Date: 13 August 2012

Essence – Kernel and Language for Software Engineering Methods

Revised Submission

In response to: Foundation for the Agile Creation and Enactment of Software Engineering Methods (FACESEM) RFP (OMG Document ad/2011-06-26)

OMG Document Number: ad/2012-08-15

Standard document URL: http://www.omg.org/cgi-bin/doc?ad/2012-08-15/PDF

Associated File(s)*: http://www.omg.org/cgi-bin/doc?ad/2012-08-15

Submission Team

OMG Submitters:

Fujitsu
Ivar Jacobson International AB
Model Driven Solutions

Supporting Organizations:

Florida Atlantic University
IICT-BAS
Impetus
International Business Machines Corporation
KnowGravity Inc.
KTH Royal Institute of Technology
Metamaxim Ltd.
PEM Systems
Stiftelsen SINTEF
University of Duisburg-Essen

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

LICENSES

PATENTS

GENERAL USE RESTRICTIONS

Table of Contents

Preface

OMG
Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.
OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.
More information on the OMG is available at http://www.omg.org/

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A Specifications Catalog is available from the OMG website at:
http://www.omg.org/technology/documents/spec_catalog.htm
Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM)

Platform Specific Model and Interface Specifications

- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*
Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

NOTE: Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

1 Scope

This document, entitled “Essence – Kernel and Language for Software Engineering Methods” (referred to herein as Essence, Version 1.0.), is submitted as a response to the OMG "Foundation for the Agile Creation and Enactment of Software Engineering Methods" (FACESEM) RFP (OMG Document ad/2011-06-26). It provides comprehensive definitions and descriptions of the kernel and the language for software engineering methods, which address the mandatory requirements set forth in FACESEM RFP.

The Kernel provides the common ground for defining software development practices. It includes the essential elements that are always prevalent in every software engineering endeavor, such as Requirements, Software System, Team and Work. These elements have states representing progress and health, so as the endeavor moves forward the states associated with these elements progress. The Kernel among other things helps practitioners (e.g., architects, designers, developers, testers, developers, requirements engineers, process engineers, project managers, etc.) compare methods and make better decisions about their practices.

The Kernel is described using the Language, which defines abstract syntax, dynamic semantics, graphic syntax and textual syntax. The Language supports composing two practices to form a new practice, and composing practices into a method, and the enactment of methods.

This document addresses the RFP mandatory requirements of the Kernel, the Language, and Practice in the following:

- It defines the Kernel and its organizations into three areas of concerns: Customer, Solution and Endeavor.
- It defines the Kernel Alphas (i.e., the essential things to work with), and Activity Spaces (i.e., the essential things to do).
- It describes the Language specification, Language elements and Language model.
- It defines Language Dynamic Semantics, Graphical Syntax and Textual Syntax.
- It describes examples of composing Practices into Methods and Enactment of Methods.

2 Conformance

This specification defines 4 classes of conformance:

1. Practice conformance (e.g., to claim that a published practice conforms to the Kernel)
2. Practice authoring tool conformance (create practices, compose practices/methods, export practices/methods)
3. Practice browsing tool conformance (load practices exported from authoring tools)
4. Tracking/endeavor management conformance (add alphas, define alphas, track alpha states, …)
    - o Kernel only
    - o Optional kernel extensions

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- Foundation for the Agile Creation and Enactment of Software Engineering Methods (FACESEM) RFP, OMG Document ad/2011-06-26, http://www.omg.org/cgi-bin/doc?ad/2011-06-26
- OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, OMG Document formal/2011-08-07, http://www.omg.org/spec/MOF/2.4.1/
- OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1, OMG Document formal/2011-

08-05, http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/

- Diagram Definition (DD), Version 1.0 - FTF Beta 2, OMG Document ptc/2011-07-13, http://www.omg.org/spec/DD/1.0/Beta2/
- Software & Systems Process Engineering Meta-Model Specification, Version 2.0, OMG Document formal/2008-04-01, http://www.omg.org/spec/SPEM/2.0/
- K. Schwaber and J. Sutherland, "The Scrum Guide", Scrum.org, October 2011. http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Activity

An activity defines one or more kinds of work items and gives guidance on how to perform these.

Activity space

A placeholder for something to be done in the software engineering endeavor. A placeholder may consist of zero to many activities.

Alpha

An essential element of the software engineering endeavor that is relevant to an assessment of the progress and health of the endeavor. Alpha is an acronym for an Abstract-Level Progress Health Attribute

Alpha association

An alpha association defines a relationship between two alphas.

Area of concern

Elements in kernels or practices may be divided into a collection of main areas of concern that a software engineering endeavor has to pay special attention to. All elements fall into at most one of these.

Check list item

A check list item is an item in a check list that needs to be verified in a state.

Competency

A characteristic of a stakeholder or team member that reflects the ability to do work.

A competency describes a capability to do a certain job. A competency defines a sequence of competency levels ranging from a minimum level of competency to a maximum level. Typically, the levels range from *0 – no competence* to *5 – expert*. (See Section 10.2.4.)

Constraints

Restrictions, policies, or regulatory requirements the team must comply with.

Enactment

The act of applying a method for some particular purpose, typically an endeavor.

Endeavor

An activity or set of activities directed towards a goal.

Invariant

An invariant is a proposition about an instance of a language element which is true if the instance is used in a language construct as intended by the specification.

Kernel

A kernel is a set of elements used to form a common ground for describing a software engineering endeavor.

Method

A method is a composition of practices forming a (at the desired level of abstraction) description of how an endeavor is performed. A team's method acts as a description of the team's way-of- working and provides help and guidance to the team as they perform their task. The running of a development effort is expressed by a used method instance. This instance holds instances of alphas, work products, activities, and the like that are the outcome from the real work performed in the development effort. The used method instance includes a reference to the defined method instance, which is selected as the method to be followed.

Opportunity

The set of circumstances that makes it appropriate to develop or change a software system.

Pattern

A pattern is a description of a structure in a practice.

Practice

A repeatable approach to doing something with a specific purpose in mind.

A practice provides a systematic and verifiable way of addressing a particular aspect of the work at hand. It has a clear goal expressed in terms of the results its application will achieve. It provides guidance to not only help and guide practitioners in what is to be done to achieve the goal but also to ensure that the goal is understood and to verify that it has been achieved. (See Section 10.2.1.11.)

Requirements

What the software system must do to address the opportunity and satisfy the stakeholders.

Role

A set of responsibilities.

Software system

A system made up of software, hardware, and data that provides its primary value by the execution of the software.

Stakeholders

The people, groups, or organizations who affect or are affected by a software system.

State

A state expresses a situation where some condition holds.

State Graph

A state graph is a directed graph of states with transitions between these states. It has a start state and may have a collection of end states.

Team

The group of people actively engaged in the development, maintenance, delivery and support of a specific software system.

Transition

A transition is a directed connection from one state in a state machine to a state in that state machine.

Way-of-working

The tailored set of practices and tools used by a team to guide and support their work.

Work

Work is defined as all mental and physical activities performed by the team to produce a software system.

Work item

A piece of work that should be done to complete the work. It has a concrete result and it leads to either a state change or a confirmation of the current state. Work item may or may not have any related activity.

5 Symbols and Abbreviations

5.1 Symbols

There are no symbols defined in this specification.

5.2 Abbreviations

- **Sub-alpha:** Subordinate alpha

6 Additional Information

6.1 Submitting Organizations

The following companies submitted this specification:

- Fujitsu
- Ivar Jacobson International AB
- Model Driven Solutions

6.2 Supporting Organizations

The following companies supported this specification:

- Florida Atlantic University
- IICT-BAS
- Impetus
- International Business Machines Corporation
- KnowGravity Inc.

- KTH Royal Institute of Technology
- Metamaxim Ltd.
- PEM Systems
- Stiftelsen SINTEF
- University of Duisburg-Essen

6.3 Submission Contacts

- Paul E. McMahon, PEM Systems, pemcmahon@aol.com
- Ian Michael Spence, Ivar Jacobson International AB, ispence@ivarjacobson.com
- Michael Striewe, University of Duisburg-Essen, michael.striewe@paluno.uni-due.de
- Ed Seidewitz, Model Driven Solutions, ed-s@modeldriven.com
- Brian Elvesæter, Stiftelsen SINTEF, brian.elvesater@sintef.no

6.4 Acknowledgements

The work is based on the Semat initiative incepted at the end of 2009, which was envisioned by Ivar Jacobson, along with the other two Semat advisors Bertrand Meyer and Richard Soley.

Among all the people who have worked as volunteers to make this submission possible, there are in particular a few people who have made significant contributions: Ivar Jacobson guides the work of this submission; Paul E. McMahon coordinates this submission; Ian Michael Spence leads the architecture of the Kernel and the Kernel specification; Michael Striewe leads the Language specification with technical leadership from Gunnar Övergaard on the metamodel, Stefan Bylund on the graphical syntax and Ashley McNeile on the dynamic semantics.

The following persons are members of the core team that have contributed to the content specification: Andrey A. Bayda, Arne-Jørgen Berre, Stefan Bylund, Bob Corrick, Dave Cuningham, Brian Elvesæter, Michael Goedicke, Shihong Huang, Ivar Jacobson, Mira Kajko-Mattsson, Prabhakar R. Karve, Bruce MacIsaac, Paul E. McMahon, Ashley McNeile, Winifred Menezes, Bob Palank, Ed Seidewitz, Ed Seymour, Ian Michael Spence, Michael Striewe and Gunnar Övergaard.

In addition, the following persons contributed valuable ideas and feedback that improved the content and the quality of the work behind this specification: Scott Ambler, Chris Armstrong, Jorn Bettin, Stefan Britts, Anders Caspar, Adriano Comai, Jorge Diaz-Herrera, Jean Marie Favre, Todd Fredrickson, Carlo Alberto Furia, Tom Gilb, Carson Holmes, Sylvia Ilieva, Capers Jones, Melir Page Jones, Mark Kennaley, Philippe Kruchten, Yeu Wen Mak, Tom McBride, Bertrand Meyer, Hiroshi Miyazaki, Martin Naedele, Jaime Pavlich-Mariscal, Jaana Nyfjord, Tom Rutt, Markus Schacher, Roly Stimson and Paul Szymkowiak.

6.5 Status of the Document

This document is a second revised specification for review and comment by OMG members.

6.6 Responses to RFP Requirements

See Annex A.

7 Overview of the Specification

7.1 Introduction to Essence

The work behind Essence is the Semat initiative[1], [2], [3] – Software Engineering Method and Theory – that was incepted at the end of 2009. Semat addresses the many issues that challenge the field of software engineering. For example, the reliance on fads and fashions, the lack of a theoretical basis, the abundance of unique methods that are hard to compare, the dearth of experimental evaluation and validation, and the gap between academic research and its practical application in industry.

Successfully developing software systems benefit from the application of effective methods and well-defined processes, as indicated in the RFP. Traditionally, a method definition is thought of as being instantiated, and the activities – created from the definition – are executed by practitioners (e.g., analysts, developers, testers, project leads) in some predefined order to get the result, specified by the definition. These software method engineering approaches are often considered by development teams as being too heavyweight and inflexible. The view – "the team is the computer, the process is the program" – is not suitable for creative work like software engineering that requires support for work, which is agile, trial-and-error based and collaboration intensive.

Essence defines a Kernel and a Language for software engineering method specification. They are scalable, extensible, and easy to use, and allow people to describe the essentials of their existing and future methods and practices so that they can be compared, evaluated, tailored, used, adapted, simulated and measured by practitioners as well as taught and researched by academics and researchers. The Kernel provides the common ground to among other things help practitioners to compare methods and make better decisions about their practices. One of the most important features is that the Kernel elements form the basis of a vocabulary – a map of the software engineering context. The map would be used as a base on top of which we can define and describe any method or practice in existence or foreseen in the near future. The Kernel should also be extensible to care for new technologies, new practices, new social working patterns, and new research. This is also an application of the principle of separation of concerns: separating the kernel elements from the specifics of the different methods.

The kernel elements are always prevalent in any software endeavors. They are what we already have (e.g. teams and work), what we already do (e.g. specify and implement), and what we already produce (e.g. software systems) when we develop software. An important goal is that the Kernel is small and light at its base but extensible to cover more advanced uses, such as dealing with life-, safety-, business-, mission-, and security-critical systems.

The Kernel and its elements are defined using a domain-specific language (the domain being practices for software development), which has a static base (syntax and well-formedness rules) to allow defining methods effectively, and with additional dynamic features (operational semantics) to enable usage, and adaption. In addition, the language is also used to define practices and methods.

Practices are described using the Kernel elements; they also allow a practice to be merged with other relevant practices to form a higher-level "method" or composed practice. The elements in the Kernel must be defined in a way that allows them to be extensible and tailorable supporting a wide variety of practices, methods, and development teams. The key concepts include:

- A Method is a composition of practices. Methods are dynamic and used. Methods are not just descriptions for developers to read, they are dynamic, supporting their day-to-day activities. This changes the conventional definition of a method. A method is not just a description of what is expected to be done, but a description of what is actually done.

[1] Software Engineering Method and Theory (Semat) website: www.semat.org
[2] Ivar Jacobson, Bertrand Meyer, and Richard Soley: "Call for Action: The Semat Initiative" Dr. Dobb's Journal December 10, 2009. Online at http://www.drdobbs.com/architecture-and-design/222001342
[3] Ivar Jacobson, Bertrand Meyer, and Richard Soley: "Software Engineering Method and Theory – A Vision Statement", online at http://www.semat.org/pub/Main/WebHome/SEMAT-vision.pdf

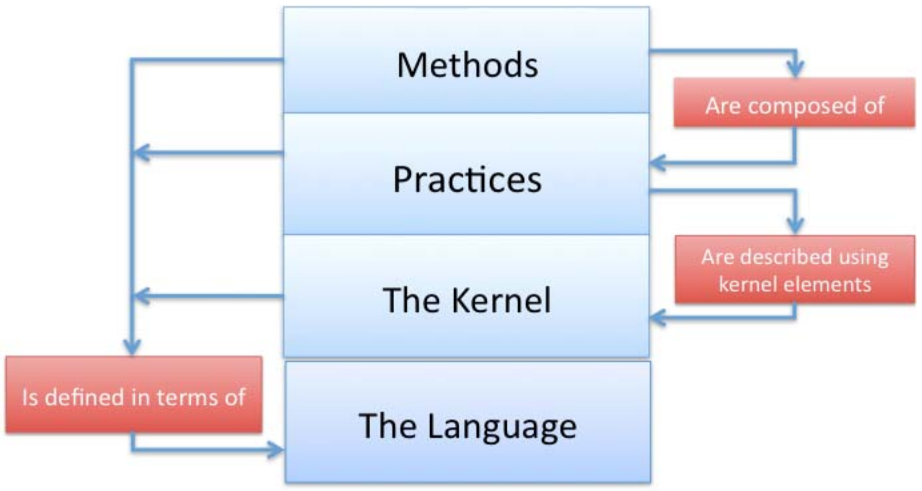


Figure 1 – Method architecture

- A Practice is a repeatable approach to doing something with a specific purpose in mind. A practice provides a systematic and verifiable way of addressing a particular aspect of the work at hand.
- The Kernel includes essential elements of software engineering.
- The Language is the domain-specific language to define methods, practices and the essential elements of the kernel.

The relationships among these concepts are depicted in **Error! Reference source not found.**[4]

The language design was driven by two main objectives: making methods visible to developers and making methods useful to developers. The first objective led to the definition of both textual and graphical syntax as well as to the development of a concept of views in the latter. This way, developers can represent methods in exactly the way that suits their purposes best. By providing both textual and graphical syntax, nobody is forced to use a graphical notation in situations where textual notation is easier to handle, and vice versa. By providing a concept of views, nobody is forced to show a complete graphical representation in situations where a partial graphical representation of a method is sufficient.

The second objective led to the definition of dynamic semantics for methods. This way, a method is more than a static definition of what to do, but an active guide for a team's way-of-working. At any point in time in a running software engineering endeavor, a method can be consulted and it returns advice on what to do next. Moreover, a method can be tweaked at any point in time and still returns (a possibly alternate) advice on what to do next for the same situation.

7.2 The Key Differentiators

The Essence work is built on the experiences and lessons learnt in the software development community. Some of the key differentiators set this work apart from what has been done in the past. These are the following[5]:

1. Finding the essence of software engineering and finding a way to embody that essence in a kernel enables us to build our knowledge on top of what we have known and learnt, and apply and reuse gained knowledge across different application domains and software systems of differing complexity.
2. Work with methods in an agile way that are as close to practitioners' practice as possible, so that they can evolve the methods and adapt them to their particular context.
3. Apply the principle of Separation of Concerns (SoC) that puts focus on the things that matter the most.

[4] Ivar Jacobson, Shihong Huang, Mira Kajko-Mattsson, Paul McMahon, Ed Seymour. "Semat - Three Year Vision" Programming and Computer Software 38(1): 1-12 (2012), Springer 2012. DOI: 10.1134/S0361768812010021.
[5] Ivar Jacobson, Pan-Wei Ng, Paul E. McMahon, Ian Spence, Svante Lidman. *The Essence of Software Engineering – Applying the Semat Kernel*, in preparation to be published

a. Focusing on what helps the least experienced developers over what helps the more experienced developers. This is motivated by the understanding that the majority of the development community is not interested in method descriptions but rather the use of the method.

b. Supporting practitioners over process engineers. This is motivated by the conviction that process engineers should work on what practitioners' need, based on the real work they must do on their software endeavor.

c. Emphasizing intuitive and concrete graphical syntax over formal semantics. This does not mean that the semantics is not as important nor as necessary. However, the description should be provided in a language that can be easily understood by the vast developer community whose interests are to quickly understand and use the language, rather than caring about the beauty of the language design. Hence, Essence pays extreme attention to syntax.

d. Focusing on method use over method definition. Most previous similar efforts have paid interest to method definition, i.e., how to capture methods. These efforts have not focused on how to support the use of a method in software endeavors. As a result, the methods became "shelf-ware" that are not relevant to practitioners who actually develop the software. This Essence proposal focuses on the use of methods so that developers themselves can take control of their own way of working and allow the method to evolve as their endeavor progresses.

For detailed descriptions of the Kernel and the Language please refer to Section 8 Kernel Specification and Section 10 Language Specification.

8 Kernel Specification

This section presents the specification for the Software Engineering Kernel. It begins with an overview of the kernel as a whole and its organization into the three areas of concern. This is followed by a description of each area of concern and its contents.

8.1 Overview

8.1.1 What is the Kernel?

The Software Engineering Kernel is a stripped-down, light-weight set of definitions that captures the essence of effective, scalable software engineering in a practice independent way.

The focus of the kernel is to define a common basis for the definition of software development practices, one that allows them to be defined and applied independently. The practices can then be mixed and matched to create specific software engineering methods tailored to the specific needs of a specific software engineering community, project, team or organization. The kernel has many benefits including:

- It allows you to apply as few or as many practices as you like.
- It allows you to easily capture your current practices in a reusable and extendable way.
- It allows you to evaluate your current practices against a technique neutral control framework.
- It allows you to align and compare your on-going work and methods to a common, technique neutral framework, and then to complement it with any missing critical practices or process elements.
- It allows you to start with a minimal method adding practices as the endeavor progresses and when you need them.

8.1.2 What is in the Kernel?

The kernel is described using a small subset of the Kernel Language. It is organized into three areas of concern, each containing a small number of:

- **Alphas** – representations of the essential things to work with. The Alphas provide descriptions of the kind of things that a team will manage, produce, and use in the process of developing, maintaining and supporting good software. They also act as the anchor for any additional sub-alphas and work products required by the software engineering practices.
- **Activity Spaces –** representations of the essential things to do. The Activity Spaces provide descriptions of the challenges a team faces when developing, maintaining and supporting software systems, and the kinds of things that the team will do to meet them.

To maintain its practice independence the kernel does not include any instances of the other language elements such as work products or activities. These only make sense within the context of a specific practice.

The best way to get an overview of the kernel as a whole is to look at the full set of Alphas and Activity Spaces and how they are related.

8.1.3 Organizing the Kernel

The Kernel is organized into three discrete areas of concern, each focusing on a specific aspect of software engineering. As shown in Figure 2, these are:

- **Customer** – This area of concern contains everything to do with the actual use and exploitation of the software system to be produced.
- **Solution** – This area of concern contains everything to do the specification and development of the software system.

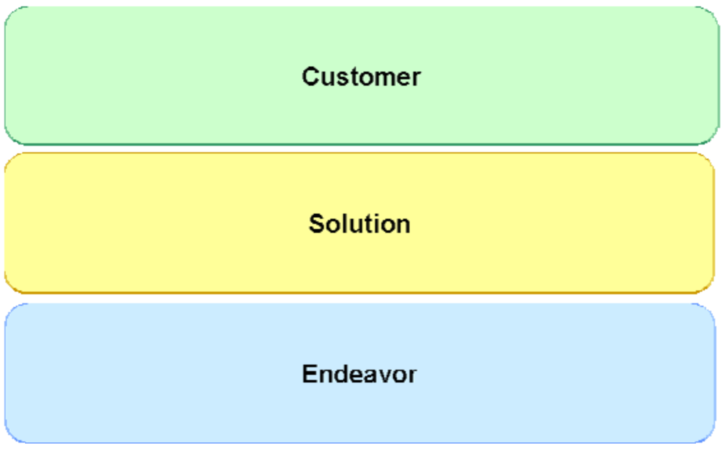


Figure 2 – The Three Areas of Concern

- **Endeavor** – This area of concern contains everything to do with the team, and the way that they approach their work.

Throughout the diagrams in the body of the kernel specification, the three areas of concern are distinguished with different color codes where green stands for customer, yellow for solution, and blue for endeavor. The colors will facilitate the understanding and tracking of which area of concern owns which Alphas and Activity Spaces.

8.1.4 Alphas: The Things to Work With

The kernel Alphas 1) capture the key concepts involved in software engineering, 2) allow the progress and health of any software engineering endeavor to be tracked and assessed, and 3) provide the common ground for the definition of software engineering methods and practices. The Alphas, their relationships and their owning areas of concern are shown in Figure 3.

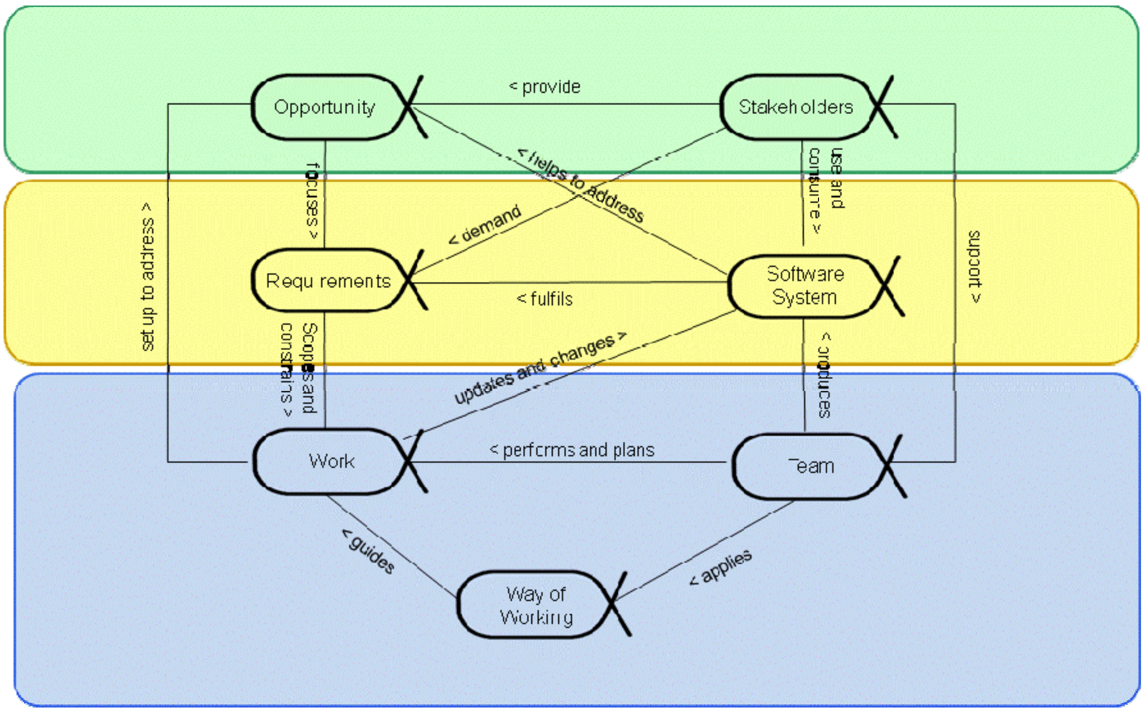


Figure 3 – The Kernel Alphas

In the **customer** area of concern the team needs to understand the stakeholders and the opportunity to be addressed:

1. **Opportunity**: The set of circumstances that makes it appropriate to develop or change a software system.

   The opportunity articulates the reason for the creation of the new, or changed, software system. It represents the team's shared understanding of the stakeholders' needs, and helps shape the requirements for the new software system by providing justification for its development.

2. **Stakeholders**: The people, groups, or organizations who affect or are affected by a software system.

   The stakeholders provide the opportunity and are the source of the requirements and funding for the software system. They must be involved throughout the software engineering endeavor to support the team and ensure that an acceptable software system is produced.

In the **solution** area of concern the team needs to establish a shared understanding of the requirements, and implement, build, test, deploy and support a software system that fulfills them:

3. **Requirements**: What the software system must do to address the opportunity and satisfy the stakeholders.

   It is important to discover what is needed from the software system, share this understanding among the stakeholders and the team members, and use it to drive the development and testing of the new system.

4. **Software System**: A system made up of software, hardware, and data that provides its primary value by the execution of the software.

   The primary product of any software engineering endeavor, a software system can be part of a larger software, hardware or business solution.

In the **endeavor** area of concern the team and its way-of-working have to be formed, and the work has to be done:

5. **Work**: Activity involving mental or physical effort done in order to achieve a result.

   In the context of software engineering, work is everything that the team does to meet the goals of producing a software system matching the requirements, and addressing the opportunity, presented by the customer. The work is guided by the practices that make up the team's way-of-working.

6. **Team**: The group of people actively engaged in the development, maintenance, delivery and support of a specific software system.

   The team plans and performs the work needed to update and change the software system.

7. **Way-of-Working**: The tailored set of practices and tools used by a team to guide and support their work.

   The team evolves their way of working alongside their understanding of their mission and their working environment. As their work proceeds they continually reflect on their way of working and adapt it as necessary to their current context.

8.1.5 Activity Spaces: The Things to Do

The kernel also provides a set of activity spaces that complement the Alphas to provide an activity based view of software engineering. The kernel activity spaces are shown in Figure 4.

In the **customer** area of concern the team has to understand the opportunity, and support and involve the stakeholders:

- **Explore Possibilities**: Explore the possibilities presented by the creation of a new or improved software system. This includes the analysis of the opportunity to be addressed and the identification of the stakeholders.
- **Understand Stakeholder Needs**: Engage with the stakeholders to understand their needs and ensure that the right results are produced. This includes identifying and working with the stakeholder representatives to progress the opportunity.
- **Ensure Stakeholder Satisfaction**: Share the results of the development work with the stakeholders to gain their acceptance of the system produced and verify that the opportunity has been successfully addressed.
- **Use the System**: Use the system in a live environment to benefit the stakeholders.

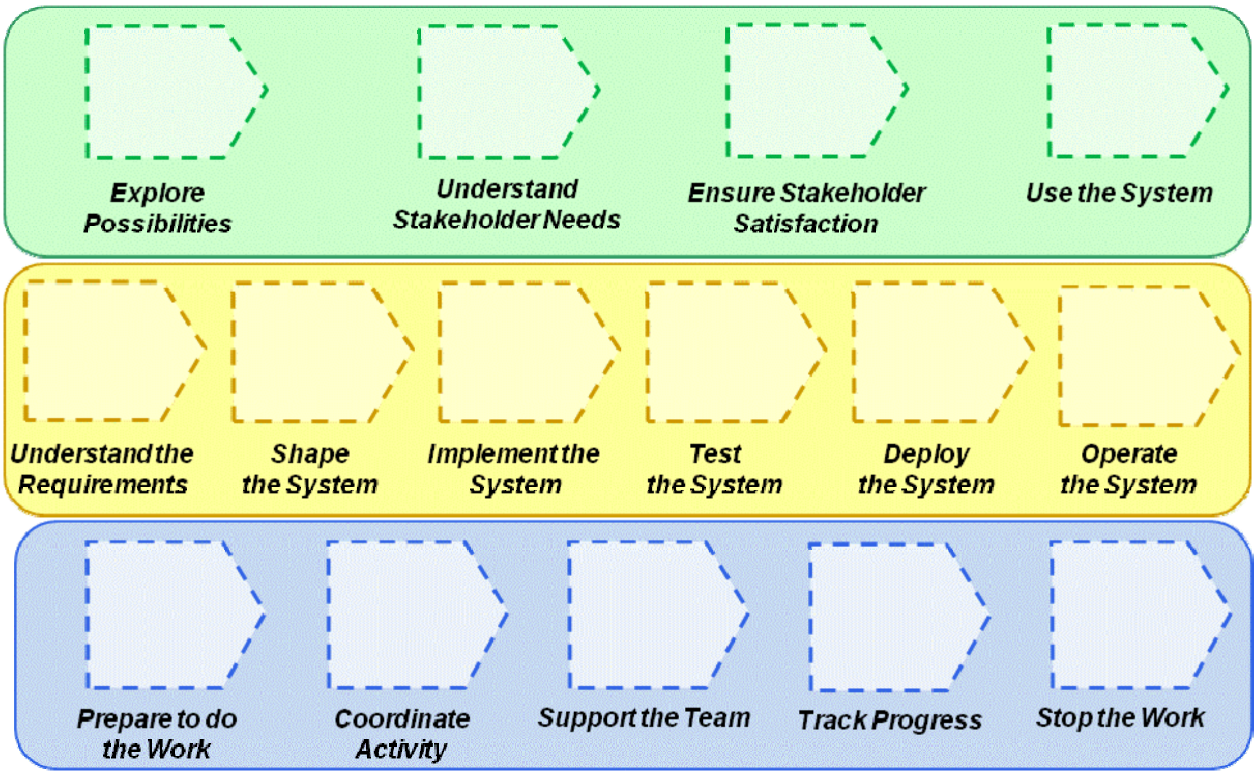


Figure 4 – The Kernel Activity Spaces

In the **solution** area of concern the team has to develop an appropriate solution to exploit the opportunity and satisfy the stakeholders:

- **Understand the Requirements**: Establish a shared understanding of what the system to be produced must do.
- **Shape the system**: Shape the system so that it is easy to develop, change and maintain, and can cope with current and expected future demands. This includes the overall design and architecting of the system to be produced.
- **Implement the System**: Build a system by implementing, testing and integrating one or more system elements. This includes bug fixing and unit testing
- **Test the System**: Verify that the system produced meets the stakeholders' requirements.
- **Deploy the System**: Take the tested system and make it available for use outside the development team.
- **Operate the System**: Support the use of the software system in the live environment.

In the **endeavor** area of concern the team has to be formed and progress the work in-line with the agreed way-of-working:

- **Prepare to do the Work**: Set up the team and its working environment. Understand and commit to completing the work.
- **Coordinate Activity**: Co-ordinate and direct the team's work. This includes all on-going planning and re-planning of the work, and adding any additional resources needed to complete the formation of the team.
- **Support the Team**: Help the team members to help themselves, collaborate and improve their way of working.
- **Track Progress**: Measure and assess the progress made by the team.
- **Stop the Work**: Shut-down the software engineering endeavor and the handover of the team's responsibilities.

8.2 The Customer Area of Concern

8.2.1 Introduction

This area of concern contains everything to do with the actual use and exploitation of the software system to be produced.

Software engineering always involves at least one customer for the software that it produces. The customer perspective must be integrated into the day-to-day work of the team to prevent an inappropriate solution from being produced.

8.2.2 Alphas

The customer area of concern contains the following Alphas:

- Stakeholders
- Opportunity

8.2.2.1 Stakeholders

Description

Stakeholders: The people, groups, or organizations who affect or are affected by a software system.

The stakeholders provide the opportunity, and are the source of the requirements for the software system. They are involved throughout the software engineering endeavor to support the team and ensure that an acceptable software system is produced.

States

Recognized	Stakeholders have been identified.
Represented	The mechanisms for involving the stakeholders are agreed and the stakeholder representatives have been appointed.
Involved	The stakeholder representatives are actively involved in the work and fulfilling their responsibilities.
In Agreement	The stakeholder representatives are in agreement.
Satisfied for Deployment	The minimal expectations of the stakeholder representatives have been achieved.
Satisfied in Use	The system has met or exceeds the minimal stakeholder expectations.

Associations

provide : Opportunity	Stakeholders provide Opportunity.
support : Team	Stakeholders support Team.
demand : Requirements	Stakeholders demand Requirements.
use and consume : Software System	Stakeholders use and consume Software System.

Justification: Why Stakeholders?

Stakeholders are critical to the success of the software system and the work done to produce it. Their input and feedback help shape the software engineering endeavor and the resulting software system.

Progressing the Stakeholders

During the development of a software system the stakeholders progress through several state changes. As shown in Figure 5, they are *recognized, represented, involved, in agreement, satisfied for deployment* and *satisfied in use*. These states focus on the involvement and satisfaction of the stakeholders, from their recognition as stakeholders through their representation in the development activities to their satisfaction with the use of the resulting software system. They communicate the progression of the relationship with the stakeholders who are either directly involved in the software engineering endeavor or support it by providing input and feedback.

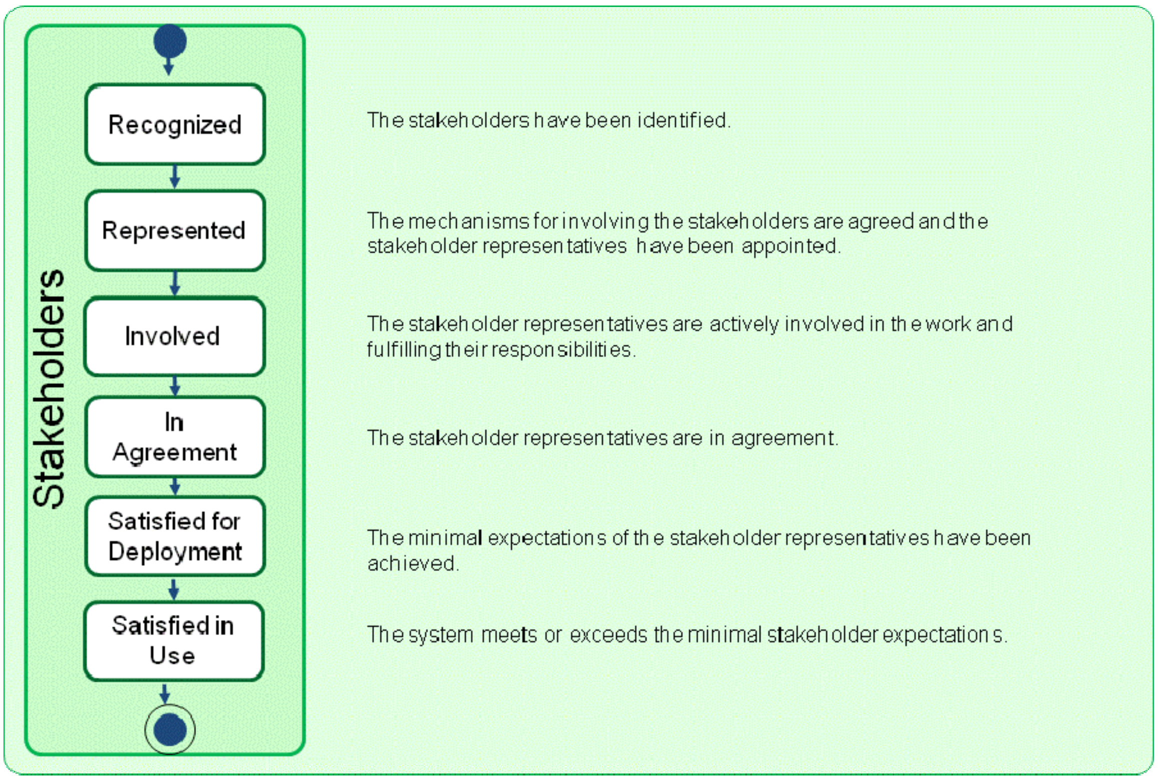

Stakeholders
Recognized
Represented
Involved
In Agreement
Satisfied for Deployment
Satisfied in Use
The stakeholders have been identified.
The mechanisms for involving the stakeholders are agreed and the stakeholder representatives have been appointed.
The stakeholder representatives are actively involved in the work and fulfilling their responsibilities.
The stakeholder representatives are in agreement.
The minimal expectations of the stakeholder representatives have been achieved.
The system meets or exceeds the minimal stakeholder expectations.

Figure 5 – The states of the Stakeholders

As indicated in Figure 5, the first thing to do is to make sure that the stakeholders affected by the proposed software system are recognized. This means that all the different groups of stakeholders that are, or will be, affected by the development and operation of the software system are identified.

The number and type of stakeholder groups to be identified can vary considerably from one system to another. For example the nature and complexity of the system and its target operating environment, and the nature and complexity of the development organization will both affect the number of stakeholder groups affected by the system.

It is not always possible to have all the stakeholder groups involved. Focus should be primarily on the ones that are critical to the success of the software engineering endeavor. It is these stakeholder groups that need to be directly involved in the work. Their selection depends on the level of impact they have on the success of the software system and the level of impact the software system has on them. The stakeholder groups that assure quality, fund, use, support and maintain the software system should always be identified.

It is not enough to determine which stakeholder groups need to be involved, they will also need to be actively represented. This means that there will be one or more stakeholder representatives selected to represent each stakeholder group, or in some cases one stakeholder representative selected to represent all stakeholder groups, and help the team. To make the contribution of the stakeholder representatives as effective as possible, they must know their roles and responsibilities within the software engineering endeavor. Without defining clear roles and responsibilities, the software engineering endeavor runs the risk that some of its important aspects may get unintentionally omitted or neglected.

Once the stakeholder representatives have been appointed, the represented state is achieved. Here, the stakeholder representatives take on their agreed to responsibilities and feel fully committed to helping the new software system to succeed. Acting as intermediaries between their respective stakeholder groups and the team, they are now granted authority to carry out their responsibilities on behalf of their respective stakeholder groups.

The team needs to make sure that the stakeholder representatives are actively involved in the development of the software system. Here, the stakeholder representatives assist in the software engineering endeavor in accordance with their responsibilities. They provide feedback and take part in decision making in a timely manner. In cases when changes need to be done to the software system, or when the stakeholder group they represent suggests changes, the stakeholder representatives make sure that the changes are relevant and promptly communicated to the team. No software engineering endeavor is fixed from the beginning. Its requirements are continuously evolving as the opportunity changes or new limitations are identified. This requires the stakeholder representatives to be actively involved throughout the development and to be responsive to all the changes affecting their stakeholder group.

It may not always be possible to meet all the expectations of all the stakeholders. Hence, compromises will have to be made. In the in agreement state the stakeholder representatives have identified and agreed upon a minimal set of expectations which have to be met before the system is deployed. These expectations will be reflected in the requirements agreed by the stakeholder representatives.

Throughout the development the stakeholder representatives provide feedback on the system's state from the perspective of their stakeholder groups. Once the minimal expectations of the stakeholder representatives have been achieved by the new software system they will confirm that it is ready for operational use and the satisfied for deployment state is achieved.

Finally, the stakeholders start to use the operational system and provide feedback on whether or not they are truly satisfied with what has been delivered. Achieving the satisfied in use state indicates that the new system has been successfully deployed and is delivering the expected benefits for all the stakeholder groups.

Understanding the current state of the stakeholders and how they are progressing towards being satisfied with the new system is a critical part of any software engineering endeavor.

Checking the progress of the Stakeholders

To help assess the state and progress of the stakeholders, the following checklists are provided:

Table 1 – Checklist for Stakeholders

State	Checklist
Recognized	All the different groups of stakeholders that are, or will be, affected by the development and operation of the software system are identified. There is agreement on the stakeholder groups to be represented. At a minimum, the stakeholders groups that fund, use, support, and maintain the system have been considered. The responsibilities of the stakeholder representatives have been defined.
Represented	The stakeholder representatives have agreed to take on their responsibilities. The stakeholder representatives are authorized to carry out their responsibilities. The collaboration approach among the stakeholder representatives has been agreed. The stakeholder representatives support and respect the team's way of working.
Involved	The stakeholder representatives assist the team in accordance with their responsibilities. The stakeholder representatives provide feedback and take part in decision making in a timely manner. The stakeholder representatives promptly communicate changes that are relevant for their stakeholder groups.
In Agreement	The stakeholder representatives have agreed upon their minimal expectations for the next deployment of the new system. The stakeholder representatives are happy with their involvement in the work. The stakeholder representatives agree that their input is valued by the team and treated with respect. The team members agree that their input is valued by the stakeholder representatives and treated with respect. The stakeholder representatives agree with how their different priorities and perspectives are

	being balanced to provide a clear direction for the team.
Satisfied for Deployment	The stakeholder representatives provide feedback on the system from their stakeholder group perspective. The stakeholder representatives confirm that the system is ready for deployment.
Satisfied in Use	Stakeholders are using the new system and providing feedback on their experiences. The stakeholders confirm that the new system meets their expectations.

8.2.2.2 Opportunity

Description

Opportunity: The set of circumstances that makes it appropriate to develop or change a software system.

The opportunity articulates the reason for the creation of the new, or changed, software system. It represents the team's shared understanding of the stakeholders' needs, and helps shape the requirements for the new software system by providing justification for its development.

States

Identified	A commercial, social or business opportunity has been identified that could be addressed by a software-based solution.
Solution Needed	The need for a software-based solution has been confirmed.
Value Established	The value of a successful solution has been established.
Viable	It is agreed that a solution can be produced quickly and cheaply enough to successfully address the opportunity.
Addressed	A solution has been produced that demonstrably addresses the opportunity.
Benefit Accrued	The operational use or sale of the solution is creating tangible benefits.

Associations

focuses : Requirements	Opportunity focuses Requirements.

Justification: Why Opportunity?

Most software engineering work is initiated by the stakeholders that own and use the software system. Their inspiration is usually some combination of problems, suggestions and directives, which taken together provide the development team with an opportunity to create a new or improved software system. Occasionally it is the development team itself that originates the opportunity that they must then sell to the other stakeholders to get funding and support. In many cases the software system only provides part of the solution needed to exploit the opportunity and the development team must co-ordinate their work with other teams to ensure that they actually deliver a useful, and deployable system.

In all cases understanding the opportunity is an essential part of software engineering, as it enables the team to:

- Identify and motivate their stakeholders.
- Understand the value that the software system offers to the stakeholders.
- Understand why the software system is being developed.
- Understand how the success of the deployment of the software system will be judged.
- Ensure that the software system effectively addresses the needs of all the stakeholders.

It is the opportunity that unites the stakeholders and provides the motivation for producing a new or updated software system. It is by understanding the opportunity that you can identify the value, and the desired outcome that the stakeholders hope to realize from the use of the software system either alone or as part of a broader business, or technical solution.

Progressing the Opportunity

During the development of a software system the opportunity progresses through several state changes. As presented in Figure 6, these are *identified*, *solution needed*, *value established*, *viable*, *addressed*, and *benefit accrued*. These states indicate significant points in the team's progression of the opportunity from the initial formulation of an idea to use a software system through to the accrual of benefit from its use. They indicate (1) when the opportunity is first identified, (2) when the opportunity has been analyzed and it has been confirmed that a solution is needed, (3) when the opportunity's value is established and the desired outcomes required of the solution are clear, (4) when enough is known about the cost of creating and using the proposed solution that it is clear that the pursuit of the opportunity is viable, (5) when a solution is available that demonstrably shows that the opportunity has been addressed, and finally (6) when benefit has been accrued from the use of the resulting solution.

As shown in Figure 6, the opportunity is first identified. The opportunity could be to entertain somebody, learn something, make some money, or even to change the world. Regardless of the kind of opportunity presented, if it is not understood by the team it is unlikely that they will produce an appropriate software system. For software engineering endeavors the opportunity is usually identified by the stakeholders that own and use the software system, and typically takes the form of an idea for a way to improve the current way of doing something, increase market share or apply a new or innovative technology.

Different stakeholders will see the opportunity in different ways, and they will be looking for different results from any software system produced to address it. It is important that the different stakeholder perspectives are understood and used

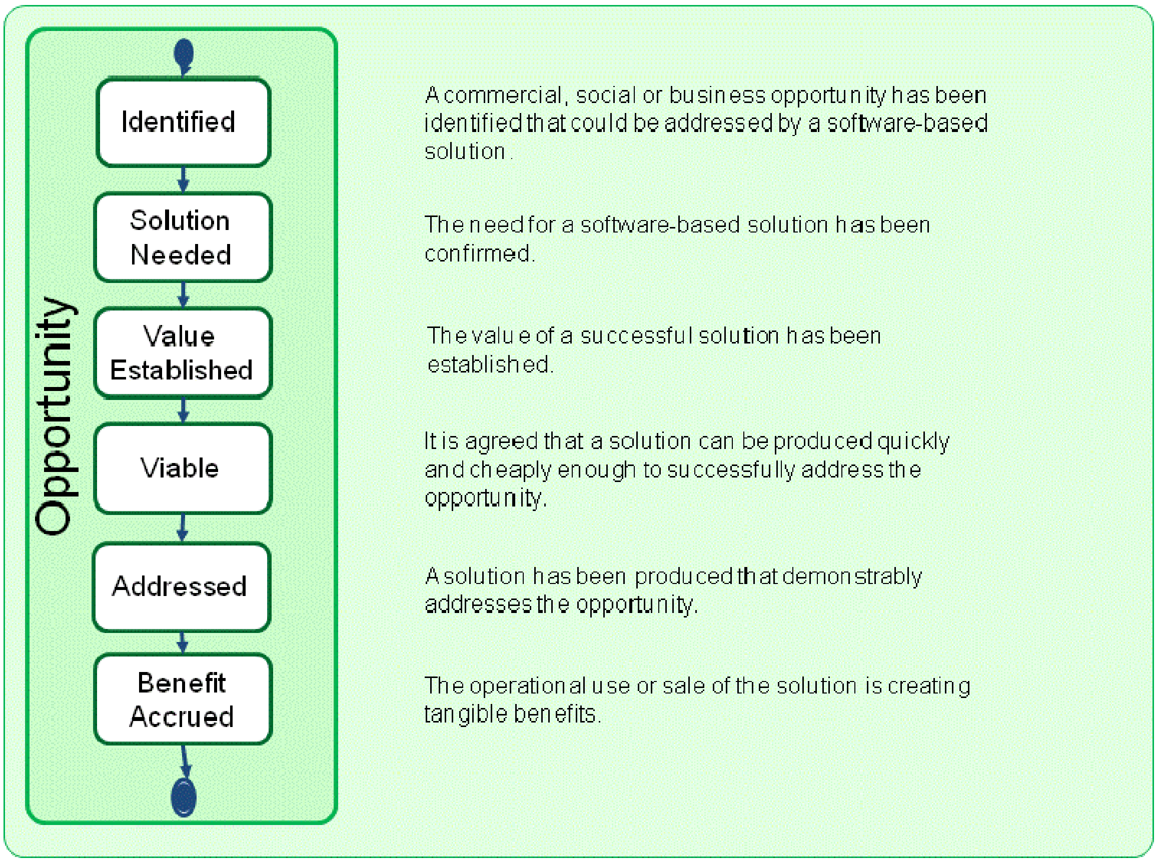

Opportunity
Identified
A commercial, social or business opportunity has been identified that could be addressed by a software-based solution.
Solution Needed
The need for a software-based solution has been confirmed.
Value Established
The value of a successful solution has been established.
Viable
It is agreed that a solution can be produced quickly and cheaply enough to successfully address the opportunity.
Addressed
A solution has been produced that demonstrably addresses the opportunity.
Benefit Accrued
The operational use or sale of the solution is creating tangible benefits.

Figure 6 – The states of the Opportunity

to increase the team’s understanding of the opportunity. Analyzing the opportunity to understand the stakeholder’s needs and any underlying problems is essential to ensure that an appropriate system is produced and a satisfactory return-on-investment is generated.

Once the opportunity has been analyzed, and it has been agreed that a software-based solution is needed, it is possible to determine the value that the solution is expected to generate. Progressing the opportunity to value established is an important step in determining whether or not to proceed with work to address the opportunity as it means that the prize is clear to everyone involved.

The next step is to establish the viability of the opportunity. An opportunity is viable when a solution can be envisaged that it is feasible to develop and deploy within acceptable time and cost constraints. Although addressing the opportunity may be a very valuable thing to do it is probably not a good idea if the resources expended will be greater than the benefits accrued.

Once it has been agreed that the opportunity is viable then the team can be confident that a software system can be produced that will not just address the opportunity but will be acceptable to all of the stakeholders. As releases of the software system become available their viability must be continuously checked to ensure that they meet the needs of the stakeholders. After a suitable software system has been made available then, as far as the development team is concerned, the opportunity has been addressed. It is now up to the users of the system to actually use it to generate value and make sure that for this opportunity there is benefit accrued.

It is important that the team understands the current state of the opportunity so that they can ensure that an appropriate software system is developed, one that will satisfy the stakeholders and result in a tangible benefit being accrued.

Checking the Progress of the Opportunity

To help assess the state of the opportunity and the progress being made towards its successful exploitation, the following checklists are provided:

Table 2 – Checklist for Opportunity

State	Checklist
Identified	An idea for a way of improving current ways of working, increasing market share or applying a new or innovative software system has been identified. At least one of the stakeholders wishes to make an investment in better understanding the opportunity and the value associated with addressing it. The other stakeholders who share the opportunity have been identified.
Solution Needed	The stakeholders in the opportunity and the proposed solution have been identified. The stakeholders' needs that generate the opportunity have been established. Any underlying problems and their root causes have been identified. It has been confirmed that a software-based solution is needed. At least one software-based solution has been proposed.
Value Established	The value of addressing the opportunity has been quantified either in absolute terms or in returns or savings per time period (e.g. per annum). The impact of the solution on the stakeholders is understood. The value that the software system offers to the stakeholders that fund and use the software system is understood. The success criteria by which the deployment of the software system is to be judged are clear. The desired outcomes required of the solution are clear and quantified.

Viable	A solution has been outlined. The indications are that the solution can be developed and deployed within constraints. The risks associated with the solution are acceptable and manageable. The indicative (ball-park) costs of the solution are less than the anticipated value of the opportunity. The reasons for the development of a software-based solution are understood by all members of the team. It is clear that the pursuit of the opportunity is viable.
Addressed	A usable system that demonstrably addresses the opportunity is available. The stakeholders agree that the available solution is worth deploying. The stakeholders are satisfied that the solution produced addresses the opportunity.
Benefit Accrued	The solution has started to accrue benefits for the stakeholders. The return-on-investment profile is at least as good as anticipated.

8.2.3 Activity Spaces

The customer area of concern contains four activity spaces that cover the discovery of the opportunity and the involvement of the stakeholders:

8.2.3.1 Explore Possibilities

Description

Explore the possibilities presented by the creation of a new or improved software system. This includes the analysis of the opportunity to be addressed and the identification of the stakeholders.

Explore possibilities to:

- Enable the right stakeholders to be involved.
- Understand the stakeholders’ needs.
- Identify opportunities for the use of the software system.
- Understand why the software system is needed.
- Establish the value offered by the software system.

Input: None
Output: Stakeholders, Opportunity
Completion Criteria: Stakeholders::Recognized, Opportunity:: Identified, Opportunity::Solution Needed, Opportunity::Value Established.

8.2.3.2 Understand Stakeholder Needs

Description

Engage with the stakeholders to understand their needs and ensure that the right results are produced. This includes identifying and working with the stakeholder representatives to progress the opportunity.

Understand stakeholder needs to:

- Ensure the right solution is created.

- Align expectations.
- Collect feedback and generate input.
- Ensure that the solution produced provides benefit to the stakeholders.

Input: Stakeholders, Opportunity, Requirements, Software System
Output: Stakeholders, Opportunity
Completion Criteria: Stakeholders::Represented, Stakeholders::Involved, Stakeholders::In Agreement, **Opportunity**:Viable

8.2.3.3 Ensure Stakeholder Satisfaction

Description

Share the results of the development work with the stakeholders to gain their acceptance of the system produced and verify that the opportunity has been successfully addressed.

Ensure the satisfaction of the stakeholders to:

- Get approval for the deployment of the system.
- Validate that the system is of benefit to the stakeholders.
- Validate that the system is acceptable to the stakeholders.
- Independently verify that the system delivered is the one required.
- Confirm the expected benefit that the system will provide.

Input: Stakeholders, Opportunity, Requirements, Software System
Output: Stakeholders, Opportunity
Completion Criteria: Stakeholders::Satisfied for Deployment, Opportunity::Addressed

8.2.3.4 Use the System

Description

Use the system in a live environment to benefit the stakeholders.

Use the system to:

- Generate measurable benefits.
- Gather feedback from the use of the system.
- Confirm that the system meets the expectations of the stakeholders.
- Establish the return-on-investment for the system.

Input: Stakeholders, Opportunity, Requirements, Software System
Output: Stakeholders, Opportunity
Completion Criteria: Stakeholders::Satisfied in Use, Opportunity::Benefit Accrued

8.3 The Solution Area of Concern

8.3.1 Introduction

This area of concern covers everything to do with the specification and development of the software system.

The goal of software engineering is to develop working software as part of the solution to some problem. Any method adopted must describe a set of practices to help the team produce good quality software in a productive and collaborative fashion.

8.3.2 Alphas

The solution area of concern contains the following Alphas:

- Requirements
- Software System

8.3.2.1 Requirements

Description

Requirements: What the software system must do to address the opportunity and satisfy the stakeholders.

It is important to discover what is needed from the software system, share this understanding among the stakeholders and the team members, and use it to drive the development and testing of the new system.

States

Conceived	The need for a new system has been agreed.
Bounded	The purpose and theme of the new system are clear.
Coherent	The requirements provide a coherent description of the essential characteristics of the new system.
Acceptable	The requirements describe a system that is acceptable to the stakeholders.
Addressed	Enough of the requirements have been addressed to satisfy the need for a new system in a way that is acceptable to the stakeholders.
Fulfilled	The requirements that have been addressed fully satisfy the need for a new system.

Associations

scopes and constrains : Work	The Requirements scope and constrain the Work.

Justification: Why Requirements?

The requirements capture what the stakeholders want from the system. They define what the system must do, but not necessarily how it must do it. They describe the value the system will provide by addressing the opportunity and how the opportunity will be pursued by the production of a new software system. They also scope and constrain the work by defining what needs to be achieved.

The requirements are captured as a set of requirement items. The requirement items can be communicated and recorded in various forms and at various levels of detail. They may be communicated explicitly as a set of extensive requirements documents or more tacitly in the form of conversations and brain-storming sessions. The requirement items themselves are always documented and tracked. The documentation can take many forms and be as brief as a one-line user story or as comprehensive as a use case.

As the development of the system proceeds, the requirements evolve and are constantly re-prioritized and adjusted to reflect the changing needs of the stakeholders. Much that is implicit at first is made explicit later by adding more detailed requirement items such as well-defined quality characteristics and test cases. This allows the requirements to act as a verifiable specification for the software system. Regardless of how the requirement items are captured it is essential that the software system produced can be shown to successfully fulfill the requirements. This is why requirements play such an essential role in the testing of the system. As well as providing a definition of what needs to be achieved, they also allow tracking of what has been achieved. As the testing of each requirement item is completed it can be individually checked off as done, and the requirements as a whole can be looked at to see if the system produced sufficiently fulfils the requirements and whether or not work on the system is finished.

It is important that the overall state of the requirements is understood as well as the state of the individual requirement items. If the overall state of the requirements is not understood then it will be impossible to 1) tell when the system is finished, and 2) judge whether or not an individual requirement item is a requirement for this system or another system.

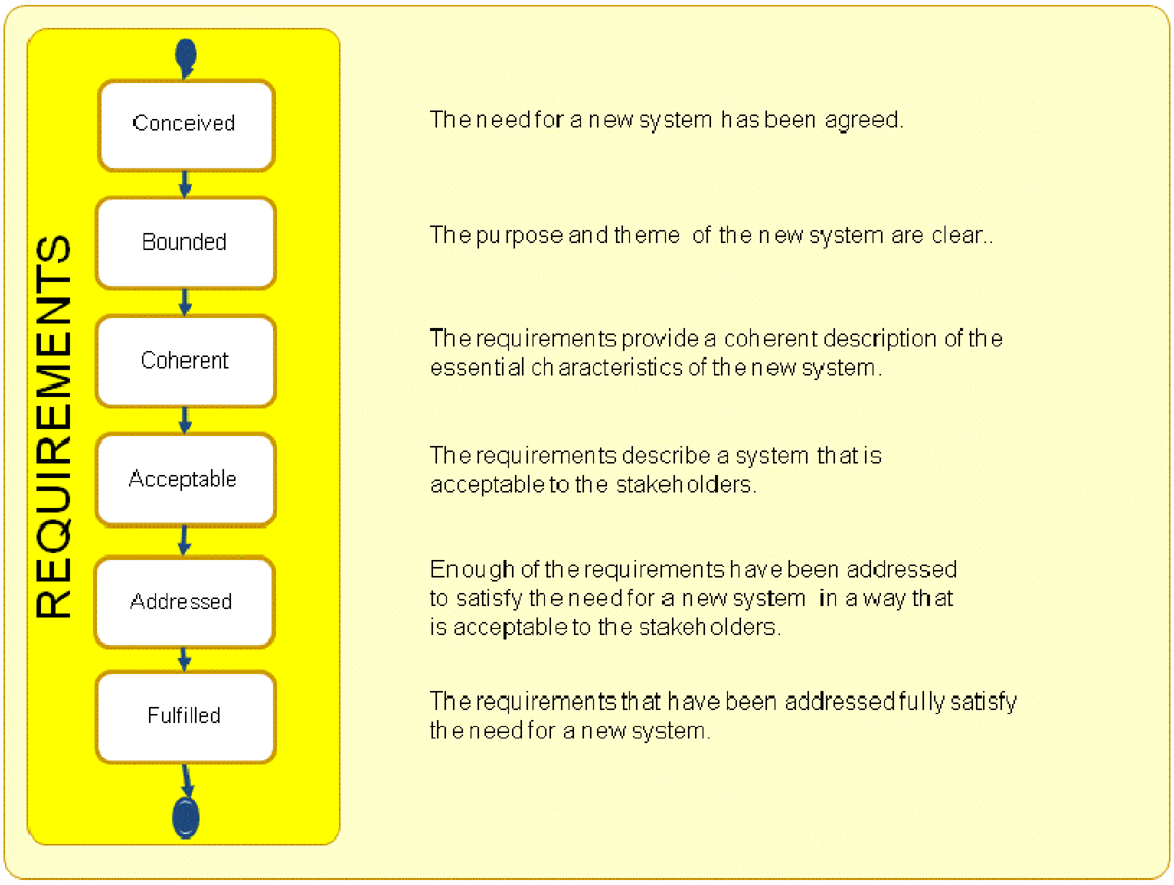

REQUIREMENTS
Conceived
Bounded
Coherent
Acceptable
Addressed
Fulfilled
The need for a new system has been agreed.
The purpose and theme of the new system are clear..
The requirements provide a coherent description of the essential characteristics of the new system.
The requirements describe a system that is acceptable to the stakeholders.
Enough of the requirements have been addressed to satisfy the need for a new system in a way that is acceptable to the stakeholders.
The requirements that have been addressed fully satisfy the need for a new system.

