D3.3 Common Guidelines for software development v1.0

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 842009

Table of contents

1. Why I should care about these guidelines

If you're reading these guidelines, it's most likely that you're involved in some step related to the creation or combination of re-usable innovative software components which includes a digital interface, be it machine-to-machine or user-to-machine, in the context of the pan-European NIVA project on the New IACS Vision in Action.

These tools and components are likely to innovate the processes and services used by European CAP Paying Agencies to manage and control agricultural funds. Ideally, the NIVA tools should be easily reusable in all EU Member States. In practice, such deliverables will be successful if they will be used in at least two Paying Agencies that will run them in different environments, not only technically but also geographically, and if they prove a tangible benefit to the business and the community.

They will be successful also if re-used by other companies to complement existing software or systems, which means that the quality of results will be high and that a high level of technical interoperability shall be ensured.

Knowing and following the guidelines will help re-usability and ensure that teams working for other clients can benefit from your results. Your investment in time and effort will allow to possibly have your innovative idea implemented in many of the EU countries.

2. Purpose and Target

These guidelines are for non-trivial software that will need to be maintained, re-used and developed further in the NIVA context and beyond. They also cover for some pseudo-software deliverables that could be produced in the form of 'non-software' algorithms, e.g. technical processes to automatically achieve a result.

They are here to ensure that every Single Member State Pilot (Single MS Pilot) will be designed to migrate as easily as possible to a Multi MS Pilot and, in general, that every component built in the NIVA context follows the same principle.

The guidelines are not intended to substitute the certifications that the NIVA Partners or contributing parties may already have on the design and development of software solutions (e.g. ISO9001 or equivalent), rather to complement them when possible. They are intended to reach a sufficient level of software quality. Software quality is difficult to define, but the developer's community agree that high quality code 'is readable and well documented', 'is reusable and without duplications', 'handles errors', 'is efficient', 'includes unit tests' and 'complies with security'.

Because NIVA is a collaborative project, adhering to consistent choices, like formatting and naming conventions, is especially important. Software development conventions are important in any multi-developer project and help developers read and understand each other's project, code or to clearly understand the intent. Please try to follow these guidelines in spirit. Your cooperation will improve the effectiveness of the collaboration.

The purpose of this document is therefore to help NIVA project managers, software architects, software analysts and software developers to achieve a more uniform style across design, code bases and interfaces. Only if components are easy to re-use, easy to maintain or 'localise', modular and with a clear purpose, we will be able to ensure that such deliverables can be transposed in other Member States and shared with others.

Some recommendations are prescriptive (e.g. code must be commented) when beneficial to the re-use by others. Some recommendations can be supported only by heuristics, rather than precise and mechanically verifiable checks. Other recommendations articulate general principles.

It's important to remember that sometimes standards are environment-specific and depend on the platform and container elements chosen. This is especially true for User Interfaces. Nevertheless, the aim is to make the portability of tools and components as smooth as possible amongst different environments and ensure they are useful and usable.

Single MS Pilots will possibly start in a local language, however it's important to remember that the NIVA project language is English, that will facilitate the re-use of deliverables on multi MS Pilots. The language issue (not the programming language) may apply to user interfaces, application programming interfaces, naming, code comments, error messages, and many other software elements. To allow Multi MS Pilots during the project, developers should write the code following internationalisation best practices which enables software to be adapted to various languages and regions. Therefore, English translation/documentation should be provided to describe, for example, the field names and functions if written in another country language. Internationalisation recommendations will be given in a dedicated section and across chapters of this document.

If you're interested knowing how these guidelines were drafted see the storytelling at the end of this document.

3. Links to WP3 tasks and other NIVA Work Packages

NIVA's Work Package 3 has a main purpose "*to provide interoperability specifications to the NIVA project, ensuring that the IACS components developed by the project can be efficiently used by the identified local test sites and then effectively reused by a wider community (such as other Member States or other application domains). Interoperability and open standards will be a necessary condition to ensure the migration from national experiences to pan-European solutions*".

This D3.3 (Common guidelines for software development) document focuses on the aspects related to general software engineering standards, schemes and best practices that will enable better interoperability amongst the NIVA eTools. Other WP3 deliverables include documents that are going to deep dive in the IACS context:

- D3.4 - Recommendations for IACS workflow
- D3.5 - Recommendations for standardised connections between IACS project and other applications

This first version of the D3.3 document will be used as an input to the D3.3 and D3.4. Those will provide more data standards and schemas to be used in the IACS context.

Most of the NIVA software deliverables will be produced by WP2 (Use Cases) and WP4 (Common components), therefore this document aims to set a common understanding of software engineering practice for the software developers and NIVA partners involved in software development. Another link to WP4 is the need for software development teams to use a common NIVA repository for source code management, which is a best practice for software engineering.

A version of this document shall be further used by WP6 (Call for software components and pilot validations) as a reference for external party called to design, develop and/or document (other) software components to be re-used in the NIVA context.

4. What these guidelines do not cover

There are many issues regarding programming style that these guidelines do not cover. They do not, for example, dictate a programming language or a style convention.

Engineers should have the experience and knowledge to choose an appropriate language for a given project based on technical and economic merits, or suitability to a specific case. They are invited to consult with their NIVA peers to verify if architectural or language choices would fit in the NIVA project.

The current version of the guidelines does not:

- Prevent you from trying something not explicitly covered by the software development standard
- Dictate the use of, or forbid the use of, any software development tool (certain enterprise-wide tools are excepted, including configuration/version management tools and software defect tracking tools).
- Specify the choice of a programming language on a project.
- Specify the complete software design process.
- Deal with platform specific issues.
- Deal with project management issues.
- Deal with End-User documentation styling-guides.
- Deal with performance issues.

Although this document does not address the above issues, other NIVA documents and initiatives may very well do so. You should consult with the WP2 or the WP4 leader for the specific liberties and responsibilities that you have.

