ORDISOFTWARE™ ENGINEERING

AGILE CREATION OF OBJECT-ORIENTED APPLICATIONS

MANUFACTURING SOFTWARE GUIDELINES

METHODOLOGY & PROGRAMMING

VERSION 0.5

OLIVIER ROGIER

WWW.ORDISOFTWARE.COM

GITHUB.COM/ORDISOFTWARE/GUIDELINES

VERSION HISTORY

VERSION 0.1 - APRIL 5, 2018

- Created on October 2016.

VERSION 0.2 - APRIL 8, 2018

- The first word of « manufacturing », « building » and « crafting » was chosen for the title of this document because the second usually means « generating the machine code » during the production process and the latter is only recently used in our business and this word designates exactly what is our job which is to create software with our hands and with our minds, and we will always do that by this way.
- Since in French « développement » and « programmation » are synonyms but in English « development » means « engineering » the title of this document has been revised.

VERSION 0.3 - JANUARY 14, 2019

- Add agility and articles about interface and singleton keywords.

VERSION 0.4 - SEPTEMBER 3, 2019

- Update tools and bibliography.

VERSION 0.5 - JULY 3, 2020

- Upgrade from Word 2010 to Word 2019
- Update tools and bibliography.
- Some writing corrections.

TABLE OF CONTENT

LICENSE AND COPYRIGHT

FOREWORD

This document presents some development guidelines to produce libre, personal, private, commercial, and military software.

They are a description of how the author tries to work currently. They are considerations coming from the practice of manufacturing own and business applications. They are generally basic and obvious. They are not absolutes and not something imposed as are arithmetic and geometry. They are malleable and improvable like lots of things in this wise human world.

Each computer practitioner as everyone has its own rules forged teacher after teacher, talk after talk, book after book, line after line, launch after launch, pixel after pixel, click after click, error after error, reboot after reboot and update after update.

Everyone mostly thinks having the best system, since it comes from learnings those work. Everyone think to have the best for self and for doing some things, each time this fact is thanked. Everyone sometimes just wants most of the time do tomorrow a better work than yesterday.

A programming system does not escape to the difficulty to work with others that have different means to do some things while improving each without making war to impose one while saying everyone is free to justify the denial of the existence of numbers and letters that are the sole cause of the reality created by the chromosomic intelligence of this area whose the first rule of any legal activity is democratically applicable for each to not willingly harm anybody.

DISCLAIMER

The author is not very advanced in the way of writing in English.

He was not able to learn to speak and write this language properly, and not so much better French. But he knows well things like `start-stop`, `if-then-else` and `call-return`.

He uses a lot Google's online search engine and translator with English⇔French articles of Wikipedia and Wiktionary, as well as MS Word's linguistic tools.

He hopes that the reader will not hold against him for his way to express, for the tone he uses and for his mistakes.

WHO THIS DOCUMENT IS FOR?

This document is for anyone who wants to know how the author uses computing technologies and tools to fabricate computer programs.

It mainly refers to Agile thoughts and C#.NET but it can be used with most of systems. It covers mechanisms related to structuring items of a project and elements of an application.

It may be enhanced as it allows well creating legal and allowed software that works properly.

HOW THIS DOCUMENT IS ORGANIZED

This document is divided in seven parts:

- « *License* » specifies the terms of use for this document.
- « *Foreword* » presents this document and the author.
- « *Methodology* » is dedicated to the agile thinking.
  - « *Agility* » resumes the agile theory.
  - « *Dichotomy* » resumes the main steps of a project.
  - « *Documentation* » indicates the types of notes produced for a project.
  - « *Modeling* » resumes how diagrams are prominent.
  - « *Ecosystem* » lists some agile variants.
- « *Programming* » is dedicated to
  - « *Tools* » indicates the means used by the author.
  - « *Paths* » indicates how are organized the file elements of a project.
  - « *Naming* » specifies the standards used to write the source code.
  - « *Commenting* » specifies the standards used to describe the source code.
  - « *Formatting* » specifies the standards used to render the source code.
- « *UI design* » indicates some user interface practices currently used by the author.
- « *Using Git and GitHub* » indicates some conventions currently used by the author.
- « *Bibliography* » indicates some books related to computers in the most recent version.

CONVENTIONS USED IN THIS DOCUMENT

Phrases use mainly the « French double angle quotes ».

The "Typewriter identical double quotes" is used to distinguish a technical thing.

A section that is intended to be written in a future release indicates:

This section is undescribed yet.

The mention of a computing artifact looks like:

```
Menu / Submenu / Action
Filename.ext
www.domain.tld
```

RGB colors are noted as: #000000

ABOUT LIBRE SOFTWARE

There are two categories of software: those which are proprietary, and we must usually pay for their use, and those which are libre and we have no obligation to purchase.

Commercial and libre software are not necessarily opposable, and sometimes the objectives have no relationship with their differences that can be mixed according to the domain, the need, the type, and the scope of a project.

In both cases, developers supply an immaterial work through a physical medium, for which they are intellectually the authors, and which takes time and investment. The purpose of a work being to live and survive, for oneself and for others, free software is not thus synonymous of gratuitous, unless it is a public service funded by taxes.

Donations are a source of income for free software designers. Just like we are free to use these programs, we are also free to define what we give according to our means. Sometimes the authors do not ask for money for various reasons.

But the shareware donation system, outside the case of amateurism regarding few currency units, except for humanitarian work not controlled by the State, is a false and problematical litigious solution for a false cash flow problem, and from a fiscal point of view it would be more accurate to consider the libre purchasing as commercial sales when the product is an intangible deliverable, which requires to not flat rate taxing the existence of an entity and only the generated flow of money.

Developing free or open source software is a vision about source code, sharing of knowledge and evolution of computing. You can read different points of view from:

- Free Software Foundation: `www.fsf.org/licensing/essays/free-sw.html`
- Open Source Initiative: `www.opensource.org/docs/osd`
- Creative Commons: `creativecommons.org/about/licenses`.

ABOUT THE AUTHOR

Olivier Rogier is a software craftsman mainly skilled in C#.NET and Delphi.

Such was the destiny of his abilities, of his will, of lived experience and of opportunities.

Despite constant unjustified and illegitimate oppressions and aggressions, he worked and works day and night every day when that is possible since his childhood for becoming and being a computer programmer, regardless of his results that were sometimes good and sometimes bad.

He was brought up with Basic, Assembler, C and C++ languages. His main aptitude is the object code, and the conceptualization of the data and its processing.

When he was ten, the school has put a computer in his hands, and one made him write a program on this machine equipped with a keyboard and a screen. One hour later, he said to himself that when he grew up, he would be a programmer.

As some of the first generation, he read some books and magazines. He read and reread them to know by heart the keywords of the language and to know how to control the elements of the machine. He entered by hand codes of little games and system hacks. Then he started writing his own programs. At first he bought a few games, then people from schools showed him how to copy the tapes to exchange them, and then they began to give themselves lots of software copies on floppy disks he accepted without knowing the value of the work.

The low secondary school guidance counselor told him that the best for him was to make an "IUT Informatique", and next an engineer school according to his results. He was entirely agreed even if he knew nothing about many things. But it did not go very well as planned and he did not follow the three quarters of courses. However, he had a particularly good teacher of analysis and design of information systems. Then his first project leader taught him everything there was to know in outline on his business and he has worked for major companies and big medical and financial organizations.

He now considers the right and need of the source code of all software sold or distributed free of charge, and therefore not falls within the internal and legal activity of a group nor the national security, to be as free and monetizable by its producer as the text of a book because of the immutable principle that a code hidden to the public is like a book hidden to the public.