Figure 7 – The states of the Requirements

Progressing the Requirements

During the development of a software system the requirements progress through several state changes. As shown in Figure 7, they are *conceived, bounded, coherent, acceptable, addressed*, and *fulfilled*. These states focus on the evolution of the team's understanding of what the proposed system must do, from the conception of a new set of requirements as an initial idea for a new software system through their development to their fulfillment by the provision of a usable software system.

As shown in Figure 7, the requirements start in the conceived state when the need for a new software system has been agreed. The stakeholders can hold differing views on the overall meaning of the requirements. However, they all agree that there is a need for a new software system and a clear opportunity to be pursued.

Before too much time is spent collecting and detailing the individual requirement items the requirements as a whole must be bounded. To bound the requirements, the overall scope of the new system, the aspects of the opportunity to be addressed, and the mechanisms for managing and accepting new or changed requirement items all need to be established. In the bounded state there may still be inconsistencies or ambiguities between the individual requirement items. However, the stakeholders now have a shared understanding of the purpose of the new system and can tell whether or not a request qualifies as a requirement item. They also understand the mechanisms to be used to evolve the requirement items and remove the inconsistencies. Once the requirements are bounded there is a shared understanding of the scope of the new system and it is safe to start implementing the most important requirement items.

Further elicitation, refinement, analysis, negotiation, demonstration and review of the individual requirement items leads to a coherent set of requirements, one that clearly defines the essential characteristics of the new system. The requirement items continue to evolve as more is learnt about the new system and its impact on its stakeholders and environment. No matter how much the requirement items change, it is essential that they stay within the bounds of the original concept and that they remain coherent at all times.

The continued evolution of the requirements leads to the capture of an acceptable set of requirements, one that defines a system that will be acceptable to the stakeholders as, at least, an initial solution. The requirements may only describe a partial solution; however the solution described is of sufficient value that the stakeholders would accept it for operational use. The number of requirement items that need to be agreed for the requirements to be acceptable to the stakeholders can vary from one to many. When changing a mature system it may be acceptable to just address one important requirement item. When building a replacement system a large number of requirement items will need to be addressed.

As the individual requirement items are implemented and a usable system is evolved, there will come a time when enough requirements have been implemented for the new system to be worth releasing and using. In the addressed state the amount of requirements that have been addressed is sufficient for the resulting system to provide clear value to the stakeholders. If the resulting system provides a complete solution then the requirements may advance immediately to the fulfilled state.

Usually, when the addressed state is achieved the resulting system provides a valuable but incomplete solution. To fully address the opportunity, additional requirement items may have to be implemented. The shortfall may be because an incremental approach to the delivery of the system was selected, or because the missing requirements were difficult to identify before the system was made available for use.

In the fulfilled state enough of the requirement items have been implemented for the stakeholders to agree that the resulting system fully satisfies the need for a new system, and that there are no outstanding requirement items preventing the system from being considered complete.

Understanding the current and desired state of the requirements can help everyone understand what the system needs to do and how close to complete it is.

Checking the Progress of the Requirements

To help assess the state of the requirements and the progress being made towards their successful conclusion, the following checklists are provided:

Table 3 – Checklist for Requirements

State	Checklist
Conceived	The initial set of stakeholders agrees that a system is to be produced. The stakeholders that will use the new system are identified. The stakeholders that will fund the initial work on the new system are identified. There is a clear opportunity for the new system to address.
Bounded	The stakeholders involved in developing the new system are identified. The stakeholders agree on the purpose of the new system. It is clear what success is for the new system. The stakeholders have a shared understanding of the extent of the proposed solution. The way the requirements will be described is agreed upon. The mechanisms for managing the requirements are in place. The prioritization scheme is clear. Constraints are identified and considered. Assumptions are clearly stated.
Coherent	The requirements are captured and shared with the team and the stakeholders. The origin of the requirements is clear. The rationale behind the requirements is clear.

	Conflicting requirements are identified and attended to. The requirements communicate the essential characteristics of the system to be delivered. The most important usage scenarios for the system can be explained. The priority of the requirements is clear. The impact of implementing the requirements is understood. The team understands what has to be delivered and agrees to deliver it.
Acceptable	The stakeholders accept that the requirements describe an acceptable solution. The rate of change to the agreed requirements is relatively low and under control. The value provided by implementing the requirements is clear. The parts of the opportunity satisfied by the requirements are clear.
Addressed	Enough of the requirements are addressed for the resulting system to be acceptable to the stakeholders. The stakeholders accept the requirements as accurately reflecting what the system does and does not do. The set of requirement items implemented provide clear value to the stakeholders. The system implementing the requirements is accepted by the stakeholders as worth making operational.
Fulfilled	The stakeholders accept the requirements as accurately capturing what they require to fully satisfy the need for a new system. There are no outstanding requirement items preventing the system from being accepted as fully satisfying the requirements. The system is accepted by the stakeholders as fully satisfying the requirements.

8.3.2.2 Software System

Description

Software System: A system made up of software, hardware, and data that provides its primary value by the execution of the software.

A software system can be part of a larger software, hardware, business or social solution.

States

Architecture Selected	An architecture has been selected that addresses the key technical risks and any applicable organizational constraints.
Demonstrable	An executable version of the system is available that demonstrates the architecture is fit for purpose and supports functional and non-functional testing.
Usable	The system is usable and demonstrates all of the quality characteristics of an operational system.
Ready	The system (as a whole) has been accepted for deployment in a live environment.
Operational	The system is in use in a live environment.
Retired	The system is no longer supported.

Associations

helps to address : Opportunity	Software System helps to address Opportunity.
fulfills : Requirements	Software Systems fulfills Requirements.

Justification: Why Software System?

Essence uses the term software system rather than software because software engineering results in more than just a piece of software. Whilst the value may well come from the software, a working software system depends on the combination of software, hardware and data to fulfill the requirements.

Progressing the Software System

The life-cycle of a software system is hard to define as there can be many releases of a software system. These releases can be worked on and used in parallel. For example one team can be working on the development of release 3, whilst another team is making small changes to release 2, and a third team is providing support for those people still using release 1. If we treat this software system as one entity what state is it in?

To keep things simple, Essence treats each major release as a separate software system; one that is built, released, updated, and eventually retired. A major release encompasses significant changes to the purpose, usage, or architecture of a software system. It can encompass many minor releases including internal releases produced for testing purposes, and external releases produced to support incremental delivery or bug fixes. In the example above the second team would be producing a series of minor releases (2.1, 2.2, 2.3, etc.) of their software system to allow the delivery of their small changes.

During its development a software system progresses through several state changes. As shown in Figure 8, they are *architecture selected, demonstrable, usable, ready, operational* and *retired*. These states provide points of stability on a software system's journey from its conception to its eventual retirement indicating (1) when the architecture is selected,

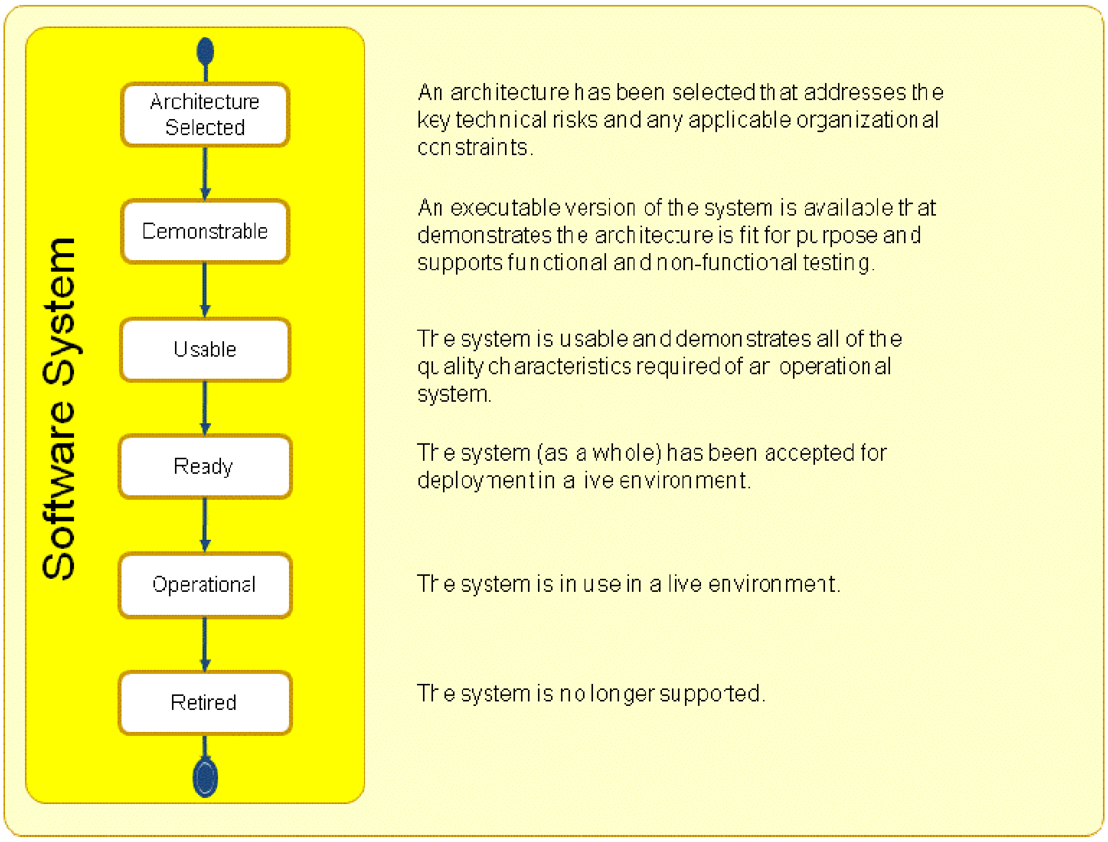

Software System
Architecture Selected
Demonstrable
Usable
Ready
Operational
Retired
An architecture has been selected that addresses the key technical risks and any applicable organizational constraints.
An executable version of the system is available that demonstrates the architecture is fit for purpose and supports functional and non-functional testing.
The system is usable and demonstrates all of the quality characteristics required of an operational system.
The system (as a whole) has been accepted for deployment in a live environment.
The system is in use in a live environment.
The system is no longer supported.

Figure 8 – The states of the Software System

(2) when a demonstrable system is produced to prove the architecture and enable testing to start, (3) when the system is extended and improved so that it becomes usable, (4) when the usable system is enhanced until it is accepted as ready for deployment, (5) when the system is made available to the stakeholders who use it and made operational, and finally, (6) when the system itself is retired and its support is withdrawn. These states can be applied to the initial release of the software system or any subsequent modification or replacement.

As indicated in Figure 8, the first thing to do for any major software system release is to make sure that there is an appropriate architecture available; one that complies with any applicable organizational constraints and addresses the key technical risks facing the new system. Achieving this may require the creation of a brand new architecture, the modification of an existing architecture, the selection of an existing architecture, or the simple re-use of whatever is already in place. Regardless of the approach taken, the result is that the system progresses to the architecture selected state.

Once the architecture had been selected, it must be shown to be fit-for-purpose by building and testing a demonstrable version of the system. It is not sufficient to just present a set of rolling screen-shots or a stand-alone version of a multi-user system. The system needs to be truly demonstrable exercising all of the significant characteristics of the selected architecture. It must also be capable of supporting both functional and non-functional testing.

The demonstrable system is then evolved to become usable by adding more functionality, and fixing defects. Once the system has achieved the usable state, it has all the qualities desired of an operational system. If it implements a sufficient amount of the requirements, if it provides sufficient business value, and if there is an appropriate window of opportunity for its deployment, then it can be considered to be ready for operational use.

Although, a useable system has the potential to be an operational system, there are still a few essential steps to be performed before it is ready. The system has to be accepted for use by the stakeholders, and it has to be prepared for deployment in the live environment. In this state, the system is typically supplemented with installation guidance, training materials and actual training for system operation.

The system is made operational when it is installed for real use within the live environment. It is now being used to generate value and provide benefit to its stakeholders.

Even after the software system has been made operational, development work can still continue. This may be as part of the plans for the incremental delivery of the system or, as is more common, a response to defects and problems occurring during the deployment and operation of the system. Support and maintenance continue until the software system is retired and its support is withdrawn. This may be because 1) the software system has been completely replaced by a later generation, 2) the software system no longer has any users or, 3) it does not make business sense to continue to support it.

During the development of a major release many minor releases are often produced. For example, many teams using an iterative approach produce a new release during every iteration whilst they keep their software system continuously in a usable, and therefore potentially shippable, state. It is then the stakeholder representatives who decide whether it is ready to be made operational. Obviously, this approach is not always possible, particularly if major architectural changes are required as these often render the system unusable for a significant period of time.

Understanding the current and desired states of a software system helps everyone understand when a system is ready, what kinds of changes can be realistically made to the system, and what kinds of work should be left to a later generation of the software system.

Checking the Progress of the Software System

To help assess the state of a software system and the progress being made towards its successful operation, the following checklist items are provided:

Table 4 – Checklist for Software System

State	Checklist
Architecture Selected	The criteria to be used when selecting the architecture have been agreed on. Hardware platforms have been identified.

	Programming languages and technologies to be used have been selected. System boundary is known. Significant decisions about the organization of the system have been made. Buy, build and reuse decisions have been made.
Demonstrable	Key architectural characteristics have been demonstrated. The system can be exercised and its performance can be measured. Critical hardware configurations have been demonstrated. Critical interfaces have been demonstrated. The integration with other existing systems has been demonstrated. The relevant stakeholders agree that the demonstrated architecture is appropriate.
Usable	The system can be operated by stakeholders who use it. The functionality provided by the system has been tested. The performance of the system is acceptable to the stakeholders. Defect levels are acceptable to the stakeholders. The system is fully documented. Release content is known. The added value provided by the system is clear.
Ready	Installation and other user documentation are available. The stakeholder representatives accept the system as fit-for-purpose. The stakeholder representatives want to make the system operational. Operational support is in place.
Operational	The system has been made available to the stakeholders intended to use it. At least one example of the system is fully operational. The system is fully supported to the agreed service levels.
Retired	The system has been replaced or discontinued. The system is no longer supported. There are no “official” stakeholders who still use the system. Updates to the system will no longer be produced.

8.3.3 Activity Spaces

The solution area of concern contains six activity spaces that cover the capturing of the requirements and the development of the software system.

8.3.3.1 Understand the Requirements

Description

Establish a shared understanding of what the system to be produced must do.

Understand the requirements to:

- Scope the system.
- Understand how the system will generate value.
- Agree on what the system will do.
- Identify specific ways of using and testing the system.
- Drive the development of the system.

Completion Criteria: Requirements::Conceived, Requirements::Bounded, Requirements::Coherent
Input: Stakeholders, Opportunity, Requirements, Software System, Work, Way-of-Working
Output: Requirements

8.3.3.2 Shape the System

Description

Shape the system so that it is easy to develop, change and maintain, and can cope with current and expected future demands. This includes the overall design and architecting of the system to be produced.

Shape the system to:

- Structure the system and identify the key system elements.
- Assign requirements to elements of the system.
- Ensure that the architecture is suitably robust and flexible.

Completion Criteria: Requirements::Sufficient, Software System::Architecture Selected
Input: Stakeholders, Opportunity, Requirements, Software System, Work, Way-of-Working
Output: Requirements, Software System

8.3.3.3 Implement the System

Description

Build a system by implementing, testing and integrating one or more system elements. This includes bug fixing and unit testing.

Implement the system to:

- Create a working system.
- Develop, integrate and test the system elements.
- Increase the number of requirements implemented.
- Fix defects.
- Improve the system

Completion Criteria: System::Demonstrable, System::Usable, System::Ready
Input: Requirements, Software System, Way-of-Working
Output: Software System

8.3.3.4 Test the System

Description

Verify that the system produced meets the stakeholders' requirements.

Test the system to:

- Verify that the software system matches the requirements
- Identify any defects in the software system.

Completion Criteria: Requirements::Sufficient, Requirements::Fulfilled, System:: Demonstrable, System::Usable, System::Ready
Input: Requirements, Software System, Way-of-Working
Output: Requirements, Software System

8.3.3.5 Deploy the System

Description

Take the tested system and make it available for use outside the development team.

Deploy the system to:

- Package the software system up for delivery to the live environment.
- Make the software system operational.

Completion Criteria: System::Operational
Input: Stakeholders, Software System, Way-of-Working
Output: System

8.3.3.6 Operate the System

Description

Support the use of the software system in the live environment.

Operate the system to:

- Maintain service levels.
- Support the stakeholders who use the system.
- Support the stakeholders who deploy, operate, and help support the system.

Completion Criteria: System::Retired
Input: Stakeholders, Opportunity, Requirements, Software System, Way-of-Working
Output: System

8.4 The Endeavor Area of Concern

8.4.1 Introduction

This area of concern contains everything to do with the team, and the way that they approach their work.

Software engineering is a significant endeavor that typically takes many weeks to complete, affects many different people (the stakeholders) and involves a development team (rather than a single developer). Any practical method must describe a set of practices to effectively plan, lead and monitor the efforts of the team.

8.4.2 Alphas

The endeavor area of concern contains the following Alphas:

- Team
- Work
- Way-of-Working

8.4.2.1 Team

Description

Team: The group of people actively engaged in the development, maintenance, delivery and support of a specific software system.

The team plans and performs the work needed to create, update and/or change the software system.

States

Seeded	The team's mission is clear and the know-how needed to grow the team is in place.
Formed	The team has been populated with enough committed people to start the mission.
Collaborating	The team members are working together as one unit.
Performing	The team is working effectively and efficiently.
Adjourned	The team is no longer accountable for carrying out its mission.

Associations

produces : Software System	Team produces Software System.
performs and plans : Work	Team performs and plans Work.
applies : Way-of-Working	Team applies Way-of-Working.

Justification: Why Team?

Software engineering is a team sport involving the collaborative application of many different competencies and skills. The effectiveness of a team has a profound effect on the success of any software engineering endeavor. To achieve high performance, team members should reflect on how well they work together, and relate this to their potential and effectiveness in achieving their mission.

Normally a team consists of several people. Occasionally, however, work may be undertaken by a single individual creating software purely for their own use and entertainment. This is however a corner case which can be treated as a team with only one team member.

Progressing the Team

Teams evolve during their time together and progress through several state changes. As shown in Figure 9, the states are *seeded, formed, collaborating, performing*, and *adjourned*. They communicate the progression of a software team on the journey from initial conception to the completion of the mission indicating (1) when the team is seeded and the individuals start to join the team (2) when the team is formed to start the mission, (3) when the individuals start collaborating effectively and truly become a team, (4) when the team is performing and achieves a crucial level of efficiency and productivity, and (5) when the team is adjourned after completing its mission.

As shown in Figure 9, the team is first seeded. This implies defining the mission, deciding on recruitment for the necessary skills, capabilities and responsibilities, and making sure that the conditions are right for an effective group to come together. As the team is formed, the people in the group, and those joining it, bring the necessary skills and experience to the team. The group becomes a team as the people begin to see how they can contribute to the work at hand. As they discover and take account of each others' capabilities, they start collaborating effectively and make progress towards completing their mission.

At its peak of performing, the team shares a way of working, and plays to its strengths to complete its mission effectively and efficiently. The performing team easily adapts to the changing context and takes appropriate measures. If a number of people join or leave the team, or the context of the mission changes, it may revert to a previous state. Finally, if the team has no further goals or missions to complete, it is adjourned.

It is important to understand the current state of the team so that suitable practices can be used to address the issues and impediments being faced, and to ensure that the team focuses on working effectively and efficiently.

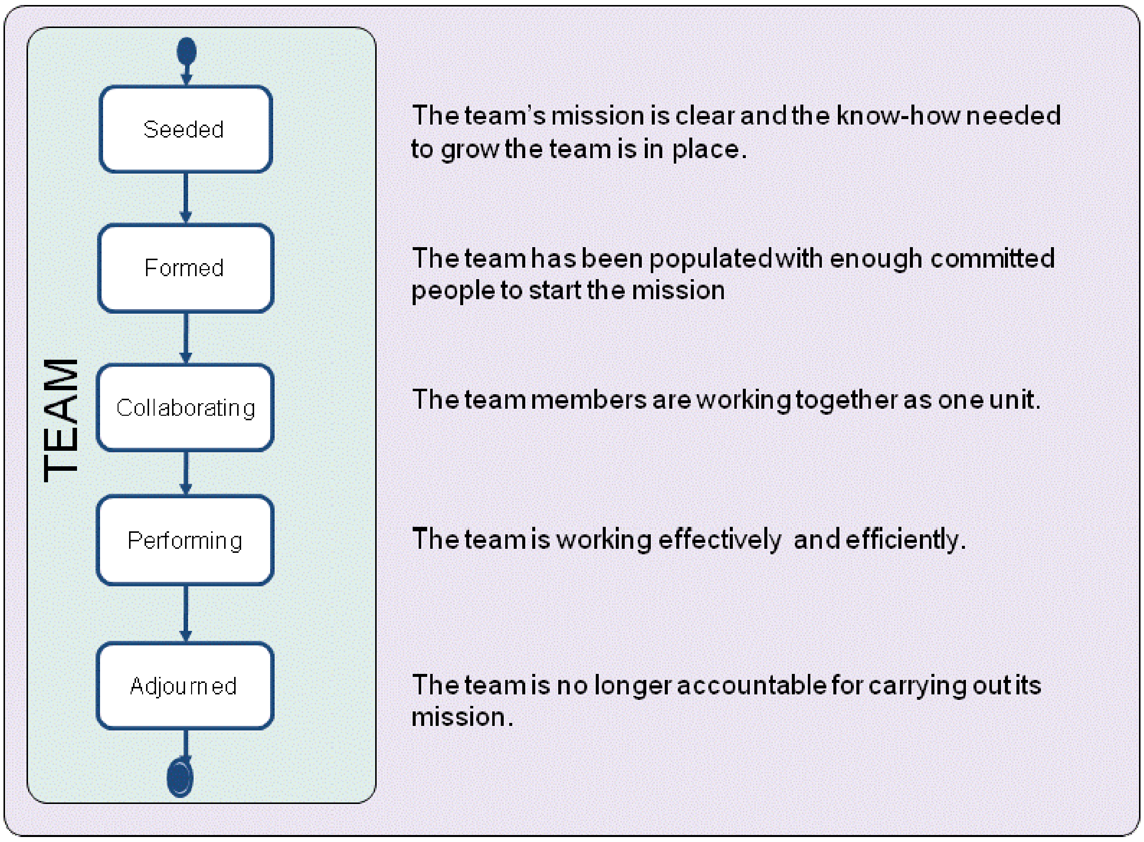

TEAM
Seeded
Formed
Collaborating
Performing
Adjourned
The team's mission is clear and the know-how needed to grow the team is in place.
The team has been populated with enough committed people to start the mission
The team members are working together as one unit.
The team is working effectively and efficiently.
The team is no longer accountable for carrying out its mission.

Figure 9 – The states of the Team

Checking the Progress of the Team

To help assess the state of a team and its progress, the following checklists are provided:

Table 5 – Checklist for Team

State	Checklist
Seeded	The team mission has been defined in terms of the opportunities and outcomes. Constraints on the team's operation are known. Mechanisms to grow the team are in place. The composition of the team is defined. Any constraints on where and how the work is carried out are defined. The team's responsibilities are outlined. The level of team commitment is clear. Required competencies are identified. The team size is determined. Governance rules are defined. Leadership model is selected.

Formed	Individual responsibilities are understood. Enough team members have been recruited to enable the work to progress. Every team member understands how the team is organized. All team members understand how to perform their work. The team members have met (perhaps virtually) and are beginning to get to know each other The team members understand their responsibilities and how they align with their competencies. Team members are accepting work. Any external collaborators (organizations, teams and individuals) are identified. Team communication mechanisms have been defined. Each team member commits to working on the team as defined.
Collaborating	The team is working as one cohesive unit. Communication within the team is open and honest. The team is focused on achieving the team mission. The team members put the success of the team as a whole ahead of their own personal objectives. The team members know each other.
Performing	The team consistently meets its commitments. The team continuously adapts to the changing context. The team identifies and addresses problems without outside help. The team is consistently producing high quality output. The team is considered a high performance team. Effective progress is being achieved with minimal avoidable backtracking and reworking. Wasted work, and the potential for wasted work are continuously eliminated.
Adjourned	The team responsibilities have been handed over or fulfilled. The team members are available for assignment to other teams. No further effort is being put in by the team to complete the mission.

8.4.2.2 Work

Description

Work: Activity involving mental or physical effort done in order to achieve a result.

In the context of software engineering, work is everything that the team does to meet the goals of producing a software system matching the requirement and addressing the opportunity presented by the stakeholders. The work is guided by the practices that make up the team's way-of-working.

States

Initiated	The work has been requested.
Prepared	All pre-conditions for starting the work have been met.
Started	The work is proceeding.

Under Control	The work is going well, risks are under control, and productivity levels are sufficient to achieve a satisfactory result.
Concluded	The work to produce the results has been concluded.
Closed	All remaining housekeeping tasks have been completed and the work has been officially closed.

Associations

updates and changes: Software System	Work updates and changes Software System.
set up to address : Opportunity	Work set up to address Opportunity.

Justification: Why Work?

The ability of team members to co-ordinate, organize, estimate, complete, and share their work has a profound effect on meeting their commitments and delivering value to their stakeholders. Team members need to understand how to carry out their work, and how to recognize when the work is going well.

Progressing the Work

During the development of a software system the work progresses through several state changes. As shown in Figure 10, they are *initiated, prepared, started, under control, concluded*, and *closed*. These states provide points of stability in the progression of the work indicating when the work is initiated and prepared, when the team is assembled and the work is started and brought under control, when the results are achieved and the development work is concluded, and finally, when the work itself is closed and all loose ends and outstanding work items are addressed.

As indicated in Figure 10, the work is first initiated. This implies that someone defines the desired result, and makes sure that the conditions are right for the work to be performed. If the work is not successfully initiated, it will never be

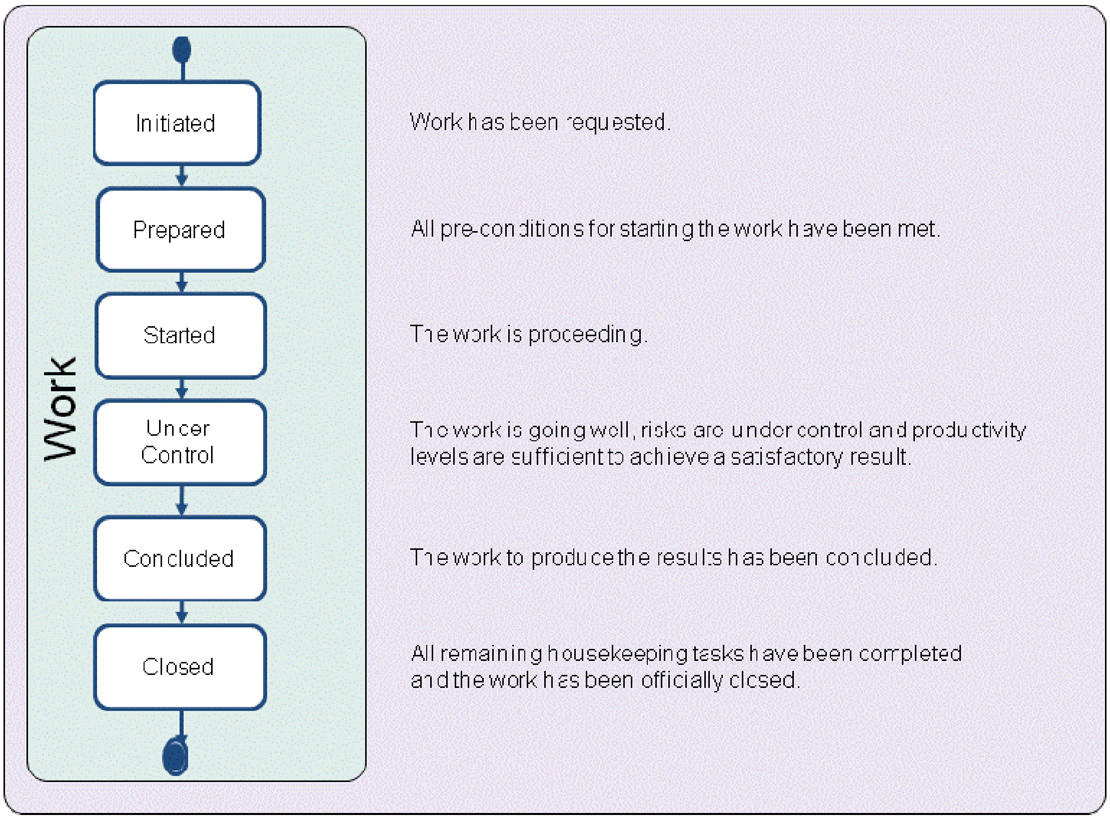

Work
Initiated
Prepared
Started
Under Control
Concluded
Closed
Work has been requested.
All pre-conditions for starting the work have been met.
The work is proceeding.
The work is going well, risks are under control and productivity levels are sufficient to achieve a satisfactory result.
The work to produce the results has been concluded.
All remaining housekeeping tasks have been completed and the work has been officially closed.

Figure 10 – The states of the Work

progressed and assigned to a team. As the work is prepared, commitments are made, funding and resources are secured, the work is organized, appropriate governance policies and procedures are put in place, and priorities, constraints and impediments are understood. Once all the pre-conditions for starting the work are addressed, the team gets the go-ahead to get the real work started. The team starts to complete the individual work items, and builds evidence showing that the work is under control.

There are many practices that can be used to help organize and co-ordinate the work including SCRUM, Kanban, PMBoK, PRINCE2, Task Boards and many, many more. These typically involve breaking the work down into:

1. Smaller, more bite sized work items that can be completed one-by-one such as work packages, and tasks.
2. One or more clearly defined work periods such as phases, stages, iterations, or sprints.

The level, depth and extent of the work breakdown depends on the style and complexity of the work and on the specific practices the team selects to help them co-ordinate, monitor, control and undertake the work.

If the team has their work under control then there will be concrete evidence that:

1. The work is going well.
2. The risks threatening a successful conclusion to the work are under control as the impact if they occur and/or as the likelihood of them occurring have been reduced to acceptable levels.
3. The team's productivity levels are sufficient to achieve satisfactory results within the time, budget and any other constraints that have been placed upon the work.

Typically, once the work has been concluded and the results have been accepted by the relevant stakeholders, there remain some final housekeeping and wrap up activities to be completed before the work itself can be closed.

If, for any reason, the work is not going well, then it may be halted, abandoned or reverted to a previous state. If the work is abandoned once it is started, it should still be properly closed even though it has not managed to pass through the concluded state.

Understanding the current and desired state of the work can help the team to balance their activities, make the correct investment decisions, nurture the work that is going well, and help or cancel the work that is going badly.

Checking the Progress of the Work

To help assess the state of the work and the progress being made towards its successful conclusion, the following checklists are provided:

Table 6 – Checklist for Work

State	Checklist
Initiated	The result required of the work being initiated is clear. Any constraints on the work's performance are clearly identified. The stakeholders that will fund the work are known. The initiator of the work is clearly identified. The stakeholders that will accept the results are known. The source of funding is clear. The priority of the work is clear.
Prepared	Commitment is made. Cost and effort of the work are estimated. Resource availability is understood. Governance policies and procedures are clear.

	Risk exposure is understood. Acceptance criteria are defined and agreed with client. The work is broken down sufficiently for productive work to start. Work items have been identified and prioritized by the team and stakeholders. A credible plan is in place. Funding to start the work is in place. The team is ready to start the work. Integration and delivery points are defined.
Started	Development work has been started. Work progress is monitored. The work is being broken down into actionable work items with clear definitions of done. Team members are accepting and progressing work items.
Under Control	Work items are being completed. Unplanned work is under control. Risks are under control as the impact if they occur and the likelihood of them occurring have been reduced to acceptable levels. Estimates are revised to reflect the team's performance. Measures are available to show progress and velocity. Re-work is under control. Work items are consistently completed on time and within their estimates.
Concluded	All outstanding work items are administrative housekeeping or related to preparing the next piece of work. Work results are being achieved. The client has accepted the resulting software system.
Closed	Lessons learned have been itemized, recorded and discussed. Metrics have been made available. Everything has been archived. The budget has been reconciled and closed. The team has been released. There are no outstanding, uncompleted work items.

8.4.2.3 Way-of-Working

Description

Way-of-Working: The tailored set of practices and tools used by a team to guide and support their work.

The team evolves their way of working alongside their understanding of their mission and their working environment. As their work proceeds they continually reflect on their way of working and adapt it to their current context, if necessary.

States

Principles Established	The principles, and constraints, that shape the way-of-working are established.
Foundation Established	The key practices, and tools, that form the foundation of the way of working are selected and ready for use.
In Use	Some members of the team are using, and adapting, the way-of-working.
In Place	All team members are using the way of working to accomplish their work.
Working well	The team's way of working is working well for the team.
Retired	The way of working is no longer in use by the team.

Associations

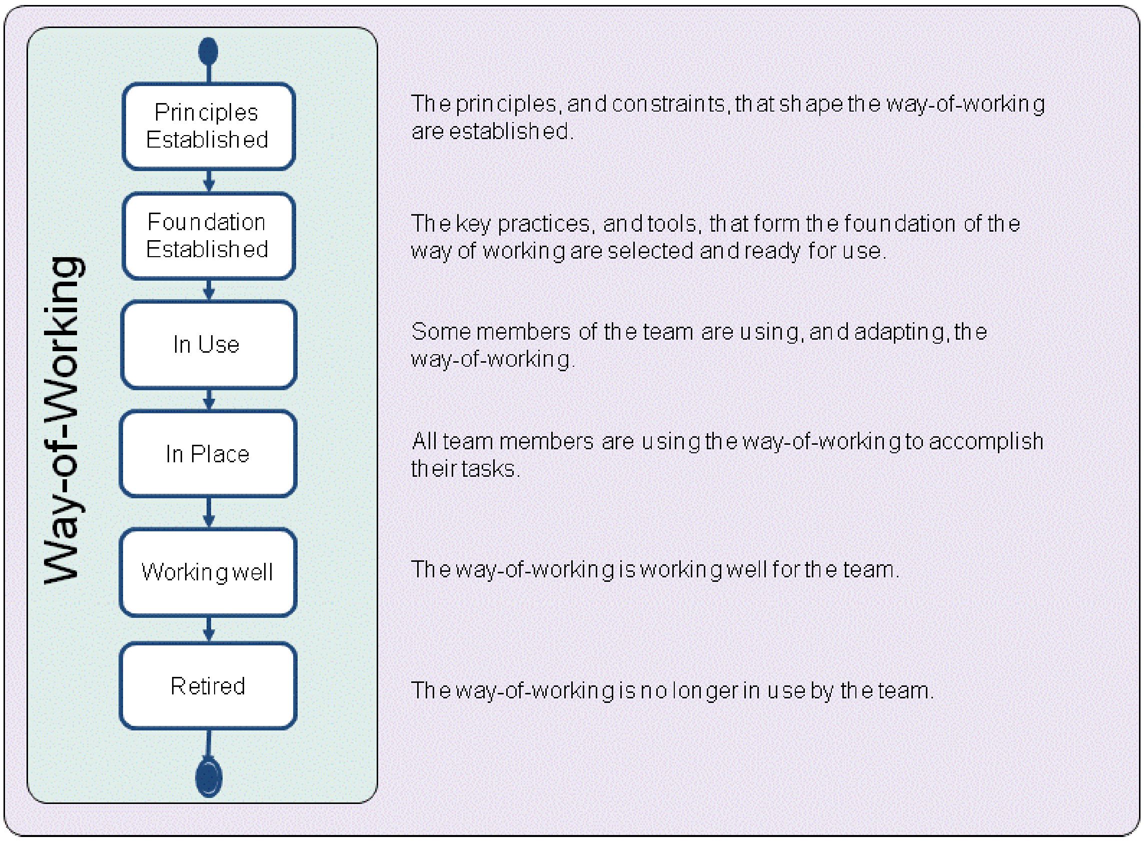
guides : Work	Way-of-Working guides Work.

Justification: Why Way-of-Working?

Software engineering is a team sport, one that requires the whole team to collaborate effectively regardless of how the team is organized. They need to agree on a way of working that will guide them throughout the software engineering endeavor.

The way of working:

- Is key to enabling a team to work together effectively.
- Focuses the team on how they will collaborate to ensure success.
- Enables the work to be planned and controlled.
- Helps the team, and their associated stakeholders, to successfully fulfill their responsibilities.



Way-of-Working
Principles Established
Foundation Established
In Use
In Place
Working well
Retired
The principles, and constraints, that shape the way-of-working are established.
The key practices, and tools, that form the foundation of the way of working are selected and ready for use.
Some members of the team are using, and adapting, the way-of-working.
All team members are using the way-of-working to accomplish their tasks.
The way-of-working is working well for the team.
The way-of-working is no longer in use by the team.

Figure 11 – The states of the Way-of-Working

Progressing the Way-of-Working

During the course of a software engineering endeavor the way of working progresses through several state changes. As presented in Figure 11, they are *principles established, foundation established, in use, in place, working well*, and *retired*. These states focus on the way a team establishes an effective way-of-working indicating (1) when the principles and constraints that shape the way-of-working are established, (2) when a minimal number of key practices and tools have been identified and integrated to establish a foundation for the evolution of the team’s way-of-working, (3) when a team's way of working is in use by the team, (4) when a team’s way of working is in place and in use by the whole team (5) when it is working well, and (6) when the way of working has been retired and is no longer in use by the team.

There are many ways of working that the team could adopt to meet their objectives and establish their approach to software engineering. As shown in Figure 11, the first step in adopting a new way-of-working, or adapting an existing way-of-working, is to understand the team’s working environment and establish the principles that will guide their selection of appropriate practices and tools. This includes identifying the constraints governing the selection of the team's practices and tools and understanding the practices and tools that the team, and their stakeholders, are already using or are required to use.

It is not enough to just understand the principles and constraints that will inform the team's way of working. These must be agreed with, and actively supported by, the team and its stakeholders. Once the principles are established the team is ready to start selecting the practices and tools that will form their way-of-working.

To establish a natural way of working the focus should first be on the key practices and tools; those that bring the team together, enable communication among the team members, support collaborative working and are essential to the success of the team. However, these practices and tools act as the foundation for the team’s way-of-working. Before the foundation can be assembled it is important to understand the gaps between the practices and tools needed by the team and the practices, and tools immediately available to the team. This enables the activities needed to fill these gaps to be planned.

Once the key practices and tools are integrated then the way-of-working’s foundation is established and the way-of-working is ready to be trialed by the team. It will however be continuously adapted as the work progresses, and additional practices and tools will be added as the team inspects their way-of-working and adapts it to meet their changing circumstances.

Rather than spending more time tailoring or tuning the way-of-working it is important that the team puts it into use as soon as possible. The way-of-working is in use as soon as any of the team members are using and adapting it as part of completing their work. As more and more of the team start to use and benefit from the way-of-working its usage will grow until it is firmly in place and all the team members are using it to accomplish their work. Some team members may still need help from their teammates to understand certain aspects of the team's way of working and to make effective progress, but the way of working is now the normal way for the team to develop software.

As the team progresses through the work, the way of working will become embedded in their activities and collaborations to such an extent that its use, inspection and adaptation are all seen as a natural part of the way the team works. The way-of-working is working well once it has stabilized and all team members are making progress as planned by using and adapting it to suit their current working environment. Finally, when the way of working is no longer in use by the team, it is retired.

Understanding the current and desired state of the team's way of working helps a team to continually improve their performance, and adapt quickly and effectively to change.

Checking the Progress of the Way-of-Working

To help assess the current status of the way of working, the following checklists are provided:

Table 7 – Checklist for Way-of-Working

State	Checklist
Principles Established	Principles and constraints are committed to by the team.

	Principles and constraints are agreed to by the stakeholders. The practice needs of the work and its stakeholders are agreed. The tool needs of the work and its stakeholders are agreed. A recommendation for the approach to be taken is available. The context within which the team will operate is understood. The constraints that apply to the selection and use of practices and tools are known. The constraints that govern the selection and acquisition of the team's practices and tools are known.
Foundation Established	The key practices and tools that form the foundation of the way-of-working are selected. Enough practices for work to start are agreed to by the team. All non-negotiable practices and tools have been identified. The gaps that exist between the practices and tools that are needed and the practices and tools that are available have been analyzed and understood. The capability gaps that exist between what is needed to execute the desired way of working and the capability levels of the team have been analyzed and understood. The selected practices and tools have been integrated to form a usable way-of-working.
In Use	The practices and tools are being used to do real work. The use of the practices and tools selected is regularly inspected. The practices and tools are being adapted to the team’s context. The use of the practices and tools is supported by the team. Procedures are in place to handle feedback on the team’s way of working. The practices and tools support team working and collaboration.
In Place	The practices and tools are being used by the whole team to perform their work. All team members have access to the practices and tools required to do their work. The whole team is involved in the inspection and adaptation of the way-of-working.
Working well	Team members are making progress as planned by using and adapting the way-of-working to suit their current context. The team naturally applies the practices without thinking about them The tools naturally support the way that the team works. The team continually tunes their use of the practices and tools.
Retired	The team's way of working is no longer being used. Lessons learned are shared for future use.

8.4.3 Activity Spaces

The endeavor area of concern contains five activity spaces that cover the formation and support of the team, and planning and co-coordinating the work in-line with the way of working.

8.4.3.1 Prepare to do the Work

Description

Set up the team and its working environment. Understand and commit to completing the work.

Prepare to do the work to:

- Put the initial plans in place.
- Establish the initial way of working.
- Assemble and motivate the initial project team.
- Secure funding and resources.

Completion Criteria: Team::Seeded, Way of Working::Principles Established, Way of Working:: Foundation Established, Work::Initiated, Work::Prepared
Input: Stakeholders, Opportunity, Requirements
Output: Team, Way of Working, Work

8.4.3.2 Coordinate Activity

Description

Co-ordinate and direct the team’s work. This includes all ongoing planning and re-planning of the work, and adding any additional resources needed to complete the formation of the team.

Coordinate activity to:

- Select and prioritize work.
- Adapt plans to reflect results.
- Get the right people on the team.
- Ensure that objectives are met.
- Handle change.

Completion Criteria: Team::Formed, Work::Started, Work::Under Control
Input: Requirements, Team, Work, Way of Working
Output: Team, Way of Working, Work

8.4.3.3 Support the Team

Description

Help the team members to help themselves, collaborate and improve their way of working.

Support the team to:

- Improve team working.
- Overcome any obstacles.
- Improve ways of working.

Completion Criteria: Team::Collaborating, Way of Working::In Use, Way of Working::In Place
Input: Team, Work, Way of Working
Output: Team, Way of Working

8.4.3.4 Track Progress

Description

Measure and assess the progress made by the team.

Track progress to:

- Evaluate the results of work done.
- Measure progress.
- Identify impediments.

Completion Criteria: Team::Performing, Way of Working::Working Well, Work::Under Control, Work::Concluded
Input: Requirements, Team, Work, Way of Working
Output: Team, Way of Working, Work

8.4.3.5 Stop the Work

Description

Shut-down the software engineering endeavor and handover the team's responsibilities.

Stop the work to:

- Close the work.
- Handover any outstanding responsibilities.
- Handover any outstanding work items.
- Stand down the team.
- Archive all work done.

Completion Criteria: Team::Adjourned, Way of Working::Retired, Work::Closed
Input: Requirements, Team, Work, Way of Working
Output: Team, Way of Working, Work

9 Optional Kernel Extensions

9.1 Overview

This section presents a number of optional extensions for use with the Software Engineering Kernel. It begins with an overview of the set of kernel extensions and their use. It then continues with a description of each extension and its contents.

This section is work in progress; in particular the Alpha state checklists will be improved and completed.

9.1.1 Introduction

Although the kernel can have many uses, including helping monitor the progress and health of your software engineering endeavors, and the completeness of your software engineering methods, it can appear to be too abstract to actually drive the software development work. This is because the kernel is designed to be used in conjunction with your selected practices. To help you understand how the kernel works, and to provide some extensible assets to help in the creation of your own practices, we present three optional kernel extensions, one for each area of concern. These are the following:

- **Business Analysis Extension** – adds two Alphas, Need and Stakeholder Representative, to drive forward the Opportunity and the Stakeholders.
- **Development Extension** – adds two Alphas, Requirement Item and System Element to drive forward the Requirements and the Software System. As well as System Element it also adds Bug to monitor the health of the Software System. Bugs are an important thing to monitor, track and address in any software development endeavor, and one which will inhibit, rather than drive, progress being made to the Software System.
- **Task Management Extension** – adds three Alphas, Team Member, Task and Practice Adoption, to drive forward the Team, Work and Way-of-Working.

9.1.2 Why the Focus on Adding Alphas?

When using the kernel it is very unlikely that you will progress any of its Alphas as a single unit. In each case you will drive the progress of the Alpha by progressing its parts. For example the Requirements will be progressed by progressing the individual Requirements Items, each of which can progress at its own speed.

The way in which the Alphas progress is, of course, practice specific. For example agile practices will progress the Requirement Items either individually or in small batches, whereas a waterfall practice will typically try to move them all at the same time.

9.1.3 Why are the Sub-Ordinate Alphas not included in the Kernel?

When you look at the suggested set of new Alphas you may well think that they themselves are universal and question why they haven't been included in the kernel.

The problem when looking at software engineering at this level of detail is that the universals tend to be types of things rather than specific things. For example although every endeavor will have Requirement Items, they won't all have the same type of Requirement Items. Some teams will be using user stories, others will be using use cases, and some even using both. Whilst it is tempting to think that one could provide a definitive definition of a Requirement Item that is satisfactory to all communities and practices, in reality this is an impossibility and would lead to the practices becoming distorted and overly complicated. It is better to provide a generic definition and allow the practice authors to either extend this or ignore it as they wish.

9.1.4 How do you use the Kernel Extensions?

The kernel extensions can be used in a number of different ways:

1. To flesh out the kernel, providing a more complete picture of software engineering.

2. As templates for the creation of your own practices – for example the Requirements Item Alpha could be extended to provide a base for the definition of your own specific types of Requirements Items.
3. As inspiration and examples. By considering the relevant extensions before defining your own practices you will find it easier to create these and understand how they would be plugged into the kernel.

9.2 Business Analysis Extension

9.2.1 Introduction

This extension provides two additional Alphas to help teams to progress their Opportunity and Stakeholders.

9.2.2 Alphas

The business analysis extension extends the customer area of concern adding the following Alphas:

- Stakeholder Representative as a sub-ordinate of Stakeholders.
- Need as a sub-ordinate of Opportunity.

9.2.2.1 Stakeholder Representative

Description

Stakeholder Representative: A person, or group, empowered to represent a subset of the stakeholders in the endeavor.

Super-Ordinate Alpha

Stakeholders

States

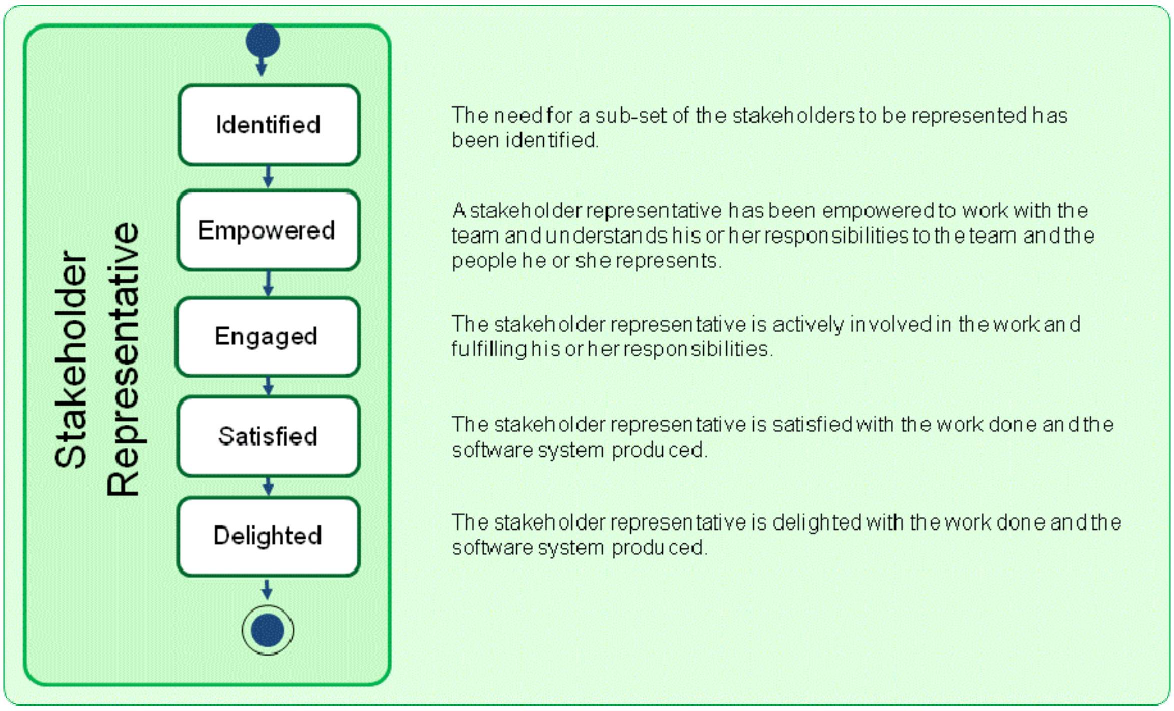
Identified	The need for a sub-set of the stakeholders to be represented has been identified.
Empowered	A stakeholder representative has been empowered to work with the team and understands his or her responsibilities to the team and the people he or she represents.
Engaged	The stakeholder representative is actively involved in the work and fulfilling his or her responsibilities.
Satisfied	The stakeholder representative is satisfied with the work done and the software system produced.
Delighted	The stakeholder representative is delighted with the work done and the software system produced.

Associations

drive : Stakeholders	The progress of the Stakeholder Representatives drives the progress of the Stakeholders.

Justification: Why Stakeholder Representative

The number of Stakeholders in any software system is often unbounded, with many systems affecting millions of people. The only practical way to engage with the Stakeholders is to appoint one or more Stakeholder Representatives to gather and reflect the opinions of the actual stakeholders. The Stakeholder Representative may be a single individual representing a sub-set of the stakeholders (or all stakeholders as is the case with the Scrum Product Owner), or some kind of official body such as a focus group or steering committee.



Stakeholder Representative
Identified
Empowered
Engaged
Satisfied
Delighted
The need for a sub-set of the stakeholders to be represented has been identified.
A stakeholder representative has been empowered to work with the team and understands his or her responsibilities to the team and the people he or she represents.
The stakeholder representative is actively involved in the work and fulfilling his or her responsibilities.
The stakeholder representative is satisfied with the work done and the software system produced.
The stakeholder representative is delighted with the work done and the software system produced.

Figure 12 – The states of the Stakeholder Representative

Progressing the Stakeholder Representatives

During the development of a software system the stakeholder representatives progress through several state changes. As shown in Figure 12, they are *identified*, *empowered*, *engaged*, *satisfied* and *delighted*. These states focus on the involvement and satisfaction of the stakeholder representatives, from the identification of a sub-set of the stakeholders that require explicit representation through the empowerment of stakeholder representatives, their engagement in the development work and their satisfaction and delight in the resulting software system. They communicate the progression of the relationship with the stakeholders who are either directly involved in the software engineering endeavor or support it by providing input and feedback.

As indicated in Figure 12, the first thing to do is to identify which sub-sets of the Stakeholders that require explicit representation in the project and to determine the number of Stakeholder Representatives required. The number of Stakeholder Representatives required can vary considerably from one system to another, but there is always at least one Stakeholder Representative available to the team.

To be effective the Stakeholder Representatives must be empowered both in their relationship with the team and in their relationship with their sub-set of the Stakeholders. Of particular importance is to make sure that they have the time available to support the team and understand the particular needs of the stakeholders they represent. Once they are empowered they need to be engaged with the team and to work with the team so that they are satisfied with the work done and the software system produced. It is key part of their responsibilities to accurately reflect the opinions of the Stakeholders they represent.

Checking the progress of a Stakeholder Representative

To help assess the state and progress of a Stakeholder Representative, the following checklists are provided:

Table 8 – Checklist for Stakeholder Representative

State	Checklist
Identified	• A person to act on behalf of the stakeholders has been identified from the stakeholder group. • The responsibilities of the stakeholder representative have been identified.
Empowered	• The stakeholder representative has domain knowledge. • The stakeholder representative has been authorized in decision making. • The stakeholder representative knows his /her responsibilities.
Engaged	• The stakeholder representative actively supports the team. • The stakeholder representative participates in decision making of the product. • The stakeholder representative provides feedback about the product.
Satisfied	• The minimum expectation of the stakeholders has been achieved.
Delighted	• The system meets or exceeds the minimum expectation of the stakeholders.

How the Stakeholder Representatives drive the progress of the Stakeholders

The progress of the Stakeholders is driven by the Stakeholder Representatives. For illustrative purposes the states of the two Alphas are shown in Figure 13.

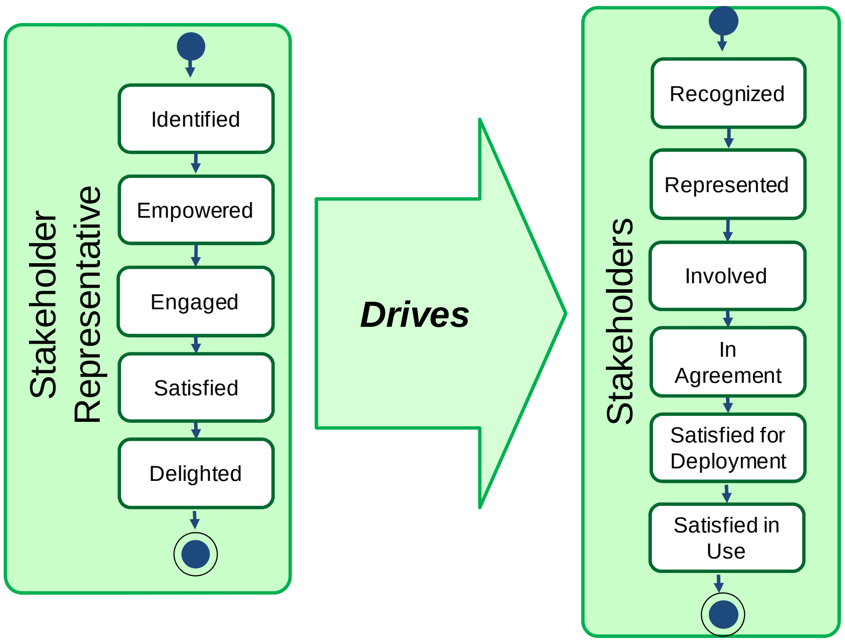


Figure 13 – The Stakeholder Representatives drive the progress of the Stakeholders

How the Stakeholder Representatives drive the progress of the Stakeholders is summarized in Table 9, along with the additional checklist items that this kernel extension adds to the Stakeholders' state checklists.

Table 9 – How the Stakeholder Representatives drive the Stakeholders

Stakeholders State	**How the Stakeholder Representatives drive the progress of the Stakeholders**	**Additional Checklist Items**
Recognized	First the Stakeholders must be *recognized*. An important part of this is to identify how they will be represented.	The proposed set of Stakeholder Representatives has been *Identified.*
Represented	Continuing to progress the Stakeholder Representatives will help to continue the progress of the Stakeholders. To ensure that the Stakeholders are *represented* it is important to make sure that all the identified Stakeholder groups have empowered Stakeholder Representatives.	All the recognized groups of Stakeholders have at least one *empowered* Stakeholder Representative.
Involved	To involve the Stakeholders their Stakeholder Representatives will have to be engaged.	All the recognized groups of Stakeholders have at least one *engaged* Stakeholder Representative.
In Agreement	Actively engaging the Stakeholder Representatives will facilitate bringing them to agreement about the Opportunity to be addressed and the Requirements for the Software System.	Enough of the Stakeholder Representatives are *engaged* in the decision making for agreement to be reached.
Satisfied for Deployment	The best indication of whether the Stakeholders are satisfied is the level of satisfaction of the individual Stakeholder Representatives. By satisfying the Stakeholder Representatives you can progress the Stakeholders to satisfied for deployment. Note: you may want to engage with more Stakeholder Representatives to verify that the Software System produced for the initial set of Stakeholder Representatives is generally applicable.	All the Stakeholder Representatives are *satisfied* or *delighted* with the Software System that has been produced.
Satisfied In Use	The best indication of whether the Stakeholders are satisfied is the level of satisfaction of the individual Stakeholder Representatives. By ensuring the continued satisfaction of the Stakeholder Representatives you can progress the Stakeholders to satisfied in use. Again you may want to engage with more Stakeholder Representatives to verify that the Software System produced for the initial set of Stakeholder Representatives is actually useful.	All the Stakeholder Representatives are *satisfied* or *delighted* with the Software System that is *operational.*

The state of the individual Stakeholder Representatives is independent of the overall state of the Stakeholders. For example an individual Stakeholder Representative may be *engaged* before the Stakeholders as a whole are *represented*.

