

Essence – Kernel and Language for Software Engineering Methods

Initial Submission – Version 1.0

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/2011-02-04

Standard document URL: <http://www.omg.org/cgi-bin/doc?ad/2011-02-04/PDF>

Associated File(s)*: <http://www.omg.org/cgi-bin/doc?ad/2011-02-04>

Submission Team

OMG Submitters:

Fujitsu
Ivar Jacobson International AB
Model Driven Solutions

Supporting Organizations:

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

Copyright © 2012, Florida Atlantic University
Copyright © 2012, Fujitsu
Copyright © 2012, Impetus
Copyright © 2012, International Business Machines Corporation
Copyright © 2012, Ivar Jacobson International AB
Copyright © 2012, KTH Royal Institute of Technology
Copyright © 2012, Metamaxim Ltd.
Copyright © 2012, PEM Systems
Copyright © 2012, Stiftelsen SINTEF
Copyright © 2012, University of Duisburg-Essen

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 140 Kendrick Street, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™, Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™, MOF™, OMG Interface Definition Language (IDL)™, and OMG SysML™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

Table of Contents

1	Scope.....	1
2	Conformance.....	1
3	Normative References	1
4	Terms and Definitions.....	2
5	Symbols	4
6	Additional Information	4
6.1	Submitting Organizations.....	4
6.2	Supporting Organizations.....	4
6.3	Submission Contacts	4
6.4	Acknowledgements	4
6.5	Status of the Document	5
6.6	Responses to RFP Requirements.....	5
7	Overview of the Specification	6
7.1	Introduction to Essence	6
7.2	The Key Differentiators	7
8	Kernel Specification	9
8.1	Overview	9
8.1.1	What is the Kernel?.....	9
8.1.2	What is in the Kernel?.....	9
8.1.3	Organizing the Kernel.....	9
8.1.4	Alphas: The Things to Work With	10
8.1.5	Activity Spaces: The Things to Do	11
8.2	The Customer Area of Concern.....	13
8.2.1	Introduction.....	13
8.2.2	Alphas	13
8.2.2.1	Stakeholders.....	13
8.2.2.2	Opportunity.....	16
8.2.3	Activity Spaces	19
8.2.3.1	Explore Possibilities	19
8.2.3.2	Involve the Stakeholders	19

8.2.3.3	Ensure Stakeholder Satisfaction	20
8.2.3.4	Use the System	20
8.3	The Solution Area of Concern.....	21
8.3.1	Introduction.....	21
8.3.2	Alphas	21
8.3.2.1	Requirements	21
8.3.2.2	Software System	24
8.3.3	Activity Spaces	28
8.3.3.1	Understand the Requirements.....	28
8.3.3.2	Shape the System.....	28
8.3.3.3	Implement the System	29
8.3.3.4	Test the System	29
8.3.3.5	Deploy the System.....	29
8.3.3.6	Operate the System.....	29
8.4	The Endeavor Area of Concern.....	30
8.4.1	Introduction.....	30
8.4.2	Alphas	30
8.4.2.1	Team	30
8.4.2.2	Work	33
8.4.2.3	Way-of-Working	36
8.4.3	Activity Spaces	39
8.4.3.1	Prepare to do the Work	39
8.4.3.2	Coordinate Activity.....	39
8.4.3.3	Support the Team.....	40
8.4.3.4	Track Progress	40
8.4.3.5	Stop the Work	40
9	Language Specification	42
9.1	Language Design.....	43
9.2	Specification Technique	43
9.2.1	Different Meta-Levels.....	43
9.2.2	Specification Format.....	44
9.2.3	Notation Used	44
9.3	Language Elements and Language Model	45
9.3.1	Layer1-Core	45

9.3.1.1	Alpha	46
9.3.1.2	AlphaAssociation.....	47
9.3.1.3	AlphaAssociationEnd	47
9.3.1.4	Checkpoint.....	48
9.3.1.5	Kernel	48
9.3.1.6	State	49
9.3.1.7	StateGraph	50
9.3.1.8	Transition.....	51
9.3.2	Layer2-PracticeAndAlpha	51
9.3.2.1	Alpha	52
9.3.2.2	AlphaContainment.....	52
9.3.2.3	AlphaManifest	53
9.3.2.4	Practice	53
9.3.2.5	WorkProduct.....	55
9.3.3	Layer3-CompletePractice	55
9.3.3.1	Activity	57
9.3.3.2	ActivityManifest.....	58
9.3.3.3	ActivitySpace.....	59
9.3.3.4	Alpha	59
9.3.3.5	AlphaAssociation.....	60
9.3.3.6	AreaOfConcern.....	60
9.3.3.7	Competency	61
9.3.3.8	CompetencyLevel.....	62
9.3.3.9	CompletionCriterion.....	62
9.3.3.10	Kernel	63
9.3.3.11	Pattern.....	64
9.3.3.12	Practice	64
9.3.3.13	RequiredCompetency	66
9.3.3.14	RequiredSkill	67
9.3.3.15	Skill.....	67
9.3.3.16	SkillLevel.....	68
9.3.4	Layer4-MethodAndLibrary.....	68
9.3.4.1	Library	68
9.3.4.2	Method.....	69
9.4	Composition	70

9.4.1	Introduction.....	70
9.4.2	Graph Algebra.....	70
9.4.2.1	Variable Definition	70
9.4.2.2	Renaming.....	72
9.4.2.3	Merge.....	72
9.4.3	Required Primitive Operations	73
9.4.4	Additional Definitions in the Algebra.....	73
9.4.5	Composition of Practices	74
9.4.5.1	Definition of the Compose Operation	74
9.4.5.2	Applying the Compose Operation	74
9.4.6	Examples.....	74
9.4.6.1	Simple Composition	74
9.5	Dynamic Semantics.....	77
9.5.1	Domain classes.....	77
9.5.1.1	Recap of Meta-modeling Levels.....	77
9.5.1.2	Naming Convention.....	77
9.5.1.3	Abstract Superclasses	78
9.5.2	Operational Semantics	79
9.5.2.1	Populating the Level 0 Model	79
9.5.2.2	Determining the Overall State	79
9.5.2.3	Generating Guidance	80
9.5.2.4	Formal definition of the Guidance Function	80
9.6	Graphical Syntax	82
9.6.1	Specification Format.....	82
9.6.2	Relevant Symbols	82
9.6.3	Default Notation for Meta-Class Constructs.....	82
9.6.4	View 1: Alphas and their States	83
9.6.4.1	Alpha	83
9.6.4.2	Alpha Association.....	83
9.6.4.3	Kernel	84
9.6.4.4	State	85
9.6.4.5	Transition.....	85
9.6.4.6	Diagrams.....	86
9.6.4.7	Cards.....	87
9.6.5	View 2: Sub-Alphas and Work Products.....	89

9.6.5.1	Work Product.....	89
9.6.5.2	Alpha Containment.....	90
9.6.5.3	Alpha Manifest	90
9.6.5.4	Practice	91
9.6.5.5	Diagrams.....	92
9.6.6	View 3: Activity Spaces and Activities.....	93
9.6.6.1	Activity	93
9.6.6.2	Activity Space.....	93
9.6.6.3	Activity Manifest.....	93
9.6.6.4	Activity Predecessor	94
9.6.6.5	Competency	95
9.6.6.6	Diagrams.....	95
9.7	Textual Syntax.....	97
9.7.1	Rules	97
9.7.1.1	Root Elements.....	97
9.7.1.2	Kernel Elements	98
9.7.1.3	Practice Elements	100
9.7.1.4	Auxiliary Elements	101
9.7.2	Examples.....	101
Annex A:	Responses to RFP Requirements	104
A.1	Mandatory Requirements	104
A.2	Optional Requirements.....	110
Annex B:	Issues to be Discussed.....	111
B.1	Kernel.....	111
B.1.1	Alphas	111
B.1.1.1	Alternatives Considered but Rejected for Opportunity	111
B.1.1.2	Alternatives Considered but Rejected for Stakeholders	112
B.1.1.3	Alternatives Considered but Rejected for Requirements	113
B.1.1.4	Alternatives Considered but Rejected for Software System	113
B.1.1.5	Alternatives Considered but Rejected for Work	114
B.1.1.6	Alternatives Considered but Rejected for Way of Working	115
B.1.1.7	Alternatives Considered but Rejected for Team.....	115
B.1.2	Activity Spaces	117
B.1.2.1	Alternative Names for the Activity Spaces.....	117

B.1.3	Alternative sets of activity spaces.....	118
B.2	SPEM 2.0	119
Annex C:	Practice Examples	120
C.1	Practices	120
C.1.1	Scrum.....	120
C.1.1.1	Practice	120
C.1.1.2	Alphas.....	121
C.1.1.3	Work Products	125
C.1.1.4	Activities.....	127
C.1.1.5	Roles	128
C.1.2	User Story	129
C.1.2.1	Practice	129
C.1.2.2	Work Products	130
C.1.2.3	Activities.....	130
C.1.3	Lifecycle Examples.....	130
C.1.3.1	The Unified Process Lifecycle	131
C.1.3.2	The Waterfall Lifecycle	132
C.1.3.3	A set of complementary application development lifecycles	133
C.2	Composing Practices into Methods.....	137
C.3	Enactment of Methods	137

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

- CORBA services
- CORBA facilities
- 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

<TBD>

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 9.3.3.7.)

Constraints

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

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 9.3.2.4.)

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

There are no symbols defined in this specification.

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
- Impetus
- International Business Machines Corporation
- 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 Cuninghame, 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, 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, Roly Stimson and Paul Szymkowiak.

6.5 Status of the Document

This document is an initial 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 inceptioned 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.
- 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.

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. Dobbs'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>

- 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 Figure 1⁴.

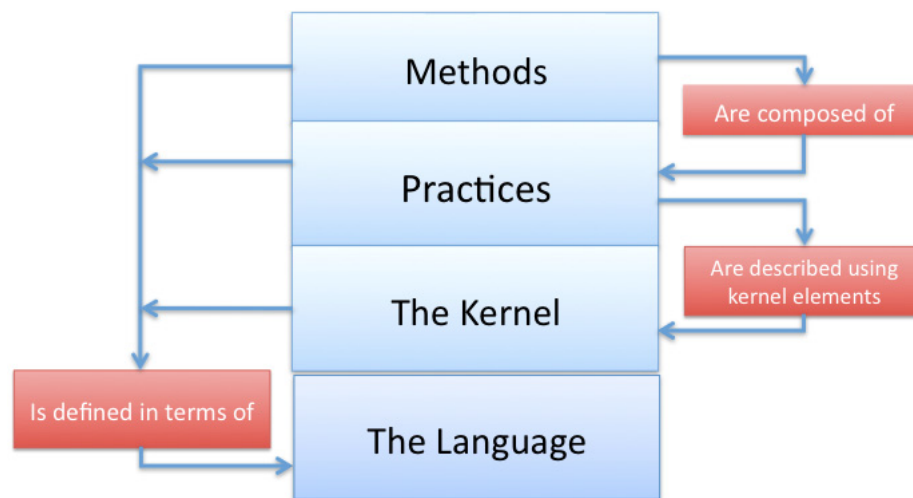


Figure 1 – Method architecture

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 following⁵:

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 Separate of Concerns (SoC) that puts focus on the things that matter the most.
 - 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

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. *The Essence of Software Engineering – Applying the Semat Kernel*, in preparation to be published

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 9 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.

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

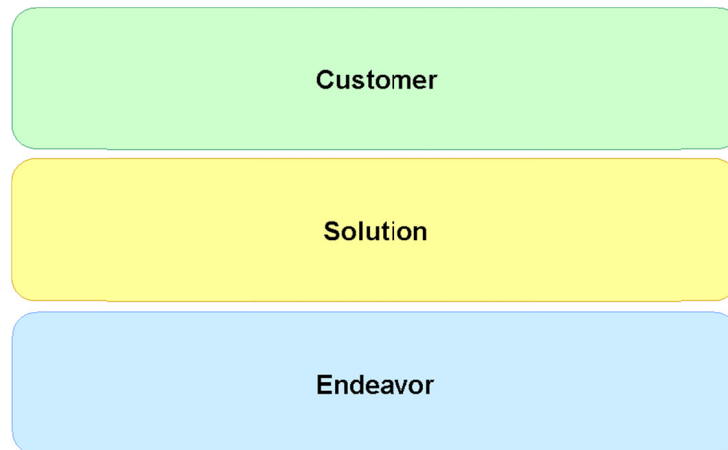


Figure 2 – The Three Areas of Concern

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.

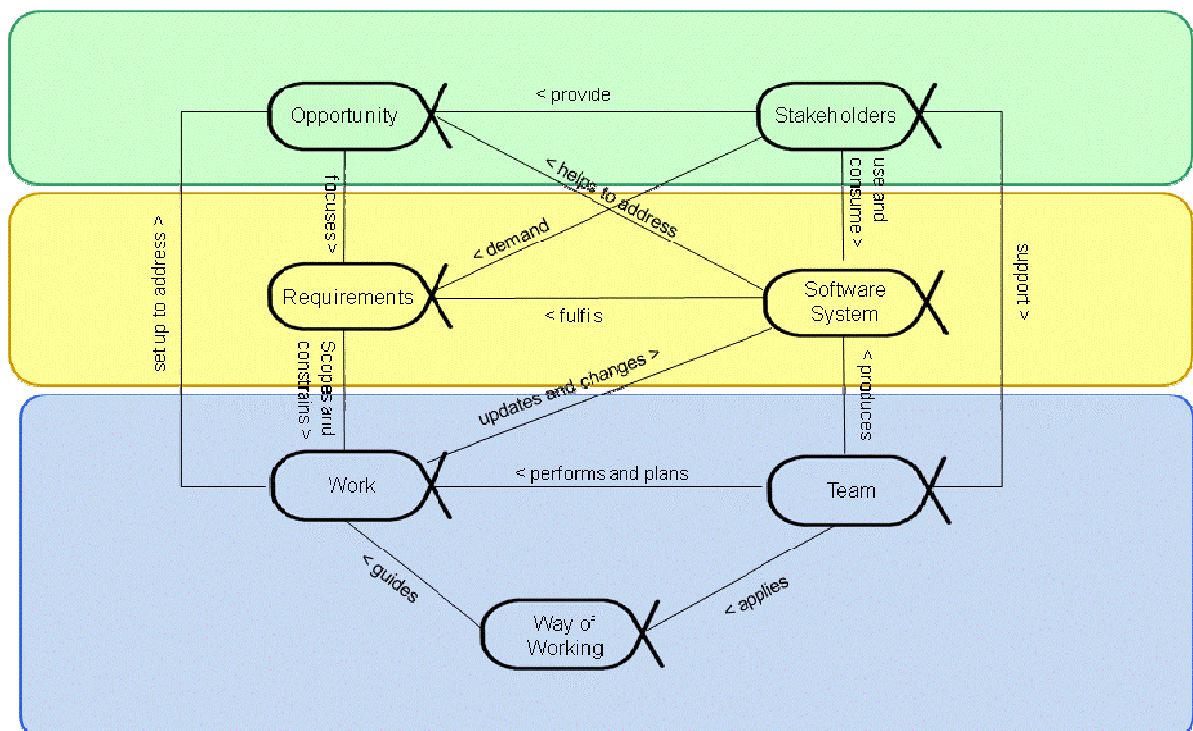


Figure 3 – The Kernel Alphas

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.

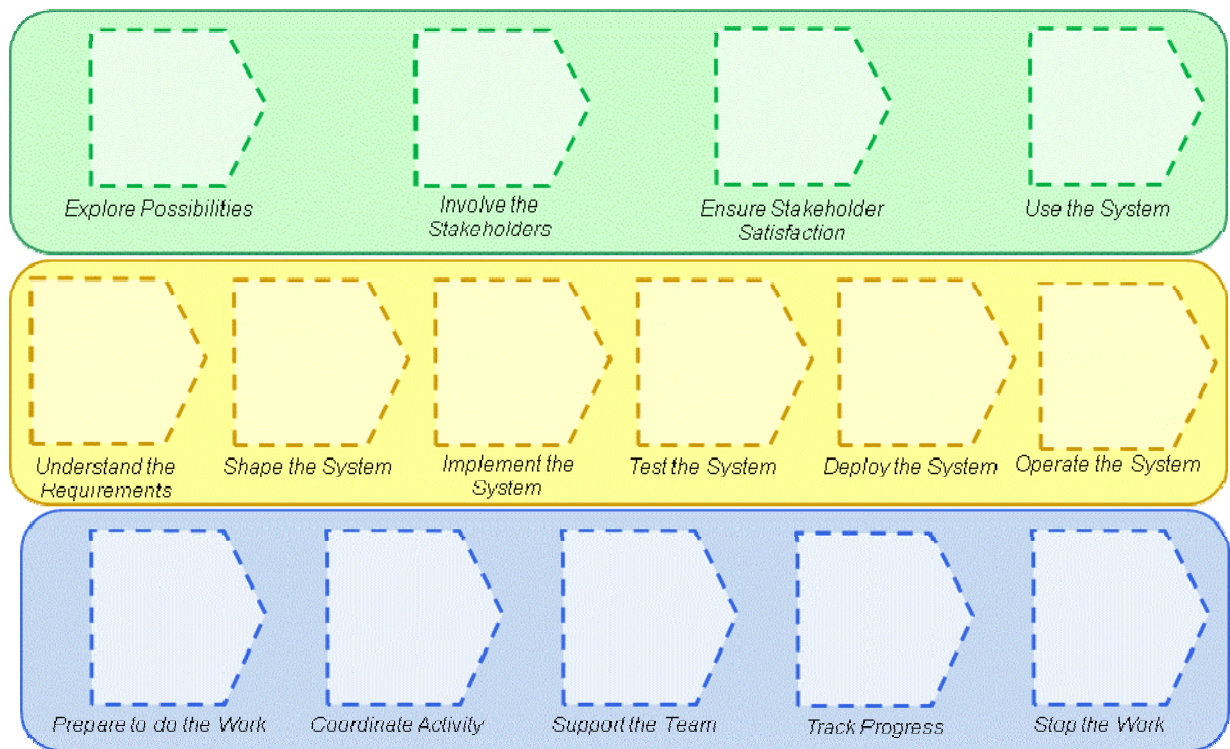


Figure 4 – The Kernel Activity Spaces

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.
- **Involve the Stakeholders:** Involve the stakeholders in the day-to-day activities of the team to 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.