To learn more about him:

- Twitter: `twitter.com/ordisoftware`
- Facebook: `www.facebook.com/ordisoftware`
- LinkedIn: `www.linkedin.com/in/ordisoftware`
- Contact: `www.ordisoftware.com/contact`
- Profile: `www.ordisoftware.com/about/author`
- Projects: `www.ordisoftware.com/projects`
- Blog: `www.ordisoftware.com/blog`
- Skills: `www.ordisoftware.com/business/skills`
- Achievements: `www.ordisoftware.com/business/history`
- Bibliography: `www.ordisoftware.com/business/bibliography`
- Service offer: `www.ordisoftware.com/services`

METHODOLOGY

AGILITY

OVERVIEW

Agile methods are the result of the practice and the afterthought from the use of methods called « traditional » that they incorporate and expand based on the following notions:

- *Iterative method*: the project is realized by compartments or portions, through the concepts of objects, components, and packages.
- *Incremental method*: the project is realized by progression or refining, through the implementation of abstraction, polymorphism, and genericity.

And:

- *Scenario*: these methods of production are based on unitary specifications of the functionalities that are derided into tasks or steps.
- *Deliverable*: a functional application is frequently and regularly built to lead the advance of these methods, from the initial model until the last prototype that became the final software.

VALUES

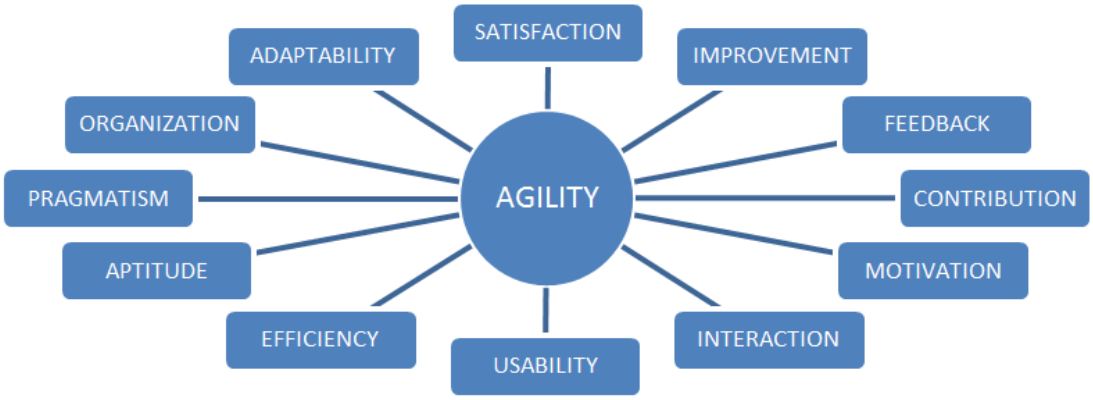
Agile methods rely on four basic values to master architectures:

- *Interaction*: communication has priority over methods and tools.
- *Result*: a program that works has priority over documentation.
- *Adaptability*: regular participation has priority over negotiations.
- *Improvement*: changing has priority over planning.

PRINCIPLES

These values are detailed in twelve principles:

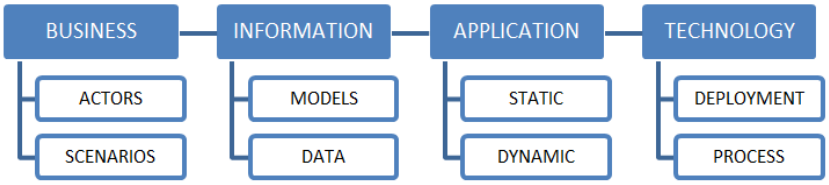
- *Improvement*: changing has priority over planning.
- *Satisfaction of the result:* priority is to deliver useful software to the user.
- *Improvement*: ability to change is a competitive advantage.
- *Feedback*: action is based on the regular delivery and the user response.
- *Contribution*: the different specialized contributors are regularly solicited.
- *Motivation*: environment and support are essential to success.
- *Interaction*: communication is the way of transmitting information.
- *Usability*: software that works is the indicator of progress.
- *Efficiency*: adopting a comfortable rhythm is the way to get the result.
- *Aptitude*: expertise and quality are continuously evaluated.
- *Pragmatism*: simplicity is even more essential that the project is complex.
- *Organization*: sharing of activities provides the best software.
- *Adaptability*: mutual and regularly introspection about the effectiveness adjusts the behavior of the team.



VIEWPOINTS

The project is usually approached from four considerations and five viewpoints:

- *Organization*: sharing of activities provides the best software.
- *Business* & *Use cases*: actors and scenarios.
- *Information* & *Design*: models and databases.
- *Application* & *Implementation*: static and dynamic aspects.
- *Technology* & *Deployment and process*: infrastructure and components.



DICHOTOMY

According to the Unified Process, phases of development are:

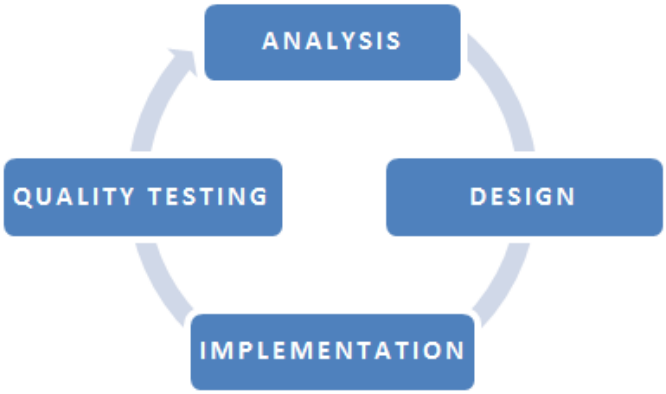
- *Inception* for *Initialization of the project*.
- *Elaboration* for *Analysis and high-level design*.
- *Construction* for *Low-level design and implementation*.
- *Transition* for *Quality testing and releasing*.

It is necessary to consider also this one:

- *Learning* for *Issues related to activities for training and technology intelligence.*

These phases of data-processing projects take place in four successive and retroactive steps that are imbricated one inside the other:

- *Transition* for *Quality testing and releasing*.
- The *analysis* defines « what to make » by specifying the technical elements to conceive to manage the entities and the data of your domain.
- The *design* defines « how to make » by identifying the items which make the program essential, as well as the choices to achieve them. Sketching the database and the user interface provides an indication of the tasks of development and the relevance of the selected solutions.
- The *implementation* is the writing of individual software components previously defined in an object-oriented language that provides quality, reliability, robustness, modularity, scalability and safety of the processings.
- The *quality testing* is an audit of the code, the checking the data coherence and the validation of the use of the program.



When there is no longer any step to perform, no action to be taken, that everything works smoothly and that the desired result is reached, the software is considered as finished and it is ready for deployment and allocation to the maintenance cycle.

MODELING

Whether thanked or represented, models precede, underlie, document, and validate the production of software of quality.

The use of relational and object-oriented modeling brings simplicity, clarity, and modularity in the conceptual representation of real things.

The author usually uses the Entity–Relationship model and the UML standard to specify and visualize structures, functions, and interactions of systems.

To obtain the desired result, the modeling and the implementation are continuously set in correspondence.

The constant review of models and code, associated with the refactoring, are essential methods to success.

DOCUMENTATION

GUIDELINES

Software guidelines are the rules that define how to create applications.

« Programming Guidelines » indicates technical and structural means used for the production.

« Methodology Guidelines » indicates executive and functional processes.

Some others can be made like for user interfaces that is included this guide, for user experience or for robots' specificities.

They both take part in the « Manufacturing Software Guidelines » package.

They are used to establish documents related to a specific project.

The programming guidelines should be used with consistency within an organization.

The methodology guidelines may vary depending on the needs.

The nomenclature set forth below is currently used by the author.

GLOBAL SPECIFICATION

This is the document for the project goals with legal-contract and links:

- The « Project Charter » describes for who, why and how exists the product.
  - Who are the project owners and users?
  - What is a simple and sketchy description of domains, problems, and goals?
  - What is the overall direction of the project?
  - What are the papers to produce?
  - What is the first estimation of means and timings?

OVERALL REALIZATION

This is the documents set for the project implementation:

- The « Application Reference » describes how is constructed and deployed the program to achieve the « Project charter ».
- The « Designer Diagram Reference » describes the organization of models and packages.
- The « Developer Data Processing Reference » describes physical schemas of classes with development help files and database tables with generation scripts as well as code algorithms and procedures-triggers.
- The « UI Reference » describes how are managed the interactions between users and computers by using keyboard, mouse, phone, etc. and screens, windows, controls, etc.