5. General recommendations for NIVA software development teams

5.1. Licensing of results: EUPL, CC BY and background IPR

Although not entering legal aspects, let's remind a few concepts on software licensing.

We define 'Results' as any (tangible or intangible) output of the NIVA action such as data, knowledge or information — whatever its form or nature, whether it can be protected or not — that is generated in the action, as well as any rights attached to it, including intellectual property rights (IPR).

The ownership of the Results is regulated by the NIVA Grant Agreement. 'Ownership' is a different concept from 'Licensing'. All Partners in the NIVA action have agreed that software Results, independently from their owner, shall be licensed under the EUPL (European Union Public License)[1].

Licensing requires to follow certain rules and to provide certain rights. Since you may be using other software to build your components, you need to care about the "cross-compatibility" of licenses.

About the compatibility of software licenses:

- Upstream compatibility: if you're incorporating, or linking with, components covered by another Free/Open Source Software (F/OSS) license, you must verify that such components provide you with the legal possibility to distribute your results as EUPL. You may not be able to distribute due to

[1] EUPL (European Union Public License) v 1.2 following EU Implementing Decision n. 2017/863 of the 18/5/2017. https://joinup.ec.europa.eu/sites/default/files/custom-page/attachment/eupl_v1.2_en.pdf

F/OSS incompatibilities results from copyleft conflicts. This effect is also known as 'contaminating licensing';

- Downstream compatibility: you must allow third parties to merge the Results under EUPL into a larger work which could be under a different license (F/OSS or commercial). That will include incorporating or linking your component in a larger work.

Despite not strictly regulated, Results in the form of documentation, or pseudo-code/algorithms, should be released with a similar 'open licensing' like, for example, the Creative Commons CC BY (https://creativecommons.org/licenses/by/4.0/).

Results in the form of research data, or scientific publication are also regulated by the NIVA Grant Agreement.

Licenses are usually regulating the rules and restrictions applicable to software components. In the NIVA project some commercial products with background IPR are also made available for the implementation of the action and for the exploitation of results (see Consortium Agreement – Attachment 1).

Recommendation: when using source code, including F/OSS, remember that there's always a background IPR. When incorporating or linking components, always remember to follow the licensing rules and restrictions applicable (for example: include the name of the owner or of the library, or to comply with the limitations of use like a maximum number of users).

The software stack needed to run your deliverables is also using third party's IPR (with related open source or commercial license), thus remember that you may need to follow the rules and restrictions applicable to those licenses.

Recommendation: in all cases, ensure that the licenses used or linked do not cause a "lock-in" in a single technology and that they allow the Results to be ported to a different software stack with a reasonable effort.

Recommendation: as part of the documentation, provide a list of third party's software libraries you have used to build your software and document their license type.

5.2. Choice of software stack (Programming languages, frameworks, libraries, databases and tools)

The Software Stack ('Stack') is meant as the environment enabling the execution of the NIVA components. It can include the Programming Languages, Frameworks, libraries, database, third party tools, operating system, linked components, virtual machines, etc.

The elements in the Stack can be open source or commercial products, as soon as they do not hinder the EUPL characteristic of the NIVA components.

Since software or algorithms produced within NIVA will initially need to work on an identified and real IACS, the choice of the 'Stack' will be possibly driven by the local existing Stack of the Target System. However, since the NIVA Results are intended to be re-usable and portable with reasonable effort across other IACS, the analysts/developers will need to follow at least the following:

Practical Rules

- Each software component shall include a short description/documentation of the elements of the Stack used;
- It shall be easy to hire people who can work with the Stack or, in other words, the elements chosen shall be known by a good number of developers;

- The elements in the Stack shall be currently supported and will continue to be supported by the ecosystem or the manufacturers, i.e. Open Source or Commercial, for at least the next 5 years
- Most recent versions are preferred over unsupported ones;
- If using third party's Open Source components in the stack: solutions with active development are preferred over inactive ones and solutions with many contributors are preferred over those with a few;
- If using third party's commercial components or solutions in the stack: where mature and popular, they can be used as soon as it is justified their advantage and if they do not hinder the portability of deliverables to a different stack, i.e. they can be replaced with similar components.

5.3. Choice of IDE – Integrated Development Environments and programming tools

The Integrated Development Environment (IDE) is the set of comprehensive software facilities that you will use to do software development. Examples are Eclipse or NetBeans. IDEs usually include the source code editor, build automation tools, compilers and debuggers.

The choice of IDE depends on the programming language chosen to create a specific deliverable or the target release platform. For certain components it may be possible to use more than one IDE, especially if you're results include both low level libraries and user facing components. And you could also use 'command line' instructions, since Linux functions can work as an IDE.

You may further choose to use a continuous integration environment or an automation server.

Recommendation: for the NIVA project use IDEs that contain widely used facilities and provide sufficient setup and operational instructions for people that needs to reuse your results.

The instructions should include:

- the list of baseline libraries, tools, IDE configuration variables, etc. that need to be initially configured inside the IDE;
- the instructions to build/compile an executable from the source code, or the related scripts;
- the instructions to deploy an executable through automation servers and/or related scripts.

When possible, organise your projects so that it can be re-used in different IDEs.

5.4. The central NIVA repository(-ies)

NIVA Results must be made available to, i.e. shared with, other developers.

A common repository for software deliverables is made available under "Work Package 4 – Knowledge Information System". Please refer to the specific deliverable "D4.3 - Procedures and instructions related to common environments" for instructions on how to use these environments.

Project documentation is also made available on the Sharepoint repository managed by WP1. Depending on the software components or pseudo-code produced, you may have to publish some documentation under the section "WP2 – Large scale pilot" or the section "WP4 – Knowledge Information System".

Recommendation: when releasing on the common environments, ensure you have given sufficient instructions on your deliverables enabling others to re-use them.

5.5. Deciding on Re-use vs Build

Your software deliverable will need to be reused or interact with other software components of larger systems. Furthermore, with nowadays richness of libraries, there's possibility that someone else has

already created parts of the software component you planned to build. This is also true inside the NIVA project.

