ETSI GR ENI 016 V2.1.1 (2021-07)

Experiential Networked Intelligence (ENI); Functional Concepts for Modular System Operation

Disclaimer

The present document has been produced and approved by the Experiential Networked Intelligence (ENI) ETSI Industry Specification Group (ISG) and represents the views of those members who participated in this ISG.
It does not necessarily represent the views of the entire ETSI membership.

Reference
DGR/ENI-0026_Funct_Concept_MSO

Keywords
architecture, functional, software

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Important notice

The present document can be downloaded from:
http://www.etsi.org/standards-search

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the prevailing version of an ETSI deliverable is the one made publicly available in PDF format at www.etsi.org/deliver.

Users of the present document should be aware that the document may be subject to revision or change of status.
Information on the current status of this and other ETSI documents is available at
https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx

If you find errors in the present document, please send your comment to one of the following services:
https://portal.etsi.org/People/CommiteeSupportStaff.aspx

Notice of disclaimer & limitation of liability

The information provided in the present deliverable is directed solely to professionals who have the appropriate degree of experience to understand and interpret its content in accordance with generally accepted engineering or other professional standard and applicable regulations.
No recommendation as to products and services or vendors is made or should be implied.
No representation or warranty is made that this deliverable is technically accurate or sufficient or conforms to any law and/or governmental rule and/or regulation and further, no representation or warranty is made of merchantability or fitness for any particular purpose or against infringement of intellectual property rights.
In no event shall ETSI be held liable for loss of profits or any other incidental or consequential damages.

Any software contained in this deliverable is provided "AS IS" with no warranties, express or implied, including but not limited to, the warranties of merchantability, fitness for a particular purpose and non-infringement of intellectual property rights and ETSI shall not be held liable in any event for any damages whatsoever (including, without limitation, damages for loss of profits, business interruption, loss of information, or any other pecuniary loss) arising out of or related to the use of or inability to use the software.

Copyright Notification

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™** and **LTE™** are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This Group Report (GR) has been produced by ETSI Industry Specification Group (ISG) Experiential Networked Intelligence (ENI).

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The purpose of the present document is to provide information on software design principles for constructing modular systems to be applied to the ENI reference system architecture (and any other applicable ETSI reports or standards). This will cover common concepts such as Functional Block design, state machines, cognition, inferencing, along with communication between different domains.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI GS ENI 005 (V2.1.1): "Experiential Networked Intelligence (ENI); System Architecture".

[i.2] Gamma, E., Helm, R. Johnson, R., Vlissides, J.: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Nov, 1994. ISBN 978-0201633610.

[i.3] Martin, R. C.: "Agile Software Development, Principles, Patterns, and Practices", Prentice Hall, 2003 ISBN 978-0135974445.

[i.4] Rowley, J.: "The wisdom hierarchy: representations of the DIKW hierarchy", Journal of Information and Communication Science, 33(2): pp. 163-180.

[i.5] Meyer, B.: "Object-Oriented Software Construction", Prentice Hall PTR, 2nd edition ISBN 0-13-629155-4.

[i.6] Liskov, B.H. and Wing, J.M.: "Behavioral Notion of Subtyping", ACM Transactions on Programming Languages and Systems, November, 1994.

[i.7] Eugster, P. Th., Felber, P.A., Guerraoui, R., and Kermarrec, A.M.: "The Many Faces of Publish/Subscribe", ACM Computing Surveys, Vol. 35, No. 2, June 2003, pp. 114-131.

[i.8] Leymann, F.: "Loose Coupling and Architectural Implications", ESOCC 2016 keynote.

[i.9] Ingeno, J.: "Software Architect's Handbook", Packt Publishing, pg. 175, 2018. ISBN 178862406-8.

[i.10] ETSI GR ENI 018 (V1.1.1): "Artificial Intelligence Mechanisms Introduction to Artificial Intelligence Mechanisms for Modular Systems".

[i.11] The SysML profile is defined at: https://www.omgsysml.org/.

[i.12] Boyd, J. R.: "The Essence of Winning and Losing", June, 1995.