HIGH-LEVEL DESIGN FOR FUNCTIONS

This is the documents set for the project elaboration with analysis and conceptual modeling:

- The « Use cases Reference » describes stories and diagrams that describe actors and scenarios acting on activities of the domain.
- The « Communication Reference » describes how actors exist as scenarios.
- The « Activity Reference » describes the dynamic view of use cases.
- The « Sequence Reference » describes how activities exist as scenarios.

LOW-LEVEL DESIGN FOR STRUCTURES

This is the documents set for the project construction with technical and physical modeling:

- The « Deployment Reference » describes how to install the product.
- The « Component Reference » describes the combination of components.
- The « Class Reference » describes abstraction of things from the domain.
- The « Object Reference » describes how class instances exist as living entities.
- The « Collaboration Reference » in describes how objects interact.
- The « State Reference » describes the comportment of objects according to scenario.
- The « DB Reference » describes tables and schemas.

USER DOCUMENTATION

This is the documents set for the users:

- The « User Manual » is the traditional installation and usage guide.
- The « Quick Start Guide » is the conventional summary of the user manual.
- The « Troubleshooting Reference » indicates what to do if the program does not do what the user want. It includes correcting the flow of operations in case of mistakes and actions to take in case of error message or even system crash.

TIME TRACKING STAGES

Any methodology acts on height main scopes over any dichotomy and nomenclature:

- The « Management » is the time to supervise the project.
- The « Training » is the time to learn things like skills and domains.
- The « Data » is the time to study and defining things like with a database.
- The « Processing » is the time to handle things like those in a database and UI.
- The « Manual » is the time spent to give instructions to users like with a guide.
- The « Setup » is the time to deliver the application to users like with an executable.
- The « Publicity » is the time to advertise potential users like with a public message.
- The « Support » is the time to help users in difficulty like with assistance or recycling, including production bugs solving.

ECOSYSTEM

As the mutation of the classical programming to object-oriented programming has taken time to mature, since the creation of punch cards, many Agile methods are developed based on the sensitivity of their creators and depending on industrial requirements.

METHODS USED PREFERENTIALLY BY THE AUTHOR

- *Rapid Application Development* (1991): Based on an analysis-design-construction iterative cycle to build a deliverable every 3-4 months by independent teams.
- *Unified Process* (1996): Based on use cases, focused on UML architectural views, driven by iterative and incremental methods.
- *Extreme Programming* (1999): Based on the construction of the application, with very short delivery cycles, the privileged integration of the customer into the team, and the use of specific coding techniques (simplicity, refactoring, conventions, common vocabulary, unit testing, pair programming, shared code, continuous integration, respect of reality and constraints).
- *Object-oriented applications analysis and design method* (2003): UP simplification associated with RAD focused on GUI and UML to define the structures and functions of the system and to achieve an incremental application prototyping.
- *Kanban* (2010): Inspired by Lean for the process management, focused on the organization, the communication, and the knowledge.

SOME VARIOUS METHODS

- *Dynamic Systems Development Method* (1995): Structured development cycle by extension of RAD, with a higher frequency of delivery controlled by tests.
- *Feature Driven Development* (1999): Similar to RAD, with priority to features that deliver value, and the use of five activities (develop overall model, build feature list, plan by feature, design by feature, build by feature).
- *Scrum* (2001): Based on the goal and the complexity of the project goal according to the philosophy of rugby.
- *Lean Software Development* (2003): Based on eliminating waste, learning, on quality, fast return of delivery, later decision making, power given to the team, and overall vision.
- *Crystal Clear* (2004): Based on communication and collaboration, for small projects.

TOOLS

The author currently uses the following tools to work on an assembled midrange PC.

They are a selection of what he personally found actually the best for him to work.

OPERATING SYSTEM

Windows @ `www.microsoft.com/windows`

Mem Reduct @ `www.henrypp.org/product/memreduct`

Process Hacker @ `processhacker.sourceforge.io`

ClassicShell @ `www.classicshell.net`

7+ Taskbar Tweaker @ `rammichael.com/7-taskbar-tweaker`

LaunchBar Commander @ `www.donationcoder.com`

AutoHotKey @ `www.autohotkey.com`

Sizer @ `www.brianapps.net/sizer`

Ruler @ `www.spadixbd.com/freetools/jruler.htm`

Just Color Picker @ `annystudio.com/software/colorpicker`

FILE MANAGEMENT

Total Commander @ `www.ghisler.com`

WDX_GitCommander @ `github.com/Darthholi/WDX_GitCommander`

Link Shell Extension @ `schinagl.priv.at`

LockHunter @ `lockhunter.com`

MiniTool Partition Wizard @ `www.partitionwizard.com`

BACKUP MANAGEMENT

O&O DiskImage @ `www.oo-software.com`

Macrium Reflect @ `www.macrium.com`

DSynchronize @ `dimio.altervista.org/eng/dsynchronize`

FreeFileSync @ `www.freefilesync.org`

AutoVer @ `github.com/hunterbeanland/AutoVer`

SOURCE CONTROL

Git @ `git-scm.com`

GitHub @ `github.com`

Tortoise Git @ `tortoisegit.org`

TEXT EDITOR

Notepad2-mod @ xhmikosr.io/notepad2-mod

WORD PROCESSOR

Word @ www.microsoft.com/microsoft-365/word

PDF files are generated with the following options:

- *ISO 19005-1 compliant (PDF/A).*
- *Create bookmarks.*
- *Document structure tags for accessibility.*

SPREADSHEET

Excel @ www.microsoft.com/microsoft-365/excel

IMAGE PROCESSOR

XnView @ www.xnview.com

GIMP @ www.gimp.org

Axialis Icon Workshop @ www.axialis.com/iconworkshop

DIAGRAMS DESIGNER

Software Ideas Modeler @ www.softwareideas.net

NOTES ORGANIZER

ActionOutline @ www.actionoutline.com

ALARM REMINDER

1st Clock @ www.1stclock.com

TIME TRACKING

AllNetic Working Time Tracker @ www.allnetic.com

DATABASE

SQL Server @ www.microsoft.com/sql-server

Access @ www.microsoft.com/microsoft-365/access

SQLite @ www.sqlite.org

SQLite ODBC @ www.ch-werner.de/sqliteodbc

SQLite.NET @ system.data.sqlite.org

SQLite Studio @ sqlitestudio.pl

DB Browser for SQLite @ sqlitebrowser.org

DbSchema @ www.dbschema.com

INTEGRATED DEVELOPMENT ENVIRONMENT

Namo Web Editor @ www.namowebeditor.com

NuSphere PhpED @ www.nusphere.com/products/phped.htm

Visual Studio @ www.visualstudio.com

VISUAL STUDIO EXTENSIONS

GitHub Extension for Visual Studio @ visualstudio.github.com

TGit @ github.com/sboulema/TGIT

ILSpy @ github.com/icsharpcode/ILSpy

Markdown Editor @ github.com/madskristensen/MarkdownEditor

DPack @ marketplace.visualstudio.com/items?itemName=SergeyM.DPack-16348

Power Commands @ github.com/Microsoft/VS-PPT

Hot Commands @ github.com/justcla/HotCommands

Editor Guidelines @ github.com/pharring/EditorGuidelines

Editor ToolTips @ github.com/Oceanware/Tame-Visual-Studio-Code-Editor-Tooltips

Visual Studio Iconizer @ marketplace.visualstudio.com/items?itemName=...

File Icons @ github.com/madskristensen/FileIcons

Solution Error Visualizer @ github.com/Microsoft/VS-PPT

SpellCheck @ marketplace.visualstudio.com/items?itemName=...

Disable Solution Explorer's Dynamic Nodes @ marketplace.visualstudio.com/...