You will frequently need to decide if it will be worth to build or buy/re-use parts of your component. This is never an easy decision also because developers may have preferences for various reasons, or there could be cost and risk reasonings, which include future maintainability of the solution.

NIVA is an 'Innovation Action', thus it shall include limited research/development activities. In other words, it is more focused on closer-to-the-market activities such as prototyping, testing, demonstrating, piloting, scaling up, etc. aimed at producing new or improved product and services. Therefore, every developer should prefer combining existing technology instead of developing new technology.

Another important characteristic of Innovation Actions is that they do not cover 'Market Replication' when the innovation has already been applied successfully once in the market (Europe or the application sector in question). Therefore, check if you're really innovating.

The above considerations are important since, although valuable, your work could not be recognised if the new technology is completely bespoke or not new at all.

Recommendation: you should not re-invent the wheel. If components or applications already exist, you're not innovating, however a new combination of existing technologies may be innovative (product or process). A minimal research inside the NIVA project or on the market shall be done to ensure you're not creating something that already exists.

Recommendation: if you're planning to re-use components from other NIVA use cases, but they're not yet ready, start with your existing components, but design/abstract your software to re-use new components in the future. Your target country may also already have some similar tools available (example you may already have a Farm Dossier or a Geotagged photo application in place), so use them first but design for the future.

6. Project structure for software components (source code project)

NIVA is a collaborating project and your deliverables will need to be re-used by others. Clarity in the organisation of your specific software project is therefore of utmost importance to allow other developers to benefit from your deliverables. This is true for software and pseudo-code as well.

A common repository based on GitLab is made available under "Work Package 4 – Knowledge Information System". Please refer to the specific deliverable "D4.3 - Procedures and instructions related to common environments" for instructions on how to get access and use this environment.

To host your codebase, you may also plan to use your own local environment, however put in place a process to release the software regularly on the common environment.

You're free to organise your software project structure, however we suggest following a few common practices ensuring that:

- It has a clear structure with meaningful names for folders, files, classes, libraries, etc.;
- It contains a README with reference links, how to get started, configuration requirements, how the project is organised, preferred IDE and its configuration), etc.;
- It contains a definition of the expected build environment and associated build scripts;
- If using third party components, it contains configuration instructions and scripts for such components;

- If using an external database, it contains DDL scripts to create the data model;
- If requiring interoperability with third party systems, you abstract the related classes or services, make a dedicated section/folder and explain how to get access to such systems or services;
- It provides test cases, test data, and a brief documentation with examples of use;
- It provides, or allow to access, a change log.

The above list is not exhaustive, since the kind of objects depend on the size and type of your project.

IMPORTANT: at some point your code will become 'public' and therefore you're required to prepare a brief note for the general NIVA documentation describing the business purpose of your deliverable, where to find the code and providing a description of the specific use cases implemented.

The following sections will provide recommendations on some specific elements on managing your project.

6.1. Source Code Management

Whether you are writing a simple component or doing a large software development, source control management (SCM) is a vital component of the Software Development Life Cycle (SDLC). We do not enter details of SCM for which you can find plenty of documentation.

Recommendation: use a Source Code Management system and release your deliverables regularly on the NIVA common environment.

6.2. Usage of external libraries

If you are using external libraries, components or specific Software Development Kits (SDK) as part of your deliverable, you must identify and explain their configuration and usage. Some documentation shall clearly explain how to get access to such elements and how to incorporate them in the project to ensure a smooth build process.

Recommendation: you shall document any dependencies and provide alternatives, if available.

6.3. Usage of external systems

If your system relies on external services, which could be another NIVA component or a service providing data, you shall give a clear indication on how to get access to such services, examples on how such services are invoked and the expected response(s).

Recommendation: you shall document the external boundaries of your system.

6.4. Development and test data

Some test data shall be provided to allow other developers using your software, i.e. once built to see how it works. If your system relies on external systems, refer to the section above.

Recommendation: you shall document some examples of use and provide test data.

7. Programming recommendations

There some general principles that you'll need to follow to ensure usability and readability of the source code or pseudo-code. The project will involve different programming languages, so it will not be possible to give detailed instructions on all of them, therefore use common sense first.

Each programming language has a common standard which will be the preferred option, however if for justified reasons you need to follow a non-standard practice, you'll need to document your choice.

Recommendation: try not to hinder other developers.

All conventions are there to collaborate, so:

- make sure code compiles before committing it;
- make sure code passes all tests (if they exist) before committing;
- if you need to check-in broken or incomplete code, use a branch, or somehow minimize the impact on other developers;
- commit code that is neat, portable, and documented.

7.1. Code and code conventions

Following a code convention greatly simplify re-use of code, apart from making it easier for your team to collaborate.

Code conventions may vary according to the programming language (you'll find a list of links to common style guides at the end of this document). Whatever the language, you shall care about the following main topics:

- **Formatting** - Sticking to a common set of formatting / indention rules makes it easier to read through our source base. Ensure indentation is used, ensure opening and closing brackets are always used in the same way, use white spaces in the right way to ensure readability, etc.;
- **Commenting** – Always comment your code (in English). If using Java, for example, you should use the java style blocks with params ("@"), so that documentation can be generated automatically. For other languages follow similar convention. Use brief 'in-line comments', too;
- **Naming** – Use English, be consistent on the use of 'camel case' or 'all upper case' depending on the object type (variables, constants, functions, types, etc.). As examples, link to the Glossary produced in D3.1, the semantic data models (D3.2) or use common standards like INSPIRE for the spatial data;
- **Programming Language Standards** – Prefer using existing standards (e.g. ISO C++) and when not possible, use clear description on what you use. Possibly avoid dialects if not justified.

