be assigned from the object that has the reference.

Example 14 *Who should make an object when receiving a new SMS message ? See the Figure 8.20 and the Figure 8.21.*

NOTE: remember that the object are software objects, not domain model objects !

8.7 Object design examples with GRASP

In starting the design phase, where to start? The UP defines the use case and an interaction diagram. Define the type of interaction diagram to use, and let the FDUCD be the starting point for the design. Start with the patterns, but remember that the patterns are only a tool for making the design, no "magic" about patterns! Use the patterns for use case realization; Make the design model by describing one or more scenarios of a use case, with collaborating objects. Use patterns to find the objects, based on responsibilities in the system. Use UML diagrams for making the design model, interaction diagrams like sequence diagrams and collaboration diagrams, and class diagrams. See Figure 8.22.

The first pattern to use is the controller pattern, start all use cases (system operations) by a controller class. An advace is to use the same name of the controller class as the name of the use case. This may

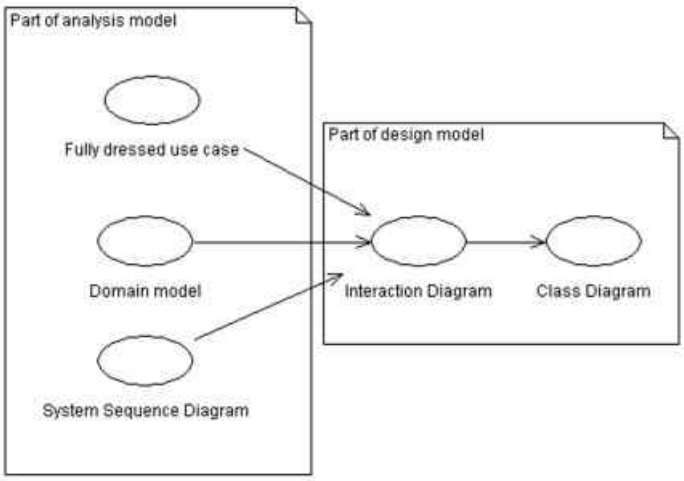


Figure 8.22: Start of the design model.

help you in different ways like 1) what to call the controller class, 2) how to relate a controller class to the analysis phase, and 3) how to relate a controller class to the documentation of the system. The second pattern to use is very often the Information Expert pattern as the software must be based on some sort of information. Where to get this type of information?

8.7.1 Grasp Examples

Advices based on the UP and GRASP patterns:

1. Select the most important use case for design (not already designed),
2. Select a controller class for this use case, normally the same name as the use case,
3. Make an interaction diagram for the use case, now for the software, the inner details of the use case,
   (a) select either sequence or collaboration diagram,
   (b) think GRASP when designing classes and methods.
4. Make or extend the class and object diagrams. Note that there will be only one class and one object diagrams for the system.
   (a) Make or extent the class and object diagrams based on the interaction diagram. Figure 8.23 shows the class diagram based on the interaction diagrams, see Figure 8.17, from the first iteration. We can see that it is from the first iterations since all classes and methods are according to one interaction diagram.

Mobile phone Examples:
Figure 8.24 shows the domain model for a mobile phone.
1: Add an element in the address list

1. Use case name: Add element in address list
2. Controller class ? (Same as use case ??)
3. Make an interaction diagram, DAdrList as controller ? Use GRASP Creator and Information Expert. Input of name and number, Use GRASP -> low coupling -> TextOp, high cohesion by letting DAdrList have the responsibility. See Figure 8.25.

2: Display the name when someone is calling

1. Use case name: Get callers name

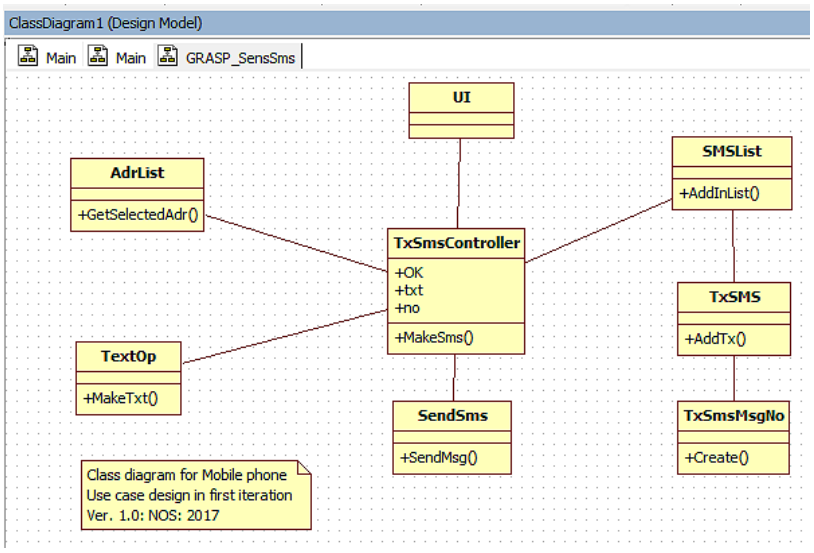


Figure 8.23: A class diagram based on the interaction diagram for TxSmS use case, see Figure 8.17.

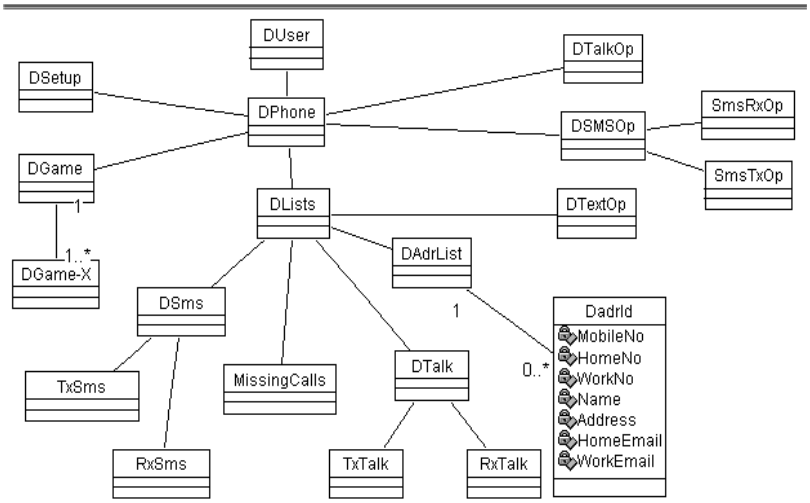


Figure 8.24: The Mobile Phone domain model.

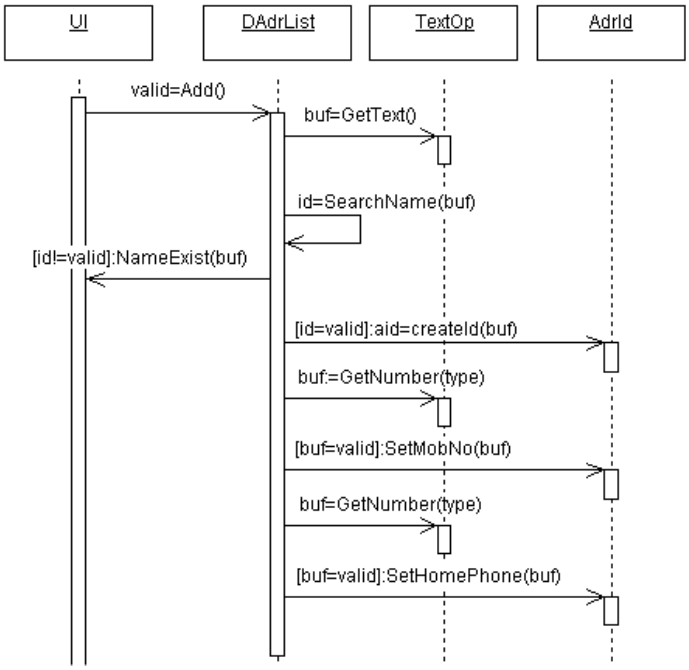


Figure 8.25: How to add an element to the address list.

2. Controller class ? (Same as use case ??)

3. Make an interaction diagram, DAdrList as controller ? Use GRASP Creator and Information Expert. Search -> Use GRASP -> low coupling and high cohesion-> A search function in each AdrId object. See Figure 8.26.

8.7.2 Application startup

The startup code of the application should be part of the elaboration phase to create the first testable version of the software. The code running at startup time:

1. Delay the interaction diagram for the startup code until the system operations are finished (easier to decide the initialization),

2. Start with creating the domain object,

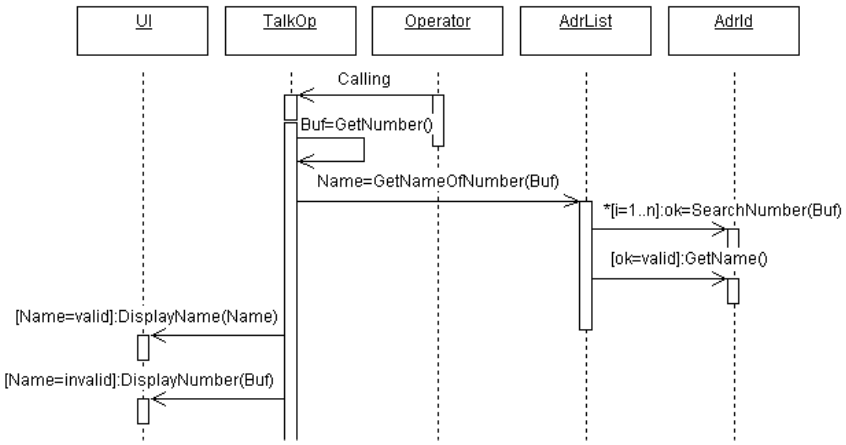


Figure 8.26: How to display the name of the caller.

3. The responsibility of the domain object is to create the main objects in the applications,
4. Real-time system should allocate as much memory as possible at startup, office application can allocate when necessary,
5. Constructors in an object is called automatically when the object is created,
6. Deconstructor in an object is called automatically when the object is deleted.

Example: MobilePhone

1. Create domain object, Phone,
2. Phone creates Setup, Lists, Game, TalkOp and SmsOp (And UI),
3. Use Setup to get data from the SIM card, and ask for password,
4. Connect the phone to the Operator and SMS central.

8.8 Modeling with sequence diagram

Interaction diagrams is either a sequence diagram or a collaboration diagram, and in this document will the focus be on sequence diagrams. Sequence diagram focus on time related behavior and information that will be the preference for process control systems. Sequence diagram was explained in chapter 7. A sequence diagram consists of objects (or classes) and messages between these objects (or classes), either other objects or an internal message. A loop or an alternative sequence can be modeled by using a combined fragment. The type of combined fragment can be set by the interaction operator like *loop*, *alt* (alternative), *break*, and *seq* (sequence) to mention some. A combined fragment can also include interaction operand to model a test with a true or false solution.

Figure 8.27 shows the behavior of a use case in a measurement system. According to the figure will the root object be "MeasurementSystem" and the use case is "ReadSensorValues". The use case will start automatically, setup the DAQ device and read the sensor values from the DAQ device. The figure contains a number of combined fragments to handle a loop and several conditional sequences.

8.9 Mapping designs to code

The interaction diagrams and the class diagrams should be used for implementing a use case.

8.9.1 Design the Class Diagram from the Interaction Diagram

The interaction diagram (sequence diagram or collaboration diagram) will contain the classes, the attributes, and the methods. This means that the interaction diagram already contains all the information necessary for class diagram.

The design of the DCD (DesignClassDiagram) will be:

1. Make a class diagram for all of the classes from the interaction diagram (sequence or collaboration UML diagram),
2. Use the method arrows to indicate the positions for the attributes and methods;
   (a) the methods must be located in the class where the arrow is pointing to (using the method in this class),
   (b) the attribute is the variable, used if any answer from the method, and must be located in the class where the arrow starts.
3. Add association between all classes that have arrow connections,
4. Make ONLY one class diagram from all the interaction diagrams.

Figure 8.15 and 8.16 shows a simple application with a sequence diagram and a related class diagram.
HINTS:

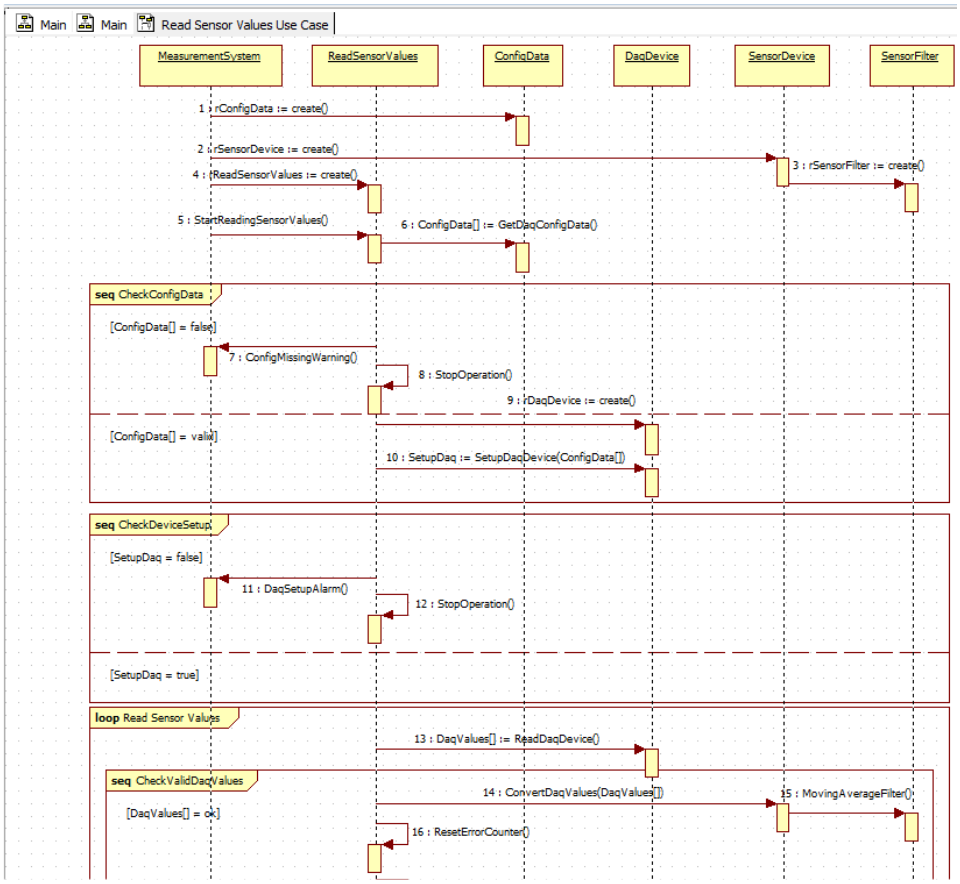


Figure 8.27: A design sequence diagram for a use case in a measurement system. The behavior of the use case is to setup a DAQ device and read the sensor values from the DAQ device. The design sequence diagram contains several combined fragments as loops and conditional sequences.

1. Use create as a common name when new instances of a class is created, do NOT use code specific solutions (CASE tool),
2. Often a get and set methods exist for every attribute, skip these in the model if many attributes (CASE tool),
3. In a multi object, let the parents have the common methods for the "sub"objects,
   (a) Example: The adresslist in mobile phone, let the AdrList object have the find() method, not all the AdrId objects. (See Figure 6.10),
4. Use UML syntax in DCD, not code specific (CASE tool),
5. The type and size of information in DCD depends on the tools and the usage (documentation, designer=programmer=tester etc).

8.9.2 Implementation model (from UP)

In the interation a use case has been selected, and this use case was analyzed and now designed using an interaction diagram. The interaction diagram contains the classes and methods that should be used for the implementation. The steps are:

1. Use the interaction diagrams, the class diagram and the object diagram,
2. Implement a class, and then the methods in this class by writing the source code for the classes and methods. Use one file for each of the classes, and implement the classes and methods with exactly the same name, structure and sequence as in the diagrams. A good advice can be not to include too much details into the design step, focus on the structure in the design steps, and the coding details in the implementation steps,
3. The class diagram (DCD; DesignClassDiagram), the object diagram (DOD; DesignObjectDiagram), and the interaction diagrams will be the documentation of the software implementation.

Chapter 9

Programming

Important aspects of coding are error handling and multiprocessing. Error handling is solved using exceptions in C# and multiprocessing can be solved by using threads.

9.1 Exceptions

1. An error in the application will generate an exception in OO programming language (Java and C#),
2. Can always arise when using a resource in the system, or doing a system operation,
3. Normally a separate class,
4. Using the words: *try, catch, throw* and *finally.* Most used is the *try* and *catch,*
   (a) try: Contains the block of code where the error can arise,
   (b) catch: Contains the block of code dealing with the error.
5. Important exceptions: IndexOutOfRangeException, DivideByZeroException and FileOperations,
6. If your application is not catching the exceptions, OS will do it and terminate your application.

Example: (Loading a file)

```
//////////////////////////////////////////////////////
//
public void loadData(string filename)
//
// Purpose: load data from a text file
//
// Version: 1.0: 12-DEC-13: NOS
{
  try
  {
    if (File.Exists(filename) == false)
      create_file() ;
    else
    {
      FileInfo fp = new FileInfo(filename);
      if (fp.Length == 0)
        create_file() ;
      else
        read_file() ;
    }
  }
```

Figure 9.1: Error message using the try/catch coding.

```
  catch (Exception e)
  {
    Console.WriteLine("Error loading data file:" + e) ;
  }
}
```

Example: (Division by zero, without try/catch)

```
/////////////////////////////////////////////////////
//
static void Main(string[] args)
{
   int value = 2009, divisor = 0, result;

   result = value / divisor;
}
```

This method will generate an exception and because the application will not catch the exception, the exception will be sent to the operating system. The operating system will shut down the applicatioon with the "standard" Windows error message: "The application has performed an illegal operation and will be shut down!".

Examples: (Division by zero, with try/catch)

The application can catch the exception by adding the try/catch code sections.

```
/////////////////////////////////////////////////////
//
static void Main(string[] args)
{
   int value = 2009, divisor = 0, result;

   try
   {
        result = value / divisor;
   }
   catch
   {
        Console.WriteLine("Division error!");
        result = 0;
   }
   Console.WriteLine("Result=" + result.ToString()) ;
   Console.Write("Press any key to quit the application ..") ;
   Console.ReadKey();
}
```

The screen output from this console application is shown in Figure 9.1. The result can be set to a "safe" value, but some more or general information about the exception would be of interest.

Figure 9.2: Error message with type information using the try/catch coding and the exception object information.

An exception object will be generated everytime an exception is generated and the object contains specific information about the exception. The reference to the object is available as an parameter from the catch function. The code is extended with the object reference.

```
////////////////////////////////////////////////////
//
static void Main(string[] args)
{
    int value = 2009, divisor = 0, result;

    try
    {
        result = value / divisor;
    }
    catch (Exception ex)
    {
        Console.WriteLine("An exception occur: Message={0}", ex.Message);
        result = 0;
    }
    Console.WriteLine("Result=" + result.ToString()) ;
    Console.Write("Press any key to quit the application ..") ;
    Console.ReadKey();
}
```

The screen output from this colsole application is shown in Figure 9.2. By using the *ex.Source* it is possible to get the type of exception as well.

It is also possible to catch a specific exception to include different code for different exceptions. This code shows several exceptions, one specific and one general. The code has also included a finally section that will always be called both after the try section or any of the catch sections.

```
////////////////////////////////////////////////////
//
static void Main(string[] args)
{
    int value = 2009, divisor = 0, result;

    try
    {
        result = value / divisor;
    }
    catch (DivideByZeroException ex)
    {
        Console.WriteLine("DivisionByZero!");
        result = 0;
    }
    catch (Exception ex)
    {
        Console.WriteLine("An unexpected exception: Type={0}", ex.Source);
```

Figure 9.3: Error message with information using the try/catch coding and the exception object information.

```
            result = 0;
        }
        finally
        {
            Console.ReadKey();
        }
    }
```

The screen output from this colsole application is shown in Figure 9.3.

9.2 Threading

Threading is useful when the application wants to do:

- I/O processing; for example reading a set of sensors or writing to a set of pumps,
- background processing; doing some calculations or checking independend of the operations of the application,
- some lengthy processing; for example a computation requiring data from a database in the network or combing information from several computers in the network,
- scaling; more easy to scale the application by using threads for I/O or device variations.

Threads are not changing the meaning of the application, only the timing of the operations. Threads are new applications running in parallell with the main application sharing the data. This means that syncronization is important when using threads.

A task or a process consists of the following segments in memory:

- code (text); the memory used for the compiled code of the application,
- stack; the memory used for automatic variables within functions. Data is stored using the Last In First Out (LIFO) method. This means that memory is allocated and deallocated only at the top of stack memory. Stack is mainly used for temporary storage of information.
- heap; the memory used for dynamic memory allocation. Blocks of memory can be allocated and freed in any order. Heavy use of the heap will give a fragmented memory area.

Threads will share the heap memory area. Data can easly be shared between threads, but syncronization must be used when writing to shared memory areas. Figure 9.4 shows the memory layout of an application with three threads, and the layout of each application consisting of code, stack, and heap. Stack is also used for globale variables and is allocated when the application is started. Heap is allocated whenever needed when the application is running.

Appendix E.1 shows the usage of threading. Remember to add the system library at the beginning of the application, using the statement:

```
//////////////////////////////////////////////////////////////////////
//
// Purpose: show the usage of threading in C#
//
// Version: 1.0: 12-DEC-13: NOS
```

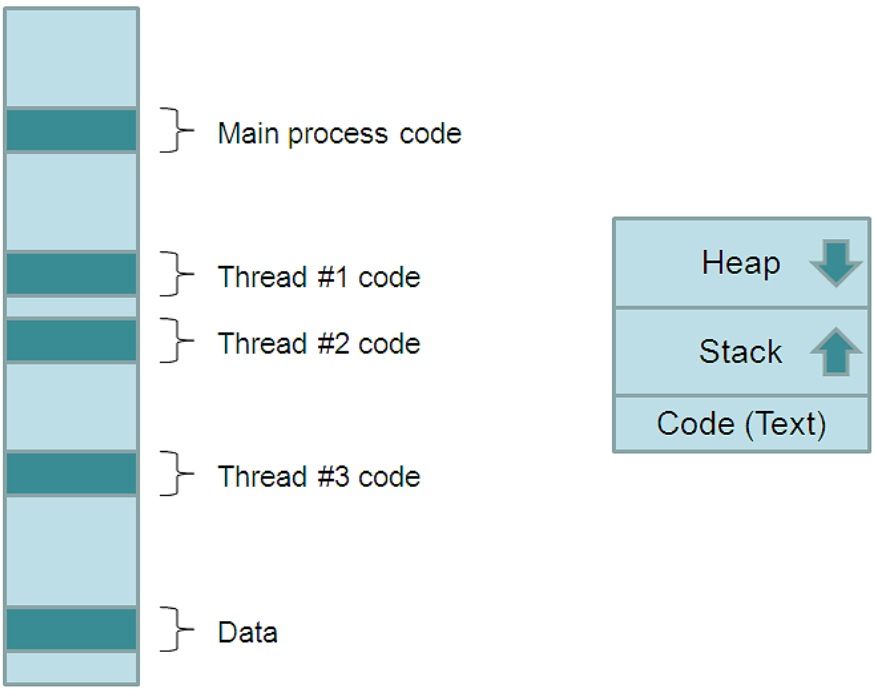


Figure 9.4: The memory layout of an application with three threads, and the layout of an application.

```
//
using System;
using System.Threading;
//Example in using sleep() and Threads i C#
namespace ThreadSystem
{
    ////////////////////////////////////////////////////////////////
    //
    class ThreadClass
    //
    // Purpose: thread class with the thread method.
    //
    // Version: 1.0: 12-DEC-13: NOS
    {
        int         loopCnt ;
        int         loopDelay ;
        Thread      cThread ;
        ////////////////////////////////////////////////////////////
        //
        public ThreadClass(string name, int delay)
        //
        // Purpose: constructor with name and delay parameters
        //
        // Version: 1.0: 12-DEC-13: NOS
        {
            loopCnt = 0 ;
            loopDelay = delay ;
            cThread = new Thread(new ThreadStart(this.run)) ;
            cThread.Name = name ;
            cThread.Start() ;
        }
        ////////////////////////////////////////////////////////////
```

```
        //
        void run()
        //
        // Purpose: running method of the thread
        //
        // version: 1.0: 12-DEC-13: NOS
        {
             Console.WriteLine(" Starting " + cThread.Name) ;
             do
             {
                  loopCnt++ ;
                  Thread.Sleep(loopDelay) ;
                  Console.WriteLine(" " + cThread.Name + ": Loop=" + loopCnt) ;
             } while (loopCnt < 5) ;
             // Ending of the thread
             Console.WriteLine(" Ending " + cThread.Name) ;
         }
     }
     ////////////////////////////////////////////////////////////
     //
     class ThreadSystem
     //
     // Purpose: The main class of the application
     //
     // version: 1.0: 12-DEC-13: NOS
     {
        static void Main(string[] args)
        {
           int cnt, MAX_THREADS = 3 ;
           ThreadClass[] atc = new ThreadClass[MAX_THREADS] ;
           Console.WriteLine(" Start of main program ") ;
           // Making the threads ..
           for (cnt = 0; cnt < MAX_THREADS; cnt++)
           {
              atc[cnt] = new ThreadClass("Thread#" + (cnt + 1),
                                          200 + (100 * cnt)) ;
           }
           // Wait while the threads are running ...
           for (int cnt = 0; cnt < 30; cnt++)
           {
              Console.Write(".") ;
              Thread.Sleep(100) ;
           }
           // End of main program
           Console.WriteLine(" End of main program ") ;
       }
   }
}
//
//////////////////////////// EOC ////////////////////////////////////
```

The output of the threading application is shown in Figure 9.5. The output shows the printouts from the main function and the threads, the orders of the printouts will be depending on the scheduler of the operating system.

The specific order from Figure 9.5 is shown in Figure 9.6. The main process will always start, starting the threads.

This shows how easy it is to make threads in C#. In a measurement system a thread can be used for each DAQ device, reading the value from the sensors devices connected to that specific DAQ device,

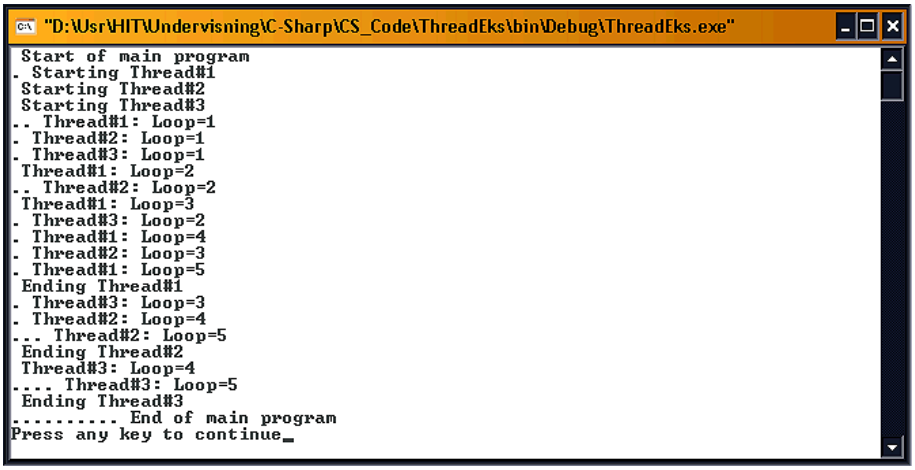


Figure 9.5: The output of the application with three threads.

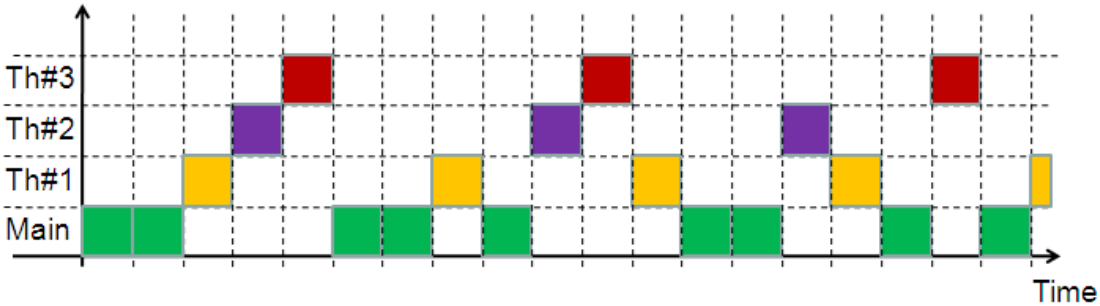


Figure 9.6: The running order of the main process and the threads.

doing the error checking for that specific DAQ device only, and then saving the values. The scaling, the number of DAQ and sensor devices, is then just a configuration parameter, defining the number of DAQ threads to start. The pseudo coding for each tread may be:

```
DaqRunLoop()
{
  while (RunFlag == true)
  {
    SetupFlag = SetupDaqDevice();   // Setup hardware and init the I/O channels
    while (SetupFlag == OK)
    {
       ReadState = ReadDaqDevice(DevId, ValBuf[]);
       if (ReadState == OK)
       {
         for (iCnt = 0; iCnt < ValBuf.Size; iCnt++)
         {
           if (CheckDaqChannel(iCnt, ValBuf[iCnt]) == OK)
           {
             SaveSensorVal(SensId, ValBuf[iCnt]) ;     // Save in shared memory using a semaphore
           }
           else
           {
             ErrorCheckSensor(SensId);
           }
         }
       }
       else
```

```
        {
           ErrorCheckDaq(DevId);
        }
    }
    ErrorCheckSetupDaq(DevId);
  }
}
```

The pseudo coding shows the code structure of the thread with reading and saving the sensor values, and the error handling of the sensor and daq devices. By using a thread with the error handling be somplified as the the thread will handle only one DAQ device and only the sensor devices connected to one DAQ device. If any error exists that will halt or terminate the thread, only one DAQ device measurement will be affected and the main software should handle that error.

Chapter 10

Data communication in C#

10.1 Introduction

The most used method for external communication with a computer is by serial communication. Serial communication means that there is a communication media where the data is transformed in serial form, each bit in every byte is transformed one by one on a single communication medium. The medium can be an electrical wire, in the air (wireless) or an optical cable (using light), and a protocol is needed to describe how the information is transformed and coded on the medium. A protocol contains a set of rules describing how to convert the information in the data between any applications and the communication media. USB, TCP/IP, MQTT, RS-485, CAN and RS-232C are a few of the communication protocols used where a protocol description is documenting the structure of the data and the usage of the communication media. This section is an introduction for using C# for accessing serial communication functions on a computer, for external communication with other communication devices, based on some sort of protocols.

10.2 Serial port data communication

10.2.1 Introduction

Data communication is important for communication between computers, for stand-alone devices like a GPS device, on local area network (LAN) and on internet. Every time you want to send or receive an online message, send or receive a SMS (short message service) send or receive an email, access a web page on internet or download any information from internet, data communication is needed. If you connect a device to your computer, like a GPS device, an USB memory stick, a mobile phone or a camera, data communication is also needed. A computer with external communication capabilities is often referred to as a node and a set of nodes need to use the same structure of the data and the communication media to be able to communicate. However, this structure can be very complex, as shown in Figure 10.1.

This structure is defined by a protocol document describing the details for how to use the communication media and how to interpret the data information. The protocol is a set of rules trying to structure sending and receiving of information. This can be compared with two humans that are trying

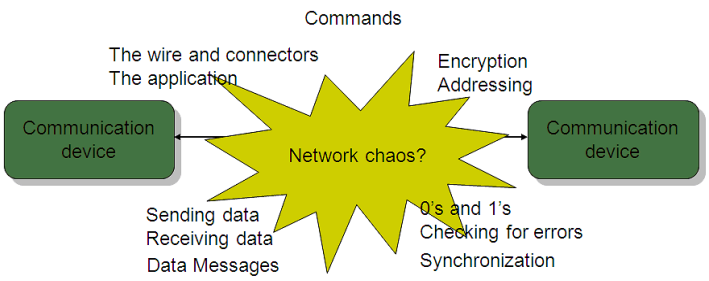


Figure 10.1: An overview of the challenges when two or more nodes (computers) need to communicate using a communication media.

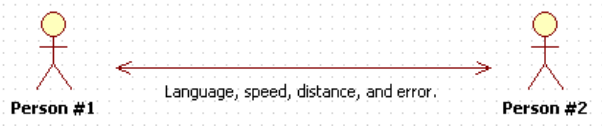


Figure 10.2: Compare a protocol with the situation where two people should be able to talk with and understand each other.

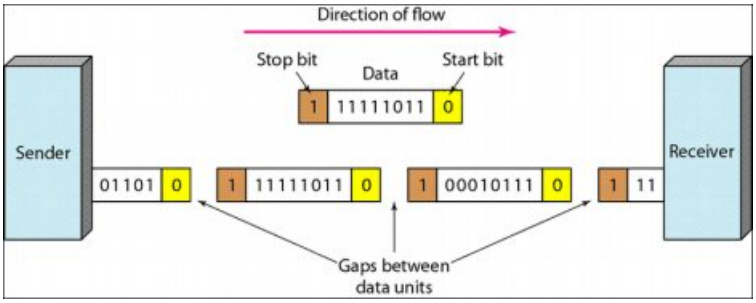


Figure 10.3: Asyncronous transmission using a start and stop bit for syncronization (copy from NetworkEncyclopedia.com / Aug-20).

to communicate, they also need to have a set of rules to understand each other like using a language that both understand, not talk at the same time, and not being too far away from each other, as indicated in Figure 10.2.

Since data communication contains many different states and rules, the Open System Interconnection reference model (OSI model) (Houldsworth 1990), (Levermore 2000), (Stevens 1990), (Stallings 2001) is often used to better structure the situation. The OSI model is a seven layer model defining different types of functionality that must be fulfilled by the protocol. The lowest layer in the OSI model is the physical layer describing the functionality for a physical connection between two or more nodes. This layer is describing the main functionality of the serial port communication. This connection may also be named M2M communication in some literature, short for machine to machine communication.

10.2.2 Physical layer

The physical layer may be wired, optical or wireless (Stallings 2001), and describes how to connect to the physical communication media, how to transfer the data on the communication media and how to handle the data flow on the communication media. The wired connection will be either parallel or serial communication. Parallel is using one wire for each bit, while serial is handling each bit in time slots on the same wire. Parallel is faster but serial is cheaper and can be used over longer distances. So serial communication is used most. Optical is using light (laser) as the communication carrier instead of electrical current as for a wired connection. Wireless can be protocols like Bluetooth and WiFi (Stallings 2001), using radio frequencies as the communication carrier. This chapter will focus further on wired connections only, for serial data communication.

10.2.3 Asynchronous serial communication

Serial data communication will be either synchronous or asynchronous communication where asynchronous is most used. In synchronous communication there will be a continuos stream of data bits and an external synchronization signal is needed to synchronize the sender and the receiver. This synchronization signal must use an extra wire. The asynchronous communication will divide the data stream into characters and add synchronization bits for each of the characters in the data stream. No external synchronization signal is needed however the transmission will take a little more time as the synchronization bits must be transmitted along with the data bits. Figure 10.3 shows asynchronous communication where each byte has a start bit and a stop bit for synchronization between the transmitter (sender) and the receiver. Each byte in this figure consists of eight bits.

The RS-232C, RS-422 and RS-485 protocols are some examples of asynchronous protocols used a lot on computer systems, and these protocols need a set of communication parameters to be configured

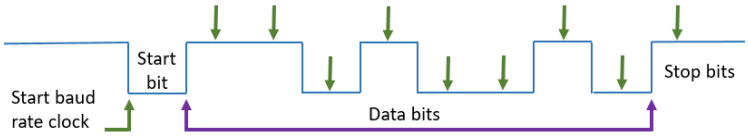


Figure 10.4: The transmiision of the bits in asyncronous serial communication. It will start with a start bit, then the data bits, and a stop bit at the end.

before starting transferring the data stream. RS-232C has normally only one transmitter node and one receiver node, RS-422 has normally one transmitter node and several receiver node, while RS-485 has normally several both transmitter and receiving nodes.

The number of data bits and stop bits can be configured while one start bit is fixed. The communication parameters that can be configured are:

- Baud rate; defining the transmission speed, the number of bits per. second. The baud rate is normally either $300, 600, 1200, 2400, 4800, 9600, 19200, 38400$ or 115200.
- Data bits; defining the number of bits in each character to transmit. The number of data bits are either 5, 6, 7 or 8.
- Parity check; single bit summing up as either None, Even or Odd parity. Even or odd parity will add an extra bit to the data stream, before the stop bit.
- Stop bits; number of bits after the data bits and any parity bit. The number of stop bits are either 1, 1.5 or 2.

The start bit is used for synchronization as the receiver(s) do not know when any node is transmitting. The start bit is normally changing the electrical carrier voltage level from high to low indicating that new data will be transmitted. The start bit will also synchronize the baud rate system for the receiver as the baud rate system is used to define when a new bit is to be transmitted and when the receiver can read this next bit. Figure 10.4 shows the message package, the transmission of one byte, starting with a start bit and ending with the stop bit. The figure also shows when the receiver should check for a new bit, indicated by the vertical green arrows.

The parity bit will be added after the data bits, and before the stop bit(s) if even or odd parity is configured. The extra bit will be 0 or 1 to fulfill the parity configuration. If the data bits are 1101 1111, as shown as one byte in Figure 10.4 will the number of "1" be 7, an odd number of bits. If odd parity is selected will the extra parity bit be "0", or if even parity is selected will the extra parity bit be "1". Parity bit checking can be useful if communication in a noise environment, or over a long distance.

The conversion between the data byte and the data bits, and including communication parameters, is handled by a hardware chip named universal asynchronous receiver and transmitter (UART). The UART will control the baud rate, the number of data bits, any optional parity check, and include the start and stop bits. The frame is the data bits and is part of the total package as shown in figure 10.4. The UART will handle any framing errors meaning any errors of the number of data bits. The reasons can be configuration of wrong baudrate or wrong number of data bits. The UART also contains a first in first out buffer (FIFO) for the receiving part, for example 16 bytes, in addition to the parity and framing error checking. The FIFO buffer allows the serial port to receive more characters before any software driver needs to read these characters. The UART is shown in figure 10.5 where UART1 is part of the transmitter node transmitting a character from the TX (transmitter) part to the RX (receiver) part of the receiver node. So UART 2 is part of the receiver node.

The control of the UART is handled by the drivers of the operating system, and most programming languages have ready made libraries for communication with these drivers. This document will explore the serial port library in C#.

10.2.4 Communication protocols

Before looking into the serial port programming, two asynchronous serial protocols will be exploited, that will be used as examples in the programming section. These protocols are the National Marine Electronics Association (NMEA) 183 protocol that is often used for GPS (Global Position System), and the on-board diagnostics (OBD) used as a network inside most cars produced after 2000. The NMEA

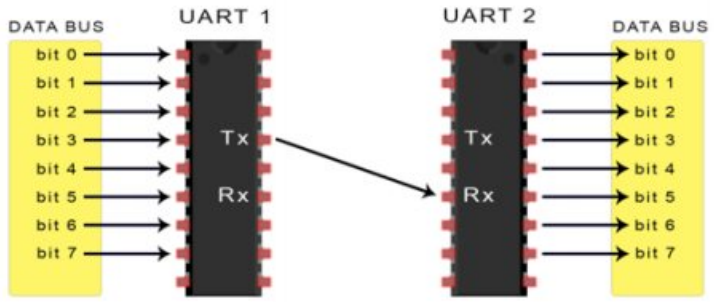


Figure 10.5: The UART hardware chips converting between data bytes and data bits for asyncronous communication between two nodes.

Figure 10.6: One version of an USB to serial port adapter, including the pins for a 9 pins serial port connector. The connector and pinning is defined in the RS-232C protocol.

183 is a one way protocol only so the transmitter is cyclic transmitting all necessary information and any receiver needs to listen to the information and store only the information needed. So only one node can be the transmitter, the rest of the nodes will be receivers. The OBD protocol is a two-way protocol where all the nodes can be either a transmitter or a receiver. Any node can send a request on the network and the node with the information will answer within a specific time frame. However, before digging into these two different protocols, a short introduction on connecting to the serial port is needed.

Serial port

Older computers had one or several RS−232C or RS−485 ports while modern computers only have USB ports. However, many versions of USB to serial port adapters exist, one of these is shown in Figure 10.6.

The three most important pins of the serial port connector is the receive data (RD) pin (rx: pin 2), transmit data (TD) pin (tx: pin 3) and ground (GND) pin (pin 5). Pin 2 is the Rx (receive) connection and pin 3 is the Tx (Transmit) connection as shown in Figure 10.5. This is shown for 9 pins connectors, but there exists 25 pins connectors as well. One way of testing such an adapter is by interconnecting pin 2 and pin 3 and make a test option in the software just to transmit a dummy text and check receiving the same text. This test option is often named loopback testing, and will test both the hardware and the software for that node.

NMEA 183 protocol

The NMEA 183 protocol is a cyclic protocol where the transmitting node is sending a set of messages in a cyclic manner, and is the protocol used for many GPS devices. One is these devices is the Pharos iGPS−180, shown in Figure 10.7. This is a standalone device, the power requirement is 5 V (normally from the USB port) and the protocol is based on RS-232C, the baud rate is 4800, 8 data bits, no parity and one stop bit.

The GPS device has an integrated antenna and is using the NMEA protocol for transmitting the GPS information. The NMEA protocol messages have the following format:

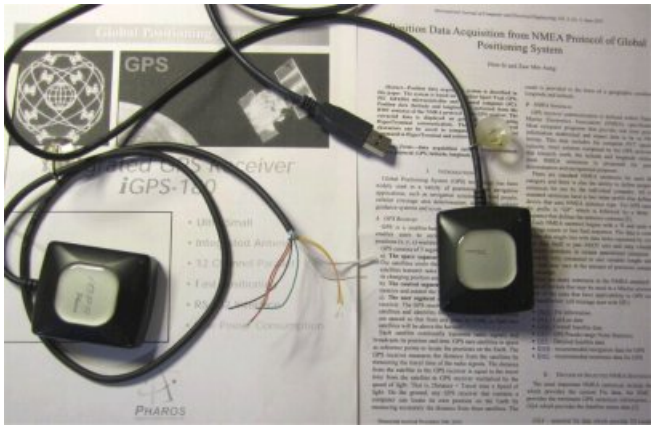


Figure 10.7: GPS devices are hardware devices available in many sizes and forms. The Pharos iGPS-180 is the device used for collecting the data in this document. The figure shows two such devices, one with an USB connection and one with just power and communication connection.

```
Rx[91]=<$GPGGA,000028.997,0000.0000,N,00000.0000,E,0,00,0.0,0.0,M,,,,0000*03
$GPGSA,A,1,,,,,,,,,,,,,0.0,0.0,0.0*30
$GPGSV,3,1,12,06,90>
Rx[92]=<,000,,22,45,000,,24,43,000,,17,42,000,*77
$GPGSV,3,2,12,10,31,000,,26,22,000,,23,20,000,,05,13,000,*78
$GPGSV,3,3,12,25,12,000>
Rx[93]=<,,29,07,000,,04,04,000,,09,02,000,*79
$GPRMC,000028.997,V,0000.0000,N,00000.0000,E,,,051299,,*1F
$GPGGA,000029.996,0000.0000,N>
Rx[94]=<,00000.0000,E,0,00,0.0,0.0,M,,,,0000*03
$GPGSA,A,1,,,,,,,,,,,,,0.0,0.0,0.0*30
```

Figure 10.8: An example of some of the NMEA commands like GPGGA, GPGSV, GPRMC and GPGSA.

$ Command , Param#1 , Param#2 , .., Param#N * ChkSum

where all the fields are based on printable ASCII characters. The $ indicates the start of a new message, the Command indicates the type and number of parameters (1 to N), comma is the standard field separator, and * (asterisk) indicates the checksum of the message. The asterisk is immediately followed by a checksum represented as a two-digit hexadecimal number, indicating the end of one NMEA message. The checksum is the bitwise exclusive OR of the ASCII codes of all characters between the $ and *. The message ends with the <CR><LF> codes. An example of some of the NMEA messages is shown in Figure 10.8.

Figure 10.8 shows one of the challenges with asynchronous serial communication as one and one character is sent. The figure shows that the receive function has been used four times, Rx[91] to Rx[94]. The first read function, Rx[91], contains three NMEA messages. However, only the first part of the last NMEA message is read, the $GPGSV command. This is quite common when reading asynchronous serial communication protocols and the software must include logic to merge the last part of this read function and the first part of the next read function. As shown in the figure, the NMEA messages start with the $ sign and end with checksum field starting with the * sign where each field is separated by the , sign.

The most used commands for GPS information in the NMEA protocol are, where some of them are also shown in Figure 10.8:

```
>
 Time=204005.999  Latitude=5901.3925  Longtude=942.7862 Quality=1
Rx[7762]=< Cmd[9911]:<GPGGA,204005.999,5901.3925,N,00942.7862,E,1,05,1.7,132.0,M,,,,0000*02>
```

Figure 10.9: An example of the GPGGA command with valid information, at a location at Stathelle in Norway.

Command	Description	Fields
GPGGA	Global Positioning System Fix Data	10 (12)
GPGLL	Geographic Latitude and Longitude	4
GPGSA	Satellite status	6
GPGSV	Satellites in view	7
GPRMC	Recommended Minimum sentence C	8
GPVTG	Track made good and ground speed	4

Normally the first two characters, GP, are omitted from the NMEA commands in the literature. The NMEA GGA command (GPGGA) can be used to receive the location of a moving device like a car. Figure 10.8 shows a GGA command, but before the GPS device has received data from enough satellites, the contents is therefor marked as invalid (field #6 is 0). Also note that one of the GGA messages are broken into two parts (in Rx[93] and Rx[94]), and must be merged into one message by logic in the software. The GPS device will use some time to get information from a set of satellites before valid information can be added to these messages. Figures 10.9 shows a message (Rx[7762]) with valid information for a GGA command (field #6 is 1).

The GGA command description in the NMEA protocol documentation has the following information for the different fields (in the message):

Field	Description	Example
1	UTC time from satellite	20:40:06 UTC
2	Latitude	5901.3925
3	Latitude direction	N
4	Longitude	942.7862
5	Longitude direction	E
6	GPS quality: 0=invalid, 1=GPS fix, 2=DGPS fix (0-8)	1
7	Number of satellites	05
8	Horizontal dilution of position	1.7
9	Altitude, Meters, above mean sea level	132.0 M
10	Height of geoid (mean sea level) above WGS84 ellipsoid	
11	Empty field, time of seconds since last DGPS update	
12	Empty field, DGPS station ID number	
	checksum after the * sign	02

The example column (to the right) contains the values from Figures 10.9. The first field, after the command, is an accurate UTC time showing that the information was received at 20:40:06 UTC time, being 22:40:06 local time (Norwegian summer time). The format of latitude and longitude coordinates in the NMEA format is (d)ddmm.mmmm where d=degrees and m=minutes. The degree part can be calculated as an integer by dividing with 100, and minutes part by dividing the rest with 60. The GPS latitude position based on the values from the example column will be:

$$\begin{aligned} 5901.3925/100 &= 59 \\ 1.3925/60 &= 0.023208 \end{aligned} \tag{10.1}$$

and the GPS longitude position will be:

$$\begin{aligned} 942.7862/100 &= 9 \\ 42.7862/60 &= 0.713103 \end{aligned} \tag{10.2}$$

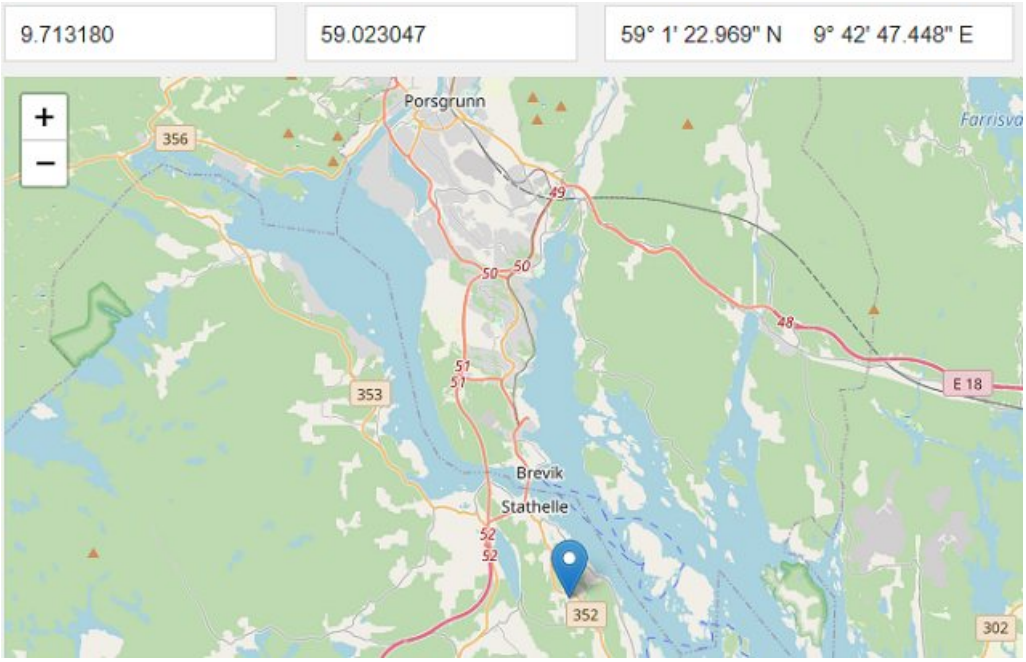


Figure 10.10: A map of the Grenland area in Norway, with the Porsgrunn, Brevik and Stathelle cities. The blue mark is the location of the GPS device at Stathelle, with the position on top of the map.

The direction factor for the latitude position is N=+1 and S=−1, and the direction factor for the longitude position E=+1 and W=−1. The address position for the GPS device is shown at the top of the map in Figure 10.10 with the longitude position = 9.713180 and the latitude position = 59.023047, very close to the position received from the GPS device. These positions are:

Type	Latitude	Longitude
Map	59.023047	9.713180
GPS	59.023208	9.713103

The deviation is due to that the position in Figure 10.10 is based on a street address while the GPS device has a specific position within that address area. Older GPS devices, like this device, have an accuracy of about 5 m while newer GPS devices (newer than 2018) may have an accuracy of about 30 cm. A GPS device can then be used to receive and store the position of a moving object, like a car.

GPS devices using the NMEA protocol has either a serial port or an USB port for connection to the computer, and the serial port (COM port) library can be used in C#. To get more information for a specific moving object, other protocols must be used. The OBD protocol is such a protocol for a car.

OBD-2 / EODB protocol

The On-Board Diagnostic (OBD-2 or OBD-II) standard is a requirement for all cars sold in the United States from 1996 and is a protocol for a network of sensors, actuators and controllers inside cars. European On-Board Diagnostics (EOBD) is the European equivalent of the US OBD II standard, which applies to petrol cars sold in Europe from 2001 (and diesel cars three years later). Cars are using the Controller Area Network (CAN bus) to let microcontrollers and devices to communicate without the need for a server, so the CAN bus is the local area network (LAN) for all the devices installed in a modern car. The OBD protocol is defining the connection port, the signals and the commands. A modern car includes an industrial control system with several distributed control systems like engine controller, braking system (ABS) controller, transmission controller, suspension controller and air conditioner controller to mention some. One possible option for an external communication with this network (LAN) is by using the OBD port.

The CAN protocol is a complex protocol that can be hard to implement in software. A better option is to use the ELM327 interface which is a protocol converter converting between the CAN commands and type of Attention (AT) commands. AT commands is normally used for communication with modems

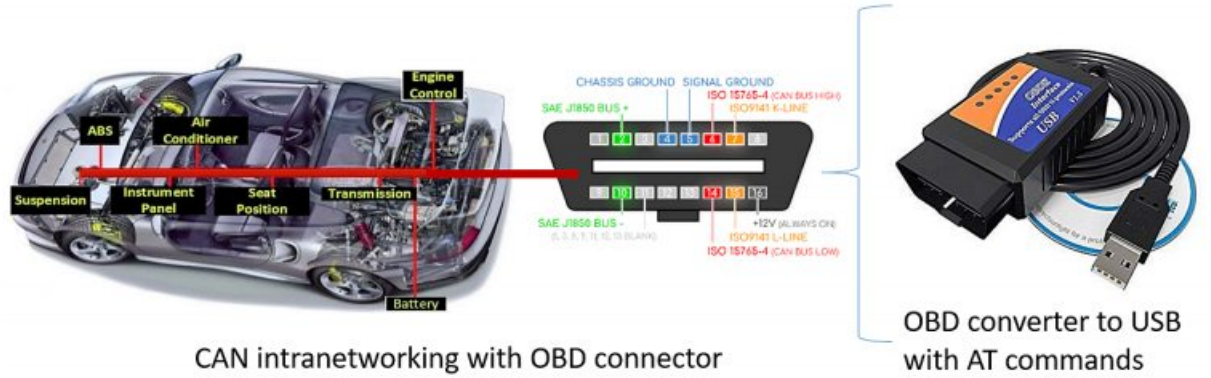


Figure 10.11: The OBD connector, and an example of the ELM327 interface.

where the commands normally start with AT. The ELM327 implementation has customized a set of these commands (Elm 2016). The OBD port and the ELM327 interface is shown in Figure 10.11.

The ELM327 interface can connect to a computer using either USB, Bluetooth or WiFi connection. The easiest way is using an USB connection as this connection can be used as a serial port (COM port) in C#, just like the NMEA protocol. Unlike the NMEA protocol, the OBD protocol is a two-way protocol meaning that the software must first send a request for a specific information, wait for the system to respond, and then send a new request. In this case must the software also handles any timeout situation in case the system is not responding. The protocol baud rate is either 9600 or 38400 depending on the setting of the ELM327 device, 8 data bits, no parity and one stop bit.

One possible startup sequence of AT commands for communicating with the ELM327 interface is:

	Tx	**Rx**	**Comments**
1	ATZ	ATZ ELM327 v1.5	Device reset
2	ATE0	ATE0 OK	Message echo On=1/Off=0
3	ATH0	OK	Message header On=1/Off=0
4	ATL0	OK	Message Line feed On=1/Off=0
5	ATSP0	OK	Set protocol: 0=Auto (Different CAN protocols)
6	ATDP	AUTO	Display protocol type
7	ATI	ELM327 v1.5	Device identification
$8-N$			Looping getting the values

Note that some of these commands may take some time before an answer is received. As shown in Figure 10.11 the car has several controllers monitoring and controlling a set of devices as sensors and actuators. The OBD protocol is handling a set of parameters and these parameters are linked to values handled by these controllers. Many of the these parameters are defined as standard parameters in OBD and are assigned a parameter id number (PID) that will be used by the protocol to get the parameter value.

The OBD protocol is defining different service types or modes and the most important service types are:

Type	**Description**
01	Show current data
03	Show stored diagnostic trouble codes
04	Clear diagnostic trouble codes and store values

The most used service type is 01, to get the current value for a specific parameter. The service types 03 and 04 are used to read any error codes (often combined with a warning on the dash board), and to clear these error codes. The OBD protocol defines at least 10 different service types, check the protocol documentation for more information.

Using the service type 01, a set of parameters can be read from the engine controller. The OBD protocol is defining the parameter ids to read these parameters and a few of these parameters ids are:

PID Hex	PID Dec	Description	Min Value	Max Value	Unit	Rx bytes	Equation
04	4	Calculated engine load	0	100	%	1 (A)	$\frac{A*100}{255}$
05	5	Engine coolant temperature	−40	215	°C	1 (A)	$A-40$
$0C$	12	Engine speed	0	16383.75	rpm	2 (A+B)	$\frac{(256*A)+B}{4}$
$0D$	13	Vehicle speed	0	255	km/h	1 (A)	A
$0F$	15	Intake air temperature	−40	215	°C	1 (A)	$A-40$
11	17	Trottle position	0	100	%	1 (A)	$\frac{A*100}{255}$

Information about more PID can be found in the OBD protocol documentation, a lot is available on internet, like the OBD-II PIDs web site[1]. Many car manufacturers may also have defined their own PIDs in addition to the standard PIDs.

The serial port driver in C# can be used together with the ELM327 device to get information from the OBD port. The commands will be to set up and initialize the ELM327 device and start requesting information. The transmit commands will be:

$$ServiceType\ PIDCode\ (Hex)\ <CR><LF>$$

and the receive information will be:

$$0x40 + ServiceType\ AnswerBytes\ (Hex)\ <CR><LF>$$

An example of the startup communication with the ELM327 device is shown in the following list, where Tx is the command sent from C# application and Rx is the data received by the C# application. The list of the first commands are:

```
16-Apr-2020;16:03:16;Tx;ATZ
16-Apr-2020;16:03:17;Rx;ELM327 v1.5
16-Apr-2020;16:03:17;Tx;ATE0
16-Apr-2020;16:03:18;Rx;ATE0 OK>
16-Apr-2020;16:03:18;Tx;ATH0
16-Apr-2020;16:03:18;Rx;OK>
16-Apr-2020;16:03:18;Tx;ATL0
16-Apr-2020;16:03:19;Rx;OK>
16-Apr-2020;16:03:19;Tx;ATSP0
16-Apr-2020;16:03:23;Rx;OK>
16-Apr-2020;16:03:23;Tx;ATDP
16-Apr-2020;16:03:24;Rx;AUTO>
16-Apr-2020;16:03:24;Tx;ATI
16-Apr-2020;16:03:25;Rx;ELM327 v1.5
16-Apr-2020;16:03:26;Tx;010D
16-Apr-2020;16:03:26;Rx;41 0D 08
16-Apr-2020;16:03:26;Tx;0104
16-Apr-2020;16:03:26;Rx;41 04 67
```

A timestamp is included and note that the respons of the ATSP0 command, set the specific protocol between the ELM327 device and the CAN bus, takes almost four seconds. However, the normal timing is faster, about 200 to 300 ms for each meassge sequence. The last four lines are OBD information from the vehicle where (notice that all values are hexadecimal values):

Id		**Service**	**Command**	**Value**	**Description**
1	Tx	01	$0D$		Request speed
2	Rx	41	$0D$	08	Speed response
3	Tx	01	04		Request engine load
4	Rx	41	04	67	Engine load response

[1] OBD-II PIDs web site: https://en.wikipedia.org/wiki/OBD-II_PIDs (Oct-19).

```
//////////////////////////////////////////////////
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
using System.IO;
using System.IO.Ports;
//
namespace NosComTerm
```

Figure 10.12: Include the serial port library with the "using System.IO.Ports;" statement.