COMMENTS GENERATOR

Atomineer Pro Documentation @ www.atomineerutils.com

DOCUMENTATION GENERATOR

Sandcastle Help File Builder @ github.com/EWSoftware/SHFB

AGILE STORYBOARD

ZenHub @ www.zenhub.com

SETUP PACKAGER

Inno Setup Installer @ www.jrsoftware.org/isinfo.php

AUDIO AND VIDEO EDITORS

Audacity @ www.audacityteam.org

AVS Video Editor @ www.avs4you.com/AVS-Video-Editor.aspx

PROGRAMMING

PATHS

This section is undescribed yet.

NAMING

FILES

This section is undescribed yet.

NAMESPACES

This section is undescribed yet.

TYPES

This section is undescribed yet.

ENUM

This section is undescribed yet.

INTERFACE

This section is undescribed yet.

CLASS AND STRUCT

This section is undescribed yet.

VARIABLES

This section is undescribed yet.

INSTANCE

Public

This section is undescribed yet.

Private

This section is undescribed yet.

Protected

This section is undescribed yet.

LOCAL

This section is undescribed yet.

METHODS

This section is undescribed yet.

COMMENTING

FILES

This section is undescribed yet.

NAMESPACES

This section is undescribed yet.

TYPES

This section is undescribed yet.

VARIABLES

This section is undescribed yet.

INSTANCE

This section is undescribed yet.

METHOD

This section is undescribed yet.

ALGORITHMS

This section is undescribed yet.

FORMATTING

INDENTATIONS

This section is undescribed yet.

LINES

This section is undescribed yet.

BRACKETS

This section is undescribed yet.

DECLARATIONS

This section is undescribed yet.

SIGNATURES

This section is undescribed yet.

STATEMENTS

This section is undescribed yet.

ALLOCATIONS

This section is undescribed yet.

UI DESIGN

CONSOLE

This section is undescribed yet.

FORMS

This section is undescribed yet.

WEB

This section is undescribed yet.

MOBILE

This section is undescribed yet.

TV

This section is undescribed yet.

USING GIT AND GITHUB

NAMING ARTIFACTS

REPOSITORY

<project-name>

Examples: Core-Library

BRANCHES

Any combination like:

- <issue-group>/<issue-type>/<issue-item(/<summary>)(/<issue-id>)
- <issue-group-or-type>/<issue-item>(/<summary>)(/<issue-id>)
- <issue-group-or-type>/<issue-item-and-or-summary>(/<issue-id>)

Or simple expressions like "dev" or "test".

Examples:

- #32
- design/method/text/markdown/#100
- bug/install/icons-desktop/#45
- test/ui-settings

TAGS

<version-or-stage>

Examples:

- v1.2.3
- v2.0.0-rc0

COMMITS

The seven rules from `chris.beams.io/posts/git-commit`:

- *Separate subject from body with a blank line.*
- *Limit the subject line to 50 characters.*
- *Capitalize the subject line.*
- *Do not end the subject line with a period.*
- *Use the imperative mood in the subject line.*
- *Wrap the body at 72 characters.*
- *Use the body to explain what and why vs how.*

Common commits actions are:

- Add, Rename, Remove, Delete.
- Set, Update, Change, Improve, Fix, Move.
- Generate, Clean, Refactor, Rework.
- Initial commit, Merge, Release.

A domain can be specified by using an issue-item token:

```
ui: Fix the main form size
db: Add a script to create a table
manual: Update thefile.html
```

A commit name can be a date to distinguish an important prototype during the development phase without using a name or tag.

MILESTONES

Milestones allow identifying project big steps as agility and UP process: Inception, Elaboration, Construction and Transition.

For simple or non-software projects such as this guide milestones can be:

- *Version 1*
- *Version 2*

ISSUE LABELS

EPIC

ZenHub allows using special stories called Epic to gather other stories.

Color is Dark Blue #3E4B9E.

GROUP

Group defines the area concerned by the issue.

Color is Teal #006B75.

```
group: project (management)
group: training (learning)
group: analysis (requirements gathering)
group: design (modeling)
group: code (implementation)
group: manual (documentation and guide)
group: deploy (setup and migration)
group: user (assistance and communication)
```

TYPE

Type defines the gender of the issue.

Color is Green #0E8A16.

```
type: legal (license)
type: layout (organization and planning)
type: method (guideline)
type: admin (supervision)
type: feature (functionality)
type: improve (extend feature)
type: check (test, revision and validation)
type: bug (error)
type: feedback (reaction)
```

ITEM

Item defines the thing affected by the issue.

Color is Blue #1D76DB.

```
item: app (product and executable)
item: diagram (representation)
item: data (information)
item: source (code file)
item: install (packager)
item: text (writing)
item: tool (third party software)
item: ui (user interface)
item: ux (user experience)
item: other
```

PRIORITY

There is no medium priority since it is a loss of time to set and read it.

Thus, it is easy to see the cards with low or high priority and others are ordinary.

```
prio: critical [Dark Red #900000]
prio: high [Red #CA2525]
prio: low [Dark Cyan #BFDADC]
```

IN PROGRESS

In progress defines an issue being solved and it is used in conjunction with some State label.

Color is Yellow #FFD700.

STATE

State indicates the progress of the work not towards the time but the remaining tasks.

Six points of a Gaussian curve are used to estimate the `In progress` pipeline.

This percentage is not about time because the tens first and last parts are generally longer while at the middle the things can be fast:

- *When the task starts there is no really competence and no good visibility.*
  It has not started because it is taking its place to run on the racetrack.
  Things often seem to be simple and easy even for big task.
  It is not uncommon to spend a quarter of the time on this inception phase.
- *When the task comes to its end, there is a need to begin checking that all is really fine.*
  It is not running anymore because and it is shutting down on the racetrack.
  Things are more complex and more interactive even they look effortless.
  This transition phase can sometimes be more half the time.

This percentage may be reevaluated according to addition or cancellation of the complexity.

Color is Yellow #FFD700.

```
state: todo (selected) [Pale Green #C2E0C6]
state: delayed (deferred) [Gray #CACACA]
state: cancelled (abandoned) [Light Gray #EAEAEA]
state: moved (to another project) [Light Gray #EAEAEA]
state: wontfix (failed) [Dark Gray #707070]
state: 10% (work started)
state: 25%
state: 50%
state: 75%
state: 90% (almost completed)
state: 100% (done) [Light Yellow #FFF3B5]
```

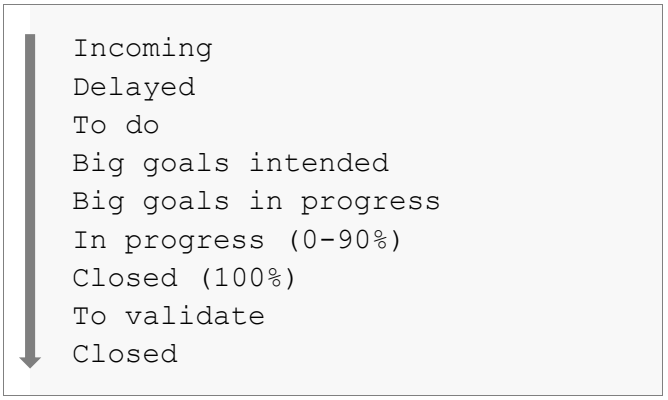
ZENHUB BOARDING

One GitHub project can be used as a storyboard for one or more use case diagrams.

While this not allows advanced management yet, the author uses ZenHub and Epics.

PIPELINES

ZenHub Pipelines allow setting the stage of issues like on a Kanban workflow board.



On small projects Incoming, Icebox, Backlog, Done and Review/QA can be omitted, and Epics planned, and live pipelines can be none or one while progress labels indicates the difference.

Epics allow distinguishing Featured User Stories from Action User Stories.

Visitors of the project's page that are not connected to an account extended with ZenHub can't see this layering yet and only `in progress` label and `closed` issues can be used to distinguish them from other, so manual labels must be used in addition to moving cards.