Note that it is possible that a team may only have one Stakeholder Representative who represents all of the Stakeholders. In this case it is still useful to track the state of the Stakeholder Representative as well as the Stakeholders.

9.2.2.2 Need

Description

Need: A lack of something necessary, desirable or useful, requiring supply or relief.

Need exists within the customer, and will be considered by product or portfolio managers who analyze whether there will be value generated by addressing the Need, and pursuing the identified opportunities.

Super-Ordinate Alpha

Opportunity

States

Identified	A need related to the opportunity and the stakeholders is identified.
Value Established	The value to the customers and other stakeholders of a successful solution that addresses the need is established.
Satisfied	The minimal expectations for a solution that addresses the need have been met.
Expectation Exceeded	The minimal expectations for a solution that addresses the need have been exceeded to the extent that the stakeholders are delighted.

Associations

drive : Opportunity	The progress of the Needs drive the progress of the Opportunity.

Justification: Why Need

Different groups of Stakeholders will respond to the Opportunity in different ways and have different needs for a solution. Explicitly tracking the individual Needs is necessary if you want to truly understand the value of an Opportunity and delight the Stakeholders. Progressing the individual Needs is the best way to ensure that you progress the Opportunity.

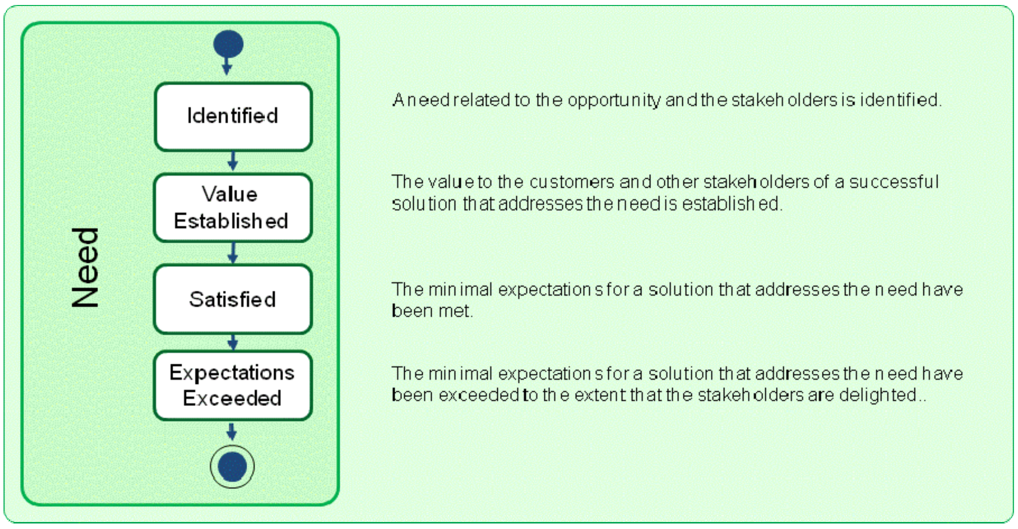

Need
Identified
Value Established
Satisfied
Expectations Exceeded
A need related to the opportunity and the stakeholders is identified.
The value to the customers and other stakeholders of a successful solution that addresses the need is established.
The minimal expectations for a solution that addresses the need have been met.
The minimal expectations for a solution that addresses the need have been exceeded to the extent that the stakeholders are delighted..

Figure 14 - The states of the Need

The Need is necessary for having a well defined Opportunity, as the Opportunity is the possibility to provide a solution/system that meets the Needs of the Stakeholders.

Progressing the Need

If the Team does not take the time to understand the Needs that drive the Opportunity they are likely to identify the wrong Requirements and develop the wrong Software System. The Needs need to be understood and individually addressed. As shown in Figure 14 Needs progress through the *identified*, *value established*, *satisfied* and *expectations exceeded* states. These states focus on understanding the value of addressing the need and the benefit that can be expected from the delivery of an appropriate Software System.

The need is the inherent lack of something necessary, desirable or useful, requiring supply or relief. As indicated in Figure 14, a Need initially is identified and described in a suitable form. One form it can take is in describing potential features of a new or existing system. Alternatively it can be described in terms of desired outcomes or benefits to be achieved. Once the Need has been *identified* the next step is to quantify the benefit that could be generated if the Need is addressed. As a next step, the Need's *value* gets *established*, the value to the customers, and other stakeholders. Here, the solution that addresses the Need is quantified and the need has been prioritized.

Finally, when a Software System is available and it fulfills the minimum expectations the Need can progress to the *satisfied* state. To truly delight the Stakeholders the Software System must surpass the minimal expectation in some way. If this happens then the Need is progressed to the *expectations exceeded* state.

Checking the progress of Need

To help assess the state and progress of Need, the following checklists are provided:

Table 10 – Checklist for Need

State	**Checklist**
Identified	• A lack of something necessary, desirable or useful to the Stakeholders and related to the Opportunity has been identified. • The Need has been clearly described. • It is clear which Stakeholder groups share the Need.
Value Established	• The value of addressing the Need has been quantified. • The relative priority of the Need is clear. • The minimum expectations of the affected Stakeholders are clear.
Satisfied	• A usable software system that addressed the Need is available. • The minimum expectations of the affected stakeholders have been satisfied.
Expectation Exceeded	• The minimum expectations of the affected stakeholders have been exceeded.

How the Need drives the progress of the Opportunity

The need will drive the opportunity by providing the targets for the opportunity to achieve. From a provider point of view the opportunity is the possibility to create a solution that meets the needs of the Stakeholders. The need also provides the foundation for the formulation of the Requirements.

The progress of the Opportunity is driven by the Needs. For illustrative purposes the states of the two Alphas are shown in Figure 15.

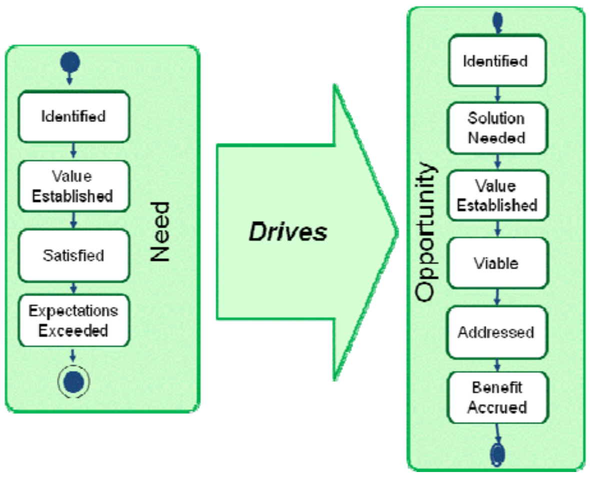


Figure 15 – The Needs drive the progress of the Opportunity

How the Needs drive the progress of the Opportunity is summarized in Table 11, along with the additional checklist items that this kernel extension adds to the Opportunity's state checklists.

Table 11 – How the Needs drive the Opportunity

Opportunity State	**How the Needs drive the progress of the Opportunity**	**Additional Checklist Items**
Identified	First an Opportunity must be identified. Although the Opportunity will be more convincing if some of the Needs that drive it have been identified progress to this state is independent of the state of any of the sub-ordinate Needs.	None.
Solution Needed	To demonstrate that a solution is needed analysis if the Opportunity and the Needs that drive it is required. If no compelling Needs are identified then there is no real need for the solution.	At least one compelling Need has been *identified*.
Value Established	To understand the value of the Opportunity one must understand the value of the Needs that drive it. Progressing the Needs to Value Established will help to progress the Opportunity to Value Established.	All of the Needs have been progressed to *value established*.
Viable	Once the value of addressing the Opportunity and its underlying Needs has been established additional work is needed to cost the solution and establish if the Opportunity is viable. No further progress on the Needs is needed at this stage.	None

Addressed	Continuing to progress the Needs will help to progress the Opportunity to Addressed. The Opportunity has not been properly addressed in there are critical Needs that have not been satisfied.	All of the critical Needs have been *satisfied*.
Benefit Accrued	It will be difficult for benefit to be accrued from the use of the Software System if it has not satisfied the critical Needs.	It is confirmed by the users that the critical Needs have been satisfied or *expectations* are *exceeded*.

Some practices, like goal oriented requirements engineering practices, will introduce the concept of goal as a link from needs and opportunities to system requirements. In such cases a new sub-ordinate alpha of requirements can be introduced for this.

9.3 Development Extension

9.3.1 Introduction

This extension provides three additional Alphas to help teams to progress the Requirements and Software System alphas.

9.3.2 Alphas

The development extension expands the solution area of concern adding the following Alphas:

- Requirement Item as a sub-ordinate of Requirements.
- Bug as a sub-ordinate of Software System.
- System Element as a sub-ordinate of Software System.

9.3.2.1 Requirement Item

Description

Requirement Item: a condition or capability needed by a stakeholder to solve a problem or achieve an objective.

Requirements are composed of Requirement Items. These are the individual requirements, which can be addressed and progressed individually. The overall progress and health of the Requirements alpha is driven by the progress and health of its Requirement Items. The number of Requirement Items can vary in a wide range from one system to another.

Super-Ordinate Alpha

Requirements

States

Identified	A specific condition or capability that the Software System must address has been identified.
Described	The Requirement Item is ready to be implemented.
Implemented	The Requirement Item is implemented in the Software System and demonstrated to work.
Verified	Successful implementation of the Requirement Item in the Software System has been confirmed.

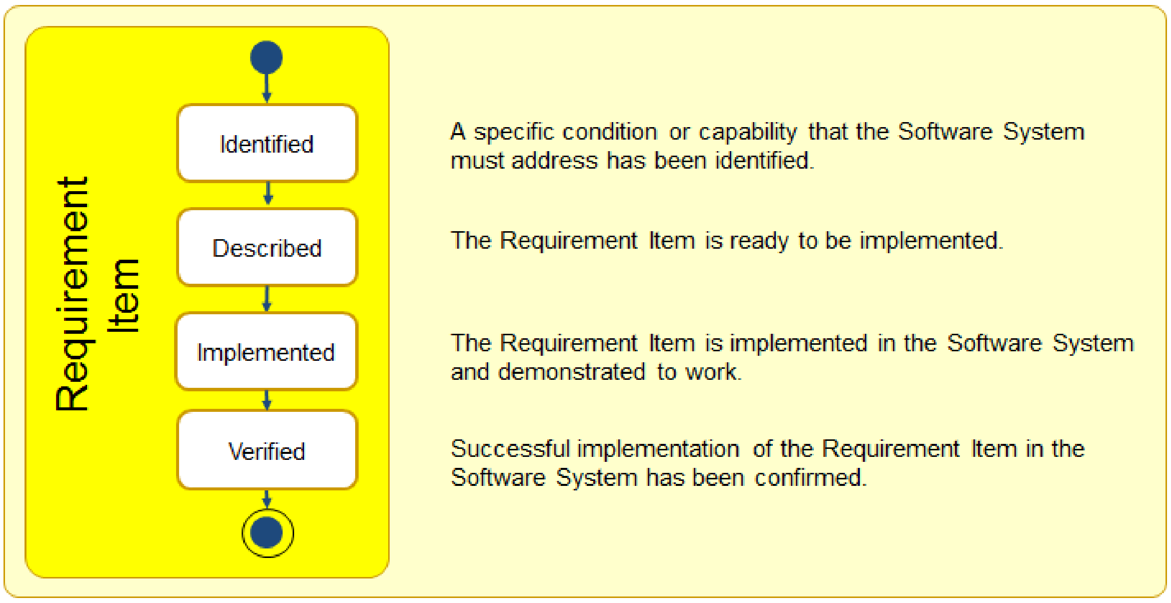

Requirement Item
Identified
Described
Implemented
Verified
A specific condition or capability that the Software System must address has been identified.
The Requirement Item is ready to be implemented.
The Requirement Item is implemented in the Software System and demonstrated to work.
Successful implementation of the Requirement Item in the Software System has been confirmed.

Figure 16 - The states of Requirement Item

Associations

drive : Requirements	The progress of the Requirement Items drives the progress of the Requirements.

Justification: Why Requirement Item

The Software System is usually developed to fulfill a number (a potentially very high number) of Requirements. The only efficient way to manage them is to manage them individually whilst being aware of their progress as a whole. Managing requirements at the Requirement Item level allows teams to ensure that the Requirements are appropriately crafted (i.e. they are necessary, implementation independent, clear and concise, complete, consistent, achievable, traceable and verifiable). It also helps when mapping them to the code and tests, and when using any form of requirements management tool.

Progressing the Requirement Items

During the development of a software system the requirement items progress through several state changes. As shown in Figure 16, they are *identified*, *described*, *implemented* and *verified*. These states focus on the progress and health of the individual Requirement Items, from their identification and description as part of the requirements elicitation to their implementation and verification by the development team. Understanding the state of the Requirement Items helps you to plan, track and drive the development of the required Software System.

The individual Requirement Items are first identified. This may be as the result of a requirements workshop, receiving a change request, or even derived from another higher-level Requirement Item. In the first state of the Requirement Item, the *identified* state, a specific condition or capability that the Software System must address has been identified. Its objectives have been briefly defined and it has been put under configuration control. Work is then needed to flesh out the Requirement Item and ensure that it is well formed and suitably described. In the *described* state, the description of the Requirement Item evolves into a clear, concise, complete, consistent and verifiable description. The Requirement Item is also as justified as necessary and achievable, and prioritized relative to its peers. Next, the Requirement Item is *implemented* as part of the Software System. Finally the last few activities and pieces of testing are completed to confirm that the Requirement Item is truly done. In the *verified* state, it has been confirmed that the Software System successfully implements the Requirement Item.

Checking the progress of a Requirement Item

To help assess the state and progress of a Requirement Item, the following checklists are provided:

Table 12 – Checklist for Requirement Item

State	**Checklist**
Identified	• Requirement Item is briefly described. • The Requirement Item is put under configuration control. • The origin of the Requirement Item is clear. • The value of implementing the Requirement Item is clear.
Described	• The Requirement Item is justified as necessary and achievable. • The Requirement Item is described clearly, concisely, and consistently. • The Requirement Item is described in a verifiable way, and is possible to test. • The Requirement Item is prioritized relative to its peers. • The Requirement Item does not specify a design or solution. • The Requirement Item is ready for development. • The impact of implementing the Requirement Item is understood.
Implemented	• The System Elements involved in the implementation of the Requirement Item are known. • The development and developer testing of the code that implements the Requirement Item is complete. • A build of the Software System implementing the Requirement Item is available for further demonstration and testing.
Verified	• Tests have been successfully executed that show that the Requirement Item has been acceptably well implemented. • Verification report is stored and available for future reference.

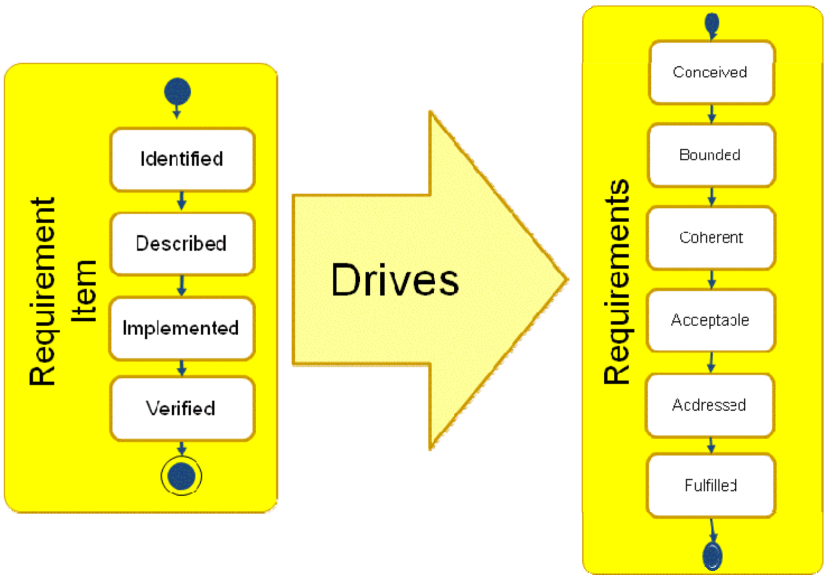


Figure 17- The Requirement Items drive the progress of the Requirements

How the Requirement Items drive the progress of the Requirements

The progress of the Requirements is driven by the associated Requirement Items. For illustrative purposes the states of the two Alphas are shown in Figure .

How the Requirement Items drive the progress of the Requirements is summarized in Table , along with the additional checklist items that this kernel extension adds to the Requirements' state checklists.

Table 13 – How the Requirement Items drive the Requirements

Requirements State	**How the Requirement Items drive the progress of the Requirements**	**Additional Checklist Items**
Conceived	Progress to this state is independent of the state of any of the sub-ordinate Requirement Items.	None
Bounded	To properly bound the Requirements some of the most important Requirement Items should be identified and described.	One or more essential Requirement Items have been *Identified and Described*.
Coherent	Continuing to progress the Requirement Items will help to continue the progress of the Requirements. Describing the Requirement Items that communicate the essential characteristics of the system will help the Requirements to become coherent.	New complete checklist: • The Requirement Items have been *Identified* and shared with the team and the stakeholders. • The Requirement Items that communicate the essential characteristics of the system have been *Described*. • Conflicting Requirement Items have been identified and attended to. • The Requirement Items communicate the essential characteristics of the system to be delivered. • The most important usage scenarios for the system can be explained. • The team understands what has to be delivered and agrees to deliver it.
Acceptable	Describing the highest priority Requirement Items will help evolve the Requirements to the point where they define a system acceptable to the stakeholders. Note: For mature systems this may only require the definition of a single Requirement Item – what makes the Requirements acceptable is up to the Stakeholders.	Enough Requirement Items are *Described* to define a system acceptable to the stakeholders.
Addressed	Implementing and verifying the Requirement Items is the only way to address the Requirements. The Requirements are addressed when the	New complete checklist: • Enough of the Requirement Items have been *Implemented* and *Verified* for the resulting system to be acceptable to the

	set of Requirement Items implemented and verified provide clear value to the stakeholders and the resulting system is worth releasing.	stakeholders. • The stakeholders accept the Requirement Items as accurately reflecting what the system does and does not do. • The set of Requirement Items implemented and verified provide clear value to the stakeholders. • The system implementing the Requirement Items is accepted be the stakeholders as worth making operational.
Fulfilled	You continue implementing and verifying additional requirement items until the resulting system fully satisfies the need for a new system, and there are no outstanding requirement items preventing the system from being considered complete.	Requirements checklist item "There are no outstanding requirement items preventing the system from being accepted as fully satisfying the requirements" is replaced with following: "All Requirement Items preventing the system from being accepted as fully satisfying the requirements have been *Verified*."

The state of the individual Requirement Items is independent of the states of their owning Requirements. It is quite possible for one or more Requirement Items to be *Verified* before the Requirements are *Bounded* or *Coherent*. For example you could implement and verify some of the most obvious, important and risky requirement items before investing the time and effort in working with the Stakeholders to make the Requirements *Bounded* or *Coherent*.

9.3.2.2 Bug

Description

Bug: An error, flaw, or fault in a Software System that causes it to fail to perform as required.

Super-Ordinate Alpha

Software System

States

Detected	An error, fault or flaw in the Software System is observed and logged.
Located	The cause of the Bug in the Software System has been found.
Fixed	The Bug has been removed from the Software System.
Closed	The removal of the Bug from the Software System has been confirmed.

Associations

inhibit : Software System	The Bugs inhibit the progress of the Software System.

Justification: Why Bug

Bugs are inevitable part of software development. The trick is to eliminate them all before the Software System is operational. The overall state of the Software System is affected by the quantity and severity of the Bugs it contains. Understanding and monitoring the progress and health of any Bugs detected is an essential part of any software engineering endeavor.

Essence uses the term Bug as it is one of the most common words in the software industry, and is more intuitive and less open to misinterpretation than the other alternatives such as problem and defect.

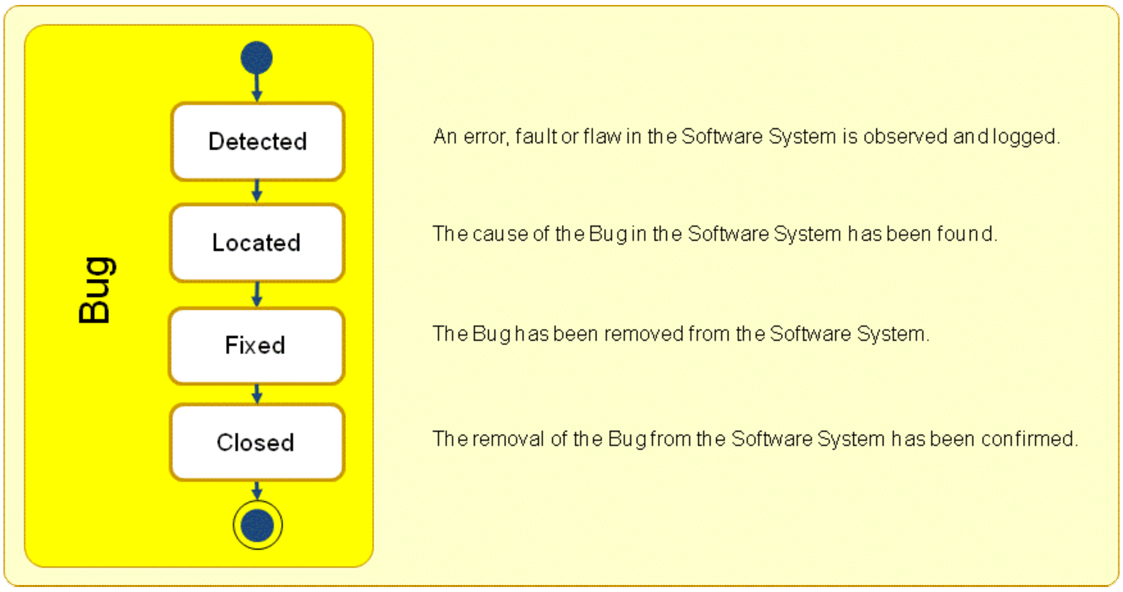

Bug
Detected
Located
Fixed
Closed
An error, fault or flaw in the Software System is observed and logged.
The cause of the Bug in the Software System has been found.
The Bug has been removed from the Software System.
The removal of the Bug from the Software System has been confirmed.

Figure 18 - The states of a Bug

Progressing the Bugs

Bugs threaten the success of any software engineering endeavor. They have to be found and resolved before they cause any damage. As shown in Figure 18, Bugs progress through the *detected*, *located*, *fixed* and *closed* states. These states focus on the management of the Bugs and provide clear understanding of whether they are inhibiting the progress of, or threatening the health of, the Software System.

The Bug first has to be *detected.* This may be as the result of testing, reviewing or using the Software System. Once a Bug is *detected* it is reported and logged. Then the Bug must be investigated and its cause must be *located.* If the cause of the bug cannot be identified then it will be impossible to fix. Once the Bug is *located* it can be *fixed* and a new bug-free version of the Software System can be made available. Finally, after the Team has confirmed its absence in the updated Software System, the Bug is *closed*.

Checking the progress of a Bug

To help assess the progress and health of a Bug, the following checklists are provided:

Table 14 – Checklist for Bug

State	**Checklist**
Detected	• Bug has been reported and given a unique identifier. • Details about the Bug, and the situation within which it occurred, have been reported. • The severity of the Bug has been assessed.
Located	• The Bug has been investigated and its impact assessed. • The System Elements causing the Bug have been identified. • The cost of fixing and testing the Bug has been estimated. • The Bug is ready to be fixed.

Fixed	• The work required to correct the offending System Elements has been completed. • A new Bug-free version of the Software System is available. • The absence of the Bug has been verified.
Closed	• Tests, reviews or other appropriate activities have been undertaken to ensure that the Bug has been corrected or shown not to actually be an error, fault or flaw. • The Bug management has been finalized.

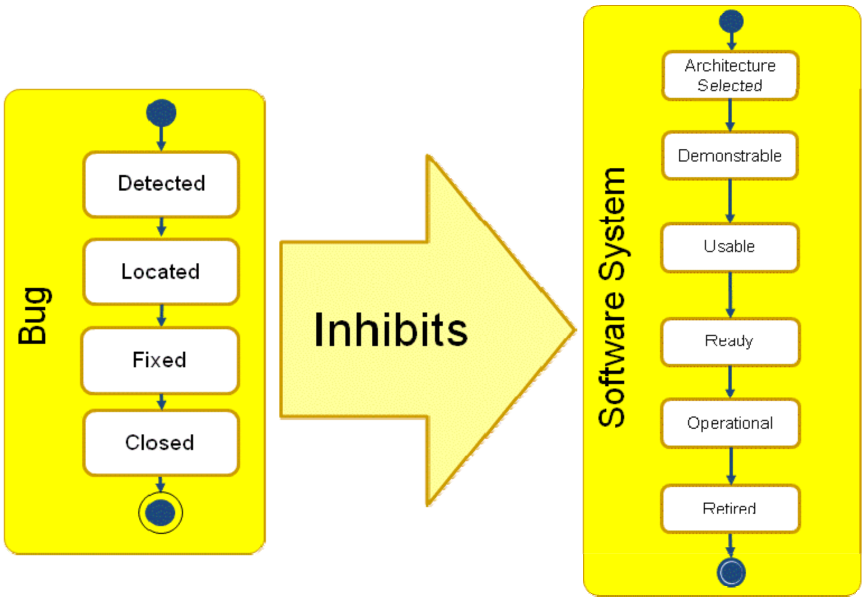


Figure 19- The Bugs inhibit the progress of the Software System

How the Bugs drive the progress of the Software System

The progress of the Software System is inhibited by the Bugs found in it. For illustrative purposes the states of the two Alphas are shown in Figure .

How the Bugs inhibit the progress of the Software System is summarized in Table , along with the additional checklist items that this kernel extension adds to the Software System's state checklists.

Table 15 – How the Bugs drive the Software System

Software System State	**How the Bugs drive the progress of the Software System**	**Additional Checklist Items**
Architecture Selected	Progress to this state is independent of the state of any of Bugs in the Software System.	None
Demonstrable	When the Software System is in demonstrable state some bugs may be detected and located.	The Bugs *Detected* and/or *Located* did not prevent the Software System from being successfully demonstrated.

Usable	Detecting and fixing Bugs will help to continue the progress of the Software System. Fixing any Bugs in the core functionality of the Software System is essential for it to become usable.	All critical Bugs have been *Fixed*.
Ready	Detecting and fixing Bugs will help evolve the Software System to the point where it is ready for deployment in a live environment.	The number and severity of the Bugs yet to be *Fixed* and *Closed* are low enough so that the system can be deployed.
Operational	Fixing any Bugs detected during live use of the Software System is an important part of keeping it operational.	The remaining Bugs, if any, do not require immediate fixing.
Retired	The system is no longer being supported	None

The state of the individual Bugs are independent of the states of their owning Software System. It is quite possible for one or more Bugs to be *Detected* or *Located* after the Software System is *Ready* or *Operational*. For example using the system which contains some non-critical Bugs may be beneficial enough for deploying and using it before these Bugs are closed.

9.3.2.3 System Element

Description

System Element: Independently developable and testable part of a system.

System Elements are the independent but interrelated parts that together comprise a Software System. Hence, the Software System's progress and health are driven by the progress and health of its System Elements.

Super-Ordinate Alpha

Software System

States

Identified	A system element has been identified as part of the Software System and its responsibilities and its position in the Software System are clear.
Interfaces Agreed	The System Elements interfaces have been agreed.
Developed	The System Element has been implemented and tested, and is believed to be ready for integration into the Software System.
Ready	The System Element has been verified and is ready for live use as part of the Software System.

Associations

drive : Software System	The progress of the System Elements drives the progress of the Software System.

Justification: Why System Element

A Software System is made up of software, hardware, and data. Each part of the Software System can be software or hardware or data or any combination of the three. A Software System usually consists of several parts or System

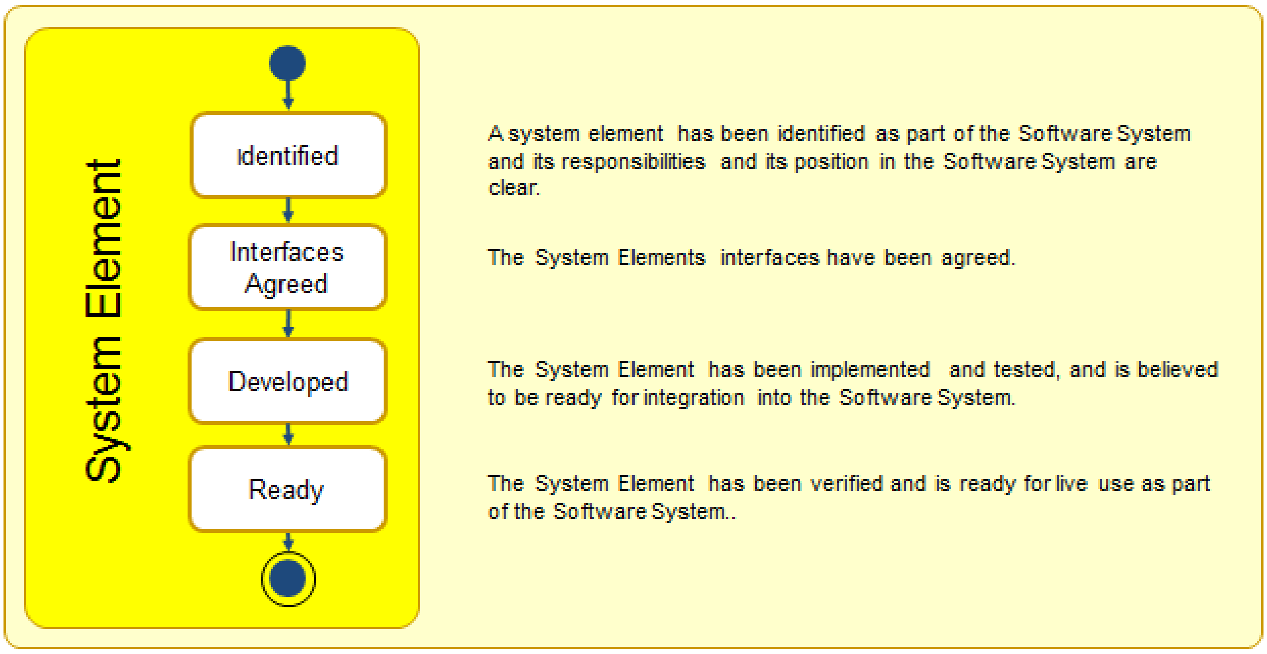

System Element
Identified
Interfaces Agreed
Developed
Ready
A system element has been identified as part of the Software System and its responsibilities and its position in the Software System are clear.
The System Elements interfaces have been agreed.
The System Element has been implemented and tested, and is believed to be ready for integration into the Software System.
The System Element has been verified and is ready for live use as part of the Software System..

Figure 20 – The states of System Element

Elements in Essence terms. Essence recognizes universal states that all system elements progress through during the development of a Software System.

Progressing the System Elements

A Software System is not usually developed as a single solid block. It is built from a numbers of System Elements, each of which may be specially built or acquired from elsewhere. During their development System Elements progress through several state changes. As shown in Figure , they are *identified*, *interfaces defined*, *developed* and *ready*. These states focus on providing clear understanding of System Element states.

As indicated in Figure , the first thing to do is to identify which System Elements are needed and assign them their responsibilities within the overall Software System. Once the System Element is *identified* its role and position in the Software System is known and the decision can be made about how to source it. The next step is to refine the System Elements responsibilities and make sure its interfaces are agreed. When the System Element is *interfaces agreed* its relationship with the other System Elements, and where necessary other systems, are defined. The Team can now complete the implementation and testing of the System Element progressing it to the *developed* state. Finally, after all the required testing is complete, the System Element is *ready* for live use as part of the Software System.

Checking the progress of a System Element

To help assess the state and progress of a System Element, the following checklists are provided:

Table 16 – Checklist for System Element

State	**Checklist**
Identified	• The need for the System Element is recognized. • The System Element's roles and responsibilities in the Software System are clear. • Any additional Software Systems that need this System Element are identified. • The options about whether to buy or build the System Element have been explored.

	• Any requirements and constraints on the component are known, such as performance requirements or memory utilization constraints.
Interfaces Agreed	• Interfaces of the System Element with the other system elements are defined. • Interfaces of the System Element with other systems are defined. • Buy or build decisions have been made. • It has been specified how other components should interact with the component. • All externally detectable outcomes are specified including data that is returned and events that may be raised.
Developed	• The System Element has been implemented in a way that is conformant with its interfaces. • The System Element implements the operations on its provided interfaces. • The System Element has been verified as conformant with its interfaces by passing all its unit tests. • The System Element is available for integration into the Software System.
Ready	• All the required testing on the System Element is complete. • The System Element can interoperate with the other System Elements in the System. • The System Element can interoperate with any external systems it communicates with. • System Element is available for use in the live environment.

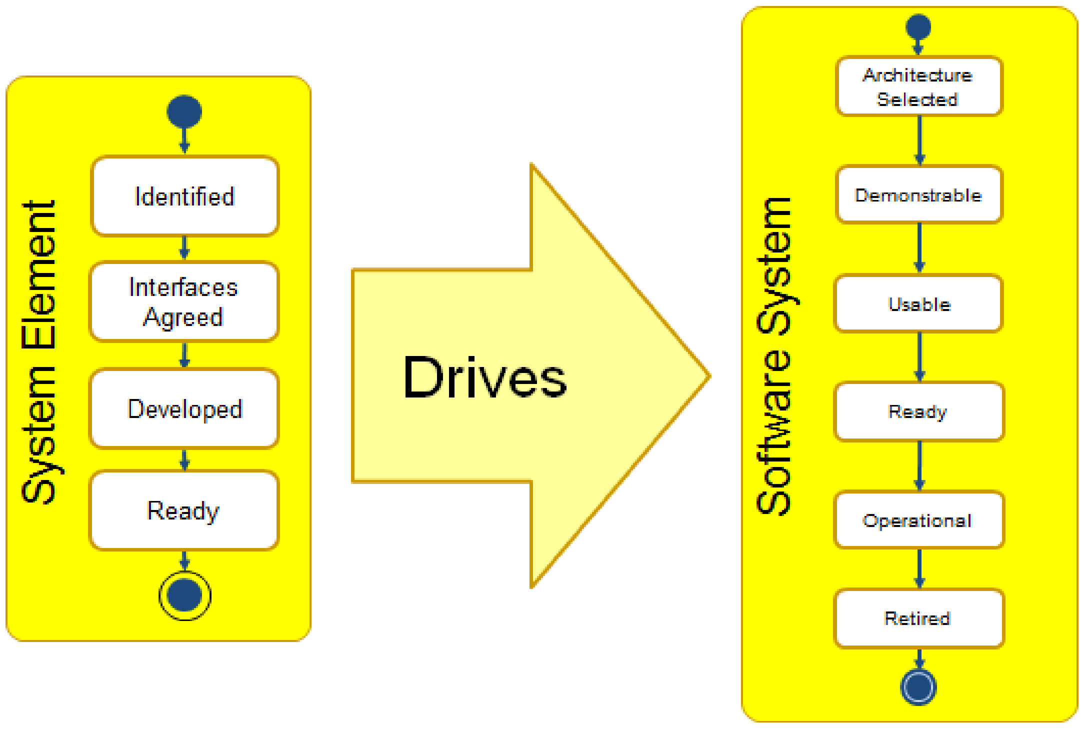


Figure 21 - The System Elements drive the progress of the Software System

How the System Elements drive the progress of the Software System

The progress of the Software System is driven by the system elements composing it. For illustrative purposes the states of the two Alphas are shown in Figure .

How the System Elements drive the progress of the Software System is summarized in Table , along with the additional checklist items that this kernel extension adds to the Software System's checklists.

Table 17 – How the System Elements drive the Software System

Software System State	**How the System Elements drive the progress of the Software System**	**Additional Checklist Items**
Architecture Selected	To progress the Software System to Architecture Selected the System Elements that make up the Software System should be identified and have their Responsibilities Assigned. The core System Elements should also have their interfaces agreed.	The System Elements are all *Identified*. The core System Elements are all *Interfaces Agreed.*
Demonstrable	The core System Elements need to be acquired or developed to be able to assemble a demonstrable Software System.	The core System Elements are all *Developed* and included in the Software System
Usable	Making ready the System Elements that implement the essential characteristics of the system will help the whole system to become usable.	The System Elements that implement the essential characteristics of the system have been made *Ready*.
Ready	Continuing to progress the System Elements will help to continue the progress of the Software System. For the Software System to be ready all of its parts must also be ready.	All of the System Elements that make up the system are *Ready*.
Operational	All the System Elements should remain ready to make, and keep, the Software System operational.	All of the System Elements that make up the system are *Ready*.
Retired	Progress to this state is independent of the state of any of the sub-ordinate System Elements.	None

The state of the individual System Elements is independent of the state of their owning Software System. It is quite possible for the System Elements to change states between *Interfaces Defined* and *Developed* in both forward and backward directions to reflect the need for their further development and maturation. In many cases once a System Element achieves the *Ready* state any additional changes are only allowed if the state is maintained.

9.4 Task Management Extension

9.4.1 Introduction

This extension provides three additional Alphas to allow teams to progress their Team, Work and Way of Working.

9.4.2 Alphas

The task management extension extends the endeavor area of concern adding the following Alphas:

- Team Member as a sub-ordinate of Team
- Task as a sub-ordinate of Work
- Practice Adoption as a sub-ordinate of Way of Working

9.4.2.1 Team Member

Description

Team Member: An individual acting as part of a team.

The Team Members are the group of people that comprise the team.

Super Ordinate Alpha

Team

States

Wanted	A team member with specific skills is sought to join the team.
On Board	The team member is on board and learning how to contribute to the team.
Contributing	The team member is helping her teammates and driving the team's performance
Exiting	The team member is preparing to leave the team.

Associations

drive : Team	The progress of the Team Members drives the progress of the Team.

Justification: Why Team Member

Team Members are needed to form a Team. A Team may range from one to many Team Members. This means that even the smallest Teams have at least one Team Member.

Progressing the Team Members

Team Members progress through a number of states. As indicated in Figure 22, these are *wanted, on-board, contributing*, and *exiting.* These states focus on how well the Team Members are integrated into the team.

First it must be decided that a new team member is *wanted*. In this state the competencies and skills that are required are identified and steps are being taken to find the new Team Member. Once a new team member has been found she needs

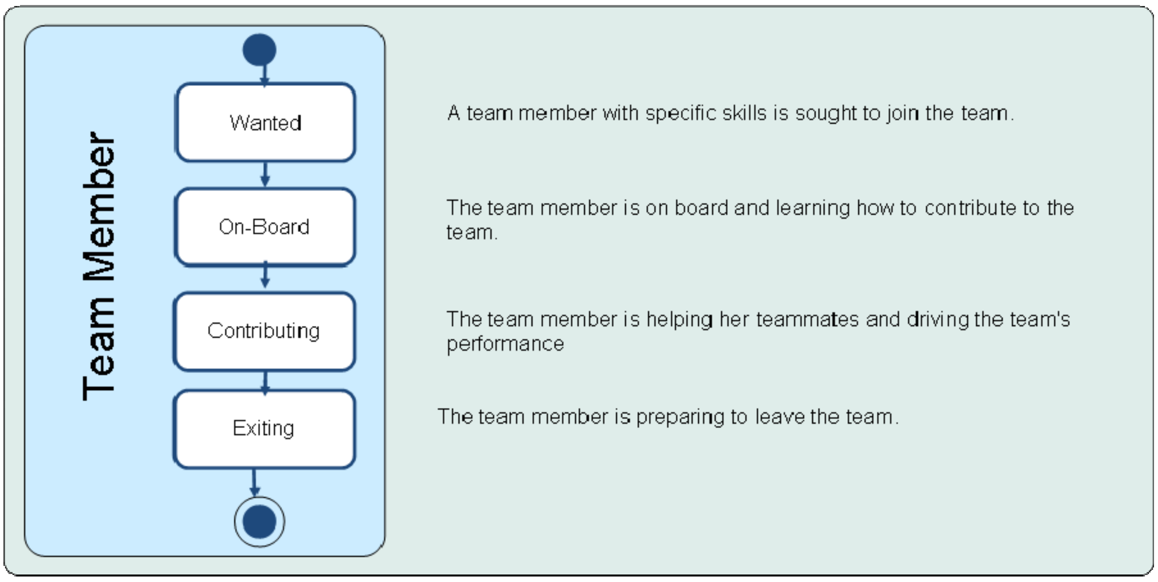

Team Member
Wanted
On-Board
Contributing
Exiting
A team member with specific skills is sought to join the team.
The team member is on board and learning how to contribute to the team.
The team member is helping her teammates and driving the team's performance
The team member is preparing to leave the team.

Figure 22 – The states of Team Member

to be brought *on board*. This means that the Team Member has been selected and inducted into the team, and is ready to learn how to fulfill her responsibilities and overcome any challenges presented by the new role. Over time she will develop and become a *contributing* member of the team. This means that she is actively fulfilling her responsibilities and helping to drive the team's performance.

When a team member decides to leave the team, or is no longer needed by the team, they are considered to be *exiting* and are transitioned out of the team.

Checking the progress of a Team Member

To help assess the state and progress of a Team Member, the following checklists are provided:

Table 18 – Checklist for Team member

State	**Checklist**
Wanted	• The required competencies and skills for a role have been identified • An Individual with required competencies and skills is being sought.
On Board	• Team member has been inducted into the team. • Team member is learning how to contribute to the work and participate on the team. • The gap between the Team Member's actual skills and competencies and those required by their new role are known.
Contributing	• The Team Member is collaborating effectively with teammates. • The Team Member actively contributes to the well-being of the team. • The Team Member can carry out task with little or no supervision.
Exiting	• The Team Member's participation to the team is coming to an end. • The Team Member is handing over their responsibilities to someone else.

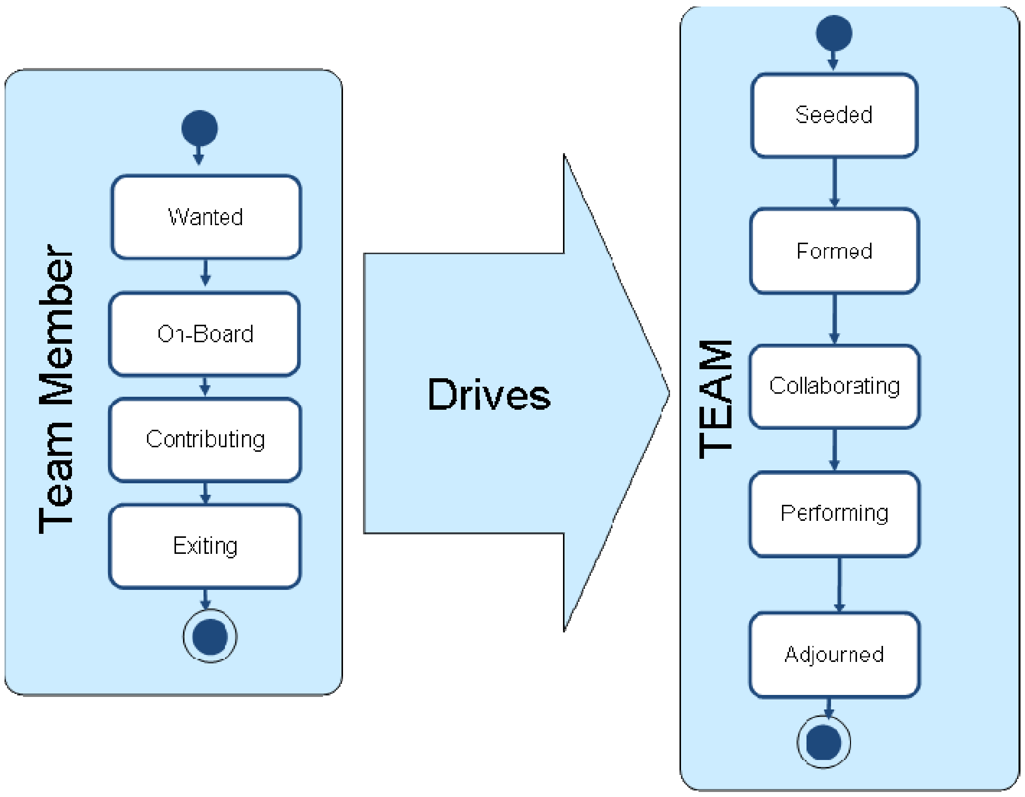


Figure 23 - The Team Members drive the progress of the Team

How the Team Members drive the progress of the Team

The progress of the Team is driven by the associated Team Members. For illustrative purposes the states of the two Alphas are shown in Figure .

How the Team Member Alpha drives the progress of the Team Alpha is summarized in Table , along with the reference to additional checklist items that this kernel extension adds to the Team state checklists.

Table 19 – How the Team Members drive the Team

Team State	**How the Team Members drive the progress of the Team**	**Additional Checklist Items**
Seeded	One or more of the expected Team Members are needed to seed the Team.	One or more key Team Members are *on board*. One or more additional Team Members are *Wanted*.
Formed	The remaining necessary Team Members are recruited to form the team.	All required team members are *on board.*
Collaborating	As the Team members start to work together they drive the team to the collaborating state.	The majority of the team members are actively *contributing* to the success of the team.
Performing	As the Team Members start to work well together and continuously improve their team working they drive the team to the performing	All team members are *contributing* to the success of the team.

	state.	
Adjourned	Finally, when the team is no longer needed it is adjourned.	All team members have exited the team.

The state of the individual Team Members is independent of the states of their owning Team. It is quite possible for one or more Team Members to be c*ontributing* before the Team is c*ollaborating* or p*erforming*. For example you might have some Team Members that are *on-board* but are still being brought up to speed, while others are fully c*ontributing.*

9.4.2.2 Task

Description

<u>Task:</u> A portion of work that can be clearly identified, isolated, and then accepted by one or more team members for completion.

Super Ordinate Alpha

Work

States

Identified	The task has been identified and is ready to be done.
In Progress	The task has been accepted by one or more teammates and work has started.
Done	The work required to do the task has been completed.

Associations

drive : Work	The progress of the Tasks drives the progress of the Work.

Justification: Why Task

Tasks are the fundamental unit of work that team members use to identify and track their work progress.

Progressing the Tasks

Tasks undergo a number of states. As indicated in Figure , these are *identified, in progress, and done.*. These states focus on the management of the Task. Tracking the progress of the tasks is important for tracking the progress of the work.

Tasks are first identified by looking at the Work that needs to be done. When a Task has been *identified*, it means that a small piece of work has been isolated from the work as a whole and is sufficiently understood that it could be accepted by one or more team members for completion. A single task could be written to develop multiple work products, or you could develop multiple tasks to complete a single work product. The granularity of the tasks is proportional to the trust you have in your team members.

Once work starts on the Task it progresses to the *in progress* state during which there is at least one team member actively working on it. Finally the task is *done* when the work required to do the task has been completed. This may be because it has been determined to be completed according to the agreed to completion criteria, which may include the time allowed running out.

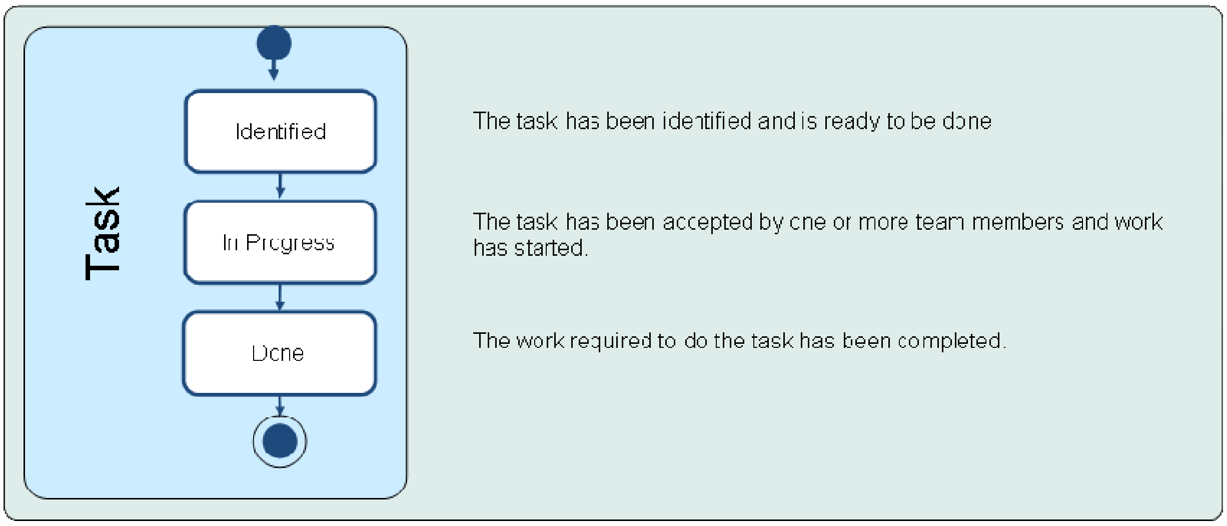

Task
Identified
In Progress
Done
The task has been identified and is ready to be done
The task has been accepted by one or more team members and work has started.
The work required to do the task has been completed.

Figure 24 – The states of the Task

Checking the progress of a Task

To help assess the state and progress of a Task, the following checklists are provided:

Table 20 – Checklist for Task

State	**Checklist**
Identified	• A portion of work has been clearly identified and isolated. • The objective of the Task is clear. • The things that need to be done have been clearly described. • It is clear whether the task is a whole team task, group task or individual task. • The completion criteria for the task are clearly defined. • The effort required to complete the task has been estimated and agreed.
In Progress	• A team member has accepted and is progressing the task. • The progress of the task is monitored. • A target completion date for the Task has been agreed. • The amount of effort required to complete the task is being tracked.
Done	• Work on the Task has stopped. • The task is determined to be complete according to its completion criteria.

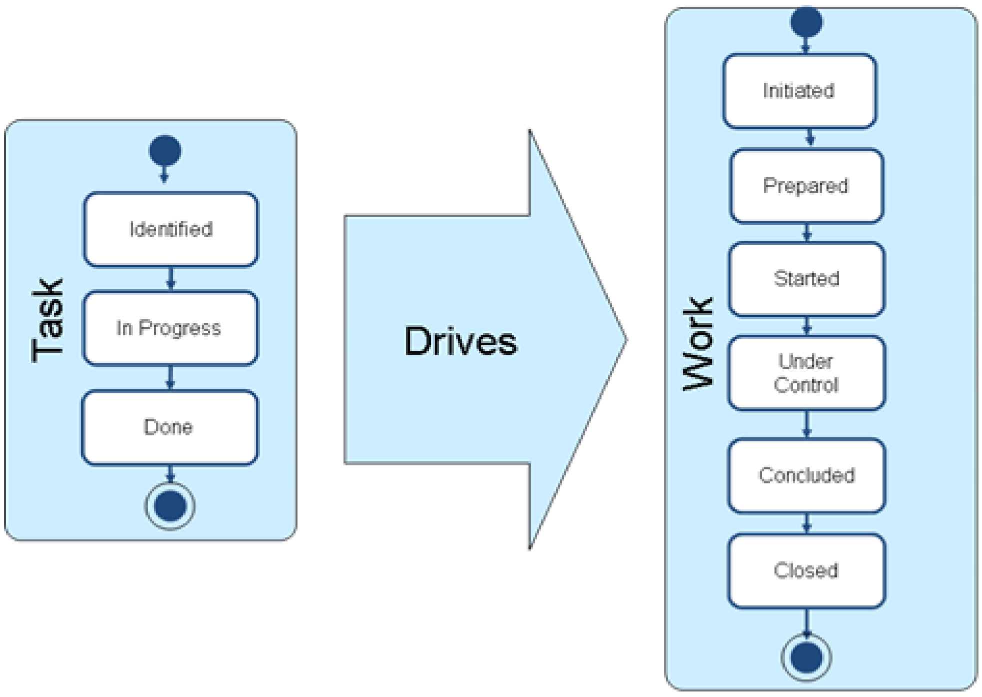


Figure 25 - The Tasks drive the progress of the Work

How the Tasks drive the progress of the Work

The progress of the Work is driven by the associated Tasks. For illustrative purposes the states of the two Alphas are shown in

25.

How the Task Alpha drives the progress of the Work Alpha is summarized in

Table 8, along with the reference to additional checklist items that this kernel extension adds to the Work state checklists.

Table 8 – How the Tasks drive the Work

Work State	**How the Task drive the progress of the Work**	**Additional Checklist Items**
Initiated	First the Tasks needed to prepare the Work are identified as part of the activity to initiate the work. By successfully completing the Tasks the Work will be prepared.	Tasks to be undertaken to prepare the work have been identified.
Prepared	Tasks are identified as part of the activity to prepare the work.	The Tasks to be undertaken to prepare the Work are Done. Enough Tasks have been Identified for the Team to start the real Work.
Started	As Tasks move to the in progress state the work is started.	At least one task has been initiated by a team member

Under control	After sufficient tasks are completed work reaches the under control state.	All team members are effectively working their tasks
Concluded	Finally, when all tasks are done the work is concluded.	All identified tasks are done
Closed	None	None

The state of the individual Tasks are independent of the state of the overall Work. For example, it is quite possible for one or more Tasks to be *In Progress*, or even *Done*, before the Work is *Under Control.*

9.4.2.3 Practice Adoption

Description

Practice Adoption: The adaptation over time of a practice and it's supporting tooling as part of a team's way of working.

Super-Ordinate Alpha

Way of Working

States

Selected	The practice is selected.
Integrated	The practice and related tools have been integrated into the way of working and are ready for use.
In Use	Team members are using the practice and related tools to accomplish their work.
Working well	The adapted practice is working well for the team members.

Associations

drive : Way of Working	The progress of the Practice Adoptions drive the progress of the Way of Working

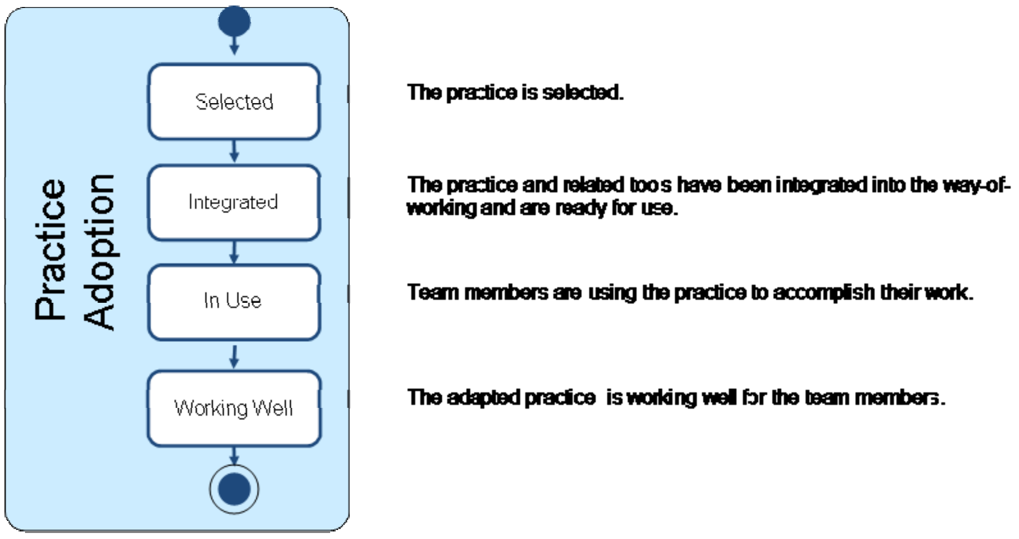

Practice Adoption
Selected
Integrated
In Use
Working Well
The practice is selected.
The practice and related toos have been integrated into the way-of-working and are ready for use.
Team members are using the practice to accomplish their work.
The adapted practice is working well for the team members.