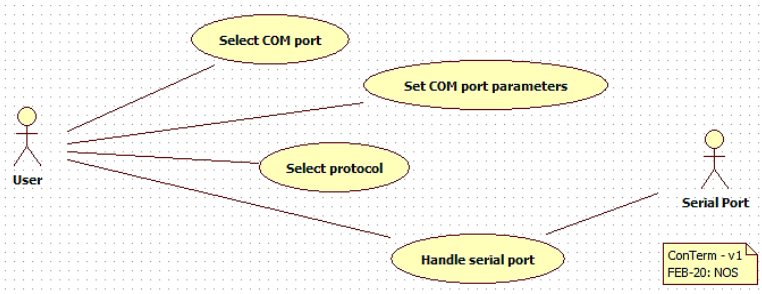


Figure 10.13: A use case diagram for the terminal application, with the user functions of selecting a COM port, set the COM port parameters, select the protocol for that COM port, and handling the data communication on that specific COM port.

The first request is for the vehicle speed, the response starts with $0x41$ which is $0x40 + 01$ which is a valid service type 01 response. The command is $0x0D$ so the current speed is 8 km/ h. The second request is the engine load, and the engine load will be (see the PID list):

$$\frac{0x67 * 100}{255}\% = \frac{103 * 100}{255}\% = 40,4\%$$

The application must, in a cyclic manner, request the wanted parameters from the car. Parameters that are changing often should be requested more frequently than parameters not changing that much. Parameters changing often can be the vehicle and engine speed while parameters not changing that often can be the engine coolant temperature and the intake air temperature. An evaluation of which parameters that are important for your application and how often each parameter should be recorded should be decided in the analysis section of the software development.

10.3 Serial port programming in C#

10.3.1 Introduction

Serial port programming in C# is quite straightforward by using the standard serial library. Include the serial port library in your code, as shown in Figure 10.12, and the serial port can be used like a text file meaning that the port can be open, read from, written to and be closed.

Note that open is exclusive so only one application can open a specific serial port at a given time and therefor it is important to check for a valid open response. For the same reason remember to close any serial port no longer in use in your application, including when changing from one serial port to another serial port.

In this chapter a console application will be developed to test the basic serial port functions with some snippets of source code of using the serial ports and decoding the protocols. As always when developing a software application, a use case diagram should be made and then choose the type of tier architecture. An use case diagram for the application is shown in Figure 10.13. The use case shows the user functions as 1) select a serial (COM) port, 2) set the parameters for this COM port, 3) select the protocol for decoding the information on this COM port and 4) handling the data communication on the COM port.

A three tier architecture is choosen, one tier for the user interface (user interaction), one tier for the business logic of the application meaning handling the logic of the serial protocols and the information

in the application, and one tier for the data layer to handle the serial port. This tier should also handle any logging of data to a file, if needed. A tier can also be named a layer in the literature. The reason for a tier architecture is to get a better structure of the application, and to better separate the code for user interface, the business logic and the data handling.

10.3.2 Three tier architecture

The application was developed as a C# console application using the Visual Studio IDE. The IDE creates a console template with the Program class and the static Main() method. The class ProgramSetupConsoleTerm was created to make the three tier architecture as shown in the C# code snippet:

```
////////////////////////////////////////////////////////////////////////////
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace NosComTerm
{
    /// <summary>
    class Program
    ///
    /// </summary>
    {
        static void Main(string[] args)
        {
            ProgramSetupConsoleTerm rProgramSetupConsoleTerm;

            rProgramSetupConsoleTerm = new ProgramSetupConsoleTerm();
        }
    }
    /// <summary>
    class ProgramSetupConsoleTerm
    ///
    /// </summary>
    {
        LayerData rLayerData;
        LayerBusiness rLayerBusiness;
        LayerUser rLayerUser;

        public ProgramSetupConsoleTerm()
        {
            /// include a try/catch block?
            rLayerData = new LayerData();
            rLayerBusiness = new LayerBusiness(rLayerData);
            rLayerUser = new LayerUser(rLayerBusiness);
            rLayerUser.MainLoop();
        }
    }
}
////////////////////////////////////////// EOC ////////////////////////////////
```

The ProgramSetupConsoleTerm class contains only the constructor running when the object was created. The object creates the tier classes, starts with the LayerData tier, then the LayerBusiness tier including a reference to the LayerData object, then the LayerUser tier object including the reference to the LayerBusiness object, and finally starts the MainLoop() method in the LayerUser object. The

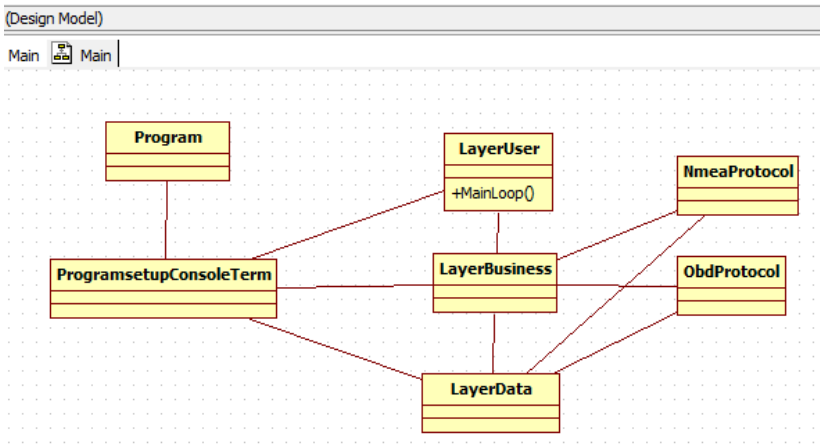


Figure 10.14: The main classes of the terminal application with a three tier architecture.

programming code standard used for development defines that any objects should start with the 'r' character to indicate a reference. The MainLoop() method will display the menu and wait for any input selection by the user.

A try/catch block could have been included as the new statements are system operation and it is a good advice to have a general try/catch block on a "high level" to catch any errors not catched by other try/catch blocks.

The class diagram for the application is shown in Figure 10.14 where the main classes are included. Two classes to handle the NMEA and OBD protocol are included by the LayerBusiness class. Normally a separate class for handling the SerialPort should have been included by the LayerData class, but the SerialPort methods are included in the LayerData to simplifying the application.

The main menu has selections for handling either the NMEA or the OBD protocol that will call an appropriate method in the LayerBusiness class. The method in the LayerBusiness class will call either the method in the NmeaProtocol or in the ObdProtocol class depending on the selected protocol, which will use the serial port methods in the LayerData class.

10.3.3 Serial port library

The serial port library has a set of predefined methods that can be used for handling any serial port on the computer. According to Figure 10.13, the use case diagram, the first option should be to get a list of available serial port, select one of the available serial port(s), and the second option should be to open the serial port with the right communication parameters as baudrate, data bits, parity and stop bits before handling any protocol information.

Available serial ports

The serial port library has a method that will return a string array with the available serial ports in the system. The source code snippet for a method getting the available serial ports in your system:

```
/// <summary>
private string[] GetListOfAvailableComPorts()
/// Purpose: fill a string array with available COM ports, null if no com ports.
/// Version: 1.0: 26-JAN-20: NOS
/// </summary>
{
    string[] saComPortNames;

    try
    {
        saComPortNames = SerialPort.GetPortNames();
    }
    catch
```

```
    {
        saComPortNames = null;
    }
    return saComPortNames;
}
```

The method will return the string array filled with the available serial ports, and saComPortNames.Length will define the number of serial ports in the system. Try/catch is used since this is a system command, and the string array will be null if an error.

Open and Close

Select an available serial port from the list, and open the serial port using the serial port name from the list, the baudrate and with 8 data bits, no parity and one stop bit as default. The serial port names in Windows are on the form *COMx* and in Linux on the form /dev/ttyx, where x is a number from 1 to N (Linux from 00 to N-1). The source code snippet for opening the serial port is:

```
/// <summary>
public bool Open(string sPortName, int iBaudrate, out string sOpenMsg)
/// Purpose: open a COM port with selectable baudrate, fixed data bits, parity and stop bits.
/// Version: 1.0: 26-JAN-20: NOS
/// </summary>
{
    if (bOpenPort == true)
    {
        rSerialPort.Close();
    }
    try
    {
        rSerialPort = new SerialPort(sPortName, iBaudrate, Parity.None, 8, StopBits.One);
        rSerialPort.ReadTimeout = 500;
        rSerialPort.WriteTimeout = 500;
        rSerialPort.Open();
        sOpenMsg = "Open <" + sPortName + "> serial port OK!";
        bOpenPort = true;
    }
    catch (Exception e)
    {
        sOpenMsg = "Error open <" + sPortName + "> serial port: " + e.Message;
        bOpenPort = false;
    }
    return bOpenPort;
}
```

The rSerialPort object reference is a global variable for the class as the object reference must be used for any read() and write() methods for serial port. The binary flag bOpenPort is used for defining if a serial port is open or not. This flag can also be used in the Read(), Write() and Close() section to verify that the serial port is already open. So the first operation when opening a serial port is to close any already opened serial port. Then make a new serialPort object with the correct settings of the baudrate, data bits, parity and stop bits. These parameter can also be changed after the serial port is opened. The parameters ReadTimout and WriteTimeout are timeout parameters for letting the serial port driver of the operating system to timeout if no characters are read or written within the timeout period. These timeout parameters are in ms, in this case the timeout parameters are 500 ms. The Read() operation will try to read a specific number of bytes and will not finish unless the number of characters are read or if the timeout parameter is set. The application will wait for ever ("hang") if no valid if the ReadTimout parameter value is not set (used). An ok open message will be returned to the calling method with the

binary open flag set to true, if the open() method was ok. An error message with information will be returned to the calling method with the binary open flag set to false if the open() method fails.

The source code snippet for closing the serial port:

```
/// <summary>
public void Close()
/// Purpose: close an open serial port
/// Version: 1.0: 26-JAN-20: NOS
/// </summary>
{
    if (bOpenPort == true)
    {
        rSerialPort.Close();
        bOpenPort = false;
    }
}
```

The binary flag bOpenPort is used to only close a serial port that is already open, see the code snippet for open the serial port.

Read and Write

The Read() method will read a number of bytes from the serial port, the parameters of the Read() method are:

public int Read(byte[] buffer, int offset, int count);

where buffer is a byte array, offset is starting point in the buffer array (normally 0), and count is the number of bytes that you want to read from the serial port. Offset is used if you have read only part of a message and want to read the rest of the message. The Read() method supports several type of parameters, check the C# serial port documentation for more information. The source code snippet for an example of reading from the serial port:

```
/// <summary>
public string Read(int iCntMax, bool bTimeoutMsg)
/// Purpose: read a serial port with a maximum number of bytes, with timeout support.
/// Version: 1.0: 26-JAN-20: NOS
/// </summary>
{
    int iLen, iMsgCnt, iOffset, iMaxLoop;
    string sMsgBuf;
    byte[] bMsgBuf;

    iLen = 0;
    sMsgBuf = "";
    try
    {
        bMsgBuf = new byte[iCntMax + 16];
        iOffset = 0;
        try
        {
            iMaxLoop = 0;
            while (iOffset < iCntMax && iMaxLoop < 64)
            {
                iLen = rSerialPort.Read(bMsgBuf, iOffset, (bMsgBuf.Length - iOffset));
                iOffset += iLen;
                iMaxLoop++;
```

```
                }
                for (iMsgCnt = 0; iMsgCnt < iOffset; iMsgCnt++)
                {
                    sMsgBuf = sMsgBuf + Convert.ToChar(bMsgBuf[iMsgCnt]);
                }
            }
            catch (TimeoutException)
            {
                // Do nothing
                if (iOffset > 0)
                {
                    for (iMsgCnt = 0; iMsgCnt < iOffset; iMsgCnt++)
                    {
                        sMsgBuf = sMsgBuf + Convert.ToChar(bMsgBuf[iMsgCnt]);
                    }
                    if (bTimeoutMsg == true)
                    {
                        sMsgBuf = sMsgBuf + "<Timeout>";
                    }
                }
            }
            catch (Exception e)
            {
                // Any other exceptions
                sMsgBuf = sMsgBuf + "<Exception=" + e.Message + ">";
            }
        }
        catch (Exception e)
        {
            sMsgBuf = sMsgBuf + "<Serial port error=" + e.Message + ">";
        }
        return sMsgBuf;
    }
```

The source code snippet for an example of writing to the serial port:

```
    /// <summary>
    public bool Write(string sBuf)
    /// Purpose: write a string buffer to the serial port
    /// Version: 1.0: 26-JAN-20: NOS
    /// </summary>
    {
        bool bWriteOk;

        try
        {
            rSerialPort.Write(sBuf);
            bWriteOk = true;
        }
        catch (Exception e)
        {
            sBuf = "Error write to the serial port: " + e.Message;
            bWriteOk = false;
        }
        return bWriteOk;
    }
```

```
---- Console Terminal : Main Menu ----
Current COM port: COM7
--------------------------------------
N: Select a COM port
O: Open a selected COM port
T: Input the loopback test string
R: Raw dumping of COM port data
L: Test loopback of COM port data
G: GPS data from NMEA protocol (S:Start/Stop)
B: OBD protocol (S:Start/Stop)
M: Modbus (dummy) protocol (S:Start/Stop)
S: Serial Terminal
Q: Quit console terminal

Select a function:
```

Figure 10.15: The output from the MainMenu() method in the application, showing the options that can be selected.

10.3.4 Terminal console application

A console application was developed in C# for testing out the serial port, and NMEA and OBD protocols are two of the protocols that are included. A console application is useful for making a test version and the developed application is developed for testing different type of serial port operations. These operations can be to test the hardware of software of the computer by loopback testing, just dumping all the information transmitted and received regardless of any protocols, a general terminal application able to transmit commands, and test out different types of data communication protocols. The application was developed according to the class diagram shown in Figure 10.14.

Menu selections

The output from the MainLoop() method is shown in Figure 10.15, showing the selectable options in the application.

The following options are available in this terminal console application, but not all of these options will be explained in this section:

Key	Description	Explained
N	Select a serial port, based on available ports in the system	X
O	Open the selected serial port, choosing the right baud rate	X
D	Input the test string used for the loopback testing of the port	
L	Loopback testing of the selected serial port	X
R	Dumping of raw data from the serial port	(X)
G	Display GPS data from the NMEA protocol (S: Start/Stop button)	(X)
B	Display OBD data from the OBD protocol (S:Start/Stop button)	
M	Display Modbus data from the Modbus protocol (S:Start/Stop button)	
S	Generic serial port terminal, transmit commands and receive data	
Q	Quit the terminal console application.	

The first option should be to select a serial port. In this case the application is running on a Windows system so the serial port names will be COMx where x is a number from 1 to N, depending on the number of available serial ports. The user must use the 'N' button to make this selection, and the output from the application is shown in Figure 10.16.

On top of the screen is the current serial port selected, in this case None. At bottom is a list of avaialble serial ports, COM1 and COM7, and the '2' button is used to choose serial port COM7. The next operation is to open this serial port with the appropriate baud rate like the snippet code shown for the Open() method. It is of course possible to combine the selection of a serial port and open a serial port in the same menu selection if wanted. The menu option 'O' will give the output as shown in Figure 10.17 with the selected serial port on the top, and a list of available baud rates at the bottom.

Use the button for the number in front of the baud rate to select the valid baudrate, and the respons shown in Figure 10.18 will be the results. In this case the serial port COM7 is now opened ok and can be used by the application. An error will be displayed if the open() statement fails, as shown in the open() source code snippet.

The port is now open with the selected baud rate and the serial port can now be used for data communication.

```
---- Console Terminal : Main Menu ----
Current COM port:  None
---------------------------------------
N: Select a COM port
O: Open a selected COM port
D: Input the loopback test string
L: Test loopback of COM port data
R: Raw dumping of COM port data
G: GPS data from NMEA protocol (S:Start/Stop)
B: OBD protocol (S:Start/Stop)
M: Modbus protocol (Limited) (S:Start/Stop)
S: Serial Terminal (Tx commands)
Q: Quit console terminal

Select a function:n
Number of available COM ports =
1: COM1
2: COM7
Select id for COM port:
```

Figure 10.16: The 'N' option in the software for selecting a serial port. Available serial ports in this system is COM1 and COM7.

```
---- Console Terminal : Main Menu ----
Current COM port: COM7
---------------------------------------
N: Select a COM port
O: Open a selected COM port
D: Input the loopback test string
L: Test loopback of COM port data
R: Raw dumping of COM port data
G: GPS data from NMEA protocol (S:Start/Stop)
B: OBD protocol (S:Start/Stop)
M: Modbus protocol (Limited) (S:Start/Stop)
S: Serial Terminal (Tx commands)
Q: Quit console terminal

Select a function:o Open serial port ..

Available baudrates:
1: 2400
2: 4800
3: 9600
4: 19200
5: 38400
6: 115200
Select baudrate for the COM port:
```

Figure 10.17: The selection of a baud rate for open the COM7 serial port.

```
Current COM port: COM7
---------------------------------------
N: Select a COM port
O: Open a selected COM port
D: Input the loopback test string
L: Test loopback of COM port data
R: Raw dumping of COM port data
G: GPS data from NMEA protocol (S:Start/Stop)
B: OBD protocol (S:Start/Stop)
M: Modbus protocol (Limited) (S:Start/Stop)
S: Serial Terminal (Tx commands)
Q: Quit console terminal

Select a function:o Open serial port ..

Available baudrates:
1: 2400
2: 4800
3: 9600
4: 19200
5: 38400
6: 115200
Select baudrate for the COM port:2 4800
Message=<Open <COM7> serial port OK!>
Press any key to clear ..
```

Figure 10.18: The '2' button is used to select 4800 as the baud rate, and the serial port is now open OK. The COM7 serial port can now be used by the application.

Figure 10.19: A serial port with a loopback serial port connector that can be inserted to test the serial port. The loopback connection is just interconnecting pin 2 and pin 3 of the serial port, the transmitter and receiver pins.

Loopback testing

The first time using a serial port, a loopback testing can be useful to check that both the hardware and software is working. Loopback testing means that an interconnection between the transmitter pin and the received pin for the serial port must be inserted, and all the data transmitted for serial port should be received by the application. In this way both the transmitter and receiving section of the application, and the serial port will be testing. Figure 10.19 shows a serial port connector with a loopback test connector (to the left, Rx and Tx pins connected) and the NMEA protocol connector (in front, only Rx and Gnd pins connected).

The loopback connector is inserted and the 'D' option is selected to make the loopback test string to be "SerialPortTesting". Then the 'L' option is selected and the output from the application is shown in Figure 10.20. The application will add a counter id after each time reading the loopback test string, so the output should be "SerialPortTestingx" where x is a number from 1 and increasing. The read opeation is defined to read 16 characters from the serial port for each operation, the reason for the figure to display only 16 characters on each line. However, all the characters are read so the serial port and the application is working. The screen update will stop when the loopback test connector is disconnected, or use the 'Q' button to stop this operation. This output also shows the challenges on receiving information on a serial port and merge the information into a valid structure. The receiving method may stop the reading at any time and position of the receiving information, depending on the impementation of reading the serial port. If reading too few characters, the application need logic to merge these characters into a valid message. If reading too many characters, the application will timeout and use more time in collecting the information. Also, reading too many characters may end up reading several messages at the same time, as shown in Figure 10.8.

Dumping of raw data

This can be a very useful option just to dump the all the caracters received from the serial port. Convert the byte[] buffer from the read() method to a string, and print the string on the console. This is shown in Figure 10.8 but remember that this requires an ASCII protocol with mainly printable characters. Any non-printable characters must be converted to printable characters like 0x08, 0x0a and 0x0d can be converted to <TAB>, <LF> and <CR>. For a binary protocol, like Modbus RTU, the information must be converted to printable characters before this option can be used.

```
---- Console Terminal : Main Menu ----
Current COM port: COM11
---------------------------------------
N: Select a COM port
O: Open a selected COM port
D: Input the loopback test string
L: Test loopback of COM port data
R: Raw dumping of COM port data
G: GPS data from NMEA protocol (S:Start/Stop)
B: OBD protocol (S:Start/Stop)
M: Modbus protocol (Limited) (S:Start/Stop)
S: Serial Terminal (Tx commands)
Q: Quit console terminal

Select a function:l Loopback testing ..
Rx[1]=<SerialPortTestin>
Rx[2]=<g1SerialPortTest>
Rx[3]=<ing2SerialPortTe>
Rx[4]=<sting3SerialPort>
Rx[5]=<Testing4SerialPo>
Rx[6]=<rtTesting5Serial>
Rx[7]=<PortTesting6Seri>
Rx[8]=<alPortTesting7Se>
Rx[9]=<rialPortTesting8>
Rx[10]=<SerialPortTestin>
Rx[11]=<g9SerialPortTest>
Rx[12]=<ing10SerialPortT>
Rx[13]=<esting11SerialPo>
Rx[14]=<rtTesting12Seria>
Rx[15]=<lPortTesting13Se>
```

Figure 10.20: The console output from the loopback testing option.

Implement NMEA protocol

A console application can be a good tools to test out the first version of the protocol coding and decoding, to get some experience how to merge and extract the protocol information. Figure 10.8 shows the raw data for the NMEA protocol showing the structure of the message, and Figure 10.9 show the extraction of some of the information from the NMEA command GPGGA. The list in section 10.2.4 shows a sequence of OBD messages, with timestamps, both transmitted and received, for the OBD protocol. Both of these protocols are ASCII protocols meaning that the information consists of ASCII characters, where many of the ASCII characters are printable characters. The decoding of such protocols will be to use different string operations in C#. The general structure is 1) to find the start of the message, 2) find the end of the message to be sure that the whole message has been received, 3) any error checking if checksum is available, 4) subtract the different fields of the message depending on the field separator, and 5) decode the information wanted from these fields.

A source code snippet for one way of extracting information from a NMEA message. Just part of the method is included to get an idea of one possible way of extracting such information:

```
/// <summary>
int decodeNmeaMsg(string sNmeaMsg, out string sErrMsg)
/// Purpose: decode a NMEA message
/// Version: 1.0: 29-JAN-20: NOS
/// </summary>
{
    string[] sNmeaFields;
    string sNmeaCmd, sNmeaChksum;
    double dGpsTime, dLatitude, dLongitude;
    int iGpsQuality;

    /// Spilt the fields based on comma and star
    sNmeaFields = sNmeaMsg.Split(',','*');
    sErrMsg = "";
    /// The last field is the checksum, after the star
    sNmeaChksum = sNmeaFields[sNmeaFields.Length - 1];
    /// No test of the checksum - must be included in the final solution!
    /// Get the NMEA commands
```

```
        if (sNmeaFields[NMEA_CMD_OFFSET].Length > 2)
        {
            /// Remove the GP part of the message command
            sNmeaCmd = sNmeaFields[NMEA_CMD_OFFSET].Substring(2);
            if (string.Equals(sNmeaCmd, "GGA", StringComparison.OrdinalIgnoreCase) == true)
            {
                try
                {
                    dGpsTime = Convert.ToDouble(sNmeaFields[NMEA_GGA_CMD_GPS_TIME_OFFSET]);
```

The message format of any protocol is important to understand when implementing such a protocol and a console application can be a very good way of testing such an implementation before integrating the protocol support into the final application.

10.4 Network communication programming in C#

10.4.1 Introduction

Data communication is important for communication between computers, both on local area networks (LAN)(intranet) and on Internet. The previous chapter looked into serial port communication, using serial ports for external communication. In this chapter will the focus be network data communication, meaning data communication over a network. Serial port data communication has a limited number of transmitters and receivers on the same communication media, and normally the communication media has also a limited length as well. The maximum length is also often depending on the baud rate for serial data communication. Network data communication is using a more complex structure with hardware devices like switches, routers, gateways and firewalls to link network system together, route the messages between these network systems, and using firewalls to protect for which messages that can be received by each sub network.

This means that a network protocol will be more complex than a serial port protocol, however the OSI model can also be used to structure the functionality of these protocols. The serial port protocols depends mainly on the lowest layer in the OSI model, while network protocols will depend on several layers in this protocol model. Still, network communication must still depend on some sort of serial data communication, like the RS-485 protocol used in many network protocols, as the physical layer. The physical layer can still use either wired, optical or wireless physical connection for data communication.

10.4.2 OSI model

The OSI model is an abstract model used to structure any communication protocol into the functionality of seven layers, as shown on the left side in Figure 10.21. The serial port communication described in the previous chapter describes the functionality in the physical layer of the OSI model, while the network data communication is described by the functionality of the first four layers, physical, data link, network and transport layers.

This chapter will not focus on the OSI model(Stevens 1990)(Stallings 2001), but on the TCP/IP protocol (Comer 1991) (Stevens 1990) (Crowcroft & Phillips 2002) (Bentham 2002) used as one protocol for the transport layer and corresponding support from C# libraries. The structure of the TCP/IP protocol is indicated to the right in Figure 10.21. The figure shows that the TCP/IP protocol is using the functionality of the Internet Protocol (IP) for communicating with the nodes on the network, where the main functionality is logical addressing. A node in one local area network (LAN) connected to the Internet can communicate with another node connected to another LAN on the Internet using an IP protocol. The IP protocol will use either the Ethernet or Token Ring protocols for handling the functionality of the two lowest layers in the OSI model, as shown to the right in the figure. The Ethernet protocol is the most used protocol today for handling the Internet communication on these lowest layers.

10.4.3 TCP/IP protocol

The TCP/IP protocol is a network protocol and is based on the client/server concept. This concept define there must be an existing server with some sort of information that a client wants get access to. In data communication a socket is used to establish a connection between two or more nodes (computer

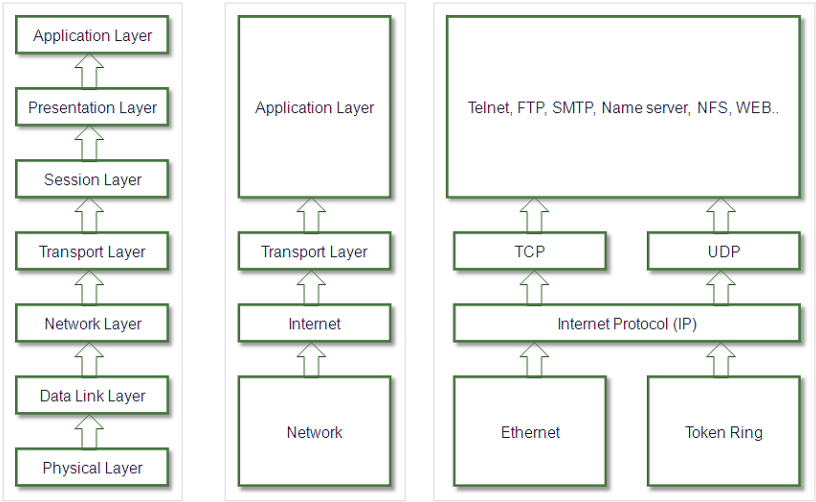


Figure 10.21: The OSI model, the general model to the left, the Internet structure in the middle, and the TCP/IP solution to the right.

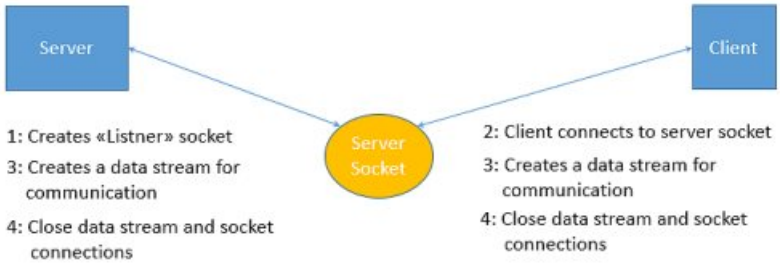


Figure 10.22: The socket connection steps for communication for server and client objects in C#.

with communication capabilities), and these sockets will be used to send data between the nodes. For the TCP/IP protocol will the server first create a listener socket, and any client will create a socket for connecting to the listener socket of a specific server. For the TCP/IP protocol will the IP addresses be used by the both the server and clients to create the socket for communicating, so the IP address of the server should be known. When the socket connection is established, the nodes can exchange information using the socket connection (read and write methods). Figure 10.22 shows the socket connection where the server first make a listener socket and waits for any client socket to connect to that socket (step 2).

When the connection is fulfilled will both systems make a data stream for exchanging information, by reading and writing. Note that this will be a multitasking system as these to applications will run independent of each other. Ending the connection by closing the data stream and socket connection at both ends.

10.4.4 Socket programming in C#

Socket programming (Stevens 1990) (Crowcroft & Phillips 2002) (Bentham 2002) (Mahmoud 2004) is based on a bidirectional communication link between two applications, a server application and a client application, see Figure 10.22. Socket programming is using the IP address so the TCP/IP protocol needs to be implemented in the system, but most nodes today have the TCP/IP protocol included. The C# socket library is included in the System.Net and System.Net.Sockets namespaces, so these two libraries must be included in your application, as shown in the following list::

```
using System;
using System.Text;
using System.Net.Sockets;
using System.Net;
```

The first step is to get the IP address of the, either the running node or the remote node. Use the following commands to get a list of the available IP addresses for the node:

```
ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
ipAddress = ipHostInfo.AddressList[0];
```

The first line gets the information from the Domain Name System, and the information is using the name of this node. The second line gets a list of available IP addresses for this node. The reason for several IP addresses is that the system has support for both IPv4 and IP v6 protocols with a different type of IP addresses. In this case is the first IP address used.

For the server will the next steps be:

```
tcpListener = new TcpListener(ipAddress, iSocketPortId);
tcpListener.Start(5);
tcpClient = tcpListener.AcceptTcpClient();
tcpNetworkStream = tcpClient.GetStream();
```

First make a listener socket that is used for a client to connect, and start this listener socket. In this case it allows for up to five clients to connect at the same time. The AcceptTcpClient() command will wait (block) until a client is connecting to this socket, with the correct IP address and socket port number. Accepting the client connection, a network data stream is created to allow for reading and writing data between the server and client.

The commands will close the connected after finishing the communication:

```
tcpNetworkStream.Close();
tcpClient.Close();
```

For the client will the next steps be:

```
tcpClient = new TcpClient(ipAddress.ToString(), iSocketPortId);
tcpNetworkStream = tcpClient.GetStream();
```

The TcpClient() command will connect to an existing socket connection, with the correct IP address and socket port number. Accepting the server connection, a network data stream is created to allow for reading and writing data between the server and client.

The network data stream can now be used for transmitting and receiving bytes using the Read() and Write() methods for the data stream references. The Read() method will read a number of bytes, and return the number of bytes read. The Read() method can also be extended with the buffer offset and maximum number of bytes to read. The Read() method reading a number of bytes and converting to a printable string:

```
byte[] baBuffer = new byte[64];
int iLen = tcpNetworkStream.Read(baBuffer);
String sBuffer = Encoding.ASCII.GetString(baBuffer);
sBuffer = sBuffer.TrimEnd('\0');
```

The Write() method will write a number of bytes, no return. The Write() method can also be extended with the buffer offset and maximum number of bytes to write. The Write() method making a string, converting the string into a byte array, and write the byte array:

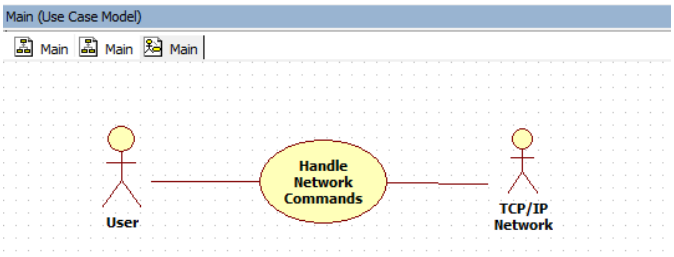


Figure 10.23: The use case diagram for the console socket test application, only one use case for testing commands between a local socket server and socket client.

```
string sBuffer = DateTime.Now.ToString() + ": Server command=<" + sBuffer + ">";
byte[] baBuffer = Encoding.ASCII.GetBytes(sBuffer);
tcpNetworkStream.Write(baBuffer);
```

The commands will close the connected after finishing the communication:

```
tcpNetworkStream.Close();
tcpClient.Close();
```

Note that these socket methods are system methods and try/catch blocks must be used to make the implementation more robust. The Read() and Write() methods handling only a byte array is the challenge for TCP/IP data communication as no standard protocol exists for handling the data structure in these byte arrays. If your application is receiving the byte stream $160A326790B386D2B4A8324447$ you will not have any information about this information unless informed by the sender application system. This is also one of the reason for many proprietary protocols for Internet of Thing (IoT) systems, using the TCP/IP protocol do NOT define any data structure of the byte buffers. The solution will often be to define a new protocol on top of the TCP/IP protocol for handling the application layer of the data communication, see Figure 10.21.

These socket methods will be used in a console test application to show the implementation of socket communication, and a simple protocol will be defined on top of the TCP/IP protocol to handle the messages between the socket server and client.

10.4.5 Console test application

A simple console application will be developed to test out the concept of client/server programming using sockets in C#. The application will also show the challenges with the TCP/IP protocol as this protocol is just defining a data buffer for exchanging information, no interpreting of the information in the data buffer. A very simple protocol has been define, two string commands, for communication between the server and the client. These commands are string based, "LIST" and "Quit"; "List" will display the IP address(es) of the server node and "Quit" will terminate the server/client connection. The TCP/IP protocol is also used for IoT and IIoT devices and these devices will have the same challenge with the TCP/IP protocol, and is the reason for many different IoT protocols, both proprietary and open protocols. These protocols will define the interpretation of the TCP/IP buffer, and support the three upper layers in the OSI model, normally only the "Application layer". The "presentation layer" contains any security functionality and should be integrated into any of these protocols.

The first task when designing a new software application is to make a use case diagram, and this diagram is shown in Figure 10.23.

Since this is a only an application for testing/showing the socket functions in C# for network communication, only an one tier application is developed. If the application is going to be extended with new functionality, a two or three tier application should have been developed. The console application consists of four classes, as shown in the class diagram in Figure 10.24.

The main method of the server class (*RunServerThread()*) will run as a separate process, making this a multitasking system. The main method in the client class (*HandleCommunication()*) will contain the

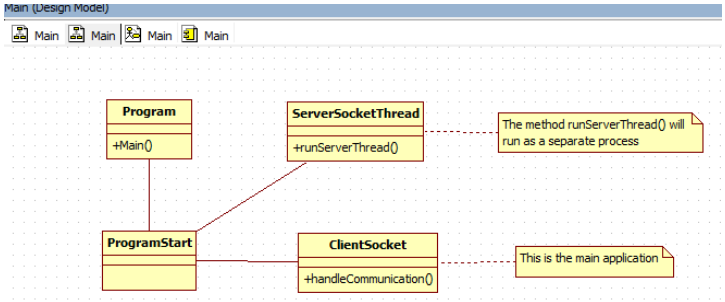


Figure 10.24: The one tier architecture shown in a class diagram, for the network test application.

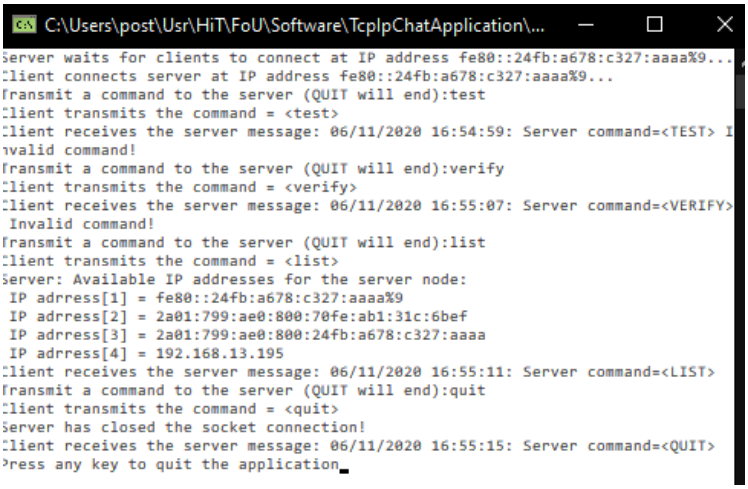


Figure 10.25: The output for the network test application, the socket server running as a background prosess (thread) and the socket client running as the foreground application with user interaction.

functionality of the client, with interface to the user selecting any commands. Program class is generated by the Visual Studio IDE, the ProgramStart class is added to make an object, and this object will first start the server thread as part of the server object, and then the main loop in the client object. The output from this multitasking system is shown in Figure 10.25.

The user first send the command "test" to the server, and the server respond with an invalid command message. Then the user tries the "verify" command with the same response from the socket server. The next command is "list", listing the IP addresses of the server, and finally the "quit" command first terminating the socket connection, then the server process and then the client application. Note that for simplicity will the server write the list messages to the console, the best solution is however for the server to send this information as text messages to the client, and for the client to display this information on the console. Why is this a better solution?

The simple protocol implemented in the application is the two text commands "*LIST*" and "*QUIT*" checked for in the server thread. Also note that the information is sent as plain text on the socket stream meaning that the information can be seen by anybody with access to the communication media and the TCP/IP protocol. Part of a data security policy can be to crypt/decrypt the byte buffers when transmitting and receiving the data. Wireshark[2] is one of the free applications that can be used to visualize data from external TPC/IP messages. The source code for the network console test application:

```
 ////////////////////////////////////////////////////////////////////////////////////////
///
/// Purpose: make a console application showing the usage of TCP/IP socket programming,
///          one thread running as a server, and the application running as a client
/// Version: 19-OCT-20: NOS
////////////////////////////////////////////////////////////////////////////////////////
```

[2] Wireshark, a free and open-source network data packet analyzer, www.wireshark.org, Nov-19.

```
using System;
using System.Text;
using System.Net.Sockets;
using System.Net;
using System.Threading;
///
namespace NosConsoleSocket
{
    ////////////////////////////////////////////////////////////////////////////////
    class Program
    {
        /// <summary>
        static void Main()
        /// Purpose: The main entry point for the application.
        /// Version: 1.0: 20-OCT-20: NOS
        /// </summary>
        {
            /// Convert from a static to a dynamic object
            new ProgramStart();
        }

    ////////////////////////////////////////////////////////////////////////////////
    class ProgramStart
    {
        const int iSocketPortNumber = 9999;
        ServerSocketThread rServerSocketThread;
        ClientSocket rClientSocket;
        /// <summary>
        public ProgramStart()
        ///
        /// Purpose: the startup object for handling the server thread and client
        /// Version: 21-OCT-20: NOS
        /// </summary>
        {
            try
            {
                /// start the socket server first as a separate thread
                rServerSocketThread = new ServerSocketThread(iSocketPortNumber);
                /// Wait to let the server create the connection point for clients
                Thread.Sleep(500);
                /// start the socket client to communicate with the server
                rClientSocket = new ClientSocket(iSocketPortNumber);
                rClientSocket.handleCommunication();
            }
            catch (exception ex)
            {
                Console.WriteLine(" Application error: " + ex.Message());
            }
        }
    }
    ////////////////////////////////////////////////////////////////////////////////
    class ServerSocketThread
    {
        int iSocketPortId;
        bool bServerThreadFlag;
        Thread thServerSocket;
        /// <summary>
        public ServerSocketThread(int iSPN)
```

```
        ///
        /// Purpose: constructor making the server thread using sockets
        /// Version: 22-OCT-20: NOS
        /// </summary>
        {
            iSocketPortId = iSPN;
            bServerThreadFlag = true;
            thServerSocket = new Thread(new ThreadStart(this.runServerThread));
            thServerSocket.Start();
        }
        /// <summary>
        private void runServerThread()
        ///
        /// Purpose: the server thread running as a separate process, handling the server
        ///          functions.
        /// Version: 22-OCT-20: NOS
        /// </summary>
        {
            IPHostEntry ipHostInfo;
            IPAddress ipAddress;
            TcpListener tcpListener;
            TcpClient tcpClient;
            NetworkStream tcpNetworkStream;
            byte[] baBuffer;
            string sBuffer;
            int iCnt;
            bool bCmdValid;

            try
            {
                /// get the IP address information this node
                ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
                ipAddress = ipHostInfo.AddressList[0];
                /// make a TCP connection point based on the IP address and the port number
                tcpListener = new TcpListener(ipAddress, iSocketPortId);
                /// Start the listner connected, allow up to five clients at the same time
                tcpListener.Start(5);
                /// wait for a client to connect to this conenction point (port ID)
                Console.WriteLine("Server waits for clients to connect at IP address " +
                                  ipAddress.ToString() + "...");
                tcpClient = tcpListener.AcceptTcpClient();
                /// a client has connected, make a data stream for communicating with the client
                tcpNetworkStream = tcpClient.GetStream();
                /// loop for communicating with the client, let the client close the connection.
                while (bServerThreadFlag == true)
                {
                    /// wait for receiving a message from the client
                    baBuffer = new byte[64];
                    tcpNetworkStream.Read(baBuffer);
                    sBuffer = Encoding.ASCII.GetString(baBuffer);
                    sBuffer = sBuffer.TrimEnd('\0');
                    /// check for a valid message from the client
                    /// Note that the TCP/IP protocol just define a byte data buffer, no data
                    /// structure. You must define your own protocol based on this buffer. The same
                    /// for IoT communication, often defining a propritary protocol on top of TCP/IP
                    /// Define a simple protocol, based on string commands; LIST and QUIT.
                    /// LIST shows the IP addresses for the server and QUIT terminates the server co
                    sBuffer = sBuffer.ToUpper();
```

```
                    if (string.Equals(sBuffer, "QUIT") == true)
                    {
                        bServerThreadFlag = false;
                        bCmdValid = true;
                    }
                    else if (string.Equals(sBuffer, "LIST") == true)
                    {
                        Console.WriteLine("Server: Available IP addresses for the server node:");
                        for (iCnt = 0; iCnt < ipHostInfo.AddressList.Length; iCnt++)
                        {
                            Console.WriteLine(" IP adrress[" + (iCnt + 1).ToString() + "] = " +
                                ipHostInfo.AddressList[iCnt].ToString());
                        }
                        bCmdValid = true;
                    }
                    else
                    {
                        bCmdValid = false;
                    }
                    /// Send an answer back to the client
                    sBuffer = DateTime.Now.ToString() + ": Server command=<" + sBuffer + ">";
                    if (bCmdValid == false)
                    {
                        sBuffer = sBuffer + " Invalid command!";
                    }
                    baBuffer = Encoding.ASCII.GetBytes(sBuffer);
                    tcpNetworkStream.Write(baBuffer, 0, baBuffer.Length);
                }
                /// The connection is closed, close the data stream and socket.
                tcpNetworkStream.Close();
                tcpClient.Close();
                Console.WriteLine("Server has closed the socket connection!");
            }
            catch (Exception ex)
            {
                /// Display any errors for the server socket thread
                Console.WriteLine("Server error: ", ex.Message);
            }
        }
    }
    ////////////////////////////////////////////////////////////////////////////////
    class ClientSocket
    {
        const int MAX_RX_LEN = 128;
        int iSocketPortId;
        IPHostEntry ipHostInfo;
        IPAddress ipAddress;
        TcpClient tcpClient;
        NetworkStream tcpNetworkStream;
        string sTxBuf, sRxBuf;
        byte[] baBuf;
        /// <summary>
        public ClientSocket(int iSPN)
        ///
        /// Purpose: constructor for the client class
        /// Version 23-OCT-20: NOS
        /// </summary>
        {
```

```
            iSocketPortId = iSPN;
        }
        /// <summary>
        public void handleCommunication()
        ///
        /// Purpose: the client application, communicating with the server process
        /// Version: 23-OCT-20: NOS
        /// </summary>
        {
            try
            {
                /// Get the IP address of this node
                ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
                ipAddress = ipHostInfo.AddressList[0];
                /// connect to server running on localhost at the specific port number id
                tcpClient = new TcpClient(ipAddress.ToString(), iSocketPortId);
                Console.WriteLine("Client connects server at IP address " + ipAddress.ToString() +
                                  "...");
                /// make the data stream for reading and writing with the server
                tcpNetworkStream = tcpClient.GetStream();
                Console.Write("Transmit a command to the server (QUIT will end):");
                sTxBuf = "";
                /// handle the simple string based protocol for the server communication
                while (string.Compare(sTxBuf, "QUIT", true) != 0)
                {
                    /// get the next command from the user
                    sTxBuf = Console.ReadLine();
                    Console.WriteLine("Client transmits the command = <" + sTxBuf + ">");
                    baBuf = Encoding.ASCII.GetBytes(sTxBuf);
                    /// transmit the command to the server, and wait for any answer
                    tcpNetworkStream.Write(baBuf, 0, baBuf.Length);
                    /// release cpu time for other processes while waiting for server response
                    Thread.Sleep(100);
                    /// Read any answer from the server
                    baBuf = new byte[MAX_RX_LEN];
                    tcpNetworkStream.Read(baBuf, 0, baBuf.Length - 1);
                    sRxBuf = Encoding.ASCII.GetString(baBuf);   // convert byte array to string
                    Console.WriteLine("Client receives the server message: " + sRxBuf);
                    if (string.Compare(sTxBuf, "QUIT", true) != 0)
                    {
                        Console.Write("Transmit a command to the server (QUIT will end):");
                    }
                }
                /// Close the socket communication with the server
                tcpNetworkStream.Close();
                tcpClient.Close();
                Console.Write("Press any key to quit the application");
                Console.ReadKey();
            }
            catch (Exception ex)
            {
                /// Handle any client erros
                Console.WriteLine("Client error: ", ex.Message);
            }
        }
    }
}
///
```

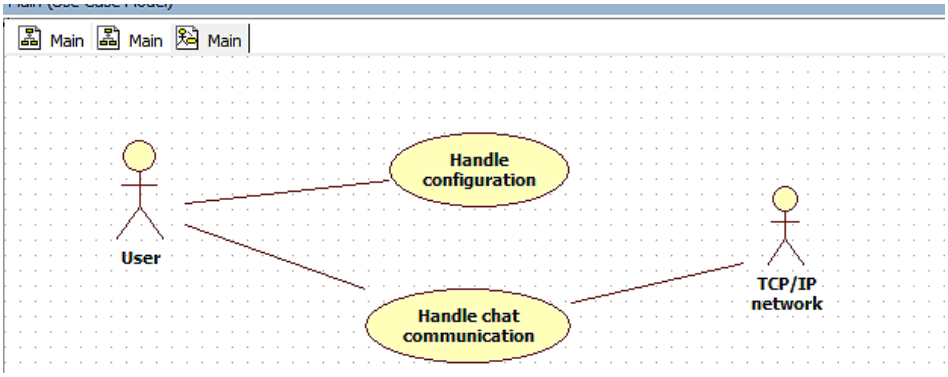


Figure 10.26: The use case diagram for the main functionality of the chat application, with two use cases. One use case for handling configuration and one use case for handling the chat communication.

// EOC //

10.4.6 GUI Chat application

The console test application is running both the server and the client on the same node which does not take any advantages of network communication. A Graphical User Interface (GUI) application that can be used as a chat application between two nodes is developed to show full utilization of network communication. With this application two user should be able to communicate (chat) running the same application on two different nodes in a local area network (LAN). It should also be possible to use the application over Internet but requires that the selected port number is available for communication through any firewalls. Note that the usage of this application should trigger your "antivirus/internet security" protection as this application is trying to make a TCP/IP connection on a non-standard port number.

The use case diagram for application is shown in Figure 10.26, showing the main user functions for the application.

The first use case will be handle the configuration of the application (server/client mode, IP address and socket port number) and the second use case for the chat function where the users can chat with each other by writing messages. The connection will be part of the second use case as this will involve both users, where the remote user is shown as the TCP/IP network actor. The structure of the application will be a three tier architecture, one tier for the user interface, one tier for the business logic for handling the communication modes and exchange of messages, and the tier for the data layer handling the socket communication. The data layer can also be extended with a log feature saving the messages on a text file. The class diagram for this structure is shown in Figure 10.27.

The most important classes is shown in this class diagram, two classes (Program and ProgramStart) for starting the application, three classes for handling the tier functionality of the application (LayerUser, LayerBusiness and LayerData), one class for the configuration parameters, and two classes handling the threads for server and client socket communication. Only one of these threads will be running in a specific application, depending on the configuration. A thread is very useful to let the socket communication be a background process independent of any user interaction. Figure 10.28 shows the deployment diagram, showing the two instances of the application running in a network. One application configured as the socket server, with the ServerSocket thread object activated, and one application configured as the socket client, running the ClientSocket thread object.

The C# library and methods are the same as used for the console application, however this application is extended with a configuration section and a three tier architecture. The user interface for the application is shown in Figures 10.30 and 10.29, where Figure 10.30 shows the configuration and some messages for the client application and Figure 10.29 shows the configuration and some messages for the server application.

The user interface consists of five sections, one section with the menu bar and the host name on the top, two message sections with the transmitted messages to the left and the received messages to the right. The section in the lower left corner is status information from the application, in this case the connection and disconnection to any remote nodes. The last section, in the lower right corner, is the

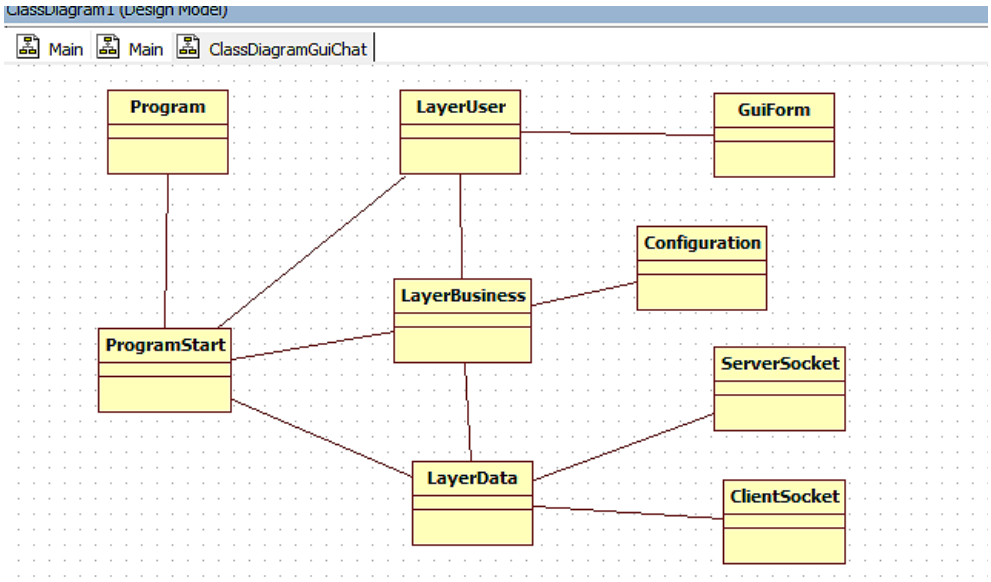


Figure 10.27: The three tier architecture class diagram of the GUI chat application, with the program startup classes, the tier classes, socket mode classes and a class for configuration parameters.

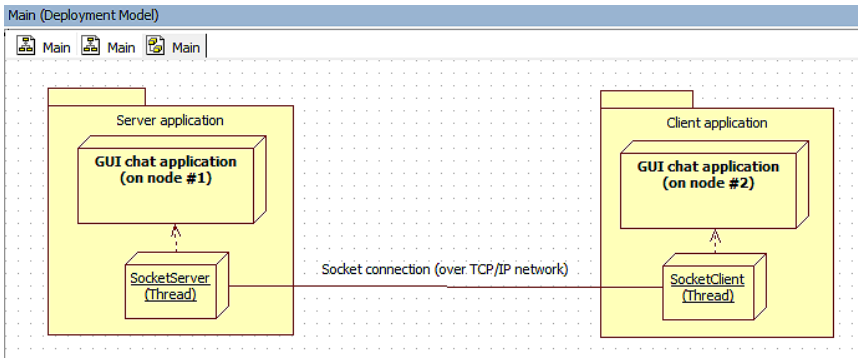


Figure 10.28: The deployment diagram for the application, running two applications. One application as the server, one application as the client, and using a socket connection as the TCP/IP communication.

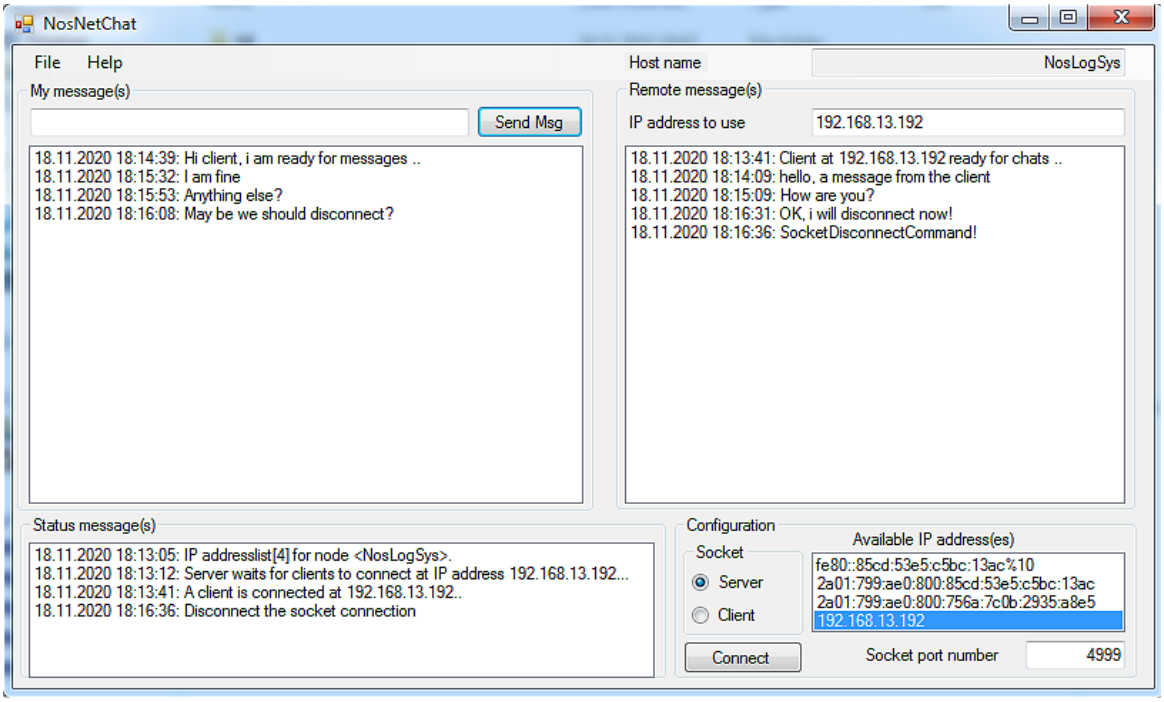


Figure 10.29: The configuration and messages for the server application.

configuration section, where the user selects the socket (server or client), the port number and the IP to use.

The setup of the system should start with the server application, and the configuration section. Select an unused port number, a good choice is to select a port number above 1024. The valid port numbers are 0 to 65535. Note that using an unused port number above 1024 should result in a warning from the Antivirus/Internet protection of your system as these ports will also be used for a cyber security attach of computers. Select the socket type, server or client, and an IP address. A socket connection consists of a socket number and an IP address. In the upper right corner of the application in Figure 10.29 the name of node is shown, the server node name for the socket the connection. The IP addresses are shown in the list in the lower right corner, shown both IPv4 and IPv6 addresses for this node. A good choice can be to use the IPv4 addresses as this is more easy for the user of the client application to input the IP address. Click on the address that you want to use, and the IP address will be shown in the right upper corner. Use the *Connect* button to start the server socket thread. Messages in the status window, lower left corner, shows that IP addresses are listed, and that the server has started with a specific IP address. The Connect button will change the text to *Disconnect* when connected.

The setup of the client application must use the same setting as the server application, as shown in Figure 10.30. The node name in the upper right corner is the name of the client node, not the same as the server node. The next field in the upper right corner must be the IP address of the server, so the same IP address as in Figure 10.29. Use the same IP address and socket number as the server, and use the *Connect* button to connect to the server socket. Information in the status windows shows the status of this connection. When connected, input a message in the input field in the upper left corner and use the *Send Msg* button to the send the message to the remote node. The upper left list window shows the messages sent from the node, and the upper right list window shows the messages received from the remote node.