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 participation 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.

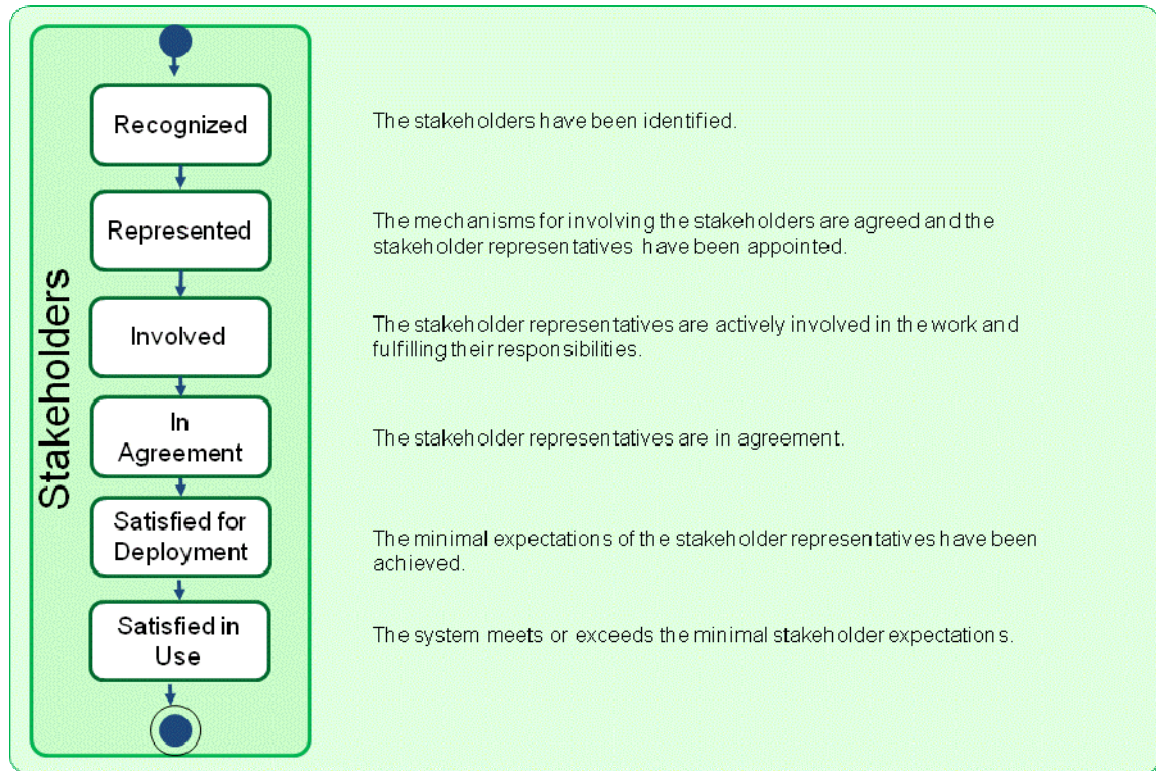


Figure 5 – The states of the Stakeholders

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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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.

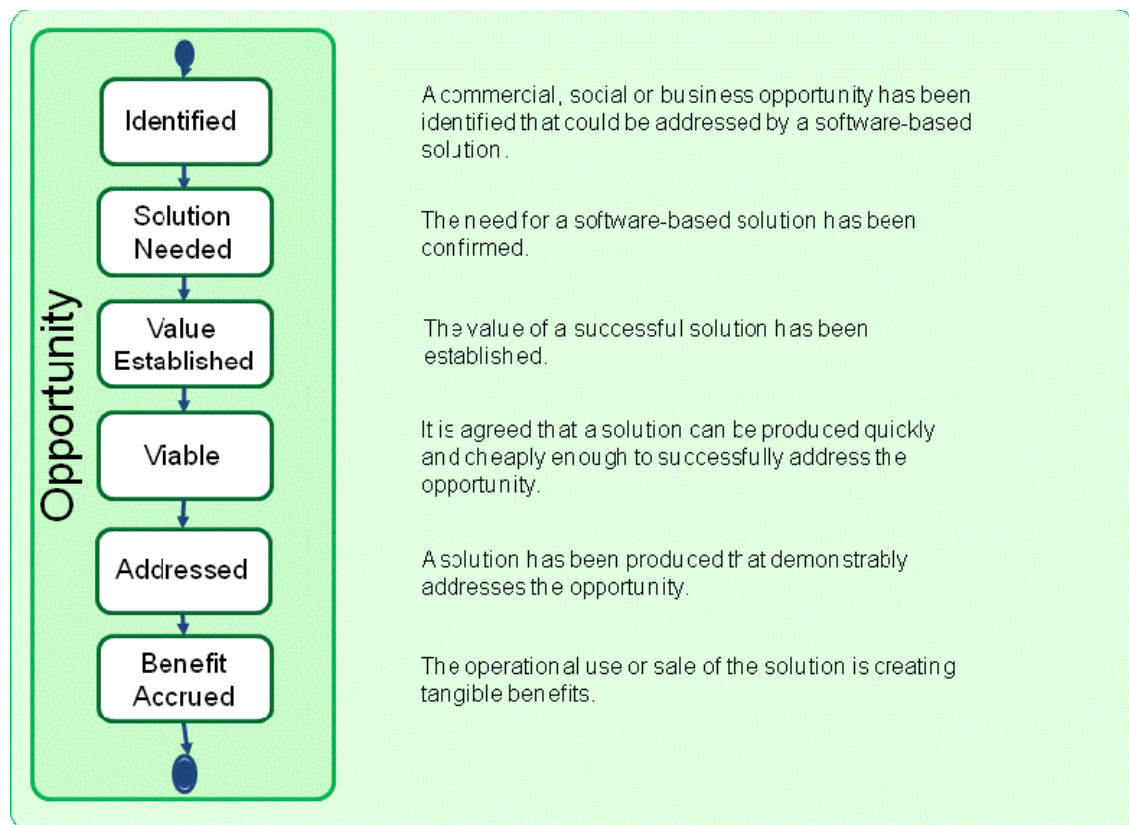


Figure 6 – The states of the Opportunity

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 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	<ul style="list-style-type: none">• An idea for a way of improving current <i>ways of working</i>, increasing market share or applying a new or innovative <i>software system</i> 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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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 <i>software system</i> is understood.

	<ul style="list-style-type: none"> • The success criteria by which the deployment of the <i>software system</i> is to be judged are clear. • The desired outcomes required of the solution are clear and quantified.
Viable	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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 Involve the Stakeholders

Description

Involve the stakeholders in the day-to-day activities of the team to ensure that the right results are produced. This

includes identifying and working with the stakeholder representatives to progress the opportunity.

Involve the stakeholders to:

- Ensure the right solution is created.
- Give all stakeholder groups a voice.
- 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.
- To gather feedback from the use of the system.
- To confirm that the system meets the expectations of the stakeholders.
- To 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.
Sufficiently Described	The requirements describe a system that is acceptable to the stakeholders.
Satisfactorily Addressed	The requirements that have been addressed 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.

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*, *sufficiently described*, *satisfactorily 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.

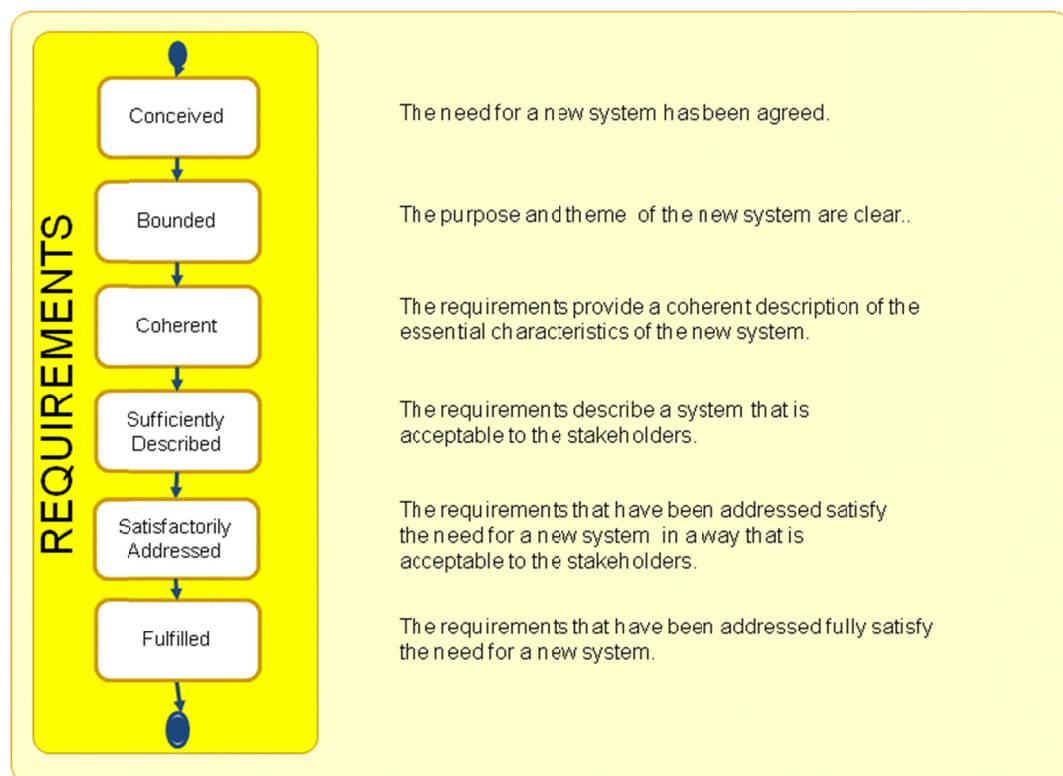


Figure 7 – The states of the Requirements

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 item 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 a *sufficiently described* 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.

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 *satisfactorily 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 *satisfactorily 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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.
Sufficiently Described	<ul style="list-style-type: none"> • 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.
Satisfactorily Addressed	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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*, (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.

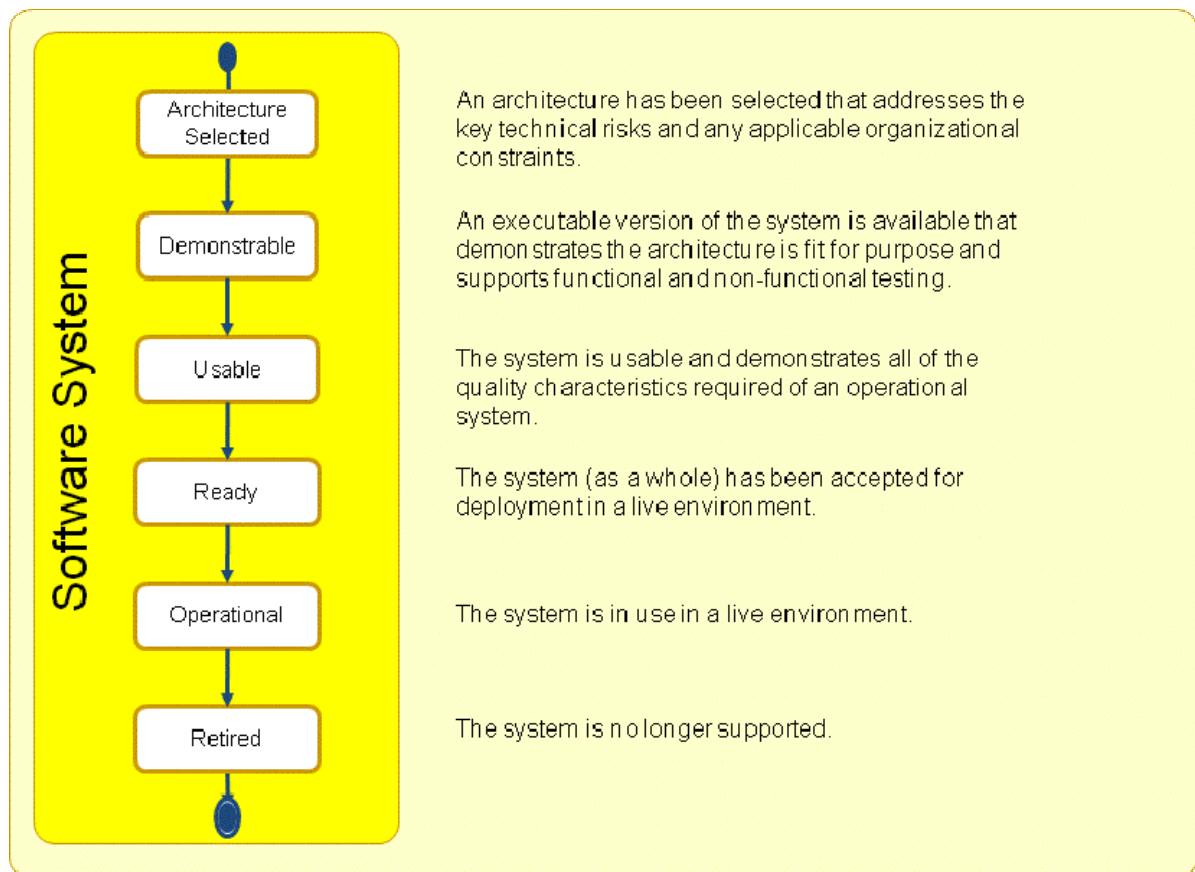


Figure 8 – The states of the Software System

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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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.

	<ul style="list-style-type: none"> Operational support is in place.
Operational	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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 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.

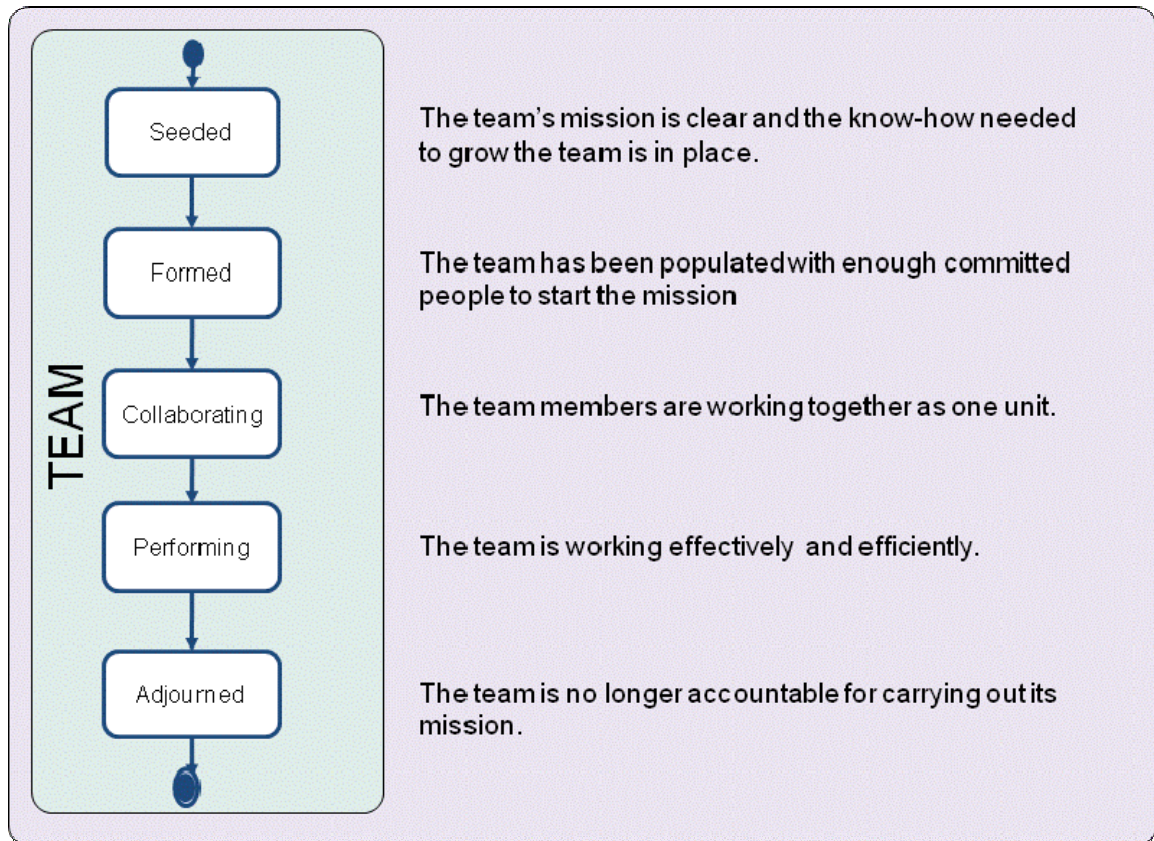


Figure 9 – The states of the Team

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.

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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• The team consistently meets its commitments.• The team continuously adapts to the changing context.

	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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.