ISSUE AS USER STORY

An Issue is used as a user story by indicating its description containing tasks in checklist.

```
[Issue #1] Prepare the repository

           As a developer,
           I want to establish the repository,
           so I can construct the software.

           ☐ Create the repository
           ☐ Setup the repository
           ☐ Specify the license
```

ISSUE ESTIMATE

Estimate field is used to define the issue complexity from `1` to `5` or to `10` for example, by considering knowledge, competence, technicity, and range required by the issue.

Epic Issue estimation is usually not done because it is finished when all linked issues are finished and this value can be viewed on Issue details in the panel added by ZenHub and the Epic points divided by the issues count rounded to the upper is thus used as an estimate.

Estimate time is out of scope of this document and falls under any appropriate methodology, but timings can be defined and adjusted using the burndown chart as the project progress.

ISSUES HIERARCHY

Visitors of the project's page that are not connected to an account extended with ZenHub cannot see this design yet without check-listing sub-issues in the description.

FEATURED USER STORY AS A HIGH-GOAL THAT ENCAPSULATES LOW-GOALS

An Issue is used as a complex story containing references to other issues by using ZenHub Epic label.

It should contain a checklist of all sub-issues as high tasks.

```
[Epic Issue #1]  Prepare the repository

                 ☐ Create the repository #2
                 ☐ Setup the repository #3
                 ☐ Specify the license #4
```

ACTION USER STORY AS LOW-GOAL TO ACHIEVE HIGH-GOAL

An Issue is used as a simple story acting as a card of what a user wants by attaching it to an Epic Issue.

```
[Issue #2] Create the repository

           ☐ Add a repository
           ☐ Create a first branch
```

```
 [Issue #3] Setup the repository

           ☐ Define Milestones
           ☐ Define Labels
```

```
[Issue #4] Specify the license

          ☐ Examine available licenses
          ☐ Choose a license
          ☐ Publish the LICENSE file
```

APPENDICES

CONSIDERATIONS ABOUT THE <INTERFACE> KEYWORD

Article written on July 2009 and named « Are interfaces evil or misused? ».

THERE ARE SEVERAL CONSIDERATIONS AND PRACTICES CONCERNING INTERFACES

Some developers say that interfaces can be used as a replacement of multiple inheritance mechanisms, which cause complexity and ambiguity. But each feature must be implemented each time it is declared: this is not an inheritance; this is a wrapper to the description of a part of a group of classes, like `IDisposable`. It is the same as a multiple inheritance with one implemented class and some abstract classes: it is a special case which allows only one-way hierarchy with interfaces as abstract connectors that describes services.

Some developers say that interfaces can be used to separate the access to an object of this instance. Historically, interfaces are a COM & DCOM heritage: they are used to manipulate components services, whatever objects are, where they are, and how they are implemented. Interfaces are not a replacement to multiple inheritance; they are something else.

Article « A plea for full multiple inheritance support in .NET »

`weblogs.asp.net/fbouma/archive/2004/01/04/47476.aspx`

Article « A Typed Intermediate Language for Compiling Multiple Inheritance »

`research.microsoft.com/apps/pubs/default.aspx?id=55775`

INTERFACES CAN BE USED TO SOLVE SOME DESIGN ISSUES IN .NET

In component communication, because they are like a transfer of addresses: « An interface describes a group of related functionalities that can belong to any class or structure. Interfaces can consist of methods, properties, events, indexers, or any combination of those four member types. An interface cannot contain fields. Interfaces members are automatically public » (MSDN Documentation). To simplify, in distributed computing, a client has an "instance" of an interface to an object that is on the server.

In the conceptualization of services provided by classes instead of multiple inheritance: this abstraction layer should be separated from the classes layer. It is not an implementation because it is a high-level design view and this should not be coded: a method should not be implemented several times in the same way, which is facilitated by the genericity. An interface corresponds to a fully implemented class, to a class that contains abstract members, or to a fully/pure abstract class.

To provide a kind of low level polymorphism for generic classes to manipulate all possible linked templates while this feature doesn't exist in C#: an undetermined type like `GenericClass<>` can't be used without providing a specific type unless reflexion is used.

But they create some problems

All functionalities must be implemented each time.

The type dependence is fragmented to the detriment of quality.

The code complexity is increased: lots of declarations and big code for one class instead of lots of small classes with small code.

Using interfaces in implementation increases the abstraction of an abstraction, reduces the code factorization, reduces the maintainability, and increases the risks of a project.

MULTIPLE INHERITANCE MAY BE USED TO SOLVE SOME DESIGN PROBLEMS IN .NET

To inherit from multiple classes and to implement only once a service provided by several classes.

To have a strongly typed design without seeing double or going crazy.

To provide a high level of polymorphism.

But it creates some problems

All functionalities must be implemented rigorously.

It requires a study and an understanding of the object programming theory.

The code complexity is increased: lots of small classes with small code instead of lots of declarations and big code for one class.

Using multiple inheritance increases the simplicity of the abstraction (of interfaces, if models are based on), reduces the code size, reduces the confusion about models and increases the safeness of a project.

INTERFACES DON'T COMPENSATE FOR THE LACK IN MULTIPLE INHERITANCE AS WELL AS IN GENERIC POLYMORPHISM

Both have their difficulties, their advantages, and their applications: discuss the pros and cons of interfaces is a wrong debate without end.

Code based on interfaces is a default programming and an entangled path based on a difficult simplification of what objects and components are. Interfaces are a full or a partial copy of a class description: they do not provide better software experience but better ways to design. The mechanisms of interfacing in distributed environment should be provided by the CLR and based on classes descriptions which are the interfaces: the virtual tables.

Of course, to not use interface and multiple inheritance reduces the code complexity, and a developer should have the choice depending of the work.

Here is an example using single inheritance and interfaces.

```
public delegate void ConfigureEvent(IConfigurable value);

public interface IConfigurable
{
  ConfigureEvent DoConfiguration { get; set; }
  void Configure();
}

public class ClassA
{
}
public class ClassB : IConfigurable
{
  ConfigureEvent IConfigurable.DoConfiguration { get; set; }
  void IConfigurable.Configure()
  {
    if ( ( (IConfigurable)this ).DoConfiguration != null )
    ( (IConfigurable)this ).DoConfiguration(this);
  }
}

public class ClassA1 : ClassA
{
}

public class ClassA2 : ClassA, IConfigurable
{
  ConfigureEvent IConfigurable.DoConfiguration { get; set; }
  void IConfigurable.Configure()
  {
    if ( ( (IConfigurable)this ).DoConfiguration != null )
    ( (IConfigurable)this ).DoConfiguration(this);
  }
}

public class ClassA3 : ClassA1, IConfigurable
{
  ConfigureEvent IConfigurable.DoConfiguration { get; set; }
  void IConfigurable.Configure()
  {
    if ( ( (IConfigurable)this ).DoConfiguration != null )
    ( (IConfigurable)this ).DoConfiguration(this);
  }
}
```

The equivalent of these 45 lines in multiple inheritance is simpler and more intuitive.

It takes only 30 lines and the method is implemented only once.