A list of suggestions can be the following:

- indent using spaces instead of tabs, usually 4 spaces but 2 is ok too, just be consistent;
- use braces consistently (cuddled or not) within a file;
- use braces for all if-statements, including one-line conditionals;
- use 80-character lines max (this is very important for code);
- use appropriate, descriptive names for classes and variables
    - camel caps for variable names, starting with lower case (e.g., myVariable);
    - camel caps for class names, starting with upper case (e.g., MyClass);
- use javadoc or equivalent comments for every class and method
    - explain the purpose and intent of a class and how it fits into the overall architecture -when writing docs;
- remove extraneous code that is not used (classes, and methods in classes).

We also suggest that developers write short, to-the-point methods that encapsulate a very specific behaviour, rather than long procedural functions. If your methods are longer that 30-40 lines of code, or if they have extensive conditional blocks or switch statements, they might be broken up into several methods. But this is very subjective. Related to this is the importance of factoring out common procedures into their own classes or methods; if you find yourself writing the same type of functionality multiple times, it's time to refactor.

7.2. Pseudo Code and Algorithms conventions

If your task or use case includes the possibility to release a result which is not software, but can be implemented with some reasonable development effort, you're possibly releasing what is called "pseudo code".

The pseudo code is an implementation of an algorithm in the form of annotations and informative text written in plain English and that mimics code. It has no syntax like any of the programming language and thus can't be compiled or interpreted by the computer. The main goal of a pseudo code is to explain what exactly each line of a program should do, hence making the code construction phase easier for the programmer.

The reason to develop and release pseudo-code instead of a working software may be due to make the code construction easier for other programmers which may decide on their specific programming language, or to replicate a code under IPR (as soon as the "process/algorithm" is not under IPR restrictions too).

Your chosen algorithm must be represented in pseudo code to allow an immediate interpretation by programmers no matter what their programming background or knowledge is. The algorithm is an organized logical sequence of the actions or the approach towards a problem. Algorithms are expressed using natural verbal but somewhat technical annotations.

The pseudo code must neither be abstract, nor generalised. It must instead use control structures (if..then, loops and iterations), a proper naming convention, and indentation.

Pseudo code shall be delivered with relevant documentation and released on the NIVA Knowledge Base.

As explained in other paragraphs, since the NIVA project is an Innovation Action, pseudo-code is not a valid deliverable if the algorithm has not been implemented at least once in a real software. As an example, taking a scientific paper and writing pseudo-code to theoretically calculate 'something' will not be allowed as soon as:

- Either you demonstrate that you are using the algorithm inside a working software created within NIVA and you have therefore released also the working software on the NIVA Knowledge base, OR
- you demonstrate that you have produced the pseudo-code following an analyses of a pre-existing working software.

Recommendation: Something released as pseudo-code must be implemented at least once in the real world and in a real software. All conventions applicable for working software shall apply also for pseudo code.

7.3. Philosophy principles on coding (that cannot be checked automatically)

There are some principles on coding which cannot be checked automatically. On the web, you can find many references to the following list and choose a source that is more appropriate to your programming language.

We strongly recommend that you follow them to ensure the best user experience to those re-using your software:

- P.1: **Express ideas directly in code** – well-designed code will reflect better your underpinning ideas. Clear coding is most of the times better than just adding a comment.

- P.2: **Write in a Standard for the language used** – avoid language dialects or extensions as much as possible, choose a known standard and be consistent. This will facilitate portability across compilers.
- P.3: **Express intent** – use the code to tell what should be done rather than what it should be done, for example prefer 'named functions' and 'named classes' to generic data types
- P.4: **Ideally, a program should be statically type safe** – use, as much as possible, static types. When not possible seek for alternative constructions that are safer for your programming language.
- P.5: **Prefer compile-time checking to run-time checking** – try to use constructions which errors can be caught at compile time. Sometimes choosing the right data type will allow to avoid writing extensive error handlers.
- P.6: **What cannot be checked at compile time should be checkable at run time** – errors that are not programmer's logic shall be 'catchable'. Ensure you have analysed your code with some quality tools, verify machine resources used by your code, etc.
- P.7: **Catch run-time errors early** – There's plenty of recommendation on how to improve code to avoid run-time errors. For example, "checking if incremental loops can generate out-of-range errors", or "avoiding passing structured data as strings". Generally, such errors can be prevented by double-checking the following elements: pointers, arrays, conversions, and unchecked input values.
- P.8: **Don't leak any resources** – don't leave instances of objects around, if you're not using them. Try to clean up unused things. If you don't do it early, you may probably lose the reference to that object which will leak resources.
- P.9: **Don't waste time or space** – performance of your code is important and that means knowing the capabilities of your programming language. You should code for efficient execution both in time and space. Choose the right functions and the right structures.
- P.10: **Prefer immutable data to mutable data** – things that change are more prone to errors.
- P.11: **Encapsulate messy constructs, rather than spreading through the code** – low-level code can become easily unreadable and can create problems. Leave more experienced people to create this code, test it, and encapsulate in specialised libraries that will be used by 'more readable code'.
- P.12: **Use supporting tools as appropriate** – use tools like 'static analysers' to verify that your code follows standards.
- P.13: **Use support libraries as appropriate** – you probably already do it, however it's worth to remind that there's plenty of well-documented and well-supported libraries that already provide many of the capabilities you need. Knowledge of a widely used library can save time on other/future projects. So, if a suitable library exists for your application domain, use it.

7.4. APIs – Application Programming Interfaces and interoperability principles

Your software components are likely to be used by other computer programs, so you should provide a way to connect with them. To a minimum, such Application Programming Interface (API) is the 'external' methods you offer to view/query, edit, add or delete data.

There's many ways and standards to provide access to your components which can depend on your programming language and the level, or tier, you will offer access to. Whatever the case, you shall ensure that your API is:

- **Tested** – ensure that the API request/response mechanism is working by testing the use cases. Amongst others, this will include the core functions, core data, performances, and ancillary data (cookies, header).
- **Monitored** – ensure it is possible to check regularly if the API is alive and eventually provide a simple API to 'ping' your service. Also ensure that any error or exception is logged.