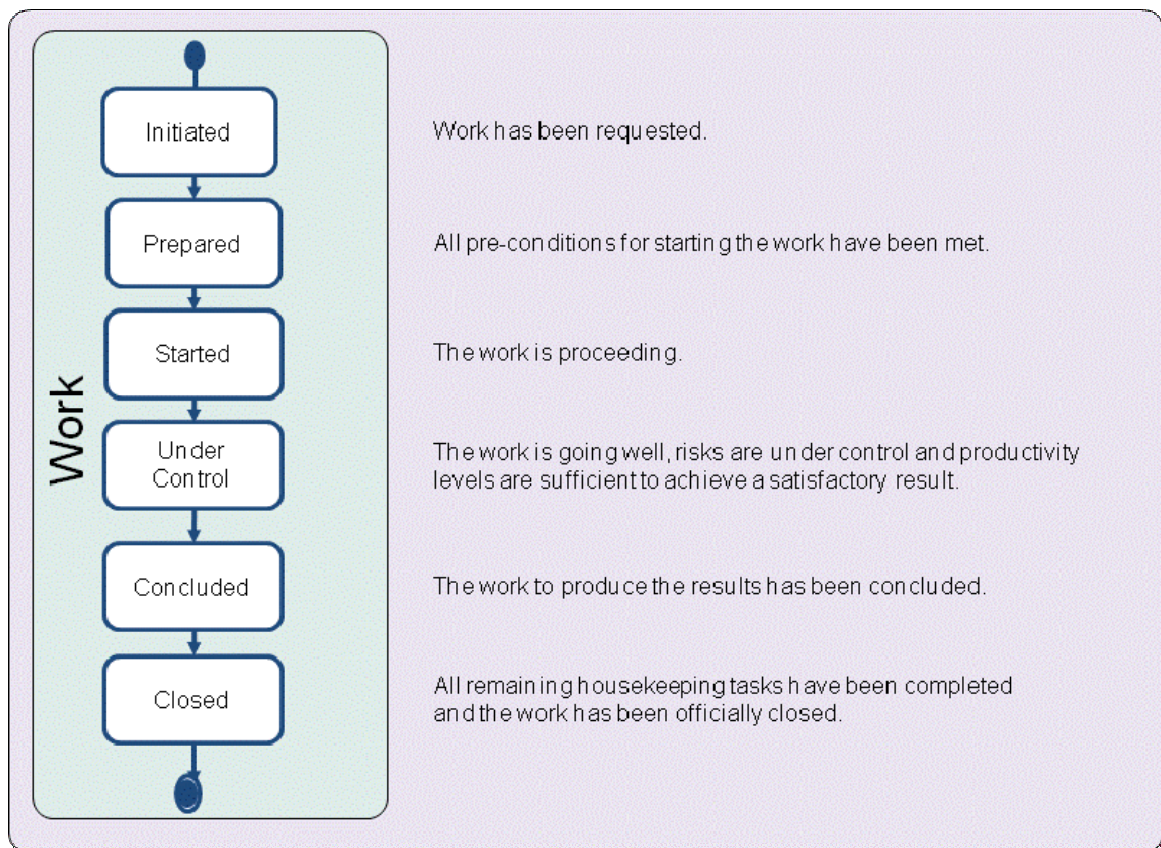


Figure 10 – The states of the Work

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 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 the as 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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

	<p>have been reduced to acceptable levels.</p> <ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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.

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.

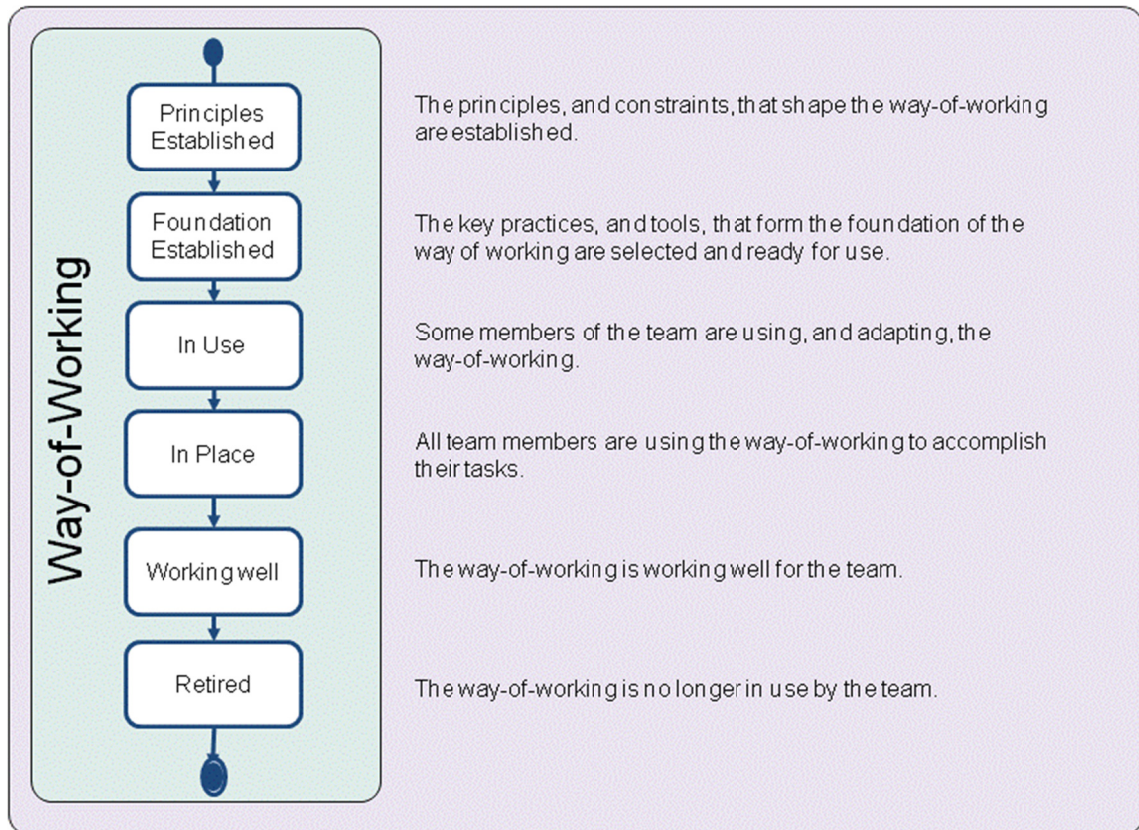


Figure 11 – The states of the Way-of-Working

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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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 Language Specification

The Essence language is based on the experience achieved in using earlier languages with a similar set of goals. Something worked and something didn't work so well.

We learnt that

1. Though there are many methods, the hypothesis (partly proven experimentally) is that each method is a composition of a set of practices. The number of practices is a factor 1000 less than the number of methods. The Essence language needs to be able to describe methods as compositions of practices, and to define each practice at the depth required by the developers using the practice, for instance in terms of the work products it is expected that developers produce (possibly tacit) while doing real work.
2. Underneath all methods and practices is a common ground, now captured as the Essence kernel. The Essence language needs to be able to define the kernel and all the elements of the kernel.
3. The discovery of the alpha construct, allowing developers to measure progress and health in a software development endeavor. The Essence language needs to be able to define alphas whether they are elements of the kernel or elements defined specific for a practice.

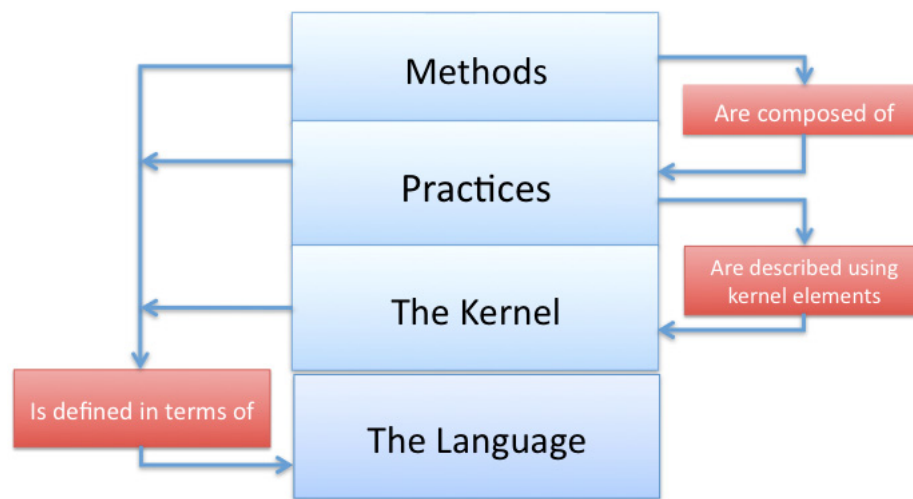


Figure 12 – The Method architecture of Semat

To get to this result a key idea applied throughout the language design is the principle of Separation of Concerns⁶.

With this background in mind, the overall goals of the Essence language are: 1) to support different levels of usages, 2) to make it easy to work with methods to create, compose, compare and change them, and 3) to make methods support the developers in their daily endeavors.

The first objective should allow developers to use just a subset of all language elements, a subset of all possible representations, or a subset of all possible usages for the language. See the concept of layers and the concept of views in the graphical syntax for answers to these challenges.

The second objective moved the graphical syntax into focus, which is considered to be more than plain representation of constructs, but a key feature of great importance to developers.

The third 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 return (a possibly alternate) advice on what to do next for the same situation.

⁶ The Principle of Separation of Concerns online at http://en.wikipedia.org/wiki/Separation_of_concerns

9.1 Language Design

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.

The abstract syntax of the language is organized in layers. Each layer contains a number of elements and their associations. Besides the bottom layer, each layer may require elements of a lower layer to create well-formed language constructs. No layer requires elements of a higher layer to create well-formed language constructs. No layer changes the semantics of the elements on lower layers. However, elements defined on one layer may be extended on a higher layer to add additional attributes or associations. The reason for designing the language in layers is to allow partial usage of the language. The layers are the following:

- Layer 1 Core, contains the base elements to form a minimal core of the language. No practices can be expressed using this layer, but a domain model for software engineering endeavors can be created.
- Layer 2 PracticeAndAlpha, contains the base elements to form minimal practices. No activities can be expressed using this layer, but concrete work products can be related to abstract domain elements.
- Layer 3 CompletePractice, contains elements to enrich practices by expressing activities, skills, and patterns.
- Layer 4 MethodAndLibrary, contains elements to organize sets of practices.

The concrete syntax of the language is organized in views. Each view provides notations for a subset of elements of the language. Views are defined and used independently from abstract syntax layers. For example, a view capable of representing elements from abstract syntax layers 1, 2 and 3 can be used to represent a language construct just containing elements from abstract syntax layers 1 and 2. The view is allowed to represent just a part of the whole language construct. In the same way, a view capable of representing just elements from abstract syntax layer 1 can also be used to represent (parts of) the same language construct. It is allowed to define and use other views than the ones defined in this language specification.

9.2 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 9.3) 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 9.4) as well as the dynamic semantics (see Section 9.5) are described using natural language (English) accompanied by a formal calculus where appropriate.

9.2.1 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].

9.2.2 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
 - UnlimitedNatural
 - 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.

9.2.3 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 layer and the class is not defined in that layer, the full name of that class is used. For instance, Layer1::Alpha refers to the class Alpha that belongs to package Layer1.

9.3 Language Elements and Language Model

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 layers and each of these layers is described in a sub-section.

The layers are presented as packages in the diagram shown in Figure 13, and the ordering between the layers are expressed with package import relationships between the packages. The relationship implies that all elements visible inside a layer (a package) are visible inside the next layer (the importing package). Note that these layers are not to be confused with the meta-levels defined in Section 9.2.1.

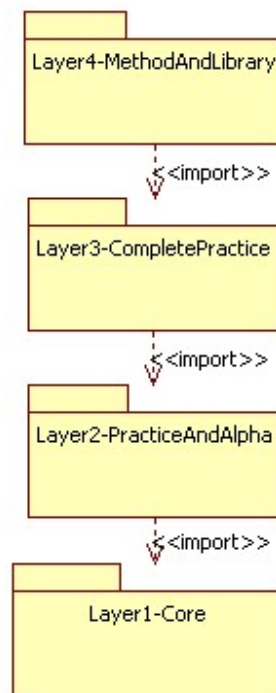


Figure 13 – The language is organized in four layers, the elements visible in one layer are imported into the next layer

9.3.1 Layer1-Core

The intention of layer 1 is to provide all elements necessary to form a kernel containing alphas and alpha associations. The elements and their relationships are presented in the diagram shown in Figure 14. A detailed definition of each of the elements is found below.

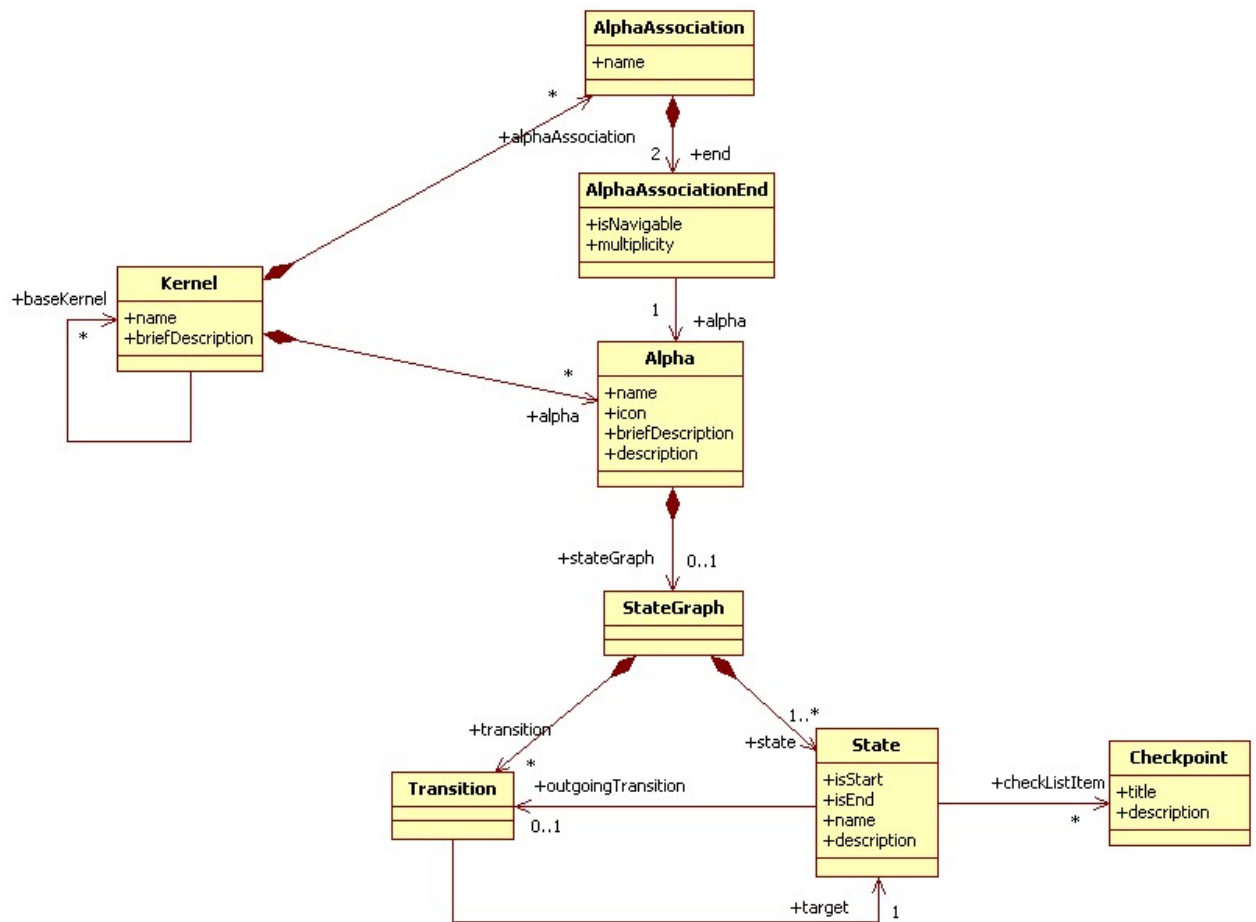


Figure 14 – Layer 1 elements

9.3.1.1 Alpha

Package: Layer1-Core

Description

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

Generalizations

N/A

Attributes

name : String [1]
 icon : Graphical Element [1]
 briefDescription : String [1]
 description : String [1]

The name of the alpha.
 The icon to be used when presenting the alpha.
 A short description of what the alpha is.
 A more detailed description of the alpha.

Associations

stateGraph : StateGraph [0..1]

The state graph contained by the alpha to describe its states.

true

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 in the state graph 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.

9.3.1.2 AlphaAssociation

Package: Layer1-Core

Description

An alpha association defines a relationship between two alphas.

Generalizations

N/A

Attributes

name : String [1]	The name of the association.
-------------------	------------------------------

Associations

end : AlphaAssociationEnd [2]	The endpoints of the association.
-------------------------------	-----------------------------------

Invariant

true

Semantics

Alpha associations are used to define a structure by describing relationships between its alphas. They contribute to the creation of a domain model for software engineering endeavors.

9.3.1.3 AlphaAssociationEnd

Package: Layer1-Core

Description

An alpha association end defines the connection point between an alpha association and an alpha.

Generalizations

N/A

Attributes

isNavigable : Boolean [1]	State if the association can be traversed from an instance at the opposite end to an instance at this end.
multiplicity : UnlimitedNatural [1]	State how many instances at this end can be linked to one instance at the opposite end.

Associations

alpha : Alpha [1]

Instances attached to this end must be of the same type (or subtype) as the alpha.

Invariant

-- The multiplicity can never be exactly zero.
multiplicity <> 0

Semantics

Alpha association ends connect the two endpoints of an alpha association to alphas. An alpha association end states whether it is possible to navigate from an instance at the opposite side of the association to instance at the side of the alpha association end. Furthermore, the multiplicity of the alpha association end states how many instances at the end may be linked to one instance at the opposite end.

9.3.1.4 Checkpoint

Package: Layer1-Core

Description

A checkpoint states an item in a check list to be verified in a state.

Generalizations

N/A

Attributes

title : String [1]
description : String [1]

The title of the checkpoint.
A description of the checkpoint.

Associations

N/A

Invariant

true

Semantics

A checkpoint defines the statement that must be satisfied if the State associated with the checkpoint is said to be reached.

9.3.1.5 Kernel

Package: Layer1-Core

Description

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

Generalizations

N/A

Attributes

name : String [1]
briefDescription : String [1]

The name of the kernel.
A short description of what this particular kernel is designed for.

icon: GraphicalElement [0..1]

The icon to be used when presenting the Kernel.

Associations

alpha : Alpha [*]

The Alphas contained in this Kernel.

alpha association : Alpha Association
[*]

The Alpha Associations contained in this Kernel.

baseKernel : Kernel [*]

The Kernels this Kernel is based on in terms of composition (see Section 9.4 for the definition of composition).

Invariant

```
-- The alphas associated by alpha associations are available within the kernel or
-- its base kernels.
alphaAssociation->forAll (aa | self.allAlphas ()->includes (aa.end->at (1).alpha)
and
self.allAlphas ()->includes (aa.end->at (2).alpha))
and
-- The alphas within the kernel have unique names.
self.alpha->forAll (a1, a2 | a1 <> a2 implies a1.name <> a2.name)
```

Additional Operations

```
-- All alphas available within the kernel and its base kernels.
Kernel::allAlphas () : set(Alpha)
alpha->union (baseKernel->collect (bk | bk.allAlphas () ) )
```

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.

9.3.1.6 State

Package: Layer1-Core

Description

A state expresses a situation where some condition holds.