Figure 26 – The states of Practice Adoption

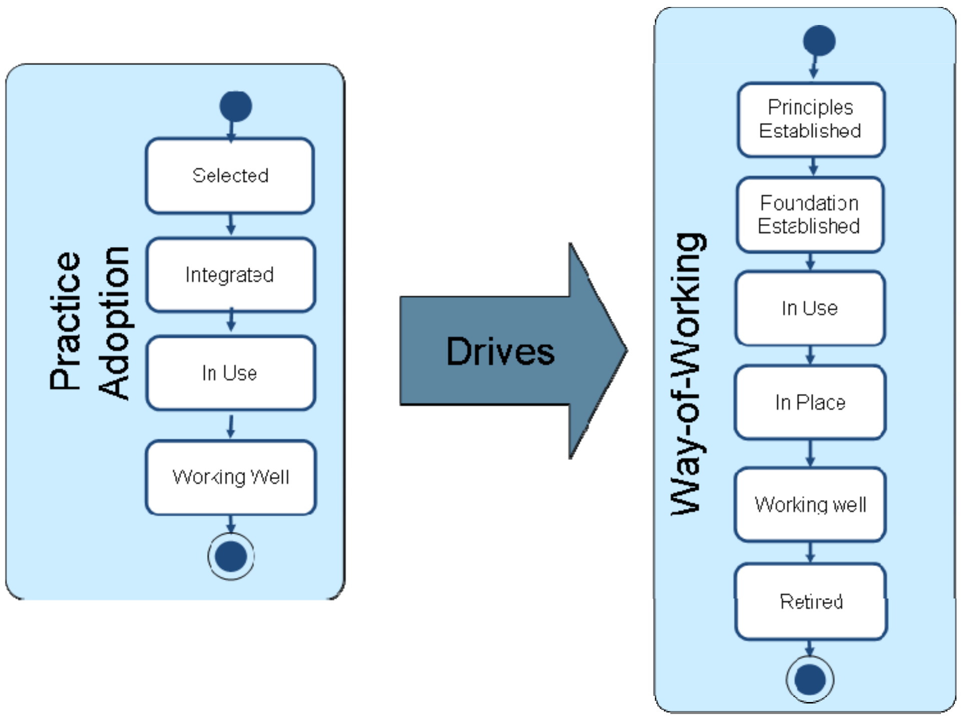


Figure 27- Practice Adoption drive the progress of the Way of Working

Justification: Why Practice Adoption

Teams improve their way of working by adopting and adapting individual practices. Even teams with the simplest way of working have at least one practice.

Progressing the Practice Adoptions

Practice Adoption undergoes a number of states. As indicated in Figure 26, these states are *selected, integrated, using and working well.* These states focus on the progression of practice adoption as they are integrated with tools and other practices. Practice use by the team and their evolution towards *working well* help team members collaborate and complete their tasks effectively.

Checking the progress of a Practice Adoption

To help assess the state and progress of a Practice Adoption, the following checklists are provided:

Table 22 – Checklist for Practice Adoption

State	**Checklist**
Selected	• The practice and related tools have been selected.
Integrated	• The practice has been tailored to meet the constraints of the work environment • The related tools have been integrated to work together with the selected practice and other selected tools
In Use	• The tailored practice is being used by team members to perform their work. • The selected tools are being used by team members integrated with their practice to perform their work.

Working well	• All team members are making progress as planned by using the tailored practice • All team members naturally apply the tailored practice without thinking about it • The practice and related tools are used routinely and effectively by the team. • The practice and tools are regularly being inspected and improved by the team.

How Practice Adoption drives the progress of Way of Working

The progress of the Way of Working is driven by the associated Practice Adoptions. For illustrative purposes the states of the two Alphas are shown in Figure .

How the Practice Adoption Alpha drives the progress of the Way of Working Alpha is summarized in Table 23, along with the reference to additional checklist items that this kernel extension adds to the Way of Working state checklists.

Table 23 – How the Practice Adoption Alpha drives the Way of Work Alpha

Way of Working State	**How the practice adoption drives the progress of the Way of Work**	**Additional Checklist Items**
Principles Established	At least one Practice has been selected in support of the established principles.	At least one Practice has been selected that supports the established principles.
Foundation Established	As each practice and related tools are selected and integrated the Way of Working foundation is established.	At least two practices have been selected and integrated
In Use	Once the foundation is established, the practices and tools are used by team members as part of their way of working.	A sufficient number of practices have been selected and integrated with selected tools to support some of the team member's needs
In Place	The Way of Working is in Place when the selected and integrated practices and tools are used by all team members.	A sufficient number of practices have been integrated to support the team member's needs At least some of the practices and tools are working well for the team
Working Well	As the practices help team members complete their work effectively the way of working reaches a working well state.	All the required practices have been integrated and are supporting all team member's needs.
Retired	None	None

The state of the individual Practice Adoptions are independent of the state of the overall Way of Working. For example, one or more Practices may be *In Use,* or even *Working Well,* before the Way of Working is *Working Well* for the full Team.

10 Language Specification

10.1 Specification Technique

This specification is constructed using a combination of three different techniques: a meta-model, a formal language, and natural language. The meta-model (see Section 10.2) expresses the abstract syntax and some constraints on the structural relationships between the elements. An invariant is provided for each element that, together with the structural constraints in the meta-model, provides the well-formedness rules of the language (the static semantics). The invariants and some additional operations are stated using the Object Constraint Language (OCL) as the formal language used in this document. The composition of elements (see Section 10.3) as well as the dynamic semantics (see Section 10.4) are described using natural language (English) accompanied by a formal calculus where appropriate.

10.1.1 Different Meta-Levels

The meta-model is based upon a standard specification technique using four meta-levels of constructs (meta-classes). These levels are:

- Level 3 – Meta-Language: the specification language, i.e. the different constructs used for expressing this specification, like “meta-class” and “binary directed relationship.”
- Level 2 – Construct: the language constructs, i.e. the different types of constructs expressed in this specification, like “Alpha” and “Activity.”
- Level 1 – Type: the specification elements, i.e. the elements expressed in specific kernels and practices, like “Requirements” and “Find Actors and Use Cases.”
- Level 0 – Occurrence: the run-time instances, i.e. these are the real-life elements in a running development effort.

For a more thorough description of the meta-level hierarchy, see Sections 7.9-7.11 in UML Infrastructure [UML 2011].

10.1.2 Specification Format

Within each section, there is first a brief informal description of the purpose of the elements in that language layer. This is followed by a description of the abstract syntax of these elements together with some of the well-formedness rules, i.e. the multiplicity of the associated elements. The abstract syntax is defined by a CMOF model [MOF 2011], the same language used to define the UML metamodel. Each modeling construct is represented by an instance of a MOF class or association. In this specification, this model is described by a set of UML class and package diagrams showing the language elements and their relationships.

Following the abstract syntax is an enumeration of the elements in alphabetic order. Each concept is described according to:

- **Heading** is the formal name of the language element.
- **Description** is a 1-2 sentence informal brief description of the element. This is intended as a quick reference for those who want only the basic information about an element.
- **Generalizations** lists each of the parents (superclasses) of the language element, i.e. all elements it has generalizations to.
- **Attributes** lists each of the attributes that are defined for that element. Each attribute is specified by its formal name, its type, and multiplicity. This is followed by a textual description of the purpose and meaning of the attribute. The following data types for attributes are used:
    - String
    - Boolean
    - Integer

    - GraphicalElement

- **Associations** lists all the association ends owned by the element. Note that this sub clause does not list the association-owned association ends. The format for element-owned association ends is the same as the one for attributes described above.
- **Invariant** describes the well-formedness rules for language constructs including this element. These are mostly described both with an informal text and with OCL expressions.
- **Additional Operations** describes any additional operations needed when expressing the well-formedness rules. These are mostly described both with an informal text and with OCL expressions. The section is only present when there are any additional operations defined.
- **Semantics** provides a detailed description of the element in natural language.

10.1.3 Notation Used

The following conventions are adopted in the diagrams throughout the specification:

- All meta-class names and class names start with an uppercase letter.
- An association with one end marked by a navigability arrow means that the association is navigable in the direction of that end, the opposite class owns that end, and the association owns the unmarked association end.
- If no multiplicity is shown on an association end, it implies a multiplicity of exactly 1.
- If an association end is unlabeled, the name for that end is the name of the class to which the end is attached, modified such that the first letter is a lowercase letter. (Note that, by convention, non-navigable association ends are often left unlabeled since, in general, there is no need to refer to them explicitly text. However, in some cases, these are used in formal (OCL) expressions.)
- If a class is presented in a diagram of a package and the class is not defined in that package, the full name of that class is used. For instance, AlphaAndWorkProduct::Alpha refers to the class Alpha that belongs to package AlphaAndWorkProduct.

10.2 Language Elements and Language Model

As with most language specifications, this specification defines the elements included in the language (the abstract syntax), some rules for how these elements should be combined to create well-formed language constructs (the static semantics), and a description of the dynamic semantics of the language. In addition, for some of the elements or language constructs a concrete syntax (notation) is also provided.

This section provides the abstract syntax and static semantics of the language by listing and describing the elements in the language and the relationships between them. The elements are grouped into five main metamodel packages as depicted in Figure 5.

- Foundation, contains the base elements to form a minimal core of the language. It contains elements to organize sets of practices.
- AlphaAndWorkProduct, contains the base elements to form minimal practices. A domain model for software engineering endeavors can be created. No activities can be expressed using this layer, but concrete work products can be related to abstract domain elements.
- ActivitySpaceAndActivity, contains elements to enrich practices by expressing activities, skills, and patterns.
- Competency, contains elements to support the specification of competencies and skills.
- View, contains elements to support the specification of view contents.

The ordering between the packages is expressed with import relationships. Each of the packages is described in a separate subsection.

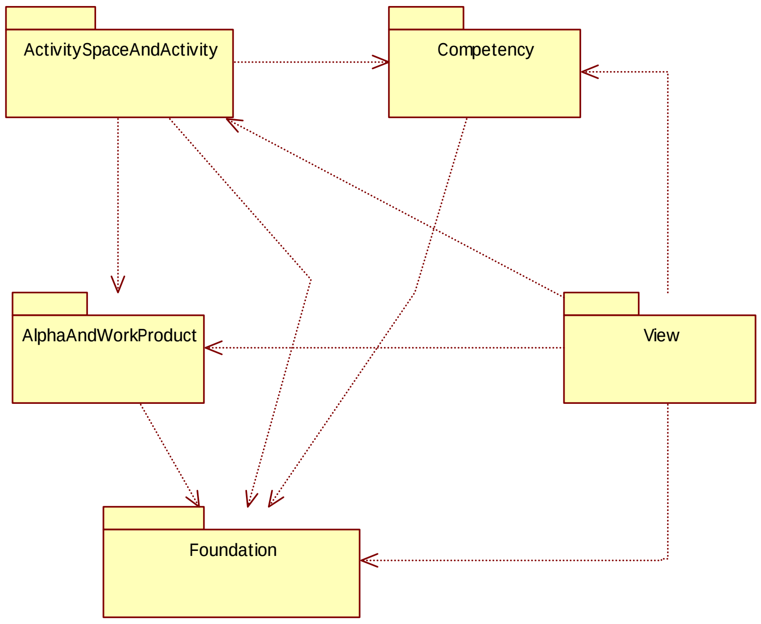


Figure 5 – Structure of the Essence Language metamodel

10.2.1 Foundation

The intention of the Foundation package is to provide all the base elements, including abstract super classes, necessary to form a baseline foundation for the Language. The elements and their relationships are presented in the diagrams below. A detailed definition of each of the elements is found in the following subsections.

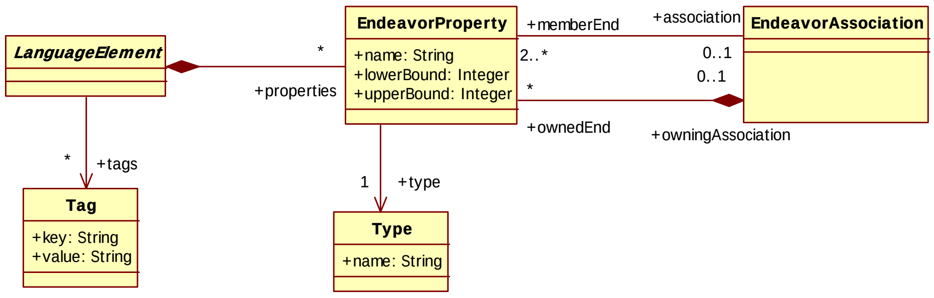


Figure 6 – Foundation::Language element super class

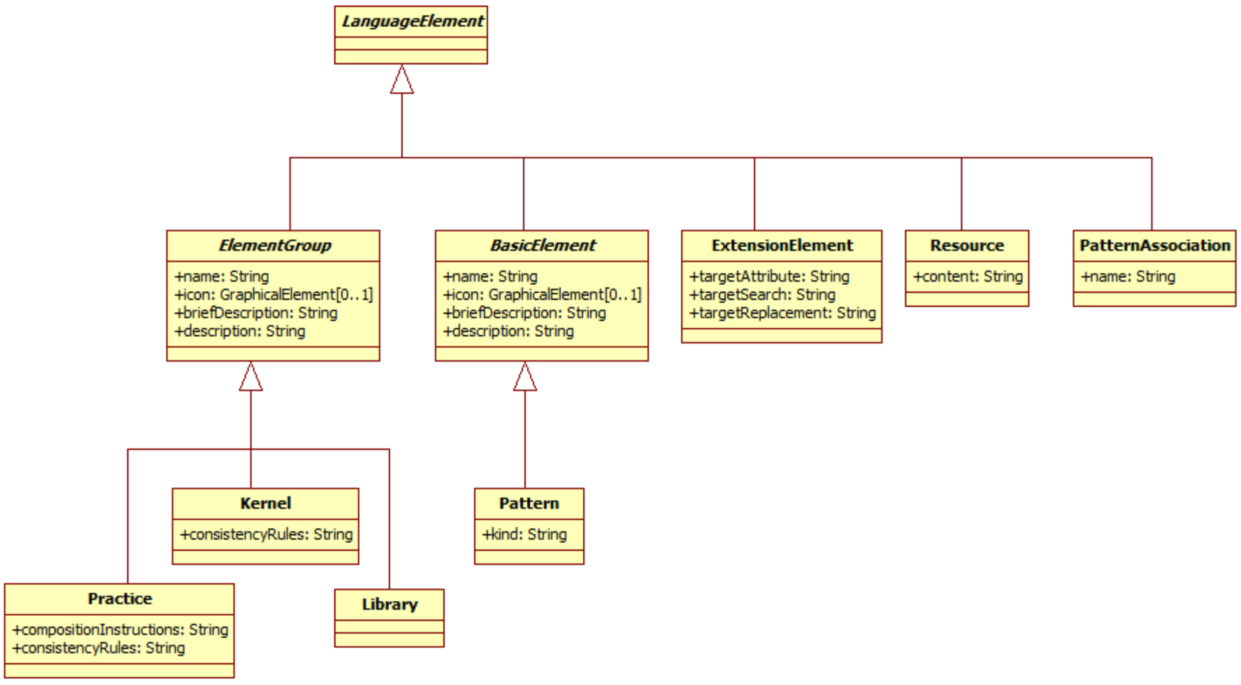
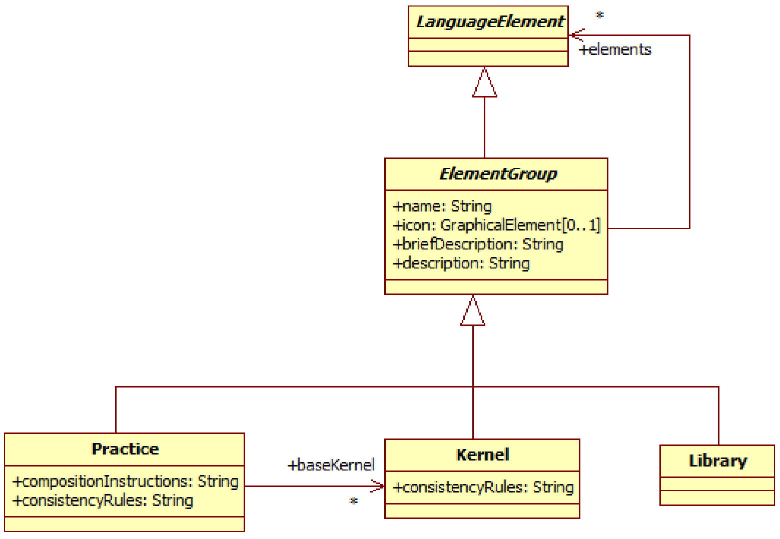


Figure 7 – Foundation::Language elements



LanguageElement
*
+elements
ElementGroup
+name: String
+icon: GraphicalElement[0..1]
+briefDescription: String
+description: String
Practice
+compositionInstructions: String
+consistencyRules: String
+baseKernel
*
Kernel
+consistencyRules: String
Library

Figure 8 – Foundation::Containers

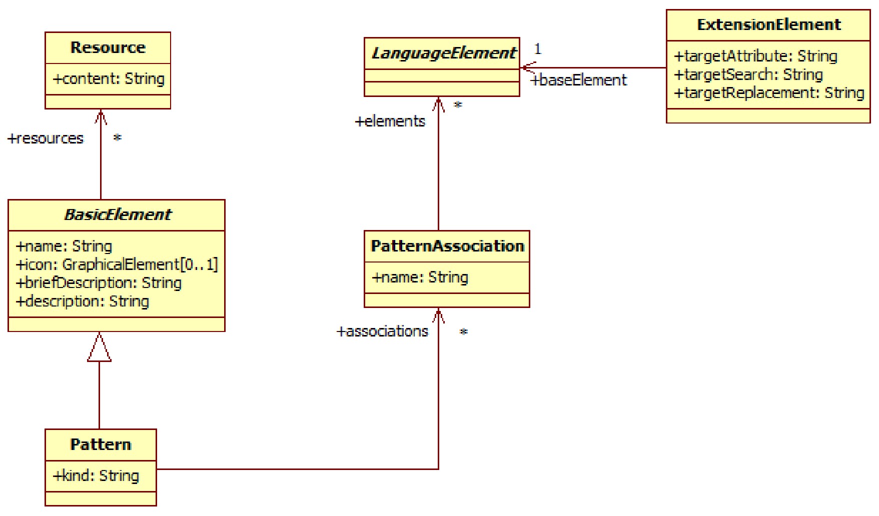


Figure 9 – Foundation::Basic elements

10.2.1.1 BasicElement

Package: Foundation
isAbstract: Yes
Generalizations: "LanguageElement"

Description

A generic name for all main concepts in Essence other than Element groups.

Attributes

name : String [1]	The name of the element.
icon : GraphicalElement [1]	The icon to be used when presenting the element.
briefDescription : String [1]	A short description of what the element is.
description : String [1]	A more detailed description of the element.

Associations

resources : Resource [*]	Resources associated with the element.

Invariant

```
true
```

Semantics

Basic elements are considered to represent the small set of main concepts within Essence. Basic elements are most likely the first elements of Essence a user interacts with. They are also most likely to be presented on individual cards (cf. Section 10.5.4.7) in the graphical syntax.

Elements of Essence which are not basic elements are considered to be auxiliary elements used to detail, connect or group basic elements.

10.2.1.2 ElementGroup

Package: Foundation
isAbstract: Yes
Generalizations: "LanguageElement"

Description

A generic name for an Essence concept that names a collection of elements. Element groups are recursive, so a group may own other groups, as well as other (non-group) elements.

Attributes

name : String [1]	The name of the element group.
icon : GraphicalElement [1]	The icon to be used when presenting the element group.
briefDescription : String [1]	A short description of what the group is.
description : String [1]	A more detailed description of the group.

Associations

elements : LanguageElement [*]	The contained language elements.

Invariant

```
-- An element group may not contain itself
self.allElements(ElementGroup)->excludes(self)
```

Additional Operations

```
-- Get all elements of a particular type which are available within this group
and its referenced groups.
context ElementGroup::allElements (t : Type) : set(t)
body: self.elements->select(e | e.oclIsKindOf(t))-
>union(self.allElements(ElementGroup)->collect(c | c.allElements(t))
```

Semantics

Element groups are used to organize Essence elements into meaningful collections such as Kernels or Practices. Elements in a particular group belong together for some reason, while elements outside that group do not belong to them. The reasoning for including elements in the group should be given in the description attribute of the group.

Element groups own all their members by reference.

If an element group owns two or more members of the same type and name, composition (cf. section 10.3) is applied to them so that only one merged element of that type with that name is visible when viewing the contents of the element group.

10.2.1.3 EndeavorAssociation

Package: Foundation
isAbstract: No
Generalizations:

Description

Represents associations that you want to track during an endeavor.

Attributes

N/A

Associations

memberEnd: EndeavorProperty [2..*]	End properties of the association.
ownedEnd: EndeavorProperty [*]	The properties of this association.

Invariant

```
true
```

Semantics

Endeavor associations are used to link actual instances of elements on the endeavor level. This can be used for instance to keep track on which particular document (an instance of a work product) was created by which particular team member (an instance of alpha “Team member”). In general, these associations have no specific semantics within Essence.

10.2.1.4 EndeavorProperty

Package: Foundation
isAbstract: No
Generalizations:

Description

An element to represent properties that you want to track during an endeavor. Each property can either be simple or be expressed via an association.

Attributes

name: String [1]	Name of the property.
lowerBound: Integer [1]	Lower bound of the property.
upperBound : Integer [1]	Upper bound of the property.

Associations

association : EndeavorAssociation [0..1]	The association used to express this property if it is not a simple property.
owningAssociation : EndeavorAssociation [0..1]	The association owning this property.
type : Type [1]	The type of the property.

Invariant

```
true
```

Semantics

Endeavor properties are used to track individual properties of actual instances of elements during an endeavor. Endeavor properties can be defined individually for language elements. See section 10.4 for the minimal set of endeavor properties that is used by the dynamic semantics of Essence.

10.2.1.5 ExtensionElement

Package: Foundation
isAbstract: No
Generalizations: "LanguageElement"

Description

An element that extends a language element by replacing the content of one of its attributes.

Attributes

targetAttribute : String [1]	The name of the attribute which is to be extended.
targetSearch : String [1]	The content to be matched in the extended attribute.
targetReplacement : String [1]	The new content that replaces the matched content in the extended attribute.

	May be an empty string.

Associations

baseElement : LanguageElement [1]	The element to be extended.

Invariant

```
-- The base element may not be an extension element
not self.baseElement.oclIsKindOf(ExtensionElement)
```

Semantics

If an extension X is associated with a basic element B and referenced by element group C then when B is viewed in C, what is seen is B modified by X. See section 10.3 for the detailed mechanism.

10.2.1.6 Kernel

Package: Foundation
isAbstract: No
Generalizations: "ElementGroup"

Description

A kernel is a set of elements used to form a common ground for describing a software engineering endeavor. A kernel is an element group that names the basic concepts (i.e. alphas, activity spaces and competencies) for a domain (e.g. Software Engineering).

Attributes

consistencyRules : String [1]	Rules on the consistency of a particular Kernel. The format for writing these rules is out of the scope of this specification. It is recommended to use either plain text or OCL.

Associations

N/A

Invariant

```
-- A kernel can only contain alphas, alpha associations, alpha containments,
activity spaces, competencies, kernels, and extension elements.
self.elements->forAll (e | e.oclIsKindOf(Alpha) or
e.oclIsKindOf(AlphaAssociation) or e.oclIsKindOf(AlphaContainment) or
e.oclIsKindOf(ActivitySpace) or e.oclIsKindOf(Competency) or
e.oclIsKindOf(Kernel) or e.oclIsKindOf(ExtensionElement))

-- The alphas associated by alpha associations are available within the kernel or
-- its base kernels.
self.allElements(AlphaAssociation)->forAll (aa | self.allElements(Alpha)-
>includesAll(aa.end))

-- All input and output alphas of the activity spaces are available within the
-- kernel or its base kernels.
self.allElements(ActivitySpace)->forAll (as | self.allElements(Alpha)-
>includesAll(as.input) and self.allElements(Alpha)->includesAll(as.output))

-- Completion criteria are only expressed in terms of states which belong to
alphas which are available in the kernel or its base kernels.
self.allElements(ActivitySpace)->forAll (as | as.completionCriterion->forAll (cc
| cc.state <> null and cc.workProduct = null and self.allElements(Alpha)-
>exists(a | a.states->includes(cc.state))))
```

```
-- The required competencies of the activity spaces are available within the
-- kernel or its base kernels.
activitySpace->forAll (as | as.requiredCompetency->forAll (rc |
self.allCompetencies ()->includes (rc)))

-- All extension elements in the kernel refer to elements that are available
within the kernel or its base kernels.
self.allElements(ExtensionElement)->forAll(e |
self.allElements(e.element.oclType)->includes(e.element))
```

Semantics

A kernel is a kind of domain model. It defines important concepts that are general to everyone when working in that domain, like software engineering development.

A kernel may be defined using other, more basic kernels. For example, a more basic kernel may contain elements that are meaningful to the domain of "Software Engineering" and that may be used in the specific context of "Software Engineering for safety critical" domains as defined by a dependent kernel.

A kernel is closed in that elements in the kernel may only refer to elements which are also part of the kernel or its base kernels.

10.2.1.7 LanguageElement

Package: Foundation
isAbstract: Yes
Generalizations:

Description

A generic name for an Essence concept. A language element may be a basic concept, an auxiliary element or an element group.

Attributes

N/A

Associations

tags : Tag [*]	Tags associated with a language element.
properties : EndeavorProperty [*]	Properties (defined at M1 level) that you want to track during the endeavor.

Invariant

```
-- A language element may not have two tags with the same key
self.tags->forAll(t1, t2 | t1 <> t2 implies t1.key <> t2.key)

-- Make sure each and every instance of LanguageElement may be related to each
other via endeavor associations
LanguageElement::allInstances->forAll(e1,e2 : LanguageElement |
EndeavorAssociation::allInstances->exists(a: EndeavorAssociation | a.member-
>exists(p1,p2 : EndeavorProperty | p1.owningAssociation=e1 and p2.
owningAssociation=e2)))
```

Semantics

Language element is the root for all basic elements, auxiliary elements and element groups. It defines the concepts within the Essence language that can be grouped to build composite entities such as Kernels and Practices.

10.2.1.8 Library

Package: Foundation
isAbstract: No
Generalizations: "ElementGroup"

Description

A library is a container that names a collection of language elements.

Attributes

N/A

Associations

N/A

Invariant

`true`

Semantics

A library contains elements relevant for a specific subject or area of knowledge, like *software development*.

Different to a kernel, the elements in a library do not necessarily form a common ground or vocabulary. Different to a practice, the elements in a library do not necessarily address a particular problem or provide explicit guidance. Instead, a library can be used to set up a meaningful collection of language elements of any scale, from a collection of work products used in a company up to a collection of practices and kernels taught in a university course.

A library is not closed in terms of its elements. I.e. elements in the library may refer to elements which are not part of the library.

10.2.1.9 Pattern

Package: Foundation
isAbstract: No
Generalizations: "BasicElement"

Description

A pattern is a generic mechanism for naming complex concepts that are made up of several Essence elements. A pattern is defined in terms of pattern associations.

Attributes

kind : String [1]	A description of the kind of pattern the element defines.

Associations

associations : PatternAssociation [*]	Named association types between elements.

Invariant

`true`

Semantics

Pattern is a general mechanism for defining a structure of Essence elements. It has a type which describes what kind of pattern it is, like a role or a phase. Typically, the pattern references other elements in a practice or kernel. For example, a role may be defined by referencing required competencies, having responsibility of work products, and participation in activities. Another example could be a phase which groups activity spaces that should be performed during that phase.

10.2.1.10 PatternAssociation

Package: Foundation
isAbstract: No
Generalizations: "LanguageElement"

Description

Pattern associations are used to create named links between the elements of a pattern.

Attributes

name : String [1]	Name of the association.

Associations

elements : LanguageElement [*]	The elements taking part in the pattern via this association.

Invariant

```
-- A pattern association may not refer to other pattern associations, element
groups, or extension elements
self.elements->forAll (e | not e.oclIsKindOf(PatternAssocation) and not
e.oclIsKindOf(ElementGroup) and not e.oclIsKindOf(ExtensionElement))
```

Semantics

Each pattern association introduces elements to take part in a pattern. The name of the pattern association should explain the meaning these elements have inside the pattern. For example, in a pattern defining a toolset there may be a pattern association named “used for” referring to an activity, another pattern association named “used on” referring to a work product, and a third pattern association named “suitable for” referring to a level of detail on the work product that can be achieved with that toolset.

10.2.1.11 Practice

Package: Foundation
isAbstract: No
Generalizations: "ElementGroup"

Description

A practice is a description on how to handle a specific aspect of a software engineering endeavor. A practice is an element group that names all Essence elements necessary to express the desired guidance. A practice can be defined as a composition of other practices.

Attributes

compositionInstructions : String [1]	Rules on how conflicts during merging the contents of this Practice are resolved (see section 10.3). The format for writing these rules is out of the scope of this specification.
consistencyRules : String [1]	Rules on the consistency of a particular Practice. The format for writing these rules is out of the scope of this specification. It is recommended to use either plain text or OCL.

Associations

N/A

Invariant

```
-- The alphas and the work products associated by the work product manifests are
-- visible within the practice.
```

```
self.allElements(WorkProductManifest)->forAll (wpm |
self.allElements(Alpha)->includes (wpm.alpha) and
self.allElements(WorkProduct)->includes (wpm.workProduct)

-- Associated activities are visible within the practice.
self.allElements(ActivityAssociation)->forAll (a | self.allElements(Activity)-
>includesAll (a.end))

-- The activities and the activity spaces associated by the activity manifests of
-- the practice are all visible within the practice.
self.allElements(ActivityManifest)->forAll (am | self.allElements(Activity)-
>includesAll (am.activity) and self.allElements(ActivitySpace)->includes
(am.activitySpace))

-- All activities' input and output work products and input and output alphas are
-- available within the practice.
self.allElements(Activity)->forAll (a | self.allElements(WorkProduct)-
>includesAll (a.inputWorkProduct) and self.allElements(WorkProduct)->includesAll
(a.outputWorkProduct) and self.allElements(Alpha)->includesAll (a.inputAlpha) and
self.allElements(Alpha)->includesAll (a.outputAlpha))

-- Completion criteria are only expressed in terms of states which belong to
alphas or levels of detail which belong to work products which are available in
the practice.
self.allElements(ActivitySpace)->forAll (as | as.completionCriterion->forAll (cc
| (cc.state <> null and cc.workProduct = null and self.allElements(Alpha)-
>exists(a | a.states->includes(cc.state))) or (cc.state = null and cc.workProduct
<> null and self.allEments(WorkProduct)->exists(wp | wp.levelsOfDetail-
>includes(cc.workProduct)))))

-- The activities' required competencies are visible within the practice.
self.allElements(Activity)->forAll(a | self.allElements(Competency)->exists (c |
c.possibleLevel->includes (a.requiredCompetencyLevel))

-- All elements associated with a patterns are visible within the practice.
self.allElements(Pattern)->forAll (p | p.associations->forAll (pa | pa.elements-
>forall (pae | self.allElements(pae.oclType)->includes(pae))
```

Semantics

A practice addresses a specific aspect of development or teamwork. It provides the guidance to characterize the problem, the strategy to solve the problem, and instructions to verify that the problem has indeed been addressed. It also describes what supporting evidence, if any, is needed and how to make the strategy work in real life.

A practice includes its own verification, providing it with a clear goal and a way of measuring its success in achieving that goal.

As might be expected, there are several different kinds of practices to address all different areas of development and teamwork, including (but not limited to):

- Development Practices – such as practices for developing components, designing user interfaces, establishing an architecture, planning and assessing iterations, or estimating effort.
- Social Practices – such as practices on teamwork, collaboration, or communication.
- Organizational Practices – such as practices on milestones, gateway reviews, or financial controls.

Except trivial examples, a practice does not capture all aspects of how to perform a development effort. Instead, the practice addresses only one aspect of it. To achieve a complete description, practices can be composed. The result of composing two practices is another practice capturing all aspect of the composed ones. In this way, more complete and powerful practices can be created, eventually ending up with one that describes how an effort is to be performed, i.e. a method.

The definition of a practice may be based on elements defined in a kernel. These elements, like alphas, may be used (and extended) when defining elements specific to the practice, like work products.

A practice may be a composition of other practices. All elements of the other practices are merged and the result becomes a new practice (see Section 10.3 for the definition of composition).

A practice is closed in that elements in the practice may only refer to elements which are also part of the practice or the element groups this practice relates to.

10.2.1.12 Resource

Package: Foundation
isAbstract: No
Generalizations: "LanguageElement"

Description

A source of information or content, such as a website, that is outside the Essence model and referenced from it, for instance by a URL.

Attributes

content : String [1]	A reference to the content of the resource.

Associations

N/A

Invariant

```
true
```

Semantics

Resources are used to make information available from an Essence model without translating this information into terms of Essence elements and their attributes explicitly. This can for instance be used if the formal model should be kept small for some reason while storing additional information informally in resources. It can also be used of a complex practice or method is to be adopted partially in Essence, while the full practice or method description lives as an external resource outside the Essence model.

10.2.1.13 Tag

Package: Foundation
isAbstract: No
Generalizations:

Description

A label that can be attached to a language element. This enables the creation of user-defined classification schemes for the content of a model.

Attributes

key : String [1]	Name of the key.
value : String [1]	Name of the value.

Associations

elements : LanguageElement [*]	Model elements that are tagged.

Invariant

```
-- Key and value may not be empty
not self.key.empty() and not self.value.empty()
```

Semantics

Tagging allows to add user defined or tool specific information to any language element. It is up to the user or tool vendor who applied the tags to interpret them.

10.2.2 AlphaAndWorkProduct

The intention of the AlphaAndWorkProduct package is to provide the basic elements needed for the simplest form of practices. The elements and their relationships are presented in the diagrams below. A detailed definition of each of the elements is found below.

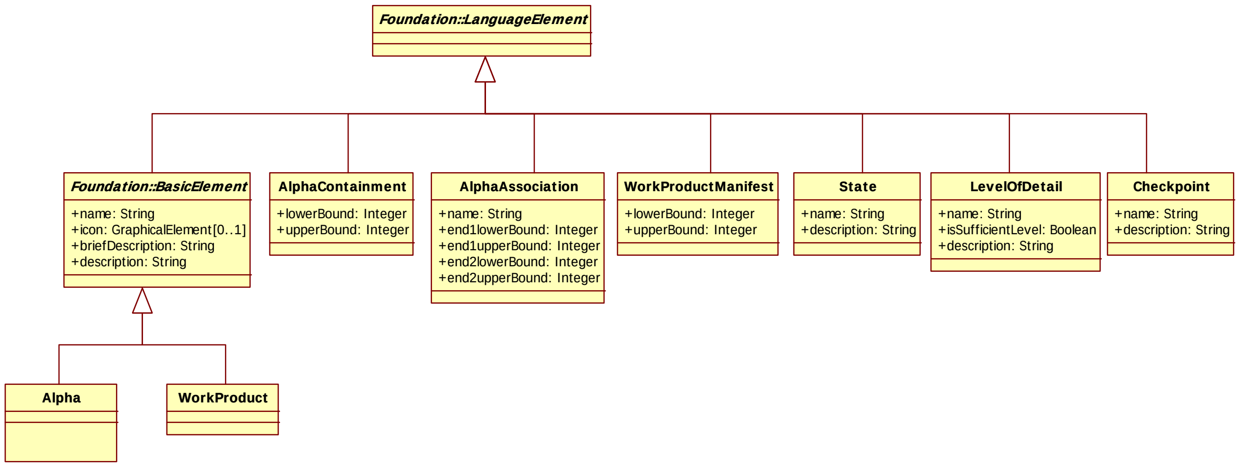


Figure 10 – AlphaAndWorkProduct::Language elements

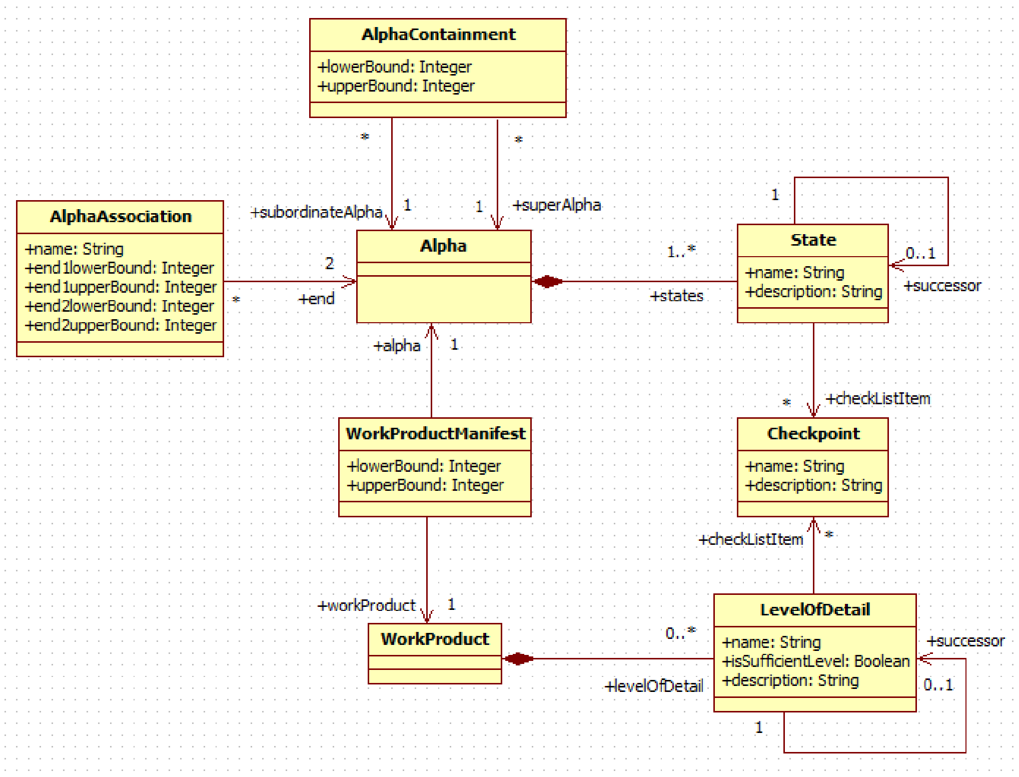


Figure 11 – AlphaAndWorkProduct::Alpha and work product

10.2.2.1 Alpha

Package: AlphaAndWorkProduct
isAbstract: No
Generalizations: "BasicElement"

Description

An essential element that is relevant to an assessment of the progress and health of a software engineering endeavor.

An alpha represents and holds the state of some element, aspect or abstraction in an endeavour that has a discernable state and knowledge of whose state is required to understand the state of progress and/or health of the endeavour.

The instances of alphas in an endeavour form acyclic graphs. These graphs show how the states of lower level, more granular instances, contribute to and drive the states of the higher level, more abstract, alphas.

Attributes

N/A

Associations

states : State [1..*]	The states of the alpha.

Invariant

```
-- All states of an alpha must have different names.
self.states->forAll(s1, s2 | s1 <> s2 implies s1.name <> s2.name)
```

Semantics

Alpha is an acronym that means “Abstract-Level Progress Health Attribute.”

Alphas are subjects whose evolution we want to understand, monitor, direct, and control. The major milestones of a software engineering endeavor can be expressed in terms of the states of a collection of alphas. Thus, alpha state progression means progression towards achieving the objectives of the software engineering endeavor.

An alpha has well-defined states, defining a controlled evolution throughout its lifecycle – from its creation to its termination state. Each state has a collection of checkpoints that describe what the alpha should fulfill in this particular state. Hence it is possible to accurately plan and control their evolution through these states.

An alpha may be used as input to an activity space in which the content of the alpha is used when performing the work of the activity space. The alpha (and its state) may be created or updated during the performance of activities in an activity space.

An alpha is often manifested in terms of a collection of work products. These work products are used for documentation and presentation of the alpha. The shape of these work products may be used for concluding the state of the alpha.

Different practices may use different collections of work products to document the same alpha. For example, one practice may document all kinds of requirements in one document, while other practices may use different types of documents. One practice may document both the flow and the presentation of a use case in one document, while another practice may separate the specification of the flow from the specification of the user interface and write them in different documents.

An alpha may contain a collection of other alphas. Together, these sub-alphas contribute to the state of the superordinate alpha. However, there is no explicit relationship between the states of the subordinate alphas and the state of their superordinate alpha.

10.2.2.2 AlphaAssociation

Package: AlphaAndWorkProduct
isAbstract: No
Generalizations: "LanguageElement"

Description

Alpha association is used to represent a relationship between alphas. Generally these associations are defined by a practice.

Attributes

end1LowerBound : Integer [1]	Lower bound of association endpoint 1.
end1UpperBound : Integer [1]	Upper bound of association endpoint 1.
end2LowerBound : Integer [1]	Lower bound of association endpoint 2.
end2UpperBound : Integer [1]	Upper bound of association endpoint 2.
name : String [1]	Name of the alpha association.

Associations

end : Alpha [2]	The alpha endpoints of the association.

Invariant

```
true
```

Semantics

Unlike a relationship between alphas defined using alpha containment, which is used for the Essence “sub-alpha” relationship, a relationship between alphas defined using alpha association has no defined semantics in Essence. An example would be between a Risk and the Team Member who identified the Risk. While Risk Management practice might recommend that this relationship be tracked, it is not a sub-alpha relationship.

A relationship modeled by an alpha association can, in general, be many-to-many.

10.2.2.3 AlphaContainment

Package: AlphaAndWorkProduct
isAbstract: No
Generalizations: "LanguageElement"

Description

Alpha association is used to represent a sub(ordinate)-alpha relationship between alphas.

Attributes

lowerBound : Integer [1]	Lower bound for the number of instances of the sub(ordinate)-alpha.
upperBound : Integer [1]	Upper bound for the number of instances of the sub(ordinate)-alpha.

Associations

superAlpha : Alpha [1]	The super alpha.
subordinateAlpha : Alpha [1]	The subordinate alpha.

Invariant

```
true
```

Semantics

The sub-alpha relationships define the graphs that show how the states of lower level, more granular alpha instances contribute to and drive the states of the higher level, more abstract, alpha instances.

The relationship between a sub(ordinate)-alpha and a super-alpha can, in general, be many-to-many. The ends of the relationship are modeled separately to indicate which is the sub(ordinate)-alpha and which is the super-alpha of the relationship.

10.2.2.4 Checkpoint

Package: AlphaAndWorkProduct
isAbstract: No
Generalizations: "LanguageElement"

Description

A condition that can be tested as true or false that contributes to the determination of whether a state (of an alpha) or a level of detail (of a work product) has been attained.

Attributes

name : String [1]	The name of the checkpoint.
description : String [1]	A description of the checkpoint.

Associations

N/A

Invariant

```
true
```

Semantics

Checkpoints are used as follows:

- The checkpoints of an alpha state are joined by AND. The state of an alpha is deemed to be the most advanced (favourable) state for which all checkpoints are true.
- The checkpoints of a work product level of detail are joined by OR. The level of detail of a work product is deemed to be the most detailed level for which at least one checkpoint is true.

10.2.2.5 LevelOfDetail

Package: AlphaAndWorkProduct
isAbstract: No
Generalizations: "LanguageElement"

Description

A specification of the amount of detail or range of content in a work product. The level of detail of a work product is determined by evaluating checklist items.

Attributes

description : String [1]	A description of the level of detail.
isSufficientLevel : Boolean [1]	Boolean value determined by the practice (author) to indicate the sufficient level of detail.
name : String [1]	Name of the level of detail.

Associations

checkListItem : Checkpoint [*]	Checklist items to determine if the level of detail has been reached.
successor: LevelOfDetail [0..1]	Next level of detail.

Invariant

```
-- All checkpoints of a level of detail must have different names
self.checkListItem->forAll(i1, i2 | i1 <> i2 implies i1.name <> i2.name)
```

```
-- A level of detail may not be its own direct or indirect successor
self.allSuccessors()->excludes(self)
```

Additional Operations

```
-- All successors of a level of detail
context LevelOfDetail::allSuccessors : Set(LevelOfDetail)
body: Set{self.successor}->union(self.successor.allSuccessors())
```

Semantics

Levels of detail describe the amount and granularity of information that is present in a work product. For example, they allow to distinguish between a sketch of a system architecture, a formally modeled system architecture, and an annotated system architecture which is ready for code generation. It depends on the practice which of these levels is considered sufficiently detailed.

It is important to note that levels of detail are not concerned with the completeness of a work product. A work product can be considered complete for the purpose of the endeavor without being in the most advanced level of detail. In turn, a work product can be in the most advanced level of detail, but not yet been completed.

10.2.2.6 State

Package: AlphaAndWorkProduct
isAbstract: No
Generalizations: "LanguageElement"

Description

A specification of the state of progress of an alpha. The state of an alpha is determined by evaluating checklist items.

Attributes

name : String [1]	The name of the state.
description : String [1]	Some additional information about the state.

Associations

checkListItem : Checkpoint [*]	A collection of checkpoints associated with the state.
successor : State [0..1]	The successor state.

Invariant

```
-- All checkpoints of a state must have different names
self.checkListItem->forAll(i1, i2 | i1 <> i2 implies i1.name <> i2.name)

-- A state may not be its own direct or indirect successor
self.allSuccessors()->excludes(self)
```

Additional Operations

```
-- All successors of a state
context State::allSuccessors : Set(State)
body: Set{self.successor}->union(self.successor.allSuccessors())
```

Semantics

A state expresses a situation in which all its associated checklist items are fulfilled. It is considered to be an important and remarkable step in the lifecycle of an alpha.

10.2.2.7 WorkProduct

Package: AlphaAndWorkProduct
isAbstract: No
Generalizations: "BasicElement"

Description

A work product is an artifact of value and relevance for a software engineering endeavor. A work product may be a document or a piece of software, but also other created entities such as:

- Creation of a test environment
- Delivery of a training course

Attributes

N/A

Associations

levelOfDetail: LevelOfDetail [0..*]	The level of details defined for the work product.

Invariant

```
-- All levels of detail of a work product must have different names
self.levelOfDetail->forAll(l1, l2 | l1 <> l2 implies l1.name <> l2.name)
```

Semantics

A work product is a concrete representation of an alpha. It may take several work products to describe the alpha from all different aspects.

A work product can be of many different types such as models, documents, specifications, code, tests, executables, spreadsheets, as well as other types of artifacts. In fact, some work products may even be tacit (conversations, memories, and other intangibles).

Work products may be created, modified, used, or deleted during an endeavor. Some work products constitute the result of (the deliverables from) the endeavor and some are used as input to the endeavor.

A work product could be described at different levels of details, like overview, user level, or all details level.

10.2.2.8 WorkProductManifest

Package: AlphaAndWorkProduct
isAbstract: No
Generalizations: "LanguageElement"

Description

A work product manifest binds a work product to an alpha.

Attributes

lowerBound : Integer [1]	Lower bound for the number of instances of the work product associated to one instance of the alpha.
upperBound : Integer [1]	Upper bound for the number of instances of the work product associated to one instance of the alpha.

Associations

alpha : Alpha [1]	The alpha bound by this manifest.
workProduct : WorkProduct [1]	The work product bound by this manifest.

Invariant

`true`

Semantics

Work product manifest represents a tri-nary relationship. It is a relationship from a practice to a work product which is used for describing an alpha. Several work products may be bound to the same alpha, i.e. there may be multiple alpha manifests within a practice binding a specific alpha to different work products.

For each work product manifest, there is a multiplicity stating how many instances there should be of the associated work product describing one instance of the alpha.

10.2.3 ActivitySpaceAndActivity

The intention of the ActivitySpaceAndActivity package is to provide additional elements to deal with more advanced practices. The elements and their relationships are presented in the diagrams shown below. A detailed definition of each of the elements is found below.

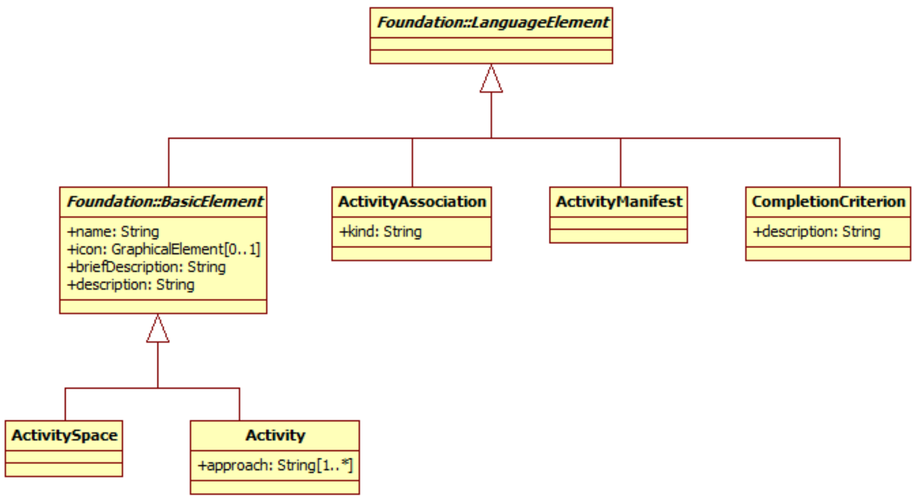


Figure 12 – ActivitySpaceAndActivity::Language elements

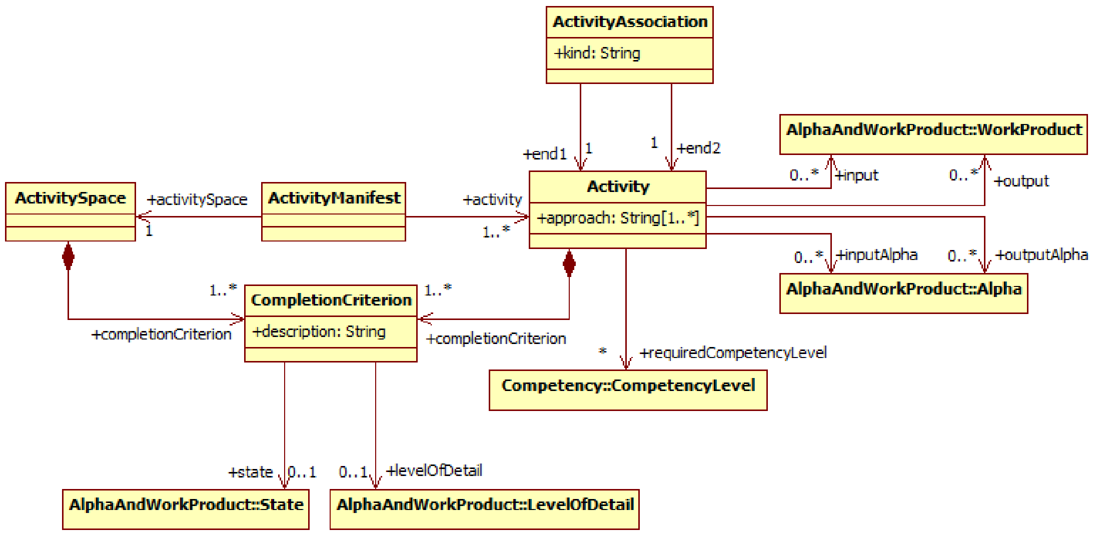


Figure 13 – ActivitySpaceAndActivity::Activity space and activity

10.2.3.1 Activity

Package: ActivitySpaceAndActivity
isAbstract: No
Generalizations: "BasicElement"

Description

An Activity defines one or more kinds of work product and one or more kinds of task, and gives guidance on how to use these in the context of using some practice.

Attributes

approach : String [1..*]	Different approaches to accomplish the activity.

Associations

completionCriterion : CompletionCriterion [1..*]	A collection of completion criteria that have to be fulfilled for considering the activity completed.
requiredCompetencyLevel : CompetencyLevel [*]	A collection of competencies required for completing this activity successfully.
inputAlpha : Alpha [*]	A collection of Alphas which need to be present in order to start this activity.
outputAlpha : Alpha [*]	A collection of Alphas that will be present when this activity is completed successfully.
inputWorkProduct : WorkProduct [*]	A collection of Work Products which need to be present in order to start this activity.
outputWorkProduct : WorkProduct [*]	A collection of Work Products that will be present when this activity is completed successfully.

Invariant

```
true
```

Semantics

An activity describes some work to be performed. It is considered completed if all its completion criteria are fulfilled; whether or not this completion was because of performance of the activity or for some other reason. Performing an activity can normally be expected to result in its completion criteria being fulfilled, but this is not guaranteed.

An activity can take alphas and/or work products as input to the work. Alphas or work products may also be created or updated during the activity, considering them to be the output of the work. However, it is not defined when or how these have been created or updated; only that this has been done when the activity is completed. Alphas and work products used in the completion criteria are considered to be input and output by default. Modeling an alpha or work product as output without using it in a completion criterion indicates that there is no specific relation between the state of that alpha or work product and the completion of the activity. For example, a Sprint Retrospective according to Scrum will have alpha "Way of Working" as an output, because it is possible that the team decides to change the way of working based on the results of the retrospective. However, there is no specific relationship indicating that the Sprint Retrospective can only be considered complete if the alpha "Way of Working" has reached a certain state.

The activity is a manifestation of (part of) an activity space through the activity manifest. The activities filling the same activity space jointly contribute to the achievement of the completion criteria of the activity space. Activities may define different approaches to reach a goal which may imply restrictions on how different activities may be combined. One activity may be bound to multiple activity spaces within a practice.

The activity may be related to other activities via an activity association. The association indicates a relationship between the activities, such as a work breakdown structure. Activity associations do not constrain the completion of the associated activities.

To be likely to succeed with the activity, the performer(s) of the activity must have at least the competencies required by the activity to be able to perform that activity with a satisfactory result.

10.2.3.2 ActivityAssociation

Package: ActivitySpaceAndActivity
isAbstract: No
Generalizations: "LanguageElement"

Description

Activity association is used to represent a relationship or dependency between activities. Generally these dependencies are defined by the practice that defines the activities.

Attributes

kind : String [1]	The kind of the association.

Associations

end1 : Activity [1]	The first member of the association.
end2 : Activity [1]	The second member of the association.

Invariant

```
-- The members of the association are distinct
self.end1 <> self.end2
```

Semantics

Activities can be related to each other via activity associations. They define relationships or dependencies between activities, but do not constrain their completion.

If the kind of the association is “part-of”, the first member of the association is considered to be part of the second member in a work breakdown structure.

If the kind of the association is “start-before-start”, it is suggested to start the first member before starting the second member.

If the kind of the association is “start-before-end”, it is suggested to start the first member before finishing the second member.

If the kind of the association is “end-before-start”, it is suggested to finish the first member before starting the second member. This may imply that the second member cannot be started before the first member is finished.

If the kind of the association is “end-before-end”, it is suggested to finish the first member before finishing the second member. This may imply that the second member cannot be finished before the first member is finished.

However, in any case a member is considered complete if its completion criteria are met, independent of the completion of its associated activities.

10.2.3.3 ActivityManifest

Package: ActivitySpaceAndActivity
isAbstract: No
Generalizations: "LanguageElement"

Description

An activity manifest binds a collection of activities to an activity space.

Attributes

N/A

Associations

activitySpace : ActivitySpace [1]	The activity space filled by this manifest.
activity : Activity [1..*]	The activities bound to the activity space.

Invariant

`true`

Semantics

The relationship between an activity and an activity space is many to one.

10.2.3.4 ActivitySpace

Package: ActivitySpaceAndActivity
isAbstract: No

Generalizations: "LanguageElement"

Description

A placeholder for something to be done in the software engineering endeavor.

Attributes

N/A

Associations

completionCriterion : CompletionCriterion [1..*]	A collection of completion criteria that have to be fulfilled for considering the objectives of this activity space to be fulfilled.
input : Alpha[*]	A collection of alphas that have to be present to be successful in fulfilling the objectives of this activity space.
output : Alpha [*]	A collection of alphas that will be present when the objectives of this activity space have been fulfilled.

Invariant

`true`

Semantics

An activity space is a high-level abstraction representing “something to be done”. It uses a (possibly empty) collection of alphas as input to the work. When the work is concluded a collection of alphas (possibly some of the alphas used as input) has been updated. The update may cause a change of the alpha’s state. When the update and the state change of an alpha takes place is not defined; only that it has been done when the activity space is completed.

What should have been accomplished when the work performed in the activity space is completed, i.e. the activity space’s completion criteria, is expressed in terms of which states the output alphas should have reached. Using the checkpoints for the states of alphas, it is at the discretion of the team to decide when a state change has occurred and thus the completion criteria of the activity space have been met.

10.2.3.5 CompletionCriterion

Package: ActivitySpaceAndActivity
isAbstract: No
Generalizations: "LanguageElement"

Description

A condition that can be tested as true or false that contributes to the determination of whether an activity or an activity space is complete. A completion criterion is expressed in terms of the state of an alpha or the level of detail of a work product.

Attributes

description : String [1]	A description of the criterion which is to be reached at the target state of an alpha or the level of detail of a work product.

Associations

state : State [0..1]	A state to be reached.
levelOfDetail : LevelOfDetail [0..1]	A level of detail to be reached.

Invariant

```
-- A completion criterion addresses either a state or a level of detail
(self.state <> null and levelOfDetail = null) or (self.state = null and
levelOfDetail <> null)
```

Semantics

The work of an activity or activity space is considered complete when its completion criteria are fulfilled, i.e. when the alpha states or work product levels of detail defined by the completion criteria are reached.

10.2.4 Competency

The intention of the Competency package is to provide facilities to add competencies to practices. The elements and their relationships are presented in the diagrams shown below. A detailed definition of each of the elements is found below.

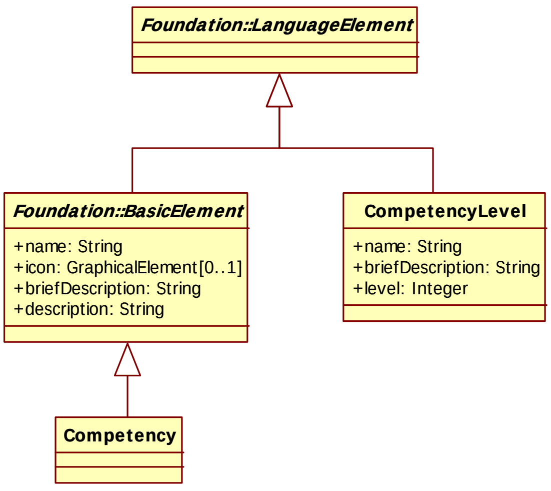


Figure 14 – Competency::Language elements

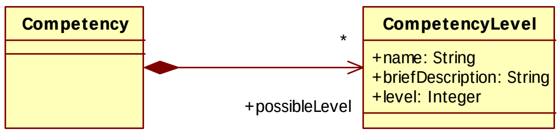


Figure 15 – Competency::Competency

10.2.4.1 Competency

Package: Competency
isAbstract: No
Generalizations: "BasicElement"

Description

A competency describes a capability to do a certain job.

Attributes

N/A

Associations

possibleLevel : CompetencyLevel [*]	A collection of levels defined for this competency.