- **Documented** – describe as much as possible the format of inputs and outputs and the functionalities you offer in a clear way, to allow proper re-use of your capabilities.
- **Secured** – ensure you program APIs with enough security if those allow for remote access, for example, on public networks.

You shall prefer APIs and Web Services following the REST architectural style, to allow for implicit understanding of the resources, and combinations of operations (HTTP(S) methods) involved. REST best practices include:

- Resources are located using nouns, e.g. "`/crops`", rather than verbs, e.g. "`/getCrops`".
- HTTP method semantics must be followed, e.g. POST to the resource "`/crops`" should not retrieve all crops but create a new crop resource.
- Proper HTTP status codes must be given and include error payload. E.g. failed authentication should give "401 Unauthorized" rather than "400 Bad Request" and include error message in the error payload.
- Endpoint or URL must give meaningful information with respect to the relationship between resources and sub-resources. E.g. "`/api/farms/{farmId}/fields/{fieldId}`" better represents the relationship between a farm that has a field than "`/api/fields/{fieldId}`".
- Proper usage of HTTP header fields, e.g. request format is defined by the HTTP header "`Content-Type`".
- Usage of **H**ypermedia **a**s **t**he **E**ngine **o**f **A**pplication **S**tate (**HATEOAS**).
- Paging of resources, allowing for limitation of response size.
- Field selection, allowing for resource attribute selection to limit unnecessary data transfer, e.g. "`/crops?fields=id,name`".
- Use API versioning, e.g. "`/api/v1/crops`", for breaking changes.

If usage of verbs as part of the endpoint cannot be avoided, then a verbose explanation should be included.

7.4.1. Inputs

Your documentation of inputs should strive to include:

- Purpose of input
  - Data Object(s)
  - Query parameter for
    - Paging
    - Field selection
    - …
  - …
- If, and when, input is mandatory or optional. In case of optional values, specify which "default" is used by your software.
- Datatype and format
  - E.g. datetime using UTC Zulu time (2019-09-06T14:45:00.000000Z)
  - E.g. JSON array of strings

Note that you shall clearly distinguish between 'Data Object(s)', that means the data that must be processed, and query parameters, that means values given to control the behaviour of the processing and the wished outputs.

You shall also ensure that inputs are validated on your side as much as possible, returning an error message which clearly indicates unexpected input formats or values.

Naming of inputs shall follow, as much as possible, a terminology that is compliant with the NIVA glossary and the NIVA data model.

7.4.2. Outputs

Documentation of outputs should strive to include:

- Purpose of output
  - Data object
  - Resource identifier
  - …
- Datatype and format
  - E.g. JSON object encapsulating a key “datetime” with a UTC Zulu time string value {"datetime":"2019-09-06T14:45:00.000000Z")
  - E.g. JSON array of strings
  - E.g. XML document with a datetime element

Naming of outputs shall follow, as much as possible, a terminology that is compliant with the NIVA glossary and the NIVA data model.

7.5. UI/UX (User Interfaces and User Experience) principles

If your software deliverable is interactive and thus includes a graphical User Interface (UI) you shall ensure to design it following common best practices allowing usability, portability and, where possible, accessibility[2].

We are aiming at making the User Experience (UX) as easy as possible. If we achieve this result, the user won’t even notice it. On the other hand, for the NIVA project, you’ll need to ensure that other programmers can take over your source code, so the UI must be also easily programmable.

The design and programming principles are plenty, so we’ll try to recommend some high-level principles which may slightly vary depending on the running platforms you’re going to choose:

1. Place users in control of the interface – this will include: make user actions reversible (example: undo/redo), create easy navigation (example: provide visual cues and be predictable), provide feedback (acknowledge user actions, provide progress status, communicate clear error messages), think about users with different skills.
2. Make it comfortable to interact – this will include: avoid overloading a single page with content or features, don’t ask data twice, use common terminology, place action elements (example buttons) near to the last expected input, think of accessibility (example use W3C – WAI standards for impaired people), use friendly and polite language, ensure users never lose their work.
3. Reduce cognitive load – split a complex process in a sequence of different pages, organise and group items, prefer visual buttons to text buttons.
4. Make User Interfaces consistent – choose a colour palette, place similar action objects always in the same position, ensure that actions on different pages always behave in the same way, be consistent with users’ expectations (Android and iPhone users expect objects in specific places).

To ensure that your deliverables can be also easily re-used by other programmers, follow the general recommendations on coding conventions previously written in these guidelines. In addition, use programming languages and methodologies widely adopted. If programming for the web, an example can

[2] Since the NIVA project is also about processing geographical information, not all the usual accessibility requirements can be followed for GIS interfaces.

be HTML5. Ensure that your GUI uses style sheets (CSS) to allow customising the look and feel easily. Another example could be using well-documented UI libraries, like the Google Web Toolkit.

Recommendation: test your UI on several browsers or mobile equipment. Your documentation will need to explicitly mention where your deliverable has been tested.

7.6. Documentation for other developers

The purpose for this recommendation is ensure you provide some documentation on the software executable and its capabilities, on the software code and on its usage, etc.

On the initial versions of the software there'll be no need not to write extensive documentation since there's more focus on 'working software', however a minimal set of documentation and/or documented code is required to enable others to re-use the components or tools.

We recommend that for each component or tool, you'll make available the description the Business Purpose and intended audience, the Use Cases or User Stories, the IT environment required (software stack), the software architecture, how to handle the source code and how to customise it, etc. There are several examples you can follow; however, we suggest doing it in an SRS-like document which has a simple structure helping taking care of those. You can find several examples of SRS (Software Requirement Documents) on the web.

The software delivered in the NIVA-project shall be published as open source software. A few guiding principles and questions for the documentation are:

- Explain in a few words, what the software will do, what business purpose and user stories it will solve;
- Explain the prerequisites for the user, what skills will the tester/extending developer need;
- Explain how to use it, installation, coding examples etc.;
- Explain how to extend it, what will be the prerequisites and requirements for contributing to the open source project;
- Explain the kind of license for the component or software project (refer to the Licensing paragraph).