Generalizations

N/A

Attributes

name : String [1]

The name of the state.

description : String [1]

Some additional information about the state.

isStart : Boolean [1]

The state is a start state of the state graph.

isEnd : Boolean [1]

The state is an end state of the state graph.

Associations

checkpoints : Checkpoint [*]

A collection of checkpoints associated with the state.

outgoing transition : Transition [0..1]

0 or 1 transition leaving the state.

Invariant

```
-- If a state has no outgoing transitions, it must be an end state.  
self.outgoingTransitions->size() = 0 implies self.isEnd
```

Semantics

A state expresses a situation where some invariant holds. This invariant may express a static situation as well as a dynamic situation, depending on what the state graph expresses in which the state is defined.

9.3.1.7 StateGraph

Package: Layer1-Core

Description

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. In this language, a state graph is always finite.

Generalizations

N/A

Attributes

N/A

Associations

transition : Transition[*]	The transitions contained in the state graph.
state : State [1..*]	The states contained in the state graph.

Invariant

```
-- One and only one State must be the start state of the State Graph.  
self.state->exists(s | s.isStart)  
    and  
not self.state->exists(s1,s2 | s1<>s2 and s1.isStart and s2.isStart)  
    and  
-- One State must be the end state of the State Graph.  
self.state->exists(s | s.isEnd)  
    and  
-- All Transitions of the State Graph must end in a State defined in the State  
-- Graph.  
self.transition->forAll(t | self.state->includes(t.target))  
    and  
-- All outgoing transitions of all states in the state graph must be defined in  
-- the state graph.  
self.state->forAll(s | s.outgoingTransition->forAll(t | self.transition-  
>includes(t)))
```

Semantics

A state graph describes a logical order in which a collection of states is supposed to be traversed. The state graphs are constrained so that every state has at most one outgoing transition. Note that the state graph is an abstraction in the sense that it does not need to capture all possible transitions. E.g., loop-backs and alternations between states may occur, although they are not formally modeled in the graph. A state S is reached when all checkpoint of S are fulfilled and when all predecessor states of S are also reached. The procedure for determining whether state checkpoints are fulfilled is manual, thereby requiring human intervention.

9.3.1.8 Transition

Package: Layer1-Core

Description

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

Generalizations

N/A

Attributes

N/A

Associations

```
target : State [1]
```

The target state of the transition.

Invariant

true

Semantics

A transition connects two states in a state graph. The target state of the transition is supposed to be the state to be reached next, if the owning state of the transition is reached.

9.3.2 Layer2-PracticeAndAlpha

The intention of layer 2 is to provide the basic elements needed for the simplest form of practices. The elements and their relationships are presented in the diagram shown in Figure 15. A detailed definition of each of the elements is found below.

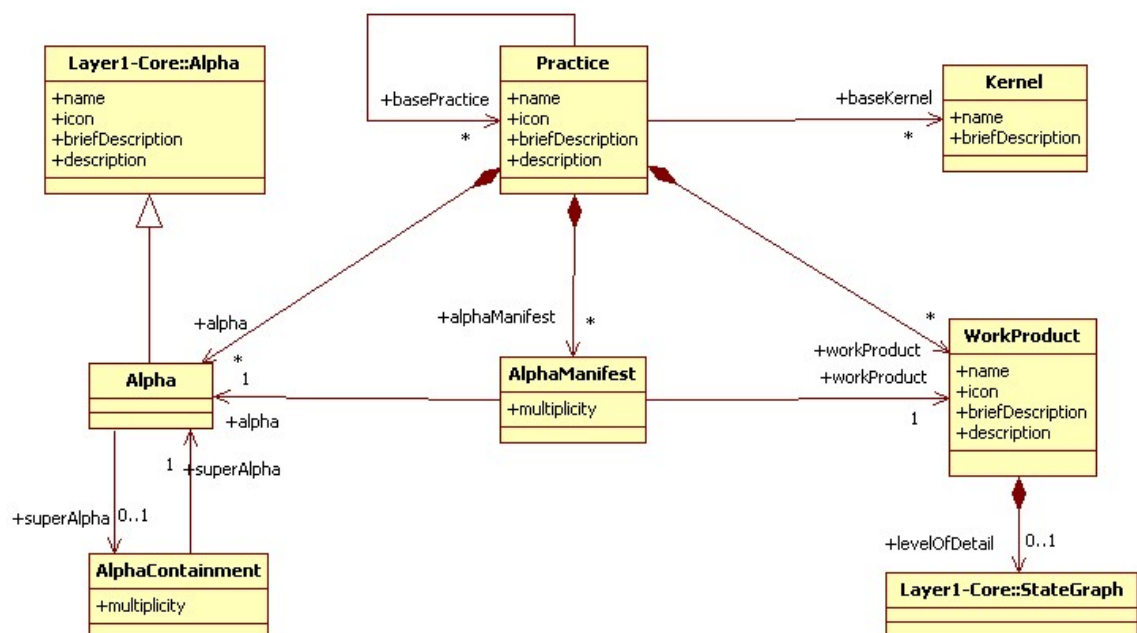


Figure 15 – Layer 2 elements

9.3.2.1 Alpha

Package: Layer2-PracticeAndAlpha

Description

The alpha construct is extended with properties for being defined in a practice, for being described by work products, and for having sub-alphas.

Generalizations

Layer1::Alpha

Attributes

N/A

Associations

superAlpha : AlphaContainment [0..1] An association referencing another alpha which is superordinate to this alpha.

Invariant

true

Semantics

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.

9.3.2.2 AlphaContainment

Package: Layer2-PracticeAndAlpha

Description

An alpha containment is a relationship between a sub-alpha and its superordinate alpha.

Generalizations

N/A

Attributes

multiplicity : UnlimitedNatural [1] How many instances of the sub-alpha there should be in one instance of the superordinate alpha.

Associations

superAlpha : Alpha [1] The superordinate alpha.

Invariant

true

Semantics

An alpha may be defined as a sub-alpha of another alpha (the superordinate alpha). The relationship between the two is expressed with an alpha containment. A sub-alpha is considered to be part of the superordinate alpha and to contribute to its state.

The multiplicity of the sub-alpha, i.e. how many instances of the sub-alpha there should be in one instance of the superordinate alpha, is defined on the relationship.

9.3.2.3 AlphaManifest

Package: Layer2-PracticeAndAlpha

Description

An alpha manifest binds a work product to an alpha.

Generalizations

N/A

Attributes

multiplicity : UnlimitedNatural [1]	The possible number of instances of the work product describing 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

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

9.3.2.4 Practice

Package: Layer2-PracticeAndAlpha

Description

A practice is a description on how to handle a specific aspect of a software engineering endeavor.

Generalizations

N/A

Attributes

name : String [1]	The name of the practice.
-------------------	---------------------------

icon : GraphicalElement [0..1]	The icon to be used when presenting the practice.
briefDescription : String [1]	A short description of what the practice is.
description : String [1]	A thorough description of what the practice is.

Associations

alpha : Alpha [*]	A collection of alphas defined in this practice.
alphaManifest : AlphaManifest [*]	A collection of alpha manifests defined in this practice.
workProduct : WorkProduct [*]	A collection of work products defined in this practice.
basePractice : Practice [*]	The set of Practices from which this Practice is composed (see Section 9.4 for the definition of composition).
baseKernel : Kernel [*]	The Kernels this Practice is based on in terms of composition (see Section 9.4 for the definition of composition).

Invariant

```
-- The alphas and the work products associated by the alpha manifests are
-- available within the practice, its base practices, or base kernels.
alphaManifest->forAll (am | self.allAlphas ()->includes (am.alpha) and
    self.allWorkProducts ()->includes (am.workProduct)
    and
-- The alphas have unique names within the practice.
self.workProduct->forAll (wp1, wp2 | wp1 <> wp2 implies wp1.name <> wp2.name)
```

Additional Operations

```
-- All the alphas available within the practice, its base practices, or base
-- kernels.
Practice::allAlphas () : set(Alpha)
alpha->union (basePractice->collect (bp | bp.allAlphas () ->union (baseKernel-
>collect (bk | bk.allAlphas () ) ) )
-- All the work products available within the practice, its base practices, or
-- base kernels.
Practice::allWorkProducts () : set(WorkProduct)
workProduct->union (basePractice->collect (bp | bp.workProduct () )
```

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 9.4 for the definition of composition).

Simple practices may contain only alphas and work products. In subsequent layers, additional properties will be added to the practice construct.

9.3.2.5 WorkProduct

Package: Layer2-PracticeAndAlpha

Description

A work product is an artifact of value and relevance for a software engineering endeavor.

Generalizations

N/A

Attributes

name : String [1]	The name of the work product.
icon : GraphicalElement [0..1]	The icon to be used when presenting the work product.
briefDescription : String [1]	A short description of what the work product is.
content : String [1]	The content of the work product.
levelOfDetail : String [1]	A description of how detailed the description of the work product should be.

Associations

levelOfCompleteness: StateGraph [0..1] The state graph contained by the work product to describe its states.

Invariant

true

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, and during its evolution it may have reached different states of completeness, like draft, outline, complete, and approved.

9.3.3 Layer3-CompletePractice

The intention of layer 3 is to provide additional elements to deal with more advanced practices. The elements and their relationships are presented in the diagrams shown in Figure 16, Figure 17, and Figure 18. A detailed definition of each of the elements is found below.

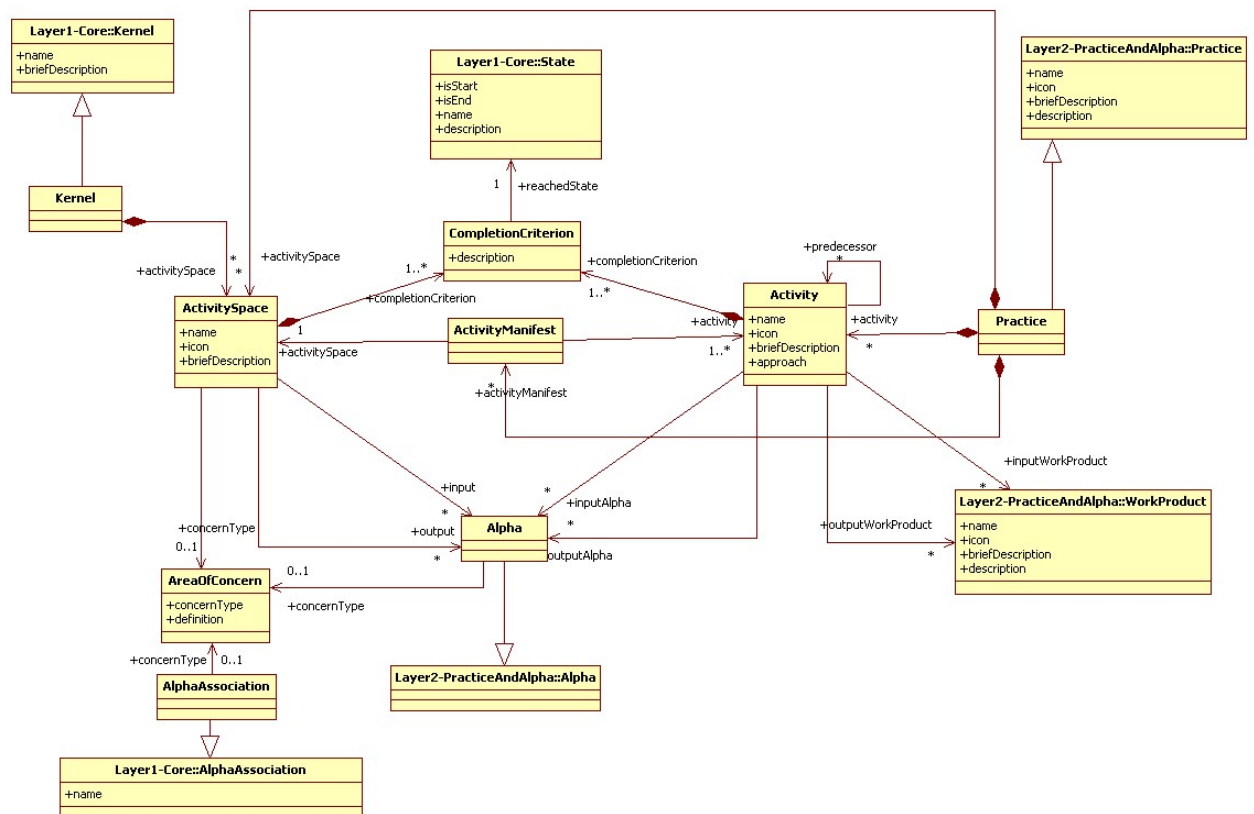


Figure 16 – Layer 3 Activity Space and Activity elements

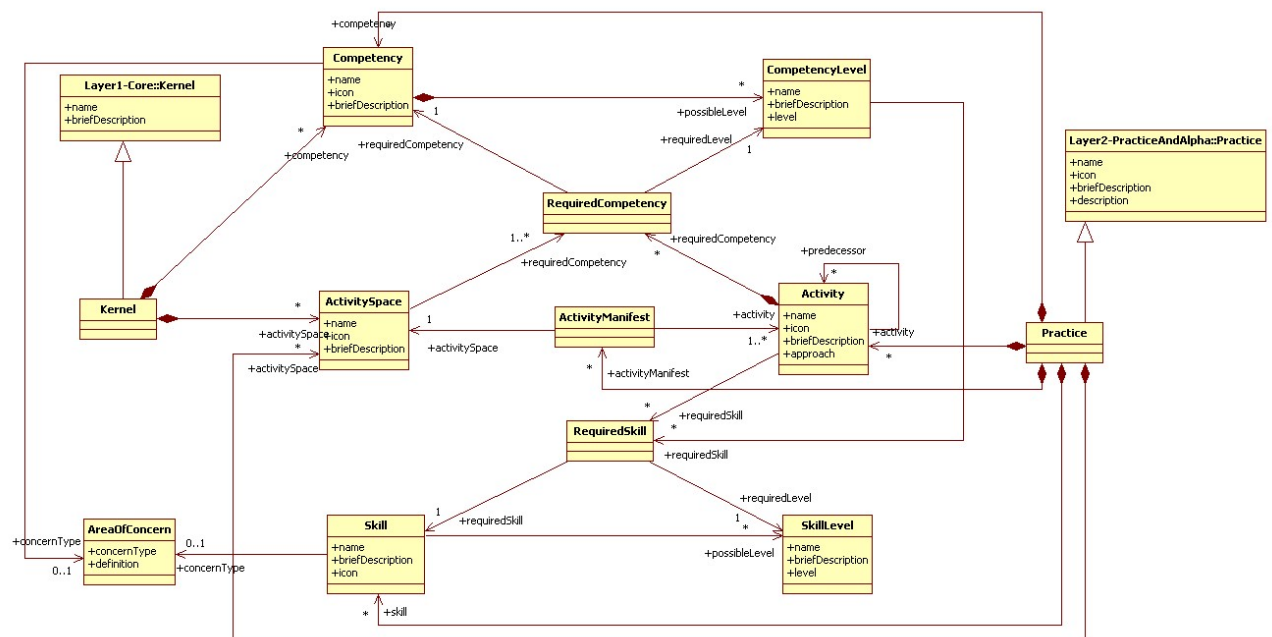


Figure 17 – Layer 3 Competency and Skill elements

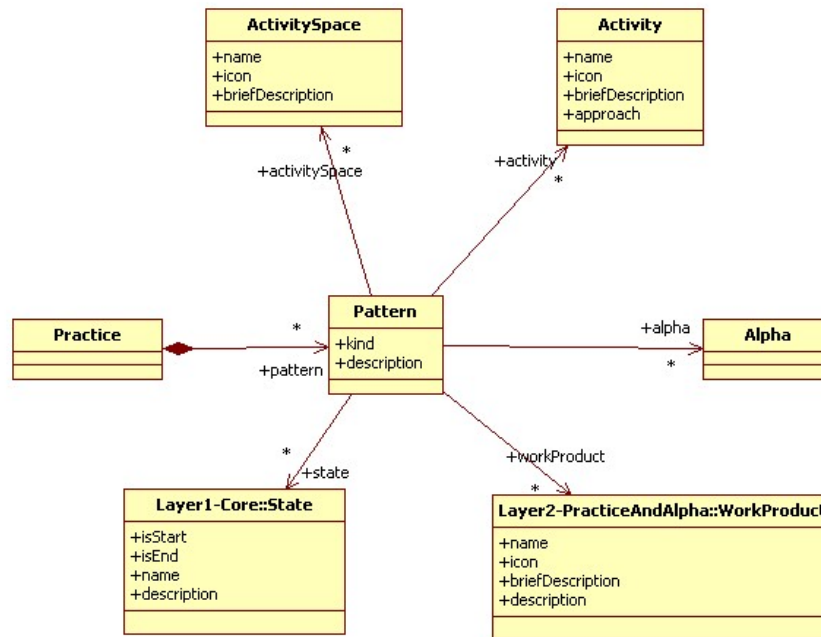


Figure 18 – Layer 3 Pattern elements

9.3.3.1 Activity

Package: Layer3-CompletePractice

Description

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

Generalizations

N/A

Attributes

name : String [1]	The name of the activity.
icon : GraphicalElement [1]	The icon to be used when presenting the activity.
briefDescription : String [1]	A short description of what the activity is.
approach : String [1..*]	Different approaches to accomplish the activity.

Associations

name : String [1]	The name of the activity.
completionCriterion : CompletionCriterion [1..*]	A collection of completion criteria that have to be fulfilled for considering the activity completed.
requiredCompetency : RequiredCompetency [*]	A collection of competencies required for completing this activity successfully.
requiredSkill : RequiredSkill [*]	A collection of skills 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

outputWorkProduct : WorkProduct [*]	to start this activity. A collection of Work Products that will be present when this activity is completed successfully.
predecessor : Activity [*]	A collection of Activities supposed to precede this Activity.

Invariant

```
-- Each completion criterion must refer to a state of an output alpha of the
-- activity.
self.completionCriterion->forAll (c | self.outputAlpha->exists (a |
a.stateGraph.state->includes(c.reachedState)))
and
-- The required skills of the activity should be part of the possible skills of
-- the activity's required competencies.
self.requiredSkill->forAll(rs | self.requiredCompetency->exists(rc |
rc.requiredCompetency->exists(pl | pl.requiredSkill.includes(rs)))
```

Semantics

An activity describes some work to be performed. It can take alphas or work products as input to the work, and alphas or work products may be created or updated during the activity. However, it is not defined when these have been created or updated; only that this has been done when the activity is completed.