Invariant

```
-- The possible levels are distinct
self.possibleLevel->forAll (l1, l2 | l1 <> l2 implies (l1.level <> l2.level and
l1.name <> l2.name))
```

Semantics

A competency is used for defining a capability of being able to work in a specific area. In the same way as an Alpha is an abstract thing to monitor and control and an Activity Space is an abstraction of what to do, a Competency is an abstract collection of knowledge, abilities and attitudes. Examples for Competencies that could be defined in a Kernel include “Analyst”, “Developer”, or “Tester”.

10.2.4.2 CompetencyLevel

Package: Competency
isAbstract: No
Generalizations: "LanguageElement"

Description

A competency level defines a level of how competent or able someone is in a subject.

Attributes

name : String [1]	The name of the competency level.
briefDescription : String [1]	A short description of what the competency level is.
level : Integer [1]	A numeric indicator for the level, where a higher number means more/better competence.

Associations

N/A

Invariant

```
true
```

Semantics

Competency levels are used to create a range of abilities from poor to excellent or small scale to large scale. While a competency describes what capabilities are needed (such as “Analyst” or “Developer”), a competency level adds a qualitative grading to them. Typically, the levels range from 0 – no competence to 5 – expert. (such as “basic”, “advanced”, or “excellent”).

10.2.5 View

A user interacts through the realization of one or more views as he or she works according to a kernel, practice or method. The views provide a means for users to interact with a relevant subset, and relevant details, of Essence language constructs as they are used to describe a method instance.

The overall objective with the views is to be able to provide the right and purposeful support for different types of users and at different points in time; and as a consequence, help in avoiding information overflow of language construct detail. This is because different types of users have different needs or interests in the details of a method instance description. Some users need very little details whereas others need more.

For this purpose, the Essence language introduces the ViewSelection construct to support the specification of view contents.

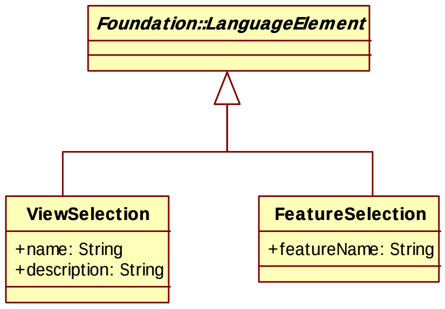


Figure 16 – View::Language elements

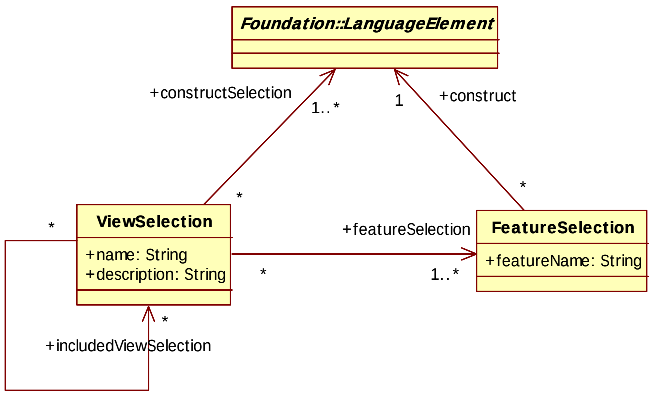


Figure 17 – View::View selection

10.2.5.1 FeatureSelection

Package: View
isAbstract: No
Generalizations: "LanguageElement"

Description

A reference to a construct feature such as a particular attribute or association.

Attributes

featureName : String [1]	The name of the referred feature, such as the name of an attribute or the role name of an association.

Associations

construct : BasicElement [1]	The construct that defines the feature.

Invariant

```
true
```

Semantics

A feature selection names a feature (property or association) from a language construct which is to be included in a view. The feature is identified by its name, since property and association names are unique within a language element. If a feature with the given name does not exist, this feature selection does not contribute anything to the view.

10.2.5.2 ViewSelection

Package: View
isAbstract: No
Generalizations: "LanguageElement"

Description

A ViewSelection selects a subset of constructs and construct features such as attributes and associations.

Attributes

name : String [1]	The name of the view.
description : String [1]	A description of the view, including the purpose of the view.

Associations

constructSelection : LanguageElement [1..*]	The selected constructs (such as Alpha, State, etc) to be included in the view.
featureSelection : FeatureSelection [1..*]	The selected features, such as attributes and associations of constructs to be included in the view.
includedViewSelection : ViewSelection [*]	ViewSelections to be included in this ViewSelection (provides a means to build extended and more sophisticated views based on existing/smaller views).

Invariant

```
-- The featureSelections in a ViewSelection V refers to constructs that are part
of constructSelections in V.
self.featureSelection->forAll(fs | self.constructSelection-
>inludes(fs.construct))
```

Semantics

A view selection names the language constructs to be included in a view. From these constructs, only features named by a feature selection are actually included in the view. A view selection may include other view selections.

A view selection only contains information about the elements and features included in a view. It does not contain any layout or presentation information.

10.2.5.2.1 Example ViewSelection 1

name: “Alpha state view”

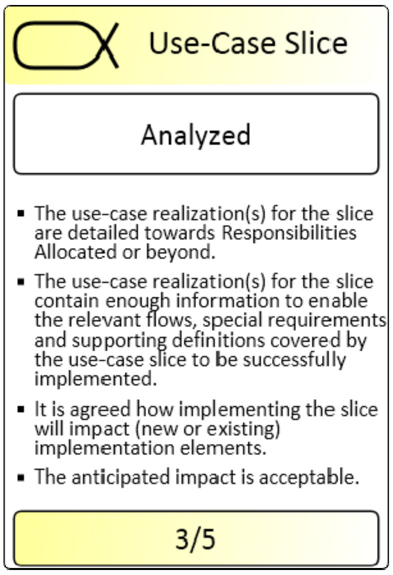
description: “The purpose of this view is to show a particular state of an alpha including the checkpoints of the state”

includedViewSelection: none

Table 9 – Included features for Example ViewSelection 1

Included selection number	Feature name	Basic element
1	name (attribute)	Alpha
2	name (attribute)	State
3	description (attribute)	Checkpoint
4	states (role name)	Alpha

This example ViewSelection can be realized with a state card i.e. the following is one possible implementation of the ViewSelection:



Use-Case Slice

Analyzed

- The use-case realization(s) for the slice are detailed towards Responsibilities Allocated or beyond.
- The use-case realization(s) for the slice contain enough information to enable the relevant flows, special requirements and supporting definitions covered by the use-case slice to be successfully implemented.
- It is agreed how implementing the slice will impact (new or existing) implementation elements.
- The anticipated impact is acceptable.

3/5

So in essence, the ViewSelection helps us define the subset of information to be shown on this specific type of card; however how to visualize the card (read: implementing the view) is not specified by the view itself but is instead something that is supported by the graphical syntax of the language.

In other words, it must be the purpose of the graphical syntax to implement (support) relevant views of the language.

10.2.5.2.2 Example ViewSelection 2

name: “Basic user view”

description: “The purpose of this view is to support a user that has very little interest in methods, but understands the value in having some kind of descriptions of the practices. This is expected to be the largest user group and the one that has high priority. This user will use a minimum number of language constructs that large user groups still can be expected to get value from. This view includes simple narrative descriptions of each practice of interest, including the work products of the practices.”

includedViewSelection: none

Table 10 – Included features for Example ViewSelection 2

Included selection number	Feature name	Basic element
1	name (attribute)	Practice

2	briefDescription (attribute)	Practice
3	description (attribute)	Practice
4	elements (role name)	Practice
5	name (attribute)	WorkProduct
6	briefDescription (attribute)	WorkProduct
7	description (attribute)	WorkProduct
8	levelOfDetail (role name)	WorkProduct
9	name (attribute)	LevelOfDetail
10	briefDescription (attribute)	LevelOfDetail
11	checkListItem (role name)	LevelOfDetail
12	name (attribute)	Checkpoint
13	description (attribute)	Checkpoint

Notes: Selection 4 returns all elements of the practice, but only the ones used in subsequent selections are actually included. Selections 8-13 are all about including work product levels of detail in the view.

10.2.5.2.3 Example ViewSelection 3

name: “Extended user view including alphas”

description: “The purpose of this view is to extend and complement the basic user view above (example 2) by also including alphas and the state of alphas.”

includedViewSelection: “Basic user view” (example 2 above) + “Alpha state view” (example 1 above)

Table 11 – Included features for Example ViewSelection 3

Included selection number	Feature name	Basic element
1	lowerBound (attribute)	WorkProductManifest
2	upperBound (attribute)	WorkProductManifest
3	alpha (role name)	WorkProductManifest
4	workproduct (role name)	WorkProductManifest
5	superAlpha (role name)	AlphaContainment
6	subordinateAlpha (role name)	AlphaContainment
7	lowerBound (attribute)	AlphaContainment
8	upperBound (attribute)	AlphaContainment

10.2.5.2.4 Example ViewSelection 4

name: “Yet another extended user view including activity flows”

description: “The purpose of this view is to extend and complement the extended user view above (example 3) by supporting complete activity flows; this will allow users to view sequences of activities, parallel activities, and understand how activities manipulate alphas and work products. Here the users can also view criteria for alpha state changes, and understand how to progress alpha states in terms of activities.”

includedViewSelection: “Extended user view including alphas” (example 3 above)

Table 12 – Included features for Example ViewSelection 4

Included selection number	**Feature name**	**Construct**
1	name (attribute)	Activity
2	briefDescription (attribute)	Activity
3	approach (attribute)	Activity
4	inputWorkProduct (role name)	Activity
5	outputWorkProduct (role name)	Activity
6	inputAlpha (role name)	Activity
7	outputAlpha (role name)	Activity
8	completionCriterion (role name)	Activity
9	description (attribute)	CompletionCriterion
10	state (role name)	CompletionCriterion

10.3 Composition

10.3.1 Introduction

Extension and composition are core ideas in the Essence language. They are the means by which more sophisticated and powerful constructs are built from smaller, simpler ones.

Extension refers to the modification or customization of an element to suit a new context. For example, a Work Product defined in practice P1 may be modified in the context of a wider practice P2 that uses P1 as a component. The extension mechanism in Essence allows elements to be modified or customized, and has two key features:

- Extension is “aspectual” in the sense that the element being modified is oblivious of the modification.
- Extension is non-destructive, in the sense that the original element still exists and is available.

Composition refers to the capability to put elements together to build more powerful elements form simpler ones. The main use of composition is to put practices together where they are to be used together in an endeavour. In this context, composition allows the way of working on a project to be established by selecting and composing “best in class” practices addressing different aspects of the endeavour. The Essence language allows composition to be used in building the other group constructs: Kernels and Libraries.

Further details are TBD.

10.4 Dynamic Semantics

Since the language defines not only static elements like Alphas and Work Products, but also states associated with them, it can not only be used to express static method descriptions, but also dynamic semantics. Using the states of the single Alphas and their constituent Work Products, the overall state of a software engineering endeavor can be expressed. Based on this, denotational semantics can be defined for a function that supports a team in the enactment of a software engineering endeavor, by using the current state and a specification of the desired state to create a “to-do” list of activities to be performed by the team.

In a large or complex endeavor this function may be provided by a specialist tool. In smaller endeavors, where the overhead of tool support cannot be justified, the function represents a manual recipe that can be followed to determine guidance on how to proceed.

10.4.1 Domain classes

10.4.1.1 Recap of Meta-modeling Levels

As stated in Section 10.1.1, the Essence language is defined as a set of constructs which are language elements defined in the context of a meta-modeling framework. In this framework all the constructs of the language, as described in Section 10.2, are at level 2.

- Level 3 – Meta-Language: the specification language, i.e. the different constructs used for expressing this specification, like “meta-class” and “binary directed relationship.”
- Level 2 – Construct: the language constructs, i.e. the different types of constructs expressed in this specification, like “Alpha” and “Activity.”
- Level 1 – Type: the specification elements, i.e. the elements expressed in specific kernels and practices, like “Requirements” and “Find Actors and Use Cases.”
- Level 0 – Occurrence: the run-time instances, i.e. these are the real-life elements in a running development effort.

A Method Engineer using the Essence language to model the Practices and its associated Activities, Work Products etc., would work at level 1. For instance, to describe an agile Practice like Scrum the Method Engineer would define activities such as “Sprint Planning Meeting” and “Daily Scrum”, and work products such as “Sprint Goal” at level 1. This is exactly analogous to a Software Engineer using the UML language (also described as constructs at level 2) to model an order processing system by define classes such as “Customer, “Order” and “Product” and use cases such as “Place an Order” and “Check Stock Availability” at level 1.

A team using Scrum on a project would be working at level 0. The project team would hold “Sprint Planning Meetings” and “Daily Scrums” and each would be a level 0 instance of the corresponding activity at level 1, and the goal set for each Sprint would be a level 0 instance of the “Sprint Goal” work product defined at level 1. This is exactly analogous to the creation of Customers “Bill Smith” and “Andy Jones” and products “Flange” and “Grommet” at level 0 in the executing order processing system.

10.4.1.2 Naming Convention

In order to define the dynamic semantics it is necessary to refer to the inhabitants of levels 1 and 0 as well as those of level 2. In order to make it clear at which level a named term belongs, we use the following naming convention:

- X (an unadorned name) is a language *Construct* at level 2 as defined in Section 10.2, such as Alpha, Practice, Activity, Work Product.
- my_X (prefixed) is a *Type* at level 1 created by instantiating X. So if X is Activity, my_Activity could be Sprint Planning Meeting.
- my_X_instance is an *Occurrence* at level 0 by instantiating my_X. So if X is Activity, my_Activity_instance could be the XYZ Project Sprint Planning Meeting no. 5 held on the 16th July 2012.

This naming convention is used in the type signatures of functions of the dynamic semantics, so that it is clear to which level of the framework the terms used in the function signature belong. Consider the function **`guidance`** which returns a set of activities to be performed to a take an endeavor forward to the next stage. The type signature of this function is:

```
guidance: (my_Alpha, State)* → (my_Alpha, my_Activity*)*
```

The terms **`my_Alpha and my_Activity`** in this type signature have names prefixed with **`my_`** and so are at level 1. The term **`State`**, on the other hand, has an unadorned name and so is at level 2. Notice here that we allow a function type signature to use elements from different levels of the meta-modeling framework.

10.4.1.3 Abstract Superclasses

To ensure that occurrences at level 0 are endowed with the attributes they need to support the dynamic semantics, we define a set of abstract superclasses at level 1 from which the types defined at level 1 are subclassed. For instance the superclass **my_Alpha** ensures that every Alpha occurrence at level 0 will have attributes "instanceName", "currentState", "workProductInstances" and "subAlphaInstances". These superclasses are named consistently with the naming convention described above.

The relationships between these superclasses and the classes created from the level 2 constructs in shown in Figure .

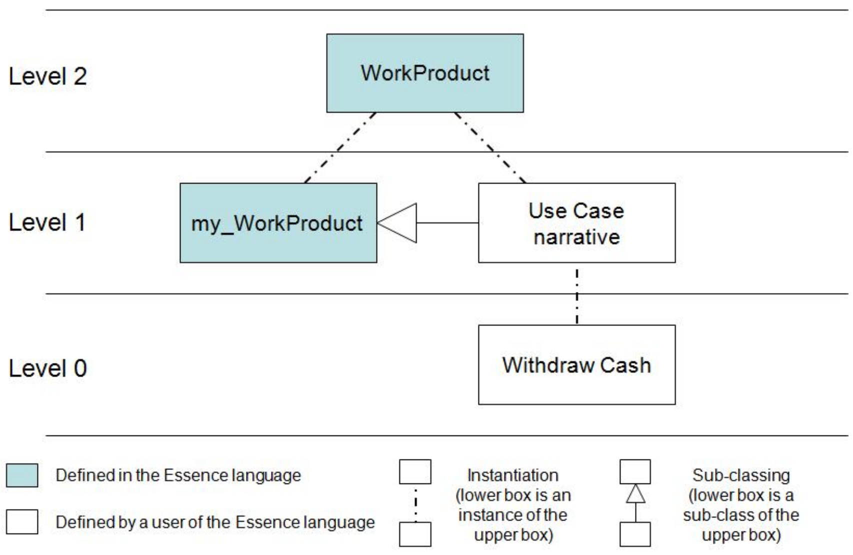

Level 2
WorkProduct
Level 1
my_WorkProduct
Use Case narrative
Level 0
Withdraw Cash
Defined in the Essence language
Defined by a user of the Essence language
Instantiation (lower box is an instance of the upper box)
Sub-classing (lower box is a sub-class of the upper box)

Figure 18 – The Essence language framework

10.4.1.3.1 my_Alpha

The superclass to all level 1 *types* instantiated from the level 2 *construct* "Alpha", i.e. the Alphas in some Kernel (such as "Requirements") or Practice as well as to Sub-Alphas added by a particular Practice (such as "Use Case").

Attributes

instanceName : String [1]	The name of an occurrence (e.g., Requirements for the XYZ Project)
currentState : my_State [1]	A pointer to the current State of an occurrence (e.g., to the state "Coherent")

10.4.1.3.2 my_State

The superclass to all level 1 *types* instantiated from the level 2 *construct* "State", i.e. the States of some Alpha.

Attributes

N/A

10.4.1.3.3 my_WorkProduct

The superclass to all level 1 *types* instantiated from the level 2 *construct* “Work Product”, i.e. to all templates representing physical documents used in the software engineering endeavor, such as “Use Case narrative”.

Attributes

instanceName : String [1]	The name of an occurrence (e.g., Use Case Narrative for Withdraw Cash)
current levelOfDetail : my_LevelOfDetail [1]	A pointer to the current LevelOfDetail of an occurrence (e.g., to the level “Sketch”)

10.4.1.3.4 my_LevelOfDetail

The superclass to all level 1 *types* instantiated from the level 2 *construct* “LevelOfDetail”, i.e. the level of detail of some work product.

Attributes

N/A

10.4.1.3.5 my_Activity

The superclass to all level 1 *types* instantiated from the level 2 *construct* “Activity”, i.e. to all templates describing work items.

Attributes

instanceName : String [1]	The name of an occurrence (e.g., Define and agree Use Case “Withdraw Cash”)

10.4.2 Operational Semantics

In this section we describe and illustrate the operational semantics. This covers how the level 0 model is created, how the state of the endeavor is tracked in the model and how the model can be used to give advice based on how to progress the state of the endeavor. For the last of these we provide a formal denotational semantics.

10.4.2.1 Populating the Level 0 Model

Generally, the appropriate Alpha instances and associated Work Product instances are created as soon as the respective Alpha is considered in the endeavor. Some may exist right from the start of the endeavor (such as the Alpha instances for Stakeholders or Requirements), while others may be created later, at the appropriate point in the conduct of a practice. This is usually the case for Sub-Alpha instances, which are instantiated as needed through the endeavor. The model of a practice is used as the basis for instantiating the appropriate sets of Alpha instances and associated Work Product instances, using the Work Product Manifests defined for the Practice as templates. Although the mechanisms of instantiation and updating Alpha instances and their associated Work Product instances can be formalized using computational semantics, it is not an automatic process and must be triggered explicitly by the team.

A team is also free to create instances in their model that do not derive by instantiating from Practice templates, and thus tailor the use of a Practice or even depart from it to create a partially or completely customized approach.

10.4.2.2 Determining the Overall State

Determining the overall state of the endeavor is done by determining the states of each individual Alpha instance in the endeavor. This is done using the checkpoints associated with each state of the respective state graphs; and the state is determined to be the most advanced in the state graph consistent with the currently met checkpoints. This means the state that has:

1. all currently fulfilled checkpoints met; and
2. no outgoing transition to a state that has also all currently fulfilled checkpoints met.

This is illustrated in Figure 19. Here the most advanced state of Software System "XYZ" consistent with the checkpoints that have been met (shown as ticked) is "Useable".

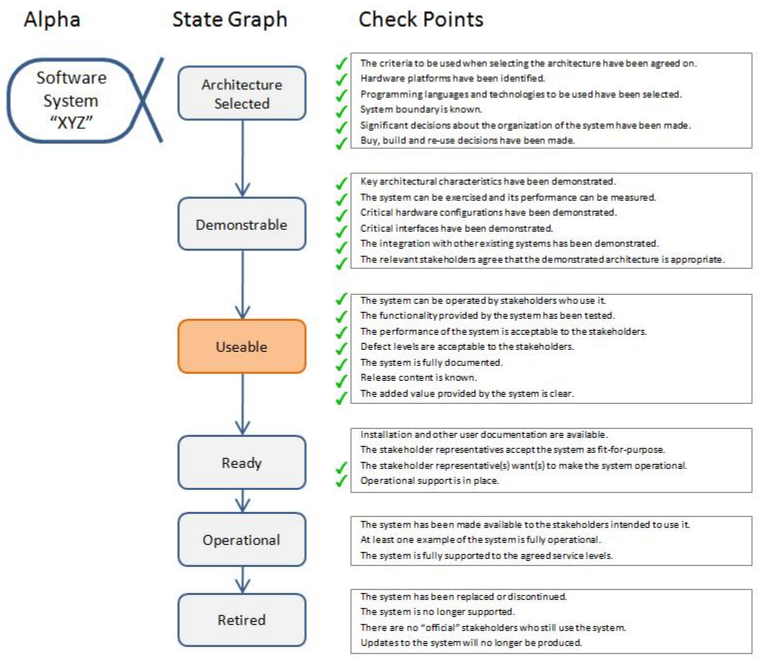

Alpha
State Graph
Check Points
Software System "XYZ"
Architecture Selected
Demonstrable
Useable
Ready
Operational
Retired
✓ The criteria to be used when selecting the architecture have been agreed on.
✓ Hardware platforms have been identified.
✓ Programming languages and technologies to be used have been selected.
✓ System boundary is known.
✓ Significant decisions about the organization of the system have been made.
✓ Buy, build and re-use decisions have been made.
✓ Key architectural characteristics have been demonstrated.
✓ The system can be exercised and its performance can be measured.
✓ Critical hardware configurations have been demonstrated.
✓ Critical interfaces have been demonstrated.
✓ The integration with other existing systems has been demonstrated.
✓ The relevant stakeholders agree that the demonstrated architecture is appropriate.
✓ The system can be operated by stakeholders who use it.
✓ The functionality provided by the system has been tested.
✓ The performance of the system is acceptable to the stakeholders.
✓ Defect levels are acceptable to the stakeholders.
✓ The system is fully documented.
✓ Release content is known.
✓ The added value provided by the system is clear.
Installation and other user documentation are available.
The stakeholder representatives accept the system as fit-for-purpose.
✓ The stakeholder representative(s) want(s) to make the system operational.
✓ Operational support is in place.
The system has been made available to the stakeholders intended to use it.
At least one example of the system is fully operational.
The system is fully supported to the agreed service levels.
The system has been replaced or discontinued.
The system is no longer supported.
There are no "official" stakeholders who still use the system.
Updates to the system will no longer be produced.

Figure 19 – Determination of State using Check Points

The determination of Alpha instance states can happen at any point in time since evaluating the checkpoints is a manual activity. When checkpoints are evaluated the result can be that an Alpha instance regresses, its current state being set back to some earlier state of its lifecycle. This happens if re-evaluation determines that a checkpoint previously thought to have been met is now deemed not to have been met.

10.4.2.3 Generating Guidance

In an actual running software engineering endeavor, a team will want to get guidance on what to do next.

Once the overall state of the endeavor is determined, the model can be used to generate such advice. This can be understood as a guidance function that takes a set of pairs of (Alpha instance and target State) as its argument and returns a set of newly instantiated Activities: a "to-do" list to be performed by the team. This function is invoked with an actual argument consisting of a set of pairs, each pair consisting of a my_Alpha_instance (at level 0) and a my_State (at level 1). For each pair the function returns guidance on how to progress each my_Alpha_instance to its target state my_State. This guidance is of the form of a set of newly instantiated activities (at level 0) for each my_Alpha_instance, constituting a to-do list to be performed by the team to advance its state. The essential idea is to assemble the to-do list by examining each Alpha instance given to the function and finding those activities that have the target state of that Alpha instance among its completion criteria.

Note that an Essence model does not specify how the team works on a set of activities. This is the dictated by the policies, rules or advice of the practices being used on the endeavor. These may require or suggest that certain activities should be prioritized, done in a particular sequence, divided among sub-teams, and so on. The team uses its expertise in the practices to work out exactly how to perform the activities required. Nor is there any ultimate guarantee that the team will follow the advice or perform the suggested activities competently: in that sense the model is an "open loop" control system. However, regular re-evaluation of the checkpoints and the consequent re-setting of the Alpha instance states will provide feedback to the team on whether or not their work is advancing as hoped.

Several other functions can be defined to measure the progress and health of the endeavor, for instance to determine whether the right set of my_Alpha_Instances and my_WorkProduct_Instances is in place, or to determine whether the endeavor has reached its final state. These have not been defined here.

10.4.2.4 Formal definition of the Guidance Function

In this section, we provide a formal description of the operational semantics in terms of the function **guidance**. This function takes a set of pairs of (Alpha instance and target State) as its argument and returns a set of to-do lists, one for each Alpha instance and target State provided to the function.

The essential idea is to compile the to-do lists by examining each Alpha instance given to the function and finding those activities that have the target state of that Alpha instance among its completion criteria. However, the target state specified for an Alpha instance may not be the next state in the state graph of the Alpha, and so a function **statesAfter** is used to find the intermediate states. The to-do list generated consists of the activities required to progress the Alpha instance through all these states in order to reach the specified target.

First we specify the **statesAfter** function. Suppose that a state graph has a sequence of states S_0, S_1, S_2, S_3. If **statesAfter** is called with (S_0, S_3) it will return {S_1, S_2, S_3}. In other words, all the states passed through to get to S_3 but not including the starting state S_0. This is easier to specify in terms of a function **fullPath** that generates the full set of states including the starting state. So if **fullPath** is called with (S_0, S_3) it will return {S_0, S_1, S_2, S_3}.

```
statesAfter: (State, State) → State*
statesAfter (s₁, s₂) =
     fullPath(s₁, s₂) - {s₁}

fullPath: (State, State) → State*
fullPath (s₁, s₂) =
      if ((s₁.successor = null) ∨ (s₁ = s₂))    {s₁}
      else {s₁} ∪ fullPath(s₁.successor, s₂)
```

We use this to specify the **guidance** function. Each (Alpha instance, target State) pair is taken in turn.

```
guidance: (my_Alpha, State)* → (my_alpha, my_Activity*)*
guidance (cas) =
      let as ∈ cas
      in to_do(as) ∪ guidance (cas – {as})
```

The **to_do** function takes a single (Alpha instance, target State) pair and creates the set of activities that are required to progress the Alpha instance to the required target State. This is done by finding those activity types that have the target state or any intermediate state among its completion criteria. The function **statesAfter** is used to find the intermediate states.

Note that the completion criteria (defined at level 1) are defined using activity types (at level 1). The function **to_do** determines the set of activity types required for each Alpha instance.

As the to-do list is to be constructed as a set of new instantiated activities (at level 0) we use **mk_w(α)** to instantiate (i.e., create an instance of) **w** at level 0. This is done by the function **create_instances**. Each newly instantiated level 0 activity stores the passed Alpha instance **(α)** as an element of its stored set of related Alpha instances, myAlphaInstances (see Section 10.4.1.3).

```
to_do: (my_Alpha, State) → (my_alpha, my_Activity*)
to_do (α, σ) =
let cw = { w | (α.type ∈ w.outputAlpha) ∧
                (σ’ ∩ completionStates(w.completionCriterion) ≠ ∅) ∧
                (σ’ ∈ statesAfter(α.currentState,σ)) }
      in (α, create_instances(α, cw))

create_instances: (my_Alpha, Activity*) → my_Activity*
create_instances(α, cw) =
      let w ∈ cw
      in mk_w(α) ∪ create_instances(α, cw - {w})
```

Finally, we specify the function completionStates which is used by the to_do function to determine the set of states forming the completion criteria of an activity.

```
completionStates:  CompletionCriterion* → State*
completionStates (ccc) =
      let cc ∈ ccc and rs = cc.reachedState
      in rs ∪ completionStates(ccc - {cc})
```

10.4.2.5 Further functions

As well as the Guidance Function, a number of other functions can be defined to support enactment. This section describes a number of these as illustration. It is expected that any Essence tool will support at least these functions.

The to_do function used to generate guidance makes use of the property “currentState” on my_Alpha. It is not specified whether tool vendors allow users to set this property directly or consider it a derived property. However, if it is handled as a derived property, it has to be derived in the following way:

```
derive_current_state: my_Alpha → my_State
derive_current_state (a) =
      let s = { s | s ∈ a.states ∧ {ps | ps.successor=s} = ∅}
      in fullfilledSuccessorState(s)

fullfilledSuccessorState: my_State → my_State
fullfilledSuccessorState (s) =
      if (s.successor = ∅) {s}
      else
            let mc = {c | c ∈ s.successor.checkpoints ∧ not c.isFullfilled}
            in (if (mc = ∅) {fullfilledSuccessor(s.successor)} else {s})
```

The same can be done for “currentLevelOfDetail” on my_WorkProduct:

```
derive_current_level_of_detail: my_WorkProduct → my_LevelOfDetail
derive_current_level_of_detail (wp) =
      let s = { l | l ∈ wp.levelOfDetail ∧ {pl | pl.successor=l} = ∅}
      in fullfilledSuccessorLevel(l)


fullfilledSuccessorLevel: my_LevelOfDetail → my_LevelOfDetail
fullfilledSuccessorLevel (l) =
      if (l.successor = ∅) {l}
      else
            let mc = {c | c ∈ s.successor.checkpoints ∧ not c.isFullfilled}
            in (if (mc = ∅) {fullfilledSuccessor(s.successor)} else {l})
```

Before using the guidance function on a set of (Alpha instance, target State) pairs, a user may want to derive a set of sensible target states from the current states.

```
nextAlphaStatesToReach: my_Alpha* → my_State*
nextAlphaStateToReach(a) =
      let oa ∈ a
      in oa.currentState.successor ∪ nextAlphaStateToReach(a - {oa})
```

10.5 Graphical Syntax

10.5.1 Specification Format

The graphical syntax provides a visual form for each construct. Each graphical notation is introduced in a separate section that provides a description and symbol of the syntax. This section includes subsections for **Style Guidelines** and **Examples** when applicable.

Diagrams are introduced by listing the graphical nodes and links to be included in the diagrams. Each node and link refers to the syntax specification of an individual element.

10.5.2 Relevant Symbols

Most of the constructs in the abstract syntax of the Kernel Language require a visual representation in terms of a symbol for the purpose of being visualized. However:

- Some constructs are visualized in terms of complete diagrams and may not require a symbol, e.g. State Graph, where the states of the state graph can be visualized in a diagram but where State Graph in itself does not require any specific symbol.
- Constructs like Completion Criterion and Required Competency may not require symbols of their own but are instead visualized textually only.

10.5.3 Default Notation for Meta-Class Constructs

The default notation for a meta-class construct in the abstract syntax is a solid-outline rectangle containing the name of the construct's type (level 1 in the abstract syntax). The name of the construct itself (level 2 in the abstract syntax) can be shown in guillemets above the type name. Alternatively, if the meta-class construct defines its own distinct symbol, this symbol can be shown above the type name in the rectangle.

This provides a default and unique visualization of each meta-class construct in the abstract syntax.

Style Guidelines

- Center the name of the construct's type in boldface.
- Center the name of the construct itself in plain face within guillemets above the type name, or alternatively:
- Include the symbol of the construct above the type name and aligned to the right.

Examples

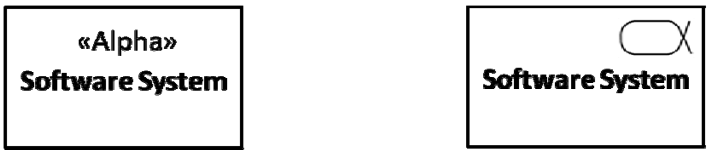


Figure 20 – Example visualizations of the Alpha meta-class construct and its Software System type

10.5.4 View 1: Alphas and their States

The following sections define relevant symbols for View 1: Alphas and the States.

10.5.4.1 Alpha

An Alpha is visualized by the following symbol, either containing the name of the Alpha or with the name of the Alpha placed below the symbol:

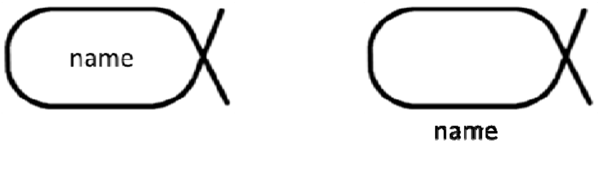


Figure 21 – Alpha symbol

Style Guidelines

- Center the name of the Alpha in boldface, either within the symbol or below the symbol.

Examples

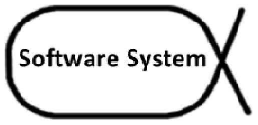


Figure 22 – Software System Alpha

10.5.4.2 Alpha Association

An Alpha Association is visualized by a solid line connecting two associated Alphas. The line may consist of one or more connected segments. The association line is adorned with the name of the association.

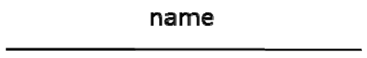


Figure 23 – Alpha Association symbol

Style Guidelines

- Center the name of the Alpha Association above or under the association line in plain face.
- An open arrowhead '>' or '<' next to the name of the association and pointing along the association line indicates the order of reading and understanding the association. This arrowhead is for documentation purposes only and has no general semantic meaning.

Examples

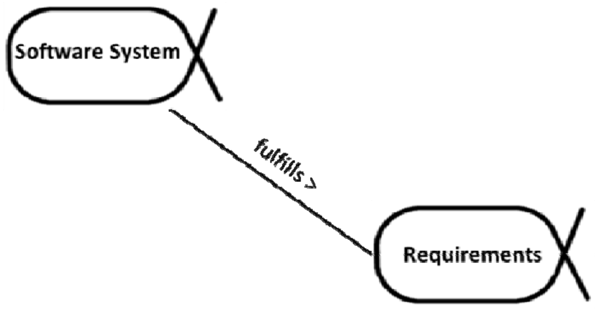


Figure 24 – Alpha Association between the Requirements Alpha and the Software System Alpha, read as: "The Software System fulfills the Requirements."

10.5.4.3 Kernel

A Kernel is visualized by a hexagon containing a cogwheel; either containing the name of the Kernel or with the name of the Kernel placed below the symbol.

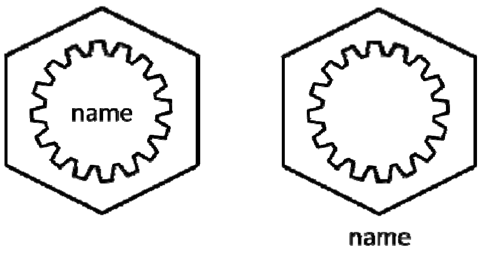


Figure 25 – Kernel symbol

Style Guidelines

- Center the name of the Kernel in boldface, either within the symbol or below the symbol.

Examples

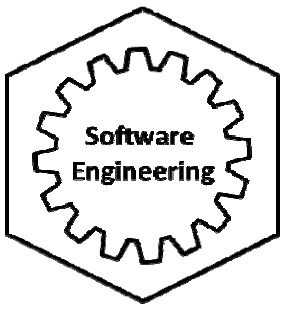


Figure 26 – Kernel for Software Engineering

10.5.4.4 State

A State is visualized by a rectangle with rounded corners containing the name of the State.

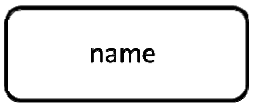


Figure 27 – State symbol

Style Guidelines

- Center the name of the State in boldface.

Examples

Milestones Agreed

Figure 28 – Milestones Agreed State

10.5.4.5 Transition

A Transition is visualized by a solid line with an open arrowhead connecting two States. The line may consist of one or more connected segments.

Figure 29 – Transition

Examples

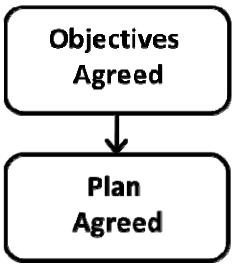


Figure 30 – Transition from the Objectives Agreed State to the Plan Agreed State

10.5.4.6 Diagrams

This section defines the graphical elements that may be shown in diagrams, and provides cross references where detailed information about the concrete notation for each element can be found.

10.5.4.6.1 Alpha Structure Diagram

Table 13 – Graphical nodes in Alpha Structure diagrams.

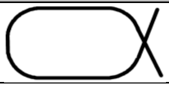
Node Type	Symbol	Reference
Alpha		Section 10.5.4.1 Alpha.

Table 14 – Graphical links in Alpha Structure diagrams.

Link Type	Symbol	Reference
Alpha Association	———	Section 10.5.4.2 Alpha Association.

Examples

Refer to kernel examples.

10.5.4.6.2 State Graph Diagram

Table 15 – Graphical nodes in State Graph diagrams.

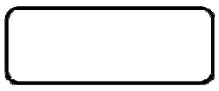
Node Type	Symbol	Reference
State		Section 10.5.4.4 State.

Table 16 – Graphical links in State Graph diagrams.

Link Type	Symbol	Reference
Transition	⟶	Section 10.5.4.5 Transition.

Style Guidelines

- Place the start state at the top of the diagram, and the stop state at the bottom of the diagram.
- Use transitions to visualize a logical sequence through states, from start to stop. Only visualize alternative transitions when there are mutually exclusive state sets involved in the sequence from start to stop. Within a specific sequence from start to stop, we may assume that any loop or alternation is permitted without visualizing corresponding transitions.

Examples

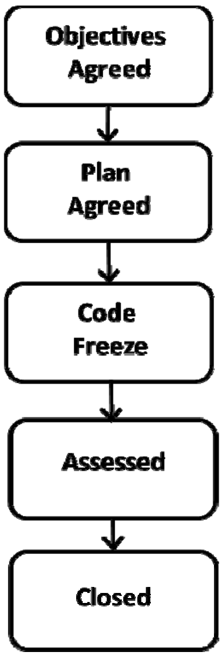


Figure 31 – State Graph example

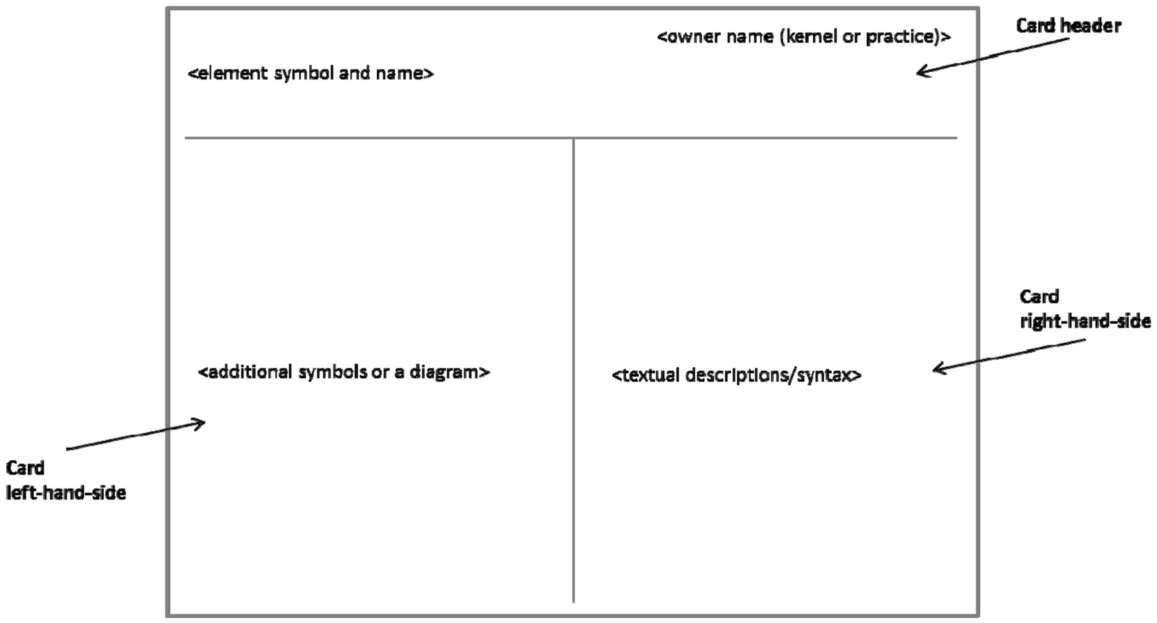
10.5.4.7 Cards

As a complement to the symbols and diagrams we use a card metaphor (as in 5x3 inch index cards) to visualize the most important aspects of an element in the Kernel Language. A card presents a succinct summary of the most important things you need to remember about an element. In many cases, all that a practitioner needs to be able to apply a kernel or a practice is a corresponding set of cards.

In particular, cards are straightforward to manifest as physical entities (print them on paper) which makes them very hands-on and natural for practitioners to put on the table, play around with, and reason about; all for the purpose to guide practitioners in their way of working.

10.5.4.7.1 The Anatomy of a Card

A card is visualized as a solid-outline rectangle containing a mix of symbols and textual syntax related to the element. The following is a basic anatomy although variations are allowed:



<owner name (kernel or practice)>
Card header
<element symbol and name>
Card
right-hand-side
<additional symbols or a diagram>
<textual descriptions/syntax>
Card
left-hand-side

Figure 32 – A basic card anatomy to visualize an element

Style Guidelines

- Place the owner name in boldface at the top-right of the card and use a font with smaller size than for the element name top-left.

10.5.4.7.2 Alpha Definition Card

An Alpha definition card is defined as follows:

- **Card left-hand-side:** State Graph Diagram for the Alpha.
- **Card right-hand-side**: Brief Description of the Alpha, as well as a listing of contents including Essential Qualities, and contained elements (sub-Alphas or Work Products, if any).

Examples

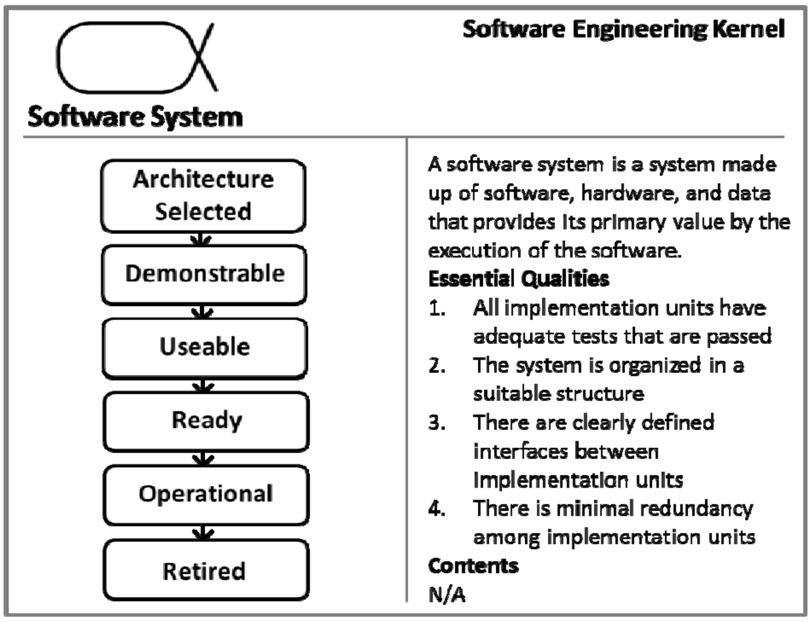

Software Engineering Kernel
Software System
Architecture Selected
Demonstrable
Useable
Ready
Operational
Retired
A software system is a system made up of software, hardware, and data that provides its primary value by the execution of the software.
Essential Qualities
1. All implementation units have adequate tests that are passed
2. The system is organized in a suitable structure
3. There are clearly defined interfaces between implementation units
4. There is minimal redundancy among implementation units
Contents
N/A

Figure 33 – Software System Alpha Definition Card

10.5.5 View 2: Sub-Alphas and Work Products

The following sections define relevant symbols for View 2: Sub-Alphas and Work Products.

10.5.5.1 Work Product

A Work Product is visualized by the following symbol, either containing the name of the Work Product or with the name of the Work Product placed below the symbol:

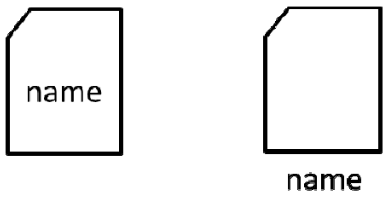


Figure 34 – Work Product symbol

Style Guidelines

- Center the name of the Work Product in boldface, either within the symbol or below the symbol.

Examples

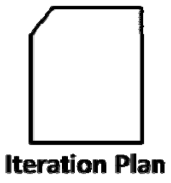


Figure 35 – Iteration Plan Work Product

10.5.5.2 Alpha Containment

An Alpha Containment is visualized by a solid line connecting a super- and a sub-Alpha. The line may consist of one or more connected segments. The line is adorned with a filled diamond placed at the end of the line connecting the super-Alpha; and with the multiplicity of the sub-Alpha placed near the end of the line connecting the sub-Alpha.

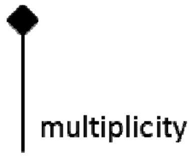


Figure 36 – Alpha Containment symbol

As an alternative, an Alpha Containment can be visualized by encompassing the sub-Alpha symbols within the super-Alpha symbol.

Style Guidelines

- Arrange the line vertically with the super-Alpha on top and the sub-Alpha at the bottom, thereby visualizing a top-down hierarchy.
- If there are two or more sub-Alphas of the same super-Alpha, they may be visualized as a tree by being placed at the same horizontal level and by merging the lines to the super-Alpha into a single segment.
- Visualizing a sub-Alpha multiplicity is optional.

Examples

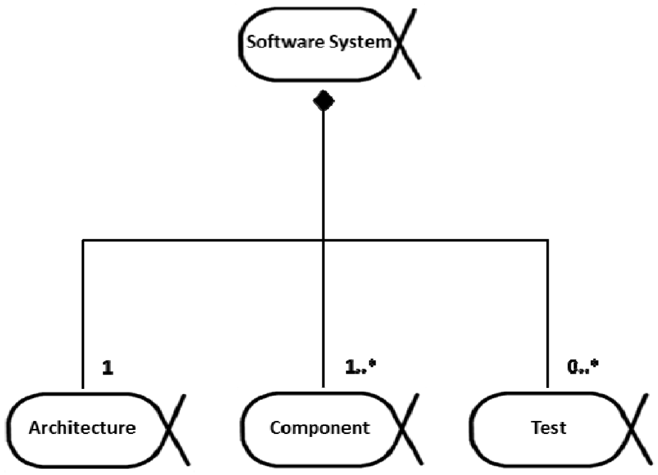


Figure 37 – Software System super-Alpha and three sub-Alphas: Architecture, Component, and Test with visualized multiplicities

10.5.5.3 Alpha Manifest

An Alpha Manifest is visualized by a solid line connecting an Alpha and a Work Product. The line may consist of one or more connected segments. The line is adorned with a filled diamond placed at the end of the line connecting the Alpha; and with the multiplicity of the Work Product placed near the end of the line connecting the Work Product.

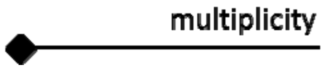


Figure 38 – Alpha Manifest symbol

Note that this is the same symbol as the Alpha Containment symbol, however the symbols are discriminated based on their context; that is, whether two Alphas are connected (Alpha Containment), or whether an Alpha and a Work Product are connected (Alpha Manifest).

As an alternative, an Alpha Manifest can be visualized by encompassing the Work Product symbols within the Alpha symbol.

Style Guidelines

- Arrange the line horizontally with the Alpha to the left and the Work Product to the right, thereby visualizing a left-to-right hierarchy.
- If there are two or more Work Products of the same Alpha, they may be visualized as a tree by being placed at the same horizontal level and by merging the lines to the Alpha into a single segment.
- Visualizing a Work Product multiplicity is optional.

Examples

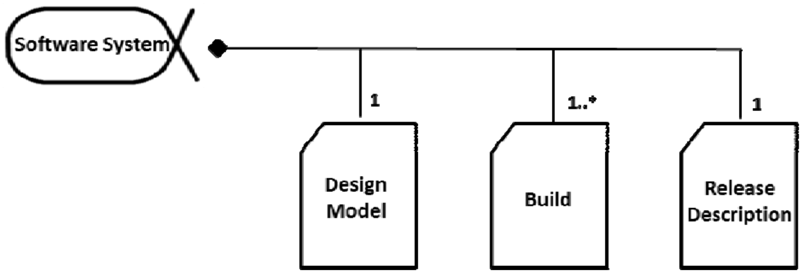


Figure 39 – Software System Alpha and three Work Products: Design Model, Build, and Release Description with visualized multiplicities

10.5.5.4 Practice

A Practice is visualized by a hexagon; either containing the name of the Practice or with the name of the Practice placed below the symbol.

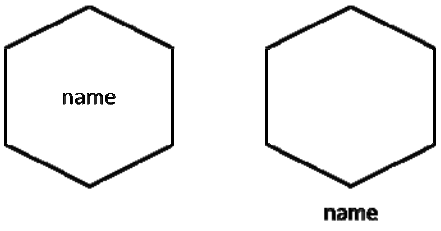


Figure 40 – Practice symbol

Style Guidelines

- Center the name of the Practice in boldface, either within the symbol or below the symbol.

Examples

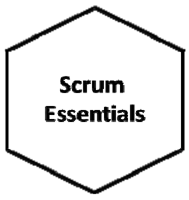


Figure 41 – Scrum Essentials Practice

10.5.5.5 Diagrams

This section defines the graphical elements that may be shown in diagrams, and provides cross references where detailed information about the concrete notation for each element can be found.

10.5.5.5.1 Alpha Hierarchy Diagram

Table 17 – Graphical nodes in Alpha Hierarchy diagrams.

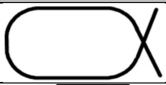
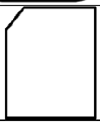
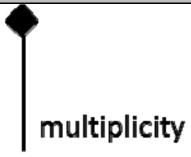
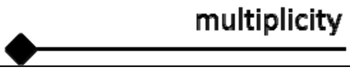
Node Type	Symbol	Reference
Alpha		Section 10.5.4.1 Alpha.
Work Product		Section 10.5.5.1 Work Product.

Table 18 – Graphical links in Alpha Hierarchy diagrams.

Link Type	Symbol	Reference
Alpha Containment		See 10.5.5.2 Alpha Containment.
Work Product Manifest		See 10.5.5.3 Alpha Manifest.

Examples

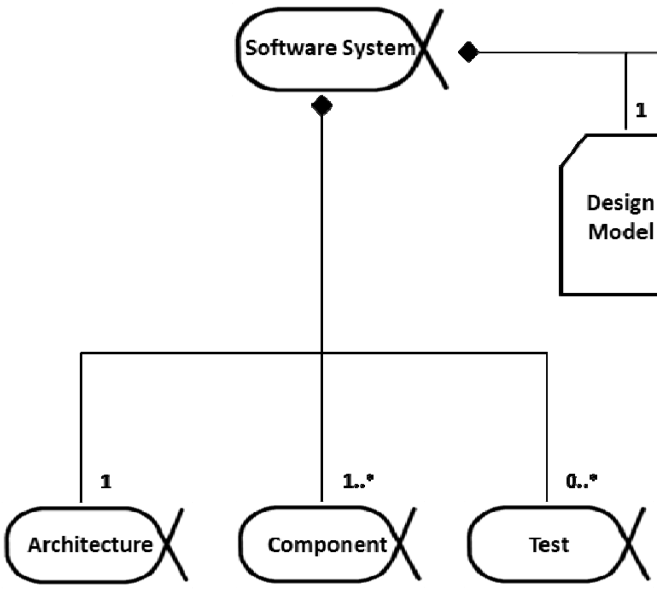


Figure 42 – Alpha Containment and Work Product Manifest relationships of the Software System Alpha

10.5.6 View 3: Activity Spaces and Activities

The following sections define relevant symbols for View 3: Activity Spaces and Activities.

10.5.6.1 Activity

An Activity is visualized by the following symbol, either containing the name of the Activity or with the name of the Activity placed below the symbol:

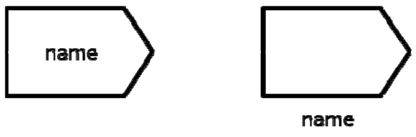


Figure 43 – Activity symbol

Style Guidelines

- Center the name of the Activity in boldface, either within the symbol or below the symbol.

Examples

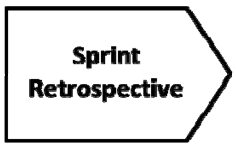


Figure 44 – Sprint Retrospective Activity

10.5.6.2 Activity Space

An Activity Space is visualized by the following dashed-outline symbol, either containing the name of the Activity Space or with the name of the Activity Space placed below the symbol:

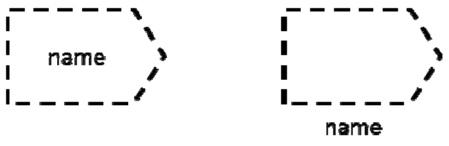


Figure 45 – Activity Space symbol

Style Guidelines

- Center the name of the Activity Space in boldface, either within the symbol or below the symbol.

Examples

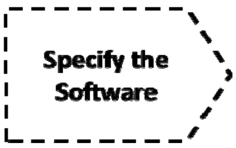


Figure 46 – Specify the Software Activity Space

10.5.6.3 Activity Manifest

An Activity Manifest is visualized by a solid line connecting an Activity Space and an Activity. The line may consist of one or more connected segments. The line is adorned with a filled diamond placed at the end of the line connecting the Activity Space.

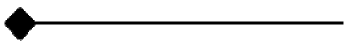


Figure 47 – Activity Manifest symbol

Note that this is the same symbol as the Alpha Containment and Alpha Manifest symbol, however the symbols are discriminated based on their context; that is, whether two Alphas are connected (Alpha Containment), or whether an Alpha and a Work Product are connected (Alpha Manifest), or whether an Activity Space and an Activity are connected (Activity Manifest).

As an alternative, an Activity Manifest can be visualized by encompassing the Activity symbols within the Activity Space symbol.

Style Guidelines

- Arrange the line horizontally with the Activity Space to the left and the Activity to the right, thereby visualizing a left-to-right hierarchy.
- If there are two or more Activities of the same Activity Space, they may be visualized as a tree by being placed at the same horizontal level and by merging the lines to the Alpha into a single segment.

Examples

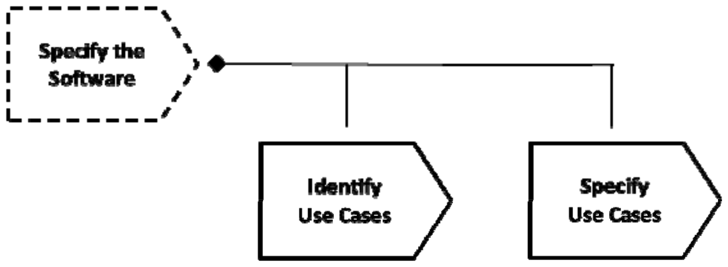


Figure 48 – Specify the Software Activity Space and two Activities: Identify Use Cases and Specify Use Cases

10.5.6.4 Activity Association

An Activity Association is visualized by a solid line connecting two Activity symbols. The line may consist of one or more connected segments. The line is adorned with a filled triangular arrowhead placed at the end of the line connecting end2.

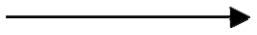


Figure 49 – Activity Association symbol.

Style Guidelines

- Lines may be drawn using curved segments.

Examples

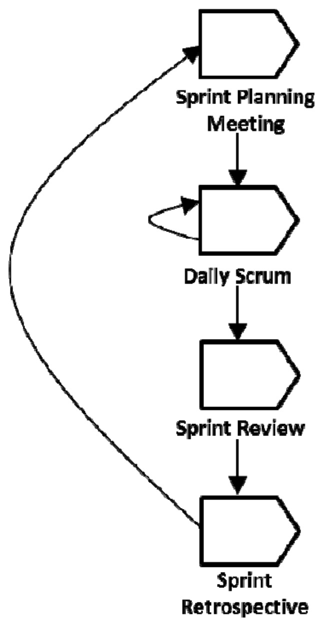


Figure 50 – Activity Association among four activities in a Scrum Essentials practice

10.5.6.5 Competency

A Competency is visualized by a 5-point star symbol with the name of the Competency placed below the symbol:

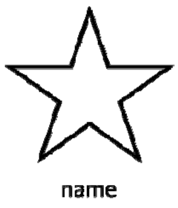


Figure 51 – Competency symbol

Style Guidelines

- Center the name of the Competency in boldface below the symbol.

Examples

Leadership

Figure 52 – Leadership Competency

10.5.6.6 Diagrams

This section defines the graphical elements that may be shown in diagrams, and provides cross references where detailed information about the concrete notation for each element can be found.

10.5.6.6.1 Activity Space Hierarchy Diagram

Table 19 – Graphical nodes in Activity Space Hierarchy diagrams.

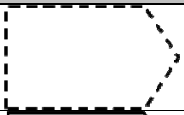
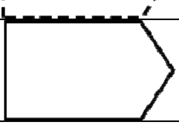
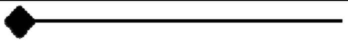
Node Type	Symbol	Reference
Activity Space		Section 10.5.6.2 Activity Space.
Activity		Section 10.5.6.1 Activity.

Table 20 – Graphical links in Activity Space Hierarchy diagrams.

Link Type	Symbol	Reference
Activity Manifest		See 10.5.6.3 Activity Manifest.

Examples

Refer to 10.5.6.3 Activity Manifest example.

10.5.6.6.2 Activity Flow Diagram

Table 21 – Graphical nodes in Activity Flow diagrams.

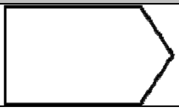
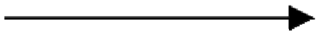
Node Type	Symbol	Reference
Activity		Section 10.5.6.1 Activity.

Table 22 - Graphical links in Activity Flow Hierarchy diagrams.