There is no interface because the definition of the class is its interface and code is better.

```
public delegate void ConfigureEvent(IConfigurable value);

public class Configurable
{
  public ConfigureEvent DoConfiguration { get; set; }
  virtual public void Configure()
  {
    if ( DoConfiguration != null ) DoConfiguration(this);
  }
}

public class ClassA
{
}

public class ClassB : Configurable
{
}

public class ClassA1 : ClassA
{
}

public class ClassA2 : ClassA, Configurable
{
}

public class ClassA3 : ClassA1, Configurable
{
}
```

Supporting generic polymorphism and multiple inheritance in .NET and C# could allow making programs more stable and efficient, as well as coding like this.

```
public class Class1
{
  public int Class1Value { get; set; }
}

public class Class2
{
  public int Class2Value { get; set; }
}

public class Class3 : Class1
{
  public int Class3Value { get; set; }
}

public class Class4<T> : Class3, Class2
{
  public int Class4Value { get; set; }
  public T Value { get; set; }
}

public class Class5<T> : Class4<T>
{
  public int Class5Value { get; set; }
  public void DoSomething() { }
}

void TestPolymorphism()
{
  var list = new List < class3 >();
  list.Add(new Class3());
  list.Add(new Class4<int>());
  list.Add(new Class5<int>());
  list.Add(new Class5<string>());
  foreach (var o in list)
  {
    o.Class1Value = 0;
    o.Class3Value = 0;
    if (o is Class2) (o as Class2).Class2Value = 0;
    if (o is Class4<int>) ) (o as Class4<int>).Value = 0;
    // Generic polymorphism and diamond operator
    // is not currently available in C#
    if (o is Class5<>) (o as Class5<>).DoSomething();
  }
}
```

It is impossible to write the last line and the problem is solved by using interfaces.

```
public interface IClass1
{
  int Class1Value { get; set; }
}

public interface IClass2
{
  int Class2Value { get; set; }
}

public class Class1 : IClass1
{
  public int Class1Value { get; set; }
}

public class Class2 : IClass2
{
  public int Class2Value { get; set; }
}

public class Class3 : Class1
{
  public int Class3Value { get; set; }
}

public class Class4<T> : Class3, IClass2
{
  public int Class2Value { get; set; }
  public int Class4Value { get; set; }
  public T Value { get; set; }
}

interface IClass5
{
  int Class5Value { get; set; }
  void DoSomething();
}

public class Class5<T> : Class4<T>, IClass5
{
  public int Class5Value { get; set; }
  public void DoSomething() { }
}
```

```
void TestPolymorphism()
{
  var list = new List<class3>();
  list.Add(new Class3());
  list.Add(new Class4<int>());
  list.Add(new Class5<int>());
  list.Add(new Class5<string>());
  foreach (var o in list)
  {
    o.Class1Value = 0;
    o.Class3Value = 0;
    if (o is IClass2) (o as IClass2).Class2Value = 0;
    if (o is Class4<int>) ) (o as Class4<int>).Value = 0;
    if (o is IClass5) (o as IClass5).DoSomething();
  }
}
```

But it is less elegant, less intuitive, less robust, and less secure.

CONSIDERATIONS ABOUT THE <SINGLETON> KEYWORD

Article written on July 2009 and named « Design flaws of the singleton pattern ».

THE PARADIGM

Here is the common singleton pattern implementation:

```
public class Singleton
{
  static private readonly object locker = new object();
  static public Singleton Instance
  {
    get
    {
      lock ( locker )
      {
        if ( _Instance == null )
          _Instance = new Singleton();
        return _Instance;
      }
    }
  }
  static private volatile Singleton _Instance;
  private Singleton()
  {
  }
}
```

The problem is that you can inherit this class and create a public constructor if there is no private constructor. Furthermore, static members are allowed. This is no longer a singleton at all. Setting the class as sealed can be an acceptable solution, but you must implement singleton by singleton, i.e., more than ten lines. Thus, coding such singleton can be the source of many errors and difficulties. Thinking with factoring is not only an agile principle, it is a mathematical theorem.

DEFINING A GENERIC SINGLETON

A generic solution is to check the absence of static members and that there is only one parameter less private constructor, or an exception is thrown. Singletons that inherit this class cannot be inherited and must be sealed. Moreover, the implementation of singleton types is checked at program startup. Therefore, it is not the best solution, but the only thing to do is to create one parameter less private constructor, no static members, and seal the class.

A sample is available online:

`www.ordisoftware.com/download/GenericPersistentSingleton.zip`

Here are members of the proposed singleton:

```
abstract public class Singleton<T> where T : Singleton;
```

This is the declaration of a generic abstract class where T is a singleton.

By writing this, the type consistency is clear.

```
static public string Filename;
static public void Save();
```

It is used to provide storage on disk for persistent singletons and to save their states.

```
static public T Instance;
static public T GetInstance();
```

This is the classic access to the instance of the singleton.

```
static public T GetPersistentInstance(string filename);
static public T GetPersistentInstance();
```

It creates a persistent instance: it deserializes the object from the disk or create a new. It uses a specific filename or a system name.

Defining the name after using the singleton does not load a new instance and should throw an error if the localization exists.

```
static private T CreateInstance();
static internal ConstructorInfo CheckImplementation();
```

This creates the instance by invoking the default private constructor.

The singleton implementation validity is checked like indicated above.

Here are Serialize and Deserialize functions:

```
static public void Serialize(this object obj, string filename)
{
  if ( !obj.GetType().IsSerializable )
    throw new IOException(
      SystemManager.Language.Get("ObjectIsNotSerializable",
                                 obj.GetType().Name));
  using ( FileStream f = new FileStream(filename,
                                        FileMode.Create,
                                        FileAccess.Write,
                                        FileShare.None) )
    new BinaryFormatter().Serialize(f, obj);
}

static public object Deserialize(this string filename)
{
  if ( !File.Exists(filename) )
    throw new IOException(
      SystemManager.Language.Get("FileNotFound",
                                 filename));
  using ( FileStream f = new FileStream(filename,
                                        FileMode.Open,
                                        FileAccess.Read,
                                        FileShare.None) )
    return new BinaryFormatter().Deserialize(f);
}
```

CODING THE SINGLETON

```
namespace Ordisoftware.Core.ObjectModel
{

  [Serializable]
  abstract public class Singleton <T> where T : Singleton
  {

    static private readonly object locker = new object();

    static protected void DoError(string s)
    {
      throw new SingletonException(SystemManager.Language.Get(s),
                                   typeof(T));
    }

    static public string Filename
    {
      get { return _Filename; }
      set
      {
        if ( _Filename == value ) return;
        lock ( locker )
        {
          if ( FileTool.Exists(_Filename) )
            FileTool.Move(_Filename, value);
          _Filename = value;
        }
      }
    }
    static private volatile string _Filename;

    static public void Save()
    {
      lock ( locker )
        if ( !( _Filename.IsNullOrEmpty() && Instance.IsNull() ) )
        {
          FolderTool.Check(_Filename);
          Instance.Serialize(_Filename);
        }
    }

    ~Singleton()
    {
      try { Save(); }
      catch (Exception e) { ShowError(e.Message); }
    }
```

```
static public T Instance
{
  get
  {
    lock ( locker )
    {
      if ( _Instance == null )
        if ( FileTool.Exists(_Filename) )
          _Instance = (T)_Filename.Deserialize();
        else
          _Instance = CreateInstance();
      return _Instance;
    }
  }
}
static private volatile T _Instance;

static public T GetInstance()
{
    return Instance;
}

static public T GetPersistentInstance(string filename)
{
  Filename = filename;
  return Instance;
}

static public T GetPersistentInstance()
{
  if ( _Instance != null ) return _Instance;
  Type type = typeof(T);
  string s = type.Namespace + '.' + type.Name.Replace('`', '_');
  foreach ( Type t in type.GetGenericArguments() )
    s += " " + t.FullName;
  s = SystemManager.FolderSystem + s + SystemManager.ExtObjectFile;
  return GetPersistentInstance(s);
}

static private T CreateInstance()
{
  return (T)CheckImplementation().Invoke(null);
}
```

```
    static internal ConstructorInfo CheckImplementation()
    {
      Type type = typeof(T);
      if ( !type.IsSealed ) DoError("SingletonMustBeSealed");
      var bf1 = BindingFlags.Static
              | BindingFlags.NonPublic
              | BindingFlags.Public;
      var bf2 = BindingFlags.Instance
              | BindingFlags.Public;
      var bf3 = BindingFlags.Instance
              | BindingFlags.NonPublic;
      if ( type.GetMembers(bf1).Length != 0 )
        DoError("SingletonNoStaticMembers");
      if ( type.GetConstructors(bf2).Length != 0 )
        DoError("SingletonNoPublicConstructors");
      ConstructorInfo[] list = type.GetConstructors(bf3);
      if ( ( list.Length != 1 )
        || ( list[0].GetParameters().Length != 0 )
        || ( !list[0].IsPrivate ) )
        DoError("SingletonOnlyOnePrivateConstructor");
      return l[0];
    }
  }
}
```