[i.13] Strassner, J., Agoulmine, N., Lehtihet, E.: "FOCALE - A Novel Autonomic Networking Architecture", ITSSA Journal 3(1), pp. 64-79, 2007.

[i.14] MEF 95: "MEF Policy Driven Orchestration:, J. Strassner, editor, April 2021.

[i.15] MEF 78.1: "MEF Technical Specification: MEF Core Model", Strassner, J., editor, July 2020.

NOTE: Available at https://www.mef.net/resources/mef-78-1-mef-core-model-mcm/.

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the terms defined in ETSI GS ENI 005 [i.1] and the following apply:

abstraction: hiding of unnecessary details to focus on data and information that is relevant for defining a particular concept or process

architecture: set of rules and methods that describe the functionality, organization, and implementation of a system

- **software architecture:** high-level structure and organization of a software-based system. this includes the objects, their properties and methods, and relationships between objects

axiom: statement that is assumed to be true, in order to serve as a starting point for further reasoning

capability: type of metadata that represents a set of features that are available to be used from a managed entity

cognition: process of understanding data and information and producing new data, information, and knowledge

context: collection of measured and inferred knowledge that describe the environment in which an entity exists or has existed

design pattern: general, reusable solution in a given context to a commonly occurring software problem

NOTE: This type of design pattern is not an architecture and not even a finished design; rather, it describes how to build the elements of a solution that commonly occurs. It may be thought of as a reusable template.

domain: collection of Entities that share a common purpose

NOTE 1: Each constituent Entity in a Domain is both uniquely addressable and uniquely identifiable within that Domain. This is based on the definition of an MCMDomain in [i.15].

- **administrative domain:** Domain that employs a set of common administrative processes to manage the behaviour of its constituent Entities. This is based on the definition in [i.15].
- **management domain:** Domain that uses a set of common Policies to govern its constituent Entities

NOTE 2: A Management Domain refines the notion of a Domain by adding three important behavioral features:

1) it defines a set of administrators that govern the set of Entities that it contains;
2) it defines a set of applications that are responsible for different governance operations, such as monitoring, configuration, and so forth;
3) it defines a common set of management mechanisms, such as policy rules, that are used to govern the behavior of MCMManagedEntities contained in the MCMManagementDomain. This is based on the definition of an MCMDomain in [i.15].

entity: object in the environment being managed that has a set of unique characteristics and behaviour

NOTE: Objects are represented by classes in an information model.

formal: study of (typically linguistic) meaning of an object by constructing formal mathematical models of that object and its attributes and relationships

functional architecture: model of the architecture that defines the major functions of each module, and how each module interacts with each other

functional block: abstraction that defines a black box structural representation of the capabilities and functionality of a component or module, and its relationships with other functional blocks

graph: collection of nodes, where some subset of the nodes is connected

NOTE: Visually, a node is a "point" and a connection is a "line", called an "edge". For the purposes of ENI, any graph may be directed, weighted, or both.

knowledge: analysis of data and information, resulting in an understanding of what the data and information mean

NOTE: Knowledge represents a set of patterns that are used to explain, as well as predict, what has happened, is happening, or is possible to happen in the future; it is based on acquisition of data, information, and skills through experience and education.

- **inferred knowledge:** knowledge that was created based on reasoning, using evidence provided
- **measured knowledge:** knowledge that has resulted from the analysis of data and information that was measured or reported
- **propositional knowledge:** knowledge of a proposition, along with a set of conditions that are individually necessary and jointly sufficient to prove (or disprove) the proposition

learning: process that acquires new knowledge and/or updates existing knowledge to optimize a function using sample observations

logic: formal or informal language that evaluates a conclusion based on a set of premises

- **first-order logic:** extension of propositional logic to include predicates and quantification
- **modal logic:** representing, using mathematical formalisms, expressions involving necessity and possibility
- **propositional logic:** manipulation of a set of propositions, possibly with logical connectives, to prove or disprove a conclusion

NOTE: Propositional logic does not deal with logical relationships and properties that involve the parts of a statement smaller than the statement itself. It also called sentential logic, zeroth-order logic, and propositional (or sentential) calculus.