Link Type	Symbol	Reference
Activity Association		See 10.5.6.4 Activity Association.

Style Guidelines

- Arrange the Activity Association arrow pointing from left-to-right or from top-to-bottom, except for loop-backs.

Examples

Refer to 10.5.6.4 Activity Association

10.5.6.7 Cards

10.5.6.7.1 Activity Definition Card

An Activity definition card is defined as follows:

- **Card left-hand-side:** Symbols for activity inputs, required competencies, and outputs.

- **Card right-hand-side**: Brief description of the activity, as well as a listing of completion criteria and approaches.

Figure 77 shows an example.

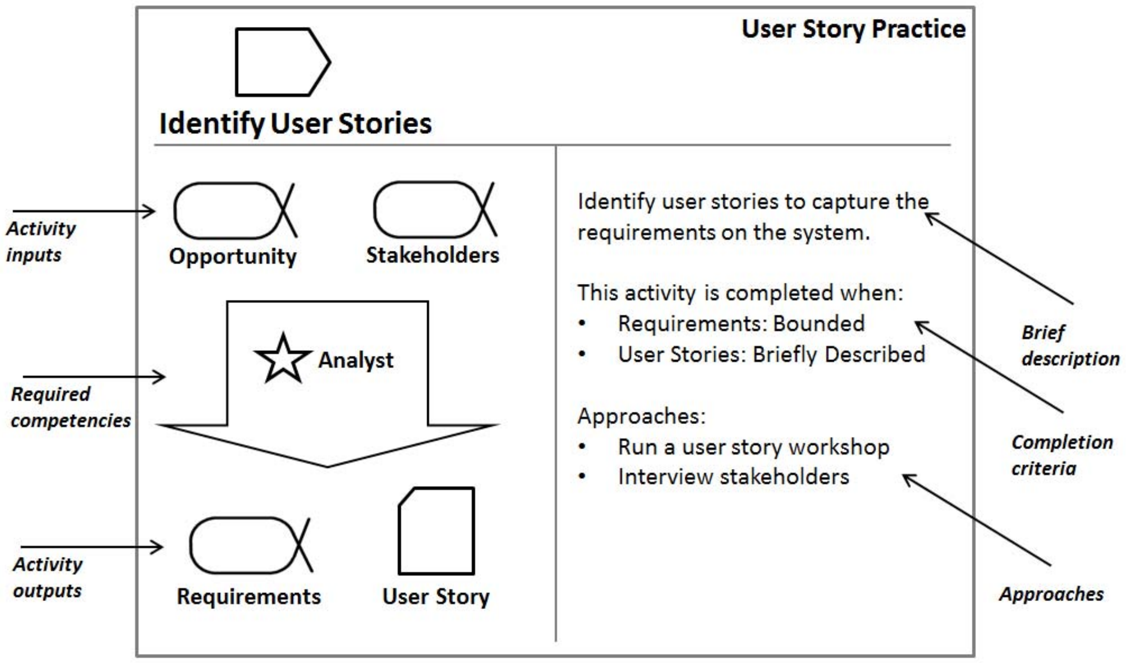


Figure 53 – "Identify User Stories" activity definition card

.

10.6 Textual Syntax

This section provides a textual syntax for the SEMAT Kernel Language and describes its mapping to the abstract syntax presented above. The rules of the textual syntax are given in BNF-style.

The textual syntax does not specify any rules for file handling. Specifically it assumes that everything to be expressed using this syntax is written in one single file. However, parser implementations may include facilities for merging files prior to parsing in order to handle contents which are split over multiple files.

References between elements specified in the textual syntax can be made via identifiers. Each element that can be referred to must define a unique identifier. Every element that wants to refer to another element can use this identifier as a reference. Identifiers are unique within the containment hierarchy. Using an identifier outside the containment hierarchy requires to prefix it with the identifiers of its parent element(s).

10.6.1 Rules

The following notation is used in this subsection:

- (…)* means 0 or more occurrences
- (…)? means 0 or 1 occurrence
- (…)+ means 1 or more occurrences
- | denotes alternatives
- ID is a special token representing a string which can be used as an identifier for the defined element
- …Ref denotes a token representing an identifier of some element (i.e. not the defined element)

10.6.1.1 Root Elements

The root element representing the file containing the specification is defined as:

```
Model:
        StandaloneElement*;
```

An empty file is a valid root. If not empty, the file may contain an arbitrary number of elements.

There are four categories of elements:

```
StandaloneElement:
      ExtensionElement | Alpha | AlphaAssociation | AlphaContainment |
WorkProduct | WorkProductManifest | Activity | ActivitySpace | ActivityManifest |
ActivityAssociation | Competency | Pattern | Kernel | Practice | Library;


AnyElement:
      StandaloneElement | State | Level | CheckListItem | CompetencyLevel |
PatternAssociation | Tag | Resource;


PatternElement:
      Alpha | AlphaAssociation | AlphaContainment | WorkProduct |
WorkProductManifest | Activity | ActivitySpace | ActivityManifest |
ActivityAssociation | Competency | Pattern;


KernelElement:
      ExtensionElement | Alpha | AlphaAssociation | AlphaContainment |
ActivitySpace | Competency | Kernel;
```

10.6.1.2 Element Groups

A Kernel declaration is defined as:

```
Kernel:
      'kernel'   ID   ':'   STRING   'contains'   '{'   KernelElementRef   (','
KernelElementRef)* '}' AddedTags?;
```

This maps directly to the language element with the same name. The ID creates a unique identifier for this Kernel, which maps to the attribute "name". The STRING is considered as content for attribute "description". If no STRING is given, the empty string must be used for attribute "description". KernelElementRef is a unique identifier to an element to be contained in this kernel.

A Practice declaration is defined similarly as:

```
Practice:
      'practice'   ID   ':'   STRING   'contains'   '{'   StandaloneElementRef   (','
StandaloneElementRef)* '}' AddedTags?;
```

A Library declaration is defined similarly as:

```
Library:
      'library' ID ':' STRING 'contains' '{' StandaloneElementRef (','
StandaloneElementRef)* '}' AddedTags?;
```

10.6.1.3 Kernel Elements

An Alpha declaration and its contents are defined as:

```
Alpha:
      'alpha' ID ':' STRING (Resource(',' Resource)*)? 'with states' '{' State+
'}' AddedTags?;


State:
      'state' ID '{' STRING ('checks {' CheckListItem+ '}')? '}' AddedTags?;

CheckListItem:
      'item' ID '{' STRING '}' AddedTags?;
```

In all cases, the ID creates a unique identifier for the element, which maps to the attribute "name". The STRING is considered as content for attribute "description". If no STRING is given, the empty string must be used for attribute "description". References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

KernelAssociation declarations resolve to two alternatives as:

```
AlphaAssociation:
      Cardinality AlphaRef '--' STRING '-->' Cardinality AlphaRef AddedTags?;

AlphaContainment:
      AlphaRef 'contains' Cardinality AlphaRef AddedTags?;
```

The STRING is considered as content for attribute "name" of this AlphaAssociation. The Cardinality maps to the attributes for lower and upper bounds in all cases. References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

An ActivitySpace declaration and its contents are defined as:

```
ActivitySpace:
      'activitySpace'                                                         ID
            '{'                                                                (
                  (STRING)?
                  (Resource(','                                     Resource)*)?
                  'targets'         StateRef            (','             StateRef*
                  (InputAlpha)?
                  (OutputAlpha)?
```

```
                (CompetencyRequirement)?
        ) '}' AddedTags?;


InputAlpha:
        'inputAlphas {' AlphaRef (',' AlphaRef)* '}'

OutputAlpha:
        'outputAlphas {' AlphaRef (',' AlphaRef)* '}'

CompetencyRequirement:
        'requires competency level' CompetencyLevelRef (',' CompetencyLevelRef)*
AddedTags?;
```

The ID creates a unique identifier for this ActivitySpace, which maps to the attribute “name”. The STRING is considered as content for attribute “description”. If no STRING is given, the empty string must be used for attribute “description”. References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

A Competency declaration is defined as:

```
Competency:
        'competency' ID ':' STRING (Resource (',' Resource)*)? ('has' '{'
CompetencyLevel* '}')? AddedTags?;

CompetencyLevel:
        'level' INT ID STRING? AddedTags?;
```

In both cases, the ID creates a unique identifier for the element, which maps to the attribute “name”. The STRING is considered as content for attribute “description”. If no STRING is given, the empty string must be used for attribute “description”. The INT maps to the attriute “level” of the CompetencyLevel element in the abstract syntax. References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

10.6.1.4 Practice Elements

A WorkProduct declaration and its usage in an AlphaManifest declaration are defined as:

```
WorkProduct:
        'workProduct' ID ':' STRING (Resource(',' Resource)*)? 'with levels' '{'
Level+ '}' AddedTags?;

Level:
        ('sufficient')? 'level' ID '{' STRING ('checks {' CheckListItem+ '}')? '}'
AddedTags?;

WorkProductManifest:
        'describe' AlphaRef 'by' Cardinality WorkProductRef (',' Cardinality
WorkProductRef)* AddedTags?;
```

The ID creates a unique identifier for this WorkProduct, which maps to the attribute “name”. The STRING is considered as content for attribute “description”. If no STRING is given, the empty string must be used for attribute “description”. The Cardinality maps to the attributes for lower and upper bounds in the WorkProductManifest. References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

An Activity declaration and its contents are defined as:

```
Activity:
        'activity' ID
            '{' (
                (STRING)?
                (Resource(',' Resource)*)?
                'targets' (StateRef | LevelRef)(',' (StateRef | LevelRef))*
                (InputAlpha)?
                (OutputAlpha)?
                (Input)?
                (Output)?
                (CompetencyRequirement)?
            ) '}' AddedTags?;
```

```
Input:
      'input {' WorkProductRef (',' WorkProductRef)* '}'

Output:
      'output {' WorkProductRef (',' WorkProductRef)* '}'
```

The ID creates a unique identifier for this Activity, which maps to the attribute “name”. The STRING is considered as content for attribute “description”. If no STRING is given, the empty string must be used for attribute “description”. References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

An ActivityManifest declaration is defined as:

```
ActivityManifest:
      'do' ActivitySpaceRef 'by' ActivityRef (',' ActivityRef)* AddedTags?;
```

References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

```
Pattern:
      'pattern'  STRING  (Resource(','  Resource)*)?  '{'  PatternAssociation+  '}'
AddedTags?;

PatternAssociation:
      'with' PatternElementRef (',' PatternElementRef)* 'as' STRING AddedTags?;
```

The STRING in a pattern is considered as content for attribute “kind” and to the “name” in pattern associations. References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

10.6.1.5 Auxiliary Elements

Tags and resources are expressed as:

```
Tag:
      STRING '=' STRING;

Resource:
      'resource' STRING;

AddedTags:
      'tagged with' '{' Tag(',' Tag)* '}';
```

On Tag, STRING refers to the key on left hand side and to value on the right hand side.

Extension elements are expressed as:

```
ExtensionElement:
      'on' AnyElementRef 'in' STRING 'replace' STRING 'with' STRING AddedTags?;
```

The STRINGs refer to targetAttribute, targetSearch and targetReplacement in this order.

A Cardinality can be specified according to the following definition:

```
Cardinality:
      CardinalityValue ('..' CardinalityValue)?

CardinalityValue:
      INT | 'N'
```

An identifier used for reference is either a single token or prefixed as following:

```
ID ('.'ID)*
```

10.6.2 Examples

A complete Alpha declaration for Kernel Alpha “Requirement”:

```
alpha                                                        Requirements:
      "What  the  software  system  must  do  to  address  the  opportunity  and  satisfy
the                                                          stakeholders."
```

```
    with states {
        state Conceived {"The need for a new system has been agreed."
            checks {
                item checkpoint1 {"The initial set of stakeholders agrees
that a system is to be produced."}
                item checkpoint2 {"The stakeholders that will use and
fund the new system are identified."}
                item checkpoint3 {"The stakeholders agree on the purpose
of the new system."}
                item checkpoint4 {"The expected value of the new system
has been agreed."}
            }
        }
        state Bounded {"The theme and extent of the new system is clear."
            checks {
                item checkpoint1 {"Stakeholders involved in developing
the new system are identified."}
                item checkpoint2 {"It is clear what success is for the
new system."}
                item checkpoint3 {"The stakeholders have a shared
understanding of the extent of the proposed solution."}
                item checkpoint4 {"The way the requirements will be
described is agreed upon."}
                item checkpoint5 {"The mechanisms for managing the
requirements are in place."}
                item checkpoint6 {"The prioritisation scheme is clear."}
                item checkpoint7 {"Constraints are identified and
considered."}
                item checkpoint8 {"Assumptions are clearly stated."}
            }
        }
        state Coherent {"The requirements provide a coherent description of
the essential characteristics of the new system."
            checks {
                item checkpoint1 {"The requirements are captured and
shared with the team and the stakeholders."}
                item checkpoint2 {"The origin of the requirements is
clear."}
                item checkpoint3 {"The rationale behind the requirements
is clear."}
                item checkpoint4 {"Conflicting requirements are
identified and attended to."}
                item checkpoint5 {"The requirements communicate the
essential characteristics of the system to be delivered."}
                item checkpoint6 {"The most important usage scenarios for
the system can be explained."}
                item checkpoint7 {"The priority of the requirements is
clear."}
                item checkpoint8 {"The impact of implementing the
requirements is understood."}
                item checkpoint9 {"The team understands what has to be
delivered and agrees that they can deliver it."}
            }
        }
        state SufficientlyDescribed {"The requirements describe a system that
is acceptable to the stakeholders."
            checks {
                item checkpoint1 {"The stakeholders accept the
requirements as describing an acceptable solution."}
                item checkpoint2 {"The rate of change to the agreed
requirements is relatively low and under control."}
                item checkpoint3 {"The value provided by implementing the
requirements is clear."}
                item checkpoint4 {"The parts of the opportunity satisfied
by the requirements are clear."}
```

```
                }
            }
            state Satisfactory {"The requirements that have been addressed partially satisfy the need in a way that is acceptable to the stakeholders."
                checks {
                    item checkpoint1 {"Enough of the requirements are addressed for the resulting system to be acceptable to the stakeholders."}
                    item checkpoint2 {"The stakeholders accept the requirements as accurately reflecting what the system does and doesn’t do."}
                    item checkpoint3 {"The set of requirement items implemented provide clear value to the stakeholders."}
                    item checkpoint4 {"The system implementing the requirements is accepted by the stakeholders as worth making operational."}
                }
            }
            state Fulfilled {"The requirements that have been addressed fully satisfy the need for a new system."
                checks {
                    item checkpoint1 {"The stakeholders accept the requirements as accurately capturing what they require to fully satisfy the need for a new system."}
                    item checkpoint2 {"There are no outstanding requirement items preventing the system from being accepted as fully satisfying the requirements."}
                    item checkpoint3 {"The system is accepted by the stakeholders as fully satisfying the requirements."}
                }
            }
        }
```

A minimal declaration of an Activity Space using the Alpha declared above:

```
activitySpace SpecifyTheSystem {
      targets Requirements.SufficientlyDescribed
}
```

Some examples for work product declarations:

```
workProduct DeveloperTest:
      "The Developer Test validates a specific aspect of an implementation element."
      with levels {
            level Sketched {""}
            sufficient level Implemented {""}
      }

workProduct Implementation:
      "Software code files, data files, and supporting files (such as online help files) that represent the raw parts of a system that can be built."
      with levels {
            level Stubs {""}
            level Partial {""}
            sufficient level Clean {""}
      }
```

An example for a activity declaration:

```
activity ImplementSolution {
      targets Implementation.Partial, TestableSystemFeature.Tested
      input {DeveloperTest, Implementation}
      inputAlphas {Requirements, SoftwareSystem}
      output {Implementation}
}
```

An example for a practice declaration using some of the elements defined above as well as some others not defined in the previous examples:

```
practice                                          TestDrivenDevelopment:
      ""
      contains                                                         {
            //                                                     Alpha
            TestableSystemFeature,
            //                        Work                      Products
            Implementation,DeveloperTest,
            //                       Activity                     Spaces
            ImplementTheSystem,
            //                                                Activities
            ImplementDeveloperTests,     RunDeveloperTests,     ImplementSolution
      }

      //                  Work                  Product               Manifest
      describe  TestableSystemFeature  by  1  Implementation,  1  DeveloperTest

      //                           Activity                       Manifest
      do                         ImplementTheSystem                      by
ImplementDeveloperTests,RunDeveloperTests,ImplementSolution
```

Annex A: Responses to RFP Requirements

(Informative)

This annex provides the responses to the RFP requirements. The following tables provide a cross-reference between the requirements as stated in the Request for Proposal and the corresponding responses provided by this submission.

A.1 Mandatory Requirements

Table 23 – Mandatory Requirements (Kernel)

Requirement	**Resolution**
6.5.1.1 Domain model The Kernel shall be represented as a domain model of a small number (expected to be closer to 10 than a 100) of essential concepts of software engineering and their relationships. The Kernel shall be expressed in the Language.	The Kernel contains 7 Alphas and 15 Activity spaces capturing the essentials of software engineering from the perspective of the things to work with and the things to be done. The Kernel is defined and presented using the language. • The Kernel may be extended to identify the essential competencies required to undertake a software engineering endeavor. This is likely to add another 5 or 6 elements. • The Kernel may be extended to include a number of essential sub-alphas such as practice, tool, work item, requirements item, system element, stakeholder representative, team member etc. These would have minimal state graphs that would be either used as is or extended to support specific practices. This would add another 10 – 15 elements.
6.5.1.2 Key conceptual elements The Kernel shall define the key conceptual elements that all software engineering endeavors have to monitor, sustain and progress, covering at least the following kinds of concepts (the specific grouping used here is not required): a. *System:* Concepts related to the system being produced, for example: software, platform, etc. b. *Functionality:* Concepts related to the required function of the system being produced, for example: requirements, needs, opportunities, stakeholders, etc. c. *People:* Concepts related to the people required to create a system with the required functionality, for example: project, team, role, etc. d. *Way of Working:* Concepts related to the way an organized team carries out its work to create a system with the required functionality, for example: method, practice, goal, etc.	The Kernel's three areas of concern (see Section 8.2, 8.3 and 8.4) and their corresponding Alphas provide this coverage: • a. Covered by the alpha Software System (see Section 8.3.2.2). • b. Covered by the alphas Requirements (see Section 8.3.2.1), Stakeholders (see Section 8.2.2.1) and Opportunity (see Section 8.2.2.2). • c. Covered by the alpha Team (see Section 8.4.2.1). • d. Covered by the alphas Work (see Section 8.4.2.2) and Way-of-Working (see Section 8.4.2.3).

6.5.1.3 Generic activities The Kernel shall define the generic activities that a team will need to undertake to successfully engineer and produce a software system, covering at least the following kinds of activities (the specific grouping used here is not required): a. *Interacting with stakeholders:* Activities related to necessary interactions with stakeholders, for example: exploring possibilities, understanding needs, ensuring satisfaction, handling change, etc. b. *Developing the system:* Activities related to actually constructing a system, for example: specifying, shaping, implementing, testing, deploying and operating the system. c. *Managing the project:* Activities related to managing a project, for example: steering the project, supporting the project team, assessing progress and concluding the project.	The Kernel's three areas of concerns (see Section 8.2, 8.3and 8.4) and their corresponding Activity Spaces provide this coverage: • a. Covered by the activity spaces in the Customer area of concern (see Section 8.2.3). • b. Covered by the activity spaces in the Solution area of concern (see Section 8.3.3). • c. Covered by the Endeavor area of concern (see Section 8.4.3).
6.5.1.4 Kernel elements The definition of each element of the Kernel shall include the following: a. A concise description of the meaning of the element and its use in software engineering, intuitively understandable to a practitioner. b. The relationships of the element to other elements in the Kernel. c. The various different states the element may take over time, including initial/entry and final/exit criteria as appropriate for the element. d. How the element is applied in practice, including how it may be instantiated, tailored or extended to support the work of a specific project team using specific practices. e. How different ways of applying the element may be compared to each other and guidance on deciding among the alternatives. f. Appropriate metrics that can be used to assess progress, quality, etc.	The Kernel element definitions cover: • a. See the element descriptions. • b. See Figure 3, Figure 4, the Alpha Associations, and the Activity Space Completion Criteria. • c. Each Alpha has a state graph and, for each state, entry criteria. Each Activity Space has completion criteria. • d. This will be covered by the examples. • e. This will be covered by the examples. • f. The Alpha states allow the measurement of progress and a subjective assessment of quality. More empirical measures can be added alongside the sub-alphas as part of maturing the kernel specification
6.5.1.5 Scope and coverage The Kernel shall be sufficient to allow for the definition of practices and methods supporting projects of all sizes and a broad range of lifecycle models and technologies used by significant segments of the software industry.	The Kernel can be extended to specific segments of the software industry by creating kernel extensions and specific practices. The Kernel is light-weight enough to be applied to even the smallest of projects and comprehensive enough to support even the largest of software endeavors. The Alphas states can be used to define all types of lifecycle model from the most lightweight agile lifecycle through more formal iterative lifecycles to the most formal and traditional

	waterfall lifecycles. See the lifecycle examples provided in Section C.1.3.
6.5.1.6 Extension The Kernel shall also allow for extension, both in terms of addition of new elements and providing additional detail on existing elements that provide for practice-specific work products. a. The Kernel shall allow for project and organization specific extensions. b. The Kernel shall be tailorable to specific domains of application and to projects involving more than software, e.g., to serve as a basis for future extensions for systems engineering.	The language allows Kernels to refer to other Kernels that are based on via composition. This way, elements of two or more Kernels can be merged to be used together in a specific situation. The composition algebra also allows merging two elements into one, that is, extending one element with the contents of the other element.

Table 24 – Mandatory Requirements (Language)

Requirement	**Resolution**
6.5.2.1.1 MOF metamodel The Language shall have an abstract syntax model defined in a formal modeling language. The submission is expected to reflect this requirement in a description or mapping to the OMG architectural framework based on MOF.	The definition of the abstract syntax is based on MOF.
6.5.2.1.2 Static and operational semantics The Language shall have formal static and operational semantics defined in terms of the abstract syntax.	See Section **10.2** for the static semantics and section **10.4** for the dynamic semantics.
6.5.2.1.3 Graphical syntax The Language shall have a graphical concrete syntax that formally maps to the abstract syntax. The submission is expected to reflect this requirement in a description following the Diagram Definition specification [DD] unless arguments are given for choosing something else.	See Section **10.5** for the definition of the graphical syntax. It is not based in the Diagram Definition specification, since this specification was only available in a beta version at the time of writing.
6.5.2.1.4 Textual syntax The Language shall also have a textual concrete syntax that formally maps to the abstract syntax.	See Section **10.6** for the definition of the textual syntax.
6.5.2.1.5 SPEM 2.0 metamodel reuse Proposals shall reuse elements of the SPEM 2.0 metamodel where appropriate. Where an apparently appropriate concept is not reused, proposals shall document the reason for creating substitute model elements.	This is discussed in Annex B: Section B.2.

6.5.2.2.1 Ease of use The Language shall be designed to be easy to use for practitioners at different competency levels: a. Those that have very little modeling experience and quickly and intuitively need to understand and learn how to use the Language. b. Intermediate users who are more advanced and willing to describe what kind of outcome they expect of their work. c. Advanced users that can work with all aspects of the Language to model their complete software endeavor.	The abstract syntax of the language provides a concept of layers, where each layer provides a subset of language elements. The graphical syntax of the language provides a concept of views, where each view is concerned with specific aspects of a kernel or method. This can be used on different competency levels: • a. Users with little modeling experience use only language layers 1 and 2 and views on Alphas and Work Products. • b. Intermediate users use language layer 3 and the view on Activities in addition. • c. Advanced users use all 4 language layers and add more sophisticated views not defined in this specification.
6.5.2.2.2 Separation of views for practitioners and method engineers The Language shall provide features to express two different views of a method: the method engineer's view and the practitioner's view. The primary users of methods and practices are practitioners (developers, testers, project leads, etc.). The proposal shall be accessible to both practitioners and method engineers, but should target the practitioners first and foremost. Extensions should support method engineers to effectively define, compose and extend practices, without complicating its usage by the practitioners.	The views defined in this language specification are simple views suitable for practitioners. They focus on a small set of elements in each view and are thus easily accessible. Moreover, no knowledge about composition is needed to define simple practices. The language specification allows to define additional views on language constructs which suit the needs of method engineers. The composition algebra allows to compose language constructs in many ways, including composition of practices and extension by composition. However, composed practices are not handled differently from simple practices, so accessibility for practitioners is not limited.
6.5.2.2.3 Specification of kernel elements The Language shall have features for specifying Kernel elements, including: a. Formal and informal descriptions of the content and meaning of an element. b. The relationship of the element of other elements. c. States the element may take over time and the events that cause transitions among those states. d. How the element is instantiated, including provisions for practice-specific tailoring of the element, and the basis for comparing different instantiations. e. Metrics defined to assess various attributes of the use of the element.	The language defines (amongst others) elements "Alpha" (see Section 10.2.1.1) and "Activity Space" (see Section 10.2.3.4) for specifying Kernel elements. The language include: • a. Attributes for covering natural language descriptions of these elements as well as state graphs (on Alphas) and completion criteria (on Activity Spaces) to formally express the key semantics of these elements. • b. Alphas and Activity Spaces that can be related to each other via states on completion criteria. Alphas can be related to other Alphas via Alpha Associations. • c. Alphas that own state graphs. Transition among these states is covered by the dynamic semantics. • d. Instantiation of Alphas that is covered by the dynamic semantics. • e. The dynamic semantics which include proposals on functions measuring progress or health of an endeavor based on the number of Alphas that are instantiated or the states they have reached.
6.5.2.2.4 Specification of practices	The language specification provides an element "Practice" (see Section 10.2.1.11) which is used and which relates to the Kernel

The Language shall have features for specifying practices in terms of Kernel elements, including: a. Description of the particular cross-cutting concern addressed by the practice and the goal of the application of the practice. b. The Kernel elements relevant to the practice and how they are instantiated for use in the practice, including any practice-specific tailoring of the elements. c. Any work products required by and produced by the practice. d. The expected progress of work under the practice, including progress states, the rules for transition between them and their relation to the states of relevant Kernel elements used in the practice. (For example, describing a practice that involves iterative development requires describing the starting and ending states of every iteration.) e. Verification that the goal of the practice has been achieved in it application, particularly in terms of measurements of metrics defined for its elements.	elements in the following ways: • a. The element "Practice" owns a description. By looking at the Alphas used in this Practice it can be determined in which area this practice can be used. • b. The element "Practice" can use Alphas and Activity Spaces from the Kernel. Through composition, it can redefine parts of these Kernel elements if necessary. Instantiation of these elements is not specific to practices. • c. The element "Practice" uses AlphaManifests to relate WorkProducts to Alphas. • d. Progress in general is covered by the state graphs on Alphas and WorkProducts. Iterations can be covered by Sub-Alphas, allowing to track states for each iteration individually. • e. The dynamic semantics can be used to determine whether all Kernel elements are in their final states.
6.5.2.2.5 Composition of practices The Language shall have features for the composition of practices, to describe existing and new methods, including: a. Identifying the overall set of concerns addressed by composing the practices. b. Merging two elements from different practices that should be the same in the resulting practice, even if they have different contents defined in the practices being composed. (For example, a use case practice may have a work product called Use Case, with a name, a basic flow etc. A testing practice may have a work product called Testable Requirement with an identifier and a description. In the method resulting from composing these two practices, these two work products should be merged into one, where the name of the Use Case is the identifier of the Testable Requirement and the basic flow of the Use Case is the description of the Testable Requirement). c. Separating two elements from different practices that should be different in the resulting practice, even though they may superficially seem to be the same. (For example, in a testing practice there may be a work product called Plan and in an iterative development practice there may also be a work product called Plan. In the method resulting from composing these two practices these two work products must be different – e.g., the Testing	The composition algebra allows for composition of practices. • a. Composed practices are in general not different from simple practices, so the concerns addressed by a composed practice can retrieved from looking at the alphas used in the composed practice. • b. The composition algebra allows for renaming of elements so that different elements can be renamed to be safely identified. Contents are merged recursively. Conflicts on descriptions have to be solved manually. • c. Renaming can also be used for changing names prior to merging, so that elements can be kept distinguishable even if they look similar in the original practices. • d. Methods know the practices they are composed of so they can be modified by redoing the composition with partially the same and partially new practices.

Plan vs. the Development Plan.) d. Modifying an existing method by replacing a practice within that method by another practice addressing a similar cross-cutting concern.	
6.5.2.2.6 Enactment of methods The semantic definition of the Language shall support the enactment by practitioners of methods defined in the Language, for the purposes of a. Tailoring the methods to be used on a project. b. Communicating and discussing practices and methods among the project team. c. Managing and coordinating work during a project, including modifications to the methods over the course of the project by further tailoring the use of the practices in the method. d. Monitoring the progress of the project. e. Providing input for tool support for practitioners on the project.	• a. Any composition of practices can be instantiated as a method and used on a particular endeavor, as long as it addresses the concerns of this endeavor. • b. Different methods can be queried for advice in a particular situation (as long as the methods address the concerns at hand), so team can discuss the different advices and communicate differences between methods based on them. • c. Dynamic semantics are partially defined as denotational semantics using the overall state of the endeavor as input, thus not being dependent on using the same method definition each time. • d. Tracing the overall state of the endeavor is part of the dynamic semantics. • e. Dynamic semantics can partially be formalized, so they can also be implemented in tools.

Table 25 – Mandatory Requirements (Practices)

Requirement	**Resolution**
6.5.3.1 Examples of Practices a. Submissions shall provide working examples to demonstrate the use of the Kernel and Language to describe practices. Preferably these examples should be drawn from existing and well-known practices. b. Submissions shall provide working examples to demonstrate the composing of practices into a method. c. Submissions shall provide working examples to demonstrate how a method can be enacted. d. Submission shall include a capability to demonstrate the operational execution of methods as a proof of concept. It is expected that the example practices are well-structured and suited to demonstrate how well the proposed Kernel and Language can be used to define good-quality practices. Each example of practice shall: a. be described on its own, independent from any other practice b. be either explicitly defined as a continuous	A set of examples is described in Annex C: • a. See Section C.1. • b. See Section C.2. • c. See Section C.3. • d. See Section C.3.

activity or have a clear beginning and end states c. bring defined value to its stakeholders d. be assessable; in other words, its description must include criteria for its assessment when used e. include, whenever applicable, quantitative elements in its assessment criteria; correspondingly, the description must include suitable assessing metrics.	

A.2 Optional Requirements

None

Annex B: Issues to be Discussed

(Informative)

This annex provides the discussions on issues to be discussed from the RFP.

B.1 Kernel

This annex contains a discussion of the alternative options considered for the kernel elements defined in the Kernel Specification. The Annex is presented in two sections:

1. Alphas – Alternatives for the names of the Alphas used in the kernel specification.
2. Activity Spaces - Alternative sets of Activity Spaces and Activity Space names.

Note: The Alphas are presented first as they were defined first and heavily influenced the selection and naming of the Activity Spaces

B.1.1 Alphas

B.1.1.1 Alternatives Considered but Rejected for Opportunity

Opportunity – the set of circumstances that makes it appropriate to develop or change a software system.

On a grand scale, the opportunity to which the software system is addressed could be:

- To go into space – needs software systems on board the spacecraft, for communication, and on the ground.
- To run a chemical plant - needs logistics systems for shipping in and out, process control, new production processes.
- To provide a new mobile phone platform - needs applications in the phone and on the web.
- To re-organize a business or government department - must continue to serve demands from customers and the public as software systems are updated, "migrated" or retired.

In a business context, opportunities could include:

- Increase customer satisfaction – for example by a focus on end-to-end performance of the business in customer terms.
- Decrease staff costs – for example by allowing expert systems to respond to customer enquiries.
- Provide better local weather forecasts – for example by using automation based on new research in meteorology.

On a more personal level, opportunities (motives) could include:

- To make my fortune by producing a hit game.
- To publicize my business to rich people.
- To educate and entertain.

The following alternatives were considered but rejected as their definitions were considered too vague or too narrow in scope. **What was required was a word that best brought together the meanings of all the alternatives.**

- **Business Context** – considered too vague to be useful. Teams need to identify the opportunity that the business context provides.
- **Domain of Expertise** – doesn’t capture the concrete opportunity / problem to be addressed.
- **Effect** – sounds too much like a side effect of the work rather than its intent.

- **Goal** – considered too general. This would be too easily confused with the use of goals in project management and other practices.
- **Motive / Motivation / Incentive** - good ways to think about the opportunity but rejected as too abstract and conceptual for most readers.
- **Needs** – considered too confusing when compared and contrasted with requirements.
- **Objectives** - considered too general. This would be too easily confused with the Team’s short-term objectives.
- **Problem / Underlying Problem** – considered too negative.
- **Purpose** – too easily confused with the requirements. It doesn’t reflect the opportunity to be addressed, and is more commonly used to construct sentences such as “the purpose of the software system is to address the opportunity”.
- **Value** – too confusing as many of the other alphas will have value associated with them. An essential property of any opportunity but considered too confusing for use as an alternative to opportunity.

B.1.1.2 Alternatives Considered but Rejected for Stakeholders

Stakeholders – The people, groups, or organizations who affect or are affected by a software system.

There are many different types of stakeholders and stakeholder groups, including:

- Users - people who use the system. One very important type of stakeholder is the user. These are a prime example of a set of stakeholders that must be involved in the development of the software system.
- Project Steering Committees / User Groups / User Communities made up of the project sponsors, users and other people affected by the development and maintenance of the software system. Many projects have a project steering committees made up of the project sponsor, the senior supplier, the senior user and other stakeholders or their representatives. This is one of the practices available to help involve the stakeholders. The same can be said for structures such as User Groups and User Communities.
- Customers and Sponsors, people who finance the development and maintenance of the software system. They are also known as the “gold owners”.
- Back-end support stakeholders such as Maintainers and Developers developing, evolving and maintaining the software system.
- Support and Operations made up of technicians providing feedback on the usage of a software system and supporting its use.
- Scrum Chickens, part of the stakeholder community in Scrum. Scrum acknowledges the presence of different types of stakeholders in its concept of pigs and chickens where the development team members are the pigs and the rest of the stakeholders, such as users and sponsors, are the chickens. Scrum focuses all of the involvement of the stakeholders through the single role of the Product Owner, which is one of the many practices available for managing the stakeholders.

The following alternatives were considered but rejected as their definitions were considered too vague or too narrow in scope. What was required was a word that best brought together the meanings of all the alternatives.

- **Customer** – this was explored as a candidate name in an attempt to show that software engineering is customer focused, but was rejected because 1) not all software engineering endeavors have customers in the traditional sense, 2) confusion arose between customers and users, purchasers, and sponsors, and 3) there are many stakeholders that people don’t consider to be customers such as internal governance bodies.
- **External Stakeholders** – rejected because there are many circumstances where members of the team are also stakeholders.
- **Set of Stakeholders** – although it has the benefit of stressing the fact that it represents all of the stakeholders it was rejected as too cumbersome for natural language use.
- **Stakeholder Community** - although it has the benefit of stressing the fact that it represents all of the

stakeholders it was rejected as too cumbersome for natural language use.

- **Users, Sponsors etc** - rejected because they are each only one type of stakeholder.

B.1.1.3 Alternatives Considered but Rejected for Requirements

Requirements: What the software system must do to address the opportunity and satisfy the stakeholders.

There are many different examples, and ways, of capturing the requirements including:

- In a development context: Declarative Requirement Documents, Use Cases, User Stories and Tests (text and or code) can all be used to record the Requirements.
- In a continuing context: Training, Service Level Agreements, Problem Investigations, Process Controls may depend on an understanding of the Requirements, and may over time contribute to learning more about them.
- In an explicit context: A specification of system attributes, with desired and measureable levels, can constitute the Requirements.
- In an implicit context: The Requirements may simply be that the Software System, or some part of it, must continue in use.

The following alternatives were considered but rejected as their definitions were considered too vague or too narrow in scope. What was required was a word that best brought together the meanings of all the alternatives.

Concerns – this was considered but was quickly discarded as being too vague and not immediately meaningful to the software engineering community.

- **Intent** – this one was considered in depth as a way of circumventing some of the bad feeling towards the word requirements in parts of the agile community. Intent is defined as “something that is intended; an aim or purpose”.

  Requirements is preferred to intent because it is more concrete and it represents a specification (whether it be explicit or tacit) against which the Software System will be accepted (and typically must be demonstrated to conform). Requirements stand for something that is required and is a necessity or obligation. In comparison with intent, requirements connote the idea of obligation or a must whereas intent connotes the idea of objective or desire. Intent was also considered to be a little too abstract to resonate with the majority of the software engineering community.

- **Requirement** – Some people would have preferred the term to be used in its singular form. Unfortunately using the singular of a definition with the word must in can lead people to think that every detailed requirement statement must be met by the software system produced. This is not the intent. “Requirement” is ambiguous because it could mean “the requirement” (for the whole system, i.e. a synonym for “the specification”) or it could mean “a requirement” (i.e. one of many that together comprise the overall requirement / specification).

- **Specification** – Wikipedia (http://en.wikipedia.org/wiki/Specification_%28technical_standard%29) defines “A specification is an explicit set of requirements to be satisfied by a material, product, or service.” In some methods there is a focus on the production of some form of external / functional specification to which the system must conform. This is often the intent of the requirements documentation.

  This term was rejected as it is too easily confused with the technical design specifications that may also be produced and because it sounds very heavy-weight.

- **Usage** - Although it is generally considered to be good practice to capture the requirements in some form of usage based description (be it scenarios, use cases or user stories) it was felt that usage was too restrictive a term and may cause practitioners to not look at their requirements holistically enough to really capture the desires of their stakeholders.

B.1.1.4 Alternatives Considered but Rejected for Software System

Software System: A system made up of software, hardware, and data that provides its primary value by the execution of the software.

There are many types of software system that can be the result of software engineering including:

- Purpose-built (bespoke) facilities including research, simulation, data capture and analysis for a scientific enterprise, such as drug discovery and testing.
- Bespoke software for a consumer platform such as mobile phone applications, games.
- Commercial-Off-The-Shelf (COTS) product for 'shrink-wrapped' sale to customers, such as office productivity.
- COTS products integrated into a business work system. These could be for resource planning (such as SAP Business Management software) or for technical models and visualization (such as Intergraph SmartPlant).

The following alternatives were considered but rejected as their definitions were considered too vague or too narrow in scope. What was required was a word that best brought together the meanings of all the alternatives.

- **Software / Working Software** – This was considered to be too limiting. Is it just running code or does it include all the information involved including the supporting documentation? If a team of people is developing a database application but does not write a single line of code is what they've produced software?

  Software was also considered to be too abstract a concept for the primary output from software engineering as in and of itself it does not require engineering. Software is zeroes and ones, in the form of computer programs and the data that they manipulate. To be useful software requires there to be a suitable computing platform upon which it can be run. The output of software engineering must also consider the computing platform as well as the software.
- **System** - Although often used within computing circles this was considered to be too general. The consensus was that all engineering disciplines produce some kind of system, and therefore software engineering needs to produce something more specialized than just a system.

  It was also thought that using system as a software engineering universal would cause confusion and friction with the systems engineering community.
- **Software Intensive System** - Originally proposed as the name, and rejected as it was considered to be limiting; software engineering is also important in some systems that are not primarily software systems. It was also considered to be too cumbersome.
- **Product / Software Product** – It seemed a little too abstract to call the product of software engineering product. There was also the problem of interpretation. Typically the term product is interpreted in one of two ways:
  - commodities offered for sale; "that store offers a variety of products"
  - an artifact that has been created by someone or some process; for example "they improve their product every year"; "they export most of their agricultural production"

  The first interpretation implies a much greater scope than just producing working software systems – it would imply that software engineering should always include marketing and product management activities and that it always produces a software intensive system that is to be sold.

  It was also considered to be too generic - there are many disciplines that produce artifacts that can be sold or treated as products. We need a universal that helps to differentiate software engineering from other forms of production and related professions that strive to produce products (such as catering and fashion industries).
- **Service** - Although it is hoped that the results of software engineering will be of service, and provide useful services to their users, to consider the product of software engineering to be a service rather than a form of goods is probably a step too far.
- **Solution** - The term solution often implies something potentially far-greater than the software system being produced. It was also considered to be too generic – there are many disciplines that produce solutions. We need a kernel that helps to differentiate software engineering from other forms of engineering and related professions that strive to produce solutions (such as medicine and politics).

B.1.1.5 Alternatives Considered but Rejected for Work

Work: Activity involving mental or physical effort done in order to achieve a result.

Examples of evidence of work in software engineering endeavors include:

- The Scrum Sprint Backlog.
- Team Task Lists.
- Work item Lists.
- Project Work Breakdown Structures.
- Work Packages.

The following alternatives were considered but rejected as their definitions were considered too vague or too narrow in scope. What was required was a word that best brought together the meanings of all the alternatives.

- **Project** - A project is one of many ways of organizing the work to be done. Project was rejected because much software engineering is done within product centers and application development teams where the development work is seen as on-going and not managed as a series of projects.

  There is also the issue of organizing support and maintenance work, which again is often not managed as a series of projects.
- **Task** - A task is typically seen as a unit of work, and a way of breaking down the work into individually addressable work items to be managed within a project plan or via a task board. Task is too specific and find-grained a term to be used to represent the work in its entirety.
- **Activity** – This was considered too general for use in the kernel. It would also cause confusion by clashing with the Kernel Language's use of the term activity.
- **Endeavor** – This was considered too abstract to appeal to most software engineers.

B.1.1.6 Alternatives Considered but Rejected for Way of Working

Way-of-Working: The tailored set of practices and tools used by a team to guide and support their work.

There are many different examples of teams adopting a specific way of working:

- Methods such as Dynamic Systems Development Method (DSDM).
- Processes such as the Rational Unified Process (RUP).
- Frameworks such as Scrum and Kanban.
- Bodies of knowledge such as SWEBOK, PMBOK and ITSQB.
- Practices such as Test-Driven Development and Continuous Integration.
- Maturity Models such as CMMI.

The following alternatives were considered but rejected as their definitions were considered too vague or too narrow in scope. What was required was a word that best brought together the meanings of all the alternatives.

- **Method** – not an appealing word to developers and other practitioners. Most practitioners see a method as being a formal, comprehensively described description of what they are supposed to do, rather than a description of what they actually do. If you ask a team to describe their way-of-working they will tell you what they do, if you ask them to describe their method they will either claim that they don't have one or point you at a stack of documentation that they generally ignore.
- **Process** – not an appealing word to developers and other practitioners. Suffers from the same problems as method.
- **Methodology** – actually means the study of methods.
- **Approach** – considered too vague a name for such an important kernel element.

B.1.1.7 Alternatives Considered but Rejected for Team

Team: The group of people actively engaged in the development, maintenance, delivery and support of a specific software system.

Software engineering is a team sport and typically involves at least one team. Types of team and team structure used in software engineering include:

- The Cross-Functional Development Team – A small team containing all the skills needed to develop a working software system, as used in Scrum and other agile methods.
- Feature Teams and Component Teams – Types of cross-functional team organized around the requirements and the architecture.
- The Segregated Team – A team that is made up of a number of specialist teams such as:
    - The Management Team.
    - The Requirements Team.
    - The Development Team.
    - The Testing Team.
    - The Support Team.
- The Maintenance Team – A team focused on doing maintenance and makings small changes to a software system.
- The Team of Teams – A team made up of a number of other teams.

The following alternatives were considered but rejected as their definitions were considered too vague or too narrow in scope. What was required was a word that best brought together the meanings of all the alternatives.

- **Development Team / Software Development Team / Software Engineering Team** - The term development team was originally proposed, but it was decided to drop the word development because it was felt it conveyed the wrong meaning, implying that team membership is limited only to software developers. Some people argued that the qualifiers made the role of the team clearer but within the context of software engineering, and our software engineering kernel, the role and purpose of the team is quite clear.

  The same reasoning holds for Software Development Team and Software Engineering Team.
- **Production Team / Enactment Team / Delivery Team** - The word "Production" could be used to help classify the team as the one actively involved in undertaking and participating in the work. "Production" distinguishes this team from other interested parties that whilst influencing, guiding and supporting the endeavor are not working directly on development activities.

  The term is in general use in the production of plays, television shows and films to describe the group of variously skilled people working to produce the play, TV show or film in question. This also has a high degree of resonance when applied to the team working on a software system.

  This term is rejected as too heavy and cumbersome, and also too limiting. The fact the Team is the Production Team can be seen from its relationship with the software system and the stakeholder community. Within the context of software engineering, and our set of software engineering universals, the role and purpose of the team is quite clear.

  The same reasoning holds for Enactment Team and Delivery Team.
- **People, Software People, Software System People, Software Engineers** - Whilst these terms do perhaps classify the interests of the group it does not suggest any accountability for the work or endeavor.

  The term 'people' was rejected as too general. The term 'software engineers' was rejected as too limiting (see also Development Team and Production Team).

B.1.2 Activity Spaces

B.1.2.1 Alternative Names for the Activity Spaces

Alternative names were considered for each of the activity spaces included in the Kernel Specification. Table 26 shows the various names considered for the Activity Spaces in the Customer Area of Concern.

Table 26 – Alternative Names for the Customer Activity Spaces

Name	Alternative	Comments
Explore Possibilities	Understand the Need	'Understand the Need' sounded too much like it should deal with the requirements rather than the stakeholders and the opportunity.
Involve Stakeholders	Engage Stakeholders	'Involve' was preferred to 'Engage' as it reinforces the fact the stakeholders must be active in supporting the team.
Ensure Stakeholder Satis-faction	Accept the System	The purpose here is to make sure that the stakeholders are happy with the software system produced, and not to force them to accept something they don't want. This is why 'Ensure Stakeholder Satisfaction' was preferred.
Use the System	Exploit the System	'Exploit' sounded too much like sales and marketing to resonate with software developers.

The merging of the two Activity Spaces 'Engage Stakeholders' and 'Ensure Stakeholder Satisfaction' into a single Activity Space was also considered but was rejected as it would have covered too many state changes.

Table 27 shows the various names considered for the Activity Spaces in the solution Area of Concern.

Table 27 – Alternative Names for the Solution Activity Spaces

Name	Alternative	Comments
Understand Requirements	Specify the System	'Specify the System' sounded very heavyweight and un-agile. 'Understand Requirements' was judged to more accurately reflect the purpose of the Activity Space and to be more widely acceptable.
Shape the System	Architect the System	Both of these alternatives seemed to be suggesting specif-ic approaches to achieving the underlying state changes.
	Design the System	
Implement the System	Implement Software	There is more than just implementing the software in-volved in implementing a software system.
	Create the System	'Create the System' sounded too much like green-field development where no earlier version of the software system exists.
Test the System	Verify the System	'Test' was considered to be simpler and more intuitive than the more formal sounding 'Verify'
Deploy the System	Release the System	These alternatives were all considered to just be one as-

	Package the System	pect of deploying the system.
	Deliver the System	
	Go Live	
Operate the System	Support the System	‘Operate’ was judged to communicate the purpose of the Activity Space better than ‘Support’.

Table 28 shows the various names considered for the Activity Spaces in the endeavor Area of Concern.

Table 28 – Alternative Names for the Endeavour Activity Spaces

Name	**Alternative**	**Comments**
Prepare to do the Work	Start the Work	The purpose of the Activity Space is to get ready to start the work, hence this alternative was rejected.
	Prepare the Endeavor	This alternative was judged less intuitive than ‘Prepare to do the Work’.
Co-ordinate Activity	Co-ordinate the Work	More than just the work is being coordinated.
	Steer the Work	‘Steer the Work’ was judged to be less accessible than ‘Coordinate Activity’. Also more than just the work is being coordinated.
Support the Team		No alternatives were suggested.
Track Progress	Track the Work	More than just the work is being tracked.
	Do the Work	Seemed to contradict the purpose of the Activity Spaces all of which contain work to be done.
	Assess Progress	Sounds too judgmental.
Stop the Work	Conclude the Endeavor	This alternative was judged less intuitive than ‘Stop the Work’.
	Closedown the Work	‘Stop’ seemed simpler and less formal.

The merging of the two Activity Spaces ‘Co-ordinate Activity’ and ‘Support the Team’ into a single Activity Space was also considered but was rejected as it would have covered too many state changes.

B.1.3 Alternative sets of activity spaces

An alternative set of Activity Spaces was also prepared, one that used four areas of concern:

- **People** – This area of concern contains everything to do with the people directly or indirectly in the development of the software system.
- **Purpose** - This area of concern contains everything to do with understanding and specifying what the software system will do.
- **Solution** - This area of concern covers everything to do with the development of the software system.
- **Endeavor** - This area of concern contains everything to do with the work to be done and the way that it is to be

approached.

This is shown in Figure 54. In this model the Alphas were also re-organized to place the team and stakeholders into the new people Area of Concern, and opportunity and requirements into the purpose Area of Concern.

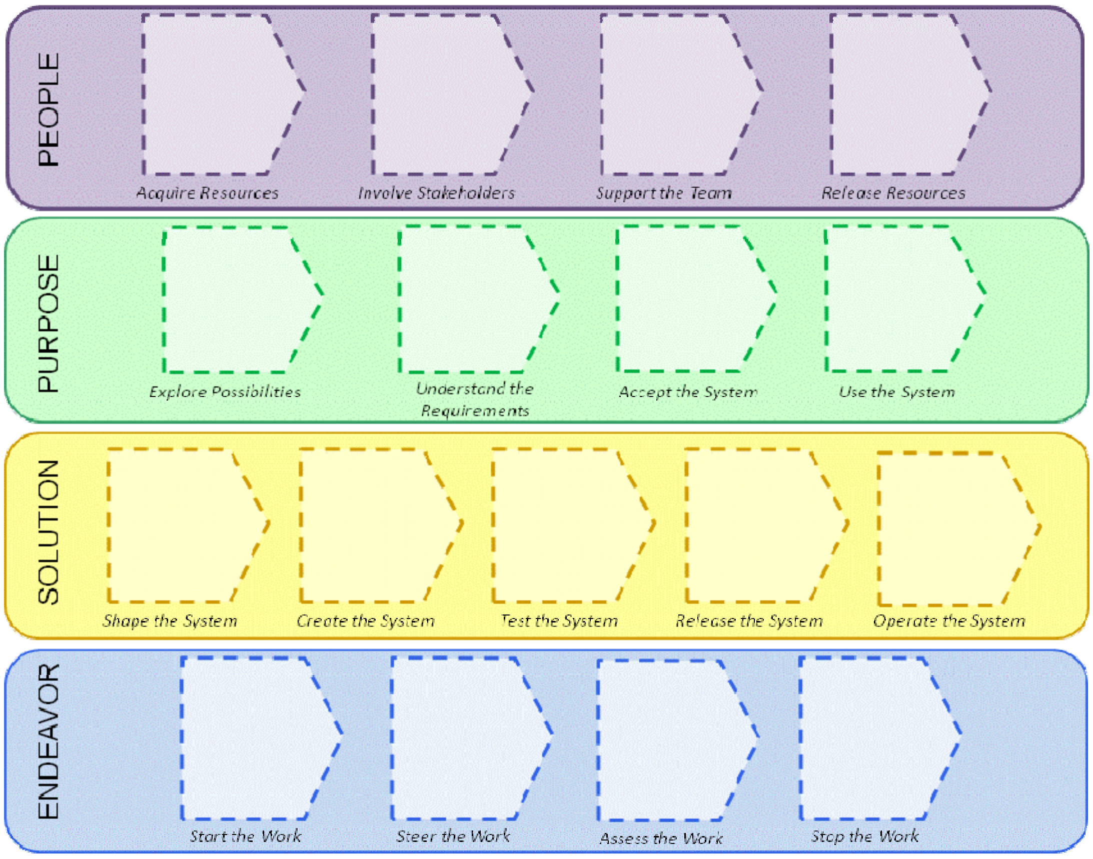


Figure 54 – Alternative Set of Activity Spaces using four Areas of Concern

In this model the number of Activity Spaces was considered to be too many to succinctly represent the things that need to be done as part of any software engineering endeavor. Some of the Activity Spaces were not considered to be discrete enough in particular the separation between ‘Acquire Resources’ and ‘Start the Work’, and ‘Release Resources’ and ‘Stop the Work’. The consensus was that the model included in the Kernel Specification was more intuitive, clearer, and succinct that the one presented here.

B.2 SPEM 2.0

This section discusses why we did not use SPEM 2.0 as a baseline and clearly describes and demonstrates the main differentiators.

B.2.1 Why do you not base your submission on SPEM 2.0?

The vision and requirements of the Essence language is different from that which drove the development of the SPEM 2.0 specification. The main objectives of the Essence language are to:

- Address the mass market of practitioners, not just the limited market of method engineers[6];
- Support a kernel that is able to represent and measure the state and health of a software engineering endeavor;
- Support agility in the adoption and adaption of software engineering practices;
- Have dynamic semantics supporting enactment built in.

These objectives have driven the architecture and design of the Essence language in a separate direction from SPEM. The underlying architecture of SPEM is not compatible with the aims of Essence. It would of course be possible to reengineer SPEM so that it better aligns with these objects. However, we believe that such a reengineered SPEM version will require more than a few changes and the result will be fundamentally different from the current SPEM specification.

Changing the underlying architecture of SPEM is not something that could be achieved with little effort, so such incompatibility points to the need for a new non-SPEM language. Starting afresh makes it easier to innovate and clearly define the essential features of the language.

The Essence language architecture has the following differences compared to SPEM:

- Focused and small specification that is extensible
- Domain-specific language instead of a UML profile.
- Support for dynamic semantics

Because of these architectural features we also believe it to have a:

- Wider market appeal

All of these points are further elaborated in the following subsections.

B.2.1.1 Focused and small specification that is extensible

One main criticism of SPEM is that it is too complex and that we need a simpler language[7] that appeals to the practitioners, which is easy to use and can be extended over time. The goal of the Essence language is not to define and include concepts that are *useful* to most software engineering endeavors, but to identify a small set of concepts language constructs that are *essential* to all software engineering endeavors.

To achieve this, the following ideas have underpinned the Essence language design:

[6] "The Software and Systems Process Engineering Meta-model (SPEM) is a process engineering meta-model as well as conceptual framework, which can provide the necessary concepts for modeling, documenting, presenting, managing, interchanging, and enacting development methods and processes. An implementation of this meta-model would be targeted at process engineers, project leads, project and program managers who are responsible for maintaining and implementing processes for their development organizations or individual projects." [SPEM2, Section 6.2, Page 9]
[7] http://philippe.kruchten.com/2011/03/11/we-do-not-need-richer-software-process-models/

- Define a minimal language focusing on the essentials of software engineering methods that can be used by practitioners.
- Separate the essentials language features from what is useful, what is nice to have and what can be provided as language extensions.
- Focus on practices and have an inherent design in the language that allows us to structure, describe and use practices in an easy and standardized manner.
- Support different user group with require different level of details and views of a method.

If SPEM is extended to accommodate extra ideas/constructs form Essence it will get larger. This size will be a barrier to adoption, especially by smaller organizations who may fear that the costs of adoption (training, tools, learning curves, customization, change management, etc.) will never be compensated by benefits.

In contrast the proposed Essence language architecture provides:

- A small set of concepts capturing the *essentials*.
- Language extensions mechanisms to introduce new *useful* concepts.
- Layering and view concepts to support incremental adoption, learning and understanding.

The extension mechanism should be powerful enough to define missing SPEM features as a possible, standardized language extension library.

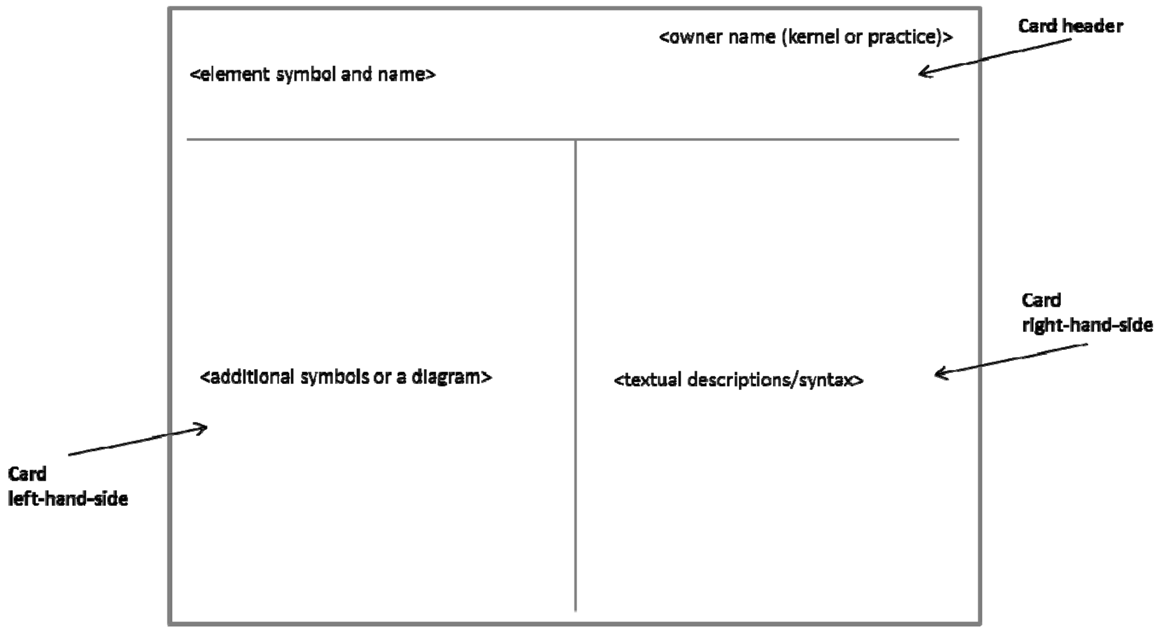
B.2.1.2 Domain-specific language instead of a UML profile

One of the ideas behind SPEM 2.0 was to closely align and reuse elements of UML 2[8] and also define a UML Profile which could be adopted by UML tool vendors. While this may have been common approach back in 2007, where UML was still being promoted as the *universal* language for everything software-related, we have later come to realize that UML is not the universal answer. OMG is embracing a family of different languages (all defined using the same metamodel architecture, i.e., MOF), where UML is just one part of a big family.