The amount of documentation may vary if you are developing a simple component or a more complex solution. Good practice for in-line documentation in the code should be followed. Documentation shall be updated along with the code and peer reviewed.

There are several standards you can refer to ensure you provide enough documentation. The information you should provide may include, for example:

- IEEE 29148-2018 – Systems and software engineering – Life cycle processes – Requirements engineering: Intended audience, solved Business Case and Use case specifications;
- IEEE 1016-2009 – Systems Design – Software Design Descriptions;
- Installation/Configuration/Build instructions. Data definition language and scripts in case of usage of a database. Installer checklist;
- FAQ for developers that will need to re-use the component;
- Release notes or Change Log.

7.7. Test cases and testing

Testing is a process that includes many activities and should be implemented from the early stages and throughout the entire development process to verify and evaluate requirements, code and to detect

potential defects. Early involvement should start with reviewing work products as soon as drafts are available before proceeding to programming. It is good practice to provide a corresponding test activity to each programming activity e.g. unit of code should be unit tested and integration of components are integration tested. In general, you should tailor the appropriate test activities to the software development lifecycle. Testing is done differently in different contexts and the appropriate test strategy and process depends on many factors. We recommend following a few principles to ensure better quality:

- Use the concepts and vocabulary introduced by ISO/IEC/IEEE 29119 Software Testing. The International Standard has defined the best approach to test. If access to the standard is limited, then ISTQB offers a glossary list which can be accessed here: https://glossary.istqb.org/en/search/.
  A 2018 foundation level Syllabus based on ISO/IEC/IEEE 29119 Software Testing can be downloaded for free here:
  https://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html.
  Be conscious of some of these Syllabus and glossaries which refer to outdated standards e.g. Foundation 2011 refers to IEEE 829-2008 which has been superseded by ISO/IEC/IEEE 29119.

- Know the general guidelines of the seven testing principles:

  1. Testing shows the presence of bugs.

     Testing does not guarantee that the application is free from defects. It cannot show the number of undiscovered defects present even after thorough testing. It can only reduce the probability of undiscovered defects.

  2. Exhaustive testing is impossible

     Testing all combinations is not possible or reasonable. Consider an application with 10 input fields with each having a possibility of 3 values. In this case, if you want to test all the combinations, then you will have to check 59049 combination of tests which is both mentally demanding, time consuming and inefficient at finding defects. Instead focus the test effort by using risk analysis, test techniques and priorities.

  3. Early testing saves time and money

     Finding and correcting defects early in the software development lifecycle can reduce costly changes. E.g. a review of a work product in an incremental software development lifecycle can quickly spot and correct an unforeseen action which could have otherwise become a defect in the code that would require lots of coding and testing effort to fix.

  4. Defects cluster together

     A small number of modules are usually responsible for accumulating the large number of defects discovered during pre-release testing. Predicted defect clusters, and the actual observed defect clusters in test or operation, are an important input into a risk analysis used to focus the test effort.

  5. Beware of the pesticide paradox

     If the same tests are repeated, eventually these tests are no longer effective at finding defects, just as pesticides are no longer effective at killing insects after a while. To detect new defects and overcome the situation of pesticide paradox, it is important to

review and upgrade the set of test cases on a frequent basis. In some cases, such as automated regression testing, the pesticide paradox has a beneficial outcome, which is the relatively low number of regression defects.

6. Testing is context dependent

   Testing is a context driven approach. For instance, testing in an agile project is done differently than testing in a sequential lifecycle project. Test should be tailored using appropriate test techniques and suitable test cases during the right phase.

7. Absence-of-errors is a fallacy

   A software product free of bugs does not guarantees success. E.g. if testing is performed on wrong requirements then in this case, fixing or finding defects is of no use.

- Implementing static testing can identify and rectify problems that dynamic testing might overlook. It also provides means to testing early such as reviewing requirements even before code is written. This can potentially improve the quality of the requirements, identify dependencies and provide clear, concise and common understanding between the developers and the requirement team. Peer code review which is a type of white-box static testing is particularly an effective type of testing that could potentially find weaknesses and defects that otherwise would be hard or nearly impossible to discover in black-box testing. A review is effective due to differences between the author's and the reviewer's cognitive biases. Implement peer code review before other black box testing activities because it can easily lead to fundamental changes in the code which would require a complete retest of the feature.

- Implementing dynamic testing is today a standard in many organizations. Developers unit tests to simply verify that individual units of code work as expected and integration test to test that the integration of units' work. Then, accept testing that work product would verify that it is aligned with the requirements. A tester could further test with various suitable test techniques to reduce the probability of defects.

- Implement means of tracking the quality of the work product e.g. it is important to establish and maintain traceability throughout the test process between each element of the test and the various test work products.

7.8. Error and Exception Handling

Programming and testing recommendations have been given earlier in this document also to ensure that your code is as error-free as possible. However, there's no software which is 100% error-free, even when you are methodically testing it.

For non-trivial software, it is not always possible to test all different ways users are interacting with an application and there is always the possibility that users will use your application in ways you may not have thought of.

Whatever the programming language you use, we can consider a 'programming error' as an event where there is not the possibility to recover/continue a process. Errors can be usually avoided by simple checks in your code, for example validating the input of a user.

Many errors can turn into 'exceptions' (the terminology may vary depending on the programming language used). An exception can be defined as a 'managed error', that means a situation where an error is thrown

but your code can catch it and your code can handle and recover such situation. An example is when an external call returns an empty record, so you have no data to process. If you handle the exceptions, you handle the process and therefore you won't need, in many cases, to deliver an error message.

Handled and unhandled exceptions can be logged to allow later fixing. So, you shall implement a logging feature and provide the possibility to set a few levels of logging to control the quantity of information written in the log.