The source code for this application will not be included, but is using the same methods as shown for the console application. Some comments with code snippets will however be included for specific part of this application, as a multitasking (multithreading) GUI application will include some challenges with sending events between the threads. The software architecture is a three tier architecture for this application, as shown in Figure 10.27 and the normal message flow is to use methods to get information from a lower tier. The challenge is to get information from a lower tier up to a higher tier when receiving external information and in this application is the publish/subscriber concept (best practice - design pattern) used. The publisher/subscriber concept consist of one publisher, active on a lower tier, and activated when receiving external information. Any method in a higher tier can subscribe on these

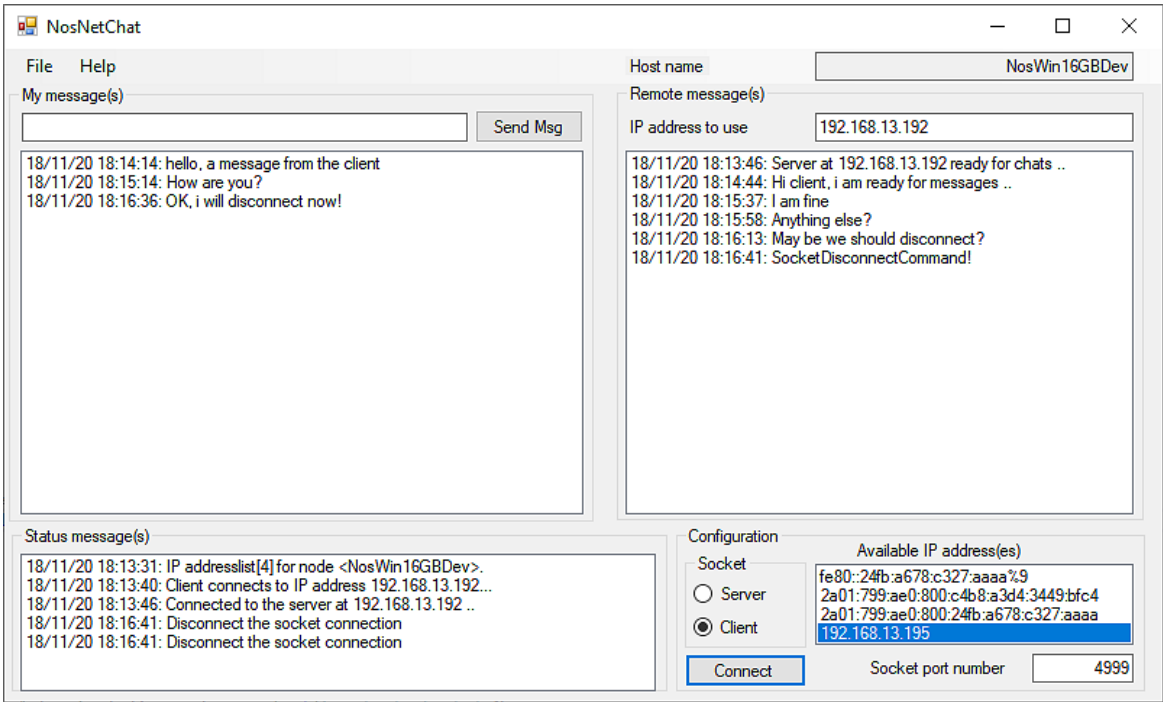


Figure 10.30: The configuration and messages for the client application.

publisher, and be informed when new external information is received.

The data flow in this application is, with reference to Figure 10.27, is when sending a message is to make the message in the user tier, send the message buffer down to business tier, further down to the data tier, and the active thread for sending to the remote node. The challenge is receiving a message from the remote node. One solution can be to let the user tier poll the socket thread all the time but this will waste cpu power and memory, here the publisher/subscribe concept should be used. The threads will be the publishers, and the user tier will be the subscriber, receiving an event whenever new information is received from the remote node.

The source code snippets for the publisher/subscriber logic:

1 : define the callback methods as delegate methods in the header of the LayerUser class:

```
/// <summary>
/// Purpose: event handler with the UiEventArgs object as information
/// </summary>
public delegate void HandlerUiMsg(object rSender, UiEventArgs eUi);
delegate void AddReceivingMsgCallback(string txt);
delegate void AddStatusMsgCallback(string txt);
```

2 : define the event handler in both the LayerUser and LayerData classes:

```
HandlerUiMsg eListenUiMsg = new HandlerUiMsg(OnHandlerUiMsg);
rDataLayer.ePublishUiMsg += eListenUiMsg;
```

3 : define the event handlers. Note that the events has to be sent between different threads so these handlers must be thread safe:

```
/// <summary>
public void OnHandlerUiMsg(object rsender, UiEventArgs eArgs)
///
/// Purpose: handling UI event commands
```

```
        /// Version: 14-NOV-20: NOS
        /// </summary>
        {
            switch (eArgs.iCmd)
            {
                case (int)Constants.UiEventCmd.MsgRxList:
                    addReceivingMsgThread(eArgs.sMsg);
                    break;
                case (int)Constants.UiEventCmd.MsgStatusList:
                    AddStatusMsgThread(eArgs.sMsg);
                    break;
                case (int)Constants.UiEventCmd.MsgSocketButton:
                    rGuiForm.buttonSocketConnect.Text = eArgs.sMsg;
                    break;
                default:
                    addStatusMsg("Error: Rx invalid UI event cmd = " + eArgs.iCmd.ToString());
                    break;
            }
        }

        /// <summary>
        public void AddStatusMsg(string sStatusMsg)
        ///
        /// Purpose: add a messge to the status window
        /// Version: 13-NOV-20: NOS
        /// </summary>
        {
            string sMsg;

            sMsg = DateTime.Now.ToString() + ": " + sStatusMsg;
            rGuiForm.listBoxStatusMessages.Items.Add(sMsg);
        }
        /// <summary>
        public void AddStatusMsgThread(string sRxMsg)
        ///
        /// Purpose: update the status window in the GUI, threadsafe
        /// Version: 1.0: 11-OCT-20: NOS
        /// </summary>
        {
            // InvokeRequired compares the thread IDs
            if (rGuiForm.listBoxStatusMessages.InvokeRequired == true)
            {
                try
                {
                    AddStatusMsgCallback rCallback = new AddStatusMsgCallback(addStatusMsgThread);
                    rGuiForm.Invoke(rCallback, new Object[] { sRxMsg });
                }
                catch
                {
                    // No support, normally at startup or close time
                }
            }
            else
            {
                AddStatusMsg(sRxMsg);
            }
        }
        /// <summary>
```

```
    private void AddReceiveMsg(string sRxMsg)
    ///
    /// Purpose: add a messge to the receive message window
    /// Version: 13-NOV-20: NOS
    /// </summary>
    {
        string sMsg;

        sMsg = DateTime.Now.ToString() + ": " + sRxMsg;
        rGuiForm.listBoxRxMessages.Items.Add(sMsg);
    }
    /// <summary>
    public void AddReceivingMsgThread(string sRxMsg)
    ///
    /// Purpose: update the receive window in the GUI, threadsafe
    /// Version: 1.0: 11-OCT-20: NOS
    /// </summary>
    {
        // InvokeRequired compares the thread IDs
        if (rGuiForm.listBoxRxMessages.InvokeRequired == true)
        {
            try
            {
                AddReceivingMsgCallback rCallback = new AddReceivingMsgCallback(addReceivingMsgT
                rGuiForm.Invoke(rCallback, new Object[] { sRxMsg });
            }
            catch
            {
                // No support, normally at startup or close time
            }
        }
        else
        {
            AddReceiveMsg(sRxMsg);
        }
    }
```

4 : define the event class for the event:

```
/// <summary>
public class UiEventArgs : EventArgs
///
/// Purpose: a public class for event messages for UI publisher/subscribe
/// Version: 14-NOV-20: NOS
/// </summary>
{
    public int iCmd;
    public int iCode;
    public string sMsg;
}
```

5 : define the event handler in the LayerData class:

```
    /// <summary>
    public void publishUiEvent(UiEventArgs eArgs)
    ///
```

```
/// Purpose: publisher event methods for sending events to the UI layer
/// Version: 13-NOV-20: NOS
/// </summary>
{
    if (ePublishUiMsg != null)
    {
        ePublishUiMsg(this, eArgs);
    }
}
```

The souce code snippets for making the either the server socket thread or the client socket thread:

```
/// <summary>
private bool makeSocketConnection(int iSocketMode, int iSockId, string sIpAddress)
///
/// Purpose: make a socket connection, ether server or client
/// Version: 12-NOV-20: NOS
/// </summary>
{
    bool bValid;

    ipAddress = IPAddress.Parse(sIpAddress);
    iSocketPortId = iSockId;
    switch (iSocketMode)
    {
        case (int)Constants.Socket.Server:
            publishEventToUi((int)Constants.UiEventCmd.MsgStatusList,
                "Server waits for clients to connect at IP address " + sIpAddress + "...", i
            iActiveSocketConnection = (int)Constants.Socket.Server;
            /// Start the server thread
            thSocketThread = new Thread(new ThreadStart(this.ServerSocketThread));
            bValid = true;
            break;
        case (int)Constants.Socket.Client:
            publishEventToUi((int)Constants.UiEventCmd.MsgStatusList,
                "Client connects to IP address " + sIpAddress + "...", iSocketMode);
            iActiveSocketConnection = (int)Constants.Socket.Client;
            /// Start the client thread
            thSocketThread = new Thread(new ThreadStart(this.ClientSocketThread));
            bValid = true;
            break;
        default:
            publishEventToUi((int)Constants.UiEventCmd.MsgStatusList,
                "Socket error: Invalid socket mode = " + iSocketMode.ToString() + "!", iSock
            iActiveSocketConnection = (int)Constants.Socket.None;
            bValid = false;
            break;
    }
    thSocketThread.Start();
    publishEventToUi((int)Constants.UiEventCmd.MsgSocketButton, "Disconnect", iActiveSocketC
    return bValid;
}
```

The souce code snippets for the server socket thread, including the receive method:

```
/// <summary>
```

```
public void ServerSocketThread()
///
/// Purpose: thread for handling the server thread
/// Version: 16-NOV-20: NOS
/// </summary>
{
    byte[] baBuffer;
    /// make a TCP connection point based on the IP address and the port number
    tcpListener = new TcpListener(ipAddress, iSocketPortId);
    /// Start the listner connection to let clients to connect
    tcpListener.Start();
    /// wait for a client to connect to this conenction point (port ID)
    tcpClient = tcpListener.AcceptTcpClient();
    /// a client has connected, make a data stream for communicating with the client
    tcpNetworkStream = tcpClient.GetStream();
    /// send a start message
    publishEventToUi((int)Constants.UiEventCmd.MsgStatusList, "A client is connected at " +
        ipAddress.ToString() + "..", 0);
    sendSocketMessage("Server at " + ipAddress.ToString() + " ready for chats ..");
    /// Connected, wait for messages
    while (bRunFlag == true)
    {
        /// wait for receiving a message from the client
        baBuffer = new byte[64];
        if (tcpNetworkStream.Read(baBuffer, 0, 64) > 0)
        {
            receiveSocketMsg(baBuffer);
        }
        Thread.Sleep(100);
    }
    /// close the data stream channel
    if (tcpNetworkStream != null)
    {
        tcpNetworkStream.Close();
    }
    /// close the socket connection
    if (tcpClient != null)
    {
        tcpClient.Close();
    }
}

/// <summary>
private void receiveSocketMsg(byte[] baBuffer)
///
/// Purpose: receiving a data buffer from the socket connection
/// Version: 13-NOV-20: NOS
/// </summary>
{
    string sBuffer;

    sBuffer = Encoding.ASCII.GetString(baBuffer);
    sBuffer = sBuffer.TrimEnd('\0');
    publishEventToUi((int)Constants.UiEventCmd.MsgRxList, sBuffer, 0);
    if (sBuffer.Equals(SOCKET_DISCONNECT_CMD) == true)
    {
        closeSocketConnections();
    }
```

```
}
```

The souce code snippets for the writing method, using the either the server or client socket:

```
/// <summary>
public bool sendSocketMessage(string sSendMsg)
///
/// Purpose: the message to send on an active socket connection
/// Version: 13-NOV-20: NOS
/// </summary>
{
    bool bValid;
    byte[] baBuffer;

    switch (iActiveSocketConnection)
    {
        case (int)Constants.Socket.Server:
        case (int)Constants.Socket.Client:
            if (tcpNetworkStream != null)
            {
                baBuffer = Encoding.ASCII.GetBytes(sSendMsg);
                tcpNetworkStream.Write(baBuffer, 0, baBuffer.Length);
                bValid = true;
            }
            else
            {
                bValid = false;
            }
            break;
        case (int)Constants.Socket.None:
        default:
            bValid = false;
            break;
    }
    return bValid;
}
```

10.4.7 Security

Both the reading and writing functions are just sending plain strings that can easy can be read by other applications, and these applications can also easy change the contents of the strings. Adding any type of security requires a protocol as TCP/IP is only handling data buffers and the receivers must have some sort of protocol for decrypting the information. Applications like Wireshark can very easly get all the data buffers for all TCP/IP connections on your computer.

Chapter 11

Testing

11.1 Introduction

Testing is necessary because human make mistakes, especially if there are several combinations. Some of the mistakes may be unimportant, but some of them may be expensive or even dangerous. It is necessary to check everything and anything because something may always go wrong. If it is not tested, how do we know that it is working? However, we might make the same mistakes when we check our own work as we made when we design/coded it. So should we test our own work?

Yes, the designer/programmer should do the first test, but someother should make the final test of the system. However you should test the work as good as possible to remove as many mistakes as possible. It is cheaper that the designer/programmer test and fix the mistakes, instead of involving other people for doing the testing with more feedback back to the designer/programmer.

Testing is important:

- to point out the defects and errors that were made during the development phases,
- since it makes sure that the customer is satisfied with the application,
- to verify the quality of the application,
- to show that the application is fulfiling the requirements.

Some comments about software testing:

- Testing is a process rather than a single activity,
- Testing is a process that's take place throughout the whole development process and should focus on early testing,
- Static Testing is used to test and find defects without executing code. Static Testing is done during verification process like reviewing and inspection of the documents (including source code),
- Dynamic Testing is executing the software to demonstrate the result of running tests. Dynamic testing can be unit testing, integration testing, and system testing,
- Test Planning to plan the test activities, test progress and be able to report on the status of the software under test,
- Test Preparation to select the test conditions and designing the test cases,
- Test Evaluation to check the results and evaluate the software under test and the completion criteria. Has the software product passed the tests?
- Testing is most effective when it is done early in the development cycle.

Testing is a three step operation, first the unit testing testing all of the units (classes), then integration testing, testing how classes are cooperating for solving a specific operation like an use case, and finally a system testing checking the requirements and use case diagram.

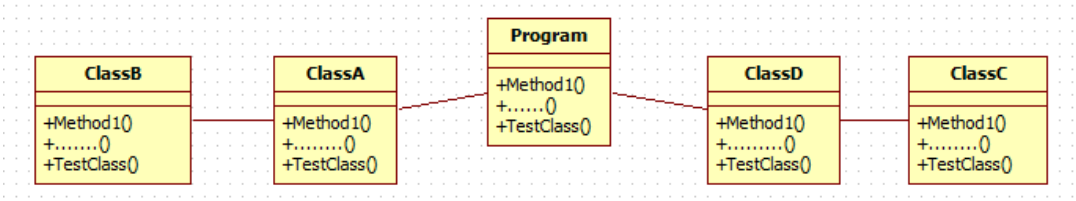


Figure 11.1: A test method in each class will test all the other methods in the specific class.

11.2 Principles

There are several principles of testing. The important principles are:

- testing shows the presence of defects; however testing can not prove that there are no defects. Testing always reduces the number of undiscovered defects remaining in the software,
- exhausting testing is impossible; testing everything including all combinations of inputs and preconditions is not possible,
- early testing; testing should be done as early as possible in the development cycle and focus on the defined objectives,
- change of test cases; repeating the same test cases over and over again will not find any new defects. The test cases should be reviewed regularly to add new or different test cases.

11.3 OOAD based testing

The last operation in each iteration is testing. How to test the use case? The requirements will give an overview description on how the system should work, and the use case diagram and the domain model will be a graphical presentation of the requirements. The Fully Dressed Use Case Document (FDUCD) will describe more in detail how the use case should work, with focus on section #8, the Main Success Scenario, and section #9, the Extenstions. These two sections can be used to make a testplan with the necessary test cases, one test case for the main success scenario, and one test case for each of the extensions. The System Sequence Diagram may also be included to give addional information about the timing of the extensions.

It can also be useful to add a test method in each class, testing all the functionality of that class (unit test). The main class will have this extra test method, and can call all the test methods to test the functonality of all objects at startup time or in a specific test mode (startup option). The test metodes can return either a boolean indication if the test was OK or not, or an integer to indicate which class that was failing. Zero can indicate no error, and a positive number will be a specific class. Any changes to a class later on that will not give a correct results should give an error in the test method.

Figure 11.1 shows this way of testing for an application consisting of five classes. The classes has been analysed and designed according to the design patterns, and a test method, *TestClass()*, has been added to each of the classes. Note that this method should have the same name in all the classes so it is easy to remember the method name.

This test method should test all the methods in a specific class properly. The references already defined in the design step, using design patterns, should be reused. Do not make any new object references for the test methods. In the figure Program object will start the method TestClass() (startup option), it will test all the methods in the Program object. If the return parameter is zero, the TestClass() method in the next object will be called. In this case, object ClassA. When testing of ClassA object is finished, the test method in the ClassB object is called. The return parameter from the test method in ClassB will be returned to the Program object that will continue calling the test method in ClassD object, and so unntil all objects are called, or one of the obejcts are returning an error code. This testing is mainly for unit testing, testing each of the single units. Integration testing and system testing will require some sort of test plans with test cases.

11.4 Test cases

A test document describe the software system under test and the test cases. The meaning with the test document is to describe the testing in such a way that it can be redone with exactly the same results. The test cases should describe the input and output in detail, remember that any should be able to redo the test with exactly the same results. How to test this method?

```
public int CalcIndex(int parameter)
{
   int Index ;

   if (parameter > 16)
      Index = 7;
   else if (parameter > 10)
      Index = 6;
   else if (parameter > 5)
      Index = 2;
   else if (parameter == 4)
      Index = 5;
   else if (parameter == 2)
      Index = 3;
   else if (parameter == 0)
      Index = 1;
   else if (parameter < 0)
      Index = 0;
   else
      Index = 4;

   return Index;
}
```

A set of test cases must be defined. A test case is defined with a number, description, results, and comment, and a test document can consists of hundreds or even thousands of test cases depending on the size and complexity of a system.

A set of test cases for the method:

Id	Description Parameter	Index	OK	Note / Comments
1	100	7		
2	17	7		
3	16	6		
4	11	6		
5	10	2		
6	6	2		
7	5	4		
8	4	5		
9	3	4		
10	2	3		
11	1	4		
12	0	1		
13	−1	0		
14	−100	0		

Testing of methods can be automated by including separate testing methods in each class, and an option can be used when starting up the application for self testing. Automated testing of specific methods in classes are possible, however the testing of several methods working together will complicate the testing. Test Driven Design (TDD) is such a software development process where the tests are designed first, and the software is designed and implemented to fullfil these tests.

Chapter 12

Case Studies

12.1 Introduction

The focus throughout this document is development of software for time dependent systems like Process Control Systems (PCS), communication protocols and real-time system to mention some. Some case studies have been chosen, systems where the reader should be familiar with the problem domain, and can act as both an user and a software developer.

12.2 Mobile Phone

We have chosen the Unified Process as the development process turning the development into smaller iterations. The first phase, Inception, has already been done defining this case study. The next phase, Elaboration, may consists of several iterations, but in this case study there will only be one iteration as this is a minor project. The Elaboration phase will consists of four steps, analysis, design, coding and testing. Since this is the first phase in defining the project, the analysis step will include business use case (if necessary), collecting requirements, making a use case diagram, making a domain model, and making the system sequence diagram. The design step will include defining the layered structure of the architecture and type of user interface. The coding step will include to define development platform, implement the architecture of the application and code the main structure. The testing step will be to test the development platform, architecture and main structure of the application. The first step in the Elaboration phase will be the analysis step.

12.2.1 Analysis

The analysis steps will be to learn more about the problem domain, how is the user going to use the system and the functions that the user wants the system to solve. In the analysis step the user and user functions must be the goal. We will start with the Business Use Case to define the functions that the user wants the system to solve, then we will collect the requirements for a system solving the business use cases. The business use case model defines how the system is going to be used in the business. The business use case for a mobile phone is that you need a device for calling, handling text messages and connect to internet. Adjacent systems will be the mobile operator by the use of a SIM card.

12.2.2 Software development

The development of the mobile phone is used as the example in this document, including most of the diagrams. The steps through the elaboration phase and the first construction phase will be:

	Elaboration	Construction #1
Analysis	Make the BUC, collect the requirements, the use case diagram, make the domain model. Logical architecture structure.	Select the most important use case, make a fully dressed use case document, make a SSD
Design	Define the layered structure of the application, define the development platform and architecture.	use the patterns to make an interaction diagram, make a class and object diagram
Coding	Make the main structure of the application	Implement the interaction diagram
Testing	Test the development platform and the main structure of the application.	Test the implementation of the use case.

12.3 Washing Machine

12.3.1 Introduction

We have chosen the Unified Process as the development process turning the development into smaller iterations. The first phase, Inception, has already been done defining this case study. The next phase, Elaboration, may consists of several iterations, but in this case study there will only be one iteration as this is a minor project. The Elaboration phase will consists of four steps, analysis, design, coding and testing. Since this is the first phase in defining the project, the analysis step will include business use case (if necessary), collecting requirements, making a use case diagram, making a domain model, and making the system sequence diagram. The design step will include defining the layered structure of the architecture and type of user interface. The coding step will include to define development platform, implement the architecture of the application and code the main structure. The testing step will be to test the development platform, architecture and main structure of the application. The first step in the Elaboration phase will be the analysis step.

12.3.2 Elaboration - Analysis

The analysis steps will be to learn more about the problem domain, how is the user going to use the system and the functions that the user wants the system to solve. In the analysis step the user and user functions must be the goal. We will start with the Business Use Case to define the functions that the user wants the system to solve, then we will collect the requirements for a system solving the business use cases. Analyse the logical architecture of the software and evalute the number of layers (tiers).

Business Use Case (BUC)

The business use case model defines how the system is going to be used in the business. The washing machine is going to be used at home and you want to use it in order to clean your cloth. You want the following functions:

1. Rinse: Fill the machine with water, wash, and empty the water,
2. Normal washing: Fill the machine with water, add the detergent, heat the water to 40 °C, wash, and empty the water,
3. Universal washing: Fill the machine with water, add the detergent, heat the water to 60 °C, wash, and empty the water,
4. Dry: turn on the heater and rotate the wash compartment without water.

Requirements

The requirements can be collected in many ways, one way is using the FURPS+ letters for checking and collecting the requirements:

F: Functional: As described in the BUC:

- Rinsing: Fill the machine with water, wash, and empty the water,

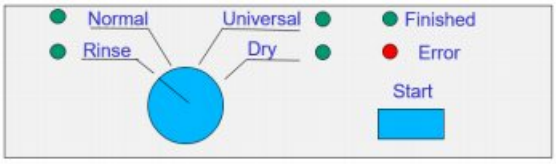


Figure 12.1: An example of the user interface (UI) for the washing machine.

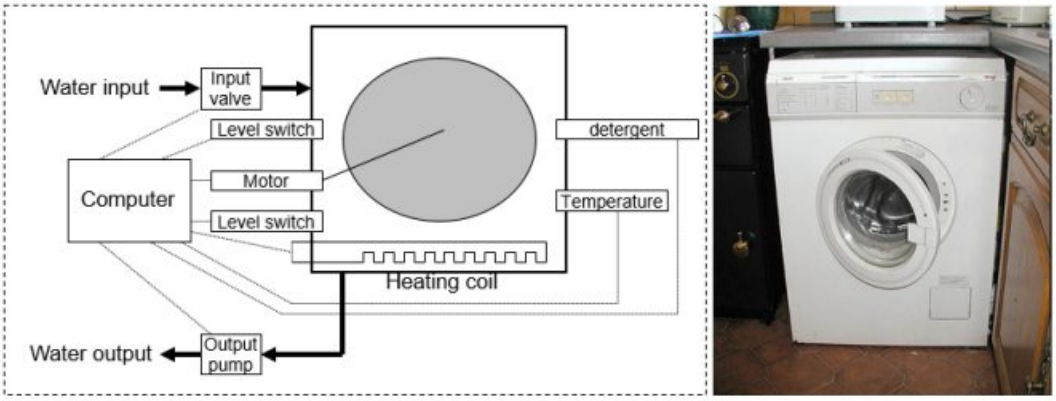


Figure 12.2: An overview of the devices for the washing machine, sort of a deployment diagram. Note that this is not an UML deployment diagram.

- Normal washing: Fill the machine with water, add the detergent, heat the water to 40 °C, wash, and empty the water,
- Universal washing: Fill the machine with water, add the detergent, heat the water to 60 °C, wash, and empty the water,
- Drying: turn on the heater and rotate the washer without water.

U: Usability: No information on how to operate the washing machine. This system will have:

- a switch to select one of the functions,
- a start button to start the system,
- lamps (function lamps) for the functions rinse, normal washing, universal washing, drying, and finished,
- one lamp for error, will be used if the system is stopped, and not finished. The function lamp will indicate in which operation the system was stopped.
- Figure 12.1 shows a sketch of the UI.

R: Reliability:

A deployment sketch can be useful here to get a better overview of the devices that may influence on the reliability of the system. Figure 12.2 shows the devices in the washing machine that must be controlled by the software, a sort of a deployment diagram. Note that this is not an UML deployment diagram as these diagrams focus more on the software modules than the hardware devices.

Based on the deployment sketch, the washing machine system will have the following devices for fulfilling the operations:

- Input valve, binary signal used for open and close the valve. Controls the loading of the water.
- Output pump, binary signal used for open and close the pump. Controls the discharging of the water.
- Level switches, one upper switch and one lower switch, binary signals if covered by liquid or not.

- Motor for turning the container of cloth, controlled by a binary ON/OFF signal.
- Heating coil for heating the water or air inside the washing machine, controlled by a binary ON/OFF signal.
- Temperature sensor for measuring the temperature of the water or air.
- Detergent compartment, released by a binary ON signal. Must be manually filled by the operator.

P: Performance:
The system will have the following time performance requirements:

- Maximum of 60 seconds if no input from the level switch when loading or discharging water into the system,
- Maximum of 240 seconds if no change in temperature when heating the water,
- Running the motor for 600 seconds when rinsing,
- Running the motor for 1200 seconds when washing,
- Heater on for maximum 1200 seconds when drying.

  These time performance requirements will depend on the specification of the I/O devices like the size of water tank, the pumps and valves, and the heating element. To make the software more general will these parameters be part of a configuration part where either the parameters can be tuned, or the I/O device and system specifications can be saved. In this case can the software be extended to handle several washing machine models. A configuration modul is always useful in making the software more general and useful for same type of systems even if they have I/O devices with different specifications.

S: Supportability

- None

+

- What if the door is open when working ? No sensor (actor) for detection !
- Should it be possible to start the system at a specific time?
- Should it be possible to stop the system at any time?

Use case diagram

A use case diagram shall contain the functions from the FURPS+ as use cases (F), and the I/O[1] (URPS) as actors. In this system the user can be an actor, but it is better to define the set of buttons and lamps for interaction with the user as these are defined in the requirement. Note that you should use the same detailing level for all the actors. If you include both a user and a display as actors this will be a conflict since the display is already included as the user actor. An example of an use case diagram is shown in Figure 12.3, the number of actors are reduced using only one type of actor for the valve and pump, and level switches.

Another example of the use case diagram is shown in Figure 6.4, with fewer details.

Domain model

A domain model is based on the specification and should focus on the real "things", not the software structure as you are still in the analysis step, not in the design step. An example of the domain model for the washing machine is shown in Figure 12.4. Note that this diagram should not show the use cases from the use case diagram as the use cases are functions or operations, not "things".

The architecture will be a two layered structure, separating the user interface and the business layer. As this will be an emebdded device, the data will be integrated into the business layer. The separation is done as the user interface may vary between different versions of washing machines, while the business logic should be the same for all versions. Logging may be a future requirement giving that a three layered architecture may also be an option.

[1] I/O: Input/Output devices

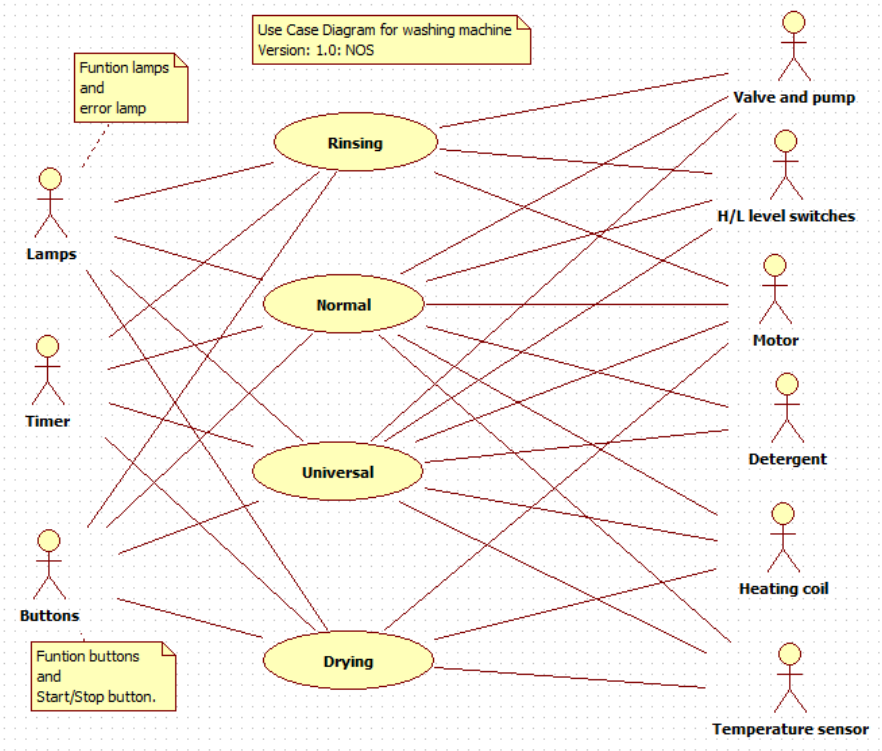


Figure 12.3: An example of a use case diagram for the washine machine, based on the FURPS+ requirements.

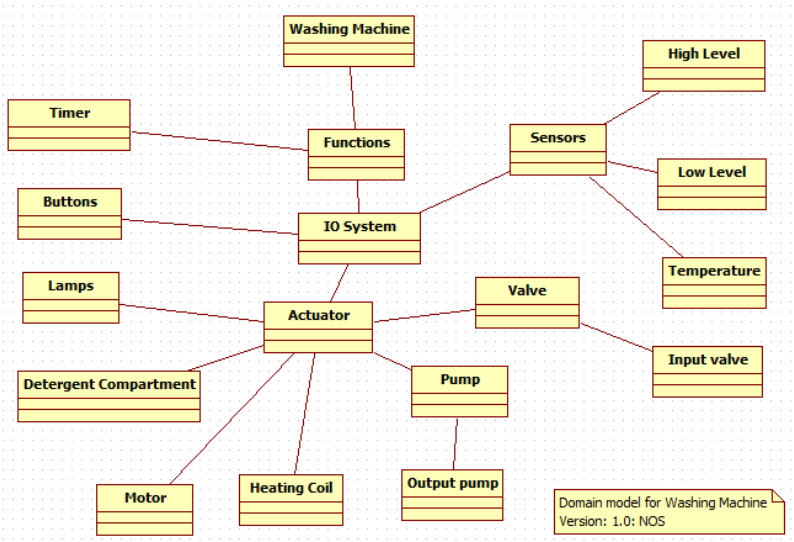


Figure 12.4: An example of a domain model for the washing machine, with focus on the real "things" in the system.

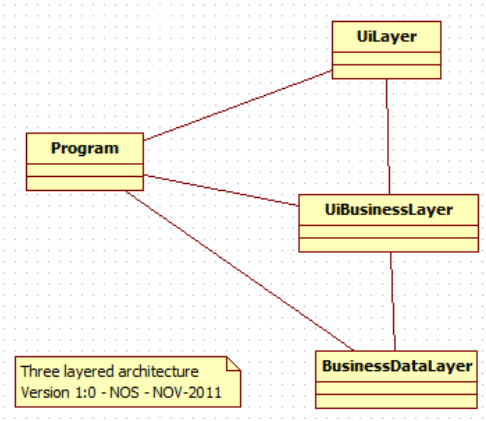


Figure 12.5: An example of a class diagram showing the selected layered architecture of the application.

Elaboration: Design

In the elaboration phase most of the work should be in the analysis step collecting the requirements, making the use case diagram and the domain model. In the design step you should define the main architecture, first define if using one, two, three or four layer architecture. The number of layers will depend on the usage, the data storage and the user interface. You should at least have a business layer where you have the specific logic for the application. An user interface layer can be very useful for separating the user interaction coding and the business logic, indicating that you should at least have a two layer architecture. If you are not sure about how to load and store the information, using either text files, csv (comma separated values) files, database, internet, or cloud based, a data layer can be useful. So normally a three layer architecture is an useful approach. Making a web based application an extra layer for web browser and operating system support can be useful. So the decision will be to use either a two, a three or a four layer architecture for your application. In the washing machine system I will use a three layer architecture due to many different I/O devices.

The first version of the class diagram will just be the layered architecture, in this case a three layered architecture. The class diagram is shown in Figure 12.5.

The test application is developed as a graphical application and the event loop in the user interface is the main loop of the application. The UiBusinessLayer class is the interface between the user interface (UI) and the business layer, and the BusinessDataLayer class is the interface between the business layer and the data layer. These objects will be created first and the references are given to the UiLayer object that can access the business layer and the data layer through these object references. The layered architecture will be covered to some extend in a later iteration, including the usage of the facade design pattern.

A graphical application in Visual Studio is based on events from the user interface and should be the last object to create as this one will handle the event in the application. The main() method needs to be static as the method needs to present before any objects are created in order to know where to start. This is the reason for the program class to be static, containing the static main method. To avoid challenges with static classes and methods, let the main() method just create a new class for making the layer classes. Start with making the data layer class, then the business layer class including a reference to the data layer class, and then finally the UI layer class including a reference to the business later class. The UI layer object will now have an object reference for accessing the business layer object, and the business layer object will have the object reference for accessing the data layer. One way of designing these two classes for a Windows MFC application is listed here.

```
///
static class program
///
/// Purpose: main start of the application
{
    /// <summary>
    /// The main entry point for the application.
```

```
    /// </summary>
    [STAThread]
    static void Main(string[] Args)
    {
        programStart pStart ;

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        pStart = new programStart(Args) ;
    }
}
///
public class programStart
///
/// Purpose: three layer generation of the application
{
    private businessLayerFacade rBusinessLayerFacade;
    private dataLayerFacade rDataLayerFacade;
    ///
    public programStart(string[] Args)
    ///
    /// Purpose: startup of the application, make a three layer structure
    {
        /// Start the application by init the business layer and
        /// data layer before the GUI layer
        rDataLayerFacade = new dataLayerFacade();
        rBusinessLayerFacade = new businessLayerFacade(rDataLayerFacade);
        /// try to catch all exceptions from the application
        try
        {
            /// Load the data after starting the GUI
            Application.Run(new FormGUI(rBusinessLayerFacade));
        }
        catch (Exception e)
        {
            MessageBox.Show("Application Error: " + e.Message + " (Info:" +
                 e.Data.ToString() + ")" + " (" + e.StackTrace + ")");
        }
    }
}
```

The create design pattern defines that the object that is going to use an object should create the object. In this case will the Program object use all the objects, but the UILayer will also use the UiBusinessLayer and BusinessDataLayer objects. The Program object will create the other objects, however the object references must be copied to UiLayer object as method parameters.

Elaboration - Coding

In the elaboration phase the development platform and the development tools will be tested out for the project. In this project we will first make a simulator of the washing machine software using the Visual Studio, and the user interface from the tool box of Visual Studio. This is also a good reason why to have a separate UI layer. The coding will be:

```
    /// <summary>
    static class program
    ///
    /// Purpose: main start of the application
```

```
///
/// Version: 1.0: 15-MAY-12: NOS
/// </summary>
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] Args)
    {
        programStart p_start ;

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        p_start = new programStart(Args) ;
    }
}
/// <summary>
public class programStart
///
/// Purpose: three layer generation of the application
///
/// Version: 1.0: 15-MAY-12: NOS
/// </summary>
{
    private UiBusinessLayer rUiBusinessLayer;
    private BusinessDataLayer rBusinessDataLayer;
    /// <summary>
    public programStart(string[] Args)
    ///
    /// Purpose: startup of the application, make a three layer structure
    ///
    /// Verison: 1.0: 2-JAN-14: NOS
    /// </summary>
    {
        // Start the application by init the business layer and
        // data layer before the GUI layer
        rUiBusinessLayer = new UiBusinessLayer();
        rBusinessDataLayer = new BusinessDataLayer();
        // try to catch exceptions from the application
        try
        {
            // Load the data after starting the GUI
            Application.Run(new UiLayer(rUiBusinessLayer,
                                        rBusinessDataLayer));
        }
        catch (Exception e)
        {
            MessageBox.Show("Application Error: " + e.Message +
                " (Info:" + e.Data.ToString() + ")" +
                " (" + e.StackTrace + ")");
        }
    }
}
```

Note that an extra class is created for the Visual Studio code as the first class must be static, meaning that an object is created automatically when starting the application. To avoid more static classes, a new class, programStart, is created before making the layered structure. The GUI form will contain

the Windows event loop handling the rest of the application. The initialization of the datalayerFacade object will also load any initialization data from a XML setup file. Note that the object references from the business layer and data layer is included as parameters when creating the UILayer object.

Elaboration - Testing

In the elaboration phase the development platform and the development tools will be tested out for the project, together with the layered architecture of the application. The UI prototype should also be part of the analysis step in the elaboration and can be implemented and tested as well in this iteration.

12.3.3 Construction - Analysis

Start the analysis phase with selecting the most important use case. If you are not sure, take the one that you think are most difficult or ask the user if he/she can have only one of the functions which function do he/she prefer. Now we need to analyze this specific use case in more detail and this done by making a fully dressed use case document (FDUCD). We will select the normal washing use case. We should try to make a set of sub actions of the use case, and evaluate any error condition for each of sub action. Note that these sub actions are mainly for interaction with the system, do not focus so much of the internal working of the system. That will be taken care of in the design step. The fully dressed use case document can be:

1	Use case name	Normal washing
2	Scope	Washing machine
3	Level	User goal
4	Primary actor(s)	User
5	Stakeholders and interest	
6	Precondition	No water present, select normal washing function, use start button,
7	Success Guarantee	Washed the cloths
8	Main Success Scenario	1: Start the system using the start button
		2: Turn off error and finish indicator
		3: Turn on the washing indicator
		4: Check for no water in the washing machine
		5: Open inlet valve for filling water
		6: Wait for level full signal
		7: Close input valve, washing machine is filled with water
		8: Open washing detergent compartment
		9: Turn on heater
		10: Wait for correct water temperature for normal washing
		11: Turn off heater
		12: Start washing motor
		13: Wait for washing period
		14: Stop washing motor
		15: Start output pump for discharging the water
		16: Wait for the level empty signal
		17: Stop the output pump
		18: Turn off the washing indicator
		19: Turn on the finishing indicator
		20: Stop the system

9	Extensions	3a: if water in the washing machine 3b: start output pump for discharging the water 3c: wait for the level empty signal 3d: stop the output pump 3e: goto 4 3ca: if timeout set the error lamp 3cb: stop the output pump 3cc: goto 20: stop the system 5a: if timeout set error lamp 5b: stop the inlet pump 5c: goto 20: stop the system 10a: if timeout set error lamp 10b: turn of the heater 10c: goto 20: stop the system 16a: if timeout set the error lamp 16b: stop the output pump 16c: goto 20: stop the system
10	Special Requirements	None
11	Technology and Data Variations List	None
12	Frequency of Occurrence	Selected by user
13	Miscellaneous	Should end with a rinse operation?

The last step in the analysis step is to make a System Sequence Diagram (SSD). Note that you are still in the analysis steps and the focus is to understand how the system is going to be used and this is the reason why this is called a system sequence diagram. The focus is the interaction with the system and the you should implement the FDUCD section 8 and section 9 into a sequence diagram with time as the reference. A SSD can be useful if you do not have enough experience in how to transform the information in the FDUCD into an interaction diagram which is the goal in the design step. As long as this information in the SSD is useful for you, make the SSD. When you feel that making the SSD is just a burden, that you get all the information you need from the FDUCD you can avoid this diagram. However, if you are not certain, make the SSD. This is still in the analyzing step and making a SSD can be a good way to get more knowledge about both the system and this specific use case. In this case you should make a SSD and an example of a SDD with the first 8 steps is shown in Figure 12.6 giving an outline for making the SSD. The final SSD must contain all the steps from the FDUCD but can be divided into several parts as it may take a lot of both horizontal and vertical space.

The SSD should give a better overview of the sequence of the operations performed by the software application, and the order of these operations.

12.3.4 Construction - Design

We will use the design patters as the helping tool for making the design models. The design models consist of an interaction diagram to show the dynamic information of the use case and a class diagram to show the static information of the use case. You should start with making an interaction diagram, and select either a sequence or a collaboration diagram. In this case time is important so we will choose a sequence diagram. The next step is to use design patterns as the helping tool for defining useful classes. The first design pattern to use is the Controller Pattern to indicate the class that will be responsible for the use case. An advice is to use the same name on the class as the use case to let it be part of the documentation. Later on you know that you just have to find the class with the same name as the use case to start checking out the code for a specific use case.

Using the Controller Pattern you will have you first class. Then you should take step for step in your FDUCD, and figure out how to implement each of these steps. In this process the SSD can be useful as you have already tried to combine section 8 (main success scenario) and 9 (extensions) in the SSD. In the SSD the focus was on the interaction with the system, in the design sequence diagram the focus is the coding of the application. This means that the design sequence diagram (DSD) should contain a lot more details about the method names, return parameters, variables and loops.

By taking step for step in the FDUCD you should use the design patterns, try to assign responsibilities to the classes, and let the name of the class reflect the responsibility of the class. A good advice is to

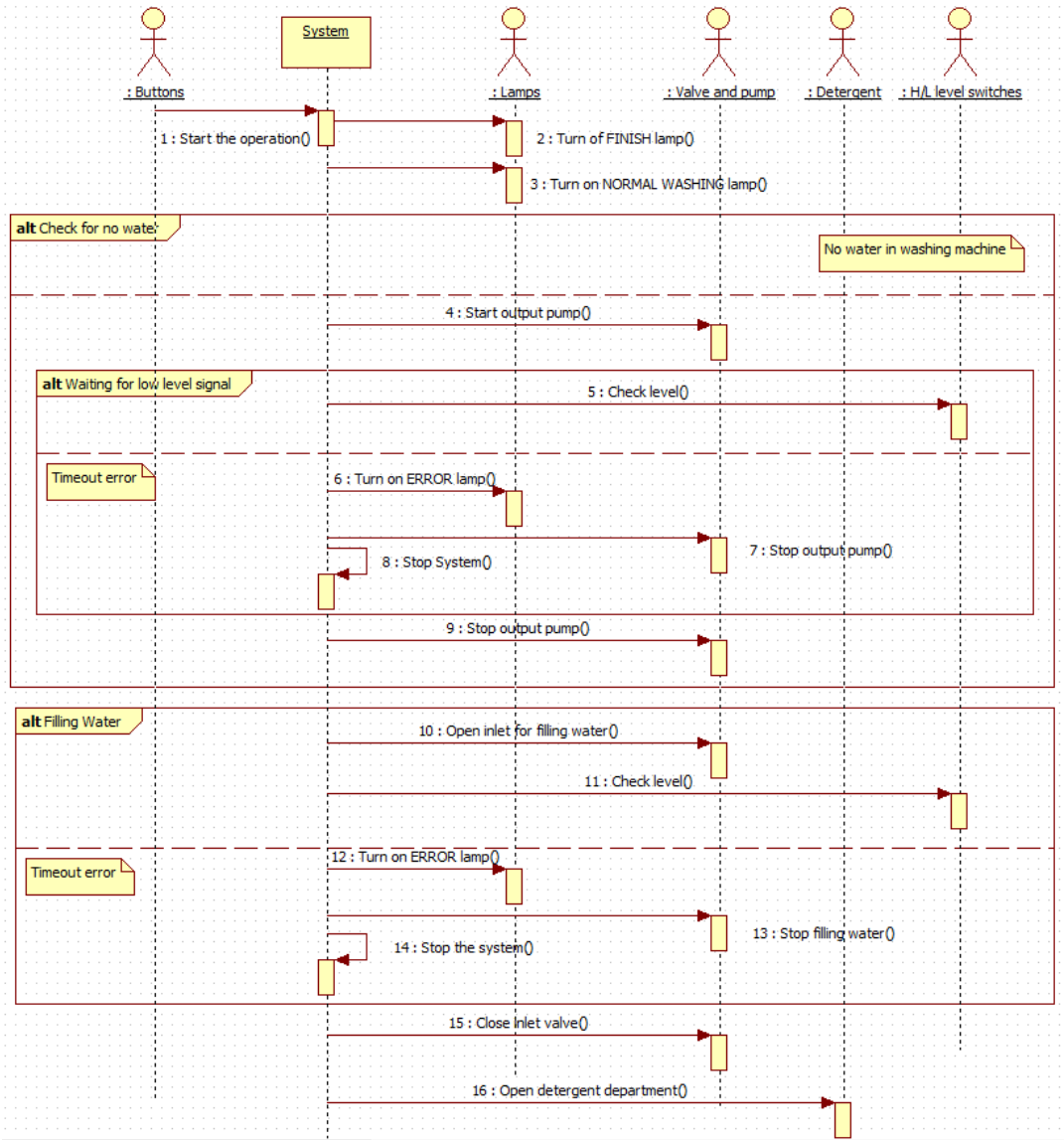


Figure 12.6: The first steps of the SSD for the Normal Washing use case. Focus on the communication between the system and the actors. The SSD is only a graphical representation of the FDUCD.

use the domain model as well when looking for any classes to use. A suggestion for the classes:

Step	Description	Design Pattern	Class Name
1	The first class, controlling the use case	Controller	Washing
2	Turn off indicator, responsibility for any lamps	Information Expert	Lamp
3	Turn on indicator, responsibility for any lamps	Information Expert	Lamp
4	Read level sensor, responsibility for any sensors	Information Expert	Sensor
5	Emptying and filling water, responsibility for these operations	Information Expert High Cohesion	WaterOperation
6	Control the valves and motors	Information Expert	Actuator
7	Handling timing periods, responsibility in a setup database	Information Expert High Cohesion	Config
8	Handling temperature, responsibility in a setup database	Information Expert High Cohesion	Config
9	Handling the motor	Information Expert	Actuator
10	...		

In this case you already have a three layered architecture structure that you should include when making the sequence diagram. As you have not yet learned about the facade design pattern these object will be skipped in this iteration. The user will start the operation by pressing the START button so the request will start from the UiLayer object. The focus will mainly be the business layer and data layer so the UiLayer object will have the responsibility for sending the start button event to the business layer. This event will need the type of operation, and using the information expert pattern we are assigning that responsibility to a button class. The button class will be responsible for getting the type of operation and start the use case operation, in this case the start() method in the washing class. According to the FDUCD and SSD the next operations will be to turn off the error and finish lamps because one of these may be on after the previous operation. A lamp class is assigned this responsibility based on the information expert pattern, and partly on the high cohesion pattern. The next operation is checking if any water in washing machine and empty the washing machine in that case. Since we are having both the empty and fill operation of water in the use case are we using the high cohesion pattern to make a class that will be responsible for the water operations, named wateroperation. The first operation will be to empty the machine, and a DischargeWater() method will be used. This method will check for an active signal from the low level signal, and start the output pump if not an active signal. The operation will run the output pump until the water is empty or a time-out, in both cases will the output pump be stopped. If a time-out will the error lamp be turned on and the system will stop. If the water tank is empty, the next operation, filling the water will start. The design, using design pattern based on the FDUCD and the SSD is shown in Figure 12.7. This diagram is showing only the first part of the design of the use case, but should give better idea of how to make a design sequence diagram.

Note the difference between Figure 12.6 being the SSD with focus on the interaction between the software and the actors, and the 12.7 being the design sequence diagram with focus on the classes inside the software and interaction between these classes. The design diagram must contain much more details than the SSD.

The focus has been the controller pattern, the information expert pattern and the high cohesion pattern so far, let us discuss the low coupling design pattern. Looking at Figure 12.6 shows that a number of objects are used and that the number of arrows between the objects only goes one way (between two and two objects), and that the number of arrows are minimized. This can also be checked in the next step, making the class diagram. The last design pattern in this iteration is the create pattern. It can be a good advice to make a separate interaction diagram for creating the objects, and making this diagram after making the class diagram as the associations between the classes are better indicated in the class diagram.

The nest step will be to make the class and object diagram where the class is the abstract model of your software while the object diagram is implemented model deployed in the computer memory. As long as you have only one object of each class these two diagrams will be the same. In this case we have the same number of classes and objects and will make only the class diagram. All the information that you need is already present in Figure 12.7, you have the classes, the main methods, the main parameters and the associations between the classes. You should have one interaction diagram for each use case, but only one class diagram. This means that you should extend the class diagram from the elaboration phase. Looking at Figure 12.7 you have all the classes on the top, all the classes must be included in

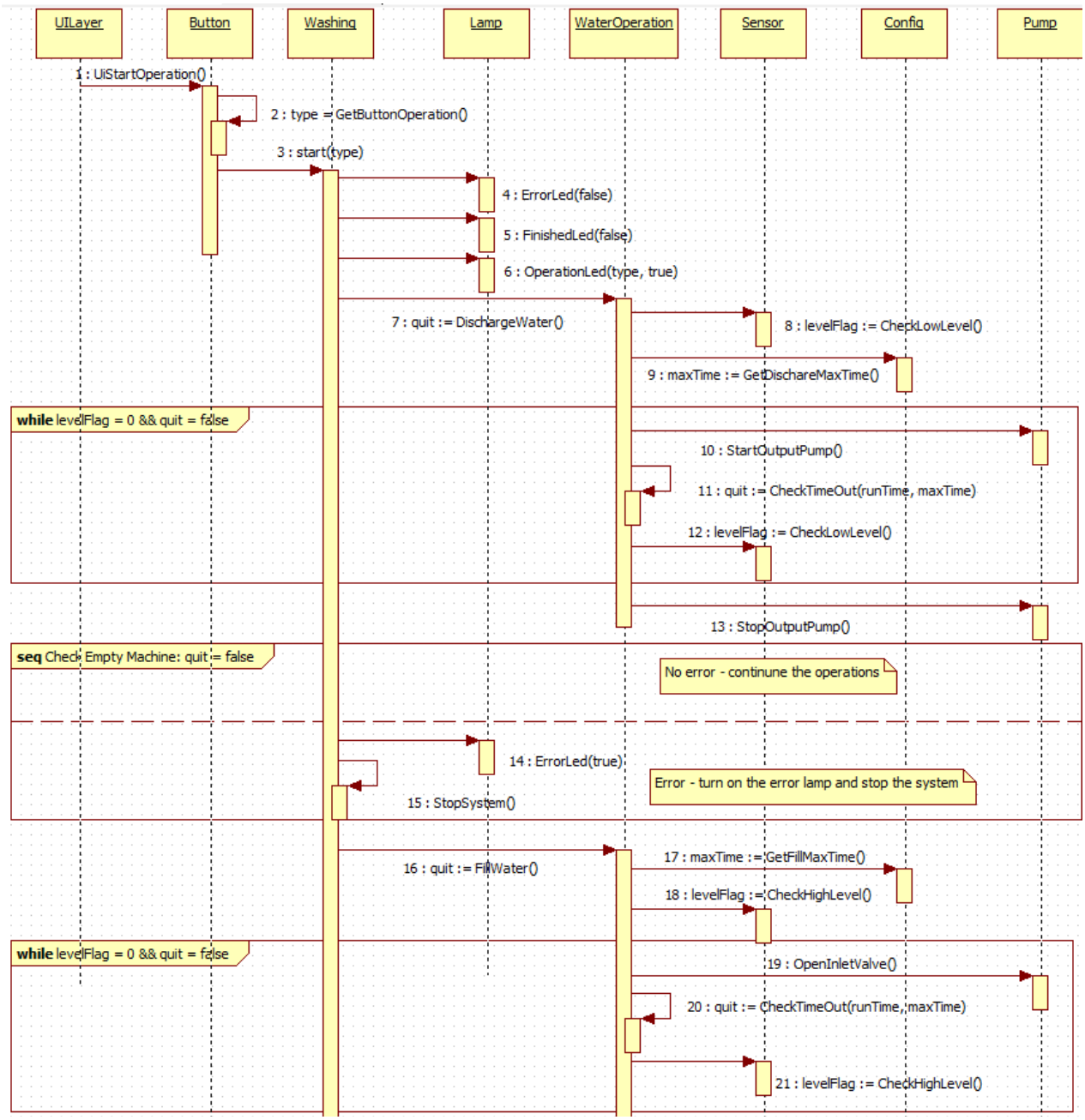


Figure 12.7: The first part of the design sequence diagram for the Normal washing use case. The sequence diagram is based on the FDUCD, SSD and design patterns.

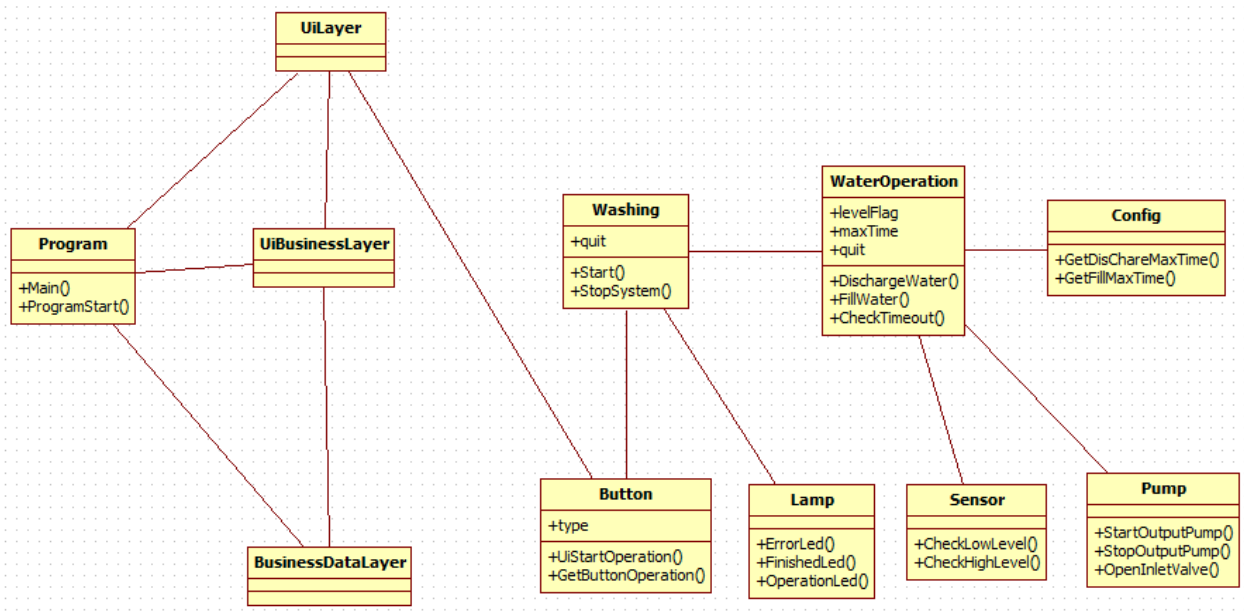


Figure 12.8: The class diagram after the first use case is designed.

the class and object diagrams. Next take step for step in the interaction diagram, each step should be a method in a class. The arrow will point to the class where the method belong, the arrow starts from the class where the return parameter, if any, belong, and the arrow indicate the association between the classes. The new class diagram, based on Figure 12.7, is shown in Figure 12.8. Note that Figure 12.7 only has designed the first part of the use case, and the class diagram will reflect only this design.

Looking at the class diagram in Figure 12.8 with reference to the create pattern is to let the UiBusinessLayer to create the Washing, WaterOperations and Config classes, while the BusinessDataLayer should create the Button, Lamp, Sensor and Pump classes. The reasons are that these objects have the initialization data, closely uses, and to some extend aggregates these objects. This is also shown in the class diagram as a layered structure where Washing, WaterOperation and Config belongs to the business slayer, while Button, Lamp, Sensor and Pump belongs to the data layer. A design sequence diagram for making the classes is shown in Figure 12.9. This diagram should be the first part of the use case as the objects must be created before they can be used in the use case. In this case will all the classes be used by several of the use cases and created by the layer classes when starting up the system.

Creating the objects include more associations between the objects as the layer objects is creating most of the objects. This will add more associations to the class diagram and an updated class diagram is shown in Figure 12.10. As this Figure shows will the coupling be higher when including create design pattern but this is normal as part of the responsibility of the objects and the use of the objects in the system.

12.3.5 Construction - Coding

You should use the fully dressed use case diagram, the interaction diagram and the class diagram to implement the use case, the main diagram will be the interaction diagram. Do not include too much details into these diagram as it will be difficult to synchronize the code and the diagram. Let the diagram contain the main structure and the main variables, and keep the temporary methods and variables out of the diagram. The main reason for the diagram is to document the structure and easy be able to figure out in which class and which method(s) to look for an error or an extension. If the main structure is changed during the construction step, which happens from time to time, please remember to update the diagram immediately!

12.3.6 Construction - Testing

You should use the requirements, use case diagram and the fully dressed use case diagram for testing the use case. The testing is for checking that the use case is working the way that you have analyzed,

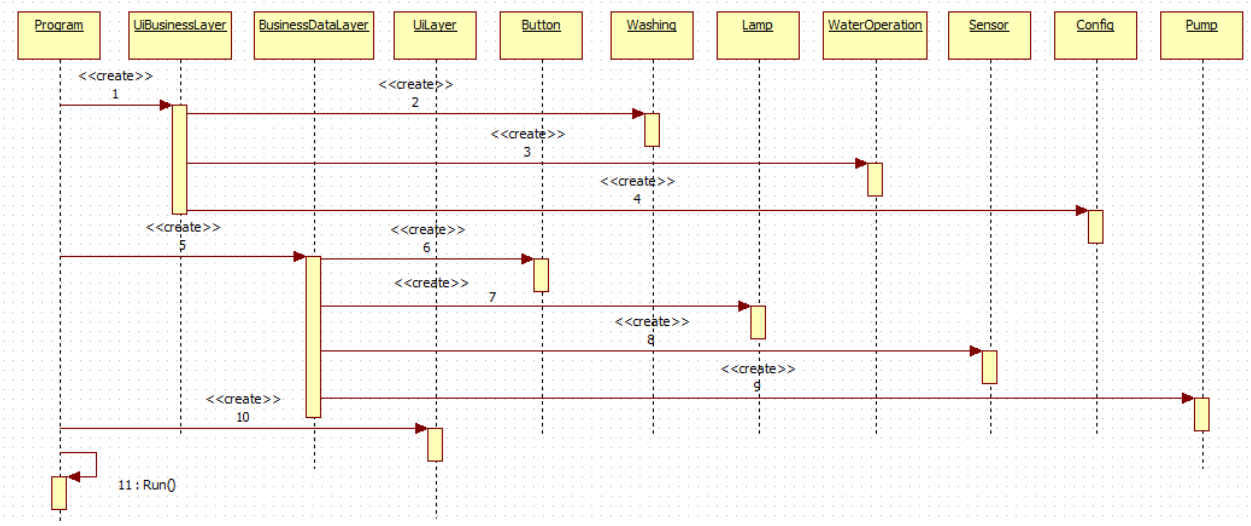


Figure 12.9: The create part of the use case, this part must be implemented before the use case implementation.