The activity is considered completed if all its completion criteria are fulfilled. However, it is not specified that this has to happen due to performing this activity. The activity is thus also considered completed if all completion criteria are fulfilled for other reasons.

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 have predecessors which are recommended to be completed before the activity can be completed as well. However, this association is just considered as a hint to the performer(s) of the activity. As stated above, the activity is considered completed if all completion criteria are fulfilled, even if some predecessor is not completed for any reason.

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

9.3.3.2 ActivityManifest

Package: Layer3-CompletePractice

Description

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

Generalizations

N/A

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

Activity manifest represents a tri-nary relationship. It states which activities are bound to which activity space in a practice.

9.3.3.3 ActivitySpace

Package: Layer3-CompletePractice

Description

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

Generalizations

N/A

Attributes

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

Associations

requiredCompetency : RequiredCompetency [1..*]	A collection of competencies and competency levels required to be successful in fulfilling the objectives of this activity space.
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.
concernType : AreaOfConcern [0..1]	The area of concern this activity space belongs to.

Invariant

```
-- Each completion criterion must refer to a state of an output alpha of the
-- activity space.
self.completionCriterion->forAll (c | self.output->exists (a |
a.stateGraph.state->includes(c.reachedState))
```

Semantics

An activity space is a high-level abstraction of 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.

9.3.3.4 Alpha

Package: Layer3-CompletePractice

Description

The alpha construct is extended with properties for being used as input to and output from activities and activity spaces,

and for having an area of concern.

Generalizations

Layer2::Alpha

Attributes

N/A

Associations

concernType : AreaOfConcern [0..1] The area of concern the alpha belongs to.

Invariant

true

Semantics

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 may belong to an area of concern.

9.3.3.5 AlphaAssociation

Package: Layer3-CompletePractice

Description

The alpha association construct is extended with properties for having an area of concern.

Generalizations

Layer1::AlphaAssociation

Attributes

N/A

Associations

concernType : AreaOfConcern [0..1] The area of concern the alpha association belongs to.

Invariant

true

Semantics

An alpha association may belong to an area of concern.

9.3.3.6 AreaOfConcern

Package: Layer3-CompletePractice

Description

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 main areas of concern.

Generalizations

N/A

Attributes

concernType : String [1]	The type of the area of concern.
definition : String [1]	A description of the area of concern.
icon : GraphicalElement [1]	The icon to be used when presenting this area of concern.

Associations

N/A

Invariant

true

Semantics

Area of concern is a grouping facility to organize the elements in kernels and practices. They provide an overview on different aspects of software engineering endeavors, but do not imply any fixed semantics.

As already described in Section 8.1.3 there are three main areas of concern that software engineering endeavors have to pay special attention to:

- Customer space – in every software engineering endeavor, there are stakeholders to satisfy. These have needs, problems to solve, and money to spend on solving them.
- Solution space – on the way to executable software, we need to consider the specification and ensure that the implementation meets requirements. The software needs to be thoroughly tested and verified before we can hand it over to the end-users.
- Endeavor space – there is work to be done and we need a team to do it. They will likely need some direction and support. Work needs to be planned and progress must be monitored.

9.3.3.7 Competency

Package: Layer3-CompletePractice

Description

A competency describes a capability to do a certain job.

Generalizations

N/A

Attributes

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

Associations

possibleLevel : CompetencyLevel [*]	A collection of levels defined for this competency.
concernType : AreaOfConcern [0..1]	The area of concern the competency belongs to.

Invariant

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

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”.

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*.

9.3.3.8 CompetencyLevel

Package: Layer3-CompletePractice

Description

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

Generalizations

N/A

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

requiredSkill : RequiredSkill [*]	The skills required at this level.
-----------------------------------	------------------------------------

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 (such as “basic”, “advanced”, or “excellent”).

Particular skills can be associated with a Competency level if some particular level in that skill is required to reach this Competency level. For example there may be no particular skills be associated with the “basic” level of “Developer”, but on an “advanced” level some skills in communicating in English are required to be able to read and write code comments. Most likely, particular skills are associated with Competency levels defined in a Practice, but not in the Kernel.

9.3.3.9 CompletionCriterion

Package: Layer3-CompletePractice

Description

A completion criterion defines which state an alpha or work product should have reached in order to consider an activity or activity space completed.

Generalizations

N/A

Attributes

description : String [1] A description of the criterion which is to be reached at the target state.

Associations

reachedState : State [1] A state to be reached.

Invariant

true

Semantics

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

9.3.3.10 Kernel

Package: Layer3-CompletePractice

Description

The kernel construct is extended with properties for containing activity spaces and competencies.

Generalizations

Layer1::Kernel

Attributes

N/A

Associations

competency : Competency [*] A collection of competencies defined in the kernel.
activitySpace : ActivitySpace [*] A collection of activity spaces defined in the kernel.

Invariant

```
-- All input and out alphas of the activity spaces are available within the
-- kernel or its base kernels.
activitySpace->forAll (as | as.input->forAll (i | self.allAlphas ()->includes (i)
)
    and
as.output->forAll (o | self.allAlphas ()->includes (o) ) )
    and
-- The reached states of the activity spaces' completions criteria are possible
-- states of the activity spaces' output alphas.
activitySpace->forAll (as | as.completionCriterion.reachedState (rs |
as.output.stateGraph.state->includes (rs)))
    and
-- 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)))
    and
-- The competencies within the kernel have unique names.
self.competency->forAll (c1, c2 | c1 <> c2 implies c1.name <> c2.name)
    and
-- The activity spaces within the kernel have unique names.
self.activitySpace->forAll (a1, a2 | a1 <> a2 implies a1.name <> a2.name)
```

Additional Operations

```
-- All activity spaces within the kernel or its base kernels.
Kernel::allActivitySpaces () : set(ActivitySpace)
activitySpace->union (baseKernel->collect (bk | bk.allActivitySpaces () ) )
-- All competencies within the kernel or its base kernels.
Kernel::allCompetencies () : set(ActivitySpace)
competency->union (baseKernel->collect (bk | bk.allCompetencies () ) )
```

Semantics

A kernel can contain not only alpha and alpha associations, but also activity spaces and competencies.

9.3.3.11 Pattern

Package: Layer3-CompletePractice

Description

A pattern is a definition of a pragmatic relationship among elements in a practice.

Generalizations

N/A

Attributes

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

Associations

activity : Activity [*]	The activities participating in the pattern.
activitySpace : ActivitySpace [*]	The activity spaces participating in the pattern.
alpha : Alpha [*]	The alphas participating in the pattern.
workProduct : WorkProduct [*]	The work products participating in the pattern.
state : State [*]	The states participating in the pattern.

Invariant

true

Semantics

Pattern is a general mechanism for defining a structure in a practice. 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 the practice. 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.

9.3.3.12 Practice

Package: Layer3-CompletePractice

Description

The practice construct is extended with properties for containing activities, activity spaces, activity manifests, and competencies.

Generalizations

Layer2::Practice

Attributes

N/A

Associations

activity : Activity [*]	A collection of activities defined in this practice.
activitySpace : ActivitySpace [*]	A collection of activity spaces defined in this practice.
activityManifest : ActivityManifest [*]	A collection of activity manifests defined in this practice.
competency : Competency [*]	A collection of competencies defined in this practice.
skill : Skill [*]	A collection of skills defined in this practice.
pattern : Pattern [*]	A collection of patterns defined in this practice.

Invariant

```
-- The predecessors of an activity are available within the practice, its base
-- practices, or its base kernels.
activity->forAll (a | a.predecessor->forAll (p | self.allActivities ()->includes
(p)) )
    and
-- The activities and the activity spaces associated by the activity manifests of
-- the practice are all available within the practice, its base practices, or its
-- base kernels.
activityManifest->forAll (am | am.activity->forAll (a | self.allActivities ()-
>includes (a) ) and self.allActivitySpaces ()->includes (am.activitySpace) )
    and
-- All activities' input and output work products and input and output alphas are
-- available within the practice, its base practices, or its base kernels.
activity->forAll (a | a.inputWorkProduct->forAll (iwp | self.allWorkProducts ()-
>includes (iwp))
    and
a.outputWorkProduct->forAll (owp | self.allWorkProducts ()->includes (owp))
    and
a.inputAlpha->forAll (ia | self.allAlphas ()->includes (ia))
    and
a.outputAlpha->forAll (oa | self.allAlphas ()->includes (oa)))
    and
-- All reached states of the activities' completion criteria are included in the
-- activities' output alphas possible states.
activity->forAll (a | a.completionCriterion.reachedState (rs |
a.outputAlpha.stateGraph.state->includes (rs)))
    and
-- The activities' required competencies are available within the practice, its
-- base practices, or its base kernels.
activity.requiredCompetency->forAll (rc | self.allCompetencies ()->includes (rc))
    and
-- The activities' required skills are available within the practice, its base
-- practices, or its base kernels.
activity.requiredSkill->forAll (rs | self.allSkills ()->includes (rs))
    and
-- The patterns' activity spaces, activities, alphas, and work products are
-- available within the practice, its base practices, or base kernels.
pattern->forAll (p | p.activitySpace->forAll (as | self.allActivitySpaces ()-
>includes (as))
    and
p.activity->forAll (a | self.allActivities ()->includes (a))
    and
p.alpha->forAll (a | self.allAlphas ()->includes (a))
    and
p.workProduct->forAll (wp | self.allWorkProducts ()->includes (wp)) )
    and
-- All activities within the practice have unique names.
self.activity->forAll (a1, a2 | a1 <> a2 implies a1.name <> a2.name)
    and
```

```
-- All activity spaces within the practice have unique names.
self.activitySpace->forall (as1, as2 | as1 <> as2 implies as1.name <> as2.name)
    and
-- All competencies within the practice have unique names.
self.competency->forall (c1, c2 | c1 <> c2 implies c1.name <> c2.name)
    and
-- All skills within the practice have unique names.
self.skill->forall (s1, s2 | s1 <> s2 implies s1.name <> s2.name)
```

Additional Operations

```
-- All activity spaces within the practice, its base practices, and base kernels.
Practice::allActivitySpaces () : set(ActivitySpace)
activitySpaces->union (basePractice->collect (bp | bp.allActivitySpaces () ) -
>union (baseKernel->collect (bk | bk.allActivitySpaces () ) ) )
-- All activities within the practice, its base practices, and base kernels.
Practice::allActivites () : set(Activity)
activity->union (basePractice->collect (bp | bp.allActivities () ) )
-- All competencies within the practice, its base practices, and base kernels.
Practice::allCompetencies () : set(Competency)
competency->union (basePractice->collect (bp | bp.allCompetencies () ) )
-- All skills within the practice, its base practices, and base kernels.
Practice::allSkills () : set(Skill)
skill->union (basePractice->collect (bp | bp.allSkills () ) )
```

Semantics

A practice could contain not only alphas, alpha associations, alpha manifests, and work products, but also activities, activity spaces, activity manifests, competencies, and skills.

9.3.3.13 RequiredCompetency

Package: Layer3-CompletePractice

Description

A required competency states which competency level is needed to perform an activity.

Generalizations

N/A

Attributes

N/A

Associations

requiredLevel : CompetencyLevel [1] The required level.
requiredCompetency : Competency [1] The required competency.

Invariant

```
-- The competency level is included the competency definition.
self.requiredCompetency.possibleLevel->includes(self.requiredLevel)
```

Semantics

An activity fills an activity space that requires a competency. The specific competency level within that competency the particular activity requires is expressed by a required competency.

9.3.3.14 RequiredSkill

Package: Layer3-CompletePractice

Description

A required skill states which skill level is needed to perform an activity.

Generalizations

N/A

Attributes

N/A

Associations

requiredLevel : SkillLevel [1]	The required level.
requiredSkill : Skill [1]	The required skill.

Invariant

```
-- The competency level is included the competency definition.  
self.requiredSkill.possibleLevel->includes(self.requiredLevel)
```

Semantics

To perform an activity successfully, a collection of skills is required. For each of these skills the necessary level is stated.

9.3.3.15 Skill

Package: Layer3-CompletePractice

Description

A skill describes the ability to use one's knowledge effectively in execution.

Generalizations

N/A

Attributes

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

Associations

possibleLevel : SkillLevel [*]	A collection of levels defined for this skill.
concernType : AreaOfConcern [0..1]	The area of concern the skill belongs to.

Invariant

```
-- The possible skill levels are distinct  
self.possibleLevel->forAll (p11, p12 | p11 <> p12 implies p11.level <> p12.level)
```

Semantics

A skill is a learned power of doing something effectively. In contrast to Competencies, a skill is more tangible and can possibly be proven by some certificate. Examples for skills include “Communicating in English”, “Programming in Java”, or “Using version control systems”.

A skill defines a sequence of skill levels. Typically, the level ranges from *0 – no skill* to *5 – excellent*.

9.3.3.16 SkillLevel

Package: Layer3-CompletePractice

Description

A skill level defines a level of skill someone is in a subject.

Generalizations

N/A

Attributes

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

Associations

N/A

Invariant

true

Semantics

Skill levels are used to create a range of abilities from poor to excellent skills. While a skill describes what abilities are needed, such as “Programming in Java” or “Communicating in English,” a skill level adds a qualitative grading to them, such as “beginner,” “average,” or “excellent.”

9.3.4 Layer4-MethodAndLibrary

The intention of layer 4 is to provide facilities to compose methods out of practices. The elements and their relationships are presented in the diagram shown in Figure 19. A detailed definition of each of the elements is found below.

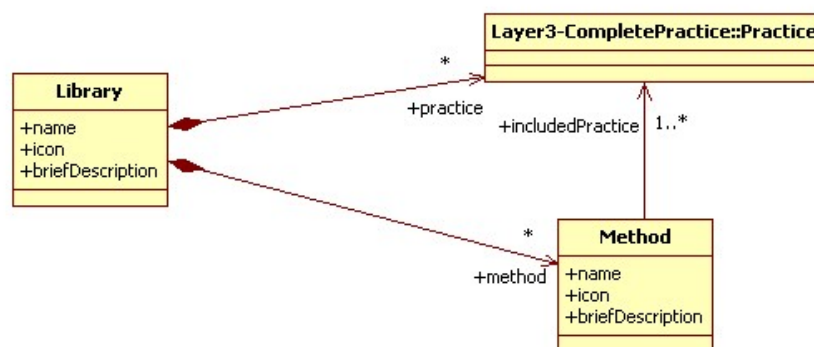


Figure 19 – Layer 4 elements

9.3.4.1 Library

Package: Layer4-MethodAndLibrary

Description

A library includes a collection of practices and methods.

Generalizations

N/A

Attributes

name : String [1]	The name of the library.
icon : GraphicalElement [1]	The icon to be used when presenting the library itself.
briefDescription : String [1]	A short description of what the library captures.

Associations

practice : Practice [*]	The practices contained in the library.
method : Method [*]	The methods contained in the library.

Invariant

```
-- The practices included in a method are available within the library.
method->includedPractice->forAll (ip | self.practice->includes (ip))
    and
-- The methods have unique names.
method->forAll(m1, m2 | m1 <> m2 implies m1.name <> m2.name)
    and
-- The practices have unique names.
practice->forAll(p1, p2 | p1 <> p2 implies p1.name <> p2.name)
```

Semantics

A library contains elements relevant for a specific subject or area of knowledge, like *software development*. The elements contained in the library are practices and methods to be used in that area.

9.3.4.2 Method

Package: Layer4-MethodAndLibrary

Description

A Method describes how an endeavor is run.

Generalizations

N/A

Attributes

name : String [1]	The name of the method.
icon : GraphicalElement [1]	The icon to be used when presenting the method.
briefDescription : String [1]	A short description of the method.

Associations

includedPractice : Practice [1..*]	The composed practices making up the method.
------------------------------------	--

Invariant

true

Semantics

A method is a composition of practices forming a (at the desired level of abstraction) complete 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. Note that a method does not add any substantial information to a composition of practices, but only a name and a description. The description is supposed to explain for which purpose and level of abstraction the composition of practices is suitable.

Different methods, i.e. different compositions of practices, are created addressing:

- A particular size or style of a software engineering endeavor.
- A particular style or type of development.
- A particular risk or set of circumstances.

Pre-built methods, i.e. methods provided in a library and not developed by the team itself composing a collection of practices, provide a set of “starter packs” for teams wishing to adopt a particular methodology or approach. These methods can be updated to describe how a team would like to apply them; they can also be composed with additional practices to specialize a method even further.

When the endeavor is initiated, instances of the alphas and work products defined in the selected method are created corresponding to the actual occurrences the team is working with. These instances change states based on the team's actions. A more thorough description of the performance of a method is found in Section 9.5.

9.4 Composition

9.4.1 Introduction

The main purpose of composing practices is to define a method. This method could be used in endeavors developing software, although other purposes and domains are also possible.

In this section, we present what it means to compose two practices to form a new practice. This practice may in its turn be composed with other practices and eventually the result can be used as a method describing the performance of a software engineering endeavor.

First we define a simple algebra for composition of graphs of instances and links of classes and associations in the metamodel. Then, we use this algebra to define what we mean by composition of practices, i.e. merging two graphs of instances of the constructs in the kernel language, or instances of the classes in the metamodel. We also provide some examples of practice composition.

9.4.2 Graph Algebra

The algebra consists of three operations that each operate on instance models of the metamodel, i.e. graphs of instances and links.

The constructs are:

- $P(x)$ – variable definition
- $P[x \leftarrow Y]$ – renaming
- $P + Q$ – merge

where P and Q are graphs of instances and links. Each of these operations is described below.

9.4.2.1 Variable Definition

A variable definition, $P(x)$, defines a named variable, x , (a placeholder) in a graph, P . The variable is to be filled, i.e. merged, either with an instance of a class in the metamodel, like an activity or an alpha, or with a link between two instances, i.e. an instance of an association in the metamodel. The variable may occur in several places within P , and all of them will be merged with the same instance.

In the trivial case, the variable is independent of the other elements in **P**, i.e. the variable has no relationships to other elements in **P**. However, in general the variable is inserted into the structure of **P**. The different options are described below.

9.4.2.1.1 Add Instance

The variable defines where an instance of a class in the metamodel is to be inserted into **P**. The variable may have links to other instances.