The error or warning messages shall be translatable to other languages, therefore follow the recommendations in the i18n paragraph.

Recommendations: Favour exception handling over error codes, program exceptions as part of your software interface specification, check inputs on the User Interface and provide a logging mechanism.

7.9. Security of the code

The NIVA project deliverables are targeted to the IACS domain. You must be aware that there is a general regulatory requirement[3] that requires all paying agencies and other bodies handling EU finance to ensure that the information systems are secure. Therefore, your components shall be secure by design.

The EU Regulation states that paying agencies with a yearly expenditure above 400M must be certified with an ISO27001 or equivalent standard. Those below the limit have to apply at least ISO27002 practices or equivalent. If they don't comply, they lose their status of Paying Agency.

ISO27001 (or ISO27002) cover the many aspects of information security amongst which some are about Software Development. However, you shall also consider the connected requirements on Information Security, Risk Management, Monitoring & Evaluation, Documented information, Corrective actions, IT Security, Backup & Recover, and Cloud Computing Security.

If you need details on the above topics, you should get in contact with the ISO27001 experts in your company, or in your target IACS, to ensure that your software complies with security requirements. They can provide a checklist, usually known as Annex 1, which contains software related checks.

For the purpose of these guidelines, we will recommend mainly to follow the best practices that limit the number and type of 'threats & vulnerabilities.' There's many automated scan software that can help you identifying them, so you will be able to create code that provides safeness from typical attacks. These checks shall be done on libraries, low-level components, or applications with a Web interface.

You shall further consider any requirements around access rights, or Role Based Access System (RBAC), which controls the authentication and authorisation of users or services using your software. You may implement your own RBAC, but you will need to think of connecting to an existing one.

Finally, your software or solution needs to ensure the Business Continuity[4] of the Paying Agency. A crash or malfunctioning of your component may hinder the main process aimed to pay beneficiaries. Software designers will have to think of this possibility and keep the business running by providing or documenting back-up solutions or workarounds.

Recommendation: design the security of your components from the outset.

[3] Commission Delegated Regulation (EU) No 907/2014

[4] See ISO 22301 "Societal security -- Business continuity management systems -- Requirements"

7.10. Portability

A main purpose of NIVA is to ensure the portability of software components across multiple IACS, that means making them usable in different environments, i.e. computing platforms or operating software stacks. Some programming languages are designed to better support the portability, however there is a general principle to be followed: design your software in tiers (example: user interface, business logic, system interfaces and database) and use abstraction layers.

Every NIVA component will be initially developed for a specific IACS instance using a specific operating stack. However, its documentation shall provide a description of at least one alternative software stack and the actions required for the porting. This will ensure that designers have thought of portability from the outset.

We do not ask to develop software that works immediately on different platforms, what we are asking is to consider portability from the beginning and avoid making design choice that hinder it. This will also require thinking on the costs for portability and choose technical solutions that lower any portability efforts.

Recommendation: design your components to be portable from the outset, even if you will start implementing it in just one environment.

7.11. i18n (internationalisation)

"*Internationalisation" (short: 'i18n') is the design and development of a product, application or document content that enables easy localization for target audiences that vary in culture, region, or language*" (source W3C)

Developers should write the code following certain processes which make it easily possible for the software to be adapted to various languages and regions. This can apply to text or buttons on user interfaces, tooltips, or error messages returned by a software component.

Techniques to ensure i18n can vary depending on the programming language chosen, however there are some recommendation which are accepted as best practices:

1. Enabling the use of Unicode – Unicode/UTF-8 is recommended practice unless dealing with Asian languages that require UTF-16;
2. Avoiding Concatenation of Strings – Developers sometimes concatenate two or more strings to save space. This can easily result in translation errors in the localization process;
3. Avoiding hard-coded strings – All user-visible strings must be externalized appropriately;
4. Taking care of "simple and formatted arguments" – All 'dynamically-built' strings shall consider that those can be rendered in different languages;
5. Support for features that aren't used until localization – Developers shall think if the local functionality may need localisation and add mark-ups to future users;
6. Support for local, regional, language, or cultural preferences – Examples – Local calendars, date and time formats, measures etc. Typically, this can be achieved by attaching predefined localisation data from existing libraries or user preferences;
7. Separation of source code/content from localisable elements.

The typical component elements to consider for i18n are:

- Information architecture and workflow;
- UI elements (messages, menus, buttons, tooltips, contextual help, etc.);
- API elements (I/O fields, messages, etc.);
- Tutorials;

- Software documentation;
- Release notes.

Recommendation: for NIVA, when developing code, the developers shall be instructed to possibly use English as a default language for code comments or when creating instructions to set-up or build the software.

Recommendation: 'double' your chosen local language with English. In other words, try to create a component with a UI that works both in your local language and English from day one.

7.12. Committing code and change logs

Whether you're using your internal Source Code Management system or you're using the NIVA Source Code Management repository (see documentation on WP4-D3.3), we recommend you follow some general principles:

- (before committing) compile the code locally or run a local build;
- (before committing) double-check that you're not making changes that can break up other's people code, especially if you work on low-level libraries;
- (before committing) update comments in your source code, which includes deleting obsolete comments;
- (before committing) remove any temporary code. For example: System.out or System.err to print messages to the console;
- (before committing) run unit tests;
- (before committing) run a static analysis;
- commit code frequently, in small and logically related patches with good log messages;
- remember to check-in added files (or removed ones);
- make sure you have given a meaningful description of your changes, so that creating a change log will be easier;
- automate the process: use a transparent build system to expose all dependencies and requirements to build a project. A trivial example is 'Apache Ant', but other build tools are possible too.

An important thing to be released on the NIVA Source Code Management repository will be a 'readable' change log. This will be used to communicate in a technical but readable language what are the new features, or the bug fixed, on a new version.

Recommendation: communicate the changes to your software in a readable format to inform the developers that would like to re-use your components.