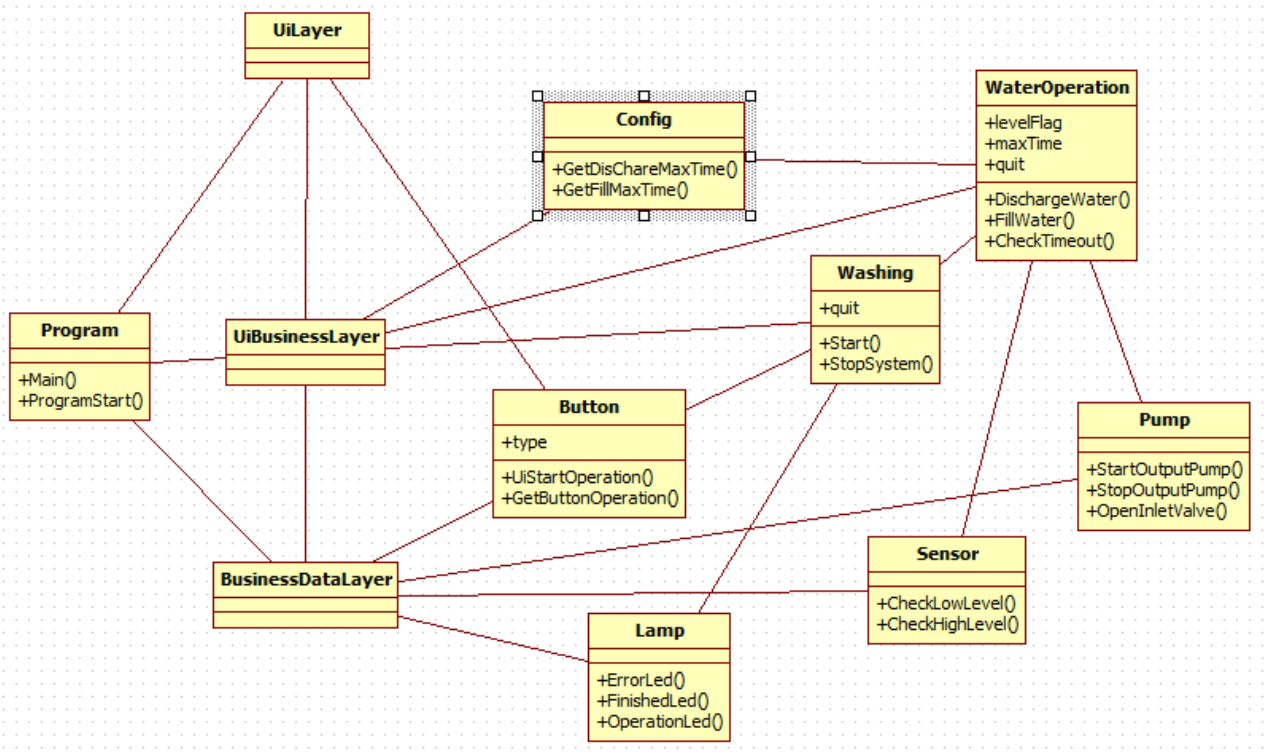


Figure 12.10: The class diagram with the associations for creating all the objects.

and implemented the correct way during the design step. Make a set of test cases for testing the steps in the fully dressed use case document.

Chapter 13

Summary

The elaboration phase and first construction phase is finished and according to the UP the following operations, diagrams and documents are implemented. The elaboration phase has implemented the following operations, diagrams and documents:

- Analysis steps making the use case diagram and the domain model diagram,
- Design steps defining the tier architecture of the application, see section 8.1.1,
- Implement the tier architecture based on the domain model by making the root object, and the main objects for each of the tiers. Part of implementing the tier architecture in the application can be to make a prototype of the user interface as part of checking the collection of requirements,
- Make a overview test plan based on the use case diagram. A tes plan must contain a system description and a set of test cases.

The first construction phase has implemented the following operations, diagrams and documents:

- Analysis steps in selecting the most important use case, and make a fully dressed use case document (FDUCD) for this use case. Remember that this is the analysis phase and making the FDUCD should make you understand the process described with this use case. This means that the main success scenario of the FDUCD should explain in much details how the use case is going to perform or operate. This section must contain enough information that use case can be designed as an software application in the next step.
- Design steps means using design patterns to design the software as an object oriented application by using an interaction diagram. Select either a sequence diagram or a collaboration diagram as the interaction diagram. Based on the interaction diagram make or extend the class and object diagrams.
- Implementation steps by implementing the use case and extending the application based on the tier architecture that was selected in the elaboration phase. Use threading to make the application more scalable and include exceptions to be able to handle system warnings or errors.
- Test the use case based on the requirements, use case diagram and the FDUCD.

Then we will be ready for the next construction iteration.

Part III

Construction Phase - Iteration #3 (CP-I2)

Chapter 14

Introduction

The iterations have been one in the elaboration phase (introduction) and three in the construction phase as shown in the table:

	Elab. Phase	**Construction Phases**	**(CP)**
	EP	CP#1	CP#2
Iteration	#1	#2	#3
Analysis	Intro Dev. Process Ver. control	Collect requirements, Use case diagram, Domain model, Use case documents, System sequence diagrams,	Include and extend use cases, Generalizations and Specialization,
Design	Intro Dev. Process Ver. control	GRASP patterns, Sequence diagrams, Class Diagrams, Object diagrams,	GRASP pattern, GOF pattern, State diagram, Activity diagram, Data structures Algorithms
Coding	C# coding Version control classes	Exceptions, Threading,	Inheritance, Software development,
Testing	Debugging	Testing.	Testing,

This second construction phase (CP) iteration, iteration #3, will include the following extensions to the analysis, design and programming phases:

1. Analysis phase
   (a) Use case: Use of *include* and *extend* in use cases,
   (b) Domain model: *Generalization* and *specialization* for classes (inheritance),
   (c) Domain model: Package diagrams,
   (d) Architectural Analysis; first iteration.
2. Design phase
   (a) More patterns,
   (b) State diagrams,
   (c) Activity diagrams,
   (d) Architectural analysis,

3. Programming phase:
   (a) Inheritance in objects,
   (b) Software development tasks.

Chapter 15

Analysis

Use cases are an important concept in both programming analysis and system analysis, defining the functionality that the users want out of the system. Use cases are not part of the object oriented analysis, but part of general analysis. Use case diagrams and use case documents can be extended with more information regarding the use cases. This section will deal with the stereotypes *include* and *extend*.

15.1 Relating Use cases

The stereotypes *include* and *extend* can be added to the use case diagram and the use case documents, in our case the fully dressed use case document (FDUCD). The *include* stereotype is the most common and important relationship.

15.1.1 include

The *include* stereotype is a way of combining information between use cases in a use case diagram and use case document.

1. shows a function or behavior that is common across several use cases,
2. Use *include* in use case diagram and use case document to avoid duplication of code,
3. Add *include* in the Extension section of the use case document
   (a) Extension: Function_name: Include Use_case_name
4. The include use case will have Subfunction as level in the use case document,
5. Can also use include if the main function (Main Success Scenario) is very long and/or complex,
6. Add as stereotypes in the use case diagram, shown as <<include>> in the diagram (Older UML tools are using <<uses>>).

Using the include stereotype is also a way of start thinking of common sub function in the system, getting more knowledge about the functionality of the system.

Use case diagram

The stereotypes are given as <<include>> and <<extend>> in the UML use case diagram. This section shows how these stereotypes can be added in some UML tools. Figure 15.1 shows a use case diagram with include in StarUML. The Figure also shows how the include and extend stereotypes can be selected from the tool bar (the ring on the left side).

Figure 15.2 shows the usage of include in Rational Rose when using the common use case, ReleaseKeyboard.

Figure 15.3 shows how to add the include stereotype to the association in Rational Rose. Starts with right click on the association and select *specification* to get this menu. Also check that *Show stereotype* is turned ON in the pull-down menu.

Visual Studio is working in the same way as the other tools. Select the Use Case diagram and the <<include>> and <<extent>> associations are available in the toolbox. See Figure 15.1 and 15.2.

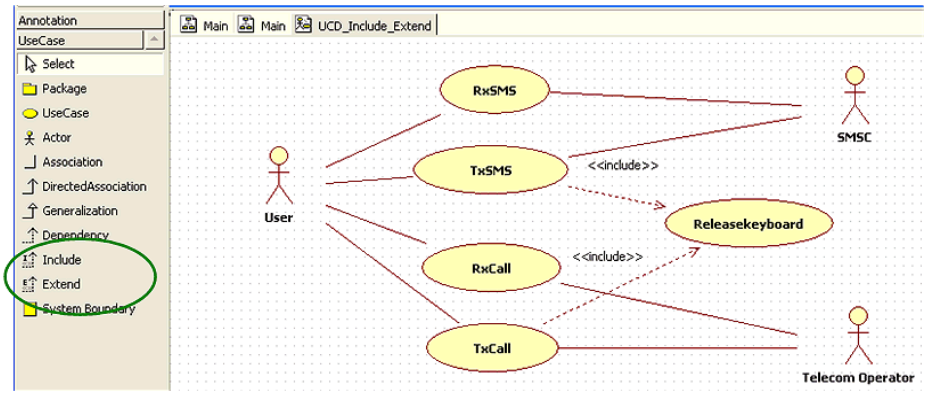


Figure 15.1: The usage of the include and extend stereotypes in the use case diagram for StarUML.

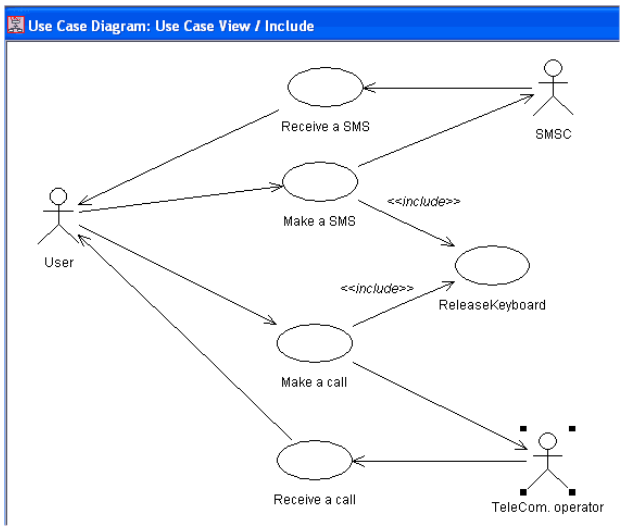


Figure 15.2: Using <<include>> for a "subfunction" in a use case diagram.

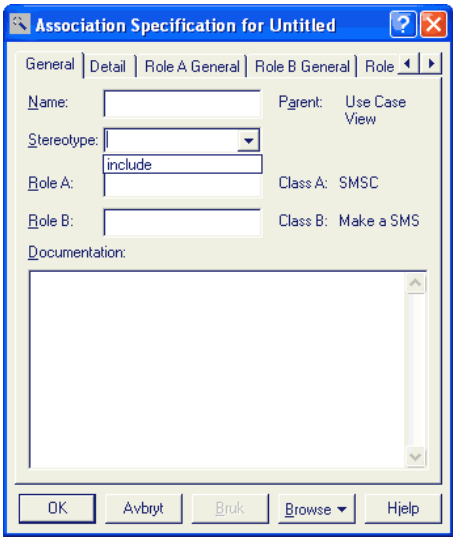


Figure 15.3: Right click on the association, pick *specification*, and select the include stereotype (or input if non-exist).

Fully dressed use case document

Part of the fully dressed use case for making a SMS message:

Use case section	Comment
Use case name	Make a SMS
Scope	Mobile phone
Level	User Goal
..	..
Main success scenario	1: Open keyboard
	2: Make the Message
	3: ..
Extensions	1a: Open keyboard: Include Release keyboard

Part of the fully dressed use case for release the keyboard

Use case section	Comment
Use case name	Release keyboard
Scope	Mobile phone
Level	sub function
..	..
Main success scenario	1: Display keyboard code
	2: ..

The stereotypes *include* and *extend* can be added to the use case diagram and the use case documents, in our case the fully dressed use case document.

15.1.2 extend

The *extend* stereotype is a way of extending information in a use case diagram and use case document without changing the current functionality.

1. can be used when extending an use case, but don't want to change/extend the main success scenario section of the use case document (the use case is already implemented, approved, etc.),
2. Add the line *Extension Points* before the Main Success Scenario (MSS) in the use case document, and define which step(s) in the Main Success Scenario to be extended.
3. Make a new use case document for this Extension, and add the extra lines:
   (a) Trigger: What is the "trigger" for the extended function,
   (b) Extension Points: Define the point and use case document to be extended,
4. Set level as sub function (just like the include stereotype).

Use case diagram

Figure 15.4 shows the usage of extend when extending the *Make a SMS* function.

Fully dressed use case document

Part of the fully dressed use case document for making a SMS message:

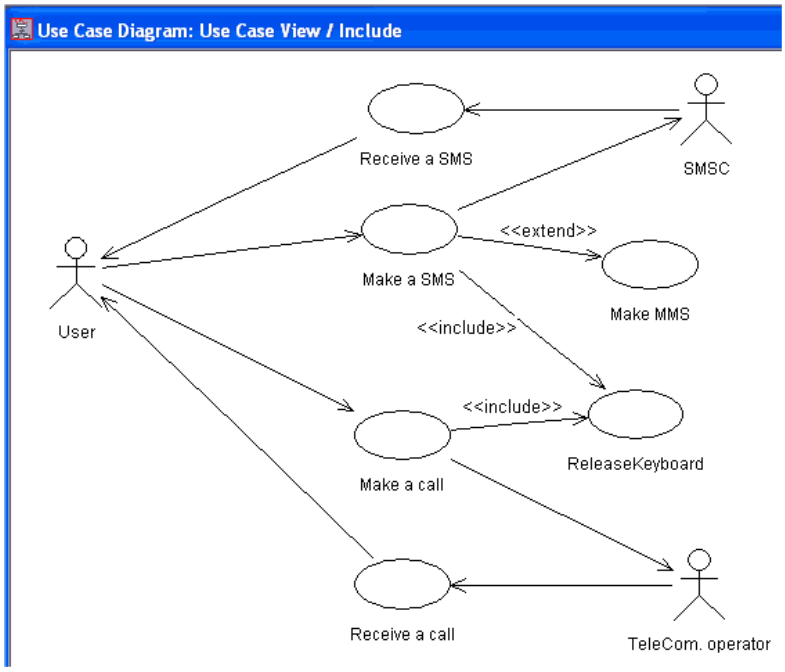


Figure 15.4: Using <<extend>> for extending a function in the use case diagram.

Use case section	Comment
Use case name	Make a SMS
Scope	Mobile phone
Level	User Goal
..	..
Extension Points	Make the message, step 2.
Main success scenario	1: Open keyboard
	2: Make the Message
	3: ..
Extensions	1a: Open keyboard: Include Release keyboard

Part of the fully dressed use case for the MMS message

Use case section	Comment
Use case name	Make MMS
Scope	Mobile phone
Trigger	User wants to add sound and/or picture
Extension Points	Make the message in Make a SMS
Level	sub function
..	..
Main success scenario	1: Options for including sound or picture in the message
	2: ..

15.2 Domain Model Refinement

Using the domain model to find

1. class hierarchy as basis for the software class hierarchy,
2. define superclasses and subclasses,

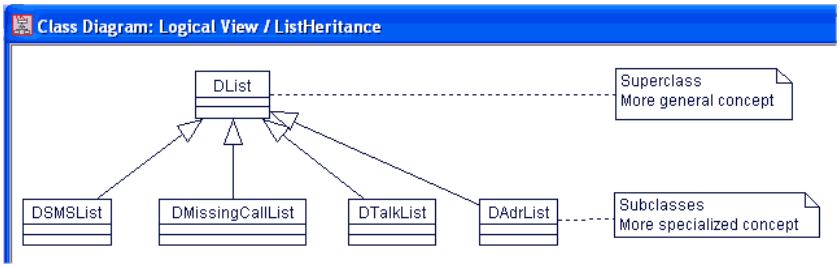


Figure 15.5: Superclass and subclasses for the lists in the mobile phone.

(a) superclass; more general concept: sensor class for temperature and pressure sensors,

(b) subclass; more specialized concept: temperature and pressure sensor classes.

3. can be used for inheritance.

See Figure 15.5 and 15.6 for superclasses and subclasses with the UML syntax. The Figure shows the usage of lists in a mobile phone having the SMS list, the missing call list, the talk (called) list and the address list. A list has a set of common methods that will be collected and implemented in the superclass DList, while the specific methods for each type of lists will be implemented in each subclass.

The implementation will require more time for design and coding, but the maintenance time by having many of the methods as common methods for all the lists will decreased.

15.2.1 Guidelines

Super- and subclasses are used when several classes have identical or almost identical methods or part of methods. The identical parts should not implemented in different classes, but should be collected in a common superclass that can be used by the subclasses.

1. Identify domain superclasses and subclasses to the current iteration, and illustrate them in the Domain model,
2. 100% rule: 100% of the conceptual superclass's definition (attributes and methods) should be applicable to the subclass,
3. Is-a rule: All members of a subclass set must be members of their superclass set (Test: subclass is a superclass),
4. Super classes are also called base classes in some documents,
5. Start implementing the superclass with the most obvious methods based on the domain model, then the subclasses. When implementing more subclasses check for more common methods and move these methods into the superclass,
6. Avoid too many levels of superclasses.

15.2.2 Motivation for superclass

Making a generalization into a common superclass when:

1. the potential conceptual subclasses represent variations of a similar concept,
2. the subclass will confirm to the 100% and Is-a rules,
3. all subclasses have the same attribute as the superclass,
4. all subclasses have the same association as the superclass.

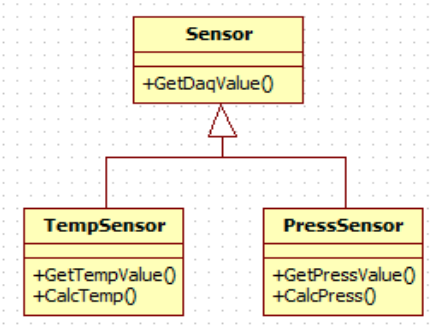


Figure 15.6: The sensor superclass.

15.2.3 Motivation for subclasses

Partition a class into subclasses when:

1. the subclass has additional attributes (of interest),
2. the subclass has additional association (of interest),
3. the subclass concept is operated on, handled, reacted to, or manipulated differently than the superclass or other subclasses (in ways that are of interest),
4. the subclass concept represents an animate thing that behaves differently than the superclass or other subclasses (in ways that are of interest).

15.2.4 Example

In your measurement system you are having two sensors, one pressure sensor and one temperature sensor. A sensor will be a superclass as both the 100% rule and the is-a rule is confirmed. The class diagram is as showed in Figure 15.6.

The sub classes will contain all the members from the super class and the code can be:

```
//////////////////////////////////////////////////////////
//
public float GetTempValue(int channel)
//
// Purpose: get the temperature value
//
// Version: 1.0: 12-DEC-13: NOS
{
    float fval ;

    fval = GetDaqValue(channel) ;  // Read the value from the sensor
    fval = CalcTemp(fval) ;        // Local method for converting to temp.
    return fval ;
}
//////////////////////////////////////////////////////////
//
public float GetPressValue(int channel)
//
// Purpose: get the pressure value
//
// Version: 1.0: 12-DEC-13: NOS
{
    float fval ;
```

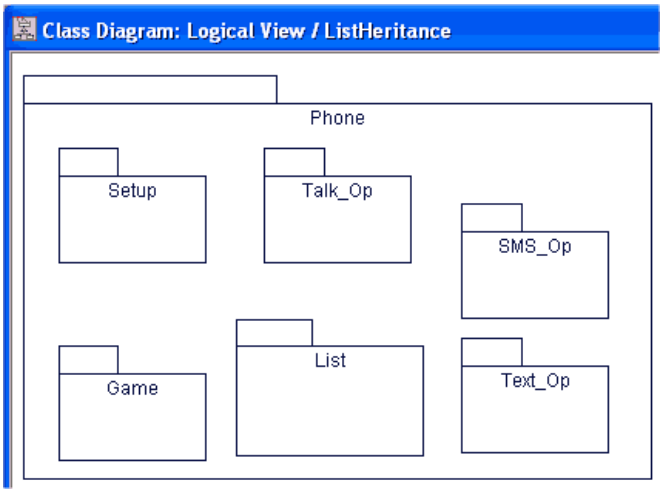


Figure 15.7: An UML package diagram of the mobile phone.

```
    fval = GetDaqValue(channel) ;   // Read the value from the sensor
    fval = CalcPress(fval) ;        // Local method for converting to press.
    return fval ;
}
```

Both subclasses are using the same method, *GetDaqValue()*, to read the value from a measurement system (DAQ) where the sensors are connected to two different channels. By using a super class will the method be defined only once. See Appendix E.2 for more extended C# code for sensor inheritance.

Both TempSensor and PressSensor classes can also have used the same name of the method, like GetValue(), however this requires a good name of the object reference to make the source code more readable. Remember that every method has to include the reference of the object to call a method inside an object. If short object references are used, like rts ()) and rps (), longer method names should be used (like *rts.GetTempValues()* and *rps.GetPressValue()*). If longer object references are used, like rTempSensor and rPressSensor, shorter method can be used (like *rTempSensor.GetValues()* and *rPressSensor.GetValue()*). Combine the object references and name of methods in such a way that the source code is readable. An advice is to use the full class name as the object name, with r (reference) or or(object reference) in front of the class name to separate between a class and an object reference to the class.

15.3 Package design

A domain model can easily grow large enough that it is desirable to factor it into sub diagrams of strongly related concepts. Package design can be used for dividing domain model, class diagram and object diagram into a more structured overview when growing large. Figure 15.7 shows an UML package diagram for the main functions of the mobile phone.

Figure 15.8 shows a detailed diagram of the superclass and subclasses of the List package.

Packages:

1. Used for different detailing levels of the domain model,
2. Group concepts belonging to each other in packages,
3. A package can be drawn inside another package.

Partition the domain model by grouping conceptual classes:

1. in the same area, closely related by concept or purpose,
2. in the same hierarchy,
3. participate in the same use cases,

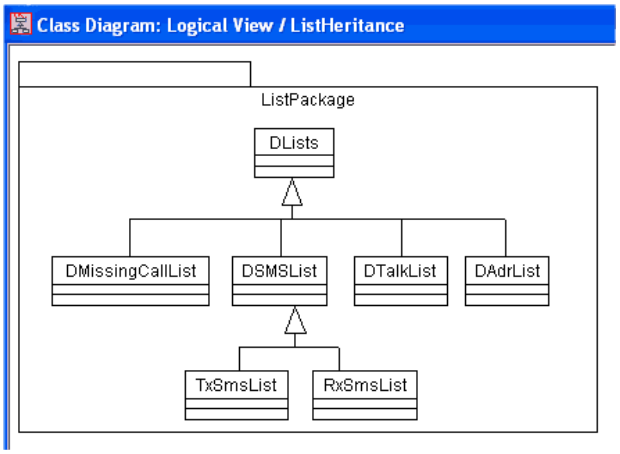


Figure 15.8: An UML package for the list classes of the mobile phone.

4. that are strongly associated.

15.4 Architectural analysis

In UP will the most important use case be analyzed, designed, coded, and tested. The selection of the most important use cases is part of the architectural analysis of the software system. The architectural analysis is to make a conceptual model for defining the structure, behavior and views of the software application.

1. Using UP, starting before the first development iteration as architectural issues need to be resolved early,
2. Using UP, programming and testing in early iterations before architectural analysis is complete,
3. Analysis part:
   (a) concerned with the identification and resolution of the system's non-functional requirements in the context of the functional requirements,
   (b) Especially R=Reliability and P=Performance in FURPS+),
4. Protected Variation pattern (GRASP):
   (a) variation points, variation of the system must be supported (different interfaces),
   (b) evolution points, variation that may arise in the future,
5. Architectural concerns involve system-level, large-scale and broad problems,
6. Reliability and fault-tolerance requirements affect the design,
7. Licensing cost of subcomponents affect profitability, and the design,
8. Adaptability and configurability requirements affect the design.

Common steps in Architectural analysis:

1. Identify and analyses the non-functional requirements that have an impact on the architecture, these requirements are named the architectural factors,
   (a) Collection and organization of the factors in a table (often named the factor table).
2. The requirements with a significant architectural impact, analyze alternatives and create solutions that resolve the impact, named the architectural decisions.

   (a) Decisions like “remove the requirements”, “hire an expert”, or “stop the project”.
   (b) Priorities:
      i. Inflexible constraints like a law, safety, and legal compliance,
      ii. Business goals,
      iii. All other goals.

3. Basic architectural design principles:
   (a) using low coupling, high cohesion and protected variation on objects, components, subsystems, and applications,
   (b) Layer patterns are probably the most important patterns to achieve these goals (common separation technique).
      i. layer patterns: Facade, observer and controller; will be part of the Design phase.

Chapter 16

Design

16.1 Introduction

The design step will involve more design patterns, in this iteration will the focus be design pattern for more layered structure. The start will be to extent the GRASP and then look into some of the GoF (Gang of Four) design patterns.

16.2 GRASP: More objects with responsibilities

The rest of the GRASP are: *Polymorphism, Indirection, Pure Fabrication* and *Protected Variations*. The properties for OO programming languages are:

1. Encapsulation; private attributes and public methods,
2. Inheritance; superclasses and subclasses,
3. Polymorphism.

Let us start with polymorphism.

16.2.1 Polymorphism pattern

Pattern name	Polymorphism
Problem	How to handle alternatives base on types? How to create pluggable software components?
Solution	Assign responsibility for the behavior, using polymorphic operations, to the types for which the behavior varies.

The user of the type should use polymorphic operations instead of explicit branching based on type.

Example 15 *Implement different create methods for making a new address element, see Figure 16.1.*

The code for the Figure 16.1 can be:

```
public AdrId CreateAdrId()
{
    AdrId _rAdrId;

    _rAdrId = new AdrId();   // Make memory location for data
    return _rAdrId;
}

public AdrId CreateAdrId(string name)
```

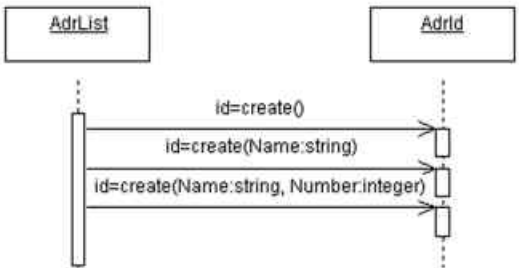


Figure 16.1: Use polymorphism for the create method.

```
{
    AdrId _rAdrId;

    _rAdrId = CreateAdrId() ;
    _rAdrId.setName(name) ;
    return _rAdrId ;
}

public AdrId CreateAdrId(string name, int number)
{
    AdrId _rAdrId;

    _rAdrId = CreateAdrId(name) ;
    _rAdrId.SetNumber(number) ;
    return _rAdrId ;
}
```

The first **create()** has the responsibility to create the object. The second **create()** will use the first cerate to cerate the obejcty and has the responisbility to set the name only. The third **create()** uses the second create function for creation and setting the name, and has the responsiblity to set the number only. If the creation of the memory (new() method) is going to be changed or extended, this is handled only in one location in the source code.

Example 16 *Use polymorphism for adding different parameter types.*

The polymorphism will use the signature of the method to decide which method to use. The signature is the number and types of the parameters. Adding numbers and adding strings requires two difrerent methods like:

```
public string Add(string s1, string s2)
{
   // add a space between the strings
   return s1 + " " + s2;
}

public int Add(int i1, int i2)
{
   return i1 + i2;
}
```

Example 17 *Use interface and polymorphism for plugable game components, see Figure 16.2.*

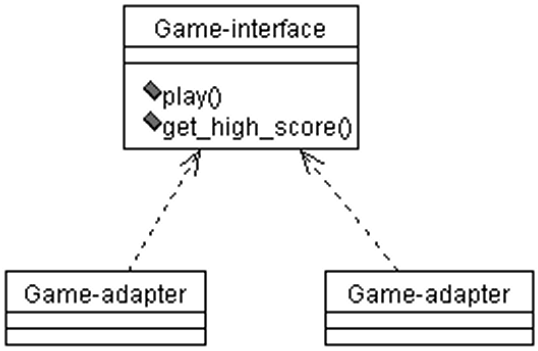


Figure 16.2: Use polymorphism for more game components.

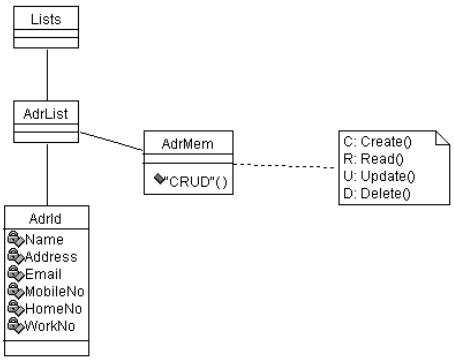


Figure 16.3: Using Pure Fabrication for saving the address list.

16.2.2 Pure Fabrication pattern

Pattern name	Pure Fabrication
Problem	How to keep a High Cohesion and Low Coupling when designing?
Solution	Assign responsibility to an artificial class, something made up that keeps High Cohesion and Low Coupling.

Example 18 *How to save the information in the AdrList when turning off the mobil phone? Made up a new class doing the CRUD (Create, Read, Update, Delete) operations, depending on type of database: memory, SIM card, IR communication, card reader etc. See Figure 16.3.*

16.2.3 Indirection pattern

Pattern name	Indirection
Problem	How to avoid direct coupling between to things?
Solution	Assign responsibility to an intermediate object to mediate between other components or services.

Examples:

In Figure 16.2 an interface class was added to mediate between the system and external game components. In Figure 16.3 a "memory" class was added to mediate between the address list and the memory.

16.2.4 Protected Variations pattern

Pattern name	Protected Variations
Problem	How to design objects, subsystems and system so that variations in these elements does not have an undesirable impact on other elements?
Solution	Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

HINT:

1. Identify modules/methods in the software with possibility for changes,
2. Add classes to get higher cohesion and lower coupling,
3. Use the patterns for polymorphism and Pure fabrication.

16.3 Applying GoF Design Patterns

GoF: Gang-of-Four design patterns: A popular work that presents 23 patterns regarding object design, and perhaps 15 of these are common and most useful. We will look into some of these patterns.

16.3.1 Adapter pattern

Pattern name	Adapter
Problem	How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
Solution	Convert the original interface of a component into another interface, through an intermediate adapter object.

Example 19 *See Figure 16.2.*

16.3.2 Factory pattern

Pattern name	Factory
Problem	Who should be responsible for creating objects when there are special considerations, as complex creation logic etc.?
Solution	Create a Pure Fabrication object called a factory that handles the creation.

Example 20 *Who is making the adapters in Figure 16.2? Use a factory object returning the reference to the adapter object.*

16.3.3 Singleton pattern

Pattern name	Singleton
Problem	Exactly one instance of a class is allowed, objects need a global and single point of access.
Solution	Define a static method of the class that returns the singleton.

An object containing a large algorithm, do we need more copies of the same algorithm?

Example 21 *An example of a Singleton class in C#:*

```
// Singleton pattern -- Structural example
// From www.dofactory.com/Patterns/PatternSingleton.aspx#intent
//
using System;
//
namespace DoFactory.GangOfFour.Singleton.Structural
{
  // MainApp test application
  //
  class MainApp
  {
    static void Main()
    {
      float  fval1, fval2 ;
      // Constructor is protected -- cannot use new
      Singleton s1 = Singleton.instance();
      Singleton s2 = Singleton.instance();
      // Check if the same object
      if (s1 == s2)
      {
        Console.WriteLine("Objects are the same instance");
      }
      fval1 = s1.Algorithme(0.0F) ;
      fval2 = s2.Algorithme(0.0F) ;
      // Wait for user
      Console.Read();
    }
  }

  // "Singleton"

  class Singleton
  {
    private static Singleton instance;

    // Note: Constructor is 'protected', can not use the new operator
    protected Singleton()
    {
        // Any setup
    }

    public static Singleton instance()
    {
      // Use 'Lazy initialization'
      if (instance == null)
      {
        instance = new Singleton();
      }
      return instance;
    }
    public float Algorithme(float fval)
    {
        // Your algorithm method
    }
  }
}
```

16.3.4 Strategy pattern

Pattern name	Strategy
Problem	How to design for varying, but related, algorithms or polices? How to design for the ability to change these algorithms or polices?
Solution	Define each algorithm/policy/strategy in a separate class, with a common interface.

16.3.5 Composite pattern

Composite pattern is used when we need to treat a group of objects and a single object in the same way. An example can be 1) when double click on a folder for your computer it should open the folder. 2) when double click on a file, the file should be opened by a corresponding application.

Pattern name	Composite
Problem	How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?
Solution	Define classes for composite and atomic objects so that they implement the same interface.

16.3.6 Fasade pattern

Pattern name	Fasade
Problem	A common, unified interface to a disparate set of implementations or interfaces, such as within a subsystem, is required. There may be undesirable coupling to many things in subsystem, or implementation of the subsystem will change. What to do?
Solution	Define a single point of contact to the subsystem, a facade object that wraps the subsystem.

The facade pattern is also used when designing a two, three og four tier architecture of a software application to separate the functionality of the user interface, business logic, data storage, and any operating system functions. The business logic should at least be separated from the user interface to get a better structure of the application, to be able to manage different user interfaces without changing the business logic, and for better maintenance of the business logic. depending on the size of the application, one or several facade classes can be useful. In larger application it can be useful to have one class for handling several subparts of the the application like the status, configuration data and the runtime data of the application to name some.

An example of an domain model is shown in Figure 16.4 where Class C, D and E consists of a sub system, like the data tier. A facade class is included as an interface class between the main system and the sub system. This way is is more easy to do changes, replacement or extensions of the sub system which will not affect the whole system only the facade class.

Another example can be storing of data like logging of data samples. Developing a general datalogging software requires that the user can choose to log on either a SQL database, a CSV file or an XML file, and let the user to configure the type storage when using the software. In this case will the rest of the application be independed of the storage type, and the facade object will use object C to use a SQL database, object D to use CSV files and object E to use an XML file. Based on the user configuration will the Facade object use either object C, D or E for data storage. It any error, or change, of the CSV file or file format, most probably only class D needs to be changed.

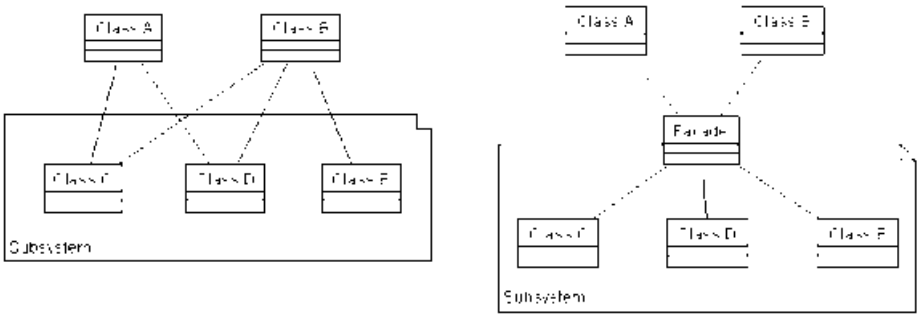


Figure 16.4: The usage of the facade pattern for a sub system (like the User Interface?).

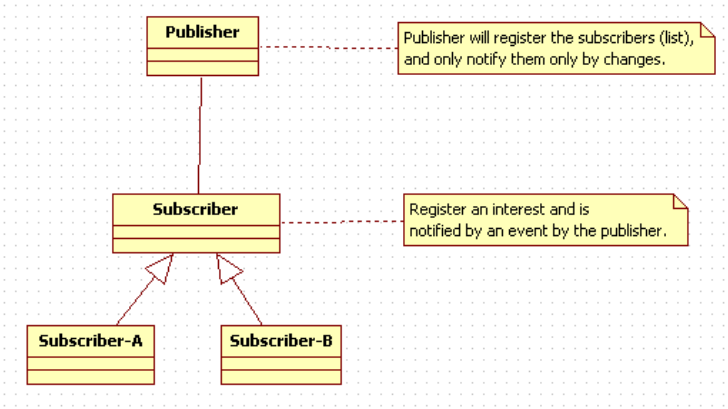


Figure 16.5: The observer pattern with a publisher and a subscriber.

See Appendix E.3 for C# classes for NI DAQ interfaces and Appendix E.4 for an overview of a three tier architecture in C# with facade classes for each tier. Also the applications in the data communication chapter are based on the Facade design pattern.

16.3.7 Observer pattern

Pattern name	Observer (Publish-Subscribe)
Problem	Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. How to retain low coupling?
Solution	Define a "subscriber" or "listener" interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

Publish and subscriber objects communicated by events, and an event is a notification sent by an object to signal the occurrence of an action. The Publisher class will raise an event and any Subscriber classes subscribing to this event will receive a notification at the occurence of the action. Events in C# (and .NET) follow the observer design pattern. In C#, an event is an encapsulated delegate and the delegate defines the signature for the event handler method of the subscriber classes.

Figure 16.5 shows the observer pattern with a publisher sending events when any changes, and a subscriber (or lisener or observer) that will update any information based on the events.

This pattern is often used in Graphical User Interfaces where a window should not be updated all the time (by polling), but only on changes. See Appendix E.4 for an overview of a three tier architecture and classes supporting the publisher/subscriber in C#. The data layer tier is a publisher for a random event for user information information, and the user interface layer tier is the subscriber on this event. So the user interface layer will use methods to send infromation to the business layer tier and the data

layer tier, but not polling the event in the data layer tier. Instead the user interface layer tier will just subscribe on this event and receive a message when the event is activated. The GUI application in the data communication chapter is based on this observer design pattern for sending information from the data layer tier to the user layer tier.

Example 22 *Open two or three file explorer windows on a specific directory on the computer, and all file explorer windows must show the same directory. In one of the file explorer windows, right click and create a new file (an empty text file). The other file explorer windows will be updated immediately after the file is created. Then delete the same file in one of the other file explorer windows, and again will all the file explorer windows be updated immediately. The Operating system file manager will be the publisher object and the file explorer windows will be the subscriber objects subscribing to any changes made by the file manager. Without the usage of the observer design pattern would every file explorer window poll the file manager system all the time wasting an huge amount of computing capacity and memory.*

16.3.8 Model View Controller pattern

The Model View Controller pattern is a type of an observer pattern with focus on the user interface. However, this pattern will be more an architectural pattern rather than a design pattern.

Pattern name	Model View Controller (MVC)
Problem	How to update the user Interface based in information from both the user and the system?
Solution	Define separate objects for the control information (from the user and/or system)(controller), the presentation information (on the UI)(View) and a data model.

The model can be part of the business layer and contains only dynamic application data, independent of the UI, and manages the data, logic and rules of the application, or part of the application. The view presents the data from the model on the UI. The view knows how to access the data from the model, but not what the data means nor what the user can do to manipulate it. Multiple views of the same information are possible. The controller listens to events triggered by the view or other sources, and executes the appropriate reaction to these events. In most cases will the reaction be to call a method on the model. So the model updates the view and the user sees the view. The user uses the controller to manipulate the model, and the view is updated based on the manipulation. The view will also be updated based on other external source on the model.

16.3.9 Alarm system

An alarm system is one example of using several of these patterns. An alarm system consists of inputs from both the process and the user and should be implemented as a three layer architecture using the facade pattern. The alarm standard logic will be in the business layer while the data layer will collect the process values. The observer pattern should be used for the data layer to inform the business layer when a new set of process values is available, using events in C#. The business layer should inform the UI layer when any change of alarm states so the UI layer can update the alarm states and any alarm lists. The UI must have support for alarm acknowledge using the facade layer to inform the business layer about any user action. In general the facade patterns are used when sending information to a lower level in the architecture model, and the observer pattern when sending information to a higher level in the architecture model. In C# methods are used for facade patterns and events are used for the observer pattern.

16.4 Activity diagrams and modeling

Activity diagrams are a diagram technique for showing workflows of stepwise activities and actions, with support for choice, iteration and concurrency (Larman 2002), (Martin 1993), (Oestereich 2001). An activity diagram shows the overall flow of control. In UML activity diagrams can be used to describe the business and operational step-by-step workflows of components (object or collection of objects) in a software system.

16.4.1 Activity diagrams

1. can be used in both analysis phase and design phase, *a*) analysis phase: a task that needs to be done. *b*) design phase: a method in a class.
2. An activity can be followed by another activity,
3. A type of flowchart but can handle parallel behavior and synchronization points,
4. can show both control flow and data flow,
5. useful for complex algorithms,
6. useful for concurrent programs (more and more important for CPUs with multi core),
7. supports decisions.

An activity diagram for MakingSMS is shown in Figure 16.6. The Save Message activity can be extended with a decision block as shown in Figure 16.7.

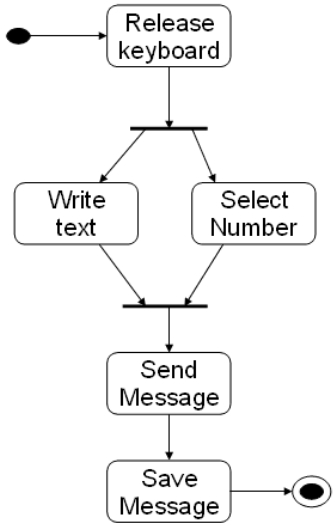


Figure 16.6: An activity diagram for MakingSMS function of the mobile phone.

An activity diagram for SaveMessage is shown in Figure 16.7.

16.4.2 Elements in an activity diagram

1. Swim lanes; who is doing what (classes),
2. Partitions; names of who is doing what (class names),
3. Actions; does something and an automatic transition (class methods),
4. Fork, one incoming transition, multiple outgoing parallel transitions,
5. Join, multiple incoming transitions, one outgoing transition, for synchronization.

An activity diagram with these elements is shown in Figure 16.8.

16.4.3 Guidelines for using an activity diagram

1. Most useful for very complex processes,
2. Use different levels of abstraction to keep the diagrams “short”,
3. Behavior analysis of a use case,

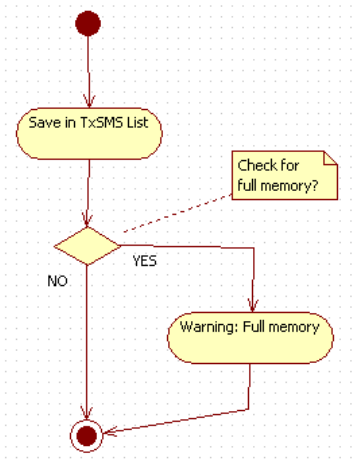


Figure 16.7: An activity diagram for the Save Message function for the mobile phone.

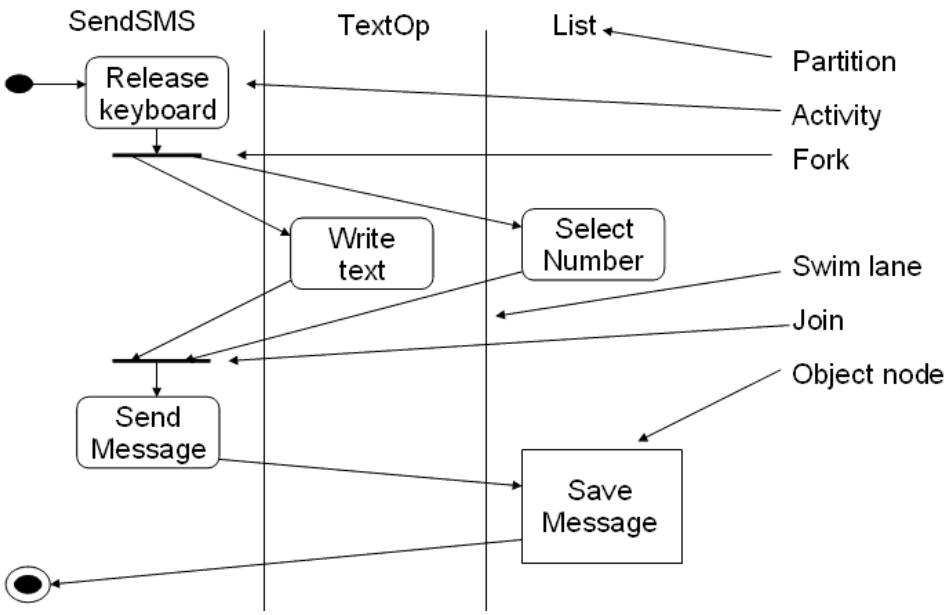


Figure 16.8: An activity diagram showing more of the design of SendSMS operation.

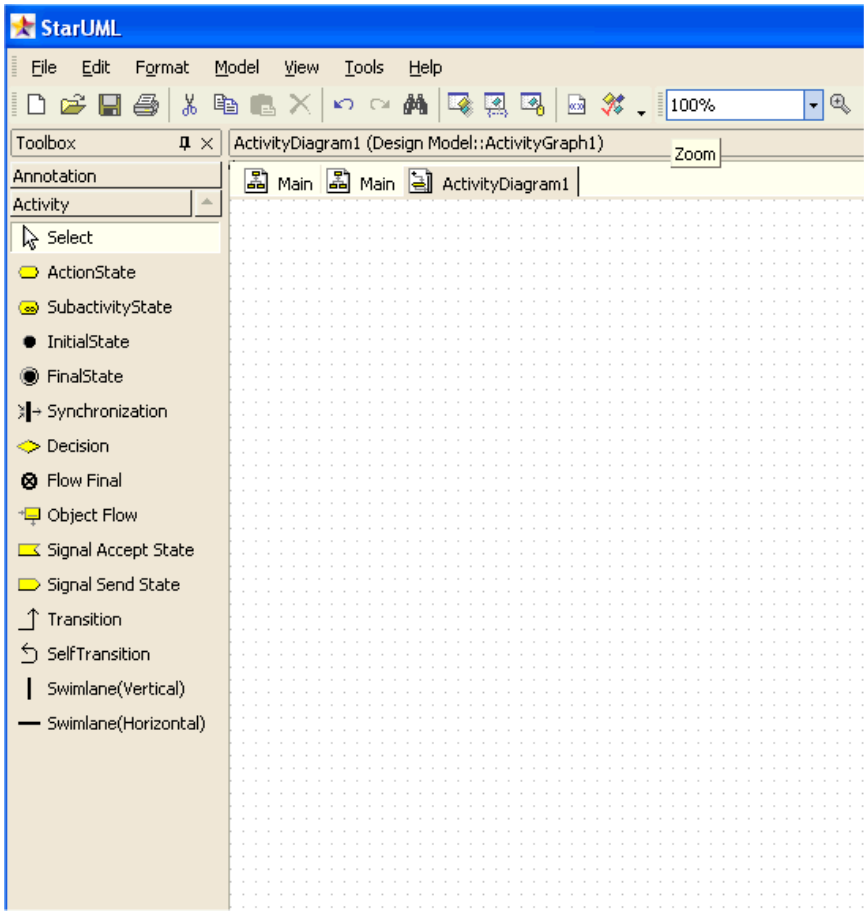


Figure 16.9: The Activity diagram in StarUML with the toolbox to the left.

4. Gives a better understanding of the workflow,
5. Overview of multi-threaded applications.

NOTE: No support for activity diagram in the demo version of Rational Rose!

16.4.4 Activity diagrams in StarUML

Activity diagrams can be added to both the analysis model and the design model. Right click on the type in the model explorer and add the Activity diagram using the Add Diagram option. Figure 16.9 shows the Activity Diagram with the toolbox to the left. The toolbox shows the ActionState box, the InitialState, the FinalState, Syncronization, transition, and the Swimlines at the bottom.

16.5 State machine diagrams and modeling

A state diagram, sometimes knowns as a state machine diagram, is a type of behavioral diagram. It specializes in showing transitions between various objects but can also be used to modeling the state changes within an object. A state diagram can be very useful for state specific systems and devices like 1) physical devices, 2) communication protocols, 3) real-time systems, and 4) (graphical) user interfaces (GUI).

16.5.1 State diagram

1. can be used in both analysis phase and design phase,
2. useful for describing the behavior of a use case,

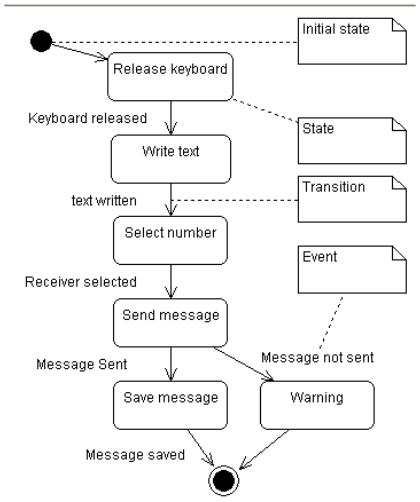


Figure 16.10: A state diagram for the MakeSMS function.

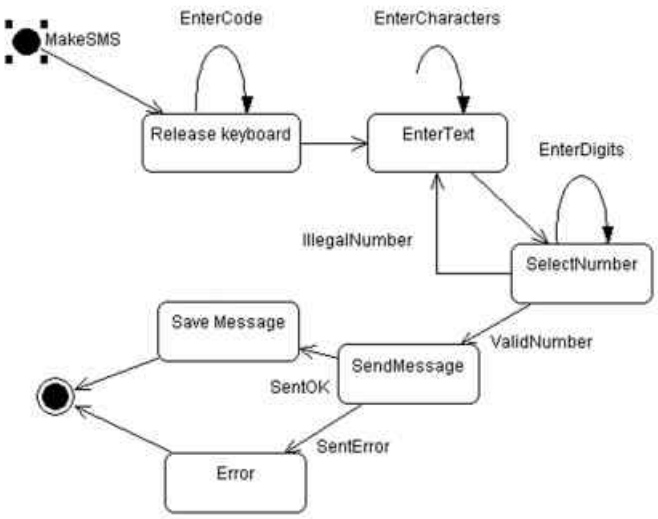


Figure 16.11: A more detailed state diagram for the MakeSMS function.

3. useful for getting an overview of valid events in a state,
4. used for making a state machine,
5. used a lot in real-time systems and communication protocols,
6. several state patterns as tools in the design phase,
7. implemented in software (C, C++, Java, or C#) as the switch / case,
8. can be used from small to large software projects, and in a large number of software projects,
9. can be used at many levels in software projects, from top-level design down to module coding,
10. gives a good documentation of the states, and easy to maintain and extend.

Examples of a state diagrams for the MakeSMS use case is shown in Figure 16.10 and 16.11.

An other example of a state machine can be the key release function of the mobile phone. A code must be entered before the keyboard is released or opened, this can also be showed by a state machine. An example of a state diagram for a three number code is shown in Figure 16.12 and 16.13. Figure 16.12 is focusing on the behavior of the method, more like an activity diagram, while Figure 16.13 is focusing more on the state changes.

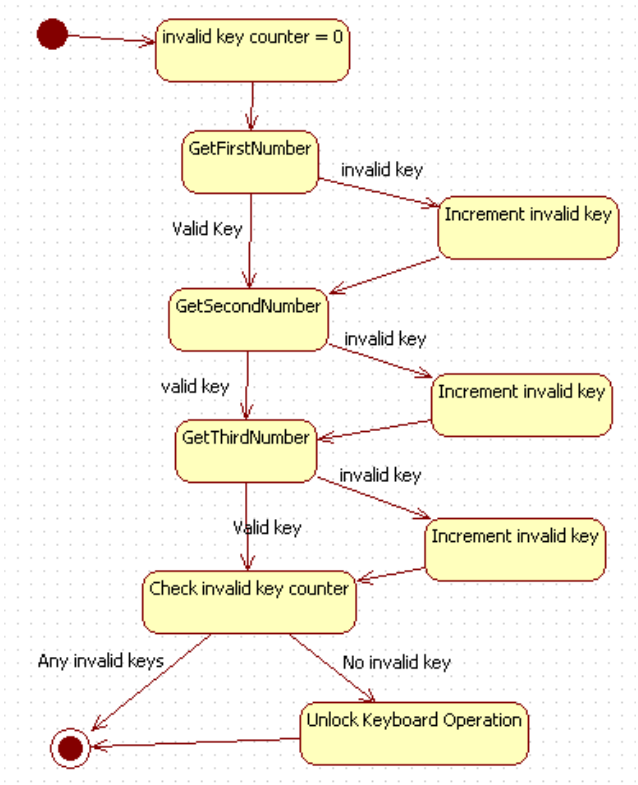


Figure 16.12: State diagram for the ReleaseKeyboard function, focus on the behavior.

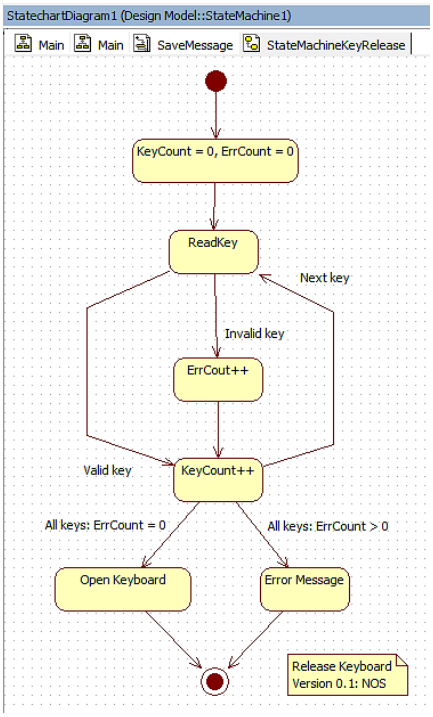


Figure 16.13: State diagram for the ReleaseKeyboard function, focus on the state changes.

One example of the source code for the state diagram can be:

```
public bool ReleaseKeyboard()
{
    const int  MAX_KEYS = 3 ;
    char       kCode[MAX_KEYS] ;
    int        kCount = 0, eCount = 0 ;
    char       key ;
    bool       status = False ;

    // Set the keyboard code
    kCode[0] = 3 ;
    kCode[1] = 4 ;
    kCode[2] = 9 ;
    // Read the keyboard
    while (kCount < MAX_KEYS)
    {
       switch (kCount)
       {
          case 0:
          case 1:
          case 2:
             key = ReadKey() ;
             // check the key sequence
             if (kCode[kCount] != key)
                eCount++ ;   // DO NOT use: kCount = MAX_KEYS
             break ;
          default :
             break ;
       }
       kCount++ ;
    }
    if (eCount == 0)         // No errors if only correct numbers
        status = KeyboardRelease() ;
    return status ;
}
```

The code is using most of the time in the *ReadKey()* function waiting for the next key. The *kCount* variable indicates the state, and the switch/case statements are used for keeping track of the current state. The state change, or transition, depends on the key variable from the *ReadKey()* function.

16.5.2 Elements in a state diagram

1. event: a significant or noteworthy occurrence (using no time),
2. state: the condition of an object at a moment in time,
3. transition: the relationship between two states, indicating that an event change the state of the object.

16.5.3 Guidelines

1. Consider state diagram only for state dependent object with complex behavior,
2. Business information system normally has few complex state dependent classes,
3. A control system normally has many complex state dependent classes.
4. The state diagram can be used used for both state changes in a system or within the same class.

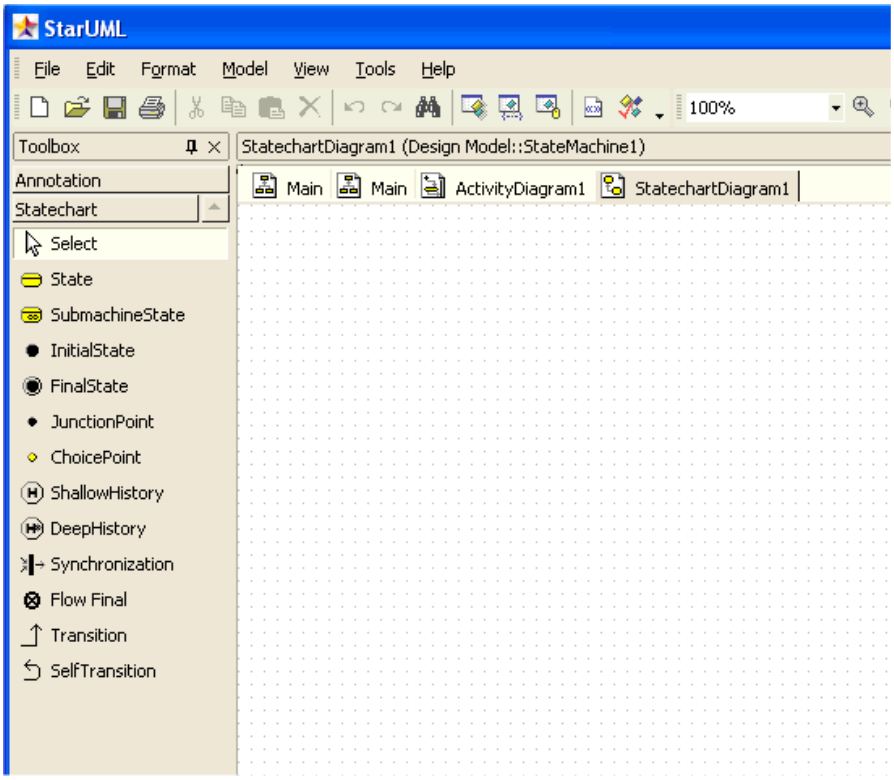


Figure 16.14: The statechart diagram in StarUML with the toolbox to the left.

Figure 16.15: An example of an alarm clock.

16.5.4 State diagrams in StarUML

State diagrams can be added to both the analysis model and the design model. Right click on the type in the model explorer and add the Statechart diagram using the Add Diagram option. Figure 16.9 shows the Statechart Diagram with the toolbox to the left.