STARTUP CHECKING

```
namespace Ordisoftware.Core
{
  static public class SystemManager
  {
    static public void Initialize()
    {
      Type type = Type.GetType("Ordisoftware.Core.ObjectModel.Singleton`1");
      if ( type != null )
      {
        MethodInfo method;
        string name = "CheckImplementation";
        var bf = BindingFlags.InvokeMethod
               | BindingFlags.FlattenHierarchy
               | BindingFlags.Static
               | BindingFlags.NonPublic;
        var list = ObjectUtility.GetClasses(
          t => ( t.BaseType.Name == type.Name )
            && ( t.BaseType.Namespace == type.Namespace ));
```

```
        foreach ( var t in list )
          try
          {
            if ( !t.ContainsGenericParameters )
              method = t.GetMethod(name, bf);
            else
            {
              Type[] p = t.GetGenericArguments();
              for ( int i = 0; i < p.Length; i++ )
                p[i] = typeof(object);
              method = t.MakeGenericType(p).GetMethod(name, bf);
            }
            method.Invoke(null, new object[0]);
          }
          catch ( Exception e ) { ShowException(e); }
      }
    }
  }
}
```

Here is the GetClasses function.

```
static public TypeList GetClasses(Func select)
{
  return GetList(t => t.IsClass, select);
}

static private TypeList GetList(Func check, Func select)
{
  TypeList list = new TypeList();
  Type[] l1 = Assembly.GetExecutingAssembly().GetTypes();
  if ( select == null )
    list.AddRange(l1);
  else
    foreach ( Type t in l1 )
      if ( check(t) && select(t) ) list.Add(t);
  Module[] l2 = Assembly.GetEntryAssembly().GetLoadedModules();
  if ( select == null )
    list.AddRange(l1);
  else
    foreach ( Module m in l2 )
      foreach ( Type t in m.Assembly.GetTypes() )
        if ( check(t) && select(t) ) list.Add(t);
  list.Sort((v1, v2) => v1.FullName.CompareTo(v2.FullName));
  return list;
}
```

EXAMPLE OF USAGE

Each execution adds 10 to the value displayed by this program:

```
[Serializable]
public class MySingleton : Singleton < MySingleton >
{
  public int Value { get; set; }
  private MySingleton() { }
}

static class Program
{
  [STAThread]
  static void Main(string[] args)
  {
    SystemManager.Initialize();
    try
    {
      var v = MySingleton.GetPersistentInstance();
      v.Value += 10;
      Console.WriteLine("MySingleton.Value = " +
                        MySingleton.Instance.Value);
    }
    catch ( Exception e )
    {
      Debugger.ManageException(null, e);
    }
    finally
    {
      SystemManager.Finalize();
    }
  }
}
```

THE MISSING "SINGLETON" LANGUAGE KEYWORD

The best way to implement a singleton in C# is to create a static class, but this may cause a problem with serialization and with when the object is initialized, whether one considers laziness.

The ideal thing would be to have a language keyword like singleton: an artifact having no static members and only one constructor with no parameter and no access modifier. It can be inherited only if marked as abstract. It may be used like a static class but will act like an instantiated class. It may be serializable and disposable: the first usage deserializes the object if a stream is associated or creates a new single instance, disposing serializes the singleton or does nothing if no stream is associated, changing the stream moves the instance from the old to the new place, and setting a stream on a singleton already instantiated causes a usage exception if the new stream localizes an item that exists.

```
[Serializable]
[SingletonPersistence(false)]  // don't use a default system stream
public singleton MySingleton
{
  public int Value {get; set; }
  MySingleton()
  {
    // Code executed on first access
  }
}
var stream1 = new SingletonFileStream("c:mysingleton.bin");
var stream2 = new SingletonSystemStream();
MySingleton.SetStream(stream1);
MySingleton.Value += 10;
MySingleton.SetStream(stream2);
MySingleton.Value += 10;
MySingleton.SaveState();
```

RECOMMENDED ARTICLES

Implementing the Singleton Pattern in C#

www.yoda.arachsys.com/csharp/singleton.html

Fun with Singletons in C# 2.0

www.codeproject.com/KB/cs/FunWithSingletonsCS.aspx

Generic Singleton Pattern using Reflection in C#

www.codeproject.com/KB/architecture/GenericSingletonPattern.aspx

Lazy Vs Eager Init Singletons / Double-Check Lock Pattern

geekswithblogs.net/akraus1/articles/90803.aspx

The quest for the Generic singleton in C#

www.c-sharpcorner.com/UploadFile/snorrebaard/
GenericSingleton11172008110419AM/GenericSingleton.aspx

BIBLIOGRAPHIE

THOMSON COMPUTERS

Manuel technique du TO7-70 (Cedic/Nathan 1984)
Initiation et Référence du Basic TO7-70 (Cedic/Nathan 1984)
Initiation et Référence du Logo TO7-70 (Cedic/Nathan 1984)
Manuel de l'Assembleur 6809 du TO7-70 (Cedic/Nathan 1984)
Le Basic Q-D.O.S. du TO7-70 (Cedic/Nathan 1985)
Guide du TO9 (Cedic/Nathan 1985)
Guide du Basic TO9 (Cedic/Nathan 1985)
Super jeux MO5 et TO7-70 (Jean-François Sehan - PSI 1985)
Pratique du TO7-70, programmation niveau 1 (Henri Lilen - Editions Radio 1984)
Pratique du TO7-70, programmation niveau 2 (Henri Lilen - Editions Radio 1984)
50 programmes assembleur T07-70 (B. Geoffrion, R. Weiss - Editions Radio 1985)

PC MICROPROCESSORS AND SYSTEMS

8088 Assembleur (Henri Lilen - Editions Radio 1986)
8088 et ses périphériques (Henri Lilen - Editions Radio 1986)
Cours pratique de logique pour microprocesseur (Henri Lilen - Editions Radio 1986)
Cours fondamental des microprocesseurs (Henri Lilen - Editions Radio 1987)
Microprocesseurs (Henri Lilen - Dunod 1995)
PC system programming for developers (Michaël Tischer, Data Becker 1989)
PC Interdit (Bertelsons, Rasch, Hoff - Micro Application 1995)
Mastering Turbo Assembler (Tom Swan - Sams 1995)
80386 Programmer's Reference Manual (Intel 1986)
80486 Programmer's Reference Manual (Intel 1990)
x86 Developer’s Manual Volume 1: Basic Architecture (Intel 1997)
x86 Developer’s Manual Volume 2: Instruction Set Reference (Intel 1997)
x86 Developer’s Manual Volume 3: System Programming Guide (Intel 1997)
Protected mode software architecture (Tom Shanley - Addison Wesley 1996)
Operating System concepts (Silberschatz Galvin - Addison Wesley 1998)
Programmation des API Win32 (Simon, Gouker, Barnes - S&SM 1998)
VBScript Programmer's Reference (A. & K. Kingsley-Hughes, D. Read - Wrox 2007)
Professional Windows PowerShell (Andrew Watt - Wrox 2007)

BORLAND IDES

Turbo Basic (Borland 1988)
Turbo Pascal (Borland 1988)
Turbo C++ (Borland 1989)
Turbo Prolog (Borland 1989)
Delphi 5 Objet Pascal Language Guide (Borland 1999)
Delphi 5 Developer's Guide (Borland 1999)
Mastering Delphi 2 (Marco Cantu - Sybex 1996)

C AND C++ LANGUAGES

Guide SOS du Turbo C (Jörg Schieb - Micro Application 1990)
Le langage C++ (Livret de cours IUT Informatique 1996-1997)
Visual C++ 5 (Christian Fleischhauer - Micro Application 1997)
C++ Primer Plus (The waite group's - Sams 1998)
Effective C++ (Scott Meyers - Addison Wesley 1996)
More effective C++ (Scott Meyers - Addison Wesley 1997)
The design and evolution of C++ (Bjarne Stroustrup - Addison Wesley 1994)