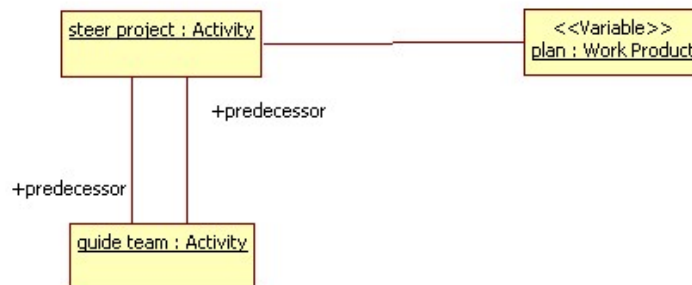


Figure 20 – A variable called “plan” which is to be of type Work Product is added and linked to an already existing Activity

9.4.2.1.2 Add Link

A second possibility is that the variable defines a link of an association in the metamodel between two instances in **P**, i.e. the variable is to be merged with a link in the other graph. Merging a variable with a link, which has the same name and type as the variable, will result in a graph where the variable has been replaced by the corresponding link.

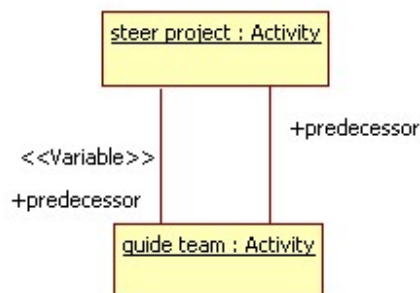


Figure 21 – A variable is a link between two existing instances

9.4.2.1.3 Insert Instance on Link

A third option is to insert a variable representing an instance of a class onto a link, i.e. to insert the variable between two linked instances. This is accomplished by replacing the link with two links and a variable representing the instance. The two links must be of the same type and have the same name as the original link.

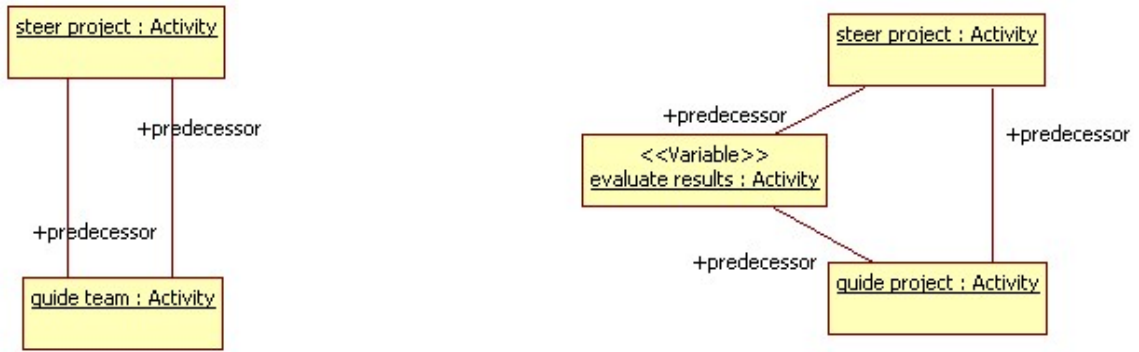


Figure 22 – A variable is inserted onto a link defined in the left graph resulting in the right graph

9.4.2.2 Renaming

A rename operation, $P [x \leftarrow y]$, replaces all occurrences of the name ' x ' within P with the name ' y ' regardless of where in P the name is used.

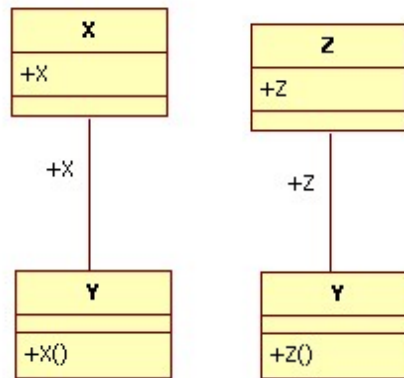


Figure 23 – After applying the $[x \leftarrow z]$ operation on the structure to the left, all occurrences of “x” in the diagram to the left are replaced with “z” in the diagram to the right

9.4.2.3 Merge

A merge of two graphs, $A + B$, results in a new graph where instances of the same type and with the same name are merged into one instance containing the composition of the two instances' contents.

```

A + B =
let
  ac = graphCopy (A)
  bc = graphCopy (B)
  ai = allInstances (ac)
  bi = allInstances (bc)
in
   $\forall y \in bi . \exists x \in ai : x \equiv y \rightarrow \text{merge} (x, y, ai)$ 

merge (a, b, c) =
 $\forall y \in b.\text{contents} . \exists x \in a.\text{contents} : x \equiv y \rightarrow \text{manuallyMerge} (x, y)$ 
^
 $\forall y \in b.\text{contents} . \neg \exists x \in a.\text{contents} : x \equiv y \rightarrow \text{addElement} (a.\text{contents}, y)$ 

```



```

^
∀y ∈ b.relationships . ∃x ∈ a.relationships : x ≡ y → DO NOTHING
^
∀y ∈ b.relationships . ¬∃x ∈ a.relationships : x ≡ y →
let
    ob = y.otherEnd (b)
in
    ∃z ∈ c : ob ≡ z → addElement (a.relationships, mk-Link (y.type, y.name, a,
z))
    ¬∃z ∈ c : ob ≡ z → addElement (a.relationships, mk-Link (y.type, y.name,
a, ob)

```

```

graphCopy (g) =
return a copy of the graph (instances and links) reachable from g

```

```

allInstances (g) =
return a set of all instances reachable from g

```

```

manuallyMerge (a, b) =
the merge of primitive types, like strings and icons, is not predefined and must
be performed manually

```

```

addElement (s, e) =
add the element e to the set s

```

The operation “mk-“ is used to create a new instance of a metaclass or a metaassociation.

9.4.3 Required Primitive Operations

In the metamodel, the following operations can be applied to all elements, i.e. they are defined in each class and each association:

- **type** – returns the type (class) of the element
- **name** – returns the name of the element

The following operations can be applied to all instances of classes, i.e. they are defined in each class:

- **contents** – returns all contained elements, like operations and attributes
- **relationships** – returns all outgoing relationships

The following operation can be applied to all links, i.e. it is defined in each association:

- **otherEnd (o)** – returns the element connected to the link which is opposite to the element, o, also connected to the link

9.4.4 Additional Definitions in the Algebra

The merge operation is both commutative and associative:

```

P + Q + R = (P + Q) + R      a renaming operation may have to be performed
P + Q = Q + P                a renaming operation may have to be performed

```

The following define obvious abbreviations that may be used to reduce the size of expressions in the algebra.

```

P (x, y) = (P (x)) (y)
P [ x <- y, u <- v ] = (P [ x <- y ]) [ u <- v ]

```

Equivalence (\equiv) between two elements means:

- the elements are instances of classes or links of associations in the metamodel – the two elements have the same name, the same type, and their contained elements are equivalent, but the two elements are distinct
- the elements are texts, integers, unlimited naturals, or icons – the two elements represent the same value, i.e. they are treated to be the same

9.4.5 Composition of Practices

Now, we can define the meaning of composing two practices using the algebra presented above. We start by defining the compose operation. Then, we provide a simple example when composing two practices.

Note, this operation can be used to compose any elements of the same type being the root nodes of graphs, like practices and kernels.

9.4.5.1 Definition of the Compose Operation

A composition of two elements (like practices), `compose (P, Q, aName)`, results in a new element of the same type as the two named `aName`. The content of the new element is a merge of the elements contained in the two graphs defined by the two original elements.

```
compose (x, y, n) =  
x.type = y.type → ( x [x.name <- n] ) + ( y [y.name <- n] )
```

9.4.5.2 Applying the Compose Operation

The composition of two practices is done in a sequence of steps.

We start (if needed) by introducing variables, i.e. placeholders, into the structure of one or both of the two practices, where elements of the other practice are to be inserted. This may introduce variables as well as new links into the structure of the practice. The name of a variable should be the same as the name of the element in the other practice to be inserted into the variable.

Since the merge is based on equivalence, i.e. the names and the types should be the same, we have to ensure that no elements of the same type have the same name and should not be merged. We therefore continue (if necessary) by renaming all elements that have the same name but are not to be merged. Furthermore, elements that should be merged are renamed so they have the same name. We also have to consider the variables and the elements to be inserted into (merged with) these variables.

Finally, the composition is made as defined by the `compose (.)` function above. The two input practices are renamed to the provided name and then the merge of the two is performed. Note, the two original practices are not affected by the composition.

9.4.6 Examples

9.4.6.1 Simple Composition

In this example, we have two practices: *Iterative Planning* and *Iterative Assessment*. (Neither of them fulfills the definition of being a practice, but they are sufficient for the example.) The composition of the two will result in a third practice: *Iterative Development*. Obviously, the result in this example is not the full Iterative Development practice. Here, it is only used to exemplify the composition of two practices.

In this example, we have excluded alphas and activity spaces and therefore also activity manifests and alpha manifests. We assume that the former are defined in a kernel and hence will be the same in both practices and obviously be merged. The latter, i.e. the manifest instances, will, by definition, be unique for each practice and hence will not be merged.

Iterative Planning – consists of two activities: *Agree Iteration* and *Guide Team*. Each of them being the predecessor of the other. *Agree Iteration* uses the *Iteration Plan* work product.

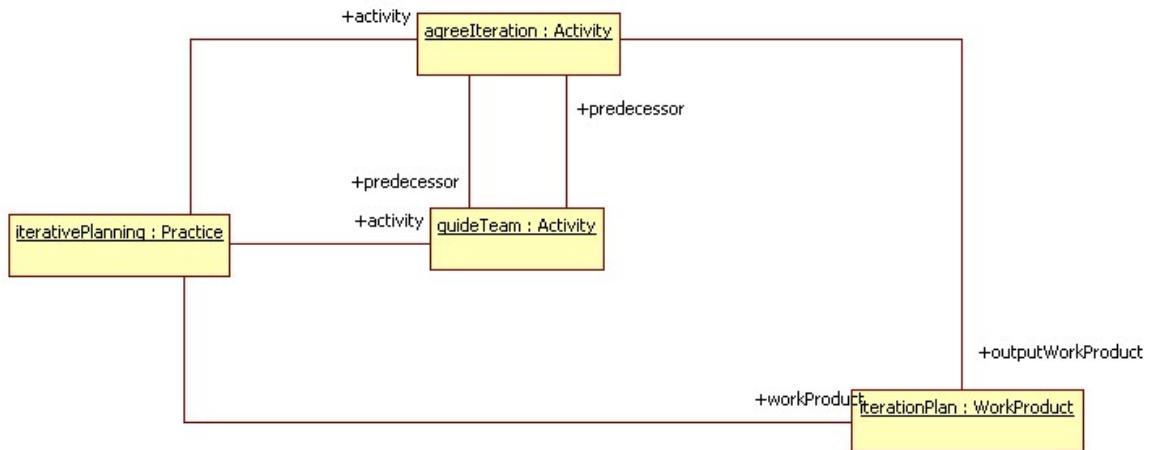


Figure 24 – The structure of the Iterative Planning practice

Iterative Assessment – consists of one activity: *Evaluate Results* and one work product: *Iteration Assessment*.

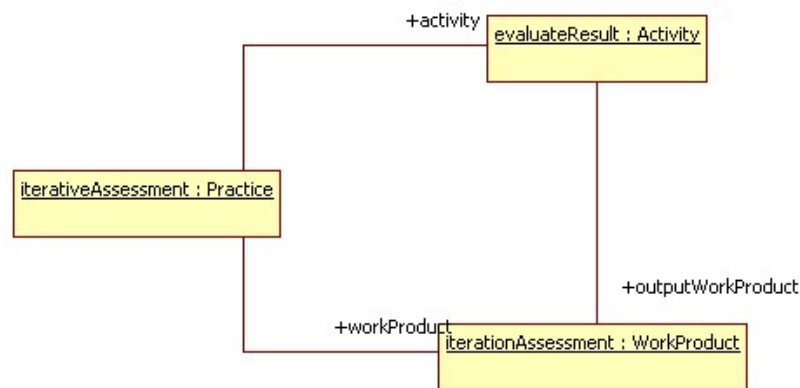


Figure 25 – The structure of the Iterative Assessment practice

The composition of the two practices is performed in two steps. First, we need to enable the insertion of the Iteration Assessment activity into the predecessor cycle defined in the Iteration Planning practice. If we do not do this, the Iteration Assessment activity will be performed independently of the other two activities. Second, the actual composition is made, which will merge elements with the same name and types, and result in a new practice.

We start by inserting a variable into the Iteration Planning practice. We call this variable Evaluate Results (the same as the name of the activity we are to insert) and it is of type Activity. The variable is inserted where the evaluation at the end of an iteration is to take place, namely on the Predecessor link from the Agree Iteration activity to the Guide Team Activity. (Formally, the link is replaced by two links of the same kind, directed in the same way as the original one, and with the inserted variable in between.)

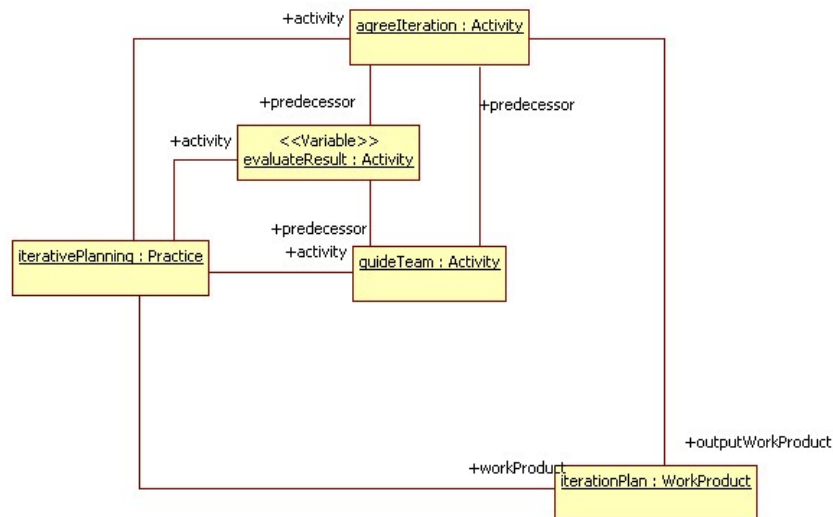


Figure 26 – A variable is inserted between Agree Iteration and Guide Team. The variable has the same name as the activity to be inserted

Now, we continue with the second step and do the actual composition of the two practices. First, the two practices are renamed during the composition; we call them Iterative Development. Then, all elements in these two practices with the same names and types are merged. In this case, the two practices have the same name and will be merged, and the variable Evaluate Results in the Iteration Planning practice and the activity with the same name in the Iteration Assessment practice will also be merged.

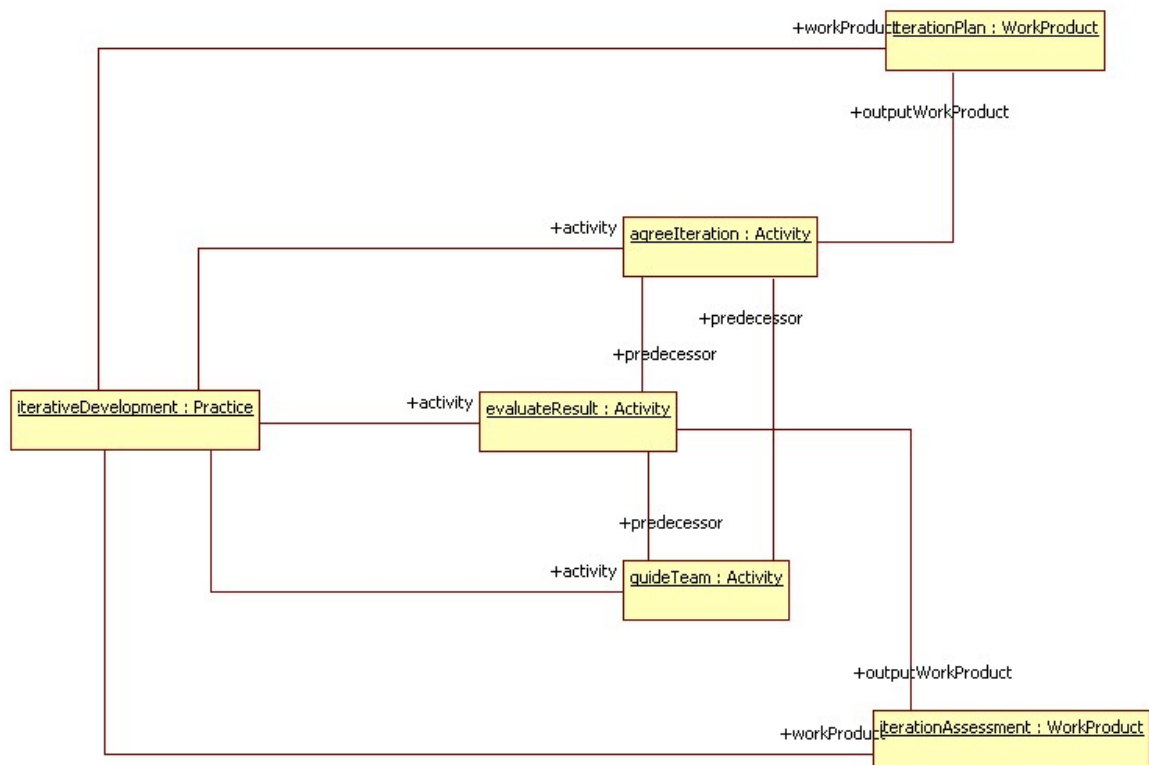


Figure 27 – The structure of the resulting practice

Hence, the formal expression for the composition is:

`compose (Iterative Planning, Iterative Assessment, ITERATIVE DEVELOPMENT)`

9.5 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.

9.5.1 Domain classes

9.5.1.1 Recap of Meta-modeling Levels

As stated in Section 9.2.1, 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 9.3, 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.

9.5.1.2 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 9.3, 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 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.

9.5.1.3 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 28 – The Essence language framework.