16.5.5 State machine Exercise

Make a state diagram for the software of a smart alarm clock (See Figure 16.15). Smart in this case means that it should keep track of week-ends and holidays. Even smarter can be to access your calender to figure out your days off, your holidays and your working schedule. Tips: (1) Start with defining the states of your alarm clock, (2) check out the events of each states, (3) add transitions between the states including event information.

Include support for 1) the "Snooze" function, 2) week-ends, 3) holidays, 4) online calender system with meetings and appointments.

This PC > (M:) USN-Personal > usr > HIT > Undervisning > OOADP-SCE1306 > TechDoc > Arduino > Te

Name	Date modified	Type	Size
AnalogInput.ino	27.03.2017 12.24	INO File	4 KB
DigitalInput.ino	27.03.2017 12.24	INO File	3 KB
Display.ino	27.03.2017 12.24	INO File	3 KB
Modbus.ino	27.03.2017 12.24	INO File	2 KB
SensorFilter.ino	27.03.2017 12.24	INO File	3 KB
TempLcdBox.ino	27.03.2017 12.24	INO File	3 KB

Figure 16.16: The source code files created in this project.

16.6 Arduino development

The normal way for Arduino development involves using Arduino IDE (Integrated Developer Environment) and saving the source files as *.ino files. Note that *.ino files are just ASCII files like a *.c, *.cpp og *.cs files. The basis for development with Arduino is the setup() and loop() methods, the setup() is called once during the startup() of the system (constructor) while the loop() is the last method called. As always, when programming in any environment, try to structure the code and use object-oriented methods. Arduino can be developed without objects, however try to make file modules based on the same principles as for making objects. One way can be to make an use case diagram, and make one file for each of the use cases. If the use cases are complex or contains several statements, make a FDUCD, use patterns and design using an interaction diagram. Based on the interaction diagram, make one file for each of the classes defined in the interaction diagram. By using the same name on the file as the class, the interaction diagram can be used as documentation even if the software is not based on objects.

The loop() method can also be structured by using state machine modeling. In a project used as an example, an Arduino device is used as an input module for a Smart House project where the Arduino device has four sensor devices connected, a LCD for local presentation and a Modbus serial port protocol (Modbus RTU) for communication with the main system. The sensors are a LDR (Light Dependent Resistor) to measure the light intensity, a temperature sensor, a humidity sensor, and a IR (infra red) sensor to detect movement in the room. Developing with Arduino, do not include all the source code in one single file, use the object-oriented way of thinking by using responsibility, and divide into files. In this project the files are shown in Figure 16.16 where each file has a specific responsibility in the system.

The main project file is the TempLcdBox, using the following files:

- AnalogInput; handling the analog sensor readings like the LDR, temperature and humidity sensor values,
- DigitalInput; handling the input from the IR sensor, detecting when movement of a person is present and not,
- Display; handling the output on the LCD, two lines of information (2x40 display),
- Modbus; handling a type of a Modbus protocol, being a Modbus master sending messages at fixed time intervals,
- SensorFilter; handling the low pass filtering of the analog sensors.

This system has different states and a state machine diagram will be one way to model such a piece of software. The software will either be reading the sensor, displaying the values or communication the values with the Modbus master. Another challange is that the analog sensors are measuring slow processes while the digital sensor is measureing a much faster process. This means that we want to read the digital sensor often, while the analog sensors do not need to be read so often. The listing of the TempLcdBox is shown with the setup() and loop() methods. The setup() method is calling a setup() method in all the files, similar to a constructor method for an object. The loop() method consists of a state machine where the variable LoopState indicates the state. A state diagram is shown in Figure 16.17.

The LoopState variable is incremented at the end of the Loop() together with a 200 ms delay indicating that the Loop() method will run 5 times each second. The LoopState will start at 0, have a startup

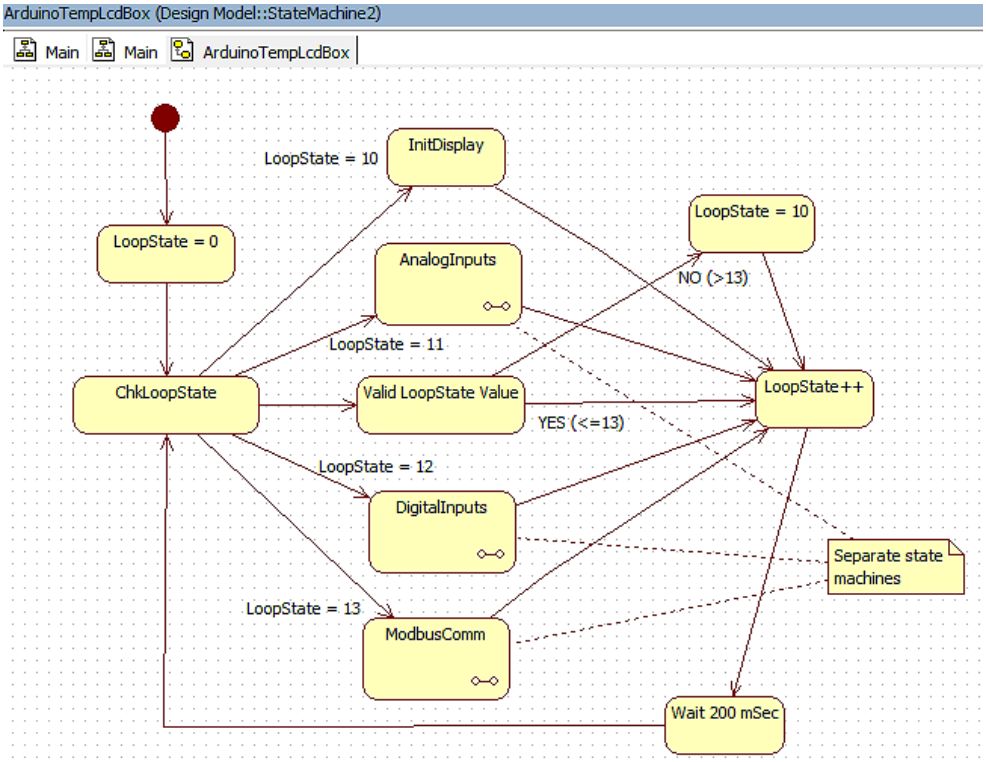


Figure 16.17: The main states of the loop() method, the main method in an Arduino software. The software has four main states, 1) init the display, 2) read analog inputs, 3) read digital inputs, and 4 send the specified parameters on the serial port using a Modbus type of protocol.

period of 2 seconds before the display will be init at LoopState = 10. At LoopState = 11 an analog port will be read, at LoopState = 12 will the digital input be read, and at LoopState = 13 the Modbus module will be active. At LoopState = 14 will the LoopState be changed to 10, and incremented som the next state will be 11 (reading a new analog input). The state machine will manage reading of one analog sensor every seconds, reading the movement sensor at least every second, and sending one Modbus message every second. If any of the states do not need to be managed so often, the state machine logic can handle this as well. The states for reading the analog and digital inputs will also update the display with any new values.

The source code for the setup() and loop() methods for the Arduino system:

```
/////////////////////////////////////////////////////////////
/// Module: TempLcdBox
/// Purpose: Read remperature wall box and display on LCD
/// Version: 1.0: NOS: 2-APR-16: First version with LCD, TEMP, LDR
///       HUMIDITY and Movement sensors.
///   1.1: 7-APR-16: Complete system in one C file: NOS
///   1.2: 9-APR-16: Making several source files: NOS
/////////////////////////////////////////////////////////////
///
const int APP_VERSION = 12;
///
/// State machine
int LoopState;
int LoopAIState;
int LoopDIState;
int LoopModbusState;
///
const int LOOP_STATE_START = 0;
```

```
const int LOOP_STATE_INIT_DISPLAY = 10;
const int LOOP_STATE_ANALOG_INPUTS = LOOP_STATE_INIT_DISPLAY + 1;
const int LOOP_STATE_DIGITAL_INPUTS = LOOP_STATE_ANALOG_INPUTS + 1 ;
const int LOOP_STATE_MODBUS_COMM = LOOP_STATE_DIGITAL_INPUTS + 1 ;
const int LOOP_STATE_WAIT = 200;
///
void setup()
///
/// Purpose: setup of the Arduino card
///
/// Version: 1.0: NOS: 2-APR-16
{
  DisplaySetup();
  SensorFilterSetup();
  ModbusSetup();
  AnalogInputSetup();
  DigitalInputSetup();
  //
  LoopState = 0;
  LoopAIState = 0;
  LoopDIState = 0;
  LoopModbusState = 0;
}
///
void loop()
///
/// Purpose: main loop of the application
/// Version: 1.0: NOS: 2-APR-16
{
  switch (LoopState)
  {
    case LOOP_STATE_START:
      break ;
    case LOOP_STATE_INIT_DISPLAY:
      // Come here only at startup, clear display
      LoopStartupDisplay();
      break;
    case LOOP_STATE_ANALOG_INPUTS:
      LoopAIState = AnalogInputLoopState(LoopAIState);
      break ;
    case LOOP_STATE_DIGITAL_INPUTS:
      LoopDIState = DigitalInputLoopState(LoopDIState);
      break;
    case LOOP_STATE_MODBUS_COMM:
      LoopModbusState = ModbusLoopState(LoopModbusState) ;
      break;
    default:
      if (LoopState > LOOP_STATE_INIT_DISPLAY)
      {
        LoopState = LOOP_STATE_INIT_DISPLAY;
      }
      break;
  }
  LoopState++;
  delay(LOOP_STATE_WAIT);
}
///
void LoopStartupDisplay()
```

```
///
/// Purpose: Init the display after the startup text
/// Version: 1.0: 8-APR-16: NOS
{
  DisplayLoopSetup();
  SensorFilterFillBuffers();
}
///
//////////////////////////////////// EOF ////////////////////////
```

The software must be based on methods not using any time or waiting, meaning that a new state machine should be used in the AnalogInput and Modbus modules as well. The listing of the AnalogInput module is shown, showing the state machine for reading the analog sensor. Only one analog sensor will be read for each loop, and only this value will be updated on the display.

```
//////////////////////////////////////////////////////////////
/// Module: AnalogInput
/// Purpose: handle the analog inputs ports values
/// Version: 1.0: 9-APR-16: Multifile design: NOS
//////////////////////////////////////////////////////////////
///
const int AI_LDR_SENSOR_PORT = A0;
const int AI_HUM_SENSOR_PORT = A1;
const int AI_TEMP_SENSOR_PORT = A2;
///
// State Machine
const int SENSOR_LDR_ID = 0;
const int SENSOR_HUMIDITY_ID = 1;
const int SENSOR_TEMP_ID = 2;
const int SENSOR_MAX_SIZE = 3;
///
void AnalogInputSetup()
///
/// Purpose: init the module
/// Version: 1.0: 9-APR-16: NOS
{
}
///
int AnalogInputLoopState(int state)
///
/// Purpose: handle the state machine for reading the inputs
/// Version: 1.0: 9-APR-16: NOS
{
  switch (state)
  {
    case SENSOR_LDR_ID:
      AnalogInputLDRSensor(true);
      break;
    case SENSOR_HUMIDITY_ID:
      AnalogInputHumiditySensor(true);
      break;
    case SENSOR_TEMP_ID:
      AnalogInputTemperatureSensor(true) ;
      break;
    default:
      state = SENSOR_LDR_ID - 1 ;
      break ;
  }
```

```
  state++;
  return state;
}
///
void AnalogInputLDRSensor(bool showDisplay)
///
/// Purpose: get the LDR sensor value and display
/// Version: 1.0: 8-APR-16: NOS
{
  double iSensVal;

  iSensVal = (double) (1023 - analogRead(AI_LDR_SENSOR_PORT));
  iSensVal = AnalogInputAdcConvertMilliVolt(iSensVal);
  iSensVal /= 50.0F;  // Convert from mV to percent
  iSensVal = SensorFilterLowPass(SENSOR_LDR_ID, iSensVal);
  if (showDisplay == true)
  {
    DisplayLcdValue(0, 10, iSensVal) ;
  }
}
///
void AnalogInputHumiditySensor(bool showDisplay)
///
/// Purpose: get the humidity sensor value and display
/// Version: 1.0: 8-APR-16: NOS
{
  double iSensVal;

  iSensVal = analogRead(AI_HUM_SENSOR_PORT);
  iSensVal = AnalogInputAdcConvertMilliVolt(iSensVal);
  iSensVal = (iSensVal - 700.0F) * (100.0F / 3100.0F) ;
  iSensVal = SensorFilterLowPass(SENSOR_HUMIDITY_ID, iSensVal);
  if (showDisplay == true)
  {
    DisplayLcdValue(1, 2, iSensVal) ;
  }
}
///
void AnalogInputTemperatureSensor(bool showDisplay)
///
/// Purpose: get the temperature sensor value and display
/// Version: 1.0: 8-APR-16: NOS
{
  double iSensVal;

  iSensVal = (int) analogRead(AI_TEMP_SENSOR_PORT);
  iSensVal = AnalogInputAdcConvertMilliVolt(iSensVal);
  iSensVal = ((iSensVal - 100.0F) / 10.0F) - 40.0F;
  iSensVal = SensorFilterLowPass(SENSOR_TEMP_ID, iSensVal);
  if (showDisplay == true)
  {
    DisplayLcdValue(0, 2, iSensVal) ;
  }
}
///
double AnalogInputAdcConvertMilliVolt(double adcValue)
///
/// Purpose: convert the ADC value to millivolt value. ADC from 0 to 1023.
```

```
/// Version: 1.0: 8-APR-16: NOS
{
  adcValue *= 5000.0F;
  adcValue /= 1023.0F;
  return adcValue;
}
///
//////////////////////////////EOF////////////////////////////
```

16.7 Logical Architecture Refinement

Due to the UP each iteration should give a tested version of the software so it is important to define the architecture as early as possible in the project. The FURPS+ requirements can be useful and remember that U (Usability), R (Reliability), and P (Performance) will reflect the architecture of the software. U with the type of I/O devices, R with any requirements for redundancy, and P with any requirements for speed and/or number of operations each second. The system architecture will often be based on experience gained from similar systems or in similar problem domains.

16.7.1 Deployment diagrams

The physical model in UML describes the components (both hardware and software) that will be deployed into the target environment. During the test steps in each iteration it is important to have a copy of the target environment for valid testing, espesially before sending any test versions to the user (or customer). The UML diagram is using nodes as computer devices, processors, network connectivity, operating systems, software components and tools. The UML diagram is a deployment diagram and shows the nodes and the relationships, where the relationships often are some sort of communication associations. An example of a deployment diagram for a mobile phone is shown in Figure 16.19. Communication between nodes are indicated by associations. Components or packages are used to illustrate executables, applications or other software components that are deployed on nodes, and connected with dependency links. Deployment diagrams can also model non-computer entities and systems within an organisation, model things like rules and procedures and business modelling.

16.7.2 Evaluation of the logical architecture

The steps for evaluating the logical architectural can be:

1. Check the requirements, especially the U, R and P,
2. Evaluate the UML use case diagram, the number and type of use cases,
3. Hardware components, UML deployment diagram, see Figure 16.19 for the mobile phone. Use the deploy diagram to structure the distribution of I/O devices and processing modules. A deployment diagram shows the architecture of the system as deployment (distribution) of software artifacts to deployment targets,
4. Defined a layered architecture, use one, two, three, or four tiers. Based on the defined layer make the objects like program, datalayer facade object, business layer facade object and user interface layer facade object, Start making the data layer facade object, then making the business layer facade object by including the reference to data layer facade object, and make the user interface facade object at the end, including a reference to the business layer facade object. All layer facade objects has now references to the other facade objects in a top-down structure. The user interface facade object can then start the event handler for user interaction.
5. Make a prototype of the user interface with focus on the interaction between the application and the user. Suggest the outline of the user interface with the main views, buttons, and information,
6. Logical view, UML packages, see Figure 16.18 for the mobile phone. Use the logical view to get an overview of the information in the system, and the data flow,

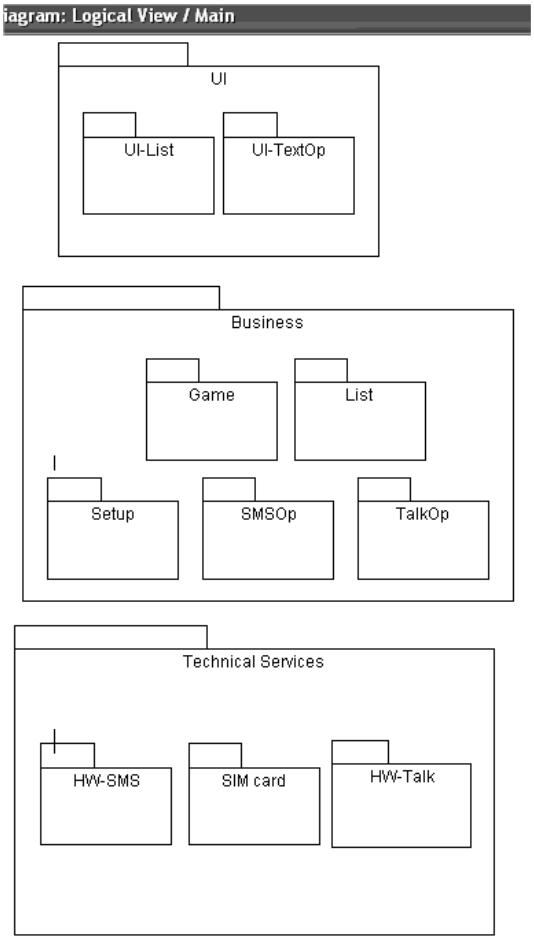


Figure 16.18: A logical architectural view of the mobile phone.

7. Use the facade, observer, and controller pattern for layered structure. The controller pattern can be used to define the main objects in the business layer, the objects responsible for each of the use cases,
8. Process, Processes and threads (parallel behavior), UML interaction and activity diagrams,
9. Data, UML class diagram,
10. Implementation, software modules,
11. Testing.

In step 4 a layered architecture should be decided, to figure out how many tier should be used in the application. The number of tier will decide how structured the software design will be.

1. Evaluate the usage of either One-tier, Two-tier, Three-tier or Four-tier.
2. The classic three-tier architecture: Interface, application logic, and storage.
   (a) Interface is the presentation layer, GUI[1], UI[2], HMI[3],
   (b) Application layer is the business layer (and domain layer),
   (c) Storage is the data layer, later separated in Technical Services, Network Services and OS Service.

[1] GUI: Graphical User Interface.
[2] UI: User Interface.
[3] HMI: Human Machine Interface.

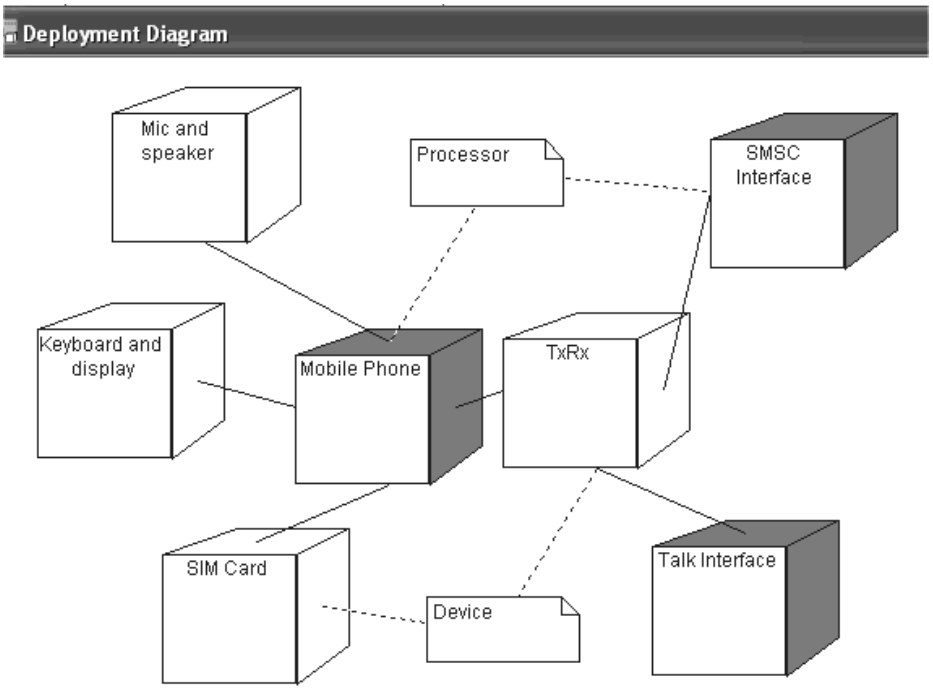


Figure 16.19: An UML deploy diagram for the mobile phone consisting of the Processor and Device modules.

In step 6 it is indicated that patterns should be used to make a layered structure, in addition to the number of tiers defined. The different tiers should collaborate by using patterns with focus on layers like:

1. use facade pattern for accessing a subsystem or a software package, a "downward" collaboration,
2. use observer pattern for "upward" collaboration, mainly the UI layer.
3. use controller pattern for collaboration with the operator or other systems.

The description of the facade, observer and controller patterns:
Fasade pattern (GOF):

Pattern name	Fasade
Problem	A common, unified interface to a disparate set of implementations or interfaces, such as within a subsystem, is required. There may be undesirable coupling to many things in subsystem, or implementation of the subsystem will change. What to do?
Solution	Define a single point of contact to the subsystem, a facade object that wraps the subsystem.

Observer pattern (GOF)(used most with UI):

Pattern name	Observer (Publish-Subscribe)
Problem	Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. How to retain low coupling?
Solution	Define a "subscriber" or "listener" interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

Controller pattern (GRASP):

Pattern name	Controller
Problem	What first object beyond the UI layer receives and coordinates a system operation ?
Solution	Assign the responsibility to an object representing one of the choices: 1: Represent the overall system, a root object. 2: Represent a use case scenario within which the system operation occurs.

16.8 Summary of using patterns

Use the patterns when designing the interaction diagram, either with focus on time or data.

1. The first pattern to use will be the *controller pattern;*
   (a) select the most important use case,
   (b) make a class with the same name as the use case,
   (c) let this class be responsible for the (main) operations of the use case.
2. The controller class will often need some information to be able to fulfil the responsibility;
   (a) Use the *information expert pattern* to make or use classes that will be responsible for the needed information,
   (b) Use the *high cohesion pattern* to separate the subfunctions in separate classes,
   (c) Use the low coupling pattern to let the controller class coordinate the messages (methods) in the system.
3. Use the creator pattern to let the controller class create the objects in use by the controller class.
4. Use the patterns fasade, polymorphisme and pure fabrication for a more layered or structured application.

16.8.1 *GRASP patterns overview*

GRASP General Responsibility Assignment Software Patterns

Name	**Controller**
Problem	Starting point of the application or use case, What controls the use-case?
Solution	Assign the responsibility to an object who: 1. represent the overall system (root object), 2. represent (control execution of) the use-case.

Name	**Information Expert**
Problem	Which object is responsible of knowing the specific information?
Solution	Assign the responsibility of knowing to the class holding that information.

Name	**Creator**
Problem	Which object creates another object?
Solution	Assign the responsibility of creation an object to: 1. Another object who “contains” it, 2. Another object who “records” it, 3. Another object who uses it (closely, meaning a lot), 4. Another object who knows the initialization parameters of it.

Name **Low Coupling**
Problem How to reduce the impact of changes in the code?
Solution Avoid "back and forth" messages, one way communication between objects.
Minimum interaction/knowledge /dependency between objects, "tree" structure.

Name **High Cohesion**
Problem How to keep objects focused, understandable, manageable?
Solution Objects should be focused about the contained task and not have multiple responsibilities, unless they belong together (tasks closely related),
Contain related functions in only one object,
Never store data in different locations,
Objects should cooperate to solve the overall task, but be responsible for their own part,
Assign responsibilities to suitable objects, or create new classes if no suitable object exists.

Name **Polymorphism**
Problem How to handle methods with different types of variables but same function?
Solution Use polymorphism overloads, and keep the function code only one place.

Name **Pure Fabrication**
Problem How to keep High Cohesion and Low Coupling?
Solution Create artificial classes when necessary to avoid high coupling. (typically interface classes are added to lower coupling).

Name **Indirection**
Problem How to avoid direct coupling between "things"?
Solution UUse separate class to mediate between components/services.

Name Protected Variations
Problem How to limit impact of changes, when changes can be predicted?
Solution Put functions with predicted changes (or instable functions, where design is not finalized or certain) and create a stable interface around it.

16.8.2 *GoF patterns overview*

GoF Gang of Four Design Patterns

Name **Adapter**
Problem How to handle classes with incompatible interfaces?
Solution Create a new artificial class, to wrap the incompatible class, in a new class that is compatible,

Name **Factory**
Problem Where to create objects, when the class is chosen at runtime by complex logic? (often a switch statement on object type)
Solution Create a new class (called factory) to create objects based on parameters/logic,

Name **Singleton**
Problem How to implement a class when only one instance of the class should exist?
Solution Use a static method to return the instance, and make constructor private so it is impossible to create multiple objects of the class.

Name **Strategy** (Very similar to protected variations)
Problem How to design for varying(able to change), but related, algorithms/polecies?
Solution Define each algorithm/polecy/stratergy in a separate class with a common interface (often common base-class with multiple overloaded versions),

Name	**Composite**
Problem	How to treat a group(composition) of objects the same way as atomic(non-composite) objects (by using polymorphism)?
Solution	Define a class for each composite group, implementing the same interface,

Name	**Facade**
Problem	How to avoid multiple connections between multiple objects between subsystems or layers of an application?
Solution	Define a single point (or a few) of contact between subsystems/layers, as interface objects or wrapper objects.

Name	**Observer**
Problem	How to handle several objects(subscribers) interested in the events/state changes of the same object(publisher)?
Solution	Define a subscriber interface in the receiving class, and register them with the publisher. Observer/subscriber willbe alerted on events/state changes by the publisher.

Chapter 17

Data Structures and Algorithms

17.1 Introduction

The data structure and algorithms are important when developing software. All software applications are dealing with data so the structure of the data and the algorithms for handling these data are important. The structure of the data is how to store the data internal in the application to be able to perform any operation on these data. When reading a data set from a database, how should the data be stored inside the application? When the data is stored inside the application, the application is going to perform some sort of operations on the data, how to define these algorithms? Typical algorithms can be creating, indexing, sorting, searching, adding, merging, extracting and deleting (or freeing). These algorithms can be for a single element in the data set, or the whole data set.

This chapter will look into different ways of structure data like array, stacks, queues and list, and sorting and searching algorithms on these data structures. The book (McMillan 2007) is a good reference on how to implement these data structures and algorithms using the C# programming language. This book is also used as basis for this chapter.

Assume reading a set of data combined from several tables in a SQL database, and doing some algorithms and analysis on the data before making a report. How to handle the data inside the software applications? Assume loading a set of files containing CSV (Comma Separated Values) used for analysis and developing a model. How to handle the data from reading one and one value from the file and saving the data in the applications? These are just two examples on software applications handling data. This chapter will deal with some of the basic data structures and algorithms that can be used. The information taken out of the data may depend on how the data are structured and organized, a type of relations are shown in Figure 17.1

17.2 Arrays

Arrays are the most common data structure and available in almost all programming language (McMillan 2007). Arrays are useful if the number of elements is constant or will not pass a maximum number of elements. However, the size of the array should always be allocated for the maximum number of elements

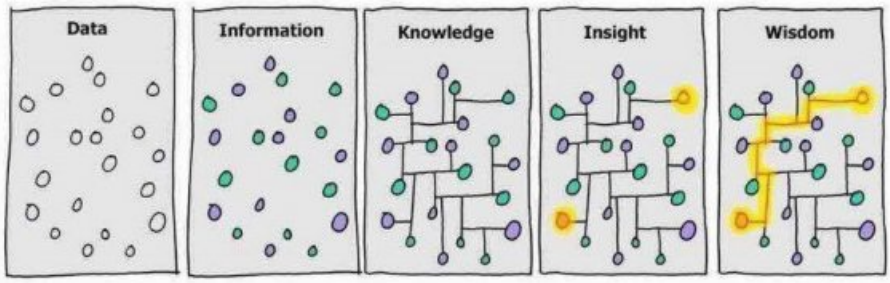


Figure 17.1: The possible type of information from data can be based on the structure and orginization of the data.

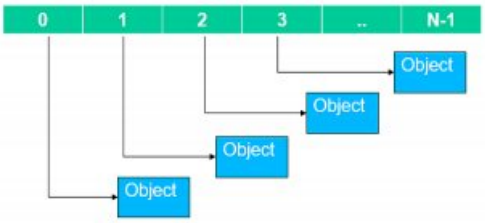


Figure 17.2: An array that can hold up to N parameters, however only 4 parameters are defined.

as changing the size of an array during the lifetime of the application should be avoided.

An arry consists of an indexed collections of data, either build-in or user-defined types. As most build-in types in C# are objects an array is just an indexed collections of objects. An array in C# is declared by using the syntax:

$$type[]\ arrayname;$$

where type is the data type and arrayname is the parameter. An array of strings can be defined as:

$$string[]\ CourseName;$$

however, this is just a reference to the array. Allocating the memory for the array, the maximum size must be known, a disadvantage by using an array. Allocating the memory is defined as:

$$CourseName = new\ string[100];$$

making an array capable of holding up to 100 CourseName parameters. Note that this statement only allocates the memory space for the array, not the contents of the array. As most of the build-in types in C# are objects, these objects will have several properties for the data type. An array is also an object having properties like:

- Length; get the maximum size of the array,
- Rank; get the number of dimensions for the array,
- GetType; get the type of array.

These are only some of the available properties for an array, check the type of properties of a specific object before making your own property or method. The property Length is important as access outside the array will cause an exception and the application will terminate if the exception is not handled. An array parameter can be accessed by using a valid index like:

$$sName = CourseName[4];$$

In this case is the index four, however it is important that the index is a valid index ($index >= 0\ \&\&\ index < CourseName.Length$) otherwise will the statement cause an exception. The index of an array is always starting at 0 giving that the maximum index will always be $Length - 1$. Figure 17.2 shows an array of N parameters and the last index is $N - 1$. An array can only be used by one data type, shown in the figure as an object. The type can be any type, defined when creating the array.

Creating this array will be like:

$$ArrayName = new\ Object[N];$$

where N is the number of Object parameters. Note that even if the array is created, the array is empty, contains only null references to any parameter. Each index in the array has to be populated with a reference to the Object.

The rank property gives the number of dimensions so multidimensional arrays can also be used. The normal use is one, two or three dimensions, but any dimension can be used. Declariation of a two dimensional array:

$$string[,]\ CourseInfo;$$

where CourseInfo can be the course code, course name, and name of responsible professor. In this case will the allocation of memory be:

$$CourseInfo = \ new\ string[100, 3];$$

giving that the first number is the number of rows and the second number is the number of columns.

As each element in an array is an object, array of standard objects can also be used. This example shows how to make a set of thread objects, stored in an array named *rThreadClass.*

```
int cnt;
ThreadClass[] rThreadClass;

// Making threads ..
rThreadClass = new ThreadClass[MAX_THREAD];
for (cnt = 0; cnt < MAX_THREAD; cnt++)
{
   rThreadClass[cnt] = new ThreadClass();
}
```

In C# there is also an ArrayList class that can be used if the maximum size is not known or will increase. An ArrayList starts with the size of 16 elements and will add 16 elements when more space is needed. Default the new elements are added to the end of the list, but can also be inserted into the index range at a specific offset.

17.3 Sorting

Sorting and searching are the two most common operations that are performed on data stored in an application. Most of the data used is sorted, the address list on your phone is sorted on first name or last name, the emails may be sorted based on the receiving time stamp or the name of the sender, addresses are sorted both by the zip code and the name of the street to mention some of the data used.

17.3.1 Bubble Sort

Bubble sort (McMillan 2007) is the first sorting algorithm to examine. The reason is not that it is used that much, it is one of the slowest sorting algorithms available (McMillan 2007). However it is very easy to understand and implement. The reason for the name is that there will be "bubble" that will travel along the numbers checking one and one number with the neighbour number. The "bubble" will enclose two and two numbers, check these two numbers, and swap them if necessary.

Assume that you have a (horizontal) array of mixed numbers. If the numbers are going to be sorted with the lowest number at the start (to the left) and the highest number at the end (to the right), any two numbers will be swapped if the right neighbour number is less then the left number. In the first iteration the whole array will be used. At the end of the first iteration the highest number will be at the end (to the right) of the array. Since the highest number is at the end of the array, the next iteration will handle all elements in the array except the last elements. Then the next iteration will handle all elements except the two last elements, and so.

The pseudo coding for the bobble sort can be:

```
Method BubbleSort(array aBuf)
{
  for (iCnt = 0; iCnt < aBuf.Length - 2; iCnt++)
  {
    for (jCnt = 0; jCnt < aBuf.Length - iCnt - 2; jCnt++)
    {
      if (aBuf[jCnt] > aBuf[jCnt + 1])
      {
        swap(aBuf, jCnt, jCnt + 1);
```

```
      }
    }
  }
}
```

Summary: Bubble sort will sort the highest number at the end of the array for each iteration. An example of performing the bubble sorting algorithm can be:

Iteration	[0]	[1]	[2]	[3]	[4]
(*start*)	17	8	1	15	5
0	8	17	1	15	5
	8	1	17	15	5
	8	1	15	17	5
	8	1	15	5	17
1	1	8	15	5	17
	1	8	5	15	17
2	1	5	8	15	17
(*end*)	1	5	8	15	17

where only the swaping actions are shown. The worst case of this sorting algorithm is $(n-1)^2$ swappings, in this case 16 swappings. The example shown ends up with 7 swappings.

17.3.2 Selecting Sort

This algorithm starts at the beginning of the array but is comparing the first element with other elements in the array. The smallest element is placed in the first location. The sort algorithm starts with the element in the second location and find the smallest element to place in this location. This continues until all positions except the last position has been handled. The pseudo coding for the selecting sort can be:

```
Method SelectionSort(array aBuf)
{
  for (iCnt = 0; iCnt < aBuf.Length - 2; iCnt++)
  {
    iMin = iCnt;
    for (jCnt = iCnt + 1; jCnt < aBuf.Length - 1; jCnt++)
    {
      if (aBuf[iMin] > aBuf[jCnt])
      {
        iMin = jCnt;
      }
    }
    swap(aBuf, iCnt, iMin);
  }
}
```

Summary: Selection Sort will find and sort the minimum value at the beginning of the array for each iteration. An example of performing the selection sort algorithm can be:

Iteration	[0]	[1]	[2]	[3]	[4]
(*start*)	17	8	1	15	5
$Min = 1$	1	8	17	15	5
$Min = 5$	1	5	17	15	8
$Min = 8$	1	5	8	15	17
(*end*)	1	5	8	15	17

where only the swaping actions are shown. The worst case of this sorting algorithm is $(n - 1)$ swappings, in this case 4 swappings. The example shown ends up with 3 swappings.

17.3.3 Insertion Sort

This algorithm is more like the way we are thinking. Start from the left side, and sort all the elements on the left side of each element. Move any elements to the right. In more detail, start with the second element, place it in the correct position, either position 0 or 1. If the corrent position is 0 move that element to position 1. Then the next element, the third element. Place that element in the correct position to the left, meaning position 0, 1 or 2. Move any elements already in position 0 or 1 to the right side.

The algorithm starts with picking the second element, and check that element with the first element. The element with the lowest value in position 0 and the element with the highest value in position 1. Then pick the third element, check the value against the elements in position 0 and 1, any move any elements to the right. This continues unntil all positions have been handled. The pseudo coding for insertion sort can be:

```
Method InsertionSort(array aBuf)
{
  for (iCnt = 1; iCnt < aBuf.Length - 1; iCnt++)
  {
    jCnt = iCnt ;
    while ((jCnt > 0) && (aBuf[jCnt] < aBuf[jCnt - 1]))
    {
      swap(aBuf, jCnt);
      jCnt--;
    }
  }
}
```

Summary: Insertion sort will sort the lowest number to the left of the aray, at the beginning of the array for each iteration. An example of performing the insertion sort algorithm can be:

Iteration	[0]	[1]	[2]	[3]	[4]
(*start*)	17	8	1	15	5
1	8	17	1	15	5
2	1	8	17	15	5
3	1	8	15	17	5
4	1	5	8	15	17
(*end*)	1	5	8	15	17

where only the swapping actions are shown. The swapping methods include a temporary variable to store the first elements, copy the second element into the first element, and copy the element in the temporary variable into the second element. The reader should try to make the pseudo coding for the swap method.

17.3.4 Sorting comparison

These three sorting algorithms are compared in (McMillan 2007) timing the results, in milliseconds, for an array of 100 elements, 1000 elements, and 10000 elements. The results are:

Type	ms	100	1000	10000
Bubble Sort		10.0	500.7	53607.1
Selection Sort		10.0	40.1	2864, 1
Insertion Sort		20.0	871.3	84751.9

The results shows that the Selection Sort is fastest, and that the timing is becoming imported at large size of data. An array of 10000 elements are not that large, but the timing differences are 2.8, 53.6 or 84.7 seconds. This shows the importants of evaluating sorting algortihms when sorting large amount of data.

17.4 Searching

Searching and sorting are the two most common operations that are performed on data stored in an application. Searching for a specific pattern in a set of data as a fundamental programming task, and there exist two ways of searching, the sequential search and the binary search, when searching for specific pattern in a data structure (McMillan 2007).

17.4.1 Sequential Searching

Sequential searching can be used both for lists with sorted items and lists with random items. The most used sequential searching is to start at the beginning of the list and go through every items unntil you find what you are looking for. Sometimes you want to search if an item exists, other times you also want to find the numbers of an item or the index of an item. The pseudo coding for sequential searching if an item exists can be:

```
Method SequentialSearch(array aBuf, int iVal)
{
  iValid = -1;
  for (iCnt = 0; iCnt < aBuf.Length - 1; iCnt++)
  {
    if (aBuf[iCnt] == iVal)
    {
       iValid = iCnt;
    }
  }
  return iValid;
}
```

This method will return -1 if the item do not exists or the item index if it exists in the array. What will be the solution if there exists several identical items? How to get the index for the first item if several identical items exists?

17.4.2 Find Minimum and Maximum Values

A common task is to find the minimum or maximum value in an array or other data structures, or even both values. In a sorted data structure this will be an easy task, the first and last items. However, in a random data structure this will be a little more challenging. Looking into finding the minimum first. Starts with assigning a variable the first element in the data structure (or the highest available value for the data type). Next loop through the data structure testing every items for the lowest value, and always store the lowest value in the variable. The pseudo coding for finding the minimum value in an array can be:

```
Method FindMinimumValue(array aBuf)
{
  iMinVal = aBuf[0];
  for (iCnt = 1; iCnt < aBuf.Length - 1; iCnt++)
  {
    if (aBuf[iCnt] < iMinVal)
    {
      iMinVal = aBuf[iCnt] ;
    }
  }
  return iMinVal;
}
```

This method returns the minimum value in an array. Alternative methods can be to return the index number instead of the value or both the index and the value. How to implement these methods?

Finding the maximum value will be the same logic except for the conditinal test. The pseudo coding for finding the maximum value in an array can be:

```
Method FindMaximumValue(array aBuf)
{
  iMaxVal = aBuf[0];
  for (iCnt = 1; iCnt < aBuf.Length - 1; iCnt++)
  {
    if (aBuf[iCnt] > iMaxVal)
    {
      iMaxVal = aBuf[iCnt] ;
    }
  }
  return iMaxVal;
}
```

Note that these minimum and maximum methods expect the array to have at least one item. These methods should be extended with a validation of the buffer length before assigning the first item to the variable. How to extend these methods with an array validation?

The iMinVal and iMaxVal is set to the first index of the array to have a valid value, however the maximum or minimum value range can also be used. The maximum range must be set for the minimum value before starting the search. In C# this can be coded as

$$iMinVal = int.MaxValue;$$

setting the integer instant to the maximum value an integer value can have. The same logic can be used for the maximum value, use the minimum value an integer can have, and also for double data types and other data types as well.

In large data structures can the sequential seaching be slow as it has to go through the whole structure. If the searched item is at the beginning of the structure the search will be fast, however if the searched item is located at the end of the data structure this search method will be slow as it has to go through most of the data structure. One way to increase the search speed is to sort the data structure first with the most use data at the beginning, called self-organizing data. Self-organizing data can be used if the data is following the "$80 - 20$" rule meaning that about 20% of the data are used in 80% of the searches. If the "$80 - 20$" rule is not followed, a binary seach can be a good solution.

17.4.3 Binary Search

Binary searching requires lists with sorted items. A sorted list has the items with the lowest value at one of the ends of the list, and the item with highest value at the other end. A sorted list may also contain several items with the same value. The algorithm is to divide the availble data "space" in half, binary

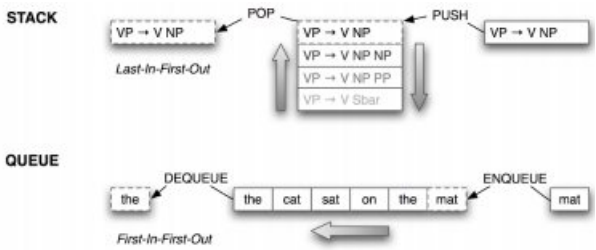


Figure 17.3: An overview of the stack and queue data structures (www.nltk.org: NOV-16).

search indicates two parts. The method is to find the number of items in the list, divide that number of items with *two*, and check the value of that index. If the value of the index is equal the value you want, the search is finished. If the index value is too low, do a new search in the upper part of the data. If the index value is too high, do a new search in the lower part of the data. Divide the new part of data in *two*, and check if the value of the index is lower or higher than the searched value. The maximum number of searches n will be

$$n = \log_2(items) + 1$$

showing that this will be an effective searching algorithm. 1000 items gives maximum 11 checks while one million gives maximum 21 checks. The pseudo coding for binary searching can be:

```
Method BinarySearch(array aBuf, int iVal)
{
  upperBound = aBuf.Length;
  lowerBound = 0;
  iValue = -1;
  while ((lowerBound <= upperBound) && (iValue == -1))
  {
    mid = (upperBound + lowerBound) / 2;
    if (aBuf[mid] == iVal)
    {
      iValid = mid;
    }
    else if (aBuf[mid] > iVal)
    {
      upperBound = mid - 1;
    }
    else
    {
      lowerBound = mid + 1;
    }
  }
  return iValid;
}
```

17.5 Stacks and Queues

The data structure in an array is a linear addressing space where all items can be accessed. Stacks and queues have a different approach. Both data structures are based on a data buffer where the data can be added or removed, but only one item at the time. The data are added and removed from only one end in a stack, while the data are added at one end and removed form the other end for a queue. Figure 17.3 shows these two data structure and the way of storing the information in the data structure.

The stack data structure is also known as the LIFO (Last In - First Out) data structure as the data is added and removed at the same location. The queue data structure is also known as the FIFO (First In - First Out) data structure as the data is added at one end and removed from the other end. Stacks

and queues are useful when you need temporary storage for information, and want the information to be discarded after usage. Use Stack if you need to access the information in reverse order. Use Queue if you need to access the information in the same order that it is stored in the buffer.

17.5.1 Stack

Stack is a frequently used data structure and C# has a ready made class for stack buffers. Stacks are used when the wanted item is always the last item stored. Stack are used by the CPU when calling a method the application, or having an interrupt. Calling a method stores the return address on the stack, meaning that an application can call an almost unlimit number of methods and always return to the previous method.

The operations for added an item is *push*() and the operation for removing an item is *pop*(). the C# class has several additional operations and properties like 1) *count* = the number of items in the buffer, 2) *peek*() = look at the item on the top of the buffer without removing it, and 3) *clear*() = removes all the items from the buffer. A default size of the stack is used when creating the stack object and the size is increased automatically when needed.

An example of using a stack can be:

Item	**Operation**	[0]	[1]	[2]
5	push	5		
17	push	17	5	
345	push	345	17	5
345	peek	345	17	5
345	pop	17	5	
17	pop	5		
7	push	7	5	

Note that C# is object oriented so the items will be objects even if it is shown only with a number in the example, in C# numbers are also treated as objects.

17.5.2 Queue

Queue is a frequently used data structure and C# has a ready made class for queue buffers. Queues are often used as an internal buffer or as an interface between two modules, where the first item to be stored is the first item that is wanted. The operation for adding an item is *enqueue*() and the operation for removing an item is *dequeue*(). Like the stack class has the queue class several additional operations and properties like 1) *count* = the number of items in the buffer, 2) *peek*() = look at the first item in the buffer without removing it, and 3) *clear*() = removes all the items from the buffer. A default size of the queue is used when creating the queue object and the size is increased automatically if needed.

An example of using a queue can be:

Input	**Operation**	[3]	[2]	[1]	[0]	**Output**
This	enqueue				This	
is	enqueue			is	This	
Hello	enqueue		Hello	is	This	
	peek		Hello	is	This	This
World	enqueue	World	Hello	is	This	
	dequeue		World	Hello	is	This
	dequeue			World	Hello	is
example	enqueue		example	World	Hello	

A message queue can be used as an interface between two systems or even two sub systems or modules within a system. The message queue is a queue used for exchanging messages, based on some sort of protocol, between the systems. As a queue is defined as a FIFO this will ensure that the receiving system will read the protocol messages in the same order as the transmitting system is writing the messages. The queue will be used as a buffer as the transmitting system can write whenever it has time, and the receiving system can read when ever it has time, hence no time syncronization is needed between the systems. Two sets of queue buffers are needed to handle communication both ways.

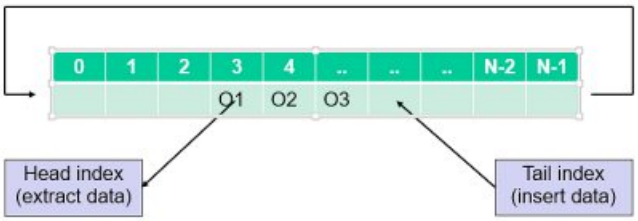


Figure 17.4: A ring buffer as an array with a head and tail index.

17.5.3 Ring Buffer

Ring buffer is a frequently used data structure and is a special version of the queue buffer. A ring buffer is also called a circular buffer. A ring buffer is a queue buffer with a fixed size, and the last input item will overwrite the first item when full. A ring buffer is useful when you are only interested in the last n events or you need to save buffer space. A queue has a fixed input location and a fixed output location in the buffer, while the ring buffer will have a dynamically input and output location.

A ring buffer is not a standard class in C# but can be implemented as an array with an input and an output index. Figure 17.4 shows a ring buffer consisting of maximum N items, with a head index to indicate the start of the buffer, and a tail index to indicate the end of the buffer. Both the head index and tail index will increase when used, and will handle the array buffer as a ring buffer. If the buffer contains more than N items the oldest items will be overwritten.

The algorithms for reading dato from the buffer, and writing data to the buffer must handle the array size and number of items in the buffer. The pseudo coding for inserting data in a ring buffer can be:

```
Method InsertVal(array aBuf, int iVal, int iTailIndex, out int iHeadIndex)
{
  abuf[iTailIndex] = iVal;
  iTailIndex++;
  // check for a ring buffer
  if (iTailIndex >= aBuf.Length)
  {
    iTailIndex = 0;
  }
  // check for a full buffer
  if (iTailIndex == iHeadIndex)
  {
    iHeadIndex++;
    if (iHeadIndex >= aBuf.Length)
    {
      iHeadIndex = 0;
    }
  }
  return iTailIndex;
}
```

The pseudo coding for removing data from a ring buffer can be:

```
Method GetVal(array aBuf, out int iHeadIndex, int iTailIndex, out bool bEmpty)
{
  if (iHeadIndex == iTailIndex)
  {
     bEmpty = true;
     iVal = -1;      // An error code
  }
  else
```

```
  {
     iVal = aBuf[iHeadIndex];
     iHeadIndex++;
     if (iHeadIndex >= aBuf.Length)
     {
       iHeadIndex = 0;
     }
     bEmpty = false;
  }
  return iVal;
}
```

The number of items in the ring buffer is the difference between the head index and tail index. Try to make the pseudo coding for estimating the number of items in the ring buffer.

17.6 Strings

Strings are common in almost every software applications and consists of any charcater based on a character set. A string can also consist of no characters. In C# a string is an object so there exists additional methods and properties for each string. A string has a fixed data structure and changing the string meaning making a new object with the new string, and the old object is discarded. The additional methods include *trim*() to remove unwanted characters at the start or end of a string, the *split*() and *join*() methods to split a string based on some critera or join several strings into one string, several methods for comparing the string or part of the string with other strings, and other methods for manipulating the string.

Many objects in C# has the method *toString*() to convert information of the object into a printable string. A string is a number of characters so a string of one character can be converted to a character. A string can be created by a number of charcaters, and a string can also be handled as an array consisting of characters.

17.7 Lists

Storing groups of related items of data is a basic requirement of most software applications and two ways of doing this are with arrays and lists. Arrays are collections of objects of the same type. Arrays can be almost any length and they can be used to store thousands or even millions of objects, but the size must be decided when the array is created. The maximum number of items must be known when creating the array. The items in an array can be accessed by an index, a number that indicates the position where the item is stored in the array.

Lists are also collections of objects of the same type and can also be almost any length. However a list do not need to know the maximum size when creating and a list is generally faster than an array when inserting items at the beginning or in the middle of the collection. List is a "build in" class in C# and a list is defined as:

$$List < type > tName = new\ List < type > ();$$

where type is the data type and tName is the parameter. An list of strings can be defined as:

$$List < string > CourseName = newList < string > ();$$

however, this is just a reference to the list. Allocating the memory for the list is done using some of the supporting methods like the Add() method. The syntax for adding an item to the list is defines as:

$$CourseName.Add("SCE1306");$$
$$CourseName.Add("SCE2006");$$

adding two items to the list. Some of the properties and methods build into a list objects are:

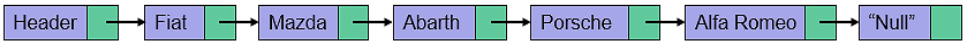


Figure 17.5: A linked list with data and link fields. The list starts with a header item and ends with a "null" item.

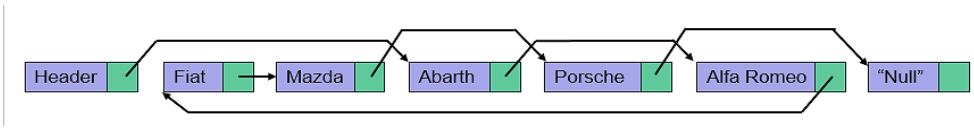


Figure 17.6: The linked list sorted by updating the links only.

- Count; get the number of items in the list,
- Item[index]; get or set a specific item in the list, accesed by the index number,
- Clear(); remove all items from the list,
- Insert(); insert an items at a specific index,
- RemoveAt(); remove an item from a specific index.

A list has several methods for manupilating the collections by sorting, searching, inserting and removal. Note that the example is shown with a string, but the type can be any object including objects created from classes defined in your software.

17.8 Linked Lists

In some situations a list where each item is connected to the neighbour items can be a good data structure. Such a list is named a linked list, each item in a linked list is named a node, the references between the nodes are named links, and each node is using a reference to the successor node. Each node consists of a field with data (reference to an object) and a field for the node reference (link). An example of a linked list is shown in Figure 17.5.

A linked list must have a header item as the first node in the list and a "null" as the last node in the list. The major difference between an array (or list) and a linked list is that an item can be referenced by the index in an array (or list) while in a linked list is the reference the relationship with other nodes. Any operation on a linked list like sorting, inserting and deleting nodes will be about handling the links. An example with sorting the linked list shown in Figure 17.5 is shown in Figure 17.6.

A linked list can also be used designed with links both ways (doble references) and also as circularly linked lists.

17.9 Multitasking

Data structures can be used as interfaces between several tasks in a multitasking systems. A common solution is having several tasks reading and/or writing I/O systems like communication systems or measurement systems. When reading data from the measurement system or receiving data from a remote computer systems, the data can be stored in a data structure, and a flag can be set to indicate new data. Another task can use the flag to figure out when new data has been received, do any calculation or estimation based on the new data, and update a new data structure and a flag. A display task and/or an output task can use the both the input data and the calculated/estimated data for update on display, storing in a database, or transmitting to another computer system.

The challange with this solution is the usage of the data structures as these data structures are used as an interface between tasks in the system. Such a data structure is also named a shared data structure or shared data buffer as it is shared among several tasks. This solution requires some sort of synchronization between the tasks as any task can be aborted or started in a multitasking system. Any task writing to a data structure needs to update the whole data structure before another task can read the data from the same data structure. One type of synchronization is the usage of a semaphore, a

service of the operating system. A semaphore can by a binary flag indicating if the data structure can be used or not. If the data structure is in use, indicated by the semaphore, the requesting task will be put to sleep by the operating system unntil the other task has finised using the data structure.

C# has a Semaphore class that can be used for synchronization of data structures. The class has the method *WaitOne()* to request access to the data structure and the method *Release()* to release the access to the data structure. It is however very important to remember to release the semaphore after finishing using the data structure as other tasks will wait for the semaphore before using the data structure. The pseudo coding for reading any information from a shared data structure:

```
Method GetNodeData(out array aBuf, int iMaxLen)
{
  // Request access to data structure from node A, put to sleep if not free
  RequestSemaphore(semaphNodeABuf);
  for (iCnt = 0; iCnt < iMaxLen; iCnt++)
  {
     // Get the data from a specific offset in the data structure
     aBuf[iCnt] = GetDataFromCommonBuffer(NodeA, iCnt + Offset);
  }
  ReleaseSemaphore(semaphNodeABuf);
}
```

The method will first request a valid access to the shared data structure by using a semaphore. If the semaphore is taken, the task will be put to sleep by the operating system (scheduler) unntil the semaphore is released by the other tasks. Then the task will be put back into the running state and can get the data in a secure way. This way the data will always be updated in a correct manner when accessed by any task.

17.10 Example

In experiments it is common to save samples in CSV (Comma Separated Values) based data files and performing several experiments the results can be a lot of CSV based data files. A software application that can read a set of CSV files for 1) merging them into one file for further analysis, 2) merge them into two files, one training set file and one validation set file for development of models, and/or 3) for importing into a SQL database can be very useful. Reading several CSV files a data structure must be used to store the data from each file before merging the data into a new file or writing the data to a database. A CSV file consists of several columns and rows, where each column is separated by a character, often a comma or semicolon (SSV). An extract of such a file can be:

```
No;Time;Interval(mi);Indoor Humidity(%);Indoor Temperature(DegC);Outdoor Humidity(%);...
1;25-04-2016 21:35;30;48;10.9;26;0.9;952.3;1.1;2.5;N;996.4;-16.5;0.9;70.8;70.8;70.8;....
2;25-04-2016 22:05;30;48;10.7;35;-1.2;952.5;1.1;2.5;N;996.6;-14.7;-1.2;70.8;70.8;70.8;..
3;25-04-2016 22:35;30;47;10.5;33;-2.2;952.7;1.1;2.5;NE;996.8;-16.3;-2.2;0.0;70.8;70.8;..
4;25-04-2016 23:05;30;47;10.3;41;-2.5;953.0;2.5;3.6;NE;997.1;-14.0;-2.5;0.0;70.8;70.8;..
5;25-04-2016 23:35;30;47;10.1;41;-2.3;953.1;2.5;3.6;NE;997.2;-13.8;-2.3;0.0;70.8;70.8;..
.....
```

where the first line is the header line with a description of each column and then one row containing each sample. Each row contains the sample time and the data according to the header. This means that the number of columns will be fixed, while the number of rows depend on the number of samples in each file. Each file from the same experiment will have the same header in each file, and the same number of columns. What kind of data structure will you use to store the CSV data?

The header line consists of text only and has a fixed length. The value lines consists mainly of values, but may also be text as shown in the extract. The number of values will be fixed, while the number of samples can vary. It can also be useful to sort the values based on the sampling time. Since the number of columns are fixed an array can be used, while the number of rows are unknown when starting so a list can be an option. Object-oriented analysis and design must be used, so objects should be used.

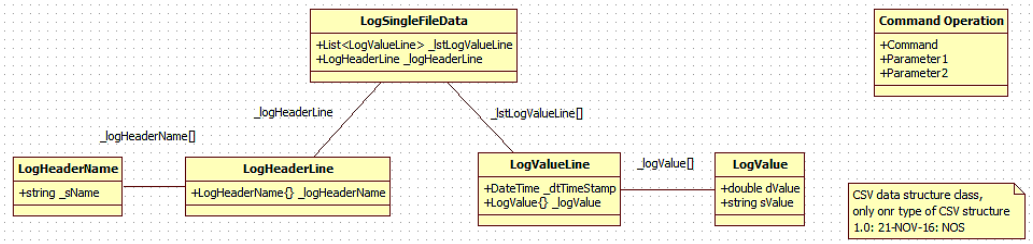


Figure 17.7: The OO data structure for storing data from CSV files, based on the Information Expert and High Cohesion patterns.

A solution can be to make with one object for each field in the header, and make an array of these objects for the header line. This will be based on the information expert pattern. The same pattern can be used to make one object for each value for the sample lines. However this object should have two attributes to store the value as both a value and a text string. Make an array of the value objects to hold one sample. Each sample contains also the sampling time. Using the information expert pattern to make a line class for the sampling time and the array of values. Using the information expert pattern to make a class responsible for the total data structure for all the CSV files. Figure 17.7 shows the class diagram for this data structure, combining classes (or objects) with arrays and list data structures.

The reading of files often takes some time so it is very useful to include a separate task (background task) reading the files and storing the information into the data structure. Since the reading operation is a slow process a CommandOperation class is included for handling the operation requests from the user. This class should be implemented as a queue data structure where the user can write any commands to the object and the task will read the next operation from the queue. In this way can the user queue a number of operations that the task will handle these operations asynchronous, but in the same sequence as the user selected them.

The data structure combines an array data structure for the data fields in each line, both header and values, a list data structure for each line of samples as the number of lines are unknown and should be sorted on the sampling time, and a queue buffer to hold the user operations as some of the operations may take some time.