The original intent of a UML profile was to tailor UML[9]. We argue that we need to define a new simpler foundation, i.e., metamodel, for the Essence language, and this basis is definitely not UML. UML is useful for describing software architectures and designs, but should not be used as a basis for describing the Kernel, Practices and Methods. By abandoning the dependency to UML, we are no longer restricted to using/extending the UML syntax, and can define a language with an easy-to-use concrete syntax (both textual and graphical).

The concept of a card (shown below) is an example of a useful and rich graphical view that would not be possible to define using the UML profiling constructs.

[8] "The ability to leverage these features, as well as the ability to work with UML 2 tools are powerful enhancements to SPEM 2.0. In addition, there was specific feedback from implementers of SPEM 1.x that have been addressed to make SPEM process models easier to enact and automate." [SPEM2, Section 6.1, Page 8]
[9] http://modelseverywhere.wordpress.com/2010/11/17/bits-of-history-spem-and-uml-profiles/



<owner name (kernel or practice)>
Card header
<element symbol and name>
Card
right-hand-side
<additional symbols or a diagram>
<textual descriptions/syntax>
Card
left-hand-side

Figure 55 – A basic card anatomy to visualize an element

B.2.1.3 Support for dynamic semantics

Another main criticism of SPEM is the lack of support for enactment built in as a feature of the language architecture. Enacting SPEM processes are typically done through mapping[10]. Designing a native dynamic semantics into the language is arguably one of the main requirements that will require a redesign of the SPEM architecture. The Situational Method Engineering (SME) community[11] has argued against SPEM as a baseline and defined its own specification, namely the ISO 24744 standard[12].

The ISO 24744 metamodel defines a dual-layer modeling approach, with a clear separation of methodology elements and endeavor elements, and uses sophisticated metamodelling constructs such as Powertypes and Clabjects to support dynamic semantics.

In the Essence language we introduce the abstract superclasses **my_Alpha**, **my_WorkProduct** and **my_Activity** that correspond to similar classes defined in the static semantics to support dynamic semantics. The dynamic semantics is a key discriminating factor of the Essence language, as compared to other language approaches. The dynamic semantics:

- Captures and pinpoints the value of the language based on concrete usage; involving practitioners as they use practices and methods in real life scenarios
- Makes it possible to formally define the provide value in terms of functions like:
    - o “Provide guidance in terms of relevant activities to perform at a particular point in time, for a particular project”
    - o “Evaluate whether a team is competent enough to execute a practice”
    - o “Evaluate the current state of a project”

[10] "Process described with the SPEM 2.0 meta-model can be enacted in different ways. The two most common ways of enactment are: Mapping the processes into Project Plans and enacting these with project planning and enactment systems such as IBM Rational Portfolio Manager or Microsoft Project (Section 16.1). Mapping the process to a business flow or execution language and then executing this representation of the processes using a workflow flow engine such as a BPEL-based workflow engine (Section 16.2)." [SPEM2, Section 16, Page 147]
[11] C. Gonzalez-Perez and B. Henderson-Sellers, "Metamodelling for Software Engineering", John Wiley & Sons, Ltd, 2008, ISBN 978-0-470-03036-3.
[12] ISO/IEC, "Software Engineering – Metamodel for Development Methodologies", International Organization for Standardisation (ISO), ISO/IEC 24744, 15 February 2007.

    - "Identify relevant objectives when taking the next step in a project"
    - "Understand what practices are relevant to a project"
    - and much more
- Is used as a built-in verification mechanism to ensure that all language elements are purposeful
- Is used to state requirements and drive the development of the language itself, to make sure it provides the right values
- Serves as a formal proof that the language fulfills the features required by the FACESEM standard

B.2.1.4 Wider market appeal

One of the original rationale behind the SPEM 2.0 development was low uptake of SPEM 1.x[13]. We argue that this has been the same situation for SPEM 2.0. The main tools supporting SPEM 2.0 are:

- IBM Rational Method Composer (RMC), http://www-01.ibm.com/software/awdtools/rmc/
- Eclipse Process Framework (EPF), http://www.eclipse.org/epf/
- IRIS Process Author (PA), http://www.osellus.com/IRIS-PA

Ideally for OMG to claim that SPEM 2.0 is a successful standard there should have been more market uptake. A new simple and easy-to-use standard is needed that targets the practitioners foremost and the method engineer secondly.

If Essence is subsumed into a new version of SPEM (SPEM 3.0) it will never have mass appeal or impact. Our vision of how Essence will be used, and the value it will bring, is substantially different from the way in which SPEM is currently used. If Essence becomes part of a new SPEM standard, it will be hard to explain this in a way that makes sense, as it will be saying "there is part of the new SPEM standard that is intended to be used in a completely different way from the rest of SPEM".

B.2.2 What are your main differentiators?

The Essence language is based on a vision that gives it a clear differentiation from earlier work in defining meta-models for software engineering processes. We discuss these differentiators under three headings:

- Underpinning Values
- Support for Enactment
- Ease of Learning and Use

All of these relate to the prime aims, that Essence provides value to practitioners, and that it is easy to understand, adopt and use in the context of wide adoption.

B.2.2.1 Underpinning Values

A set of core values have driven the Essence work, and these have shaped and directed the Essence language development. These values are:

a. *What helps the least experienced developers before what helps the experts*. The least experienced do not need to bother with more advanced features of the Essence approach. This is motivated by the understanding that, among the many millions of developers in the world, a high proportion are not interested in 'method stuff'.

b. *What helps the practitioners before what helps the process engineers*. This is motivated by our conviction that process engineers will have to stand on what practitioners' need, and they will have to work from there – and

[13] "SPEM 1.x saw low uptake. Since its issuance, few implementations have been released and it has not been recognized by industry analysts who also failed to acknowledge its relevance to the methodology and process tools market. There have been a number of low-profile or casual adopters of the specification as well as few commercial implementations. It is suspected that ease of adoption has been an issue, and some of the SPEM 1.x semantics were ambiguous and hard to understand by adopters and hence not used in their practices." [SPEM2, Section 6.1, Page 8]

they do. “Practitioners are kings; process engineers are knights serving the kings”. Of course, we must support the experts and process engineers as well, but not by having the practitioners to pay a price.

c. *Intuitive, concrete graphical syntax before formal semantics*. Now we need to speak in a language easily understandable by the millions of developers who care about quickly being able to read and use the language. Of course, we have formally defined the semantics, but we have given extreme attention to syntax.

d. *Method use before method definition*. In the past similar initiatives have only paid interest to method definition, namely how to capture methods. They have not focused on how to support the use of a method while actually working in a software endeavor. Thus the methods became shelf-ware, not relevant for the developers running software development. Instead, the Essence approach supports the developers so they themselves can take control of their method and allow the method to evolve as their endeavor progresses. Of course, we can also define methods, but most importantly we have made methods useful while you actually work in real endeavors.

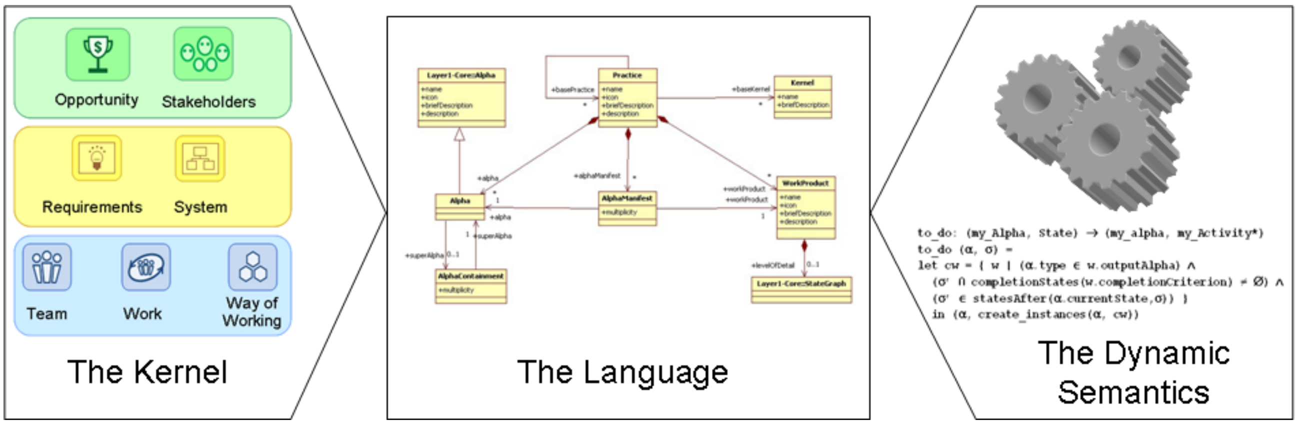
B.2.2.2 Support for Enactment

In the past, the primary aim of process description languages has been to achieve the necessary flexibility and expressive power required by process engineers in authoring or describing processes. The focus of Essence is different, as the primary focus is on delivering value to practitioners in undertaking a software development endeavror. I Essence this value comes from:

- The Kernel, which provides the means to track and understand the state and health of an endeavor along all its important dimensions. Practices and Methods are described in the Essence Language using the concepts of the Kernal (Alphas and Activity Spaces) and relate back to them.
- The Dynamic Semantics, which allows practitioners to interact with an Essence model so that the model helps guide and assess progress. The Dynamic Semantics is articulated in a formal language whose vocabulary is provided by the abstract syntax of the Essence Language.

A critical success factor for Essence is that the form and scope of the process language is precisely aligned to both the Kernel and the Dynamic Semantics, as shown in schematically Figure 56 – The Essence Language. The way these three interact and support each other is unique to the Essence vision. In particular:

- At its heart, the abstract syntax has the set of features (Alphas, Alpha States and Activity Spaces) that are used to define the Kernel. The full language uses and connects to these.
- The choice of features and relationships in the abstract syntax has been engineered to support a set of functions (the Dynamic Syntax) that will provide comprehensive support to practitioners during a project, helping to plan resourcing, guide the way forward and track the state and health of the project.
- The language is, and must be, kept lean. The scope of the abstract syntax is set by the need to align to the Kernel and support the Dynamic Semantics. In particular it is not the function of the Essence language to describe the process details of particular practices, as this is the realm of the practice process definitions which are outside of Essence. This relieves Essence of the need to support complete expressive power in process definition. It is the aim of achieving this full expressive power that makes conventional process definition languages large and difficult to assimilate.



```
to_do: (my_Alpha, State) → (my_alpha, my_Activity*)
to_do (α, σ) =
let cw = { w | (α.type ∈ w.outputAlpha) ∧
  (σ' ∩ completionStates(w.completionCriterion) ≠ ∅) ∧
  (σ' ∈ statesAfter(α.currentState,σ)) }
  in (α, create_instances(α, cw))
```

Figure 56 – The Essence Language

B.2.2.3 Ease of Learning and Use

The ambition of penetrating the mass market of practitioners at all levels of experience and ability, across all sizes of organization makes ease of learning understanding and use an imperative. These considerations have therefore been paramount in thinking about how the language is structured and how methods and practices expressed in the language are presented. Three aspects of the language are central to this.

1. The first is that users should be able to learn and adopt the language incrementally. To this end, the language is structured in **Layers**. The idea of this is that users may explore, understand and adopt the language incrementally as the layers represent self-contained and coherent subsets. This makes the learning curve shallower and reduces barriers to adoption by making it possible to gain value without needing to make a high investment.
2. The second is the emphasis put on a **Graphical Syntax** and related presentational mechanisms, such as cards. The idea is to provide a medium whereby practitionerss can access material in the Essence model in a way that easy to learn and remember, without the need to become fluent in any formal modeling language. The icons of the graphical syntax provide cues which enable people to orient themselves in the material, and recognize and interpret information easily. This accelerates adoption and use.
3. The third aspect of the languiuage is **Views** which provide a means of specifying selections and configurations elements from a model tailored to the needs of different types of users. The specification of views is made according to the job roles and preferences of users making it possible to customize the way material presented to match a context, so that the presentation is focused and the user or not distracted or overwhelmed by detail that is not relevant to the task or concern at hand.

While other process modeling formalisms have (or allow) similar constructs they tend to be positioned as optional extras, ranking below the abstract syntax in importance. Because the vision for Essence has wide and easy adoption by practitioners who do have any significant expertise in process modeling or formal modeling languages, these constructs have primary rank in Essence. This has guided the development of the Essence abstract syntax, which has been engineered to support and integrate with these constructs. The primary rank of these constructs is also a differentiator of the Essence language.

B.2.3 SPEM 2.0 metamodel reuse

As argued above the architecture of the Essence language is fundamentally different from the SPEM 2.0 architecture, so reusing elements from SPEM would not be compatible. However, we do see some areas where there are some similar concepts. In the table below we give an explanation on how the Essence concepts differs similar concepts in SPEM 2.0.

Table 29 – SPEM 2.0 metamodel reuse

Essence language construct	**Corresponding SPEM construct**	**Reason (Discussion)**

Activity	n/a	An activity defines one or more kinds of work items and gives guidance on how to perform these. Tasks (or work items as Essence prefers to call them) are not instances of guidance or definitions provided by the method. Because of this we prefer not to use the word Task in the method space. Work Items are an Alpha and can be created with the help of the Activity descriptions in the Method. It may take one or many work items to complete the work defined by an Activity. Alternatively one work item may complete the work defined by multiple Activities. **Work Around: Task Definition / TaskDescriptor –** The closest analogue of the Essence Activity in SPEM is the Task Definition / Task Descriptor. This would require the ability to handle state based completion criteria as well as output Alphas and Work Products. **Additional Notes:** Activities in Essence are non-nestable, and only support a predecessor (a finish-to-finish) relationship to other activities. To document some practices some of the other similar relationships (such as finish to start, start to start) may be required. All project planning tools have breakdown structures. Many, if not most, projects organize assigned tasks/work items in hierarchical structures. The creation of these hierarchies should be considered as part of enactment. Allowing instances of Activity Spaces and Activities as well as Work Items would enable this.
Activity.approach	n/a	There are usually many ways to complete an activity and these often change over time. By allowing the approaches to be defined separately from the more goal-based activities we can produce a more robust, extensible, tailorable and usable set of practices. **Work Around: Document Approaches as (Activity) Guidance:** Today in SPEM we do this by putting approaches into guidelines and attaching them to a task or other kinds of method elements. There are some cases where a task has a specific series of steps - there are no alternative approaches. Then there is the case where there are no steps - there are different approaches, and the practitioner can choose the approach. Note: There are many other guidelines that could be attached to an Activity such as staffing, timing, tool mentors, transformation algorithms and other hints and tips.
ActivityManifest	n/a	SPEM does not have separate relationship elements. Having a separate activity manifest gives the flexibility to associate an activity with different activity spaces. **Work Around: Use Contributing Elements:** SPEM has a workaround, however, which is to put optional relationships in contributing elements and group them in packages.
ActivitySpace	n/a	Activity Spaces are placeholders for activities, which are to be added by the practices. More than just empty slots Activity Spaces have clearly defined results and can be enacted directly when no appropriate Activities have been added. **Work Around: Use SPEM Activities or UMF Process Slots** In SPEM an activity is a breakdown element which supports the

		nesting and logical grouping of related process elements such as descriptor and sub-activities, thus forming breakdown structures. However, they are not useable on their own. Alternatively UMF extends SPEM with the concept of Process Slot. An activity space also defines a set of inputs and outputs, so it is more than just a UMF process slot.
Alpha	n/a	Alphas are a completely new concept that doesn't exist in SPEM. They are not a type of Work Product, they are things described by the Work products. They are not simply a Work Product Slot as they can be used without the addition if any Work products. **Work Around: Treat as a special kind of Work Product or Work Product Slot** SPEM Work Products can have state machines and so can mimic Alphas. An Alpha has different semantics to a Work Product, so they are not quite equivalent. UMF extends SPEM with the concept of Work Product Slot. An alpha has more semantics than work product slot, so they are not quite equivalent.
AlphaManifest	n/a	SPEM does not have separate relationship elements. Having a separate alpha manifest gives the flexibility to associate an alpha with different work products. **Work Around: Use Contributing Elements:** SPEM has a workaround, however, which is to put optional relationships in contributing elements and group them in packages.
AreaOfConcern	n/a	The kernel is divided into a number of discrete areas of concern. Each item belongs to one and only one area of concern. **Work Around: Use Context** UMF extends SPEM with the concept of Context. The semantics for a context is that elements within a context have no dependencies on elements in a separate context. In the UMF practices layered architecture, practices and elements used by practices belong to one and only one context. However, processes and configurations can cross contexts. UMF uses a different set of contexts than the Essence examples, so more discussion is needed to confirm that this mapping is correct.
Checkpoint		No equivalent in SPEM
Competency		No equivalent in SPEM.
CompetencyLevel	n/a	No equivalent in SPEM. Competency level is a useful concept. We may define some standard competency levels to differentiate between familiar with a subject area, knowledgeable of the subject area, applied the subject area, and master (able to teach others).
CompletionCriterion	n/a	Not in SPEM/UMF.

Kernel	n/a	No equivalent in SPEM.
Library	MethodLibrary	Essence and SPEM 2.0 are conceptually aligned.
Method	MethodConfiguration	Method configuration and method appear to be synonyms. Method configuration is a better term for one who is tailoring. Method is a better term for one who is using a configuration.
Pattern	n/a	An arrangement of the other method elements to define additional less constrained guidance such as Milestones, Planning Patterns, Team Structures, Team Roles, Job Descriptions, Measurements etc. **Work Around: Model as type of guidance**
Practice	Practice	The Practice concept in SPEM is only a specific type of Guidance. The concept of Practice has been extended in UMF.
Skill	Qualification	Qualification is more general than skill, as being certified is not a skill, and yet may be an important attribute to model.
SkillLevel	n/a	No equivalent in SPEM.
WorkProduct	WorkProductDefinition	Essence and SPEM 2.0 are conceptually aligned.

Annex C: Practice Examples

(Informative)

This annex provides working examples to demonstrate the use of the Kernel and Language to describe practices.

C.1 Practices

This section contains illustrative examples of the following:

- Scrum
- User Story
- Lifecycle examples

C.1.1 Scrum

This section illustrates the Essence approach by modeling the Scrum project management practice. The Scrum practice as documented here is for illustrative purposes only and explores how the Scrum practice may be mapped to the Essence Kernel and Language. It should not be interpreted as a definitive example of how Scrum should be represented. There may be multiple ways for different communities to represent Scrum.

C.1.1.1 Practice

The following Scrum concepts were identified from the Scrum guide [Schwaber and Sutherland 2011]:

- Scrum team (roles)
    - Product Owner
    - Development Team (of developers)
    - Scrum Master
- Scrum events
    - The Sprint
    - Sprint Planning Meeting
    - Daily Scrum
    - Sprint Review
    - Sprint Retrospective
- Scrum artifacts
    - Product Backlog
    - Sprint Backlog
    - Increment

Graphical syntax

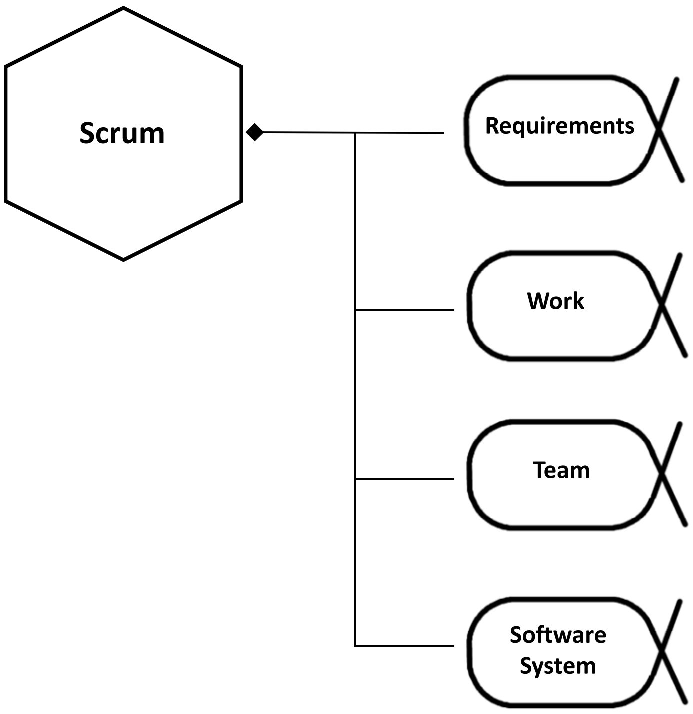


Figure 57 – Scrum practice

C.1.1.2 Alphas

C.1.1.2.1 Work

We extend the Work alpha for Scrum. The Work alpha is typically used for the duration of a development project that may cover a number of sprints. Thus we define a new sub-alpha called Sprint.

- "The heart of Scrum is a Sprint, a time-box of one month or less during which a “Done”, useable, and potentially releasable product Increment is created. Sprints have consistent durations throughout a development effort. A new Sprint starts immediately after the conclusion of the previous Sprint." [Schwaber and Sutherland 2011]

Graphical syntax

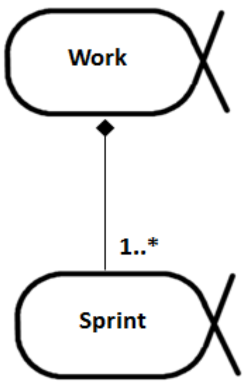


Figure 58 – Sprint sub-alpha of Work

The Sprint has its own state graph. Scrum comes with its own specific set of rules that should be defined as part of the practice, whereas the Work state machine and its associated checkpoints are more general.

Graphical syntax

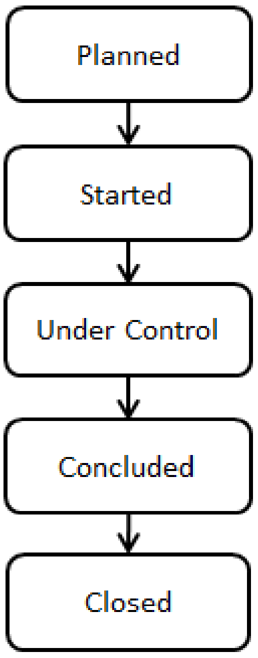


Figure 59 – The states of the Sprint sub-alpha

Textual syntax

```
alpha Work {
      contains 1..N Sprint
}

alpha Sprint {
      "The heart of Scrum is a Sprint, a time-box of one month or less during
which a "Done", useable, and potentially releasable product Increment is created.
Sprints have consistent durations throughout a development effort. A new Sprint
starts immediately after the conclusion of the previous Sprint.
```

```
      (...continues...)"

      has {
            state Planned {"The work has been requested and planned."
                  checks {
                        item c1 {"Sprint Planning Meeting is held."}
                        item c2 {"Product Owner presents ordered Product
                              Backlog items to the Development Team."}
                  }
            }
                        item c3 {"Development Team decides how it will build this
                              functionality into a "Done" product Increment
                              during the Sprint"}
                        item c4 {"Scrum Team crafts a Sprint Goal."}
                        item c5 {"Development Team defines a Sprint Backlog."
                  }
            state Started {"The work is proceeding."
                  checks {
                        item c1 {"Team is taking their work items from the Sprint
                              Backlog"}
                  }
            state Under Control {"The work is going well, risks are under
                        control, and productivity levels are sufficient to
                        achieve a satisfactory result."
                  checks {
                        item c1 {"Daily Scrum optimizes the probability that the
                              Development Team will meet the Sprint Goal."}
                        item c2 {"Every day, the Development Team should be able
                              to explain to the Product Owner and Scrum Master
                              how it intends to work together as a self-
                              organizing team to accomplish the goal and create
                              the anticipated increment in the remainder of the
                              Sprint."}
                  }
            }
            state Concluded {"The work to produce the results has been
                        concluded."
                  checks {
                        item c1 {"During the Sprint Review, the Scrum Team and
                              stakeholders collaborate about what was done in the
                              Sprint."}
                  }
            }
            state Closed {"All remaining housekeeping tasks have been completed
                        and the work has been officially closed."
                  checks {
                        item c1 {"A Sprint Review Meeting is held at the end of
                              the Sprint."}
                        item c2 {"The Sprint Retrospective occurs after the
                              Sprint Review and prior to the next Sprint Planning
                              Meeting."}
                  }
            }
      }

}
```

C.1.1.2.2 Team

The Scrum practice relates to the Team alpha. The Team alpha refers to the individuals working in the team, i.e. members that may be represented by a sub-alpha. Scrum defines a specific Scrum Team which consists of a Product Owner, the Development Team, and a Scrum Master.

- "The Scrum Team consists of a Product Owner, the Development Team, and a Scrum Master. Scrum Teams are self-organizing and cross-functional. Self-organizing teams choose how best to accomplish their work, rather than being directed by others outside the team. Cross-functional teams have all competencies needed to accomplish the work without depending on others not part of the team. The team model in Scrum is designed to optimize flexibility, creativity, and productivity." [Schwaber and Sutherland 2011]

Graphical syntax

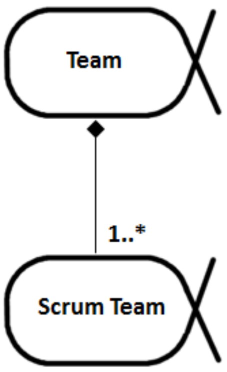


Figure 60 – Scrum Team

Scrum mandates that one sole person should take on the role of a Product Owner and another sole person should take on the role of the Scrum Master. These types of constraints could be added as checkpoints on the Team alpha itself, but another alternative would be to define a specific Scrum Team as a sub-alpha. The introduction of a specific sub-alpha would allow us to easier extend and scale the practice to Scrum of Scrums, including managing different types of teams not all following Scrum.

Graphical syntax

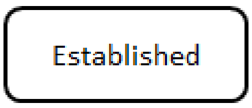


Figure 61 – The states of the Scrum Team sub-alpha

Textual syntax

```
alpha Team {
      contains 1 Scrum Team
}

alpha Scrum Team {
      "The Scrum Team consists of a Product Owner, the Development Team, and a
Scrum Master. Scrum Teams are self-organizing and cross-functional. Self-
organizing teams choose how best to accomplish their work, rather than being
directed by others outside the team. Cross-functional teams have all competencies
needed to accomplish the work without depending on others not part of the team.
The team model in Scrum is designed to optimize flexibility, creativity, and
productivity.
      (...continues...)"

      has {
            state Established {"Scrum Team is established."
                  checks {
                        item c1 {"The Product Owner is assigned."}
                        item c2 {"Developers are assigned to the Development
                              Team."}
                        item c3 {"The Scrum Master is assigned."}
                        }
                }
        }
}
```

C.1.1.3 Work Products

C.1.1.3.1 Product Backlog

The Product Backlog and Sprint Backlog are associated with the Requirements alpha.

- "The Product Backlog is an ordered list of everything that might be needed in the product and is the single source of requirements for any changes to be made to the product. The Product Owner is responsible for the Product Backlog, including its content, availability, and ordering." [Schwaber and Sutherland 2011]

Graphical syntax

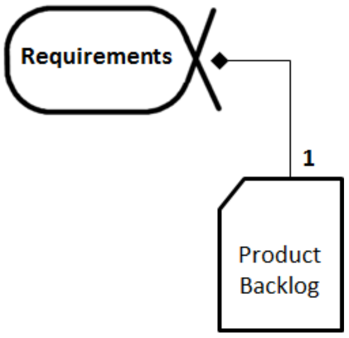


Figure 62 – Product Backlog

Textual syntax

```
workProduct Product Backlog {
        "The Product Backlog is an ordered list of everything that might be needed
in the product and is the single source of requirements for any changes to be
made to the product. The Product Owner is responsible for the Product Backlog,
including its content, availability, and ordering.
        (...continues...)"
}
```

C.1.1.3.2 Sprint Backlog

The Sprint Backlog is associated with the Sprint sub-alpha.

- "The Sprint Backlog is the set of Product Backlog items selected for the Sprint plus a plan for delivering the product Increment and realizing the Sprint Goal. The Sprint Backlog is a forecast by the Development Team about what functionality will be in the next Increment and the work needed to deliver that functionality." [Schwaber and Sutherland 2011]

Graphical syntax

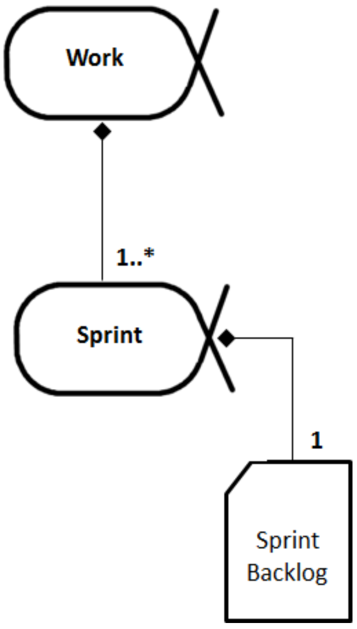


Figure 63 – Sprint Backlog

Textual syntax

```
workProduct Sprint Backlog {
      "The Sprint Backlog is the set of Product Backlog items selected for the
Sprint plus a plan for delivering the product Increment and realizing the Sprint
Goal. The Sprint Backlog is a forecast by the Development Team about what
functionality will be in the next Increment and the work needed to deliver that
functionality.
      (...continues...)"
}
```

C.1.1.3.3 Increment

The Increment is associated with the Software System alpha.

- "The Increment is the sum of all the Product Backlog items completed during a Sprint and all previous Sprints. At the end of a Sprint, the new Increment must be "Done," which means it must be in useable condition and meet the Scrum Team's Definition of "Done." It must be in useable condition regardless of whether the Product Owner decides to actually release it." [Schwaber and Sutherland 2011]

Graphical syntax

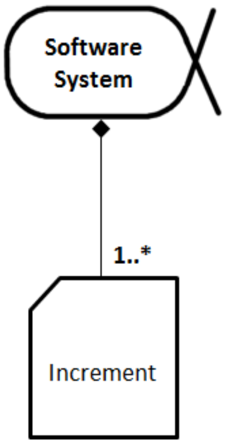


Figure 64 – Increment

Textual syntax

```
workProduct Increment {
      "The Increment is the sum of all the Product Backlog items completed during
a Sprint and all previous Sprints. At the end of a Sprint, the new Increment must
be "Done," which means it must be in useable condition and meet the Scrum Team's
Definition of "Done." It must be in useable condition regardless of whether the
Product Owner decides to actually release it.
      (...continues...)"
}
```

C.1.1.4 Activities

The identified Scrum events may be mapped to corresponding activities. The concept of sprint however describes an iteration that we will map to a sub-alpha of Work. This gives us the following activities:

- Sprint Planning Meeting
- Daily Scrum
- Sprint Review
- Sprint Retrospective

Graphical Syntax

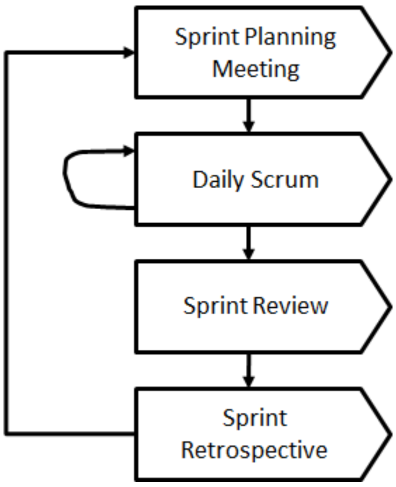


Figure 65 – Scrum activities

C.1.1.4.1 Sprint Planning Meeting

The Sprint Planning Meeting is associated with the Prepare to do the Work activity space.

- "The work to be performed in the Sprint is planned at the Sprint Planning Meeting. This plan is created by the collaborative work of the entire Scrum Team. The Sprint Planning Meeting is time-boxed to eight hours for a one-month Sprint. For shorter Sprints, the event is proportionately shorter. For example, two-week Sprints have four-hour Sprint Planning Meetings." [Schwaber and Sutherland 2011]

Graphical syntax

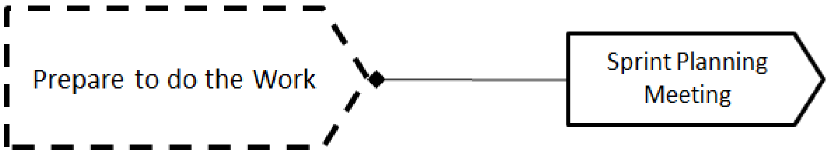


Figure 66 – Sprint Planning Meeting

C.1.1.4.2 Daily Scrum

The Daily Scrum is associated with the

- "The Daily Scrum is a 15-minute time-boxed event for the Development Team to synchronize activities and create a plan for the next 24 hours. This is done by inspecting the work since the last Daily Scrum and forecasting the work that could be done before the next one." [Schwaber and Sutherland 2011]

Graphical syntax

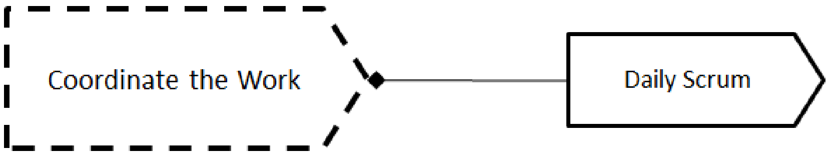


Figure 67 – Daily Scrum

C.1.1.4.3 Sprint Review

The Sprint Review is associated with the Track Progress activity space.

- "A Sprint Review is held at the end of the Sprint to inspect the Increment and adapt the Product Backlog if needed. During the Sprint Review, the Scrum Team and stakeholders collaborate about what was done in the Sprint. Based on that and any changes to the Product Backlog during the Sprint, attendees collaborate on the next things that could be done. This is an informal meeting, and the presentation of the Increment is intended to elicit feedback and foster collaboration." [Schwaber and Sutherland 2011]

Graphical syntax

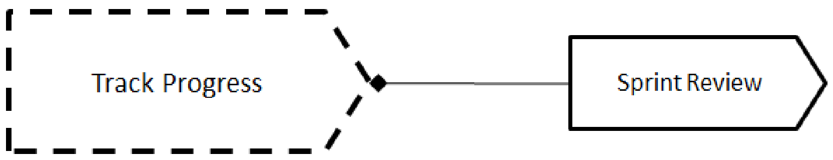


Figure 68 – Sprint Review

C.1.1.4.4 Sprint Retrospective

The Sprint Retrospective is associated with the Support the Team activity space.

- "The Sprint Retrospective is an opportunity for the Scrum Team to inspect itself and create a plan for improvements to be enacted during the next Sprint. The Sprint Retrospective occurs after the Sprint Review and prior to the next Sprint Planning Meeting. This is a three-hour time-boxed meeting for one-month Sprints. Proportionately less time is allocated for shorter Sprints." [Schwaber and Sutherland 2011]

Graphical syntax

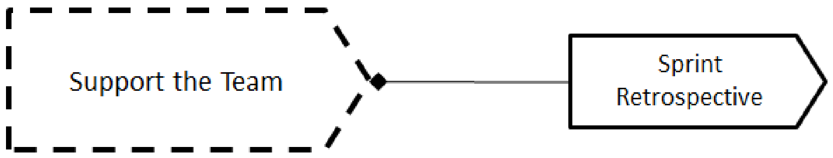


Figure 69 – Sprint Retrospective

C.1.1.5 Roles

Roles can be described as patterns:

- Product Owner
- Development Team (of developers)
- Scrum Master

C.1.1.5.1 Product Owner

Textual syntax

```
role Product Owner {
      "The Product Owner is responsible for maximizing the value of the product
and the work of the Development Team. How this is done may vary widely across
organizations, Scrum Teams, and individuals.
      (...continues...)"

}
```

C.1.1.5.2 Development Team

Textual syntax

```
role Development Team {
      "The Development Team consists of professionals who do the work of
delivering a potentially releasable Increment of "Done" product at the end of
each Sprint. Only members of the Development Team create the Increment.
      (...continues...)"

}
```

C.1.1.5.3 Scrum Master

Textual syntax

```
role Scrum Master {
      "The Scrum Master is responsible for ensuring Scrum is understood and
enacted. Scrum Masters do this by ensuring that the Scrum Team adheres to Scrum
theory, practices, and rules. The Scrum Master is a servant-leader for the Scrum
Team.
      The Scrum Master helps those outside the Scrum Team understand which of
their interactions with the Scrum Team are helpful and which aren't. The Scrum
Master helps everyone change these interactions to maximize the value created by
the Scrum Team.
      (...continues...)"

}
```

C.1.2 User Story

C.1.2.1 Practice

Graphical syntax

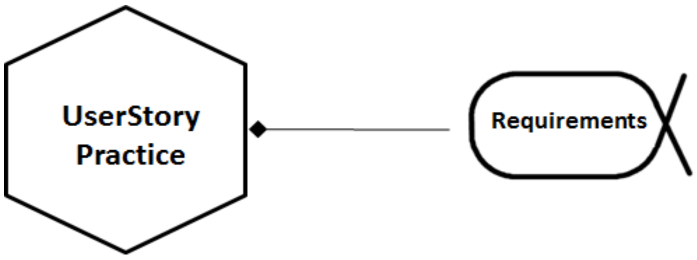


Figure 70 – User Story practice

C.1.2.2 Work Products

C.1.2.2.1 User Story

A User Story can be seen as a requirements item sub-alpha of Requirements that you want to monitor the state of. This requirements item is described by a User Story Card.

Graphical syntax

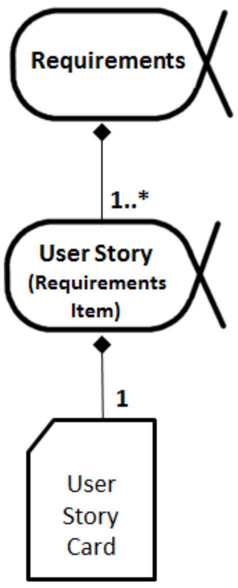


Figure 71 – User Story

Textual syntax

```
alpha User Story {
      "A User Story is an Independent, Negotiable, Valuable, Estimatable, Small,
Testable requirement (INVEST)"

      has {
            state Described {"The User Story is described."}
                  checks {
                        item c1 {"User Story is described by the customer."}
                        item c2 {"User Story is prioritized by the customer."}
                  }
            state Understood {"The User Story has been analyzed by the Team"}
                  checks {
                        item c1 {"The User Story has been broken down into tasks
                              by the developers."}
                        item c2 {"The User Story has been estimated by the
                              developers."}
                  }
            }
            state Implemented {"The User Story has been implemented."}
                  checks {
                        item c1 {"The User Story has been implemented."}
                        item c2 {"The implementation has been tested."}
                  }
            }
            state Fulfilled {"The User Story has been fulfilled."}
                  checks {
                        item c1 {"The Customer has approved the implementation."}
                  }
            }
      }
}


workProduct User Story Card {
      "The User Story Card contains the description of the User Story. User
```

```
stories generally follow the following template:
    "As a <role>, I want <goal/desire> so that <benefit>"
    "As a <role>, I want <goal/desire>""
}
```

C.1.2.3 Activities

C.1.2.3.1 Write User Story

Graphical syntax

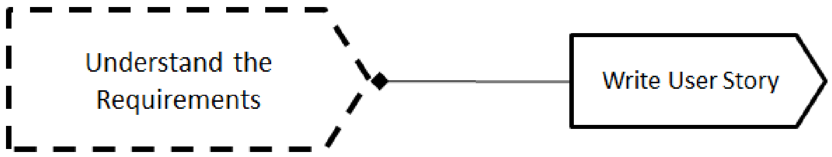


Figure 72 – Write User Story

C.1.2.3.2 Prioritize User Story

Graphical syntax

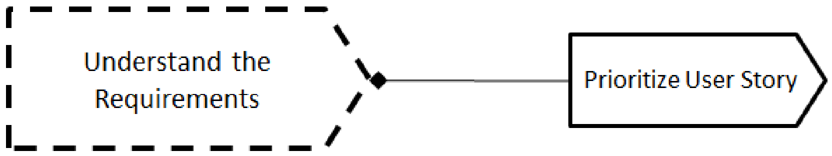


Figure 73 – Prioritize User Story

C.1.2.3.3 Estimate User Story

Graphical syntax

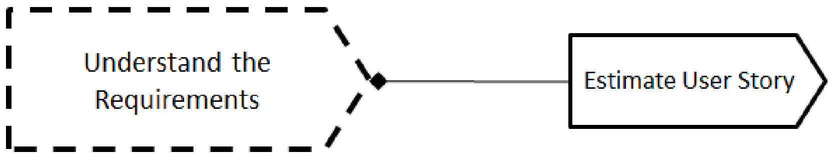


Figure 74 – Estimate User Story

C.1.3 Multi-phase Waterfall

In some practices in common use, there are multiple phases of Requirements Definition, each adding more detail.

- Multiple Requirements and Design Activities normally flow top down.
- Multi-phase Testing Activities normally flow bottom up.

This practice example is closely related to the so-called V-Model for software process engineering http://www.the-software-experts.de/e_dta-sw-process.htm .

- Actual Flow of Activities associated with each phase can be quiet complex in a real project.
- Requirements alpha specializations are needed to model requirement documents from each phase.

C.1.3.1 Activities

The general form of the V-model of Activities for the Muti-phase Waterfall practice is shown in Figure A.x

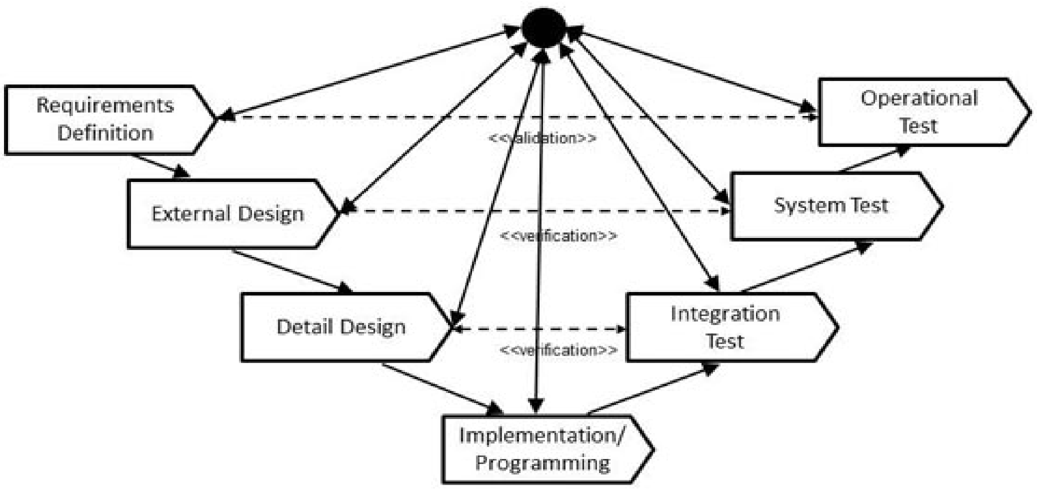


Figure 75 – Multi-phase Waterfall Practice Activities Flow

Figure 75 shows an example of "V-Model" for Multi-phase Waterfall Practice. Each Test Activity verifies/validates work products of one Requirements/Design Activity. Normal progression flows from left to right. If defects are detected or rewind is required, process flows back to appropriate point thru the depicted virtual node

C.1.3.1.1 Requirements Definition Phase

Description	**Major work products**
• Confirm the systematization requirements to define functional (system functions, data, interface) and non-functional requirements • Define and outline design of the system and examine the feasibility of the system. • Develop a project plan and establish management measurers to carry out the project.	• Use cases & Scenario • Business flows • Business rules • Data model (High-level) • Execution environment prescription (as Non-functional requirement) • Business operational test spec.

C.1.3.1.2 External Design Phase

Description	**Major work products**
• Design high-level specifications for end users such as system functions, data, interfaces, screens and print-form • Design the system architecture and operation measures. • Investigate the current assets (applications, system configuration, data) to determine which resources should be transferred to the new system. • Develop a total test plan.	• Application architecture spec. • Conceptual data model • Screen Design spec. • Printing-form design spec. • Process structure spec. • Interface design spec. • Message & Code design • Detail Non-functional requirements

	• System test specification.

C.1.3.1.3 Detailed Design Phase

Description	**Major work products**
• Design the system internal structure (ex. program unit, database physical structure) and interfaces between programs based on the outline specifications. • Design an operation management system, security system, and methods for transition of the current resources.	• Software component/module spec. • Physical Database schema specification • Detail screen spec.(screen constituent) • Performance design • Security design • Integration test spec

C.1.3.1.4 Implementation/Programming Phase

Description	**Major work products**
• Define program structure and design program logic • Develop and complete programs based on the program design • Implement the database based on the data model. • Test each program module individually to verify correctness and quality.	• Source code • Middleware/Hardware configuration specification. • Database definition Language

C.1.3.1.5 Integration Test Phase

Description	**Major work products**
• Test each process by integrating programs to verify the application. • Test interfaces between all processes • Confirm interfaces between external systems	• Result reports for Integration test spec.

C.1.3.1.6 System Test Phase

Description	**Major work products**
• Test the business system functions on the actual machines. • Test the entire system by evaluating system performance, reliability, operability, security, etc.	• Result reports for System test spec.

C.1.3.1.7 Operational Test Phase

Description	**Major work products**

• Test business operations in the real environment with actual machines and real data. This test is performed by end users. • Validate the business functions, performance, reliability, operability, and security. • Make decision to transit from test operation to real operation, and perform a transition.	• Result reports for business operational test.

C.1.3.2 Alpha Extensions for Multi-Phase Waterfall Requirements

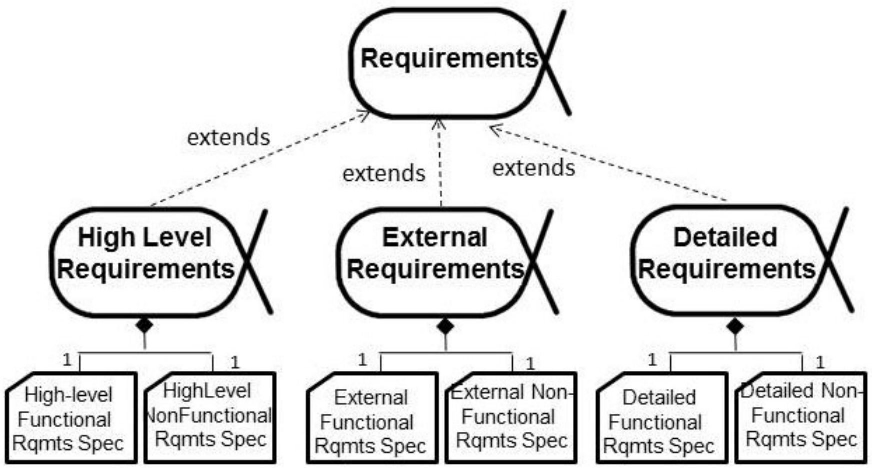

Requirements
extends
extends
extends
High Level Requirements
External Requirements
Detailed Requirements
1
1
1
1
1
1
High-level Functional Rqmts Spec
HighLevel NonFunctional Rqmts Spec
External Functional Rqmts Spec
External Non-Functional Rqmts Spec
Detailed Functional Rqmts Spec
Detailed Non-Functional Rqmts Spec

Figure 76 – Multi-phase Waterfall Requirements Alpha Extensions and Requirements Spec Work Products

High Level Requirements Specs (Functional and Non-Functional) are produced by Requirements Definition Activity.

External Requirements Specs (Functional and Non-Functional) are produced by External Design Activity.

Detailed Requirements Specs (Functional and Non-Functional) are produced by Detailed Design Activity.

Each Requirements extension Alpha has:

- Its own state values, the same as specified for the Requirements Alpha;
    - Conceived; Bounded; Coherent; Described; Addressed; Fulfilled
- Functional and Non-Functional Requirements Spec Work Products
    - each having Sub-Alphas for every requirement Item, with their own state values (the same as specified for the Requirments Item Sub-Alpha Kernel Extension
    - Requirements Alpha Extension state transitions conditional on Requirements Item Sub-alpha state transitions

C.1.3.3 Lifecycle Diagram for Multi-Phase Waterfall Requirements Alpha Extensions

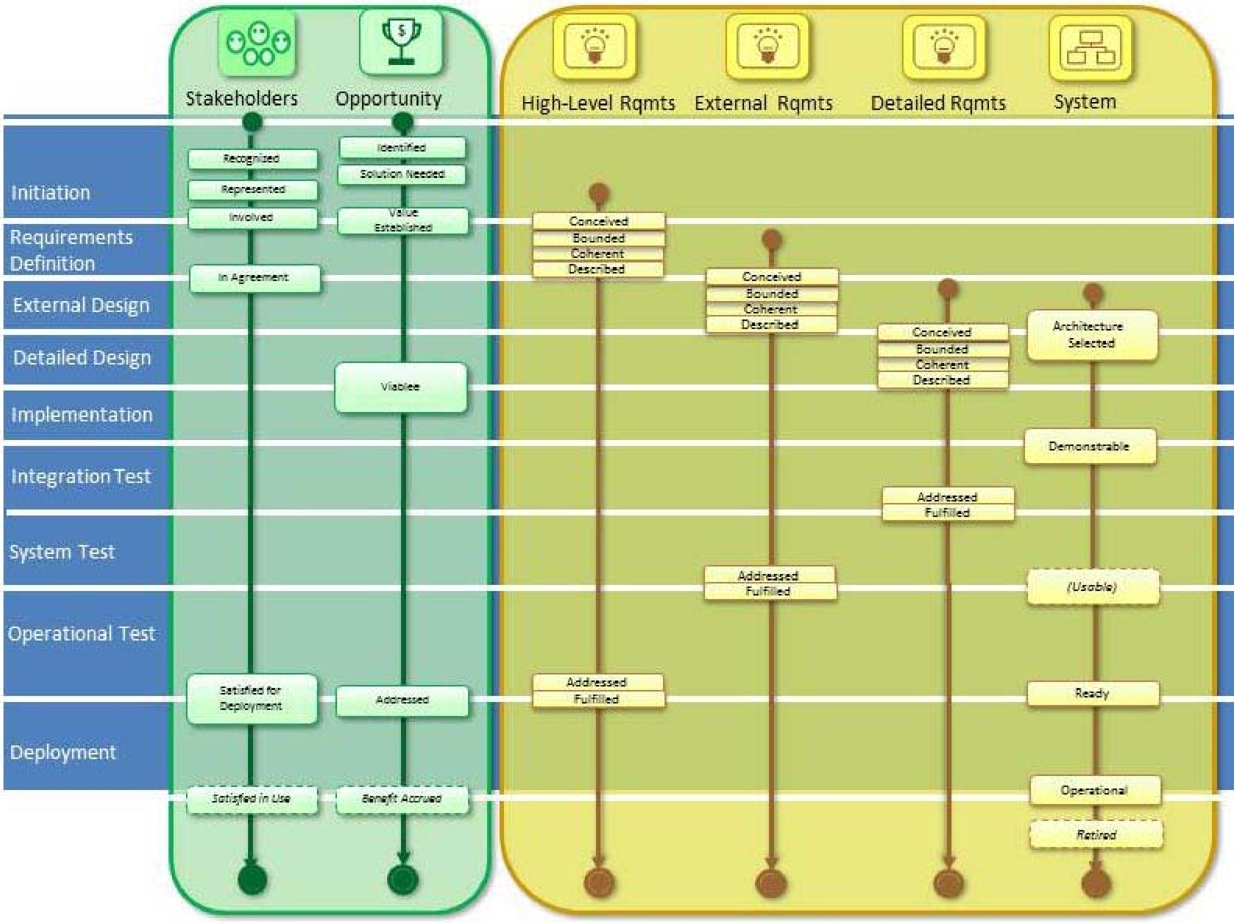


Figure 77 – Lifecycle Diagram for Multi-Phase Waterfall Requirements Alpha Extensions

C.1.3.4 Extensions of Requirements Item Alpha for Tracking Individual Multi-Phase Waterfall Requirement Items

If a project needs to track to state of each individual requirements item, the following Sub-Alpha extensions of the Requirements Item kernel Extension Sub-alpha can be employed.

The individual Requirement Work products are part of their respective Requirements Spec (Functional or Non-Functional) associated with their parent Requirements Alpha Extension.

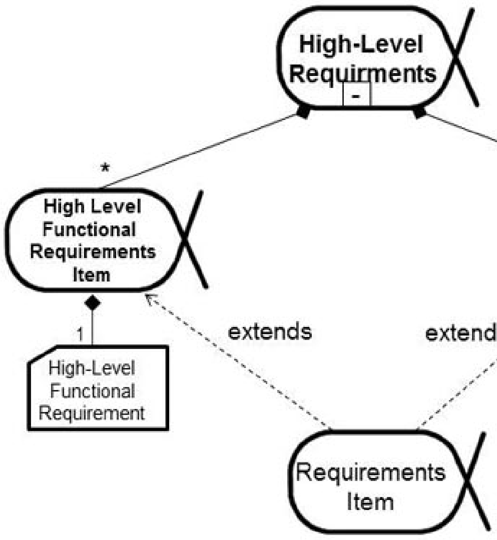

High-Level Requirments
-
*
*
High Level Functional Requirements Item
High Level Non-Functional Requirements Item
extends
extends
1
1
High-Level Functional Requirement
High-Level Non-Functional Requirement
Requirements Item

Figure 78 – High-Level Requirements Sub-Alphas and Requirement Work Products

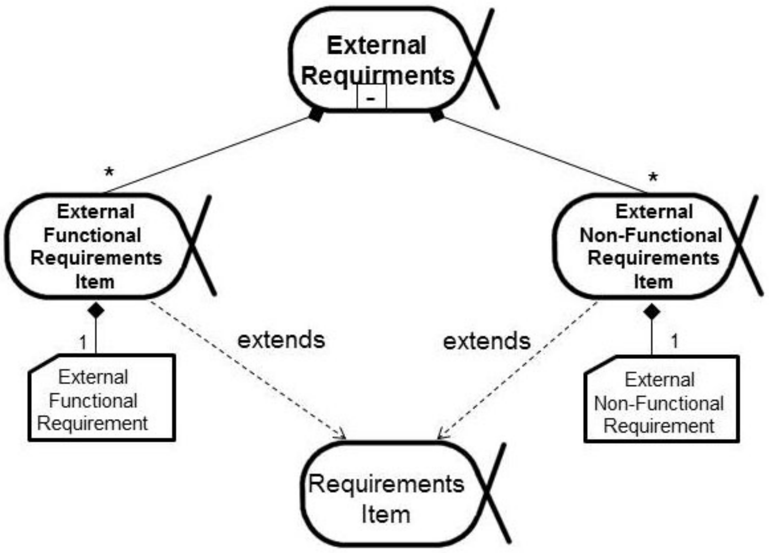

External Requirments
-
*
*
External Functional Requirements Item
External Non-Functional Requirements Item
extends
extends
1
1
External Functional Requirement
External Non-Functional Requirement
Requirements Item

Figure 79 – External Requirements Sub-Alphas and Requirement Work Products

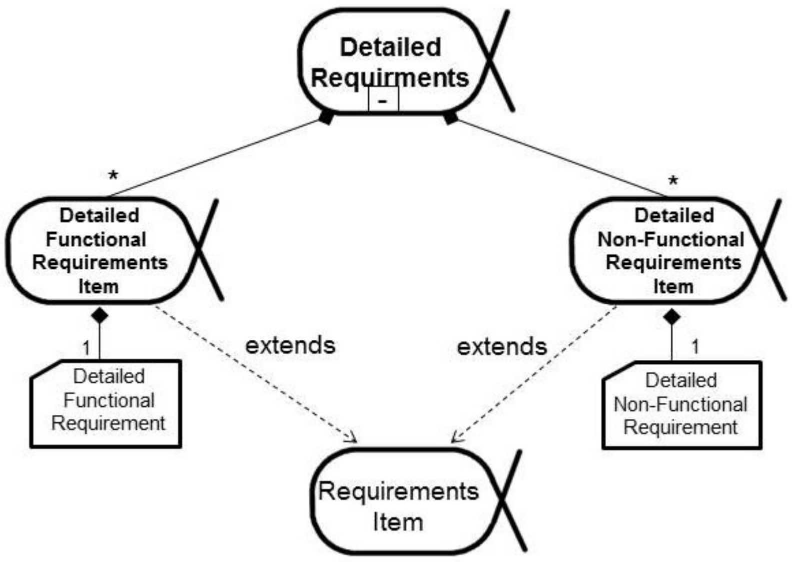

Detailed Requirments
-
*
*
Detailed Functional Requirements Item
Detailed Non-Functional Requirements Item
1
1
extends
extends
Detailed Functional Requirement
Detailed Non-Functional Requirement
Requirements Item

Figure 80 – Detailed Requirements Sub-Alphas and Requirement Work Products

C.1.4 Lifecycle Examples

The Essence Kernel enables practices to define lifecycles by sequencing a number of patterns, one for each phase and/or milestone in the lifecycle.

This section provides illustrations of a number of typical software engineering lifecycles:

- A Unified Process lifecycle
- A waterfall lifecycle
- A set of complementary application development lifecycles
- A funding and decision making lifecycle

When reading these sections one should bear in mind that a lifecycle practice can do more than just arrange the alpha states, it can also add items to the checklists, activities to formally review the milestones and any other planning or review guidance it sees fit.

All the lifecycles are illustrated using the template shown in Figure 81.