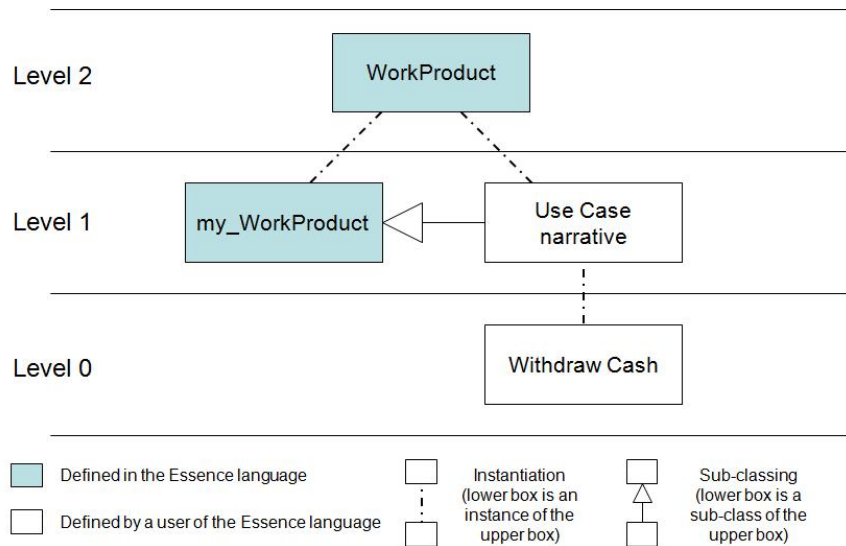


Figure 28 – The Essence language framework

9.5.1.3.1 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

<code>instanceName : String [1]</code>	The name of an occurrence (e.g., Requirements for the XYZ Project)
<code>currentState : State [1]</code>	A pointer to the current State of an occurrence (e.g., to the state “Coherent”)
<code>myWorkProductInstances : my_WorkProduct [*]</code>	The set of WorkProducts this alpha is manifested by.
<code>mySubAlphaInstances : my_Alpha [0..*]</code>	A set of Sub-Alphas from AlphaContainment relationships.

9.5.1.3.2 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 : State [1]	A pointer to the current State of an occurrence (e.g., to the state “Not Started”)

9.5.1.3.3 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”)
myAlphaInstances : my_Alpha [*]	A pointer to the set of Alphas that this Activity is concerned with (either by using it as reference or doing work that will change its state).
myWorkProductInstances : my_WorkProduct[*]	A pointer to the set of Work Products used by this Activity.

9.5.2 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.

9.5.2.1 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 subAlpha 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 my_AlphaManifests defined for the my_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.

9.5.2.2 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 **Error! Reference source not found.** Here the most advanced state of Software System “XYZ” consistent with the checkpoints that have been met (shown as ticked) is “Useable”.

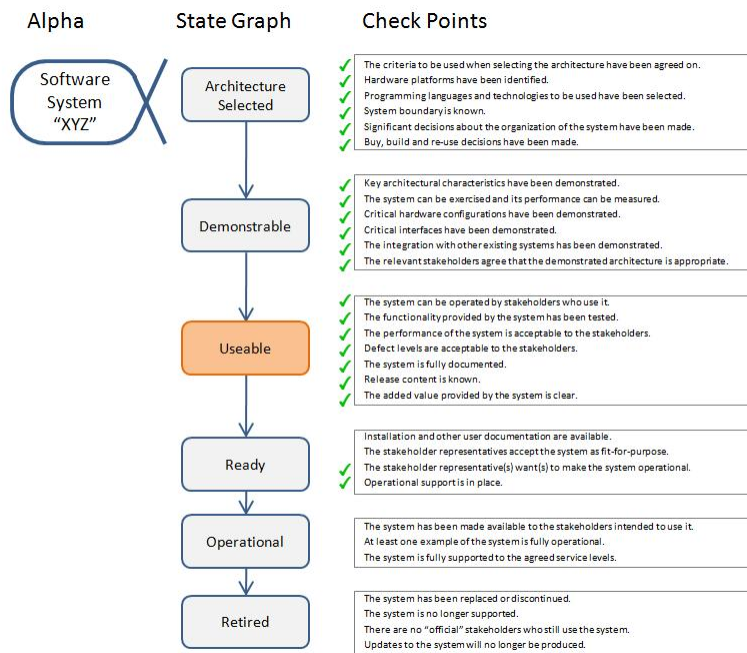


Figure 29 – Determination of State using Check Points

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.

9.5.2.3 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 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.

9.5.2.4 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.outgoingTransition = null) ∨ (s1 = s2)) {s1}
    else {s1} ∪ fullPath(s1.outgoingTransition.target, 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 9.5.1.3).

```
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})
```

9.6 Graphical Syntax

9.6.1 Specification Format

The concrete syntax of the language is organized in views. Each view provides notations for a subset of elements of the language. Views are defined and used independently from abstract syntax layers. For example, a view capable of representing elements from abstract syntax layers 1, 2 and 3 can be used to represent a language construct just containing elements from abstract syntax layers 1 and 2. The view is allowed to represent just a part of the whole language construct. In the same way, a view capable of representing just elements from abstract syntax layer 1 can also be used to represent (parts of) the same language construct. It is considered correct to define and use other views than the ones defined in this language specification.

The following views are defined in the graphical syntax:

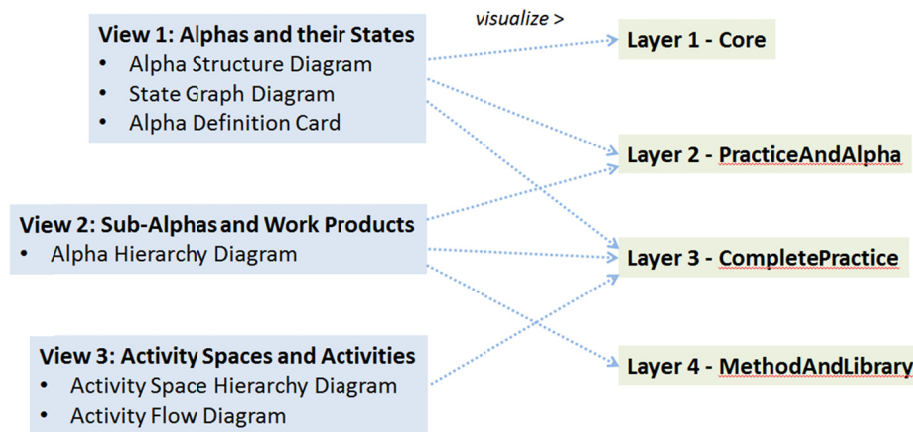


Figure 30 – View to layer mapping overview

Moreover, the graphical syntax of each construct to be visualized 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.

Given this, 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.

9.6.2 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.

9.6.3 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 31 – Example visualizations of the Alpha meta-class construct and its Software System type

9.6.4 View 1: Alphas and their States

The following sections define relevant symbols for View 1: Alphas and the States. This view is thereby capable of visualizing elements in abstract syntax layers 1, 2 and 3.

9.6.4.1 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 32 – Alpha symbol

Style Guidelines

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

Examples



Figure 33 – Software System Alpha

9.6.4.2 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.

name

Figure 34 – 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

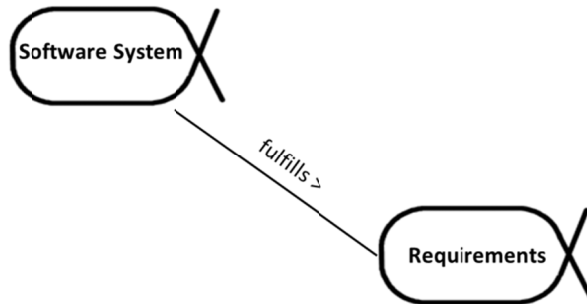


Figure 35 – Alpha Association between the Requirements Alpha and the Software System Alpha, read as: “The Software System fulfills the Requirements.”

9.6.4.3 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.

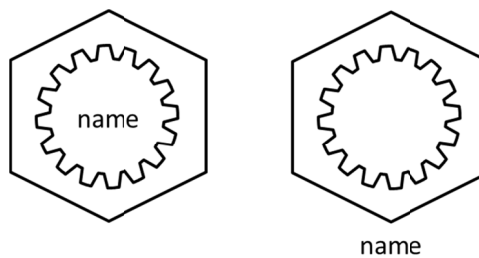


Figure 36 – Kernel symbol

Style Guidelines

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

Examples



Figure 37 – Kernel for Software Engineering

9.6.4.4 State

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

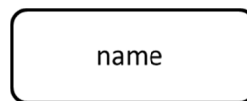


Figure 38 – State symbol

Style Guidelines

- Center the name of the State in boldface.

Examples



Figure 39 – Milestones Agreed State

9.6.4.5 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 40 – Transition

Examples

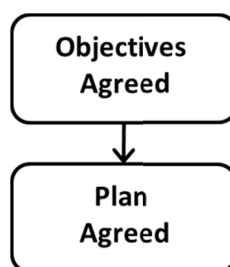


Figure 41 – Transition from the Objectives Agreed State to the Plan Agreed State

9.6.4.6 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.

9.6.4.6.1 Alpha Structure Diagram

Table 8 – Graphical nodes in Alpha Structure diagrams.

Node Type	Symbol	Reference
Alpha		Section 9.6.4.1 Alpha.

Table 9 – Graphical links in Alpha Structure diagrams.

Link Type	Symbol	Reference
Alpha Association		Section 9.6.4.2 Alpha Association.

Examples

Refer to kernel examples.

9.6.4.6.2 State Graph Diagram

Table 10 – Graphical nodes in State Graph diagrams.

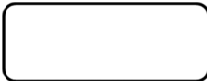

Node Type	Symbol	Reference
State		Section 9.6.4.4 State.

Table 11 – Graphical links in State Graph diagrams.

Link Type	Symbol	Reference
Transition		Section 9.6.4.5 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

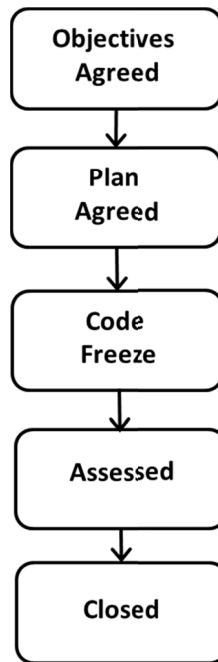


Figure 42 – State Graph example

9.6.4.7 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.

9.6.4.7.1 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:

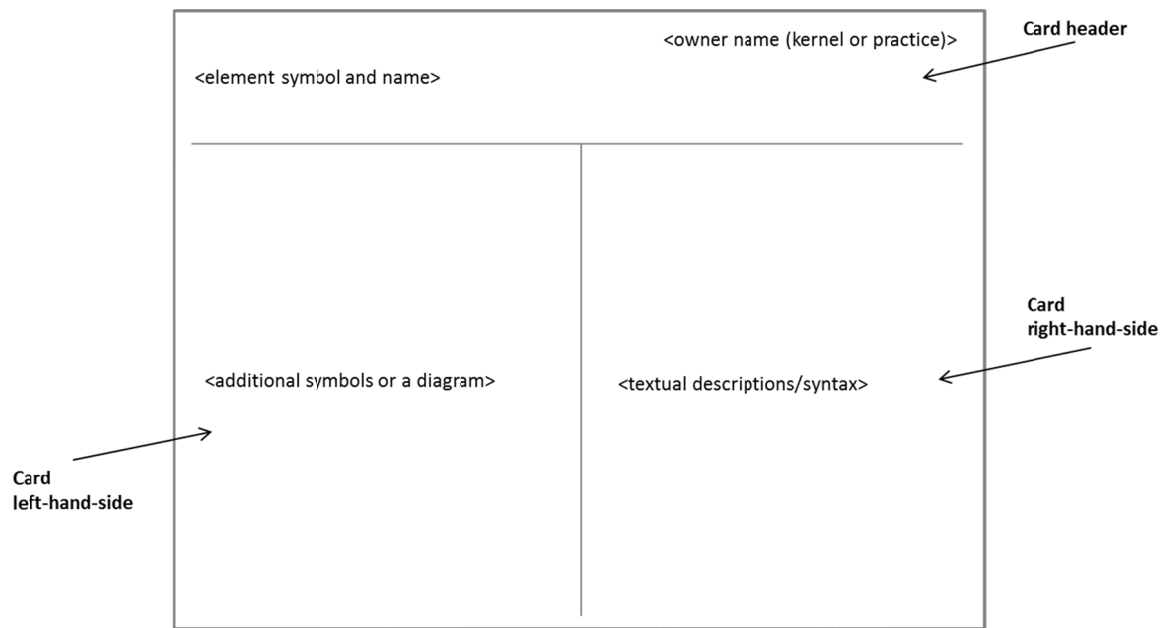


Figure 43 – A basic card anatomy to visualize an element

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.

9.6.4.7.2 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

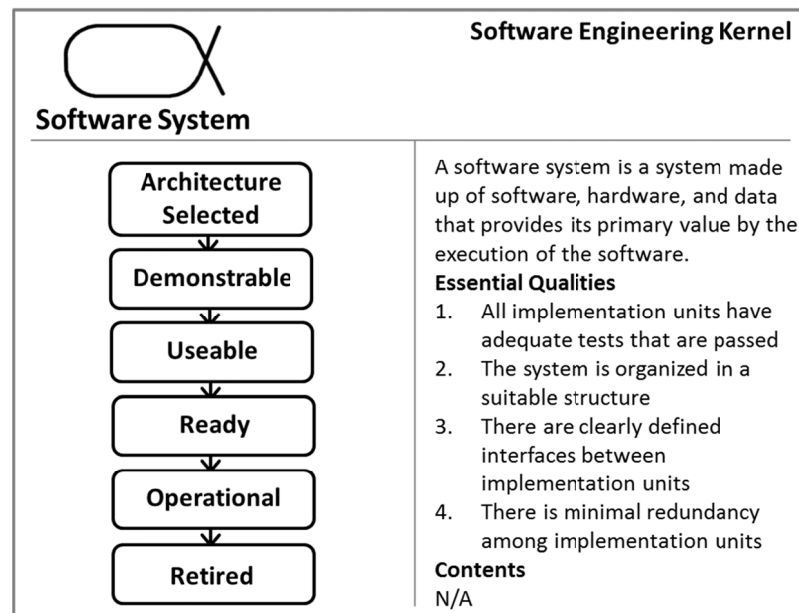


Figure 44 – Software System Alpha Definition Card

9.6.5 View 2: Sub-Alphas and Work Products

The following sections define relevant symbols for View 2: Sub-Alphas and Work Products. This view is thereby capable of visualizing elements in abstract syntax layers 2 and 3.

9.6.5.1 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:

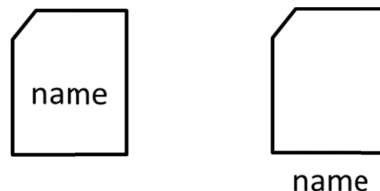


Figure 45 – Work Product symbol

Style Guidelines

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

Examples

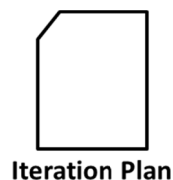


Figure 46 – Iteration Plan Work Product

9.6.5.2 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 47 – 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

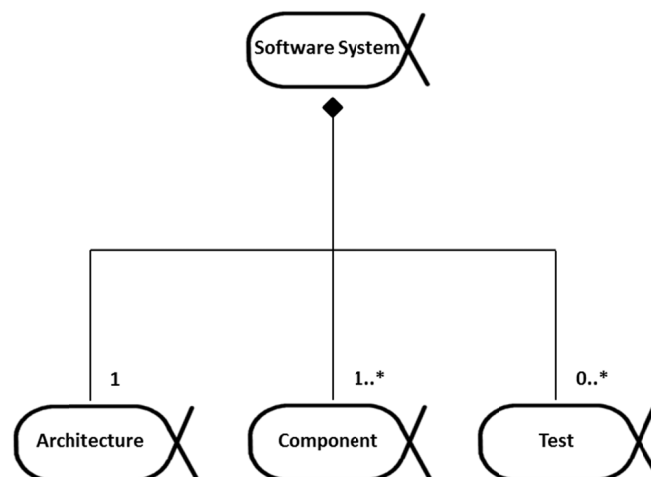


Figure 48 – Software System super-Alpha and three sub-Alphas: Architecture, Component, and Test with visualized multiplicities

9.6.5.3 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.

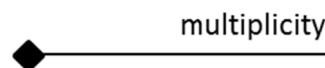


Figure 49 – Alpha Manifest symbol

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

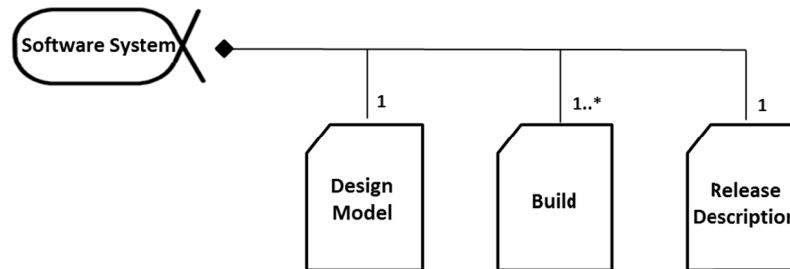


Figure 50 – Software System Alpha and three Work Products: Design Model, Build, and Release Description with visualized multiplicities

9.6.5.4 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.

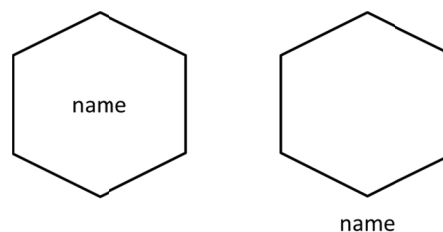


Figure 51 – Practice symbol

Style Guidelines

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

Examples



Figure 52 – Scrum Essentials Practice

9.6.5.5 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.

9.6.5.5.1 Alpha Hierarchy Diagram

Table 12 – Graphical nodes in Alpha Hierarchy diagrams.

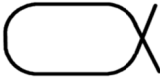
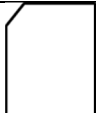

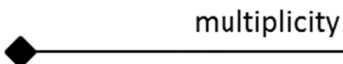
Node Type	Symbol	Reference
Alpha		Section 9.6.4.1 Alpha.
Work Product		Section 9.6.5.1 Work Product.

Table 13 – Graphical links in Alpha Hierarchy diagrams.