Chapter 18

Data Security

18.1 Introduction

Data security refers to the process of protecting data from unauthorized access and data corruption throughout its life cycle. Data security includes data encryption and data management practices that protect data across all applications and platforms. The data management practices should include an "Identification and Access Management" (IAM) system (Daler, Gulbrandsen, Høie & Sjølstad 2019) handling 1) the identification of the users, 2) the access rights to the data of each user, 3) maintenance of identification, and 4) maintenance of the access rights. Part of the data management practice is to have information and control of where the data is stored, a valid backup of the data, and verify that each part of the data is only stored in one location. Duplication of data should be avoided. Part of defining the access right for each part of the data (write and/or read) is to define the owners of the data.

The most used method for identification is by passwords and a password should be of minimum eight characters and should contain both characters and numbers. Passwords should always be handled encrypted inside a system and never store the password as plain text in a database. Combining the password with another personal device like a mobile phone in a two steps login process will make the system more secure. First step can be to login with the user name and password and the second step using a random number sent in a SMS to the mobile phone from the application. In this case there will be a better identification by using two different personal informations, password and the mobile phone.

18.2 Software security

Software security is about making software application that can handle any attach. Software security has to be part of the development of the software, starting from the analysis phase. Software security can not be added to an application after the application is designed and deployed. Therefor the security aspects must be added to the software using the development process. Figure 2.4 shows the agile development process and Figure 18.1 shows this agile development process with security activities added (or included) (Jøsang 2021), (Viega & McGraw 2002).

The STRIDE analysis (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service and Elevation of privilege) (Ansari, Pandey & Alenezi 2018), (Jøsang 2021) is one method that can be used for the security activities in the development process. The OWASP (Open Web Application Security Project)[1] is an organization working with security on Internet and offers advice, guidance and tools to strengthen application security (Jøsang 2021).

Some advises, based on (Cruzes, Jaatun, Tøndel & Bersmed 2020), are 1) define what is "good enough" security for the application, 2) study and evaluate the existing security activities used when developing inhouse software, 3) make sure that security activities are explicitly added to the development process, 4) evaluate roles and responsibilities for software security, 5) create your own training program for the developers, 6) find ways to let the developers start thinking like an attacker, and 7) Introduce security tools into the workflow (development process).

[1] https://owasp.org (DEC-21)

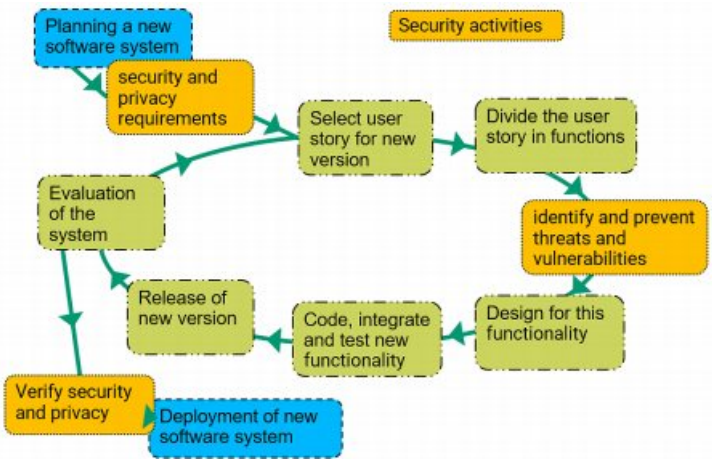


Figure 18.1: The agile development process for development of software with security included (Jøsang 2021).

18.3 Grading of information

Companies should have a good overview of the information needed and know which part of the information that represents a great value and should be protected and which is less valuable. It is not possible to protect the information 100% but the company should try to grade the importance of the information based on value. There exists different types of systems (Daler et al. 2019) for grading of information, an often used system is to divided into three categories like 1) unclassified, 2) protection #1 and 2) protection #2. Unclassified information is information open to public. Protection #1 is partly protection of information and apply to information that do not need to be available for all employers in a company. This protection is also named confidential. Protection #2 is for the information that is of a great value for the company and should be available only for a few employers. This protection is also named secret.

It is important for a company to have knowledge about the type of information used and where this information is stored. It is also important that the information is not duplicated into several locations and that the company have backup procedures for making copies of the information.

18.4 General Data Protection Regulation (GDPR)

The General Data Protection Regulation (GDPR) is a regulation protecting the data of citizens within the European Union and other countries in Europe including Norway. It will affect every company making business within these countries. This regulation is protecting the information about the citizens that every computer system is storing, saying that the owner of the computer system should protect the data, know the type of data stored, and be able to delete all the data whenever a specific citizen request to be removed from a specific system. The type of information that is protected by GDPR is name, photo, email addresses, social media posts, personal medical information, IP addresses, bank details to mention some (Daler et al. 2019). The GDPR covers any information that can be classified as personal details or that can be used to determine the identity of a citizen. Parental consent will be required to process any data relating to children younger than 16. Have the GDPR in mind whenever designing software or systems containing information about persons.

18.5 Encryption / Decryption

Encryption is the process of translating plain text data into something that appears to be random and/or meaningless type of information, also called ciphertext. Some sort of rules must be used in the encryption process to be able to convert the ciphertext back into the same plain text. Decryption is this process, converting ciphertext back into the plaintext. Figure 18.2 shows the processes of encryption and decryption. The quality of the encrypted message depends on the strength of the encryption algorithm and the secrecy of the encryption key.

Figure 18.2: The processes of encryption and decryption using an algorithm to convert between plaintext (cleartext) and ciphertext (kryptophone.kryptotel.net - Nov-19).

There exist two main types of encryption algorithms, symmetric and asymmetric. In the symmetric algorithm will the same key be used for both the encryption process and the decryption process. One simple symmetric algorithm is the substitution algorithm where every character in a text is replaced with another character. The key is a character substitution table. The first used symmetric encryption algorithm for public use was the Data Encryption Standard (DES) in 1977 with a key length of 64 bits. 64 bits are too short and the algorithm was replaced by the Advanced Encryption Standard (AES) with key length of 128, 192 or 256 bits. Both DES and AES is symmetric encryption algorithms and is based on only one key. The goal of every encryption algorithm is to make it as difficult as possible to decrypt the generated ciphertext without knowing the key.

Asymmetric algorithms are based on two keys, one public key and one private key. The encryption process is using the public key meaning that everybody can perform the encryption process. The decryption process can only be performed with the private key, so only the owner of the private key can decrypt the text. The RSA crypto system (developed by Rivest, Shamir and Adleman in the 80's) is one of the most used asymmetric algorithm and is widely used for secure data transmission.

18.5.1 Character rotation

The encoding/decoding process started with character rotation, rotating the characters in the alphabet a number of location to the right or left. The key could be like 4R or 4L defining a rotation of four characters left or right. An example of 4R is:

A	B	C	D	E	F	.	.
W	X	Y	Z	A	B	.	.

with removing of all spaces and all characters converted to uppercase. The text "An example of " will then be converted to "WRABWQTPASB". An extension of the character rotation is a random mixture of the character positions in the alphabet, however in this case must the complete table be the key. A challenge with this encoding/decoding process is space and special characters.

18.5.2 XOR encoding/decoding

There exists several software libraries and modules for data encryption and decryption, and such tools should be used if available, both for storing important information on local servers and for data transmission. When handling important information like passwords the information should be encrypted in some way before storing. In the absence of anything better the xor encoding and decoding can be used. This encoding/decoding is a symmetric process require only one key as the xor operation changes the same bits to a zero bit and different bits to an one bit. The table shows the xor encoding and decoding processes using single bits, and the xor operation is a single and fast operation in many programming languages, including C#.

Text bits	Key code	Encoding	Key code	Decoding
0	0	0	0	0
0	1	1	1	0
1	0	1	0	1
1	1	0	1	1

The decoding column will be the decoded text, the same as the text bits column. The encoding column will be the crypted information. The key can be any length but should be a key that is easy to remember. The listing shows a console based C# application that takes an input text and key of any length, shows the text after the encryption process and the text after the decryption text. The source code of such an applications can be (all comments are removed to save space):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
//
namespace CryptApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string sUsrTxt;
            string sEncrypt, sDecrypt, sKey;

            Console.WriteLine("\n\n  -- Example of using XOR as an encryption/decription method --")
            Console.Write("  Input the text to encrypt:");
            sUsrTxt = Console.ReadLine();
            Console.Write("  Input the key for encryption:");
            sKey = Console.ReadLine();
            Console.Write("   Simple checksums: Byte=0x{0:X}", CalcChecksumByte(sUsrTxt));
            Console.WriteLine("  Int=0x{0:X}", CalcChecksumInt(sUsrTxt));
            Console.WriteLine("  Encrypting ..");
            sEncrypt = crypt(sUsrTxt, sKey);
            Console.WriteLine("   Encrypted Text=" + sEncrypt);
            Console.WriteLine("  Decrypting ..");
            sDecrypt = crypt(sEncrypt, sKey);
            Console.WriteLine("   Decrypted Text=" + sDecrypt);
            Console.ReadKey();
        }
        static private string crypt(string sMessage, string sKey)
        {
            string sCrypt;
            int iMsgCnt, iKeyCnt;
            char cCrypt;

            iMsgCnt = 0;
            iKeyCnt = 0;
            sCrypt = "";
            while (iMsgCnt < sMessage.Length)
            {
                /// XOR coding of each character
                cCrypt = (char)(sMessage[iMsgCnt] ^ sKey[iKeyCnt]);
                sCrypt = sCrypt + cCrypt;
                iMsgCnt++;
                iKeyCnt++;
                if (iKeyCnt >= sKey.Length)
                {
                    iKeyCnt = 0;
                }
            }
            return sCrypt;
```

```
--- Example of using XOR as an encryption/decription method ---
Input the text to encrypt:this is an example text that is encrypted and decrypted using the xor method and a short key ..
Input the key for encryption:USN
 Simple checksums: Byte=0x50  Int=0x2250
Encrypting ..
 Encrypted Text=!;'&s'&s/;s+-2#%?+u'+-'n!;/!s'&s+;0<,#:07n4=*u7+6!7%'+1s;&: 2s:=6n-<<u>+!;!1s/;7n4s==<<!s%0*n{}
Decrypting ..
 Decrypted Text=this is an example text that is encrypted and decrypted using the xor method and a short key ..
```

Figure 18.3: The output from the XOR crypt application with a user selectable text and key.

```
        }
        static private byte CalcChecksumByte(string sMessage)
        {
            int iMsgCnt;
            byte bSum = 0;

            for (iMsgCnt = 0; iMsgCnt < sMessage.Length; iMsgCnt++)
            {
                bSum += (byte) sMessage[iMsgCnt];
            }
            return bSum;
        }
        static private uint CalcChecksumInt(string sMessage)
        {
            int iMsgCnt;
            uint iSum = 0;

            for (iMsgCnt = 0; iMsgCnt < sMessage.Length; iMsgCnt++)
            {
                iSum += (uint)sMessage[iMsgCnt];
            }
            return iSum;
        }
    }
}
```

The output of the application is shown in Figure 18.3 with the input text and key selected by the user, the output from the encryption process and the output from the decryption text.

As shown in the figure the input text and the decryption text is the same, while the encryption text will take some time to decrypt if the encryption method and the key is unknown. All encrypted texts are possible to crack, it is just be a matter of time. If the data should be protected, use some sort of encryption method to delay the time to crack the information.

18.5.3 PGP

In the beginning of the 1990s Philip Zimmermann (at Florida Atlantic University) developed an encryption and decryption algorithm for communicate and store files in open networks more securely. The method is called *Pretty Good Privacy* (PGP) and can be used without involving any official authorities, using a type of symmetric algorithm. PGP can be used for encrypting and decrypting texts, emails, files, directories, and even whole disk partitions. PGP is independent of the operating system and is available in many different software modules and libraries like OpenPGP[2], Gpg4win[3] and PGPCore. PGPCore is available as a NuGet package[4] for C#, one usage of the library methods is shown in Figure 18.4.

The PGP algorithm is using a random generated one-off key for encryption of the data, known as the session key. This type of encryption is very efficient, but the challenge is the session key. PGP solves

[2] www.openpgp.org: Nov-2019
[3] www.gpg4win.de: Nov-2019
[4] www.nuget.org/packages/PgpCore: Nov-2019

```
using (PGP pgp = new PGP())
{
    // Generate keys
    pgp.GenerateKey(@"C:\TEMP\keys\public.asc", @"C:\TEMP\keys\private.asc", "email@email.com", "password");
    // Encrypt file
    pgp.EncryptFile(@"C:\TEMP\keys\content.txt", @"C:\TEMP\keys\content__encrypted.pgp", @"C:\TEMP\keys\public.asc", true, true);
    // Encrypt and sign file
    pgp.EncryptFileAndSign(@"C:\TEMP\keys\content.txt", @"C:\TEMP\keys\content__encrypted_signed.pgp", @"C:\TEMP\keys\public.asc", @"C:\T
EMP\keys\private.asc", "password", true, true);
    // Decrypt file
    pgp.DecryptFile(@"C:\TEMP\keys\content__encrypted.pgp", @"C:\TEMP\keys\content__decrypted.txt", @"C:\TEMP\keys\private.asc", "passwor
d");
    // Decrypt signed file
    pgp.DecryptFile(@"C:\TEMP\keys\content__encrypted_signed.pgp", @"C:\TEMP\keys\content__decrypted_signed.txt", @"C:\TEMP\keys\private.
asc", "password");

    // Encrypt stream
    using (FileStream inputFileStream = new FileStream(@"C:\TEMP\keys\content.txt", FileMode.Open))
    using (Stream outputFileStream = File.Create(@"C:\TEMP\keys\content__encrypted2.pgp"))
    using (Stream publicKeyStream = new FileStream(@"C:\TEMP\keys\public.asc", FileMode.Open))
        pgp.EncryptStream(inputFileStream, outputFileStream, publicKeyStream, true, true);

    // Decrypt stream
    using (FileStream inputFileStream = new FileStream(@"C:\TEMP\keys\content__encrypted2.pgp", FileMode.Open))
    using (Stream outputFileStream = File.Create(@"C:\TEMP\keys\content__decrypted2.txt"))
    using (Stream privateKeyStream = new FileStream(@"C:\TEMP\keys\private.asc", FileMode.Open))
        pgp.DecryptStream(inputFileStream, outputFileStream, privateKeyStream, "password");
}
```

Figure 18.4: An example of using a PGP library in C# for encryption and decryption (From blog.bitscry.com: Oct-18)

this challenge with public key cryptography, also known as asymmetric cryptography. Public keys can be found on key servers or by asking the person directly. Public keys are normally used by the sender to encrypt data, but they can not decrypt the data. Normally only the private key can decrypt the data. In PGP the public key is used to encrypt the session key only, not the data. Encrypt the data with the public key uses a larger amount of computational resources and takes longer time than using the session key. The receiver needs to receive both the encrypted data, encrypted with the session key, and the encrypted session key, encrypted with the public key. Only the private key of the receiver can decrypt the session key, and only the decrypted session key can be used for decrypt the data.

Note that there has been several versions and features of the PGP algorithm over the years, meaning that older versions of PGP algorithms will not be able to decrypt messages that has been encrypted by newer versions. Both the sender and the receiver must be aware of which version and features that is used for decrypting the message. The PGP algorithm is not perfect but is known to be one of the safer ways to communicate online.

18.5.4 Block chains

Block chains are a special type of distributed databases containing a set of blocks (posts) connected in a chain with a crypted connection between each block. Each block contains a cryptically hash value of the previous block in the chain, a time stamp and transaction information (Mazonka 2016). Block chains can be used for digital descriptions of a long set of transaction where none of the blocks can be changed without changing all subsequent blocks, as these blocks contains a cryptically hash value of the previous block. The original block chain was the database for the electronic currency bit coin back in 2008, but are today also used for many other type of applications. The encryption process is based on an asymmetric algorithm, using two keys, one private key and one public key. A block chain can exists over several computers. The focus for block chains are different types of transactions and figure 18.5 shows a step by step view of a transaction that are added to an existing transaction block chain.

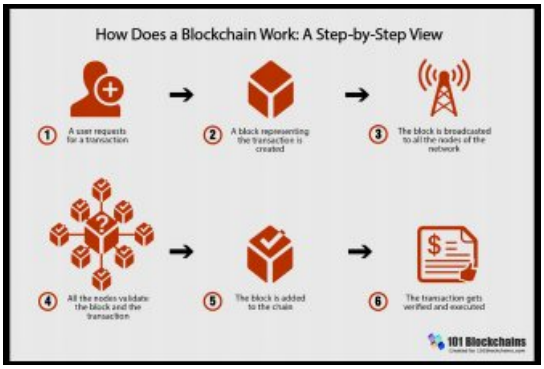


Figure 18.5: A step by step view of adding a transaction in to a block chain (copy from www.101blockchain.com: Nov-19).

18.6 Input fields for information

A common way of hacking software is try to input a large amount of data in any input field. The input data can just be random data to try to destroy the structure of the program or the binary code of another program to try to patch this new application overwriting the code of the existing application. The software designer should limit all input fields to the maximum length of the information that are valuable for that field and also take care when copy information from the user interface into buffers and/or structures in the business layer of the application. In object-oriented languages like C# and Java a data buffer is an object with boundaries to avoid overwriting the buffers while in programming languages like C and C++ no such buffer boundaries are implemented. So take special care when using non object-oriented programming languages to not overwriting the allocated buffer spaces.

The source code listed in 18.5.2 is using the Console.ReadLine() method for reading the information from the user, as strings. Included in the method is a maximum length of 256 characters meaning that these input methods are safe for overwriting any code or data in this application.

18.7 External access

Storing information on a server or transmitting information using a protocol, the data should be protected in such a way that you can verify if the data has been changed or not. Storing information on a server, in a file, a checksum is often used as a verification of the contents of the file. The same checksum can also be used if the file is moved from one location to another location, or downloaded to a new position. A tool is used to produce the checksum, putting the contents of the file through an algorithm. Typical algorithms used for this purpose include MD5, SHA-1, SHA-256, and SHA-512. These algorithms use a cryptographic hash function that takes an input and produces a string (a sequence of numbers and letters) of a fixed length. These algorithms produces checksum of the same length independent of the size of files. Different checksum algorithms produce different results given that a file will have different MD5, SHA−1, SHA−256 and SHA−512 checksum. These checksums are also called "hashes".

In development systems that do not have these checksum algorithms available, simpler algorithms can be used. The simplest form is just summarizing all bytes either as a byte or an integer (16 bits) in the string as shown in two of the methods in the source code in 18.5.2, the methods *CalcChecksumByte()* and *CalcChecksumInt()*. These methods are not part of the encryption nor the description process, just an extension for calculating the checksum of the input information. The checksum is one way of verifying that a specific set of information has not been changes, either when stored in a file on a computer, or when transmitted between several computers. Most protocols today, at higher levels of the OSI model (Houldsworth 1990), (Levermore 2000), have checksum included. Many of these protocols are using different versions of the cyclic redundancy check (CRC) method (McNamara 1988) like CRC−12, CRC−16 and CRC−32 to mention some. The number indicates the length of the generator polynomial used in the CRC algorithm.

18.8 Passwords

Passwords must NOT be stored in plain text, neither inside your software nor in any databases. A good practice is to crypt the password in the user interface when the user is entering the password, and always use the crypted version when checking if a valid password. Always start with checking if a password is needed, do you need to include a username and a password in your application. If a username and a password is needed, remember that the GDPR requirement must also be fulfilled for your application. If a password is needed, choose the appropriate crypt/decrypt methods for your software, and any database for storing the username, password and information to maintain the user and password information. Any minimum requirements for the password? Any logic for changing the password? How to handle users that have forgotten the password? How to handle users that want to be removed from your system? Note that according to GDPR a user may request to be removed from all systems in a company, not only for your application.

The most used passwords in 2020 and 2021, according to the "*Top 200 most common passwords*" list from Nordpass[5], are:

	Globally	**Globally**	**Norway**
	2020	2021	2021
1	123456	123456	123456
2	123456789	123456789	123456789
3	picture1	12345	12345
4	password	qwerty	12345678
5	12345678	password	passord
6	111111	12345678	1234
7	123123	111111	password
8	12345	123123	1234567
9	1234567890	1234567890	lol123
10	senha	1234567	abc123
11		qwerty123	123123
12		000000	qwerty
13		1q2w3e	111111
14		aa12345678	1234567890
15		abc123	liverpool

Be aware that any hacker will launch an attack by testing these passwords, and this is usually done automatically. So choose your passwords carefully. Some tips for choosing good passwords, according to Nordpass, are:

- Use a phrase in stead of a word,
- Use a sentence in stead of a word,
- Do not use the same "password" everywhere, better to use fixed parts that can be combined in different ways,
- Combine the "password" with numbers and special characters.

[5] www.nordpass.com (27-DEC-21)

Chapter 19

Programming

Object oriented programming is about encapsulation of data, inheritance and polymorphism. Encapsulation means to avoid global data, hide data inside classes and use methods or properties to get or set the data, or any other operations on the data. Polymerism means different interfaces solved by using the same method name, but with different number and type of parameters. The correct method will be resolved at run-time. Inheritance means to reuse one class into several other classes.

19.1 Inheritance

Inheritance is used when super and sub classes are identified in the analysis or design phase using among others the domain model. The main reason for using inheritance is to avoid duplication of code, meaning to make the code more maintainable. It will be faster and easier to just duplicate the code when writing the code, but the code will end up as a disaster to maintain. If changing or fixing the code, how to find all the duplication of the code?

One example of using super and sub classes is regarding different sensors connecting to a DAQ (Data Acquisition) device. A DAQ device normally convert the sensor device signal into a voltage signal in a predefined range (often $[-1\,\mathrm{V}, 1\,\mathrm{V}]$, $[-5\,\mathrm{V}, 5\,\mathrm{V}]$ or $[-10\,\mathrm{V}, 10\,\mathrm{V}]$) and each sensor type must convert the DAQ value into the correct unit value for the type of sensor devices. All sensor devices will get a voltage signal (in the same range) from the same DAQ device or several DAQ devices. The sensor class must have the same method for reading the value from the DAQ device before converting to the correct unit value. What if the range of the DAQ device is changed? What if another type of DAQ device is going to be used? To avoid changing all the different type of sensor classes, sub and super classes of sensor devices should be used.

Figure 19.1 shows the class diagram of a small application including three different type of sensor devices. Each sensor type will be a sub class and a new super class named sensor is defined. The rules for making sub and super classes are:

1. 100% rule: all the attributes and methods of the superclass should be applicable to the subclass; in our case we want the superclass to have the main information about the sensor device and get the voltage value from the DAQ device,

2. Is-a rule: Flow sensor is a sensor device, temperature sensor is a sensor device and level sensor is a sensor device.

The console application consists of four classes, as shown listed in Figure 19.1. All the classes are listed in the same source file module, normally there will be one source file for each class. Both the super and sub classes contain the GetValue() method. In C#, a method in a super class can have the same name as a method in the sub class. The methods interact by using either the virtual or the override keywords. The override modifier extends the base class method, while the virtual modifier hides it. Hide in this case means that the method is overridden by the method in the derived class. In this example the GetValue() of the superclass can be over written and both the virtual and the override modifiers are used. The source code of the application is listed (note that most of comments are removed to save space, normally each method and property should have a header with comments.

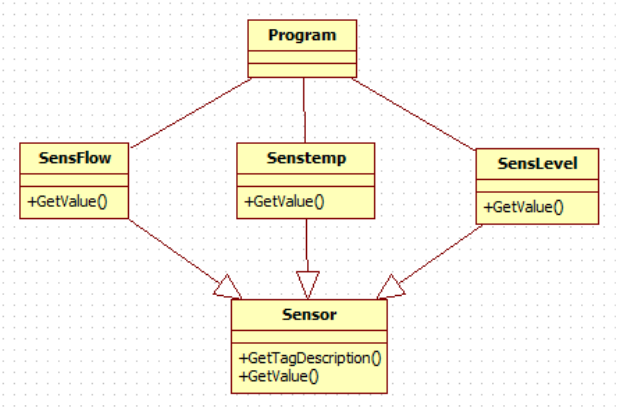


Figure 19.1: A class diagram for an application with sub and super classes for sensor devices.

```
////////////////////////////////////////////////////////////////////////////
//
// Application: SensorInheritance
//
// Purpose: show the inheritance in classes. The code is without comments to
//              minimize the text when showing in class.
//
// Version: 1.0: 19-NOV-14: NOV
//
////////////////////////////////////////////////////////////////////////////
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
//
namespace SensorInheritance
{
    ////////////////////////////////////////////////////////////////////////
    // Sensor base class
    // Used for reading a channel value from the DAQ system
    class Sensor
    {
        int chId;
        string sTag;
        Random rVal;
        double dVal;
        //
        public Sensor(int ch, string sinfo)
        {
            chId = ch;
            sTag = sinfo;
            rVal = new Random(ch);
            // Simulate the start 0-5V input from the DAQ
            dVal = rVal.NextDouble() * 5.0;
        }
        //
        public virtual double GetValue()
        {
            // Simulate a new reading from the DAQ device
            dVal += ((rVal.NextDouble() - 0.5F) / 10.0F);
            return dVal ;
```

```
        }
        //
        public string GetTagDescription
        {
            get
            {
                return sTag;
            }
        }
    }
    ////////////////////////////////////////////////////////////////////////
    // Temperature Sensor
    class SensTemp : Sensor
    {
        public SensTemp(int ch, string txt) : base(ch, txt)
        {
        }
        public override double GetValue()
        {
            double dVal = base.GetValue();
            // Convert DAQ voltage to temperature range : -25 to 125
            dVal *= 30F;        // Range 150/5
            dVal -= 25.0F;      // Offset adjust
            return dVal;
        }
    }
    ////////////////////////////////////////////////////////////////////////
    // Level Sensor
    class SensLevel : Sensor
    {
        public SensLevel(int ch, string txt) : base(ch, txt)
        {
        }
        public override double GetValue()
        {
            double dVal = base.GetValue();
            // Convert DAQ voltage to level maximum range: 0-100 meter
            dVal *= 20F;
            return dVal;
        }
    }
    ////////////////////////////////////////////////////////////////////////
    // Flow Sensor
    class SensFlow : Sensor
    {
        public SensFlow(int ch, string txt) : base(ch, txt)
        {
        }
        public override double GetValue()
        {
            double dVal = base.GetValue();
            // Convert DAQ voltage to flow range: 0-200 l/s
            dVal *= 40F;
            return dVal;
        }
    }
    ////////////////////////////////////////////////////////////////////////
    // Program Class
```

```
    class Program
    {
        static void Main(string[] args)
        {
            const int MAX_DEVICES = 4;
            int id, loopId;
            string sOutput;
            Sensor[] sid = new Sensor[MAX_DEVICES] ;

            for (id = 0; id < MAX_DEVICES; id++ )
            {
                switch (id)
                {
                    case 0: sid[id] = new SensTemp(id + 1, "InTemp"); break;
                    case 1: sid[id] = new SensTemp(id + 1, "OutTemp"); break;
                    case 2: sid[id] = new SensFlow(id + 1, "Flow"); break;
                    case 3: sid[id] = new SensLevel(id + 1, "Level"); break;
                    default: sid[id] = new SensTemp(id + 1, "Temp" + (id + 1).ToString());
                        break;
                }
            }

            Console.WriteLine() ;
            sOutput = string.Format("{0,10}", "Time");
            Console.Write(sOutput);
            for (id = 0; id < sid.Length; id++)
            {
                sOutput = string.Format("{0,10}", sid[id].GetTagDescription) ;
                Console.Write(sOutput);
            }
            Console.WriteLine() ;
            for (loopId = 0; loopId < 16; loopId++)
            {
                sOutput = string.Format("{0,10}", DateTime.Now.ToString("HH:mm:ss"));
                Console.Write(sOutput);
                for (id = 0; id < sid.Length; id++)
                {
                    sOutput = string.Format("{0,10}", sid[id].GetValue().ToString("0.00")) ;
                    Console.Write(sOutput);
                }
                Thread.Sleep(2000) ;
                Console.WriteLine();
            }
            Console.Write("Press any key to terminate ..");
            Console.ReadLine();
        }
    }
}
//
////////////////////////////////// EOA ////////////////////////////////////
```

The output from the application is shown in Figure 19.2, showing the time and the four sensor device values for each sample time.

As can be seen from the source code, the method for reading from the voltage from the DAQ device is defined only once. When changing the DAQ device or changing the range of the DAQ inputs, the source code needs to change only one method in one class, not all the sensor type classes.

The domain model will be a very good helping tool for finding sub- and superclasses in a system.

```
      Time     InTemp    OutTemp       Flow      Level
  10:40:52      11,13      90,38      59,49      82,58
  10:40:54      11,03      89,37      60,95      82,70
  10:40:56      11,85      90,83      59,75      82,53
  10:40:58      12,32      89,66      59,99      81,55
  10:41:00      12,12      89,08      58,71      80,66
  10:41:02      11,68      89,98      57,71      81,06
  10:41:04      13,01      89,82      59,50      80,96
  10:41:06      11,82      88,99      58,90      80,90
  10:41:08      12,24      87,53      58,42      81,40
  10:41:10      10,83      88,32      58,43      80,88
  10:41:12      10,07      86,91      59,66      81,06
  10:41:14       9,53      85,43      60,44      80,82
  10:41:16      11,00      85,46      58,56      80,92
  10:41:18      11,55      85,11      56,89      81,49
  10:41:20      12,01      84,45      58,51      81,54
  10:41:22      11,36      85,06      57,00      81,63
Press any ket to terminate ..
```

Figure 19.2: The output for four different sensor devices.

19.2 Interfaces

An interface looks like a class, but has no implementation. It contains only declarations of events, indexers, methods and/or properties. The interfaces are inherited by structs and classes that must provide an implementation for each interface member declared, so the interfaces are said to define a contract that the struct or class must fulfill.

The reason for using interfaces, even if they don't implement functionality, is that they are great for putting together structs and classes in architectures where components can be interchanged more simply. Since all interchangeable components implement the same interface, they can be used without any extra programming. The interface forces each component to expose specific public members that will be used in a certain way, defined as a contract. However, the most important difference is that while C# doesn't allow multiple inheritance, where classes inherit more than a single base class, it does in fact allow for implementation of multiple interfaces.

19.3 API

Application Programming Interface (API) is a way of defining how to interface to an existing application, to be able to exchange information with the application. API can be a set of defined subroutines that must be implemented, a communication protocols, or tools like using a csv file or a database to interact with a software application. The API must be documented to let software developers understand how to integrate with the software. In the world of IoT (Internet of Things) and IIoT (Industrial Internet of Things) internet or intranet will be used for communication with these devices. Internet or intranet is based on the TCP/IP protocol, a message based protocol, where another protocol must be used for defining the messages. This can be protocols like OPC UA, CoAP (Constrained Application Protocol), MQTT (Message Queue Telemetry Transport), XMPP (Extensible Messaging and Presence Protocol) or Modbus, or a service must be used. As there exists no good standard protocol for IoT communication often a proprietor protocol is used by a service, like a cloud service. The service will communicate with the IoT device, and your software must communicate with the service by some type of API.

One example of an API can be REST (Representational State Transfer), a way for two nodes to communicate over HTTP in a similar way to web browsers and servers. REST was defined in 2000 as a simple API, it is not a standard but contains a set of recommendations and constraints for web services communication.

Rest

19.4 Robotic Process Automation

Robotic Process Automation (RPA) is software robots that can automate boring and repetitive tasks in existing systems. These robots can handle manual operations that often takes a lot of time and/or are boring, mainly input of information. The type of information can be to handle invoices, get information

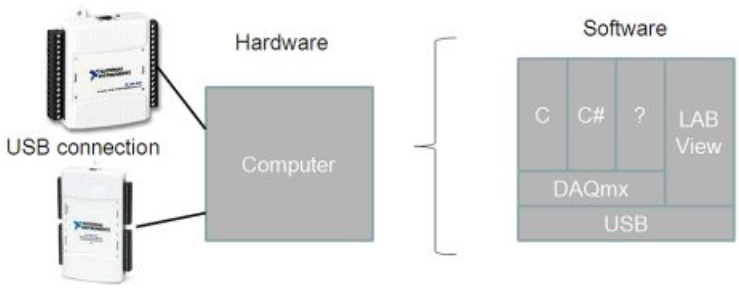


Figure 19.3: An overview of software and hardware using USB devices from NI.

from emails, handle written orders and booking, making reports and form operations in systems. RPA can be combined with Artificial Intelligence (AI) to be able to get more information out of the input material to make the information into the system as complete as possible. RPA can also be an important tool for digitalization. RPA must understand the type of information to input in each field in a form and how to navigate in the form, and is mainly a software application that is configured to replace the manual operations for input of information into a systems, normally by forms in the user interface. Several RPA tools exist like BluePrism, Pega, UIPath and Visual Cron to mention a few.

19.5 Design of a measurement system

A measurement system consists of both hardware and software. The hardware will be a computer device and some type of DAQ device. The DAQ device is the connection between the computer and any sensor devices and/or actuator devices, the connection between the computer "world" and the "real" world. The DAQ device consists of a combination of input and output ports, like Analog Inputs (AI), Analog Outputs (AO), Digital Inputs (DI) and Digital Outputs (DO). The software will be a combination of a driver or library for the DAQ device and an application communicating with this driver or library. DAQ devices are RTUs (Remote Terminal Units) and can have several shapes and forms. The DAQ device can be an Arduino card, a Raspberry PI card, a PLC (Programmable Logic Controller) or other devices using a specific protocol. Several useful protocol exists, however a protocol supporting layer 7 in the OSI model, like USB and Modbus TCP, is the best choice.

In this case study DAQ devices from NI (National Instruments) using the USB protocol will be used as an example. These DAQ devices can be controlled by the LABview software from NI. However, LABView has an integrated graphical design and configuration "tool" that may not be that useful in an industrial application. NI has a separate library named DaqMx handling the USB protocol and communication with the DAQ devices. The DaqMx library contains an API (Application Programming Interface) meaning that you can write your own software in an object orientated programming language like C#, instead of LABView, for getting the input values from any sensor devices, and write the output values to any actuators. The DaqMx library or DLL (Dynamic Link Library) can be downloaded from NI web site and configured at installation time with the API for the .NET platform (C# programming language).

Figure 19.3 shows the hardware and software needed for such a system. One solution can be to use LABView, another solution can be to make an application yourself to interface to the DAQmx library (DLL) shown in Figure 19.3. The same concept can be used to make interfaces to other type of DAQ devices like the Arduino platform or other type of platforms. More examples of DAQ using a PC are described in (Magda 2009) and (Magda 2011).

Making your own software in C, C# or other programming languages supported by the DAQmx library or other type of DAQ platforms, the following sections need to be addressed:

- configuration,
- calibration,
- sampling time(s),
- reading the sensor values,

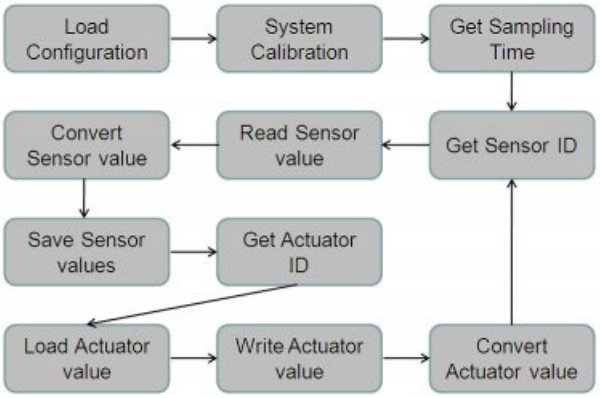


Figure 19.4: The main steps that should be addressed when designing a measurement system.

- converting the sensor values from the voltage value to a valid unit value, display and log the values,
- writing the actuator values,
- converting the output unit value to a valid voltage output value,
- error handling,
- looping reading all sensor values and writing all actuator values.

These sections are shown in Figure 19.4, indicating the sequence of the section in the software application. The application also need a configuration section to be able to add and remove sensors and/or actuators. The configuration and calibration data should be stored on files or in a database so the application can use stored data and reload data when starting and stopping.

The section are normally solved by looping through all the sensor devices and actuator devices, these devices are connected to different input and output channels on a DAQ device. Some DAQ devices can be used to read or write all channels in one command, other DAQ devices only each channel can be used at a specific time. The software can be divided into more details as shown in Figure 19.5. This Figure shows the startup sections and two main loops, one loop for handling all the sensor devices, and one loop handling all the actuator devices. Most of the systems today are sequential meaning that the system can handle only one type of input or output at a time. The software application must control the sequence. Also note that in this Figure the sampling time is checked for every sensor ID, some system can or must use a common sampling time, other system can use different sampling time for each channel. The sampling time can be controlled either by hardware or software, or both, depending on the design of the software.

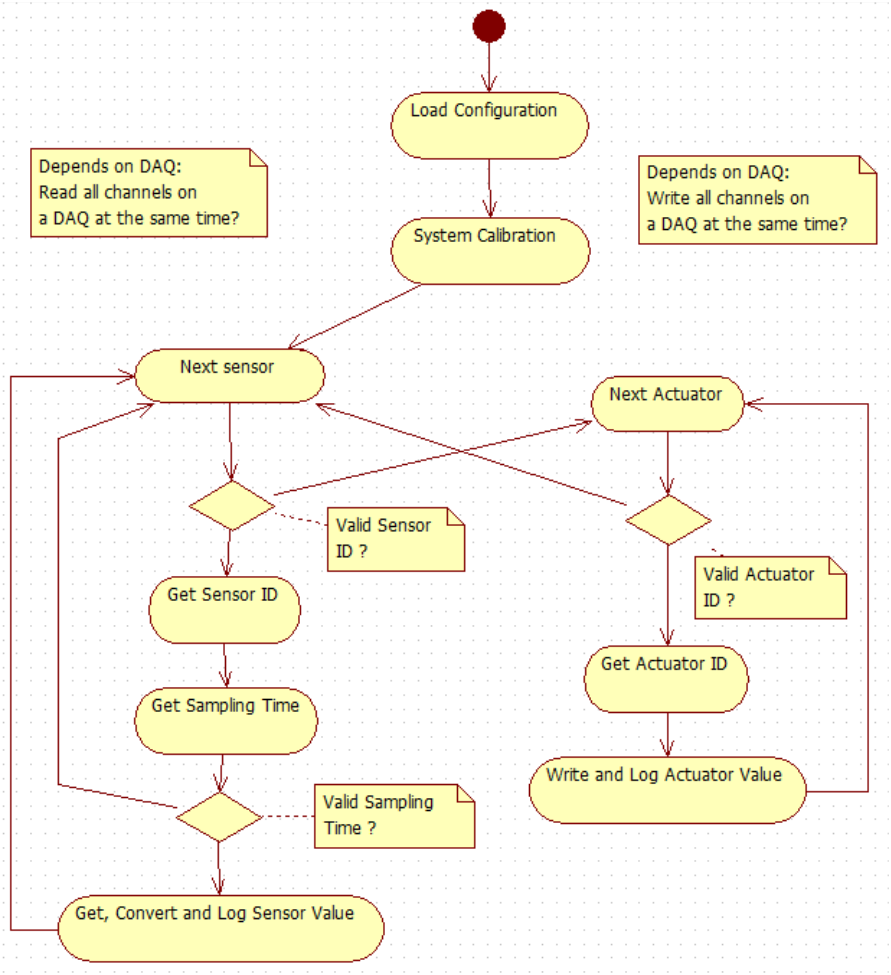


Figure 19.5: The main steps that should be handled in a DAQ measurement software, documented using an UML Activity diagram.

Chapter 20

Testing

20.1 Introduction

A test case in hardware engineering, software engineering or system engineering is a set of conditions or variables under which a tester will determine whether a application or system meets specifications. Testing means often validation of several combination of hardware and software functions. Testing of the software can be a challenge and will consist of several different type of tests.

20.2 Test Plan

A test plan is a systematic approach to test a system such as a machine or software, and must include the test coverage, the test methods, and the test responsibilities. The plan typically contains a detailed understanding of what the workflow will be. The test coverage defines what will be tested, test methods describes how each sections of the test coverage will be tested, and test responsibility assign the tasks and responsibilities. The test plan should contains at least three parts. Part one is describes the system to test, part two includes a detailed list of the test to be performed on this system, and part three as section to be signed by any testers. The test plan should also contain a PASS column to be marked by the tester, for every test. The test plan should also contain a comment field for any comments from the tester(s).

A test plan will be:

- system description and diagram; devices used and how the devices were interconnected. This part should describe in detailed how the system is used with all inputs and outputs defined. Any simulators/emulators should also be included, including any test databases used.
- test lists; test cases with pass and comment sections.
- responsibility section, stating who is doing the testing with a field to sign.

A test plan should be a technical document and should contain at least the following chapters:

- introduction; the reason for the test plan,
- system description; description of the system and/or devices to test,
- results; the test plan consisting of a set of test cases. The test cases can be in a table with four columns; index, description, pass, and no pass or comment.
- conclusion; the test conclusion with the name and signatures of the testers.

The test plan should contain detailed information in such a way that the tests can be redone with exact same results, meaning have the same input signals, combinations and results. Any results, OK or wrong, should be able to reconstruct based on the test plan. Why? One reason is to let the developer be able to have the same error(s) when trying to fix the error. Another reason is to be able to test the same error(s) after it has been corrected, how to be sure that the error(s) have been corrected?

Test cases are dealing with an input, an action and an output. The action acts upon the input to derive a specific output. A test case must be specific by defining the inputs, action and any outputs,

and the outcome should be OK or an error. The test cases are often structured in a table, and here is an example of such a table:

Id	Description	Pass	Comments
..			
5	Conversion from meter to feet		
5.1	Convert 0.75 m to 2.461 feet	Fail	2.460, no round up.
5.2	Convert 1.55 m to 5.085 feet	OK	
5.3	Convert 9.86 m to 32.349 feet	OK	
..			

The columns Id and Description are predefined, while the columns Pass and Comments have to be filled in during the test. This table shows the result after the testing. How can the following software method be tested?

```
public int CheckTypeStep(int val)
{
  if (val > 10)
  {
    ....
    val = 1 ;
  }
  else if (val < -10)
  {
    ....
    val = -1 ;
  }
  else
  {
    ....
    val = 0 ;
  }
  return val ;
}
```

Will this test case be a valid test of the function?

Id	Description	Pass	Comments
7	Testing of CheckTypeStep() function	____	
7.1	Input value = 23, Output value = 1		

Will these test cases be a valid test of the function?

Id	Description	Pass	Comments
7	Testing of CheckTypeStep() function	____	
7.1	Input value = 18, Output value = 1		
7.2	Input value = 5, Output value = 0		
7.3	Input value = −18, Output value = −1		

What will be the minimum number of tests or test cases?

The software method contains three *if* - *else* blocks, the minimum will be to have one test of each block. Sometimes the limits are important meaning that the minimum number of test cases should be increased to five.

What should the test cases be?

Good values can be −11, 11 and 0, however depending on the data type.

An example of good test cases for this function:

Id	Description	Pass	Comments
7	Testing of CheckTypeStep() function	----	
7.1	Input val = 11, Output val = 1		
7.2	Input val = 10, Output val = 0		
7.3	Input val = 0, Output val = 0		
7.4	Input val = 10, Output val = 0		
7.5	Input val = −11, Output val = −1		
7.6	Input val = −23456, Output val = −1		

All three blocks have been tested, both limitations of the if tests have been tested, the special case with 0 has been tested, and a large number has been tested. The large number can be either positive or negative, or both. A good advice is to test at least one value "far out" related to the "normal data" range.

An example of a small test plan is shown in the appendix. This is a small overall system test with very few test cases. A test plan can be from a couple of pages, as shown in the appendix, up to several hundred of pages very depending of the complexity of the system. The time duration can be from a couple of days, as shown in the appendix, and up to several weeks.

A rule is to try to avoid that the testing is done by the developers. The developers should of course test when developing the system. When releasing a new software version or a new product or system, a test plan should document the testing and preferential be done by other people than the developers. Larger companies have their own test department and their own testers, however smaller companies may use the developers as testers as well. So in smaller companies a test plan is even more important than in larger companies with their own test department.

20.3 Test Levels

Testing of systems should always start with testing each sub system of the lowest level. Testing each sub device with a manageable size. When all sub devices are tested, sub systems will be interconnected new combinations must be tested. The number of test cases will increase when interconnecting or combining several sub devices.

20.3.1 Unit or Device Testing

In computer programming, unit testing is a method of testing that verifies the individual units of source code are working properly. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual program, function, procedure, etc., while in object-oriented programming, the smallest unit is a method, which may belong to a base/super class, abstract class or derived/child class.

Unit testing is also valid for devices, testing the hardware functions of each sub device. Unit testing is a type of white box testing as we need to know the working details inside the unit or device (box) to be able to test the box.

20.3.2 Module testing

Module testing (sometimes called Integration and Testing, I&T) is the phase of software testing in which individual software modules are combined and tested as a group. It follows unit testing and precedes system testing.

Module testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing. Module testing is based on integration and is a type of grey box testing as we need to have some information about both the working details inside and the I/O information of the modules.

20.3.3 System Testing

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic. System testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called assemblages) or between any of the assemblages and the hardware. System testing is a more limiting type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole. System testing is also called black box testing as the focus is interconnections and combinations, not the "functions" of each module.

20.3.4 Examples

Testing large and / or complex systems can be a challenge and it is important to test each individual part or module. Most often, it is the integration that is the challenge, when all possible combinations for each module are combined in a complete system. Therefore, it will be important to test each module properly, the integration of the sub modules, so that the system design does not become too complex.

This section will show some examples from major development projects where the programming system has created challenges in the final system test.

20.3.5 Control system for train traffic

Trains are an important part of the public communication for Oslo, the capital in Norway. Summer time in 2020 a new control system for handling the the train trafic was installed, but the system was not working satisfactory, the test results were not approved, and passengers had to use buses instead of trains, which caused major delays. Figure 20.1 shows the information in one of the largest newspaper i Norway saying that the passengers have to use the bus in stead of the train at least 10 more days. Meaning that the software system must be debugged and retested at least 10 more days, and this debugging period will not be any normal working days of 8 working hours.

The challenges in this case is the system integration at site as the communication between two sub modules is not stable enough at the site. It worked in the test location, but was not stable when tested in the real location. This shows how important a real system testing is with a test site simulating the real situation in the best way, otherwise your system may end up in the newspaper. Simulators, hardware in the loop (HIL) and/or digital twins can be useful testing tools when testing more complex system. These tools should contain a model of the environment for your system who will model the surroundings as realistic as possible to make the test as realistic as possible. This must include in many cases a noisy environment and input of unpredicitive events, and the testing tools should check the relationships between the inputs and outputs for the system. Development of a suitable test environment tools may also be an important part of development of such large and complex systems, and these test tools

16 | Nyheter

Aftenposten
Fredag 14. august 2020

Slik så det ut på Ski stasjon i går morgen. Store deler av stasjonen er splitter ny. Det som mangler, er tog og passasjerer.

Buss for tog i ti dager til - minst ...

Sveinung Berg Bentzrød og Tor G. Stenersen (foto)

Nye Ski stasjon står klar. Med ny overbygning, skinner, sviller og signalanlegg. Men togtrafikken som skulle startet 10. august, er utsatt i minst to uker.

Det går ikke så bra. Det er et nederlag at vi ikke kommer i mål som planlagt, sier utbyggingsdirektør Stine Undrum til Aftenposten.

Først opplyste Bane Nor at gjenåpningen av trafikken på Østfoldbanen etter sommerens buss for tog-ruteene utsettes fra mandag 10. til mandag 17. august. Onsdag kveld kom beskjeden om at gjenåpning utsettes til mandag 24. august.

Men Undrum kan ikke love at det blir åpning denne dagen. Fortsatt leter Bane Nor etter de feilene som fører til at nye Ski stasjon ikke kan tas i bruk.

Vi kan ikke garantere passasjerene et opplegg som er OK. Det synes vi er utrolig strevsomt.

Åge-Christoffer Lundeby, kommunikasjonssjef i Vy

Det betyr at det inntil videre er full stans i trafikken mellom Kolbotn i nord og Ås og Kråkstad i sør. Passasjerer fraktes delvis til Oslo med direktebusser fra Ski, Ås og Kråkstad, delvis tog mellom Kolbotn og Oslo og delvis med tog sør for Ås og Kråkstad.

Situasjonen rammer i utgangspunktet tusenvis av passasjerer, og til uken skal barn og ungdom tilbake til skolen.

Det er første gang siden 2009 at man ikke har vært klare til varslet dato for gjenåpning etter sommerstenging.

– Jeg har ikke vært med på at vi ikke har lykkes før, sier Undrum.

Vy: Vet ikke om de kan skaffe nok busser

– Vi har bedt om et møte med Bane Nor for å fortelle dem at dette gir store konsekvenser en særdeles vanskelig situasjon, sier Åge-Christoffer Lundeby, kommunikasjonssjef i Vy.

– Denne første uken har busselskapene vært ekstremt positive og stilt opp. Neste uke begynner skolen. Vi gjør det vi kan for å skaffe nok busser, men vet ikke nå hvordan det blir. Vi kan ikke garantere passasjerene et opplegg som er OK. Det synes vi er utrolig strevsomt, sier han.

– Også de unge kundeveilederne har vært velvillige. Men til uken begynner også mange av dem på skolen, sier Lundeby.

En annen usikkerhetsfaktor er hvordan det nå blir fremover med reiseråd. Om det strammes inn slik at det igjen bare blir tillatt med halvt belegg på busser og tog, kan dette vanskeliggjøre situasjonen ytterligere.

Stine Undrum i Bane Nor forklarer at det ikke er feil på utstyret som er problemet på Ski.

Sikkerheten i det tekniske anlegget for den nye stasjonen er formelt godkjent og funnet i orden. ATC-systemet, som binder to tog i å møtes på samme spor, virker også som det skal. Men stabiliteten i det elektroniske kommunikasjonssystemet mellom stasjonen og togførere som sitter i Oslo, har for dårlig kvalitet.

Problemet er bildene som togførerne får inn. De har forstyrrelser som gir uvitt, sier Undrum.

– Her handler det ikke om at noen kan gå ut på anlegget, finne en feil og skru litt for å rette den opp. Dette er feil som må rettes med programmering, sier hun.

Hun forklarer at et nytt fjernstyringsanlegg først testes i laboratorium.

– Her fungerte alt som det skulle. Det er når vi skal teste det i full skala, at det oppstår små og store «bugs». Det er det normalt at det gjør. Det pleier vi å kunne løse. Nå sitter vi med å rette alle feilene, sier Undrum.

Figure 20.1: A new control system for handling the train traffic control in the vicinity of the capital of Norway was delayed in the summer of 2020, and created major delays in public communication in the capital (in Norwegian, from the national Aftenposten newspaper 14–AUG-20).

Chapter 21

Summary

The elaboration phase and construction phases are finished and according to the UP the following operations, diagrams and documents are implemented. The elaboration phase has implemented the following operations, diagrams and documents:

- Analysis steps making the use case diagram and the domain model diagram,
- Design steps defining the tier architecture of the application, see section 8.1.1,
- Implement the tier architecture based on the domain model by making the root object, and the main objects for each of the tiers. Part of implementing the tier architecture in the application can be to make a prototype of the user interface as part of checking the collection of requirements,
- Make a overview test plan based on the use case diagram. A test plan must contain a system description and a set of test cases.

The construction phases has implemented the following operations, diagrams and documents:

- Analysis steps in selecting the most important use case, and make a fully dressed use case document (FDUCD) for this use case. Remember that this is the analysis phase and making the FDUCD should make you understand the process described with this use case. This means that the main success scenario of the FDUCD should explain in much details how the use case is going to perform or operate. This section must contain enough information that use case can be designed as an software application in the next step. The second iteration added 1) include and extension use cases to the use case diagram, 2) super and sub classes, 3) package diagrams, and extend the knowledge about the architecture analysis for the elaboration phase.
- Design steps means using design patterns to design the software as an object oriented application by using an interaction diagram. Select either a sequence diagram or a collaboration diagram as the interaction diagram. Based on the interaction diagram make or extend the class and object diagrams. The second construction diagram added knowledge about more design pattern, more UML diagrams as activity and state machine diagrams, and some knowledge about data structures and algorithms.
- Implementation steps by evaluating and implementing inheritance.
- Testing included more about test cases, different levels of testing, and merge all the testing into a complete test plan.

Part IV

Managing Software Projects

Chapter 22

Introduction

In software projects is software engineering important as the quality of the software developed will depend highly on the engineering principles. There must be a leader or a manager defining the software development process including the documentation that should be created according to this process. Today an iterative development process is often used, including use cases or user stories. It will also be important to include the customer as early as possible in the process for good feedbacks. The management must also define any coding standard, change tracking, and handling of source code changes and versions.

Based on the iterative development process and interactions with the customers several reports will be necessary in such a project being both informal and formal reports. The type and number of these reports should also be defined in the software development process.

Chapter 23

Software development

23.1 Introduction

A software application is made in several steps as:

- analyzing and designing,
- writing the code,
- compiling (converting the human understandable instructions to a computer understandable instructions),
- testing and/or debugging.

The compiler will produce an executable file[1] that can run on the computer. This executable file will be the software application and will be a combination of the compiled code from the developer and a set of libraries. These libraries are already compiled code for this type of microprocessors (platform), type of operating system, and type of services in the operating system or computer system. The merging of the compiled code from the developer and the libraries is called linking.

Static linking means that the libraries are merged into the application at compile time giving a complete application. As the application contains all the libraries, every application will require more space on disk.

Dynamic linking means that the libraries are "merged" with the application at run time, when the application is running only. One example is the Dynamic Link Library (DLL) on the Windows platform. The application will use operating system services to load the dynamic libraries at runtime for using the library code.

The differences of static and dynamic linking is shown in Figure 23.1, static linking to the left, and dynamic linking to the right. Static linking gives a static application and will always run (as tested), but will require more space on the disk. Dynamic linking will use the current libraries at that computer so a version check has to be performed of the library modules (done automatically by the operating system[2]). The developer has to decide if dynamic or static linking should be used in the project. The type of library used will be a configuration options in the IDE, default setting is normally dynamic linking.

23.2 Overview structure

Start with an overview structure of the software, what should be the main functions of the software? Why do you want to develop this software? Use Case diagram and deploy diagram can be a good starting point for defining the overview structure of the software. Then collect the requirements for getting more information about the main functions of the software. Important to get an overview of the input and output for the software, what kind of interfaces are needed for the software system. Also try to decide the type of modules, type of IDE and programming language, and ready made components.

[1] On Windows systems will the extensions be either *.exe* or *.com*.
[2] Known as the DLL hell on Windows systems.

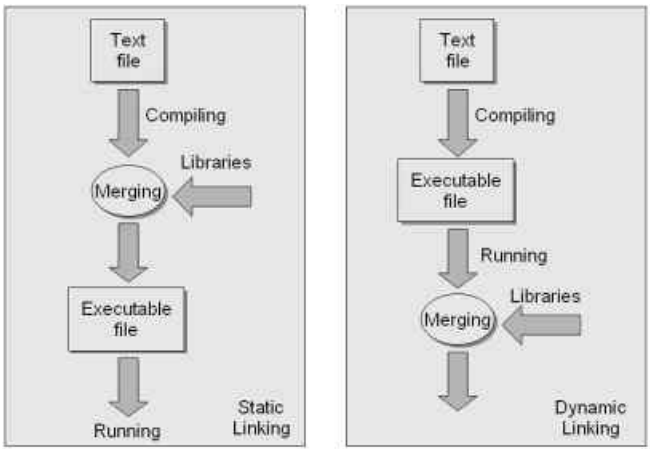


Figure 23.1: Compiling and linking a software application, static linking to the left, and dynamic linking to the right.

23.3 Layered design

Define the number of layers to separate any GUI layer and data layer, the focus should be the business layer. Use any patterns to make the application layered?

23.4 Component-Based Development

Component Based Development (CBD) is a software engineering discipline with focus on developing functional or logical components of the engineered systems with well-defined interfaces used for communication across the components. CBD is also known as Component-Based Development or Software Compontentry.

The focus in OOADP is to modelling real-world interactions and attempting to create objects for the "verbs" and "nouns" for the engineered systems The focus in CBD is to develop components of a higher level of abstraction than objects and as such they do not share state and communicate by exchanging messages carrying data. One of the main idea of both approaches is to make reusable modules that can be reused in the same project or in other projects. It takes significant effort and awareness to write a software component that is effectively reusable. Such a component needs(www.wikipedia.org 2006):

1. to be fully documented,
2. more thorough testing,
3. robust input validity checking,
4. to pass back useful error messages as appropriate,
5. to be built with an awareness that it will be put to unforeseen uses,
6. a mechanism for compensating developers who invest the (substantial) efforts implied above.

23.5 Configuration

Software applications will consists of several constants, should it be possible to adjust or change any of these constants? Will the software be used at different locations with different requirements? If configuration is wanted, the user interface must be extended with a configuration section, and the configuration data must be stored in a separate system like a database or a file. The configuration module must contain methods for reading configuration data from a database or a file at startup time, methods for interacting with the users or an administration user, and methods for writing the configuration data to a database or a file.

Configuration data is best to store in a setup file, in an ASCII format like XML, in the current directory. This way the file can be changed using a standard text editor without running the application.

Stored in the current directory means that the application can always find the configuration data. Any databases in use should have a directory link to be stored in an other directory, this link should be configured in the configuration data. This means that the application should have two different files or system for storing the data, one file for storing the setup data, and one file or database for storing the rest of the data, both configuration data, running data and historical data.

Configuration data is often about constants in the application, like

```
if (iSensorId >= 0 && iSensorId < MAX_SENSOR_ID)
{
```

In this test is MAX_SENSOR_ID a constant and can not be changed. An advance is to avoid such constants as they will always put some constrains to the application. A better solution will be to use configuration data, that means setup data that can be changed or set at startup time of the system, or an user with administrator rights. The test can then be:

```
if (iSensorId >= 0 && iSensorId < getConfigurationData(MaxSensorId))
{
```

where getConfigurationData() method is getting a parameter from the configuration section, and the parameter id is MaxSensorId. This is shown with a method from the same object, normally this should be a separate object (and class), according to the design patterns.

The same solution can also be done with different type of options. Like an option that should be turned on or off, do not hard coded the option by using constants. In stead define the option as a flag in the configuration section, and just test if the flag is true or false, like:

```
if (getConfigurationFlag(PrintingReportX) == true)
{
```

In this case can a specific report be turned on or off depending on the configuration. These options make the software much more flexible, and more easy to reuse, but it will also complicate the configuration section of the system. In larger systems the configuration module tend to be one of the biggest modules.

23.6 Testing and debugging

Software testing is the process of checking software, to verify that it satisfies its requirements and to detect errors regarding these requirements. Software testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate. This includes, but is not limited to, the process of executing a program or application with the intent of finding software bugs.

Testing can never completely establish the correctness of computer software. Instead, it furnishes a criticism or comparison that compares the state and behavior of the product against a specification. Software testing should be distinguished from the separate discipline of Software Quality Assurance (SQA), which encompasses all business process areas, not just testing.

Over its existence, computer software has continued to grow in complexity and size. Every software product has a target audience. For example, the audience for video game software is completely different from banking software. Therefore, when an organization develops or otherwise invests in a software product, it presumably must assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. Software testing is the process of attempting to make this assessment.

Testing can be done on the following levels:

1. Unit testing tests the minimal software component, or module. Each unit (basic component) of the software is tested to verify that the detailed design for the unit has been correctly implemented. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors. Testing each class,

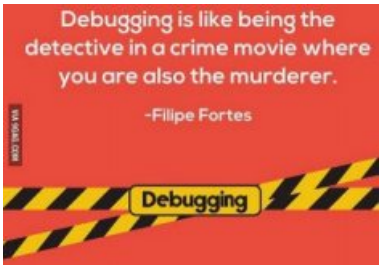


Figure 23.2: Debugging a system can be hard!

2. Integration testing exposes defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system. Testing each use case,

3. System testing tests a completely integrated system to verify that it meets its requirements. Testing the use case diagram with focus on use case combinations,

4. System integration testing verifies that a system is integrated to any external or third party systems defined in the system requirements. Testing the use case diagram with focus on actors.

Testing consists of using test cases. Test cases are dealing with an input, an action and an output. The action acts upon the input to derive a specific output. A test case must be specific by defining the inputs, action and any outputs, and the outcome should be OK or an error.

Any test cases that are not fulfilled must be fixed, and fixing involves debugging. Debugging is a multistep process that involves identifying a problem, isolating the source of the problem, and then either correcting the problem or determining a work around of the problem. The debugging steps often follow the testing step, and the final step of debugging is to test the correction or work around. Debugging for locating an error or deviation can be hard especially if you are debugging your own code, as shown in Figure 23.2.

23.7 Comments

Source code can be comments in many different ways. One way can be to add comments in the header of the class, function, or method. Another way will be to use meaningful names on the function or method name, layout, and identifiers. Often companies have style guidelines and naming conventions that everybody should follow. A rule of thumb: *If the code and the comments disagree, then both of them are probably wrong!*

23.8 Coding standard

Regardless of which software development model (iterative, waterfall, eXtreme programming, UP, and so on) is being used to create an application, coding standards are language-specific programming rules that greatly reduce the probability of introducing errors into the applications. These standards are defined by industry experts who analyzed how bugs were generated when code was written and correlated these bugs to specific coding practices. The correlations between bugs and coding practices was used to make a set of rules that prevent coding errors from occurring.

A coding standard ensure an uniform coding practices in organizations, reducing oversight errors and the time spent in code reviews. New programmers may also understand the code easier by the use of coding standards. Outsourcing the code to a third-party contractor, having a set of coding standards in place ensures that the code produced by the contractor meets all quality guidelines of the company. The coding standard will also define how the code should be documented. One example of such a coding standard is (McCall, Burden, Burnard, Hennell, Hills, Montgomery, Tapp & Whiting 2008).

See also Appendix D for another example of the contents of a coding standard.

23.9 Bug tracking system

A bug tracking system is a software application that is designed to help quality assurance and programmers keep track of reported software bugs in their work. Many bug-tracking systems, such as those used by most open source software projects, allow users to enter bug reports directly. Other systems are used only internally in a company or organization doing software development. Typically bug tracking systems are integrated with other software project management applications (www.wikipedia.org 2006). Some of these are:

1. Bugzilla; a "Defect Tracking System" or "Bug-Tracking System". Defect Tracking Systems allow individual or groups of developers to keep track of outstanding bugs in their product effectively. Most commercial defect-tracking software vendors charge enormous licensing fees. Despite being "free", Bugzilla has many features its expensive counterparts lack. Consequently, Bugzilla has quickly become a favorite of hundreds of organizations across the globe. (www.bugzilla.org; JUN-09).

2. Trac; Trac is an enhanced wiki and issue tracking system for software development projects. Trac uses a minimalistic approach to web-based software project management. Our mission is to help developers write great software while staying out of the way. Trac should impose as little as possible on a team's established development process and policies. (trac.edgewall.org; JUN-09).

23.10 Procedure based programming languages

Procedure based programming languages like Basic, Pascal, C and Matlab, to mention a few, breaks down the programming task into a collection of variables, data structures and sub routines. Object oriented programming languages break down the programming task into objects that expose the behavior and data. The behavior is based on methods, both public and private, while the data are properties or attributes and is normally private. In procedural programming procedures are used to operate on "global" data structures, while in object-oriented programming objects are used to operate on "private" data structures. Object is an instance of a class so each object operate on its own data structure only.

Can we use object-oriented methods with procedure based programming? YES! - and you should! You should always use a development process, and development processes like UP (Unified Process) can also be used together with procedure based programming languages. Note that Arduino software is also mainly based on procedural programming, as the programming language is plain C.

23.10.1 Analysis

Collect the requirements using FURPS+, and make a use case diagram. If this is a small software application, make one file for each of the use cases, and collect the procedure for each use case in these files. This way will the use case diagram be a documentation of the source code files in your system, and the type of procedures in each file. Normally will a software application require some analysis of each use case, and a Fully Dressed Use Case Document (FDUCD) should be used. A System Sequence Diagram (SSD) can also be used to structure the behavior of the use case even more.

23.10.2 Design

In procedural programming will the procedures be located in one or several source code files. By using an object-oriented approach will procedural programming use files in stead of classes. However, when designing, design the classes and use files in stead of objects when making the source code. If you need several objects of a class, this has to be solve by a data structure in file. This means that the software can be structured with the same type of procedures in each file, and the name of the files will indicate the type of functions in each file. Even better, the object-oriented documents and diagrams can still be used as documentation of the software application.

In UP will the first step be to use patterns for making an interaction diagram. The interaction diagram will document how the objects will cooperate and share responsibility to fulfill the requirements for each use case. In a procedural programming language will the classes (or objects) be the files, and the methods will be the procedures for each file. The interaction diagrams will document which procedures in which files are used, and how these procedures are calling each other. The class diagram

will document the number of files, filenames, and the main procedures in each file. Just like the object-oriented programming language should the size of the classes be manageable, and the name of class should clearly indicate the responsibility of each class. The same should also be for the procedural programming language as long as an object-oriented analysis and design approach is used.

23.10.3 Programming

Just like the object-oriented way of doing it, the name of the files are defined and the most important procedures are defined. This way will the software be analyzed, designed and structured as an object-oriented application, and documented.

23.10.4 Testing

Just like the object-oriented way of doing it, the requirements and the use case diagram can be used for making the test plan. The FDUCD can be used for making more specific test cases.

23.11 Reverse Engineering

Reverse engineering means to make documentation of the code. Reverse engineering is documenting the actual code, not how it is suppose to work. The code can be produced with very limited analysis and design documentation, or the code is extended without updating the documentation. Reverse engineering is the process of taking the existing code, making one or several documents, often interaction diagrams, based on the current version of the code, documenting the current version. An advice in reverse engineering is to focus on the objects and message flow between the objects (methods), not too much on the details (inside the methods).

Reverse engineering can be done automatically with UML tools, however these tools are often expensive. An other way is to do it manually by making interaction diagrams, but remember to follow the code strictly. Reverse engineering means to work as the cpu, find the start of the code, in C, C++, C# and Java this will be the main() method, and follow the sequence of the methods. An example can be the following code, a console application for merging and creating csv (comma separated values) files. The application consists of four classes; Program, Text, Database, and ExsportCSV. The code is:

```
//////////////////////////////////////////////////////////////
Most of the comments have been removed to save space.
//////////////////////////// Class ////////////////////////////
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace SamplingData
{
    class Program
    {
        static void Main(string[] args)
        {
            Text rText;
            Database rDatabase;
            bool runFlag = true;

            rDatabase = new Database();
            rText = new Text(rDatabase);
            rText.DisplayMenu();
            while (runFlag == true)
            {
                Thread.Sleep(500);
                runFlag = rText.CheckUsrInput();
```

```
            }
        }
    }

    class Text
    {
        ExportCSV rExportCSV;
        Database rDatabase;

        public Text(Database pDatabase)
        {
            rDatabase = pDatabase;
            rExportCSV = new ExportCSV(this);
        }
        public void DisplayMenu()
        {
            Console.WriteLine(" E: Export CSV file ");
            Console.WriteLine(" Q: Quit ");
            Console.Write(" Your choice?");
        }
        public bool CheckUsrInput()
        {
            bool RunFlag = true ;

            switch (Console.Read())
            {
                case 'Q':
                case 'q':
                    RunFlag = false;
                    break;
                case 'E':
                case 'e':
                    rExportCSV.MakeFile(rDatabase);
                    break;
                default :
                    break;
            }
            return RunFlag;
        }
        public string GetFileName()
        {
            return "Export.csv";
        }
        public void WriteStatusMessage(string msg)
        {
            Console.WriteLine(msg);
        }
    }

    class Database
    {
        public int GetSampleData(int NoOfSamples, double[] dData)
        {
            int Sample;

            for (Sample = 0; Sample < NoOfSamples; Sample++)
            {
                // Simulating retriving data from the SQL database
```

```
                dData[Sample] = (double)(Sample + 15.0F);
            }
            return NoOfSamples;
        }
    }

    class ExportCSV
    {
        const int MaxSamples = 128;
        double[] dData;
        Text rText;

        public ExportCSV(Text pText)
        {
            int Sample;

            rText = pText;
            dData = new double[MaxSamples];
            for (Sample = 0; Sample < MaxSamples; Sample++)
            {
                dData[Sample] = 0.0F;
            }
        }
        public void MakeFile(Database rDatabase)
        {
            int Samples = 25;
            string Filename;

            if ((Samples = rDatabase.GetSampleData(Samples, dData)) > 0)
            {
                Filename = rText.GetFileName();
                // Make the CSV file
            }
            else
            {
                rText.WriteStatusMessage("Error retrieving the samples from the database!");
            }
        }
    }
}
```

The steps for reverse engineering will be:

1. Find the starting point for the application. In a C, C++, Java or C# application this will be the main() method.
2. Start with the first method in the starting method making an interaction diagram.
3. Document all the steps in starting method, in the right sequence. Focus on the classes and the messages (methods), not on the details like variables. Include the main loop and conditions.

The interaction diagram for the presented console application is shown in Figure 23.3. Using this interaction diagram should give you a better overview of the application then looking at the code.

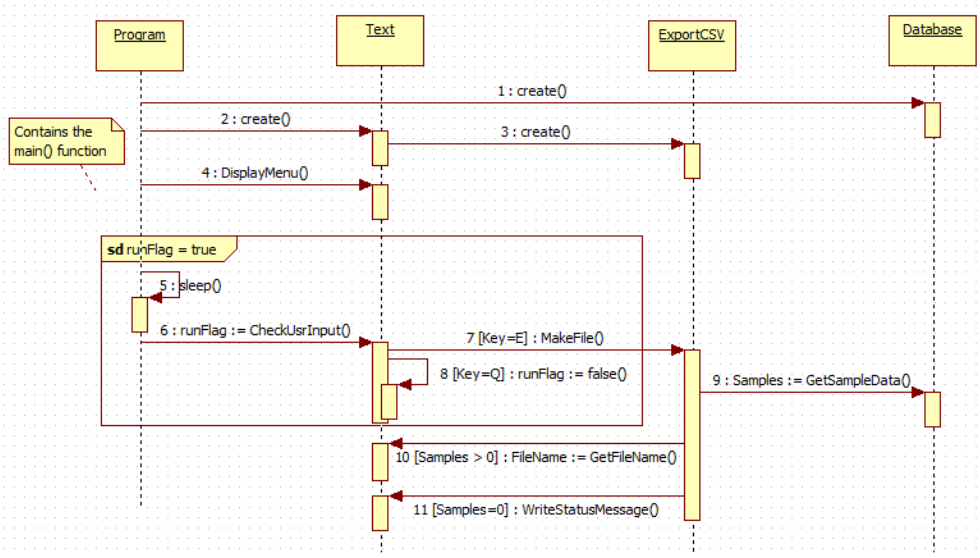


Figure 23.3: Reverse engineering of the console application with focus on the classes and messages between the classes.

Chapter 24

Guidelines for reports and projects

A report will be a document that communicates the work that has been carried out during some sort of experiments, group work, literature study, evaluations, and any combination. It is important to figure out who will be the reader of the report and the competence, experience and/or skills of these readers, and adjust the contents of the report to these readers.

24.1 Reports

24.1.1 Informal report

An informal report is mainly used for internal purposes, and should normally be as brief as possible. The main focus is to document an experiment, a meeting, or a discussion. This report should normally include an introduction, body, conclusion and recommendations. Always start the report with to whom the report is going, whom it is from, the date and the subject. After the conclusion include your contact information if needed, at least the company, email and phone number. The "Recommendations" section can be used to list any other people supporting the information of your report.

24.1.2 Formal report

A report shall communicate the work done and the results of the work. Remember that the grading or any decisions will normally only be based on the information in the report. Therefor focus on getting all the necessary information into the report. A formal report should contain:

1. Title page; title, author, and date,
2. Abstract (or summary); brief problem description, brief description of methods, and the main results,
3. Preface; what the report is about, any changes of tasks, and credits,
4. List of tables and figures (optional)
5. Table of Contents
6. Introduction; short about the background, previous work, and new work,
7. Problem description; process and equipment description, measurement setup, etc.,
8. Theory; model and/or method development,
9. Methods; Method development,
10. Results; simulation results, model fitting, optimization results,
11. Discussions; the results, as expected or what are causes, uncertainty, remaining work,
12. Conclusions; focus on the results and what you have learned,
13. Appendices; task description, details, listing, data sheets, etc,

14. References; necessary information for finding the references,
15. Index (optional).

24.1.3 References

Literature references can be done in several ways and it is up to writer to select the right style. The most used styles are the Harvard style and the Vancouver style. Do NOT mix these styles within the same document.

References to web pages should be avoided, but if you really need to use such a reference, you should include the author (if the name of the author is not available, you may use the name of the organization/institution) as well as the title of the web article, the URL and the date when you accessed the web page.

All Figures and Tables MUST also be referenced in the report text, including any sections in the Appendix. A Figure or a Table without a reference in the text do not need to be in the report.

Harvard style

In the Harvard style, the reference in the text body should be placed in parentheses after (or in some cases inside) the sentence. It should include the author's name and the year of publication (example (Flor, 2006)).

The references should be listed in detail in a separate list at the end of the report. The references should be listed alphabetically, using the authors last name.

Vancouver style

In the Vancouver style, the references in the text are given as numbers in brackets (example: [3]), and the references in the reference list should be listed using the numbers in brackets as well. When using this style, the references should be given according to the order of appearance in the body of the text.

24.1.4 Article or paper

An article or paper is used to document any research work and should contain the following sections:

1. Introduction; introduce the problem you will be discussing in your article or write a short story of your experience with the problem.
2. Methods; discuss the methods that you want to use on the problem or challenge you outlined in the introduction. Break up each point into separate paragraphs,
3. Results; discuss the results of any experiments or simulations used with the methods. Break up each point into separate paragraphs,
4. Discussion; this should include a brief summary of the article.

A mnemonic rule can be IMRAD (Introduction, Methods, Results, And Discussions).

24.2 Presentation

Documenting a project can often be combined with a presentation. Some advices keeping a presentation (Kolda & Torczon 2011):

- Make eye contact with the audience, do not look to much at the screen,
- Try to use some colors on your presentations, but not too much. Focus on the contents!
- Do not overcrowd the slides,
- Avoid acronyms and math symbols,
- Make good graphics (and animations),

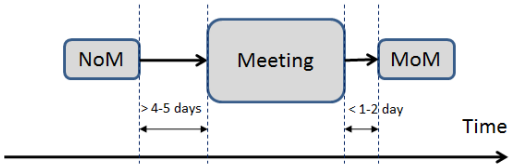


Figure 24.1: Suggestions for minimum timing of "Notice of Meeting" (NoM) and "Minutes of Meeting" for a formal meeting.

- Put your work into context, but limit the time used on background material,
- Use time preparing your slides,
- Limit your time!

24.3 Meetings

24.3.1 Non formal meetings

Small meetings, often short meetings once a day or week. Use these meetings for discussions about challenges in the project, the work done and/or the plans before the next meeting.

24.3.2 Formal meetings

Formal meetings include a "Notice of meeting" (NoM) and a "Minutes of meeting" (MoM) and these meetings are often used with milestones in the project. The milestones will be part of the planning and should be part of the schedule. Figure 24.1 indicates the timing of these documents with reference to the meeting. Note that the NoM timing is the absolute minimum, at least a couple of weeks is optimum in the industry. The timing for the MoM is maximum, try to make the MoM as fast as possible after the meeting while everybody is remembering the cases.

"*Notice of meeting*" should contain:

1. Name of the project; which project is the meeting for,
2. Place; the location, often a building and room number,
3. Date and time,
4. The name of the members of the meeting,
5. The agenda.

Remember to inform/schedule the meeting early,
"*Minutes of meeting*" should contain:

1. Name of the project,
2. The name of the members of the meeting (indicate who was present and who was not present),
3. The date and time,
4. A table of the subject discussed on the meeting. Very often will this table contains 3 columns, the ID, the subject, and the responsible. An example of such a table can be:

ID	**Subject**	**Responsible**
1	Subject #1 discussed ..	
2	Subject #2 discussed ..	Name and date
..	..	
N	Subject #N discussed ..	Name and date

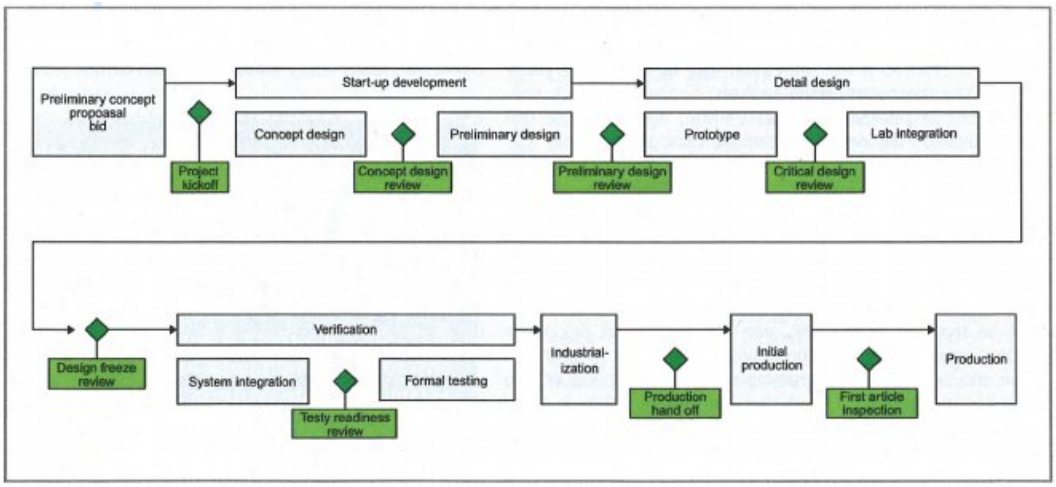


Figure 24.2: Product development steps with the main milestones.

24.3.3 Effective meetings

Some tips for both the attendees and the chair of the meeting.

1. Determine whether the meeting is necessary, determine the number of people and the location,
2. Be precise, do not let the late comers control the start of the meeting! Do NOT recapping the meetings for latecomers!
3. Be prepared, the background material should be received and reviewed prior to the meeting!
4. Have an objective, "what should be the result of the meeting",
5. Have an agenda and distribute the agenda prior to the meeting!
6. Address the most important items first. If you run out of time, the most important items have been addressed.
7. Be clear about the responsibilities,
8. Document the meeting, include deadlines and responsibilities.

24.4 Projects

Figure 24.2 indicates the fundamental stages in product design. It does not matter whether there is a big project or a small project, a large company or a small company, the stapes are the same. The milestones are marked as green blocks and each milestone must be measurably.

Even if the steps are the same will the extent of each step vary according to the type of product, system requirements and size of the project team. A competent engineer must conceptualize the project, make a proper development plan, and then try to follow this plan. Any deviations, update the plan to reflect the real situations. An important document in project management is the schedule, made as a Gantt diagram. An example of a Gantt diagram is shown in Figure 24.3 with the tasks, subtasks and milestones. This diagram also shows the timing of each task (horizontal), any dependencies between the tasks and resources used in each task. Based on this diagram the fastest delivery of the product is possible and the status of the project at any time as long as the project members update the status of each subtasks.

The tasks is based on Work Breakdown Structure (WBS) of the project. Start a project planning by defining a set of WBS. A WBS defines a set of tasks that can be completed independently of other WBSs (tasks). Each WBS must be braked down into manageable tasks. If a task is not manageable, break down the task into sub tasks. If a sub task is not manageable, break down the sub task into sub sub tasks, and so one. A task is manageable when it is possible to estimate the amount of work to do, and the time for doing this work. Use the WBS to facilitating resource allocation and assignment of

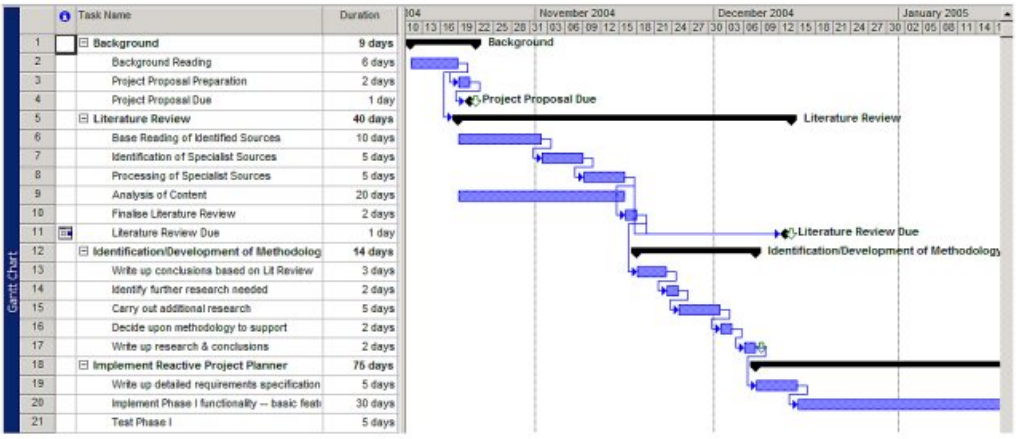


Figure 24.3: An example of a product development schedule (https://www.sitepoint.com/project-management-web-geeks: oct-16).

responsibilities for the tasks. The WBS will also be used for measurement and control of the project as the schedule indicates the start of each task and the responsibility of each task. Several software tools can be used to make the Gantt diagram like Microsoft Excel and Microsoft Project. Microsoft Project is a tool used a lot in the industry.

The development process consists of a set of milestones, shown as green diamonds in Figure 24.2 and black diamonds in Figure 24.3. The milestones are assessments of the project status and are important for formal reviews of the project and making decisions for further progress of the project. Reviews are to the engineer's advantage as the customer can confirm the project's compliance according to the requirements. These reviews prevents the customer from demanding (unpaid) changes or a redesign at a later time. The number and type of milestones may depend on the project, the milestones indicated in Figure 24.2 are:

- Project kickoff: just a startup of the project,
- Concept design review: defining the overall structure and architecture of the system,
- Preliminary design review (PDR): define the system architecture, partitioning of functions between hardware and software using block diagrams and pseudo coding, preliminary reliability analysis (if needed). An approval from the PDR indicates that the design of the system can start,
- Critical design review (CDR): the culmination of the detail design phase of the project, both hardware and software modules with bill of material (BOM) of hardware modules, and tested and documented software modules,
- Design freeze review: testing of the prototypes and normally this will consists of loops before the design is freezed,
- Testly readiness review, production hand off and first article inspection are milestones for system testing and production of the system or product.

Important guidelines for planning a project include:

- Plan the milestones, not the tasks: Focus on the milestones. A milestone is either OK or not, while it is very hard to measure how complete a task is.
- Include milestones into your WBS to manage the "deliverables" of the tasks.
- Each milestone should be assigned to only one person (resource): Clear responsibility makes it easier to get the work done and the project team will trust the results of the milestones as well,
- Let all members of the project be responsible for the planning of the tasks behind each milestone. If a person has been planning a task, he may also have the responsibility and ownership of the task to fulfill it within the schedule.

Part V

Appendixes

Appendix A

Avoiding software development nightmares?

This is a copy of an article printed in the magazine "Control Engineering Europe" in April 2016, by Tim Follows. The magazine is available for free, both on paper and on web, the web header is shown in Figure A.1.

Figure A.1: The web page for the magazine "Control Engineering Europe" - www.controlengeurope.com: Nov-19.

Tim Fellows, a certified LabVIEW developer and system integrator at LiveWires Automation, discusses how to avoid problems during software development in this article:

In the last few years I have come across the following scenario in industry many times: A complex piece of software is being written by a team of developers. ***The initial deadline for completion has passed and there is pressure to complete the work.*** *The developers are working all hours, but no-one really knows how long it will take. Everyone's stress level is high and the health of some staff members is being affected. Management are trying to work out how to limit the damage – whether to recruit more programmers, let things continue as they are or maybe even pull the plug on the whole project.*

Unfortunately at this stage there are rarely any good options. *Throwing more developers into the mix can actually slow the project down because they will need help 'getting up to speed'. Adding developers also creates a need for more communication owing to the fact that, sooner or later, their work will have to mesh together. However, one instance where adding more developers can work is where the new developers bring specific skills that the existing ones lack.*

I believe that the above scenario can be avoided. To understand how, it is necessary to think about some of the characteristics of software development work.

Estimation of timescales

Each new software project is unique. It will most likely not be possible to compare its likely timescale with that of another project. In addition, the complexity of software adds to the difficulty of estimating reasonable timescales. Caution should be exercised in estimating timescales and the best estimate of timescales is likely to come from a developer who has previously been involved in many time-critical projects, preferably projects that have similarities with your new project.

Up front design

Studies have shown that time spent designing software at the start of a project is time well spent. *Making design changes once work is under way is much less efficient and becomes*

progressively worse the later the changes are made. Unexpected changes also make the program 'messy' and harder to work with, and this is when the phenomenon of 'software decay' begins to set in. In some cases, insufficient up-front design can mean that the program is given the wrong overall structure at the start and this can become a major problem.

Great care should be taken in the design process. Underestimating the complexity of what is required is a mistake, as is failing to spot elegant solutions where things can be made simpler. The design process also needs to involve all 'interested parties.' For example, there is nothing to be gained from the developers starting to design a program with insufficient input from those who know what the market requires and wants.

Sometimes there can be pressure to start the development process before sufficient design work has been completed, so the end user can see that progress is being made. This pressure should be resisted, where possible, by stressing the importance of 'proper design'.

For anyone who has some computer literacy, programming is an easy field to get into. Even after a week or two of study to learn some of the basics, the demand in the market is such that an agency may be keen to sign up a fledgling programmer and promote him or her to various organisations. In reality, though, programming is an unforgiving and often complex task. **A highly analytical mind is required, with a flair for solving complex logic problems.**

There are a multitude of different qualifications across the software world with no common standard. Developers may question the value of such qualifications and decide not to bother with them. On the positive side, subjects studied in further education, such as maths and science subjects (including computer science and engineering) may give a rough gauge of a programmer's ability to handle the complex logic of programming.

A number of different methods of finding programmers are worth trying, including Google searches, LinkedIn, agencies and personal recommendation. If you have any doubt about a given candidate, it would be worth arranging a short initial trial period. Up front design is essential in software development, as is the appointment of good quality developers. The most effective developers will have the experience to estimate timescales properly, have a good grasp of their language, be good at solving complex logical problems and should also have good interpersonal skills.

Appendix B

GiT Command Overview

An overview of the most used GiT commands.

GitHub
GIT CHEAT SHEET

Git is the open source distributed version control system that facilitates GitHub activities on your laptop or desktop. This cheat sheet summarizes commonly used Git command line instructions for quick reference.

INSTALL GIT

GitHub provides desktop clients that include a graphical user interface for the most common repository actions and an automatically updating command line edition of Git for advanced scenarios.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

Git distributions for Linux and POSIX systems are available on the official Git SCM web site.

Git for All Platforms
http://git-scm.com

CONFIGURE TOOLING

Configure user information for all local repositories

`$ git config --global user.name "[name]"`

Sets the name you want attached to your commit transactions

`$ git config --global user.email "[email address]"`

Sets the email you want attached to your commit transactions

`$ git config --global color.ui auto`

Enables helpful colorization of command line output

CREATE REPOSITORIES

Start a new repository or obtain one from an existing URL

`$ git init [project-name]`

Creates a new local repository with the specified name

`$ git clone [url]`

Downloads a project and its entire version history

MAKE CHANGES

Review edits and craft a commit transaction

`$ git status`

Lists all new or modified files to be committed

`$ git diff`

Shows file differences not yet staged

`$ git add [file]`

Snapshots the file in preparation for versioning

`$ git diff --staged`

Shows file differences between staging and the last file version

`$ git reset [file]`

Unstages the file, but preserve its contents

`$ git commit -m "[descriptive message]"`

Records file snapshots permanently in version history

GROUP CHANGES

Name a series of commits and combine completed efforts

`$ git branch`

Lists all local branches in the current repository

`$ git branch [branch-name]`

Creates a new branch

`$ git checkout [branch-name]`

Switches to the specified branch and updates the working directory

`$ git merge [branch]`

Combines the specified branch's history into the current branch

`$ git branch -d [branch-name]`

Deletes the specified branch

GIT CHEAT SHEET

REFACTOR FILENAMES
Relocate and remove versioned files

```
$ git rm [file]
```

Deletes the file from the working directory and stages the deletion

```
$ git rm --cached [file]
```

Removes the file from version control but preserves the file locally

```
$ git mv [file-original] [file-renamed]
```

Changes the file name and prepares it for commit

SUPPRESS TRACKING
Exclude temporary files and paths

```
*.log
build/
temp-*
```

A text file named `.gitignore` suppresses accidental versioning of files and paths matching the specified patterns

```
$ git ls-files --other --ignored --exclude-standard
```

Lists all ignored files in this project

SAVE FRAGMENTS
Shelve and restore incomplete changes

```
$ git stash
```

Temporarily stores all modified tracked files

```
$ git stash pop
```

Restores the most recently stashed files

```
$ git stash list
```

Lists all stashed changesets

```
$ git stash drop
```

Discards the most recently stashed changeset

REVIEW HISTORY
Browse and inspect the evolution of project files

```
$ git log
```

Lists version history for the current branch

```
$ git log --follow [file]
```

Lists version history for a file, including renames

```
$ git diff [first-branch]...[second-branch]
```

Shows content differences between two branches

```
$ git show [commit]
```

Outputs metadata and content changes of the specified commit

REDO COMMITS
Erase mistakes and craft replacement history

```
$ git reset [commit]
```

Undoes all commits after `[commit]`, preserving changes locally

```
$ git reset --hard [commit]
```

Discards all history and changes back to the specified commit

SYNCHRONIZE CHANGES
Register a repository bookmark and exchange version history

```
$ git fetch [bookmark]
```

Downloads all history from the repository bookmark

```
$ git merge [bookmark]/[branch]
```

Combines bookmark's branch into current local branch

```
$ git push [alias] [branch]
```

Uploads all local branch commits to GitHub

```
$ git pull
```

Downloads bookmark history and incorporates changes

GitHub Training

Learn more about using GitHub and Git. Email the Training Team or visit our web site for learning event schedules and private class availability.

training@github.com
training.github.com

Appendix C

C# Introduction presentation

A presentation given as an introduction to C# and the Visual Studio IDE tools, based on the .NET platform. The presentation will cover the basics of the C# programming language, the usage of classes and objects, and how to create GUI and command line based applications in Visual Studio IDE.

Høgskolen i Telemark

C#
(C sharp)

Programming Language

Nils-Olav Skeie
C# introduction for the course SCE1306 at Telemark University College (TUC)

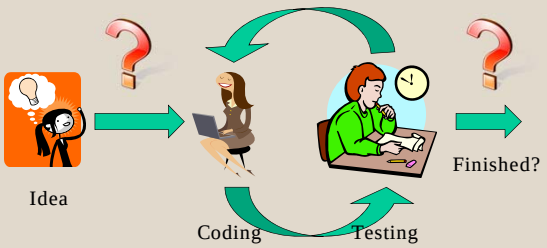
Høgskolen i Telemark

Agenda: The C# Language

- Introduction,
- Basics,
- Arrays, structures and strings,
- Program Flow, Loops and Decisions,
- Objects: Methods and classes,
- Objects: Creating,
- Threading,
- Graphics,
- Debugging.

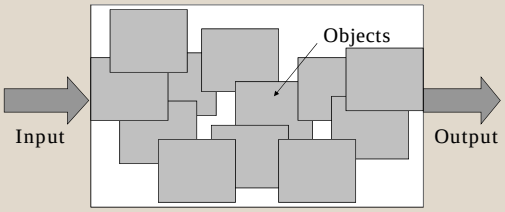
Høgskolen i Telemark

Introduction: Software development



Høgskolen i Telemark

Introduction: Object Oriented



Software application

Object: Combination of data and methods (private and public)

4

Høgskolen i Telemark

Introduction: Programming Structure

- Divide the problem(s) in objects,
- Divide the objects in methods and data,
- Use one module (file) for each object,
  - Use the same name of the module (file) and the object.
- Let each object have responsibility for a specific sub task,
- The objects must cooperate to solve the whole task.
- Program:
  - sum_value = sum(value#1, value#2),
    - Sum_value is a variable, the value from the sum function,
    - Sum is a function, defined in an object,
    - Value#1 and value#2 are parameters used by the sum function,
- C# (C sharp)
  - Based on C, C++ and Java.

5

Høgskolen i Telemark

Basics (1)

- Comments:
  - Block: /* and */
  - Rest of line: //
  - Y=sum(x1,x2) ; /* Sum of x1 and x2 */
  - Y=sum(x1,x2) ; // Sum of x1 and x2
- Naming convention:
  - The development process should define the naming of variables and constants,
  - Lowercase, uppercase and underscore
- Declarations:
  - int iNumber = 5 ;
  - string sTxt = "SCE introduction" ;
  - sum = x1 + x2 ;
  - x = sum(x1, x2) ;
- Simple operations:

sum = x + 1 ;	sub = x – 1 ;	mult = x * 2 ;	div = x / 2 ;	remainder = x % 3 ;
x = x + 1 ;	x = x – 1 ;	x = x * 2 ;	x = x / 2 ;	x = x % 3 ;
x++ ;	x-- ;	x *= 2 ;	x /= 2 ;	x %= 3 ;

6

Høgskolen i Telemark

Basics (2)

- Scope: private or public;
  - Define the access levels on variables, methods, structures, and classes.
- Block of code;
  - Section of the code is defined inside curly brackets: {}
- End of statements;
  - Every statement ends with a semicolon: ;
- **Duplication of code;**
  - Do NOT !! duplicate code (reuse of code)

7

Høgskolen i Telemark

Basics: Data types

Data type	Type	Description	Example
bool	boolean	TRUE or FALSE	bool flag = false ;
byte	Byte	0 to 255	byte bit_flag = 0x7f ; or byte bit_flag = 128 ;
sbyte	Signed byte	-128 to +127	sbyte sval = -128 ;
short	Short	-32768 to +32767	short sval = 28500 ;
ushort	Unsigned short	0 to 65535	ushort sval = 48900 ;
int	Integer	-2.147.483.648 to +2.147.483.647	int ival = 98600 ; int ival = -115200 ;
uint	Unsigned integer	0 to 4.294.967.295	uint ival = 98600 ;
long	Long	Large integer	long lval = 3600l ;
ulong	Unsigned long	Unsigned large int.	ulong lval = 3600l ;
float	Float	$\approx -3.4*10^{38}$ to $3.4*10^{38}$	float fval = 1.23f ;
double	Double	$\approx -1.8*10^{208}$ to $1.8*10^{208}$	double dval = 1.23d ;
decimal	Decimal	Used for currency (10^{28})	decimal money = 1.23m ;
char	Character	Letters of the alphabet	char cval = 's' ;
string	string	"Any combination"	string stxt = "Course SCE1306"

8

Høgskolen i Telemark

Basics: Declarations

- Define the variables in a function or class;
  - Assign default values if necessary,
  - Example:
    - int inumber;
    - int ival = 12;
    - const int MAX_LIMIT = 48;
    - float fnumber;
    - float fval = 12.6F;
    - char cval1, cval2, cval3;
    - char cval = 'N';
    - string sval = "SCE1306 demo application ";
    - byte bval = 0x04;
    - bool on_off_switch = false;
- Define values from functions;
  - fval = sum(fval1, fval) ;

9

Høgskolen i Telemark

Arrays, structures, and strings (1)

- Arrays
  - Using square brackets: []
    - int[] ibuffer ; // defining an integer array
    - ibuffer = **new** int[64] ; // create an array of 64 items, from 0 to 63!
      - or
    - int[] ibuffer = {1,2,3,4,5,6,7,8,9,0} ; // 10 items, from 0 to 9
    - string[] sbuffer = {"SCE1106", "SCE1206", "SCE1306"} ;
  - Access: use the index
    - ibuffer[ival] = 12 ; // ival must be between 0 and max length - 1
    - scourse = sbuffer[2] ;

10

Høgskolen i Telemark

Arrays, structures, and strings (2)

- Multidimensional arrays;
  - Using square brackets with comma: [,,]
    - int[,] ibuffer ; // defining an integer array
    - ibuffer = **new** int[64,4] ; // create an array of 4x64 items,
      - or
    - int[,] ibuffer = {{1,2,3},{4,5,6},{7,8,9}} ; // 3x3 items
    - string[,] sbuffer = {{"SCE1106", "Control"}, {"SCE1206", "Sensor"}, {"SCE1306", "OOADP"}} ;

11

Høgskolen i Telemark

Arrays, structures, and strings (3)

- Structures;
  - combining data of different types,
  - Using the key word struct,
  - Can make arrays of structs,
  - Example of a struct:

```
struct student_data
{
        public string lastname ;
        public string firstname ;
        public int[] courses ;
        public bool gender ;
} ;

const int MAX_STUDENTS = 32 ;
student_data[] student = new student_data[MAX_STUDENTS] ;

student[4].lastname = "Skeie";
```

12

Høgskolen i Telemark

Arrays, structures, and strings (4)

- String: a collection of characters;
  - string stxt = ”SCE1306: OOADP” ;
  - char ctxt = stxt[4] ;
  - char[] ctxt = new char[64] ;
- String: manipulation;
  - string stxt1 = ”SCE1306” ;
  - string stxt2 = ”OOADP” ;
  - string stxt = stxt1 + ”: ” + stxt2 ;
  - Length: int len = stxt.length;
- String: Conversion
  - float fval = 2.456 ;
  - Convert to string: string stxt = ”Value=” + fval.ToString()) ;

13

Høgskolen i Telemark

Arrays, structures, and strings (5)

- String: Methods;
  - stxt.Substring(x,y) ; // Display only the characters between x and y,
  - stxt.IndexOf(seachString) ; // If stxt contains the searchString,
  - stxt.Trim(), stxt.TrimEnd(), stxt.TrimStart() ; // Remove spaces
  - stx.Replace(stxt1, stxt2) ; // Replace stxt1 with stxt2
  - stxt.Split()
    - char[] splitChars = {’ ’, ’,’, ’;’} ; // Space, comma, and semicolon
    - string stxt = ”SCE1306 OOADP, F29” ;
    - String[] words = stxt.Split(splitChars) ;
  - stxt.ToUpper() ; // Uppercase letters
  - stxt.ToLower() ; // Lowercase letters

14

Høgskolen i Telemark

Program structure (1)

- Divide the program into manageable methods;
  - A method is a subroutine or subprogram performing a specific task independent of the remaining code,
  - A method is also called a procedure, function, or routine,
- The methods will cooperate to solve the tasks for the application,
  - An application will have input data and output data,
  - The methods will generate the output data depending on:
    - Algorithms (one or several),
    - the input data,
    - Previous data (database),
    - the time (real-time system).
- Several levels of methods,
  - A method can call another method calling another method calling ….

15

Høgskolen i Telemark

Program structure (2)

- Reuse;
  - Make the methods as general as possible,
  - Try to reuse methods in the application,
  - Reuse already tested methods,
- Data;
  - Try to avoid using global data, let data be private to methods
- Method

```
Return_type method_name(parameters)
{
    method code ;
    return parameter ;
}
```

16

Høgskolen i Telemark

Program flow, loops, and decisions

- Each method must contain some program flow containing:
  - loops;
    - For
    - Foreach
    - While
    - Do while
  - Decisions;
    - If
    - If else
    - If else if else
    - Switch case

17

Høgskolen i Telemark

Loops: for loop

Used when you know the start and finish points of the loop:

```
for (variable=start expression; end condition; increment expression)
{
    statements ;
}
```

Notes:
1: the loop variable has to be an integer,
2: increment can also be negative (decrement),
3: end condition is only check at each start of the loop.

Example:

```
int  count, sum = 0 ;

for (count = 1; count < 4; count++)
{
    sum = sum + count ;
}
```

18

Høgskolen i Telemark

Loops: foreach

Used for accessing elements in an array or collection of objects

```
foreach (variable in list)
{
    statements ;
}
```

Notes:
1: foreach know the length of the array or collection,
2: foreach will check the number of iterations.

Example:

```
int[] student_list = {321, 435, 432, 213, 324, 354, 621, 534, 523} ;
char grade ;
String name ;

foreach (int student_id in student_list)
{
      name = get_name(student_id) ;
       grade = get_grade(student_id) ;
       // Display the names and the grades
}
```

19

Høgskolen i Telemark

Loop: while

Use a while loop when the loop variable or the end condition is changed within the loop.

```
while (expression)
{
    statements ;
}
```

Notes:
1: The expression is true/false test,
2: The expression is only check at the start of each loop.

Example:

```
Int count = 1, sum = 0 ;

while (sum < 100)
{
    sum += count ;
    count++ ;
}
```

20

Høgskolen i Telemark

Loop: do while

Use a do while loop when the loop variable or the end condition is changed within the loop.

```
do
{
    statements ;
} while (expression) ;
```

Notes:
1: The expression is true/false test,
2: The expression is only check at the end of each loop.

Example:

```
Int count = 1, sum = 0 ;

do
{
    sum += count ;
    count++ ;
} while (sum < 100) ;
```

21

Høgskolen i Telemark

Decision: if, if else, if else if else

```
if (condition)
{
    statements ;
}

if (condition)
{
    statements1 ;
}
else
{
    statements2 ;
}
```

```
if (condition)
{
    statements3 ;
}
else if (condition)
{
    statements 4 ;
}
else if (condition)
{
    statements5 ;
}
else
{
    statements6 ;
}
```

22

Høgskolen i Telemark

Desicion: condition

- The condition is a logical comparison giving true or false
  - if statements if true
  - else statements if false
- The logical operators:
- Can be combined with &&
  - if ((x>2) && (x<10))

Operator	Type	Example
= =	Is equal to	(x = = 2)
! =	Is not equal to	(x ! = 2)
<	Is less than	(x < 2)
<=	Is less or equal to	(x < = 2)
>	Is greater than	(x > 2)
>	Is greater or equal to	(x > = 2)

23

Høgskolen i Telemark

Decision: switch case

Can be used in stead of multiple if or if – else statements

```
Switch (expression)
{
    case statement1:
        break ;
    case statement2:
        break ;
    ........
    default :
        break ;
}
```

Notes:
1: The expression must be an integer or a string,

```
switch (state)
{
    case 1:  // Open state
        statements1 ;
        state = 2 ;
        break ;
    case 2 :  // Read state
        statements2 ;
        state = 3 ;
        break ;
    case 3 :   // Write state
        statements3 ;
        state = 2 ;
        break ;
    default :
        // error or warning
        state = 2 ;
        break ;
}
```

24

Høgskolen i Telemark

Decision: try and catch

- Used for handling any exceptions in your code,
- Must be used for any part of the code that can cause an error, like:
  - Open a file,
  - Dividing by zero,
  - Using communication,
  - …
- DivideByZero, Format, IndexOutOfRange, FileNotFound, ….

```
int x, x1 = 6, x2 = 4 ;
try
{
   x2 -= 4 ;
   x = x1 / x2 ;
}
catch (DivideByZeroException)
{
   // Display an error or warning
   x = 1 ;
}
```

Windows Message: "The application has performed an illegal operation and will be shut down"

25

Høgskolen i Telemark

Objects: Methods and classes (1)

- An object is a collection of methods and data to perform a specific subtask,
- A class is an abstract definition of the object;
  - Classes are used in the model,
  - Objects are used in the application.
- A method consists of a sequence of programming statements to perform an action, alone or in cooperation with other methods.
  - Class method in the class,
  - Instance method in the object.
- The methods are used for:
  - Decisions,
  - Communication inside the class,
  - Communication with other classes.
- Use methods to get and/or set the data, let the data be encapsulated !
  - Data is normally private.
- Define methods as private or public.

26

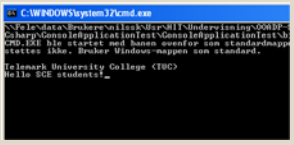
Høgskolen i Telemark

Objects: Namespaces

- A namespace is an collection of objects and their methods,
- .NET consists of many namespaces,
- Use the keyword *using* to include other namespaces,
- Use the same namespace for all source code files in your project,
- Example:

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

namespace ConsoleApplicationTest
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine(" ");
         Console.WriteLine("Telemark University College (TUC)");
         Console.Write("Hello SCE students!");
         System.Console.ReadKey();
      }
   }
}
```



27

Høgskolen i Telemark

Objects: Methods and classes (2)

- Parameter passing;
  - Passing data to and from the methods,
  - Four options:
    1. Pass nothing to the method, return nothing from the method (void),
    2. Pass something to the method, return nothing,
    3. Pass nothing, return something,
    4. Pass something, return something.
  - input parameters are only copies, the original parameter is not changed!
  - Can ONLY return one parameter.

Example:

```
public int sum(int ix1, int ix2, int ix3)
{
    ix1 += ix2 ;
    ix1 += ix3 ;
    return ix1 ;
}
```

```
public bool read_switch(void)
{
    bool   bval ;
    bval = ioread_switch() ;
    return bval ;
}
```

28

Høgskolen i Telemark

Objects: Methods and classes (3)

- Passing a reference to a parameter (address or pointer),
  - Using the *ref* and the *out* keywords,
  - ref: passing a reference to the parameter, not a copy,
  - out: return several parameters

Example:

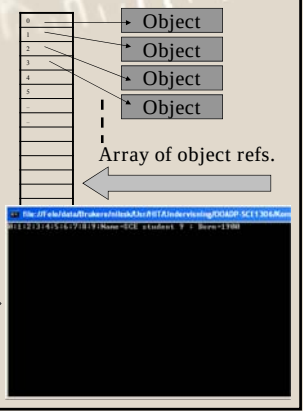
```
public float sum_and_average(float fx1, float fx2, ref float fx3, out float average)
{
    fx1 += fx2 ;
    fx1 += fx3 ;            // fx1 is sum of all parameters
    fx3 += fx2 ;            // fx3 is sum of fx2 and fx3
    average = fx1 / 3.0F ;
    return fx1 ;
}
```

29

Høgskolen i Telemark

Objects: Creating

```
using System;
using System.Text;
namespace ConsoleApplicationTest
{
  class student                    // Class #1
  {
    public string name;
    public int born_year;
  }
  class Program                    // Class #2
  {
    static void Main(string[] args)     // ALWAYS starts from main
    {
      student[] stud_id;
      int count;
      const int MAX_STUDENT = 10 ;
      string stxt;
      stud_id = new student[MAX_STUDENT];   // Make an array
      for (count = 0; count < MAX_STUDENT; count++)
      {
        stxt = "SCE student " + count;
        stud_id[count] = new student() ;          // Make each class
        stud_id[count].name = stxt;
        stud_id[count].born_year = 1900 ;
        System.Console.Write(count + ".");
      }
      count--;
      System.Console.Write("Name=" + stud_id[count].name);
      System.Console.Write(" : Born=" + stud_id[count].born_year);
      System.Console.ReadKey();
    }
  }
}
```



Høgskolen i Telemark

Objects: Constructor and overloading

Constructor:

- A method that is called only once, when the object is created,
- The method has the same name as the object,
- Can be used to initialize the object,

Overloading:

- Methods with the same name, but different type of parameters,
- Do NOT duplicate the code, let one of the methods be the base method.
  - Base method: the method with the decisions and/or algorithms.

31

Høgskolen i Telemark

Objects: An application

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

namespace ConsoleApplicationTest
{
  class student
  {
    // Private data
    string name;
    int born_year;
    // Public methods
    public student(string sname, int year)
    {
      name = sname;
      born_year = year;
    }
    public string get_name()
    {
      return name ;
    }
    public string get_born()
    {
      return born_year.ToString();
    }
  }

class Program
  {
    static void Main(string[] args)
    {
      student[] stud_id;
      int count;
      const int MAX_STUDENT = 10 ;
      string stxt;

      stud_id = new student[MAX_STUDENT];
      for (count = 0; count < MAX_STUDENT; count++)
      {
        stxt = "SCE student " + count;
        stud_id[count] = new student(stxt, 1980 + count) ;
        System.Console.Write(count + ":");
      }
      count--;
      System.Console.Write("Name=" + stud_id[count].get_name());
      System.Console.Write(" : Born=" + stud_id[count].get_born());
      System.Console.ReadKey();
    }
  }
}
```

Høgskolen i Telemark

Objects: Inheritance

- An important concept in OOP,
- Create a new class by extending an existing class,
- Example:
  - make a specific sensor class from a general sensor class
    - General sensor class: address, raw value, filtered value,
    - Specific sensor class: unit conversion (temperature, pressure, ..)
  - The new class can use all methods in the existing class.
  - Define the base methods in only one class,
  - Public class press_sensor : sensor ;

33

Høgskolen i Telemark

MS Visual Studio IDE Demo

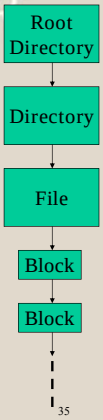
- IDE; Integrated Development Environment.
- Command line based application,
  - Input methods: Console.Read() or Console.ReadLine()
  - Output methods:
    - Console.WriteLine(" .. ") ;
      - Write a new line with the text, and ends with a newline code,
    - Console.Write(" .. ") ;
      - Write the text, without the newline code,
    - Console.WriteLine(" .. " + ivalue1 + " .. " + ivalue2 + " .. ") ;
      - Output of integer values,
    - Console.WriteLine(" ..{argnum:Fx} .. ", fvalue) ;
      - Formatted output, argument displayed as float, with x decimal(s).
- GUI based application.
  - Use buttons, textfields, and other components in Visual Studio IDE.

34

Høgskolen i Telemark

File operations

- Write or read information from disk,
- Text or binary files,
- File operations;
  - Open; must always open before any other operation,
  - Read; read data from the file, from the start or any position,
  - Write; write data at any position,
    - Normally from the start of the file or append (at the end),
  - Close;
- Ready made components for the GUI application;
  - OpenFileDialog,
  - SaveFileDialog,
  - A lot of other file and directory classes
- Remember to use try and catch on all file operations!



35

Høgskolen i Telemark

Threading

- If part of your application needs to wait for an event,
- If part of your application is duplicated,
  - Split your application into separate tasks,
  - Each task will run at the same time,
- Threading tasks use the same data segment,
  - Make sure the data is synchronized,
- Very easy to make threads in C#,

36

Høgskolen i Telemark

Graphics (1)

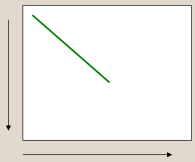
- Extend the GUI with dots, lines, or images,
- Basic drawing:
  - Create a graphic object on a form (panel, menu, etc.),
  - A pen or brush object is defined,
    - Pen; draw lines,
    - Brush; fill areas,
  - Use the graphic object methods for draw or paint lines and shapes,
  - Dispose the object when finish (dispose method),
  - Use the invalidate method to redraw a graphical object.
- System.Drawing; namespace for the graphic objects,

37

Høgskolen i Telemark

Graphics (2)

(0,0)



```
Draw a line in panel:
System.Drawing.Graphics      line ;
Pen linePen ;
Line = panel.CreateGraphics() ;
linePen = new Pen(System.Drawing.Color.Green) ;
// Draw a green line from 10, 10 to 250, 250
Line.DrawLine(linePen, 10, 10, 250, 250) ;
```

NOTE: The line must be redrawn when the panel is resized!
- Use the paint event to redraw the line,

Pen; Color, Width, DashStyle
- linePen.Width = 3 ; // Width in pixels
- linePen.DashStyle = System.Drawing.Drawing2D.DashStyle.DashDotDot ;

38

Høgskolen i Telemark

Debugging

- Trace what is happening inside the application,
- Trace any error like run-time error, ”hanging”, ”hanging the computer”, or fails to work,
- Set breakpoints to stop the application at a specific statement,
- Check the value of any variables in the application,
- Run or single step,
- Write info to the Debug Window
  - System.Diagnostic.debug.Writeline(”..”) ;

39

Høgskolen i Telemark

References

- A lot of books about C#, OOADP, and .NET
  - This presentation is based on the books;
    - ”C# 2008 and .NET Programming for Electronic Engineers”,
      - J. Allwork, Elektor, ISBN 978-0-905705-81-1
    - “Visual Studio C# 2010 Programming and PC Interfacing”,
      - J. Allwork, Elektor, ISBN 978-0-905705-95-8

  - C# , The complete Reference,
    - H. Schildt,
    - McGrawHill, ISBN 0-07-213485-2
  - C#, How to Program, introducing .NET and Web Services,
    - H.M.Deitel,
    - Prentice Hall, ISBN 0-13-062221-4
  - C# and the .NET Platform
    - A. Troelsen,
    - Apress, ISBN 1-59059-055-4

40

Appendix D

Code standard overview example

The contents only (two pages) of a C programming language coding standard from the KROHNE company. The contents indicates the number and type of sections where each section describes a specific part of the programming language and how that part should be coded.

2 Content

1 Intended Audience 1

2 Content 2

3 Introduction 4

4 Coding rules list 5

4.1 Coding rules list 5

4.2 Coding rules common mistakes examples 7

5 Coding rules explanations 8

5.1 Procedure name must not be reused 8

5.2 Assignment operator in expression is not permitted. 9

5.3 Brackets to loop body are mandatory. 10

5.4 Brackets to then/else are mandatory. 11

5.5 goto statements not permitted. 12

5.6 Number of parameters must match between definition and call. 13

5.7 Infinite loop is not permitted 14

5.8 Usage of ++ or -- operators in complex statement is not permitted. 15

5.9 Use of break statement in loop is not permitted. 16

5.10 Do not use banned function or variable. 17

5.11 Array Bound must not be exceeded 18

5.12 Do not use shift operator on signed type. 19

5.13 Shifting value must not exceed length. 20

5.14 Equality comparison of floating point is not permitted 21

5.15 #define must not be used in a block. 22

5.16 #pragma must not be used. 23

5.17 Pointer must not be assigned to wider scope 24

5.18 Executable code before an included file is not permitted 25

5.19 Duplicate use of a name in an enumeration is not permitted. 26

5.20 Casting operation on a pointer is not permitted. 27

5.21 Casting operation to a pointer is not permitted. 28

5.22 #include filename must be conformant. 29

5.23 Function return type must be consistent 30

5.24 Function and prototype return must be consistent. 31

5.25 Condition expression must be Boolean. 32

5.26 Use of bit operator on signed type is not permitted. 33

5.27 Use of ## or # in a macro is not permitted. 34

5.28 Parameter and global variable must not have same name. 35

5.29 Name must not be reused in inner scope. 36
5.30 Construct must not leads to infeasible code. 37
5.31 Macro name must be upper case. 38
5.32 Included file must be protected with #define. 39
5.33 For loop initialisation must be simple. 40
5.34 Default must be last case of switch. 41
5.35 Operation to plain char is not permitted. 42
5.36 Widening cast on complex integer expression is not permitted. 43
5.37 Cast must be applied to small types when using ~or <<. 44
5.38 operator "defined" must not contain illegal items. 45
5.39 Complex expression must have brackets. 46
5.40 Value assignation to enum is not permitted. 47
5.41 Char must not be used instead of (un)signed char. 48
5.42 Type conversion must be done with a cast. 49
5.43 Signed/unsigned conversion must be done with a cast. 50
5.44 Float/integer conversion must be done with a cast. 51
5.45 Operator [] must not be used with a pointer. 52
5.46 Function return must not be implicitly converted. 53
5.47 Procedure must not have more than one exit point. 54
5.48 All variables must be set before its use. 55
5.49 Recursion in procedure is not permitted. 56
5.50 Procedure definition must have an associated prototype. 57
5.51 Declaration types must match across a system. 58
5.52 Unreachable Code is not permitted. 59
5.53 Functions without parameter must include void explicitly 60
5.54 A function must have an explicit return value 61
5.55 A procedure must not have a return statement 62
5.56 Function parameters must have a type 63
5.57 Mod and % must be use on unsigned types only 64

Appendix E

C# examples

E.1 OOADP introduction source