model: representation of the entities of a system, including their relationships and dependencies, using an established set of rules and concepts

- **information model:** representation of concepts of interest to an environment in a form that is independent of data repository, data definition language, query language, implementation language, and protocol

NOTE: This definition is taken from [i.15].

ontology (for ENI): language, consisting of a vocabulary and a set of primitives, that enable the semantic characteristics of a domain to be modelled

policy: set of rules that is used to manage and control the changing and/or maintaining of the state of one or more managed objects

NOTE: This is defined in [i.13] and [i.14].

repository: centralized location of a set of storage devices that enable different functional blocks to store and retrieve information

- **active repository:** repository that pre- and/or post-processes information that is stored or retrieved

NOTE: It may contain dedicated (typically internal) Reference Points that provide the loading, activation, deactivation, and unloading of specialized functions that change the pre- and/or post-processing functionality according to the needs of the application.

- **passive repository:** repository that stores or retrieves information without pre- or post-processing

semantics: study of the meaning of something (e.g. a sentence or a relationship in a model)

situation: set of circumstances and conditions at a given time that may influence decision-making

telemetry: automated process of recording and transmitting data to receiving equipment for monitoring purposes

NOTE: The process is typically automated, and the data transfer may include wireless, cellular, optical, and other mechanisms.

theorem: set of statements that has been mathematically proven to be true, based on a set of axioms and/or (previously proven) theorems

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AD	Access Device
AG	Aggregation Device
API	Application Programming Interface
CR	Core Router
DIKW	Data-Information-Knowledge-Wisdom
IP	Internet Protocol
OODA	Observe-Orient-Decide-Act
OWL	Web Ontology Language

4 Introduction

4.1 Fundamental Software Design Principles

4.1.1 Introduction

The purpose of the clauses below is to describe some key software architecture principles that were used in the design of the ENI System Architecture document. Each of these principles is based on designing modular software components and systems, and can be used for general purposes. In each of the following, the term "unit" means class, component, or module.

4.1.2 Information Hiding and Encapsulation

Information hiding is a design principle that states that if one unit does not need to know how another unit works, then it does not need do so. This ensures that each unit can be developed independently. This principle applies to classes, components, and modules (hereafter referred to as "units").

Put another way, information hiding mandates *loose coupling* (see clause 4.1.8). If there is a possibility that the functionality of the unit will change, then that unit needs to be separated from other units.

Encapsulation is *not* the same as information hiding. Encapsulation is an implementation mechanism that defines the boundaries of a unit. For example, a class is a collection of attributes and methods that are part of a single object. Put another way, encapsulation prevents the direct access to a unit's implementation details by a client.

Continuing the example of a class, it hides information by hiding implementation detail, and it encapsulates the object by combing code and data. Information hiding prevents clients of the class from knowing too much about the details of the class, and reduces coupling (see clause 4.1.8). Encapsulation prevents clients from accessing the implementation of the class, and increases cohesion (see clause 4.1.9).

4.1.3 Single Responsibility Principle

The original definition of this principle comes from [i.3], and is stated as follows:

A class needs to have only one reason to change.

In this definition, "reason to change" is what the unit is designed to do. This does not mean that a unit consists of one attribute or method; rather, it means that the set of all attributes and methods in a unit are related to a single responsibility. In practical terms, this means that different functions that have different purposes (e.g. analysis and printing data), and therefore, need to be split into separate units. This also increases the readability, testability, and maintenance of the unit.

4.1.4 Open-Closed Principle

This was first defined in [i.5] as follows:

"Software entities (classes, modules, functions, etc.) need to be open for extension, but closed for modification."

This principle is best illustrated by an example. Consider the action of writing to an entity, such as a disk file. This principle states that the write action can apply to any device (e.g. a printer or a screen, which makes it *open for extension*) without having to change the implementation to be device-specific (which makes it *closed for modification*).

Note that this can be implemented in two different ways: via inheritance and via interfaces. The problem with using inheritance is that subclasses are tightly coupled to their superclasses if they depend on the implementation details of their superclasses. In contrast, interfaces introduce an additional level of abstraction, which enables loose coupling. Interfaces can be changed without effecting the implementation that uses them. Furthermore, the interfaces of a unit are independent of each other. The most common way to do this effectively is to use composition.