C# AND .NET FRAMEWORK

The C# language (Hejlsberg, Wiltamuth, Golde - Addison Wesley 2004)
Component-based development with Visual C# (Ted Faison - M&T Books 2002)
Professional .NET framework (Collective - Wrox 2001)
Beginning Visual C# 2008 (Collective - Wrox 2008)
Beginning C# 2005 Databases (Karli Watson - Wrox 2006)
Professional C# 2008 (Collective - Wrox 2008)
Professional C# 7 and .Net Core 2.0 (Christian Nagel - Wrox 2018)
Professional Visual Studio 2017 (Bruce Johnson - Wrox 2017)
Professional .NET 2.0 Generics (Tod Golding - Wrox 2005)
Professional ADO.NET 3.5 with LINQ and Entity Framework (Roger Jennings - Wrox 2009)
LINQ Pocket Reference (Joseph Albahari - O'Reilly 2008)
LINQ Unleashed for C# (Paul Kimmel - SAMS 2008)
Code-First Development with Entity Framework (Sergey Barskiy - Packt Publishing 2015)
Beginning ASP.NET for Visual Studio 2015 (William Penberthy - Wrox 2016)
Professional WCF Programming (Scott Klein - Wrox 2007)
Professional Visual Studio Extensibility (Keyvan Nayyeri - Wrox 2008)
Professional Refactoring in C# & ASP.NET (Danijel Arsenovski - Wrox 2009)
Professional Test Driven Development with C# (J. Bender, J. McWherter - Wrox 2011)
C# Design and Development Expert One on One (John Paul Mueller - Wrox 2009)
C# 3.0 Cookbook (Jay Hilyard - O'Reilley 2008)
C# 4.0 How-To (Ben Watson - Sams 2010)
Effective C# (Bill Wagner - Addison Wesley 2016)
More Effective C# (Bill Wagner - Addison Wesley 2017)
C# 3.0 Design Patterns (Judith Bishop - O'Reilley 2008)
Design Patterns pour C# (Laurent Debrauwer - ENI Editions 2011)
Agile principles, patterns and practices in C# (R. C. & M. Martin - Prentice Hall 2006)

JAVA AND WEB

Pure Java 2 (Kenneth Litwak - Sams 1999)
Using XHTML, XML & Java 2 (Eric Ladd, Jim O'Donnell - Que 1999)
PHP4 & MySql (G.A. Leierer, R. Stoll - Micro Application 2000)
Beginning HTML and CSS (Rob Larsen - Wrox 2013)
Professional WordPress (Hal Stern, David Damstra, Brad Williams - Wrox 2010)

Professional WordPress Plugin (Brad Williams, Ozh Richard, Justin Tadlock - Wrox 2011)
JavaScript 24–Hour Trainer (Jeremy McPeak - Wrox 2010)
Beginning JavaScript (Jeremy McPeak - Wrox 2015)
Professional Javascript for Web Developers (Matt Frisbie - Wrox 2019)

DATABASES AND SQL

Oracle et SQL (Livret de cours IUT Informatique 1996-1997)
Oracle 8 (Roger Chapuis - Dunes-Laser 1998)
The SQL Guide to SQLite (Rick F. van der Lans - Lulu 2009)
Access 2010 24–Hour Trainer (Geoffrey L. Griffith, Truitt L. Bradly - Wrox 2011)
Professional Access 2013 Programming (Collective - Wrox 2013)
Access 2010 Programmer's Reference (Collective - Wrox 2010)
Beginning SQL Server 2008 Programming (Robert Vieira - Wrox 2006)
Beginning Database Design Solutions (Rod Stephens - Wrox 2008)
GitHub Essentials (Achilleas Pipinellis - Packt Publishing Limited 2015)
Professional Git (Brent Laster - Wrox 2016)

ALGORITHMIC AND ARTIFICIAL INTELLIGENCE

Programmation structurée en BASIC (Francis Crochet - Editions Radio 1987)
Programmation récursive (Livret de cours IUT Informatique 1996-1997)
Programmation des listes chainées (Livret de cours IUT Informatique 1996-1997)
Programmation des graphes (Livret de cours IUT Informatique 1996-1997)
Mathématiques pour l'informatique (Collective - Dunod 2008)
Logic Programming with Prolog (M.A. Bramer - Springer London Ltd 2005)
L'Intelligence Artificielle pour les développeurs C# (Virginie Mathivet - Eni Editions 2017)
Essential Algorithms (Rod Stephens - Wiley 2019)

SOFTWARE DEVELOPMENT

Le génie logiciel (Livret de cours IUT Informatique 1996-1997)
The UML User Guide (Booch, Rumbaugh, Jacobson - Addison-Wesley 2001)
UML 2.5 (Laurent Debrauwer, Fien Van Der Heyde - Eni Editions 2016)
Code Leader (Patrick Cauldwell - Wrox 2008)
Beginning Software Engineering (Rod Stephens - Wrox 2015)
Patterns, Principles, and Practices of Domain-Driven Design (Millett, Tune - Wrox 2015)
Méthode orientée-objet intégrale MACAO (Jean-Bernard Crampes - Ellipses 2003)
Kanban (David J. Anderson - Blue Hole Press 2010)
Extreme Programming (Chromatic - O'Reilly 2005)
Practices of an Agile Developer (Subramaniam & Hunt - Pragmatic Bookshelf 2006)
Software Craftsmanship (Sallah Kokaina - ENI Editions 2019)
The Software Craftsman (Sandro Mancuso - Prentice Hall 2014)
The clean coder (Robert C. Martin - Prentice Hall 2011)
Clean code (Robert C. Martin - Prentice Hall 2008)
Clean architecture (Robert C. Martin - Prentice Hall 2017)
Code complete (Steve McConnell - Microsoft Press 2004)

UX design et ergonomie des interfaces (Jean-François Nogier - Dunod 2020)
Producing Open Source Software (Karl Fogel - O'Reilly Media 2005)

ONLINE COURSE (FRENCH)

Les nouveautés du C# 3 (Microsoft TechDays)
C# Linq (Microsoft TechDays)
Programmation fonctionnelle F# (Microsoft TechDays)
Plongée au cœur du Framework .NET 4.5 (Microsoft TechDays 2013)
C# - Enigmes et puzzles (Microsoft TechDays 2013)
Git - La gestion des versions de vos projets (ENI Editions)
Kanban - Améliorer et fluidifier la coordination d'un projet (ENI Editions)
Méthodes Agiles - Mettre de l'agilité dans la gestion de vos projets (ENI Editions)
C# 7 et Entity Framework - Maîtrisez l'accès aux données (ENI Editions)
SQL - Les fondamentaux du langage (ENI Editions)
SQL Server 2014 - Conception et réalisation d'une base de données (ENI Editions)
ASP.NET MVC5 - Développez des applications Web (ENI Editions)
Python 3 - Les fondamentaux du langage (ENI Editions)
Expressions régulières - Optimisez vos recherches de motifs textuels (ENI Editions)
Windows PowerShell - Les fondamentaux du langage (ENI Editions)
Gestion de tests logiciels (ENI Editions)
Réseaux informatiques - Les bases essentielles (ENI Editions)
Réseaux sociaux - Facebook, Twitter, LinkedIn et Viadeo (ENI Editions)
Social Media - Le webmarketing par les réseaux sociaux (ENI Editions)
Machine Learning (ENI Editions)

OTHER TOPICS

Tennis (Nathan 1986)
Eat to win (Robert Haas - Scribner 1985)
Blood and guts (Dorian Yates - 1993)
Relativity (Albert Einstein - Digireads 1990)
What is life (Erwin Schrödinger - Cambridge University Press 1992)
Light and Matter (Richard Feynman - Princeton University Press - Seuil 1985)
Histologie (Jacques Poirier, Jean-Louis Ribadeau Dumas - Masson Abrégé 1988)
Cours de biologie cellulaire (Pierre Cau, Raymond Seïte - Ellipses 2009)
Dictionnaire de la Bible Hébraïque (Marchand Ennery - Colbo 1996)
Les bases de l'harmonie (Philippe Ganter - Dareios IDMusic 2007)
Neuromarketing (Patrick Renvoisé, Christophe Morin - SalesBrain Publishing 2005)
The Mental Game of Poker (Jared Tendler - 2011)