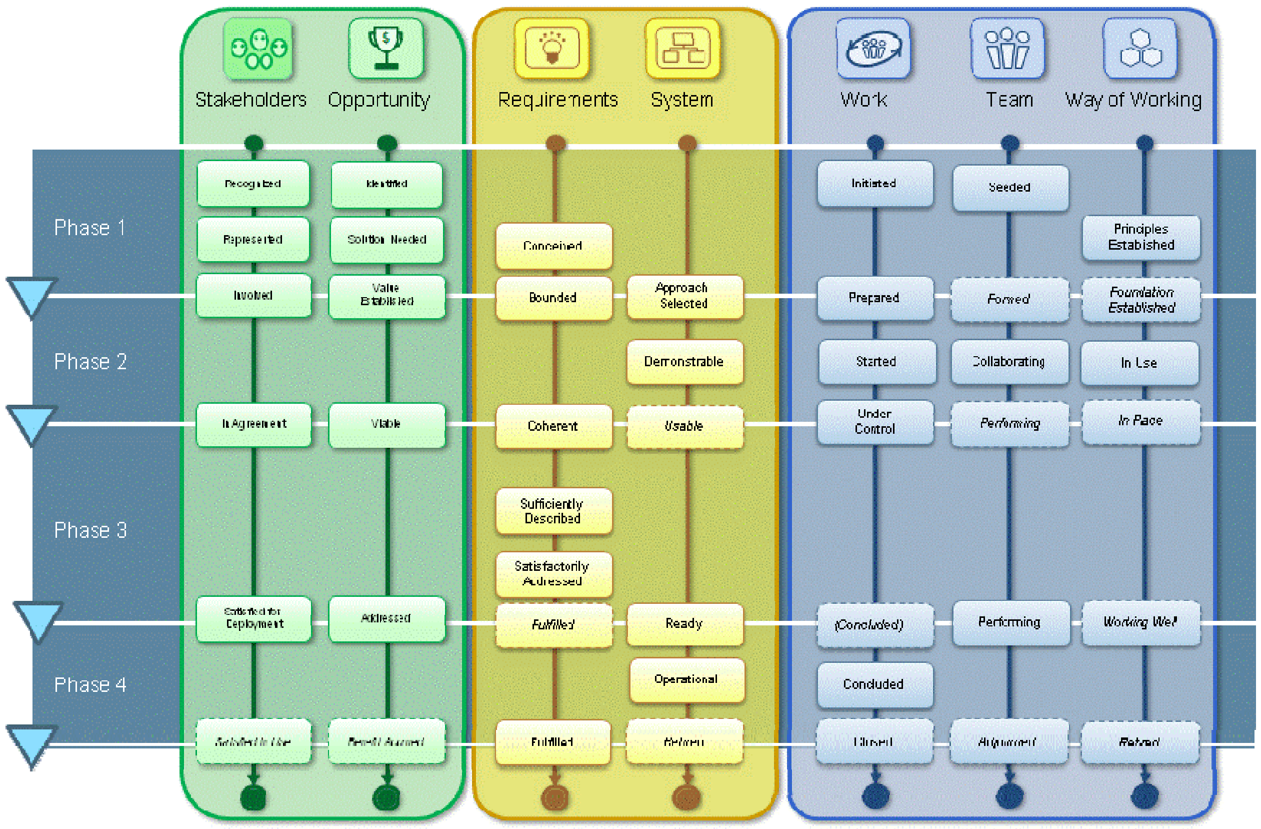


Figure 81 – Lifecycle template

Each Kernel Alpha and its states are shown in a vertical column with their creation at the top and their destruction at the bottom. Milestones are shown as a vertical bar across the grid starting with an inverted triangle to represent the milestone and continuing with a white line over which are shown the states to be achieved to successfully pass the milestone. Where achieving a state is either recommended or optional the state is shown with a dashed outline and italicized text.

C.1.4.1 The Unified Process Lifecycle

An illustration of the Unified Process Lifecycle is shown in Figure 82. In the Unified Process Lifecycle there are four phases: Inception, Elaboration, Construction and Transition. Each of these ends in a distinct milestone: Lifecycle Objectives Milestone, Lifecycle Architecture Milestone, Initial Operational Capability, Project End. In Figure 82, the milestones are represented by the blue inverted triangles but the names are suppressed to keep things simple.

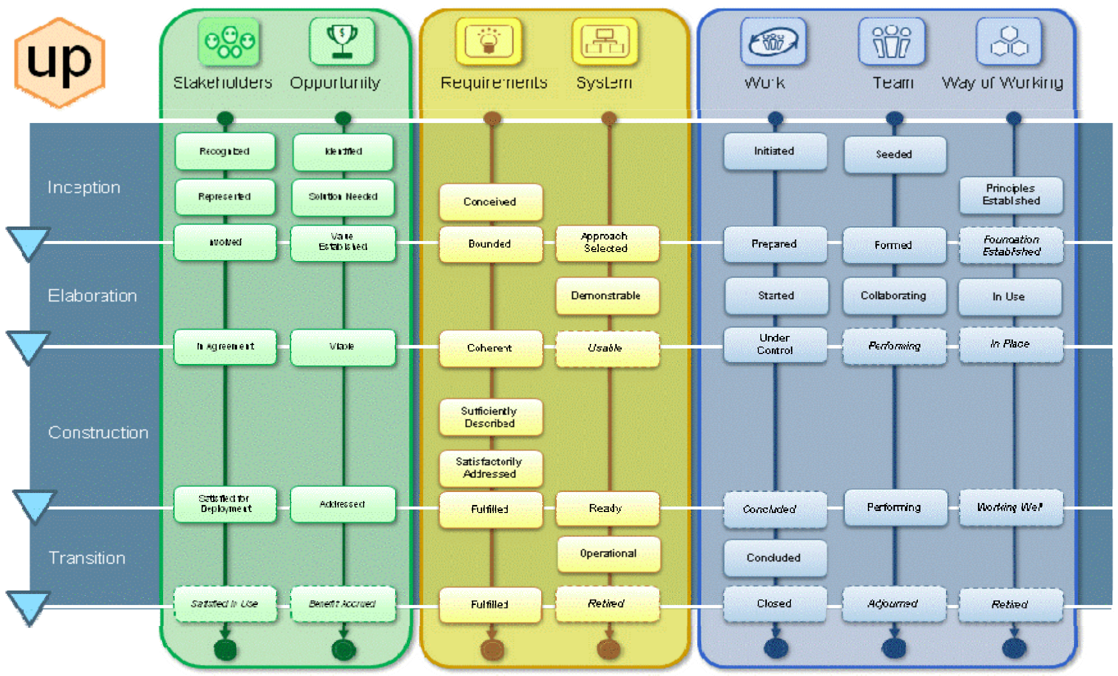


Figure 82 – The Unified Process lifecycle

C.1.4.2 The Waterfall Lifecycle

An illustration of a Waterfall Lifecycle is shown in Figure 83. In this case there are six phases: Initiation, Requirements, Analysis and Design, Implementation, Testing, and Deployment. Each of these ends in a distinct milestone, which in this case are not named.

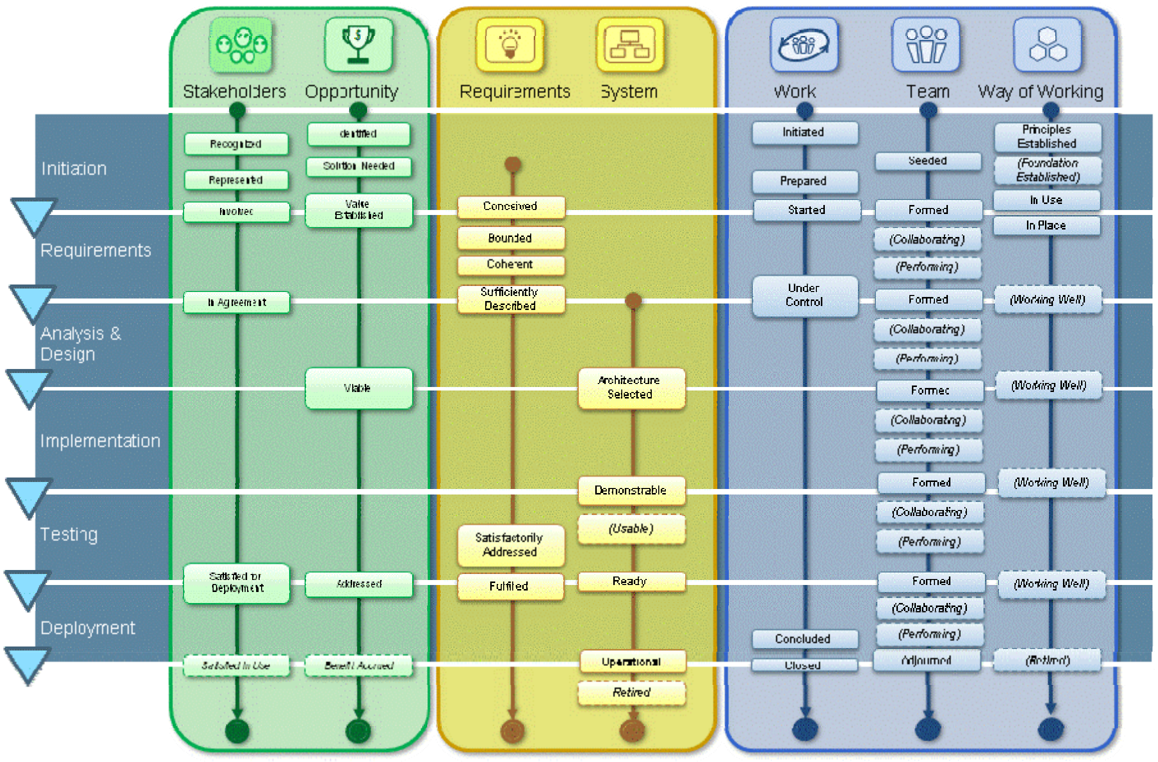


Figure 83 – A Waterfall lifecycle

Of most interest here are:

1. The fact that there is no work on the system itself until the Analysis and Design Phase at the earliest.
2. Different team formations are used for each phase and so the state of the team keeps getting set back to *formed* with the hope that the new team will be *collaborating and performing* before the end of its phase.
3. The Requirements are *sufficiently described* by the end of the Requirements Phase and then not progressed again until the Testing Phase.

C.1.4.3 A set of complementary application development lifecycles

The Kernel can be used in much more subtle ways than in the previous two examples. It is not un-common for application development organizations to need multiple lifecycles to cope with the different types and styles of development that they undertake. Figure 84 shows four complementary lifecycle models illustrating the typical demands made upon an application development organization. This example is taken from a real software development organization and uses their names for the four lifecycle models.

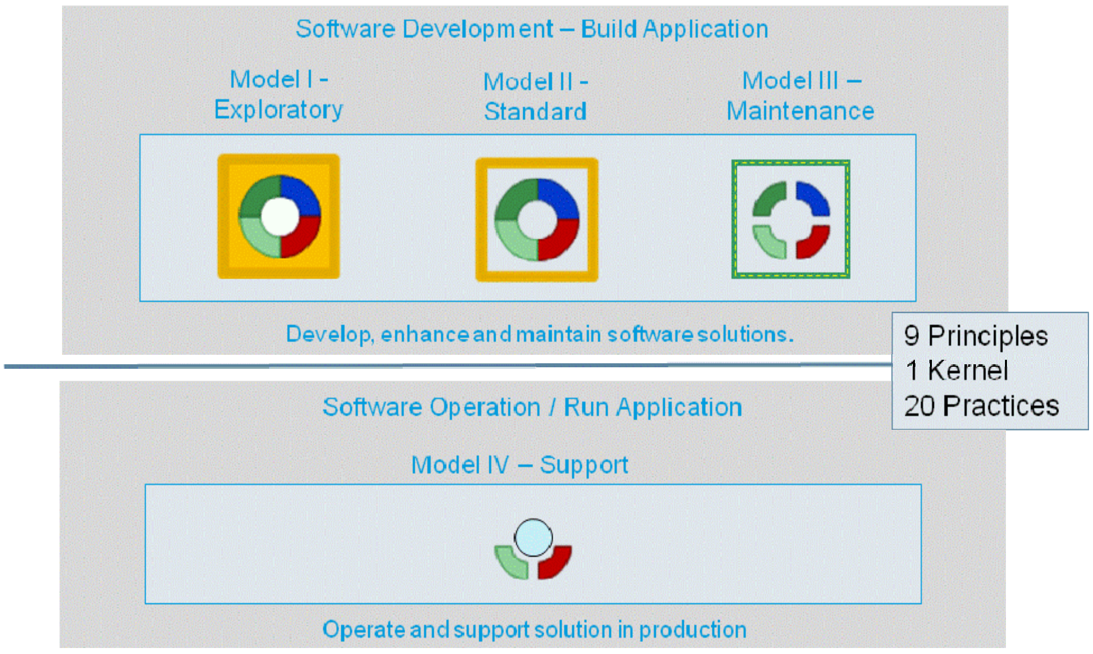

Software Development – Build Application
Model I - Exploratory
Model II - Standard
Model III – Maintenance
Develop, enhance and maintain software solutions.
9 Principles
1 Kernel
20 Practices
Software Operation / Run Application
Model IV – Support
Operate and support solution in production

Figure 84 – Different types of development need different methods and lifecycles

Each lifecycle model is supported by a method, each of which is built on the same kernel, many of which share the same practices, and each of which has its own lifecycle. The four lifecycles are shown in Figure 85. Here the four lifecycles are deliberately shown in a single diagram to make the differences in the arrangements of the states easily visible. Unfortunately this makes the wording very hard to read. If you are interested in the details of the figures they are repeated at a larger size in Figure 86, Figure 87, Figure 88 and Figure 89.

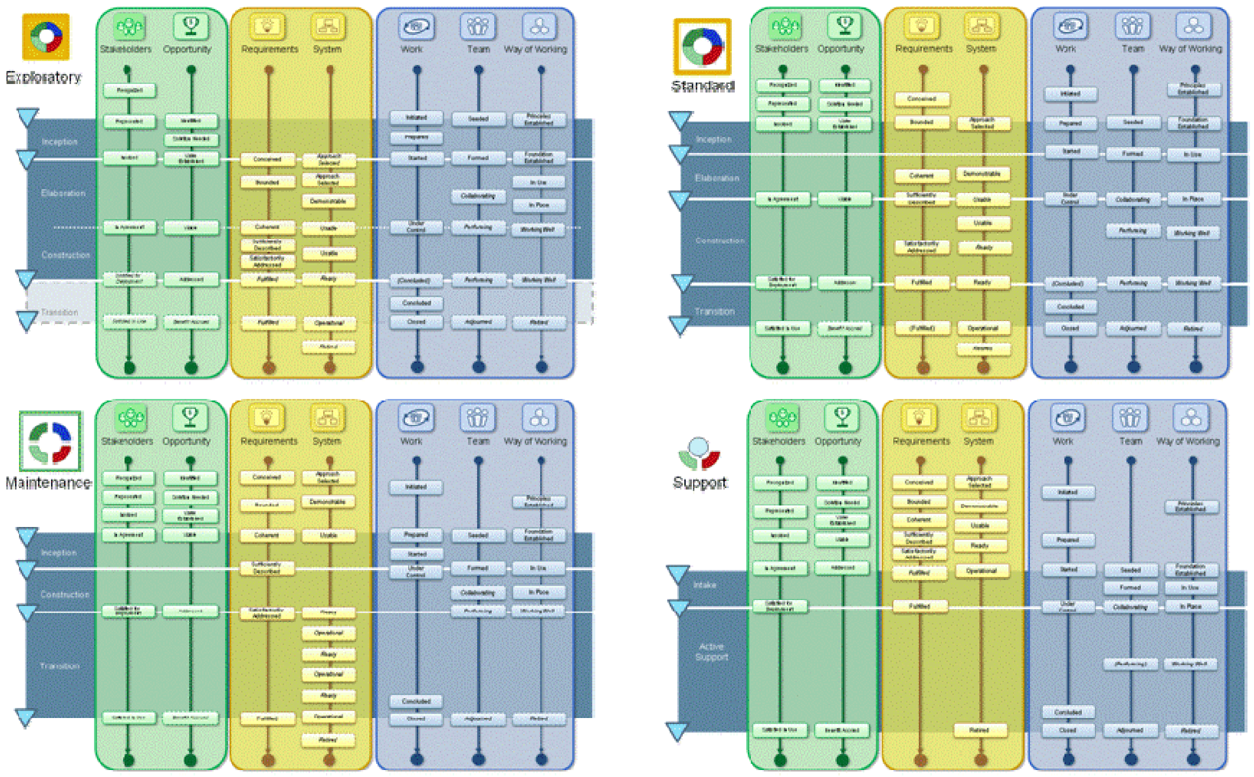


Figure 85 – Four complementary lifecycles to support application development

The interesting things to note here are:

1. The different starting points of the different lifecycles. In this case much of the preparation work for standard developments is done outside the Application Development project; hence the fact that the Opportunity is *value established*, the Requirements are *bounded* and the System is *architecture selected* before the standard method is used.

2. The way that maintenance doesn't start until there is a *usable* system, and Support doesn't start until there is an *operational* System. These two methods are very focused with the Maintenance lifecycle only supporting small changes and not allowing architectural change. If you want to change the architecture you must apply either the Exploratory or the Standard lifecycles and their supporting methods.

3. The different end points of the different lifecycles. For example Transition is optional in the Exploratory method and the Support method continues until the system is *retired*.

4. The Standard lifecycle is called standard as this is the default lifecycle for the teams to follow.

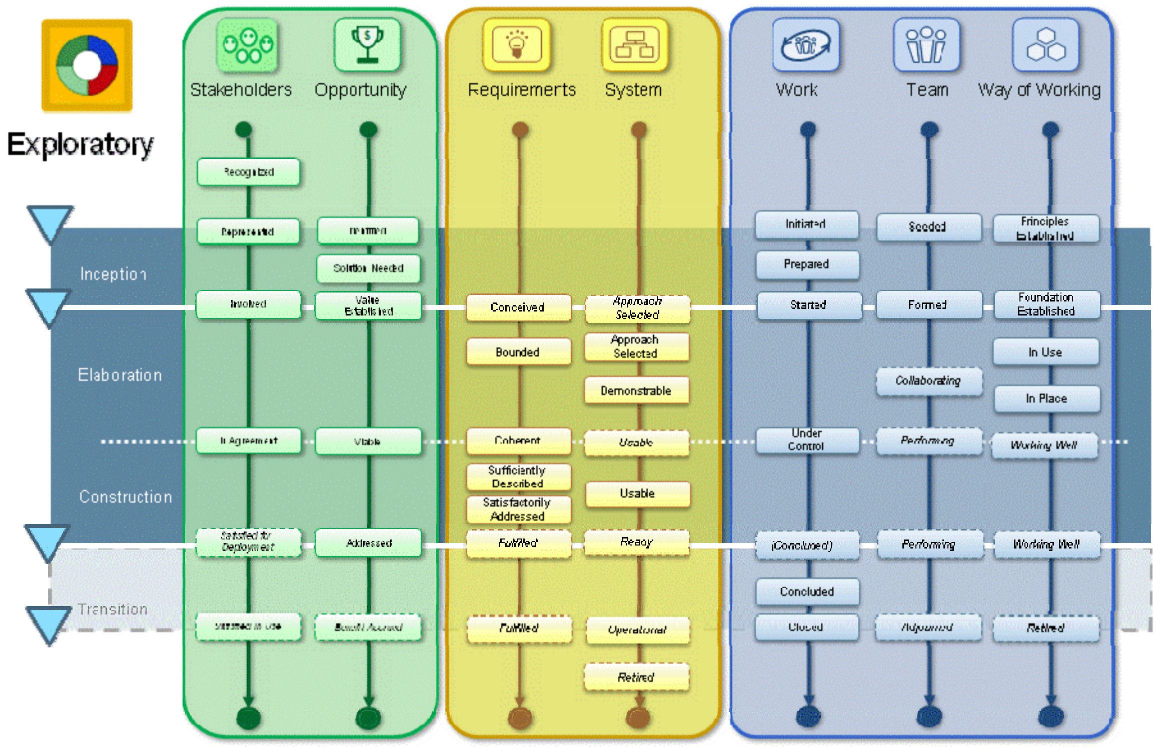


Figure 86 – The Exploratory lifecycle

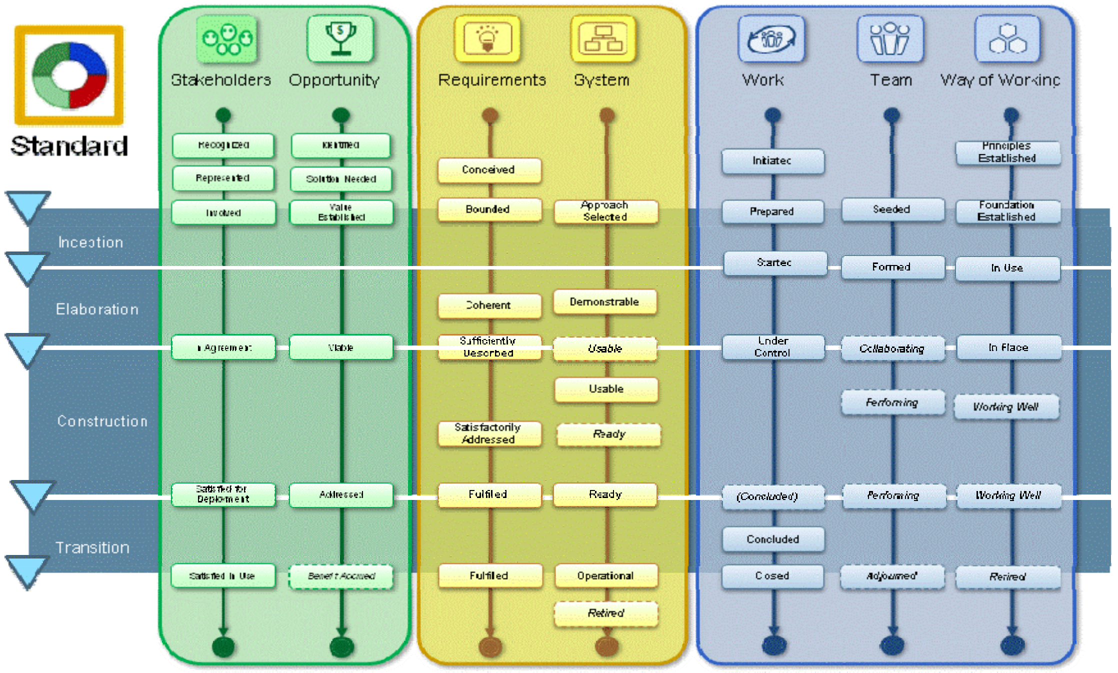


Figure 87 – The Standard lifecycle

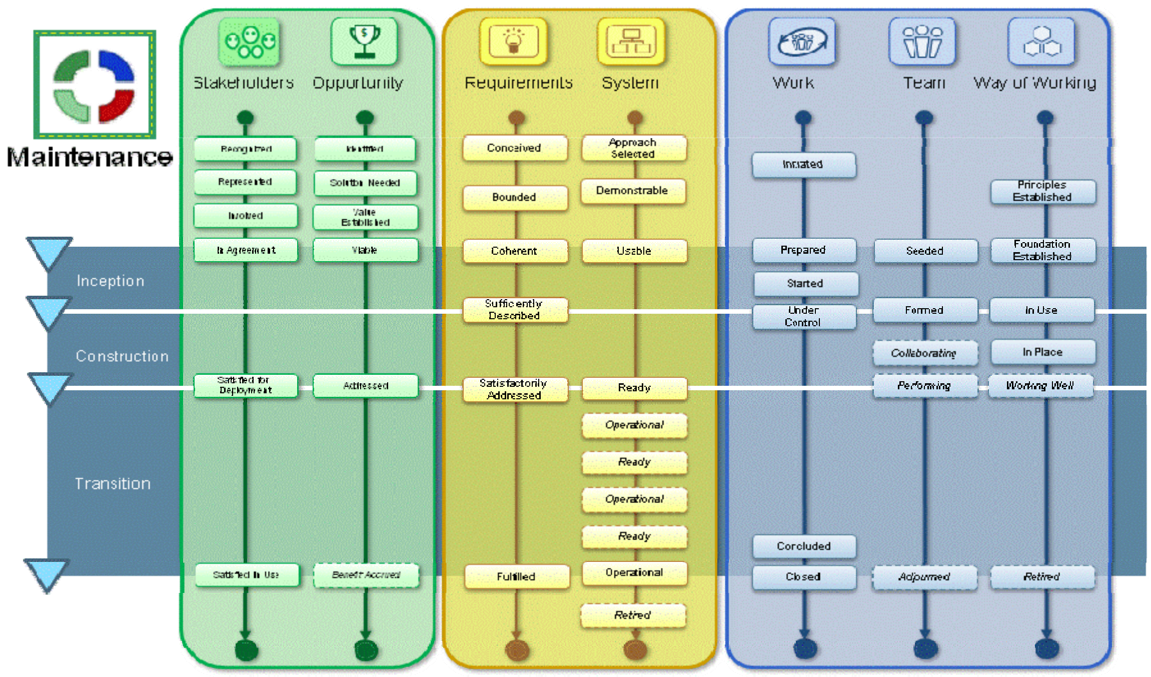


Figure 88 – The Maintenance lifecycle

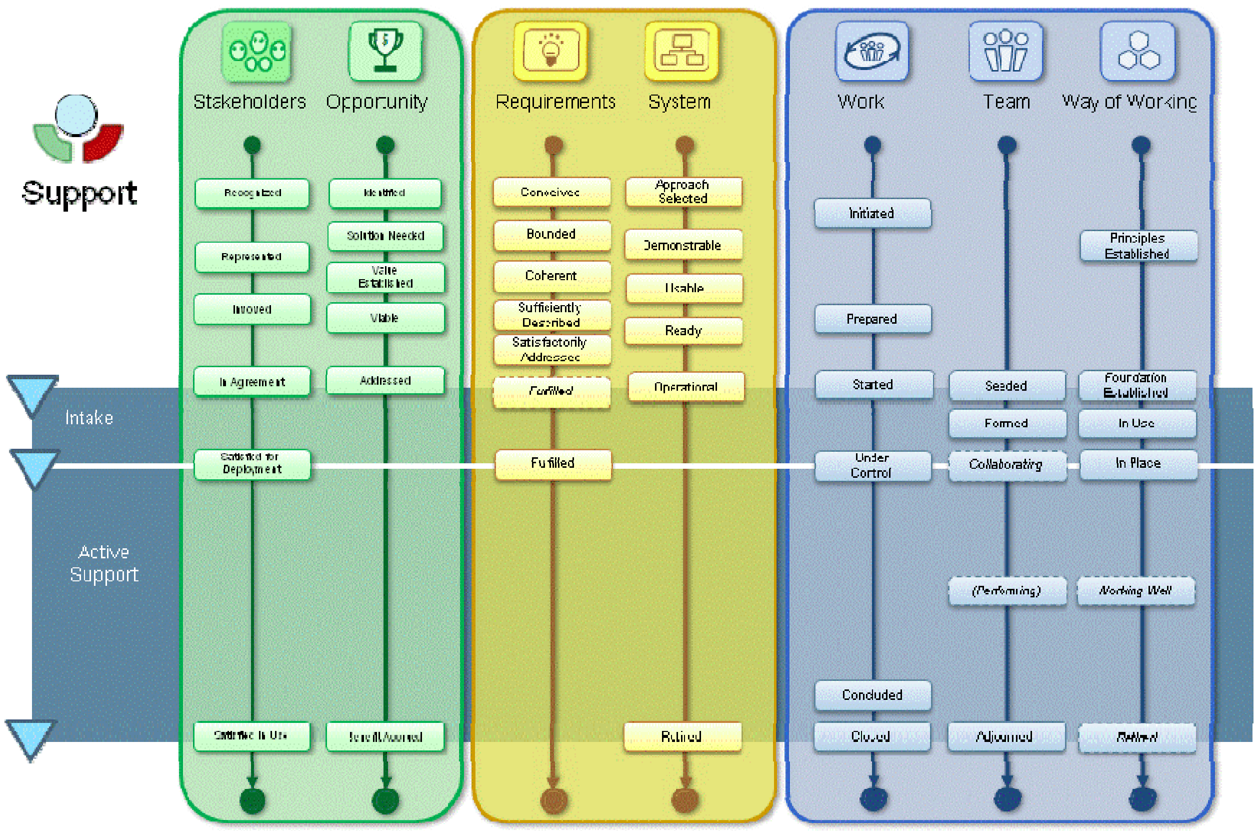


Figure 89 – The Support lifecycle

C.2 Composing Practices into Methods

C.2.1 Composing Scrum and User Story

In Scrum requirement items are expressed as Product Backlog items. Scrum does not provide any guidance on how to express these requirements items. Many Scrum teams adopt user stories to express their requirements. A simple composition of the Scrum and User Story work products with respect to the Requirements alpha is shown below.

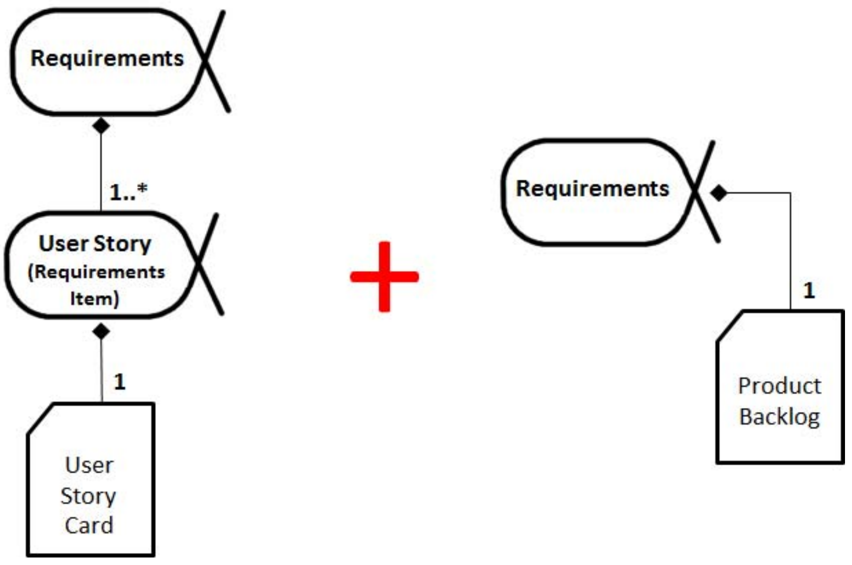


Figure 90 – Merging User Story with Scrum

This simple composition adds work products from different practices to the same alpha, and also relates the sub-alphas of Requirements. The result of the merger is shown below.

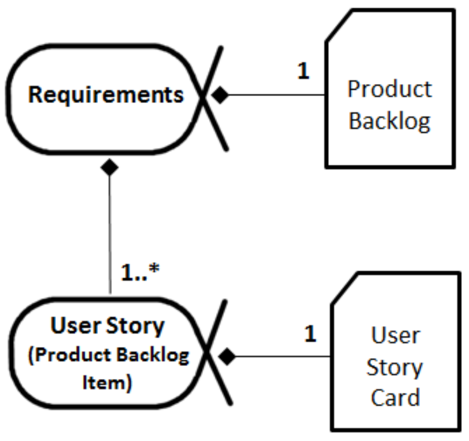
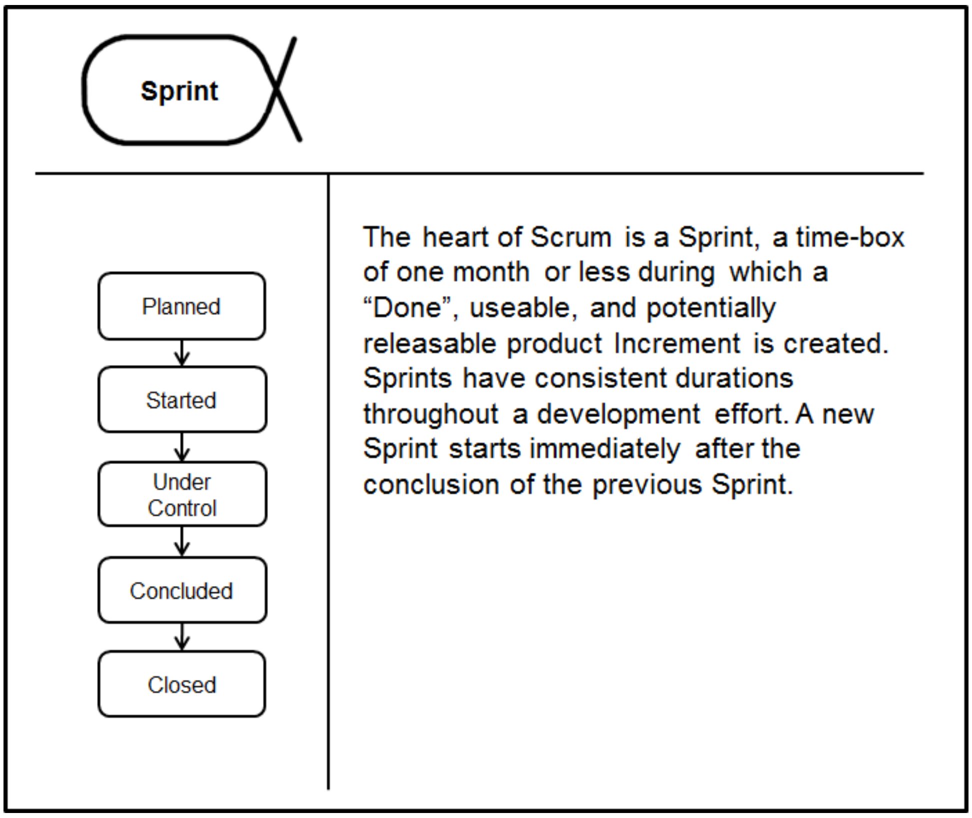


Figure 91 – Scrum with User Story

C.3 Enactment of Methods

C.3.1 Enactment using Alpha State Cards

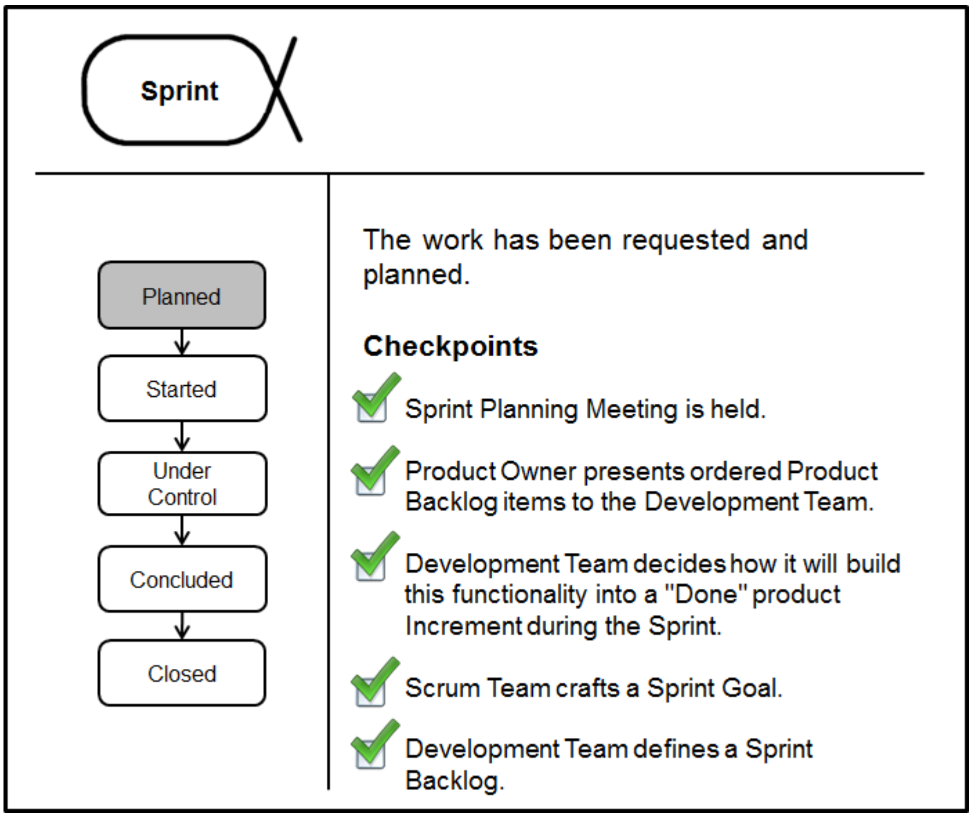
The concrete syntax of the language has been designed so that usage of the composed practices, i.e., the method, should be easy by the practitioners. One key idea here is the concept of Alpha state cards. Below we show the state card for the Sprint alpha at the initial state.



The heart of Scrum is a Sprint, a time-box of one month or less during which a "Done", useable, and potentially releasable product Increment is created. Sprints have consistent durations throughout a development effort. A new Sprint starts immediately after the conclusion of the previous Sprint.

Figure 92 – Sprint state card (initial state)

These cards can be used for reading and understanding the practice, and also how to progress the states of the Sprint according to the checklist defined. Below we show the state card for the Sprint alpha in the Planned state. This requires that all checkpoints are ticked off.



Sprint
Planned
Started
Under Control
Concluded
Closed
The work has been requested and planned.
Checkpoints
Sprint Planning Meeting is held.
Product Owner presents ordered Product Backlog items to the Development Team.
Development Team decides how it will build this functionality into a "Done" product Increment during the Sprint.
Scrum Team crafts a Sprint Goal.
Development Team defines a Sprint Backlog.

Figure 93 – Sprint state card (planned state)

These state cards may also have different views so that instead of the checklist items one could get a list of activities to do in order to move from one state to another, or which work products to produce. Using the concept of ViewSelection in our language would allow us to define the necessary views that are suitable for different kinds of practitioners.

Annex D: Overview of SPEM 2.0 features

(Informative)

This annex provides an overview of SPEM 2.0 features.

Table 30 – SPEM 2.0 features

SPEM 2.0 language construct	Enumeration literal/association stereotype & SPEM 2.0 base extension	Description (single sentence)
Activity		An Activity is a Work Breakdown Element and Work Definition that defines basic units of work within a Process as well as a Process itself.
	NestedBreakdownElement	This association represents breakdown structure nesting. It defines an n-level hierarchy of Activities grouping together other Breakdown Elements such as other Activities, Milestones, etc.
	Suppressed	The suppressed association allows hiding any Breakdown Element from the interpretation of a process structure.
ActivityKind		Activity Kinds provides the capability for a process engineer to define life-cycle models using the terminology they are used to.
	Phase	Phase represents a significant period in a project, ending with major management checkpoint, milestone, or set of Deliverables.
	Iteration	Iteration groups a set of nested Activities that are repeated more than once.
	Process	A Process is a special Activity that describes a structure for particular types of development projects or parts of them.
	DeliveryProcess	A Delivery Process is a special Process describing a complete and integrated approach for performing a specific project type.
	ProcessPattern	A Process Pattern is a special Process that describes a reusable cluster of Activities in a general process area that provides a consistent development approach to common problems.

	ProcessPlanningTemplate	A Process Planning Template is a special Process that is prepared for instantiation by a project planning tool.
ActivityUseKind		This enumeration defines the nature of the reuse for an Activity that relates to exactly one other Activity via the used Activity association.
	UsedActivity (extension)	Extends provides a mechanism for dynamically reusing Activity substructures (elements contained via the nested Breakdown Element composition) in other Activities.
	UsedActivity (localContribution)	Local Contribution defines a mechanism for defining specific local additions (or contributions) to breakdown elements inherited via the extension Activity Use Kind within the context of the reusing Activity.
	UsedActivity (localReplace)	Local Replace defines a mechanism for defining local replacements to specific breakdown elements inherited via the Extension Activity Use Kind in the context of the reusing Activity.
BreakdownElement		Breakdown Element is an abstract generalization for any type of Process Element that is part of a breakdown structure.
Category		A Category is a Describable Element used to categorize, i.e., group any number of Describable Elements of any subtype based on user-defined criteria.
Category (View)		"View" is not explicitly called out as a type in SPEM 2.0, but is mentioned as a kind of category that can be used as views / navigation structures.
CategoryKind		Category Kinds are a flexible way of defining different groupings for Content Categories.
	Discipline	A Discipline is a categorization of work (i.e., Tasks for Method Content), based upon similarity of concerns and cooperation of work effort.
	RoleSet	A Role Set organizes Roles into categories.

	Domain	Domain is a refineable hierarchy grouping related work products.
	Tool Category	A Tool Category is a container/aggregate for Tool Mentors.
CompositeRole		A Composite Role is a special Role Use that relates to more than one Role Definition.
	AggregatedRole	This association lists all the Roles Definitions represented by the Composite Role.
DescribableElement		Describable Element is an Extensible Element that represents an abstract generalization for all elements in SPEM 2.0 that can be documented with textual descriptions.
ExtensibleElement		Extensible Element is an abstract generalization that represents any SPEM 2.0 class for which it is possible to assign a Kind to its instances expressing a user-defined qualification.
Guidance		Guidance is a Describable Element that provides additional information related to Describable Elements.
GuidanceKind		Allows to define commonly used guidance kinds.
	Checklist	A Checklist is a specific type of guidance that identifies a series of items that need to be completed or verified.
	Concept	A Concept is a specific type of guidance that outlines key ideas associated with basic principles underlying the referenced item.
	Estimate	An Estimate is a specific type of Guidance that provides sizing measures, or standards for sizing the work effort associated with performing a particular piece of work and instructions for their successful use.
	EstimationConsideration	Estimation Considerations qualify the usage and application of estimation metrics in the development of an actual estimate.

	EstimationMetric	Estimation Metric describes a metric or measure that is associated with an element and which is used to calculate the size of the work effort as well as a range of potential labor.
	Example	An Example is a specific type of Guidance that represents a typical, partially completed, sample instance of one or more work products or scenario-like description of how Task may be performed.
	Guideline	A Guideline is a specific type of guidance that provides additional detail on how to perform a particular task or grouping of tasks (e.g., grouped together as activities), or that provides additional detail, rules, and recommendations on work products and their properties.
	Practice	A Practice represents a proven way or strategy of doing work to achieve a goal that has a positive impact on work product or process quality.
	Report	A Report is a predefined template of a result that is generated on the basis of other work products as an output from some form of tool automation.
	ReusableAsset	A Reusable Asset provides a solution to a problem for a given context.
	Roadmap	A Roadmap is a special Guidance Kind that is only related to Activities.
	SupportingMaterial	Supporting Materials is a catch-all for other types of guidance not specifically defined elsewhere.
	Template	A Template is a specific type of guidance that provides for a work product a predefined table of contents, sections, packages, and/or headings, a standardized format, as well as descriptions how the sections and packages are supposed to be used and completed.
	TermDefinition	Term Definitions define concepts and are used to build up the Glossary.
MethodConfiguration		A Method Configuration is a collection of selected Method Plugins, as well as subsets of Method Content Packages and

		Process Packages of respective Method Plugins.
	BaseConfiguration	The definition of a configuration can be based on the definitions of other configurations.
	PackageSelection	A selection of packages to be included in the configuration.
MethodContentElement		Method Content Element is an abstract Describable Element that represents an abstract generalization for all Method Content Elements in SPEM 2.0.
MethodContentPackage		A Method Content Package is a Method Content Packageable Element and Package that contains Method Content Elements only.
MethodLibrary		A Method Library is a physical container for Method Plugins and Method Configuration definitions.
MethodPlugin		A Method Plugin is a Package that represents a physical container for Content and Process Packages.
	BasePlugin	This association defines that Method Plugins could extend many other Method Plugins.
Metric		A Metric is a special Describable Element that contains one or more constraints that provide measurements for any Describable Element.
Milestone		A Milestone is a Work Breakdown Element that represents a significant event for a development project.
	RequiredResults	This association links the Work Product Uses instances to a Milestone instance that need to be produced for that Milestone.
OptionalityKind		This enumeration provides the values for the Task Definition Parameter attribute optionality.
	OptionalityMandatory	It is mandatory to provide the Work Product Definition specified in this parameter as input or to provide an instance of the Work Product Definition as output respectively.

	OptionalityOptional	It is optional to provide the Work Product Definition specified in this parameter as input or to provide an instance of the Work Product Definition as output respectively.
Performer		A Process Performer is a Breakdown Element and Work Definition Performer that represents a relationship between Activity instances and Role Use instances.
Planning Data		Planning Data is a Process Element that adds planning data to Breakdown Elements when it is used for generating project plans from a process.
	PlannedElement	The Planned Element stereotype can be used as a superclass for other stereotypes that need to store planning data such as Activity or Task Use.
ProcessComponent		A Process Component is a special Process Package that applies the principles of encapsulation.
	WorkProductPort	This association defines the ports required or provided by the Process Component.
ProcessComponentUse		A Process Component Use represents a Process Component application in any other Process defined by a breakdown structure.
ProcessElement		Process Elements is an Extensible Element that represents abstract generalization for all elements that are part of a SPEM 2.0 Process.
ProcessPackage		Derived from the UML 2 package with additional constraints that enforce the physical separation of method content and process definitions.
Qualification		Qualification is a Method Content Element that documents zero or more required qualifications, skills, or competencies for Role and/or Task Definitions.
ResponsibilityAssignment		A Default Responsibility Assignment is a Method Content Element that represents a relationship between instances of Role

		Definition and Work Product Definition.
RoleDefinition		A Role Definition is a Method Content Element that defines a set of related skills, competencies, and responsibilities.
RoleUse		A Role Use represents a Role in the context of one specific Activity.
Section		A Section is a special Class that represents a structural subsection of a Content Description's mainDescription attribute. It is used for large scale documentation of Describable Elements organized into sections, as well as to flexibly add new Sections to Describable Elements using contribution variability.
Step		A Step is a Section and Work Definition that is used to organize a Task Definition's Content Description into parts or subunits of work.
TaskDefinition		A Task Definition is a Method Content Element and a Work Definition that defines work being performed by Roles Definition instances.
TaskUse		A Task Use is a Method Content Use and Work Breakdown Element that represents a proxy for a Task Definition in the context of one specific Activity.
	MethodContentTrace	This association represents the reference from the Method Content Use to the Method Content Element it refers to.
TeamProfile		A Team Profile is a Breakdown Element that groups Role Uses or Composite Roles defining a nested hierarchy of teams and team members.
ToolDefinition		A Tool Definition is a special Method Content Element that can be used to specify a tool's participation in a Task Definition.
VariabilityElement		Variability Element is an abstract class derived from Classifier that provides capabilities for content variation and extension to a specific list of SPEM 2.0 classes.
	VariabilitySpecialization	This stereotype is abstract and intended to serve as the base for the three concrete

		stereotypes defined for Variability Type.
VariabilityType		Variability Type is an Enumeration used for values for instances of Variability Element's attribute variabilityType.
	VariabilityContributes	Contributes provides a way for instances of Variability Elements to contribute their properties into their base Variability Element without directly altering any of its existing properties, i.e., in an additive fashion.
	VariabilityExtendsReplaces	Extends-replaces combines the effects of extends and replace variability into one new variability type.
	VariabilityExtends	Extension allows Method Plugins to easily reuse elements from a Base Plugin by providing a kind of inheritance for the special Variability Element.
	VariabilityReplaces	Replaces provides a way for Variability Elements to define a replacement of a base Variability Element without directly changing any of its existing properties.
WorkBreakdownElement		A Work Breakdown Element is a special Breakdown Element that provides specific properties for Breakdown Elements that represent work (see Figure 9.11).
WorkDefiniton		Work Definition is an abstract Classifier that generalizes all definitions of work within SPEM 2.0.
WorkDefinitionParameter		A Work Definition Parameter is an abstract generalization for Process Elements that represent parameter for Work Definitions.
	ParameterIn	This attribute represents the kind of the input as specified by the enumeration Parameter Direction Kind.
	ParameterInOut	This attribute represents the kind of the input as specified by the enumeration Parameter Direction Kind.
	ParameterOut	This attribute represents the kind of the input as specified by the enumeration Parameter Direction Kind.
WorkProductDefinition		Work Product Definition is Method

		Content Element that is used, modified, and produced by Task Definitions.
WorkProductKind		Allows to define commonly used work product kinds.
	Outcome	Outcome Definition is a Work Product Definition that provides a description and definition for non-tangible work products.
	Deliverable	A Deliverable Definition is a Work Product Definition that provides a description and definition for packaging other Work Products, and may be delivered to an internal or external party.
	Artifcat	Artifact Definition is a Work Product Definition that provides a description and definition for tangible work product types.
WorkProductRelationship		A Work Product Definition Relationship expresses a general relationship among Work Products Definitions.
WorkProductRelationshipKind		Work Product Relationship Kinds define relationships among work products.
	Composition	‘composition’ expressing that a work product use instance of an instance is part of another work product instance of an instance.
	Aggregation	‘aggregation’ indicating that a Work Product Use is used with another Work Product Use.
	ImpactedBy	‘impacted by’ indicating that a work product use impacts another work product use.
WorkProductUse		A Work Product Use represents a Work Product Definition in the context of one specific Activity. Every breakdown structure can define different relationships of Work Product Uses to Task Uses and Role Uses.
WorkSequence		Work Sequence is a Breakdown Element that represents a relationship between two Work Breakdown Elements in which one Work Breakdown Elements depends on the start or finish of another Work Breakdown Elements in order to begin or

		end.
WorkSequenceKind		Work Sequence represents a relationship between two Work Breakdown Elements in which one Work Breakdown Element (referred to as (B) below) depends on the start or finish of another Work Breakdown Element (referred to as (A) below) in order to begin or end.
	finishToStart	Work Breakdown Element (B) cannot start until Work Breakdown Element (A) finishes.
	finishToFinish	Breakdown Element (B) cannot finish until Work Breakdown Element (A) finishes.
	startToStart	Breakdown Element (B) cannot start until Work Breakdown Element (A) starts.
	startToFinish	Breakdown Element (B) cannot finish until Work Breakdown Element (A) starts.

D.1 RMC/EPF extensions to SPEM 2.0

Rational Method Composer (RMC) and Eclipse Process Framework (EPF) are compliant with the SPEM 2.0 specification from 2008. RMC and EPF have both evolved and now contains some new useful language features to support practice composition that are not part of the SPEM 2.0 specification, but they can be seen as proposed extensions to SPEM 2.0.

- The main difference between UMF and SPEM 2.0 is the introduction of a “kernel” to support a practices framework.
- This kernel is defined mainly with SPEM 2.0 constructs, but a few extensions to the meta-model were needed for practice composition to work.

D.1.1 Extensions to SPEM 2.0 to support Practice Composition

Table 31 – SPEM 2.0 features

SPEM 2.0 extension	**Description**
Work Product Slot	These are similar to Alphas, except that there is no state information associated with them. They are used to decouple tasks – so a task can take “requirements” as an input, without specifying whether the requirements are “use cases”, “user stories” or something else. This is implemented in RMC as a flag on a work product – marking it as a “slot”. A work product can “fulfil” a slot – this is a new relationship between work products.
Process Slot	This is similar to activity space. It is used to create a WBS that is independent of the selected practices. EPF does not leverage this, however IBM internal methods make extensive use of process slots.

Namespace	This is implemented as a "." notation in naming plug-ins. This allows plug-ins to be grouped by context, type, and practice.
Practice (redefined)	It now is like a special kind of custom category • it groups elements (including other practices) • has standard attributes like "purpose" • has some unique publishing characteristics to make it easy to browse o roadmaps, then concepts, work products, tasks, guidance.
Supporting elements	Practices often share elements, such as work products. Shared elements are published only if used. To mark an element as "publish only if used", it is placed in a plug-in or packaged marked as "supporting".

D.1.2 UMF Kernel

The UMF kernel is built with the standard SPEM 2.0 language with the above extensions. The UMF kernel consists of:

- A standard set of work product slots
- Naming conventions for organizing plug-ins, including:
    - Plug-in category – "core", "practice", "process" and "publish" to reflect different kinds of method plug-ins
    - "context"- a top level prefix to organize content into the major contexts of "technical", "management", "business" and "general")
    - Suffixes to indicate whether the plug-in contains assignments only, or contains content owned by a specific company, such as "-IBM"
- Authoring guidelines
- A replaceable default set of roles and categories (you can use the default, or substitute with your own set)

Annex E: Overview of ISO 24744 features

(Informative)

This annex provides an overview of ISO 24744 features.

Table 32 – ISO 24744 features

ISO 24744 language construct	Description (single sentence)
Action	An action is a usage event performed by a task upon a work product.
ActionKind	An action kind is a specific kind of action, characterized by a given cause (a task kind), a given subject (a work product kind) and a particular type of usage.
Build	A build is a stage with duration for which the major objective is the delivery of an incremented version of an already existing set of work products.
BuildKind	A build kind is a specific kind of build, characterized by the type of result that it aims to produce.
CompositeWorkProduct	A composite work product is a work product composed of other work products.
CompositeWorkProductKind	A composite work product kind is a specific kind of composite work product, characterized by the kinds of work products that are part of it.
Conglomerate	A conglomerate is a collection of related methodology elements that can be reused in different methodological contexts.
Constraint	A constraint is a condition that holds or must hold at certain point in time.
Document	A document is a durable depiction of a fragment of reality.
DocumentKind	A document kind is a specific kind of document, characterized by its structure, type of content and purpose.
Element	An element is an entity of interest to the metamodel. Element is an abstract class, specialized into MethodologyElement and EndeavourElement.
EndeavourElement	An endeavour element is an element that belongs in the endeavour domain.
Guideline	A guideline is an indication of how a set of methodology elements can be used during enactment.
HardwareItem	A hardware item is a piece of hardware of interest to the endeavour.
HardwareItemKind	A hardware item kind is a specific kind of hardware item, characterized by its mechanical and electronic characteristics, requirements and features.
InstantaneousStage	An instantaneous stage is a managed point in time within an endeavour.
InstantaneousStageKind	An instantaneous stage kind is a specific kind of instantaneous stage, characterized by

	the kind of event that it represents.
Language	A language is a structure of model unit kinds that focus on a particular modelling perspective.
MethodologyElement	A methodology element is an element that belongs in the methodology domain.
Milestone	A milestone is an instantaneous stage that marks some significant event in the endeavour.
MilestoneKind	A milestone kind is a specific kind of milestone, characterized by its specific purpose and kind of event that it signifies.
Model	A model is an abstract representation of some subject that acts as the subject’s surrogate for some well defined purpose.
ModelKind	A model kind is a specific kind of model, characterized by its focus, purpose and level of abstraction.
ModelUnit	A model unit is an atomic component of a model, which represents a cohesive fragment of information in the subject being modelled.
ModelUnitKind	A model unit kind is a specific kind of model unit, characterized by the nature of the information it represents and the intention of using such a representation.
ModelUnitUsage	A model unit usage is a specific usage of a given model unit by a given model.
ModelUnitUsageKind	A model unit usage kind is a specific kind of model unit usage, characterized by the nature of the use that a given model kind makes of a given model unit kind.
Notation	A notation is a concrete syntax, usually graphical, that can be used to depict models created with certain languages.
Outcome	An outcome is an observable result of the successful performance of any work unit of a given kind.
Person	A person is an individual human being involved in a development effort.
Phase	A phase is a stage with duration for which the objective is the transition between cognitive frameworks.
PhaseKind	A phase kind is a specific kind of phase, characterized by the abstraction level and formality of the result that it aims to produce.
PostCondition	A postcondition is a constraint that is guaranteed to be satisfied after an action of the associated kind is performed.
PreCondition	A precondition is a constraint that must be satisfied before an action of the associated kind can be performed.
Process	A process is a large-grained work unit that operates within a given area of expertise.
ProcessKind	A process kind is a specific kind of process, characterized by the area of expertise in which it occurs.

Producer	A producer is an agent that has the responsibility to execute work units.
ProducerKind	A producer kind is a specific kind of producer, characterized by its area of expertise.
Reference	A reference is a specific linkage between a given methodology element and a given source.
Resource	A resource is a methodology element that is directly used at the endeavour level, without an instantiation process.
Role	A role is a collection of responsibilities that a producer can take.
RoleKind	A role kind is a specific kind of role, characterized by the involved responsibilities.
SoftwareItem	A software item is a piece of software of interest to the endeavour.
SoftwareItemKind	A software item kind is a specific kind of software item, characterized by its scope, requirements and features.
Source	A source is a source of information, experience or best practices.
Stage	A stage is a managed time frame within an endeavour.
StageKind	A stage kind is a specific kind of stage, characterized by the abstraction level at which it works on the endeavour and the result that it aims to produce.
StageWithDuration	A stage with duration is a managed interval of time within an endeavour.
StageWithDurationKind	A stage with duration kind is a specific kind of stage with duration, characterized by the abstraction level at which it works on the endeavour and the result that it aims to produce.
Task	A task is a small-grained work unit that focuses on what must be done in order to achieve a given purpose.
TaskKind	A task kind is a specific kind of task, characterized by its purpose within the endeavour.
TaskTechniqueMapping	A task-technique mapping is a usage association between a given task and a given technique.
TaskTechniqueMappingKind	A task-technique mapping kind is a specific kind of task-technique mapping, characterized by the mapped task kind and technique kind.
Team	A team is an organized set of producers that collectively focus on common work units.
TeamKind	A team kind is a specific kind of team, characterized by its responsibilities.
Technique	A technique is a small-grained work unit that focuses on how the given purpose may be achieved.
TechniqueKind	A technique kind is a specific kind of technique, characterized by its purpose within the endeavour.

Template	A template is a methodology element that is used at the endeavour level through an instantiation process.
TimeCycle	A time cycle is a stage with duration for which the objective is the delivery of a final product or service.
TimeCycleKind	A time cycle kind is a specific kind of time cycle, characterized by the type of outcomes that it aims to produce.
Tool	A tool is an instrument that helps another producer to execute its responsibilities in an automated way.
ToolKind	A tool kind is a specific kind of tool, characterized by its features.
WorkPerformance	A work performance is an assignment and responsibility association between a particular producer and a particular work unit.
WorkPerformanceKind	A work performance kind is a specific kind of work performance, characterized by the purpose of the inherent assignment and responsibility association.
WorkProduct	A work product is an artefact of interest for the endeavour.
WorkProductKind	A work product kind is a specific kind of work product, characterized by the nature of its contents and the intention behind its usage.
WorkUnit	A work unit is a job performed, or intended to be performed, within an endeavour.
WorkUnitKind	A work unit kind is a specific kind of work unit, characterized by its purpose within the endeavour.