4.1.5 Liskov Substitution Principle

This was first defined in [i.6] as follows:

"If an object X is a subclass of an object Y, then objects of type Y may be replaced with objects of type S without altering the behaviour of the program".

This principle is also called (strong) behavioural subtyping, because it guarantees semantic interoperability between object types in a system. This is often enforced using Design by Contract (see clause 4.1.11). In particular, Liskov Substitution requires the following restrictions to be true when a subclass is substituted for its superclass:

- Pre-conditions cannot be strengthened in the subclass (see clause 4.1.11).
- Post-conditions cannot be weakened in the subclass (see clause 4.1.11).
- Invariants are preserved in the subclass (see clause 4.1.11).
- New exceptions cannot be generated by methods in the subclass unless they are subtypes of exceptions that are generated by methods of the superclass.
- Method return types in the subclass are preserved (e.g. the return type of the subclass is a subtype of the return type of the superclass).

Method parameter types in the subclass is reversed (e.g. the parameter types of the superclass are subtypes of the parameter types of the subclass).

Liskov substitution provides more robust and modular unit designs.

4.1.6 Interface Segregation Principle

This was first defined in [i.3] as follows:

> *"Clients have a duty to not be forced to depend upon interfaces that they do not use".*

In other words, instead of having a small number of interfaces that have multiple responsibilities, a modular unit designed using this principle will have a large number of client-specific interfaces, where each client-specific interface has a single responsibility. Separate clients need separate interfaces. This increases the cohesion between the interfaces of a given unit.

The goal of this design principle is to reduce side effects caused by changing a unit's implementation. It is similar to the Single Responsibility Principle (see clause 4.1.3), in that by splitting software into multiple independent parts, it enables each part to evolve independently. For example, if unit A depends on unit B at *compile* time, but not at *run time*, then changes to unit B will force unit A to change. This is especially important for statically typed languages, like Java, C++, and C#. The Adaptor software pattern [i.2] is an example of a design pattern that can be used to support interface segregation.

4.1.7 Dependency Inversion Principle

This principle is based on the Open-Closed Principle and the Liskov Substitution Principle (see clauses 4.1.3 and 4.1.4, respectively), and was defined in [i.3]. It enables higher-level units to be loosely coupled to any lower-level units that depend on them. Specifically, the principle states:

> *High-level modules have a duty to not depend on low-level modules. Both have a duty to depend on abstractions (e.g. interfaces).*
> *Abstractions have a duty to not depend on details. Details (concrete implementations) have a duty to depend on abstractions.*

The essence of this principle is that both high-level and low-level modules depend on the abstraction. Put another way, this principle splits the dependency between the high-level and low-level modules by introducing an abstraction between them.

High-level modules, which provide complex logic, have a duty to be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, an abstraction that decouples the high-level and low-level modules from each other is required. For example, if this principle is not followed, then high-level business logic will depend on low-level implementation details.

4.1.8 Loose Coupling

Coupling [i.8] refers to how inextricably linked different aspects of an application are. High coupling needs to be avoided between units, because this forces the evolution of each unit to depend on each other. In contrast, low coupling ought to be used whenever possible, as this enables each unit to evolve independently. This also enables each unit to be more easily reused, since it has fewer dependencies to encumber its usage. Hence, units in a loosely coupled system may be replaced with different implementations that provide the same services.

The concept of loose coupling may be applied to classes, interfaces, services, and data. The Enterprise Service Bus, and more specifically, ENI's Semantic Bus, are designed to promote loose coupling.

Loose coupling is typically achieved in APIs by using standard datatypes in parameters, and ensuring that a standard format is used in communication between units.

Loose coupling is associated with high cohesion, and vice versa.

4.1.9 High Cohesion

Cohesion [i.9] refers to how closely related the contents of a particular unit are. It answers the question "do these elements inside a unit belong together". It can be thought of as a measure of how well the elements of a unit serve the purpose of a unit.