7.13. Delivery, deployment and/or installation

Your deliverable is not just the core code, classes and the database. You will need to provide everything that is needed to create and deploy/install an executable version of your software.

Therefore, the software must include instructions on how to make it really work in a chosen environment. This may take the form of an installation script, an installation program (in which case you'll need to provide source its code too), some written instructions, a list of external components, etc.

If you plan to use a Continuous Integration system, then provide its type, the configuration, the instructions, the scripts, etc. so that someone can easily re-create a similar environment in another place.

The deployment of your software may depend on many factors, from the development environment chosen to the target system, or from the application container chosen (e.g. JEE, docker, …) to the automation deployment process (e.g. Puppet tasks).

Recommendation: prepare instructions for other developers in order to enable them setting up and use a deployment environment which delivers working solutions.

8. Taking care of users

8.1. Provide support and bug reporting/fixing

You shall provide a service for users and testers to report and log information about any adverse event that may occur in your software. And, you shall describe the service you provide and the service level you offer.

Remember that your software is going to be used on real IACS environments for 12-months and serving a main mission critical process: paying CAP subsidies. Even if you build a simple component, it may be a critical brick of a larger application. Make your users aware on how to get in contact with you to ask for assistance and provide a clear indication on your service levels so that, if something goes wrong, remedial actions can be planned accordingly.

Users should be guided to state the problem as clearly as possible so that developers and other parties can identify the adverse event(s), replicate and isolate the problem and correct the potential defect and/or solve the problem. During the defect management process, some of the reports may turn out to be false positives. You shall clearly state who is responsible to receive and solve bugs in your organization and where to issue the report.

You should instruct your users to report issues effectively so that time and effort is not unnecessarily wasted in trying to understand and reproduce the defect. Explain them to:

- Be specific and to the point. Bulleted list of information can contribute to that. Do not combine multiple problems but instead write different reports for each problem.
- Be detailed and provide as much information as possible.
- Be objective and use facts and avoid using subjective statements and emotion.
- Review the report and correct typographical error or mistakes.

Provide your users with a template to report an issue. A simple defect report should include but is not limited to:

- Title;
- Environment e.g. version and source URL;
- Date and time of the incident;
- A short summary of the adverse event;
- A step-by-step description of how to reproduce the adverse event;
- Expected and actual results;
- If possible, any attachment such as logs, database dumps, screenshots, videos etc.

If you're using a defect management tool or system, provide users with an access and offer pre-established forms with standard data, such as:

- **Defect Severity** can be assigned to state the degree of impact that a defect has on the development or operation of a component or system.
- **Defect Priority** can be assigned to indicates the importance or urgency of fixing the defect.

- **Reported By** can be assigned to know who reported the defect for instance in case more information is needed.
- **Assigned To** can be assigned to know who this bug is assigned to currently e.g. to analyse or fix the defect.
- **Status** tracks the defect life cycle.
- **Fixed Build** Version which is simply the build version where the defect was fixed.
- **Remarks**.

Recommendation: provide a support service to those willing to re-use your software components.

9. Useful links and other reference documentation

9.1. Links to specific Language guidelines and stylesheets

This list is not exhaustive and will be updated in future versions of the guidelines.

- Java: Java programming guidelines
- C++: ISO C++ or Google C++
- JavaScript: Google JavaScript Style Guide
- PHP: PSR-2: Coding Style Guide
- PERL: perlstyle
- JSP: Code Conventions for JSP or JSP - Quick Guide
- Phyton: PEP 8 -- Style Guide for Python Code
- R language: Google's R Style Guide or R Coding Style Guide
- HTML/CSS: Google HTML/CSS Style Guide or Xfive CSS Coding Standards
- SQL and PL/SQL (Oracle): SQL Coding Guidelines, or TopCoder
- pgSQL and PL/pgSQL: no defined standard, use PostgreSQL documentation examples
- Pseudocode

9.2. Other documentation

- Best Coding Practices (on Wiki)
- Pseudo-code (on Wiki)
- Android Mobile Apps – Design and quality guidelines
- Apple iOS Design Themes

10. How we developed the NIVA software development guidelines

These guidelines are based on recommendations put together by software development and IT experts in consultation with people using services and the public administration.

Several factors influence the guidelines and the order of development.

10.1. Topics chosen (June 2019)

Topics are covering the IACS (Integrated Administration and Control System) domain as described by the relevant EU regulations in conjunction with the need to favour the exchange of software deliverable amongst NIVA partners and the European Paying Agencies that will need to pilot or implement such software inside their national IACS implementations.

10.2. Purpose and target produced (July 2019)

The purpose outlines why there is a need for the guidelines, the areas the guidelines will and will not cover, and what they intend to achieve. A draft version is provided to organisations (stakeholders) with a knowledge and an interest in the topic to comment on, i.e. the NIVA technical Partners.

10.3. Guidelines developed (August 2019)

We review the comments of the stakeholders and the evidence relevant to the guidelines. The comments helped to identify literature searches to complement the guidelines and act as a guide to develop this document.

A literature search is carried out and a new draft of the guideline is prepared to be considered by the committee responsible for the deliverable.

10.4. Draft sent for consultation (early September 2019)

The draft is sent for consultation to the NIVA partners and the guidelines are assessed for their impact.

10.5. Comments considered and guidelines revised (mid-September 2019)

The parties responsible for the development consider comments from stakeholders and agree any changes. The revised version is checked for quality.

10.6. Guidelines signed off and published (end-September 2019)

According to the project plan the first version of the guidelines is submitted to the Work Package leader and the NIVA Coordinator and published for sign-off. We then work with the coordinator and the WP leader to communicate, disseminate, promote awareness and implement the guidelines

10.7. Updating the guidelines (May 2020, May 2021)

Guidelines are updated yearly according to NIVA plans, priorities, users' needs (mainly the Paying Agencies), and feedback from the developers. Routine maintenance can be done more frequently in case of mistypes, errors.