```
////////////////////////////////////////////////////////////////////////
///
/// Application for reading and displaying the temperature in a console window
///
/// Demo application in C# for the SCE1306 course
///
/// Version 1.0 : NOS : 15-AUG-06
/// 1.1: Extend with time and better OO structure : NOS : 16-AUG-08
/// 1.2: Use CamelCase notation of classes, variables and methods: 11-NOV-14
///
////////////////////////////////////////////////////////////////////////
#define APP_DEBUG
using System;
using System.Collections.Generic;
using System.Text;
// Add the Threading package
using System.Threading;
//Make an application reading and displaying the temperature every 10 second
namespace TempConsoleApp
{
    /// <summary>
    ////////////////////////////////////////////////////////////////////
    public class Display
    ///
    /// Purpose: the display class and methods
    ///
    /// Version: 1.0: First version: 11-NOV-14: NOS
    /// </summary>
    {
        /// <summary>
        /// ////////////////////////////////////////////////////////////
        public Display()
        ///
        /// Purpose: constructor
        ///
        /// Version: 1.0: 11-NOV-14: NOS
        /// </summary>
        {
        }
        ////////////////////////////////////////////////////////////////////
```

```
        //
        public void Update(int loopCnt, double inTemp, double outTemp)
        //
        // Purpose: display all the temperatures
        //
        // Version: 1.0: First version: 11-NOV-14: NOS
        {
#if APP_DEBUG
            Console.Write("  [{0:D2}]", loopCnt + 1);
            Console.Write(" {0:T}", DateTime.Now);
#endif
            Console.Write(" Temperaures: Inside={0:F1}", inTemp);
            Console.Write(" Outside={0:F1}", outTemp);
            Console.WriteLine();
        }
    }
    /// <summary>
    ////////////////////////////////////////////////////////////////////
    public class TempSensor
    ///
    /// Purpose: the temp sensor id class
    ///
    /// Version: 1.0: First version: 11-NOV-14: NOS
    /// </summary>
    {
        Random RanTemp;
        double _fTemp;
        /// <summary>
        /// ////////////////////////////////////////////////////////////
        public TempSensor(int id)
        ///
        /// Purpose: constructor
        ///
        /// Version: 1.0: 11-NOV-14: NOS
        /// </summary>
        {
            RanTemp = new Random(id);
            _fTemp = 15.0F + (float)(RanTemp.NextDouble() * 10.0F);
        }
        /// <summary>
        ////////////////////////////////////////////////////////////////////
        public double GetSensVal()
        //
        // Purpose: simulating the temperature using a random sensor value
        //
        // Version: 1.0: First version: 11-NOV-14: NOS
        /// </summary>
        {
            // Simulate the temperatur from hardware +/- 0.5 deg.C.
            _fTemp += (float)RanTemp.NextDouble();
            _fTemp -= 0.5F;
            return _fTemp;
        }
    }
    /// <summary>
    ////////////////////////////////////////////////////////////////////
    public class TempMeasurement
    ///
```

```
    /// Purpose: the temp measurement class
    ///
    /// Version: 1.0: First version: 11-NOV-14: NOS
    /// </summary>
    {
        const int LOCATIONS = 2;    // number of sensor devices
        const int LOC_IN = 0, LOC_OUT = 1;
        const int MAX_LOOP = 16;    // Number of loops in demo version
        TempSensor[] rTempSensor;
        Display rDisplay;
        double[] fTempVal;
        /// <summary>
        /// ////////////////////////////////////////////////////////////
        public TempMeasurement()
        ///
        /// Purpose: constructor
        ///
        /// Version: 1.0: 11-NOV-14: NOS
        /// </summary>
        {
            int cnt;
#if APP_DEBUG
            Console.WriteLine(" Init the default temperatures ..");
#endif
            fTempVal = new double[LOCATIONS];
            rTempSensor = new TempSensor[LOCATIONS];
            // Init a set of temperature values
            for (cnt = 0; cnt < LOCATIONS; cnt++)
            {
                rTempSensor[cnt] = new TempSensor(cnt);
                fTempVal[cnt] = rTempSensor[cnt].GetSensVal();
            }
            rDisplay = new Display();
        }
        /// <summary>
        ////////////////////////////////////////////////////////////////////
        public void Run()
        ///
        /// Purpose: main loop of the application
        ///
        /// Version: 1.0: First version: 11-NOV-14: NOS
        /// </summary>
        {
            int LoopCnt = 0, TempCnt;
#if APP_DEBUG
            Console.WriteLine(" Start measuring the temperature ..");
#endif
            do
            {
                for (TempCnt = 0; TempCnt < LOCATIONS; TempCnt++)
                {
                    fTempVal[TempCnt] = rTempSensor[TempCnt].GetSensVal();
                }
                rDisplay.Update(LoopCnt, fTempVal[LOC_IN], fTempVal[LOC_OUT]);
                LoopCnt++;
                Thread.Sleep(10000);
            }
            while (LoopCnt < MAX_LOOP);
```

```
#if APP_DEBUG
            Console.WriteLine(" End measuring the temperature ! ");
#endif
            Console.Write(" Use any key to quit the application ..");
            Console.ReadKey();
        }
    }
    /// <summary>
    ////////////////////////////////////////////////////////////////////
    class TempApp
    ///
    /// Purpose: startup class of the application
    ///
    /// Version: 1.0: First version: 11-NOV-14: NOS
    /// 1.1: new structure with more classes: 31-OCT-16: NOS
    /// </summary>
    {
        static void Main(string[] args)
        {
            TempMeasurement rTempMeasure;

#if APP_DEBUG
            Console.WriteLine(" Start of main program ");
#endif
            rTempMeasure = new TempMeasurement();
            rTempMeasure.Run() ;
#if APP_DEBUG
            Console.WriteLine(" End of main program ");
#endif
        }
    }
}
//
//////////////////////////// EOC ////////////////////////////////////
```

E.2 Sensor classes with Inheritance

```
////////////////////////////////////////////////////////////////////////
///
/// Application for inheritance of sensor classes
///
/// Demo application in C# for the SCE1306 course
///
/// Version 1.0 : NOS : 15-AUG-06
///    1.1: Update methods name and use double, not float: 3-JAN-18: NOS
///
////////////////////////////////////////////////////////////////////////
using System;
using System.Collections.Generic;
using System.Text;
///
namespace SensorInheritance
{
    // Sensor base class
    // Used for reading the channels from the DAQ system and
    // tag information.
```

```
class Sensor
{
    int chId;
    string sTag;
    Random rVal;

    public Sensor(int ch, string sinfo)
    {
        chId = ch;
        sTag = sinfo;
        rVal = new Random(ch);
    }
    public virtual double GetValue()
    {
        double dVal;
        // Simulate a 0-5V input from a DAQ
        dVal = rVal.NextDouble() * 5.0F;
        return dVal ;
    }
    public string GetTag()
    {
        return sTag ;
    }
}
// Temperature Sensor
// Contains the special temperature sensor function
class SensTemperature : Sensor
{
    public SensTemperature(int ch, string txt) : base(ch, txt)
    {
    }
    public override double GetValue()
    {
        double dVal = base.GetValue();
        // Should include some linearitaztion of the PT-100 sensor
        // Convert DAQ voltage to temperature range : -25 to 125
        dVal *= 30F;
        dVal -= 25.0F;
        return dVal;
    }
}
// Level Sensor
// Contains the special level sensor function
class SensLevel : Sensor
{
    public SensLevel(int ch, string txt) : base(ch, txt)
    {
    }
    public override double GetValue()
    {
        double dVal = base.GetValue();
        // Convert DAQ voltage to level maximum range: 0-100 meter
        dVal *= 20F;
        return dVal;
    }
}
// Program Class
class Program
```

```
    {
        static void Main(string[] args)
        {
            int id, vid;
            Sensor[] sId = new Sensor[2] ;

            sId[0] = new SensTemperature(1,"Temp#1") ;
            sId[1] = new SensLevel(2,"Level#1") ;

            Console.WriteLine(" ") ;
            for (id = 0; id < sId.Length; id++)
            {
                Console.Write("  " + sId[id].GetTag());
                Console.Write(": Value:");
                for (vid = 0; vid < 4; vid++)
                    Console.Write("["+ (vid + 1) + "]=" + sId[id].GetValue() + " ");
                Console.WriteLine() ;
            }
            Console.WriteLine(" ");
        }
    }
}
//
/////////////////////////////////////////////////////////////////
```

E.3 A Facade class in C# for the NI-USB devices

```
/////////////////////////////////////////////////////////////////////////
///
/// Facade class for reading from NI-USB DAQ devices
///
/// Demo code in C# for the SCE1306 course
///
/// Version 1.0 : NOS : 15-AUG-06
///
/////////////////////////////////////////////////////////////////////////
using System ;
using System.ComponentModel ;
using NationalInstruments.DAQmx ;

namespace MeasurementSystem
{
  /// <summary>
  /// Summary description for ni_daq_mx.
  /// </summary>
  public class ni_daq_mx
  {
    private fasade_config        o_fconfig ;
    private fasade_gui          o_fgui ;
    private int              callback_error ;
    public string            dev_name ;
    public string            dev_channels ;
    private  Task            ai_task ;
    private bool            active_task ;
    private AnalogMultiChannelReader  ai_values ;
    private double[]          d_val ;
    const int DAQmxErrorPALTransferAborted = -50405 ;
```

```
//
//////////////////////////////////////////////////////////////
//
public ni_daq_mx(fasade_config or_fconf, fasade_gui or_fgui)
//
// Purpose: constructor
//
// Version: 1.0: 12-DEC-11: NOS
{
  o_fconfig = or_fconf ;
  o_fgui = or_fgui ;
  active_task = false ;
  callback_error = 0 ;
}
//////////////////////////////////////////////////////////////
//
public void set_device_name(int dev)
//
// Purpose: set the name of a DAQ device
//
// Version: 1.0: 12-DEC-11: NOS
{
  switch (dev)
  {
    case 0 :    // First I/O device
      this.dev_name = o_fconfig.iosystem_device1_name ;
      break ;
    case 1 :    // second I/O device
      this.dev_name = o_fconfig.iosystem_device2_name ;
      break ;
    default :
      this.dev_name = string.Format("Dev{0}", dev) ;
      break ;
  }
}
//////////////////////////////////////////////////////////////
//
public void set_device_channels(int[] io_ch)
//
// Purpose: set the type of channels for a DAQ device
//
// Version: 1.0: 12-DEC-11: NOS
{
  bool  first = true ;
  string  ch_str ;

  for (int ch = 0; ch < io_ch.Length; ch++)
  {
    // Check for valid sensor ID
    if (io_ch[ch] >= 0)
    {
      if (first == true)
      {
        this.dev_channels = string.Format("{0}/ai{1}", this.dev_name, ch) ;
        first = false ;
      }
      else
      {
```

```
          this.dev_channels = string.Concat(this.dev_channels, ", ") ;
          ch_str = string.Format("{0}/ai{1}", this.dev_name, ch) ;
          this.dev_channels = string.Concat(this.dev_channels, ch_str) ;
        }
      }
    }
  }
  ////////////////////////////////////////////////////////////////
  //
  public void reset_device()
  //
  // Purpose: reset a DAQ device
  //
  // Version: 1.0: 12-DEC-11: NOS
  {
    Device dev = DaqSystem.Local.LoadDevice(this.dev_name);
    dev.Reset() ;
  }
  ////////////////////////////////////////////////////////////////
  //
  public bool chk_device_connected()
  //
  // Purpose: check if DAQ device is connected to system
  //
  // Version: 1.0: 12-DEC-11: NOS
  {
    int      id = 0 ;
    bool    valid = false ;

    while (valid == false && id < DaqSystem.Local.Devices.Length)
    {
      if (string.Compare(this.dev_name, DaqSystem.Local.Devices[id].ToString(),
        true) == 0)
      {
        valid = true ;
      }
      else
        id++ ;
    }
    return valid ;
  }
  ////////////////////////////////////////////////////////////////
  //
  public bool init_device(int dev_id, ref int state, out int error,
    int first_sid)
  //
  // Purpose: Init the I/O device, read all channels at once.
  ///      NOTE: Use settings from first sensor as channel
  ///      settings.
  //
  // Version: 1.0: 12-DEC-11: NOS
  {
    error = 0 ;
    state = 0 ;
    if (active_task == false)
    {
      double            min_volt, max_volt ;
      AITerminalConfiguration    ch_term  ;
```

```
      try
      {
        state = 1 ;
        // Create a task for AI measurement
        ai_task = new Task(dev_name+"_NI_Task") ;
        switch (o_fconfig.get_iosystem_device_type(dev_id))
        {
          case (int) Constants.IOSYS_TYPE.ni_6008 :
          case (int) Constants.IOSYS_TYPE.ni_6009 :
            ch_term = AITerminalConfiguration.Rse ;
            break ;
          case (int) Constants.IOSYS_TYPE.ni_6015 :
          case (int) Constants.IOSYS_TYPE.ni_6016 :
            ch_term = AITerminalConfiguration.Nrse ;
            break ;
          default :
            ch_term = (AITerminalConfiguration) (-1) ;
            break ;
        }
        switch (o_fconfig.get_sensor_id_interface(first_sid))
        {
          case (int) Constants.IOSYS_INTERFACE.volt_2_10 :
            min_volt = 0.0 ;
            max_volt = 10.0 ;
            break ;
          case (int) Constants.IOSYS_INTERFACE.volt_1_5 :
            min_volt = 0.0 ;
            max_volt = 5.0 ;
            break ;
          default :
            min_volt = -10.0 ;
            max_volt = 10.0 ;
            break ;
        }
        // Create the virtual channels
        ai_task.AIChannels.CreateVoltageChannel(dev_channels, "",
          ch_term, min_volt, max_volt, AIVoltageUnits.Volts) ;
        // Verify the task
        ai_task.Control(TaskAction.Verify) ;
        // Start MultiChannel reading
        ai_values = new AnalogMultiChannelReader(ai_task.Stream) ;
        // Set the task to active
        active_task = true ;
      }
      catch (DaqException e)
      {
        error = e.Error ;
        o_fgui.usr_message_warning("NI-DAQmx init error: " + e.Message) ;
        active_task = false ;
        ai_task.Dispose() ;
      }
    }
    return active_task ;
  }
  ////////////////////////////////////////////////////////////////
  //
  public int read_ai_channels(int adr, ref double[] d_result, ref int error)
```

```
//
// Purpose: read a sample set from a specific DAQ device
//
// Version: 1.0: 12-DEC-11: NOS
{
  int    numbers = 0 ;

  for (int id = 0; id < d_result.Length; id++)
    d_result[id] = -1.0 ;
  if (active_task == true)
  {
    error = 0 ;
    numbers = read_sample_ai_channels(ref d_result) ;
  }
  else
    error = callback_error ;
  return numbers ;
}
//////////////////////////////////////////////////////////////
//
private int read_sample_ai_channels(ref double[] d_result)
//
// Purpose: read a single set of samples from AI channels
//
// Version: 1.0: 12-DEC-11: NOS
{
  int    numbers = 0 ;

  try
  {
    d_val = ai_values.ReadSingleSample() ;
    int ch_len = d_val.GetLength(0) ;
    if (ch_len > d_result.Length)
      ch_len = d_result.Length ;
    if (ch_len > 0)
    {
      for (int ch = 0; ch < ch_len; ch++)
      {
        d_result[ch] = d_val[ch] ;
      }
      numbers = ch_len ;
    }
  }
  catch (DaqException e)
  {
    o_fgui.status_message("NI-DAQmx read value error: " + e.Message) ;
    active_task = false ;
    callback_error = e.Error ;
    ai_task.Dispose() ;
  }
  return numbers ;
}
//////////////////////////////////////////////////////////////
//
public void close_device(int adr, ref int thread_state, out int error)
//
// Purpose: Close the NI task device driver for this device
//
```

```
    // Version: 1.0: 12-DEC-11: NOS
    {
      error = 0 ;
      if (active_task == true)
      {
        try
        {
          ai_task.Dispose() ;
          active_task = false ;
        }
        catch (DaqException e)
        {
          o_fgui.usr_message_warning("NI-DAQmx close error: " + e.Message) ;
          error = e.Error ;
          active_task = false ;
        }
      }
    }
    ////////////////////////////////////////////////////////////////
    //
    public bool valid_error_code(int error, ref string etxt)
    //
    // Purpose: check for valid return code from the NI-DAQmx library
    //
    // Version: 1.0: 12-DEC-11: NOS
    {
      bool  valid = true ;

      switch (error)
      {
        case DAQmxErrorPALTransferAborted :
          break ;
        default :
          break ;
      }
      return valid ;
    }
  }
}
//
/////////////////////////////// EOC ////////////////////
```

E.4 Facade and Publisher/Subscriber patterns

This code shows an overview of the facade pattern and publisher/subscriber pattern using a three thier architecture with the three classes TierUiLayer, TierBusinessLayer and TierDayaLayer. The TierUiLayer class is communicating with the business and data layer using methods, and the data layer is sending random events to the user interface layer by a publisher/sucscriber solution. The TierDataLayer includes a thread methods that will randomly publishing an event that the TierUiLayer is subscribing to. This publisher/subscribe solution is implemented in C# using delegate objects. Delegate objects refers to objects where a member (property or method) of one object (the receiver) is evaluated in the context of another original object (the sender). Most of the comments in the source code are removed to save some space.

```
////////////////////////////////////////////////////////////////////////////////////
///
/// Purpose: console application using both the facade and publish/observber patterns
```

```
///
/// Version: 1.0: 17-NOV-20: NOS
////////////////////////////////////////////////////////////////////////////////
using System;
using System.Threading;

namespace ObserverPublisherPattern
{
    delegate void Publisher(object sender, UiEventArgs e);
    class Program
    {
        static void Main(string[] args)
        {
            /// Make an object for starting the application
            new ProgramStart();
        }
    }
    class ProgramStart
    {
        public ProgramStart()
        {
            TierUiLayer rTierUiLayer;
            TierBusinessLayer rTierBusinessLayer;
            TierDataLayer rTierDataLayer;
            try
            {
                rTierDataLayer = new TierDataLayer();
                rTierBusinessLayer = new TierBusinessLayer(rTierDataLayer);
                rTierUiLayer = new TierUiLayer(rTierBusinessLayer, rTierDataLayer);
                rTierUiLayer.MainLoop();
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error: " + ex.Message);
            }
        }
    }
    class TierUiLayer
    {
        Publisher SubscriberUiEvent;
        TierBusinessLayer rTierBusinessLayer;
        public TierUiLayer(TierBusinessLayer prbl, TierDataLayer ptdl)
        {
            rTierBusinessLayer = prbl;
            SubscriberUiEvent = new Publisher(OnUiEventHandler);
            ptdl.PublisherUiEvent += SubscriberUiEvent;
        }
        public void MainLoop()
        {
            ConsoleKey cKey = ConsoleKey.S;
            ConsoleKeyInfo cKeyInfo;

            Console.WriteLine(" MainLoop: Q to quit ..");
            rTierBusinessLayer.InitLayer();
            while (cKey != ConsoleKey.Q)
            {
                Thread.Sleep(100);
                cKeyInfo = Console.ReadKey();
```

```
                cKey = cKeyInfo.Key;
            }
            rTierBusinessLayer.CloseLayer();
            Console.WriteLine(" MainLoop: Ending (after 10 seconds)!");
            Thread.Sleep(10 * 1000);
        }
        public void OnUiEventHandler(object sender, UiEventArgs eUiEvent)
        {
            Console.WriteLine(DateTime.Now.ToString() + " Rx the event: " + eUiEvent.sMsg + " with i
        }
    }
    class TierBusinessLayer
    {
        TierDataLayer rTierDataLayer;
        public TierBusinessLayer(TierDataLayer ptdl)
        {
            rTierDataLayer = ptdl;
        }
        public void InitLayer()
        {
            rTierDataLayer.InitLayer();
        }
        public void CloseLayer()
        {
            rTierDataLayer.CloseLayer();
        }
    }
    class TierDataLayer
    {
        bool bRunFlag;
        public event Publisher PublisherUiEvent;
        Random rRandomSleep;
        Thread thEventGeneration;
        public TierDataLayer()
        {
            rRandomSleep = new Random(DateTime.Now.Second);
        }
        public void InitLayer()
        {
            bRunFlag = true;
            // Run the EventGeneration as a thread
            thEventGeneration = new Thread(new ThreadStart(this.EventGenerationThread));
            thEventGeneration.Start();

        }
        public void CloseLayer()
        {
            bRunFlag = false;
        }
        public void FireUiEvent(UiEventArgs eUi)
        {
            if (PublisherUiEvent != null)
            {
                PublisherUiEvent(this, eUi);
            }
        }
        private void EventGenerationThread()
        {
```

```
            double dSleep;
            UiEventArgs eUiEventArgs;

            eUiEventArgs = new UiEventArgs();
            while (bRunFlag == true)
            {
                dSleep = 2.0F + (rRandomSleep.NextDouble() * 8.0F);      // Events between 4 and 10 s
                Thread.Sleep((int) (dSleep * 1000.0F));
                eUiEventArgs.iId = (int)dSleep;
                eUiEventArgs.sMsg = "SleepEvent";
                FireUiEvent(eUiEventArgs);
            }
        }
    }
    class UiEventArgs : EventArgs
    {
        public int iId;
        public string sMsg;
    }
}
///
////////////////////////////////////////// EOC ////////////////////////////////////////////
```

Appendix F

Documents

F.1 Fully dressed use case document

An example of a template for a fully dressed use case document:

Use case section	Comment
Use case name	
Scope	
Level	
Primary actor	
Stakeholders and Interests	
Preconditions	
Success Guarantee	
Main success scenario	
Extensions	
Special requirements	
Technology list	
Frequency of occurrence	
Miscellaneous	

Appendix G

OOAD software examples

G.1 Mobile Phone software OOAD

The analysis and design of different parts for the mobile phone software will be used to show the OOADP. The mobile phone is selected as this is a "process" that is well known to the readers.

G.1.1 Analysis

The analysis is to understand how the software should work for the mobile phone, what will be the main functions of this device.

Use case diagram

The use case diagram for the mobile phone is shown in Figure G.1.

Use case documents

A use case document is a text document describing one use case in more detail.

Fully dressed use case document This use case document will describe the function of the *make call* use case. The description is according to the fully dressed use case document (FDUCD), a full description of the use case behavior.

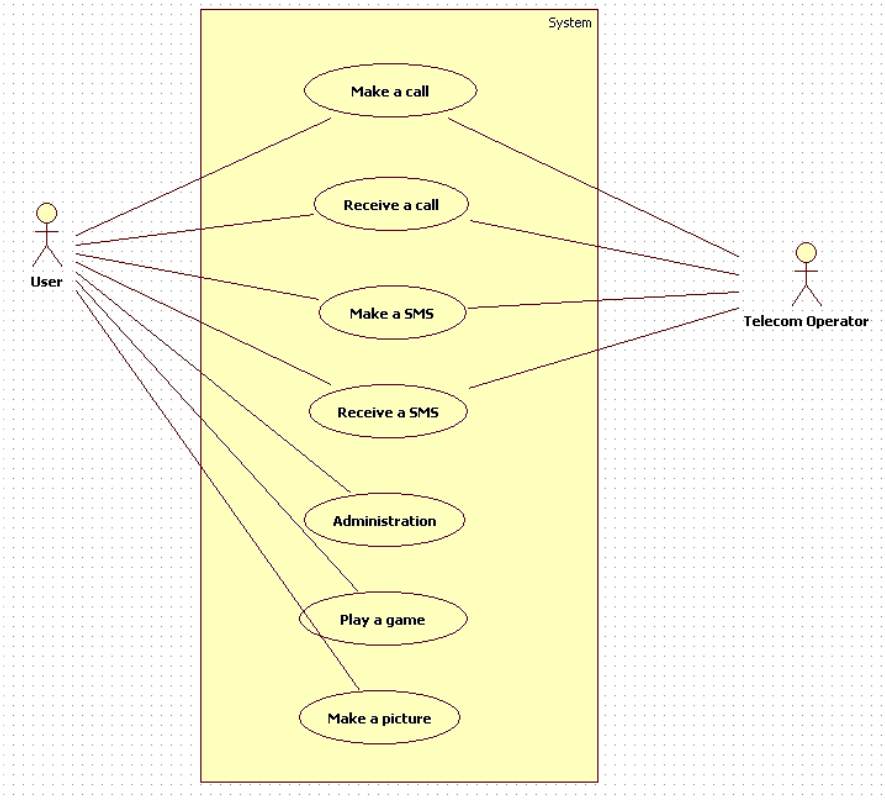


Figure G.1: The use case diagram for the software to the mobile phone.

Use case section	**Comment**
Use case name	Make call
Scope	Mobile Phone
Level	User goal
Primary actor	User
Stakeholders and Interests	None
Preconditions	None
Success Guarantee	Calling another phone
Main success scenario	
	1: Unlock the keyboard
	2: Select Call Operation
	3: Select Address List
	4: Select Receiver from the address list
	5: Press the call button
	6: Save the number in the calling list
	7: Wait for the receiver
Extensions	4a: dial the number if not in the list
	4b: Make a new index in the address list?
	6a: warning if memory full
	7a: Timeout after 30 seconds if no receiver
Special requirements	
Technology list	
Frequency of occurrence	Everytime making a call
Miscellaneous	

Fully dressed use case document with include The use case document can be extended with an include use case, in Figure G.1, an include use case can be the *release keyboard* used by the *make sms* and the *make sms* use cases. A fully dressed use case document can also indicate the include option, meaning that the use case is including a common function among several use cases. The fully dressed use case document contains the description of the *make sms* use case. **Notice the extension 1a.**

Use case section	**Comment**
Use case name	Make sms
Scope	Mobile Phone
Level	User goal
Primary actor	User
Stakeholders and Interests	None
Preconditions	None
Success Guarantee	Sending a SMS
Main success scenario	
	1: Unlock the keyboard
	2: Select the number from the address list
	3: Make the text message
	4: Save the message in the tx sms list
	5: Send the message
Extensions	1a: unlock keyboard: include release keyboard
	2a: input the number if not in the list
	2b: Make a new index in the address list?
	4a: warning if memory full
	5a: warning if no contact with the operator
Special requirements	
Technology list	
Frequency of occurrence	Everytime making a sms
Miscellaneous	

The include use case must be defined in a fully dressed use case document as well, the description will be:

Use case section	**Comment**
Use case name	Release keyboard
Scope	Mobile Phone
Level	Sub function
Primary actor	User
Stakeholders and Interests	None
Preconditions	None
Success Guarantee	Open keyboard
Main success scenario	
	1: Display the code on the display
	2: Get the open code
	3: Open the keyboard
Extensions	3a: Message if wrong code
Special requirements	
Technology list	
Frequency of occurrence	Everytime unlocking the keyboard
Miscellaneous	

Notice the **Level** (section #3) is a sub function, not a user goal.

Fully dressed use case document with extend The use case document can be extended with an extend use case, in Figure G.1, an extend use case can be the *make mms* used by the *make sms* to make mms messages. A fully dressed use case document can also indicate the extend option, meaning that the use case is including a new function to extend the functionallity. The fully dressed use case document contains the description of the *make sms* use case with this extension.

Use case section	**Comment**
Use case name	Make sms
Scope	Mobile Phone
Level	User goal
Primary actor	User
Stakeholders and Interests	None
Preconditions	None
Success Guarantee	Sending a SMS
Extension Points	Make the message, step 3
Main success scenario	
	1: Unlock the keyboard
	2: Select the number from the address list
	3: Make the text message
	4: Save the message in the tx sms list
	5: Send the message
Extensions	1a: unluck keyboard: include release keyboard
	2a: input the number if not in the list
	2b: Make a new index in the address list?
	4a: warning if memory full
	5a: warning if no contact with the operator
Special requirements	
Technology list	
Frequency of occurrence	Everytime making a sms
Miscellaneous	

The include use case must be defined in a fully dressed use case document as well, the description will be:

Use case section	**Comment**
Use case name	Make MMS
Scope	Mobile Phone
Trigger	User wants to add an image
Extension Points	Make the text message in Make sms
Level	Sub function
Primary actor	User
Stakeholders and Interests	None
Preconditions	Image(s) available in the phone
Success Guarantee	Added an image
Main success scenario	
	1: Add a menu for including text, sound or picture
	2: Include the image in the message
	3: Show the image information in message
Extensions	1a: select text, use the make SMS function
	2a: error including image, inform user
Special requirements	
Technology list	
Frequency of occurrence	Everytime making a MMS
Miscellaneous	

Notice the **Trigger** (section #3), **Extension Points** (section #4), and **Level** (section #5) sections of the document.

G.2 I/O measuring device software

The I/O measurement system software will be used for reading and writing values from the I/O devices connected to the measurement system.

G.2.1 Analysis

The analysis is to understand how the software should work for the measurement system.

Use case diagram

The use case diagram for the measurement system is shown in Figure G.2.

Use case documents

A use case document is a text document describing one use case in more detail. This use case document will describe the function of the *read channel* use case.

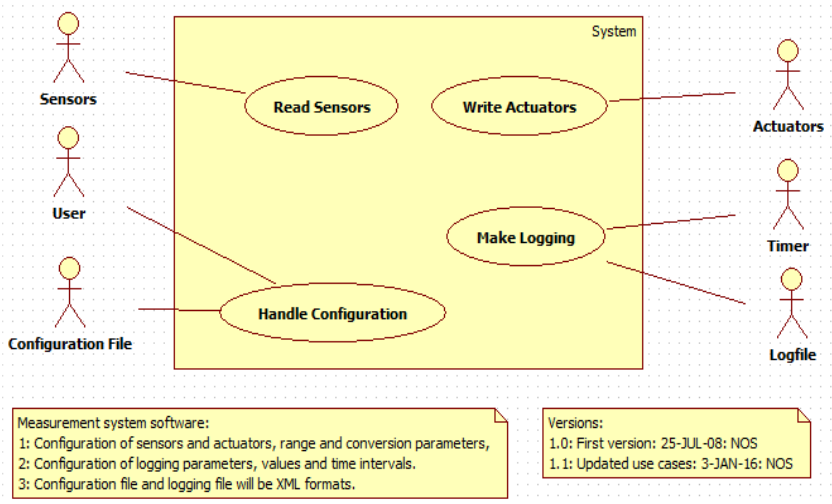


Figure G.2: The use case diagram for the software to the measurement system.

Use case section	**Comment**
Use case name	read channel
Scope	Measurement system
Level	User goal
Primary actor	Sensor
Stakeholders and Interests	None
Preconditions	None
Success Guarantee	The status and the value from the sensor
Main success scenario	1: get the channel number, 2: get the configuration of this channel, 3: setup the channel configuration 4: read the value from the channel 5: timestamp the value 6: save the value and the timestamp 7: set the next channel number
Extensions	2a: no configuration of this channel, go to step 7, 3a: error setting up the channel set channel error, goto step 7, 4a: error reading the channel set channel error, goto step 7, 7a: the last channel number, reset the channel number.
Special requirements	None.
Technology list	Support 4-20mA devices.
Frequency of occurrence	
Miscellaneous	Different I/O interfaces?

G.3 Alarm clock

Figure G.3 shows many of the states in an alarm clock. Include more information as the events and transistions. How to deal with user inputs? How to deal with programming the alarm clock?

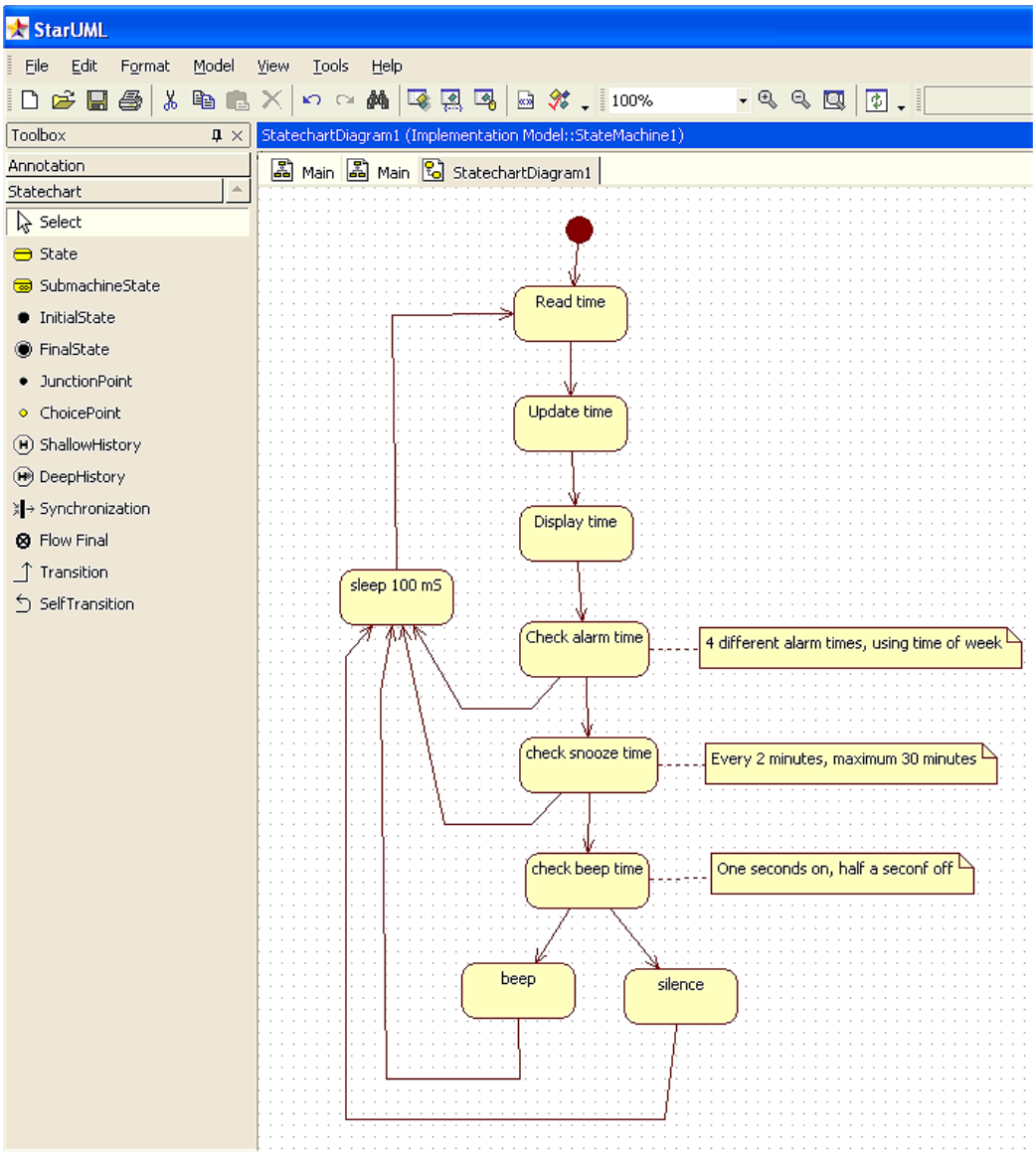


Figure G.3: A statechart diagram with many of the states in an alarm clock.

Appendix H

GiT installation

Download the GiT software from internet and install a version on your system. The following Figures H.1, H.2, H.3 and H.4 give an indication about how to install git on a Windows platform. Note that these Figures may depend on the current version of the software, these Figures are from a version installed in 2015.

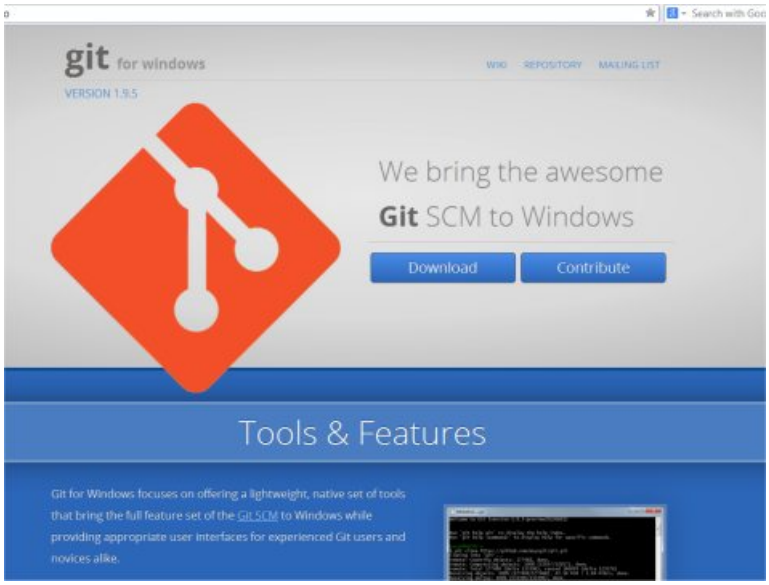


Figure H.1: Starting downloading the version for the Windows platform.

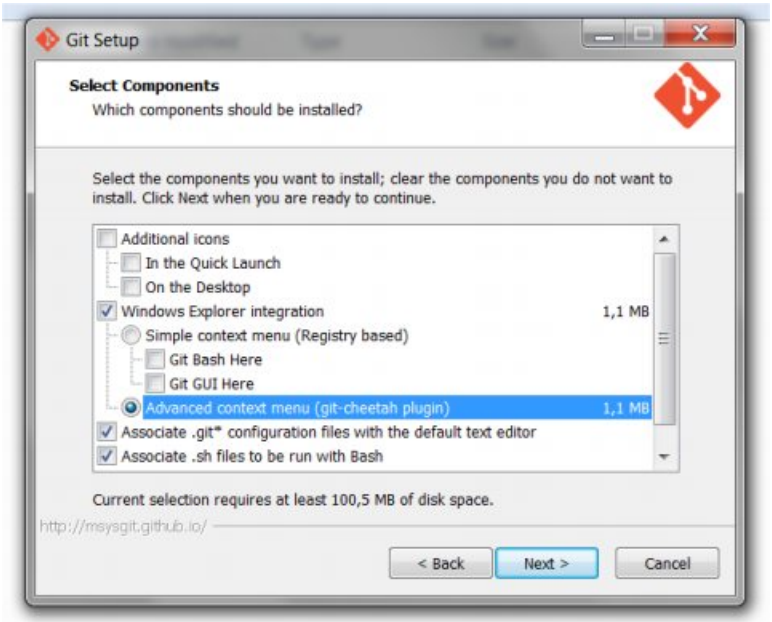


Figure H.2: Selections #1 when installing.

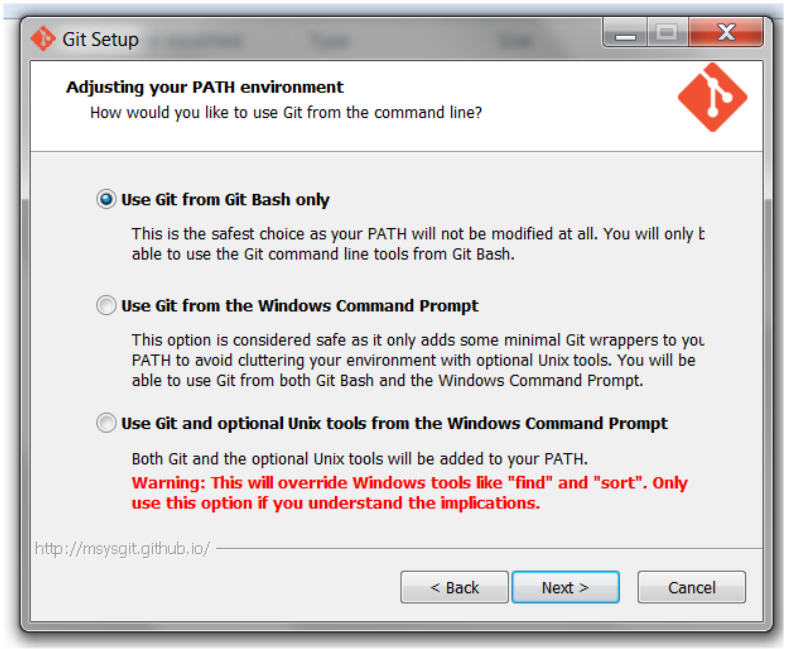


Figure H.3: Selections #2 when installing.

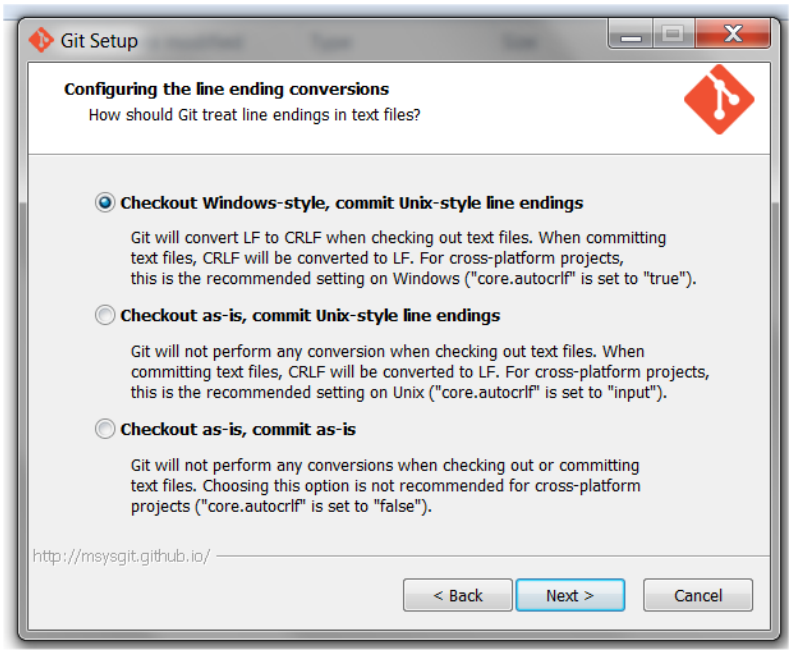


Figure H.4: Selections #3 when installing.

Bibliography

Allwork, J. (2009), *C-Sharp 2008 and .NET Programming for Electronic Engineers*, Elektor International Media BV, Postbus 11, 6114 ZG Susteren, The Netherlands. ISBN 978-0-905705-75-0.

Allwork, J. (2010), *C-Sharp 2010 Programming and PC interfacing*, Elektor International Media B.V.

Ansari, M. T. J., Pandey, D. & Alenezi, M. (2018), 'STORE: Security Threat Oriented Requirements Engineering Methodology', *Journal of King Saud University ŰComputer and Information Sciences* .

Bentham, J. (2002), *TCP/IP Lean, Web servers for embedded systems*, second edn, CMPBooks.

Bentley, J. P. (2005), *Principles of Measurement Systems*, 4th. edn, Pearson Education Limited, Essex, England.

Bittner, K. & Spence, I. (2006), *Mangaing Iterative Software Development Projects*, Addison Wesley Professional.

Bruegge, B. & Dutoit, A. H. (2010), *Object-Oriented Software Engineering*, third edn, Pearson.

Chacon, S. (2009), *Pro Git; Description of the Git version control system.*

Cole, M. R. (2018), *Hands-On Machine Learning with C#*, Packt>.

Comer, D. E. (1991), *Internetworking with TCP/IP; Volume I; Principles, Protocols and Architecture*, second edn, Prentice-Hall International Editions.

Crowcroft, J. & Phillips, I. (2002), *TCP/IP and Linux Implementation; Systems code for Linux Internet*, Wiley.

Cruzes, D. S., Jaatun, M. G., Tøndel, I. A. & Bersmed, K. (2020), Sju steg til bedre programvaresikkerhet (in Norwegian), resreport, Sintef Digital. Published in ComputerWorld Norway.

Daler, T., Gulbrandsen, R., Høie, T. A. & Sjølstad, T. (2019), *Håndbok i datasikkerhet (in Norwegian)*, 4 edn, Fagbokforlaget. ISBN 978-82-450-2789-1.

Dickinson, M. (2011), *Introduction to Control Engineering*, Elektor International Media BV. ISBN 978-0-905705-99-6.

Douglass, B. P. (1999), *Doing Hard Time Developing Real-Time Systems with UML, OObject, Frameworks, and Patterns*, Addison Wesley.

Dybå, T., Dingsøyr, T. & Moe, N. B. (2004), *Process Improvement in Practice A Handbook for IT Companies*, Kluwer Academic Publisher. SINTEF Information and Communication Technology, Trondheim, Norway.

Elm (2016), *ELM327 - OBD to RS232 Interpreter.* Accessed FEB-2019.

Fowler, M. & Scott, K. (1997), *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, USA.

Holmstedt, V. (2019), *Objektorientert systemutvikling og UML (in Norwegian)*, third edn, Fagbokforlaget, Norway.

Houldsworth, J. (1990), *OSI Handbook*, International Computers Ltd. (ICL), UK.

Jeffries, R. (2004), *Extreme Programming Adventures in C#*, Microsoft, USA.

Jensen, H. H. & Billington, E. (2007), *Integrasjon Som Konkurranseparameter, Suksess Med SOA (In Norwegian)*, daVinci Conculting. ISBN 978-82-8152-011-0.

Jøsang, A. (2021), *Informasjonssikkerhet - Teori og praksis (in Norwegian)*, Universitetsforlaget.

Kniberg, H. (2009), Kanban vs. scrum, Technical report, InfoQ.

Kolda, T. & Torczon, V. (2011), 'Top ten ways to lose an audience', *SIAM News* **44**(3).

Kruchten, P. (2000), *The Rational Unified Process (UP) An Introduction*, second edn, Addison-Wesley, USA.

Larman, C. (2002), *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2. edn, Prentice Hall, USA.

Larman, C. (2005), *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and Interactive Development*, 3. edn, Prentice Hall, USA.

Leffingwell, D. & Widrig, D. (2000), *Managing Software Requirements - A Unified Approach*, Addison-Wesley.

Levermore, G. J. (2000), *Building Energy Management Systems Applications to Low-Energy HVAC and Natural Ventilation Control*, second edn, E and FN Spon.

Lieberman, B. A. (2007), *The Art of Software Modeling*, Auerback Publications, Taylor and Francis Group.

Magda, Y. (2009), *Complete Practical Measurement Systems Using a PC-based*, elektor. ISBN 978-0-905705-79-8.

Magda, Y. (2011), *Linux PC-based Measurement Electronics, hardware and software*, elektor. ISBN 978-1-907920-03-5.

Mahmoud, Q. H. (2004), *Middleware for Communications*, Wiley, West Sussex PO19 8SQ, England.

Manassis, E. (2004), *Practical Software Engineering: Analysis and Design for the .NET Platform*, Pearson Education, Inc, Boston, USA.

Martin, J. (1993), *Principles of Object-Oriented Analysis and Design*, Prentice Hall. ISBN 978-0137208715.

Mazonka, O. (2016), Blockchain: Simple explonation, Technical report, Defence Science and Technology Group (DST) Canberra, Australia Join this Institution.

McCall, G., Burden, P., Burnard, A., Hennell, M., Hills, C., Montgomery, S., Tapp, C. & Whiting, L. (2008), Misra-c: Guidelines for the use of the c language in critical systems, Technical report, MISRA (The Motor Industry Software Reliability Association). 2. Ed.

McMillan, M. (2007), *Data Structures and Algorithms using C#*, Cambridge University Press.

McNamara, J. E. (1988), *Technical Aspects of Data Communication*, Digital Equipment Corporation, USA.

Mueller, J. P. (2009), *C Design and Development*, Wiley Publishing, Inc.

Newkirk, J. W. & Vorontsov, A. A. (2004), *Test-Driven Development in Microsoft .Net*, Microsoft professional.

Oestereich, B. (2001), *Developing Software with UML Object-Oriented Analysis and Design in Practice*, 2. edn, Addison Wesley.

Olsson, G. & Piani, G. (1998), *Computer Systems for Automation and Control*, 2nd edn, Prentice Hall International (UK) Ltd., London, UK.

Page-Jones, M. (2000), *Fundamentals of Object-Oriented Design in UML*, Addison-Wesley, USA.

Pan, J.-S., Lin, J. C.-W., Sui, B. & Tseng, S.-P. (2019), Genetic and Evolutionary Computing, *in* 'Advances in Intelligent Systems and Computing', Springer. Proceedings of the Twelfth International Conference on Genetic and Evolutionary Computing, December 14-17, China.

Pender, T. (2003), *UML Bible*, Wiley Publishing, Inc., USA.

Rational (1998), Rational unified process best practices for software development teams, White Paper TP026B, Rev 11/01, Rational Software.

Robertson, S. & Robertson, J. (2013), *Mastering the Requirements Process Getting Requirements Right*, third edn, Addison Wesley.

Royce, W. W. (1970), 'Managing the development of large software systems', *IEEE WESCON* **26**, 328–338.

Santos, E. T. (2009), Bim and erp: Finding similarities on two distinct concepts, *in* 'CIB-W102 Rio de Janeiro, RJ, Brasil'.

Scacchi, W. (2001), *Encyclopedia of Software Engineering*, 2 edn, John Wiley and Sons, chapter Process Models in Software Engineering.

Schildt, H. (2002), *C#: The Complete Reference*, Mc Graw Hill, Osborne, USA.

Schwaber, K. & Sutherland, J. (2011), The scrum guide; the definitive guide to scrum: The rules of the game, Technical report, scrum.org.

Shoemaker, M. L. (2004), *UML Applied: A .NET Perspective*, Apress, USA.

Skeie, T. (2014), Does limit on work-in-progress (wip) in software development matter?, Master's thesis, University in Oslo (UiO).

Sommerville, I. (2011), *Software Engineering*, 9. edn, Pearson.

Stallings, W. (2001), *Wireless communication and networks*, Prentice-Hall International Editions.

Stevens, W. R. (1990), *Unix networking programming*, Prentice-Hall International Editions.

Troelsen, A. (2003), *C# and the .NET Platform*, Apress, USA.

Viega, J. & McGraw, G. (2002), *Building Secure Software*, Addison-Wesley Pearson Education.

Wang, A. J. A. & Qian, K. (2005), *Component-Oriented Programming*, John Wiley and Sons, New Jerey, USA.

Whitten, J. L. & Bentley, L. D. (1998), *Systems Analysis and Design Methods*, fourth edn, McGrawHill.

Williams, J. D. (1995), *What every software manager must know to succed with Obect Technology*, Prentice Hall.

www.wikipedia.org (2006), 'Wikipedia'. Wikipedia, the free encyclopedia.

Özgül, F. (2012), *Design your own PC visual processing and recognition system in C#*, Elektor.

Index

active class, 101
actor, 68, 69
    time, 70
adapter
    USB to serial port adapter, 126
agile, 20, 25
agile development process models, 20
api
    application programming interface, 244
    rest, 244
architecture
    tier, 91, 93, 94
arduino, 207, 260
    loop, *207*
    setup, 207
association class, 101
asymmetric algorithm
    cryptic, 234
asynchronous communication, 124
at commands
    attention commands, 129

binary searching, 224
block chains
    cryptic, 237
bok
    body of knowledge
        system architect, 93
bubble sort, 220
buffer
    circular, 227
    FIFO, 225
    LIFO, 225
    queue, 226
    ring, 227
    stack, 226
bug tracking system, 260
business model, 24, 63, 65, 75, 76
business use case, 63

C#
    Arrays and strings, 53
    Classes, objects, and methods, 54
    Constructor, 56
    Data types, 51
    Function parameters, 56
    General, 50
    Inheritance, 57
    Method overloading, 57
    Operators, 51
    Program control, 51
camel case
    lower camel case, 35
    upper camel case, 35
checksum
    cryptic, 238
ciphertext, 233
circular buffer, 227
class, 9, 36, 54, 56, 57
    diagram, 97
class responsibility collaboration card
    CRC card, 97
coding standard, 12, 35, 259
    camel case, 35
    guidelines, 35
    naming convention, 35
collaboration diagram, 98
Common Language Infrastructure, 43
Common Language Runtime, 43
Common Language Specification, 43
communication diagram, 98
compiler, 43
component, 257
component based development, 257
component oriented development, 5
construction, 17
controller area network
    CAN bus, 129
crypt, 146
cryptic
    advanced encryption standard (AES), 234
    asymmetric algorithm, 234
    block chains, 237
    checksum, 238
    data encryption standard (DES), 234
    pgp, 236
    public keys, 237
    rsa, 234
    symmetric algorithm, 234
CSV
    Comma Separated Values, 230
CVS, 28
cyclic redundancy check
    CRC, 238

DAQ
    data acquisition system, 72
daq, 245
decrypt, 146

decryption
encryption, 233
delegate objects, 305
deployment diagram, 212
development process
agile, 20
kanban, 21
lean, 21
rapid prototyping, 19
RUP, 21
scrum, 20, 21
spiral, 19
UP, 21
waterfall, 19
XP, 21
devops, 26
diagram
activity, 200
class, 10, 13, 54, 62, 86, 97, 100, 112
collaboration, 99
communication, 99
deployment, 212
interaction, 10, 12, 112
object, 13, 54, 62, 97
package, 189
sequence, 10, 62, 86, 98
state, 10, 202
system sequence, 62, 76
UML, 6
use case, 10, 11, 62, 66, 68, 83, 183, 185
document
fully dressed use case, 75–77, 104, 161, 183, 185, 260, 309
domain model, 72
dynamic link library
DLL, 256

Eclipse, 36
elaboration, 17
elm327 interface
ELM327, 129
encryption, 232
decryption, 233
EOBD
obd, 129
event, 198, 199, 305

FIFO buffer, 225
file format
CSV, 230
fully dressed use case document, 75–77, 104, 161, 183, 185, 260, 309
function, 34
FURPS, 66, 190

gdpr
general data protection regulation, 233
git, 30, 31
file states, 31
github, 31
gps
global position system, 125
Grady Booch, 6
GRASP, 190, 192, 215
Controller, 105
Creator, 105
General Responsibility Assignment Software Patterns, 105
High Cohesion, 105
Information Expert, 105
Low coupling, 105

IDE
Integrated Development Environment, 35
MS-Visual Studio 2010, 36
IIoT
industrial internet of things, 244
inception, 17
incremental, 5
insertion sort, 222
Integrated Development Environment
IDE, 17
interaction diagram, 95, 99
interface, 244
internet of things
IoT, 5
interpreter, 43
IoT, 145
internet of things, 244
iteration, 20
iterative, 5
Ivar Jacobsen, 6

James Rumbaugh, 6
java, 3
javascript, 3

kanban, 21, 26
Kristen Nygaard, 6

layer design
tier architecture, 94
LIFO buffer, 225
loopback testing, 126

meeting
minutes of meeting, 267
notice of meeting, 267
method, 4, 34, 50, 95
mob programming, 21
module testing, 250
MS-Visual Studio, 82

naming convention, 35
guidelines, 35
NetBeans, 36
NMEA 183
gps, 126

OBD-2
    obd, 129
OBD-II
    obd, 129
object, 4, 9
    diagram, 97
object reference, 189
object-oriented analysis and design
    OOAD, 8
Ole-Johan Dahl, 6
On-board diagnostic
    obd, 129
OOAD, 8, 18
operation contract, 80
OSI model, 124, 142
override, 240

pascal style
    camel case, 35
password, 239
    most used, 239
    tips, 239
pattern, 7, 10, 27
    adapter, 195, 216
    composite, 197, 217
    controller, 105, 215
    creater, 107
    creator, 215
    design pattern, 104
    facade, 217
    factory, 195, 216
    fasade, 197, 214
    GoF (Gang-of-Four), 195
    GRASP, 105
    high cohesion, 106, 216
    indirection, 194, 216
    information expert, 105, 215
    low coupling, 107, 216
    Model View Controller
        MVC, 199
    Observer, 198
    observer, 214, 217
    polymorphism, 192, 216
    protected variations, 195, 216
    pure fabrication, 194, 216
    singleton, 195, 216
    strategy, 197, 216
Perforce, 30
pgp
    cryptic, 236
    pretty good privacy, 236
procedure, 34
programming language
    Basic, 260
    C, 260
    Matlab, 260
    Pascal, 260
property, 49, 50, 95
prototype, 68
pseudo coding, 100, 220–225, 227, 230
public keys
    cryptic, 237
publisher/subscribe, 153
publisher/subscriber, 198, 305
python, 3

queue buffer, 226

rapid protoypes process model, 19
Rational Rose, 82
rational unified process, 5
RCS, 28
refactoring, 7
report
    formal report, 265
    informal report, 265
requirement, 64
reverse engineering, 93, 261
ring buffer, 227
robotic process automation
    rpa, 244
rpa
    robotic process automation, 244
rs-232c, 124
rs-422, 124
rs-485, 124

SCCS, 28
scrum, 20, 25
scrum master
    scrum, 20
searching
    binary, 224
    sequential, 223
selection sort, 221
semaphore, 229
sequence diagram, 97
sequential searching, 223
serial port
    Close() method, 136
    GetPortNames() method, 134
    Open() method, 135
    Read() method, 136
    Write() method, 137
serial port adapter, 126
service oriented architecture
    SOA, 5
Simula, 6
    Kristen Nygaard, 6
    Ole-Johan Dahl, 6
singleton class, 101
socket
    Close() method, 144
    connection, 142
    GetHostName() method, 144
    GetStream() method, 144
    Ip Address, 144

Read() method, 144
TcpClient() method, 144
TcpListener() method, 144
Write() method, 144
software development, 2
software engineering, 2, 5, 27
software security
security, 232
sorting, 220
bubble, 220
insertion, 222
selection, 221
source control
CVS, 28
git, 30
Perforce, 30
RCS, 28
SCCS, 28
Sourcesafe, 30
SVN, 30
Sourcesafe, 30
specification, 63, 64
spiral process model, 19
sprint
scrum, 20
stack buffer, 226
StarUML, 82
sub routine, 34
SVN, 30
symmetric algorithm
cryptic, 234
synchronous communication, 124
system architect
body of knowledge
bok, 93
system testing, 251

TCP/IP protocol, 142
test case, 248
test levels, 250
test plan, 248
textbook, viii
tier architecture, 197
layer design, 94
transition, 17

uart, 125
UML, 6
UML tool
Rational Rose, 82
StarUML, 82
Visual Studio, 82
unified modeling language, 5, 6
UML, 62
unified process, 5, 6, 260
UP, 18
unified process model, 22
Unified Software Development Process
UP, 18
unit test, 161
unit testing, 250
UP
RUP, 22
Unified Software Development Process, 18
use case, 5, 68, 69, 132
use case diagram, 68, 183, 185, 212
actor, 68
include, 183
measurement system, 70, 72
stereotype, 183
use case, 68
use case document, 75, 183, 185
brief, 75
casual, 75
fully dressed, 75
use case driven, 5, 6, 25
use case model, 68
user story, 5

Version Control System
Source control, 27
virtual, 240

waterfall, 25
waterfall process model, 19
wireshark
TCP/IP packet analyzer, 146

xp
extreme programming, 20