Link Type	Symbol	Reference
Alpha Containment		See 9.6.5.2 Alpha Containment.
Alpha Manifest		See 9.6.5.3 Alpha Manifest.

Examples

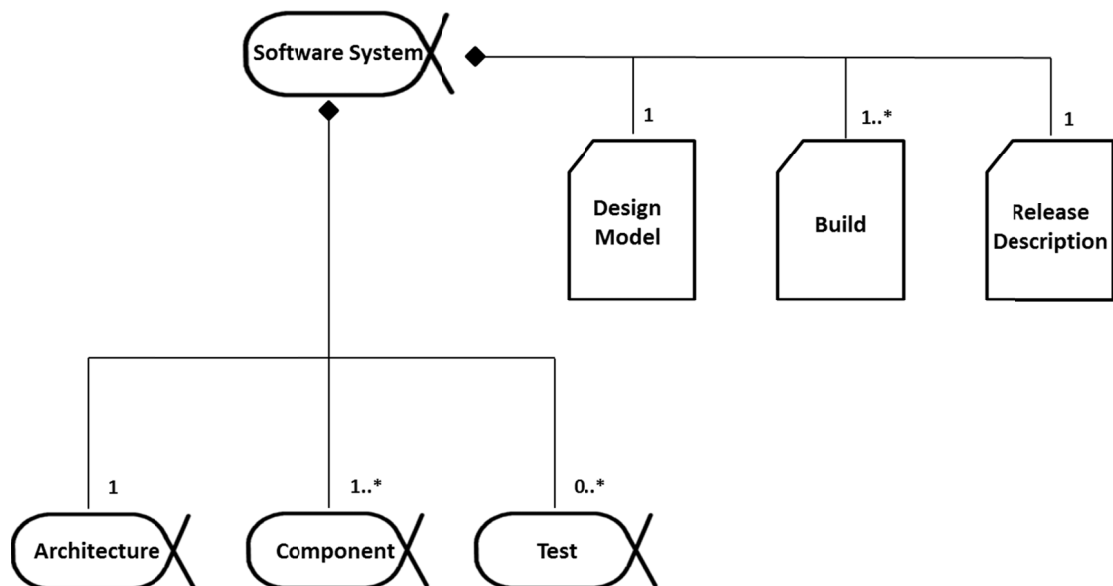


Figure 53 – Alpha Containment and Alpha Manifest relationships of the Software System Alpha

9.6.6 View 3: Activity Spaces and Activities

The following sections define relevant symbols for View 3: Activity Spaces and Activities. This view is thereby capable of visualizing elements in abstract syntax layer 3.

9.6.6.1 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:

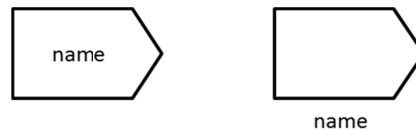


Figure 54 – Activity symbol

Style Guidelines

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

Examples

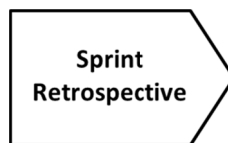


Figure 55 – Sprint Retrospective Activity

9.6.6.2 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:

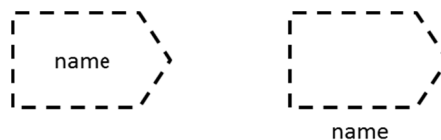


Figure 56 – Activity Space symbol

Style Guidelines

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

Examples



Figure 57 – Specify the Software Activity Space

9.6.6.3 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.

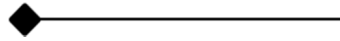


Figure 58 – Activity Manifest symbol

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

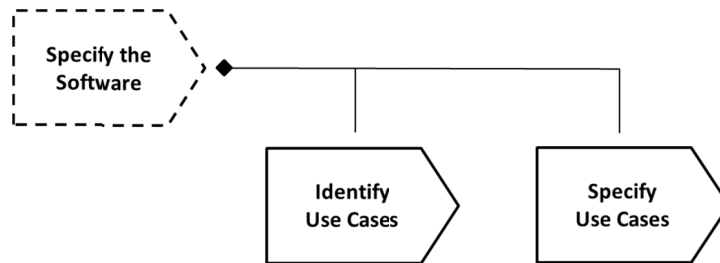


Figure 59 – Specify the Software Activity Space and two Activities: Identify Use Cases and Specify Use Cases

9.6.6.4 Activity Predecessor

An Activity Predecessor 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 the successor, that is, the opposite of the predecessor.

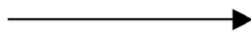


Figure 60 – Activity Predecessor symbol.

Style Guidelines

- Lines may be drawn using curved segments.

Examples

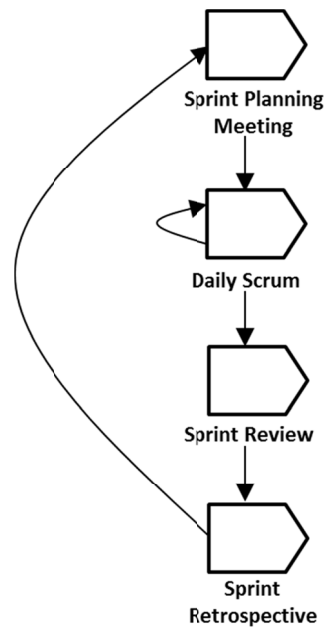


Figure 61 – Activity Predecessor among four activities in a Scrum Essentials practice

9.6.6.5 Competency

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



Figure 62 – Competency symbol

Style Guidelines

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

Examples



Figure 63 – Leadership Competency

9.6.6.6 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.

9.6.6.6.1 Activity Space Hierarchy Diagram

Table 14 – Graphical nodes in Activity Space Hierarchy diagrams.




Node Type	Symbol	Reference
Activity Space		Section 9.6.6.2 Activity Space.
Activity		Section 9.6.6.1 Activity.

Table 15 – Graphical links in Activity Space Hierarchy diagrams.

Link Type	Symbol	Reference
Activity Manifest		See 9.6.6.3 Activity Manifest.

Examples

Refer to 9.6.6.3 Activity Manifest example.

9.6.6.6.2 Activity Flow Diagram

Table 16 – Graphical nodes in Activity Flow diagrams.



Node Type	Symbol	Reference
Activity		Section 9.6.6.1 Activity.

Table - Graphical links in Activity Flow Hierarchy diagrams.

Link Type	Symbol	Reference
Activity Predecessor		See 9.6.6.4 Activity Predecessor.

Style Guidelines

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

Examples

Refer to 9.6.6.4 Activity Predecessor.

9.7 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).

9.7.1 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)

9.7.1.1 Root Elements

The root element representing the file containing the specification is defined as:

Model:

```
(AreaOfConcern) * (Kernel) * (Practice) *
```

An empty file is a valid root. If not empty, the file may contain an arbitrary number of AreaOfConcern declarations, an arbitrary number of Kernel declarations and an arbitrary number of Practice declarations.

An AreaOfConcern declaration is defined as:

AreaOfConcern:

```
'areaOfConcern' ID (STRING) ?
```

This maps directly to the language element with the same name as defined on Layer 3. The ID creates a unique identifier for this AreaOfConcern, 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”.

A Kernel declaration is defined as:

Kernel:

```
'kernel' ID  
  ('based on kernels' KernelRef (',' KernelRef) *) ?  
  '{'  
    (STRING) ?  
    (Alpha) *  
    (KernelAssociation) *  
    (Competency) *  
    (ActivitySpace) *  
  '}'
```

This maps directly to the language element with the same name as defined on Layer 3. 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”. KernelRef is a unique identifier of another Kernel, thus mapping to attribute “baseKernel”. The remaining elements are declarations for elements that can be owned by a Kernel.

A Practice declaration is defined as:

```
Practice:
    'practice' ID
    ('based on kernels' KernelRef (',' KernelRef)*)?
    ('based on practices' PracticeRef (',' PracticeRef)*)?
    '{'
        (STRING)?
        (Alpha)*
        (KernelAssociation)*
        (WorkProduct)*
        (AlphaManifest)*
        (ActivitySpace)*
        (Activity)*
        (ActivityManifest)*
        (Competency)*
        (Skill)*
        (Pattern)*
    '}'
```

This maps directly to the language element with the same name as defined on Layer 3. The ID creates a unique identifier for this Practice, which maps to the attribute “name”. The STRING is considered as content for attribute “description”. KernelRef is a unique identifier of a Kernel, thus mapping to attribute “baseKernel”. PracticeRef is a unique identifier of a Practice, thus mapping to attribute “basePractice”. The remaining elements are declarations for elements that can be owned by a Practice.

9.7.1.2 Kernel Elements

An Alpha declaration and its contents are defined as:

```
Alpha:
    'alpha' ID
        ('concerns' AreaOfConcernRef)?
        '{' (STRING)? StateGraph '}'

StateGraph:
    'has {' (StateGraphElement)+ '}'

StateGraphElement:
    State | Transition

State:
    'state' ID ('{' STRING ('checks {' (CheckListItem)+ '}')? '}')?

CheckListItem:
    'item' ID '{' STRING '}'

Transition:
    'transition' StateRef '->' StateRef
```

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:

```
KernelAssociation:
    AlphaAssociation | AlphaContainment

AlphaAssociation:
    Cardinality AlphaRef '--' STRING '-->' Cardinality AlphaRef
    ('in concern' AreaOfConcernRef)?

AlphaContainment:
    AlphaRef 'contains' Cardinality AlphaRef
```

The STRING is considered as content for attribute “name” of this AlphaAssociation. The Cardinality maps to the attribute “multiplicity” in both 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
    ('concerns' AreaOfConcernRef)?
    '{' (STRING)?
        'targets' StateRef (',' StateRef)*
        (InputAlpha)?
        (OutputAlpha)?
        (CompetencyRequirement)?
    '}'
```

```
InputAlpha:
    'inputAlphas {' AlphaRef (',' AlphaRef)* '}'
```

```
OutputAlpha:
    'outputAlphas {' AlphaRef (',' AlphaRef)* '}'
```

```
CompetencyRequirement:
    'requires competency' CompetencyRef 'at level' CompetencyLevelRef (','
CompetencyRef 'at level' CompetencyLevelRef)*
```

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
    ('concerns' AreaOfConcernRef)?
    '{' (STRING)? ('has {' (CompetencyLevel)* '}')? '}'
```

```
CompetencyLevel:
    'level' INT ID (STRING)? (SkillRequirement)?
```

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 attribute “level” of the CompetencyLevel element in the abstract syntax. See below for the SkillRequirement declaration, since this is usually added by a practice via composition. References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

9.7.1.3 Practice Elements

A WorkProduct declaration and its usage in an AlphaManifest declaration are defined as:

```
WorkProduct:
    'workProduct' ID '{' (STRING)? StateGraph '}'

AlphaManifest:
    'describe' AlphaRef 'by' Cardinality WorkProductRef (',' Cardinality
WorkProductRef)*
```

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 attribute “multiplicity” in the AlphaManifest. 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
        ('follows' (ActivityRef)*)?
        '{' (STRING)?
            'targets' StateRef (',' StateRef)*
            (InputAlpha)?
            (OutputAlpha)?
            (Input)?
            (Output)?
            (CompetencyRequirement)?
            (SkillRequirement)?
        '}'

Input:
    'input {' WorkProductRef (',' WorkProductRef)* '}'

Output:
    'output {' WorkProductRef (',' WorkProductRef)* '}'

SkillRequirement:
    'requires skill' SkillRef 'at level' SkillLevelRef (',' SkillRef 'at level'
SkillLevelRef)*
```

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)*
```

References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

A Skill declaration is defined as:

```
Skill:
    'skill' ID
        ('concerns' AreaOfConcernRef)?
        '{' (STRING)? ('has {' (SkillLevel)* '}')? '}'

SkillLevel:
    'level' INT ID (STRING)?
```

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 attribute “level” of the SkillLevel element in the abstract syntax. References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

Pattern:

```
'pattern' STRING '{' (
    ('with alphas' AlphaRef (',' AlphaRef)*)?
    ('with workProducts' WorkProductRef (',' WorkProductRef)*)?
    ('with states' StateRef (',' StateRef)*)?
    ('with activities' ActivityRef (',' ActivityRef)*)?
    ('with activitySpaces' ActivitySpaceRef (',' ActivitySpaceRef)*)?
) '}'
```

The STRING is considered as content for attribute “kind”. References via identifiers directly map to the respective associations of the meta-classes as defined in the abstract syntax.

9.7.1.4 Auxiliary Elements

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)*
```

9.7.2 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."

    has {
        state Conceived {"The need for a new system has been agreed."
            checks {
                item cli1 {"The initial set of stakeholders agrees that a
system is to be produced."}
                item cli2 {"The stakeholders that will use and fund the
new system are identified."}
                item cli3 {"The stakeholders agree on the purpose of the
new system."}
                item cli4 {"The expected value of the new system has been
agreed."}
            }
        }
        state Bounded {"The theme and extent of the new system is clear."
            checks {
                item cli1 {"Stakeholders involved in developing the new
system are identified."}
                item cli2 {"It is clear what success is for the new
system."}
                item cli3 {"The stakeholders have a shared understanding
of the extent of the proposed solution."}
                item cli4 {"The way the requirements will be described is
agreed upon."}
                item cli5 {"The mechanisms for managing the requirements
```

```

are in place."}
        item cli6 {"The prioritisation scheme is clear."}
        item cli7 {"Constraints are identified and considered."}
        item cli8 {"Assumptions are clearly stated."}
    }
    state Coherent {"The requirements provide a coherent description of
the essential characteristics of the new system."
        checks {
            item cli1 {"The requirements are captured and shared with
the team and the stakeholders."}
            item cli2 {"The origin of the requirements is clear."}
            item cli3 {"The rationale behind the requirements is
clear."}
            item cli4 {"Conflicting requirements are identified and
attended to."}
            item cli5 {"The requirements communicate the essential
characteristics of the system to be delivered."}
            item cli6 {"The most important usage scenarios for the
system can be explained."}
            item cli7 {"The priority of the requirements is clear."}
            item cli8 {"The impact of implementing the requirements
is understood."}
            item cli9 {"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 cli1 {"The stakeholders accept the requirements as
describing an acceptable solution."}
            item cli2 {"The rate of change to the agreed requirements
is relatively low and under control."}
            item cli3 {"The value provided by implementing the
requirements is clear."}
            item cli4 {"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 cli1 {"Enough of the requirements are addressed for
the resulting system to be acceptable to the stakeholders."}
            item cli2 {"The stakeholders accept the requirements as
accurately reflecting what the system does and doesn't do."}
            item cli3 {"The set of requirement items implemented
provide clear value to the stakeholders."}
            item cli4 {"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 cli1 {"The stakeholders accept the requirements as
accurately capturing what they require to fully satisfy the need for a new
system."}
            item cli2 {"There are no outstanding requirement items
preventing the system from being accepted as fully satisfying the requirements."}
            item cli3 {"The system is accepted by the stakeholders as
fully satisfying the requirements."}
        }
    }
}

```

```

        transition Conceived -> Bounded
        transition Bounded -> Coherent
        transition Coherent -> SufficientlyDescribed
        transition SufficientlyDescribed -> Satisfactory
        transition Satisfactory -> Fulfilled
    }
}

```

A minimal declaration of an Activity Space using the Alpha declared above:

```

activitySpace SpecifyTheSystem {
    targets Requirements.SufficientlyDescribed
}

```

A minimal declaration of a Practice using the Alpha and Activity Space declared above:

```

practice UserStoryPractice {
    workProduct UserStory {
        has {
            state Requested
            state Written
            state Realized
            transition Requested -> Written
            transition Written -> Realized
        }
    }

    workProduct UserAcceptanceTest {
        has {
            state Planned
            state Written
            state Executed
            state Passed
            transition Planned -> Written
            transition Written -> Executed
            transition Executed -> Passed
        }
    }

    activity WriteUserStories {
        targets UserStory.Written
    }

    activity WriteUserAcceptanceTests {
        targets UserAcceptanceTest.Written
    }

    describe Requirements by 1..N UserStory, 1..N UserAcceptanceTest
    do SpecifyTheSystem by WriteUserStories,WriteUserAcceptanceTests
}

```

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 17 – Mandatory Requirements (Kernel)

Requirement	Resolution
<p>6.5.1.1 Domain model</p> <p>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.</p>	<p>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.</p> <ul style="list-style-type: none"> 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.
<p>6.5.1.2 Key conceptual elements</p> <p>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):</p> <p>a. <i>System</i>: Concepts related to the system being produced, for example: software, platform, etc.</p> <p>b. <i>Functionality</i>: Concepts related to the required function of the system being produced, for example: requirements, needs, opportunities, stakeholders, etc.</p> <p>c. <i>People</i>: Concepts related to the people required to create a system with the required functionality, for example: project, team, role, etc.</p> <p>d. <i>Way of Working</i>: 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.</p>	<p>The Kernel's three areas of concern (see Section 8.2, 8.3 and 8.4) and their corresponding Alphas provide this coverage:</p> <ul style="list-style-type: none"> 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).

<p>6.5.1.3 Generic activities</p> <p>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):</p> <p>a. <i>Interacting with stakeholders</i>: Activities related to necessary interactions with stakeholders, for example: exploring possibilities, understanding needs, ensuring satisfaction, handling change, etc.</p> <p>b. <i>Developing the system</i>: Activities related to actually constructing a system, for example: specifying, shaping, implementing, testing, deploying and operating the system.</p> <p>c. <i>Managing the project</i>: Activities related to managing a project, for example: steering the project, supporting the project team, assessing progress and concluding the project.</p>	<p>The Kernel's three areas of concerns (see Section 8.2, 8.3 and 8.4) and their corresponding Activity Spaces provide this coverage:</p> <ul style="list-style-type: none"> • 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).
<p>6.5.1.4 Kernel elements</p> <p>The definition of each element of the Kernel shall include the following:</p> <p>a. A concise description of the meaning of the element and its use in software engineering, intuitively understandable to a practitioner.</p> <p>b. The relationships of the element to other elements in the Kernel.</p> <p>c. The various different states the element may take over time, including initial/entry and final/exit criteria as appropriate for the element.</p> <p>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.</p> <p>e. How different ways of applying the element may be compared to each other and guidance on deciding among the alternatives.</p> <p>f. Appropriate metrics that can be used to assess progress, quality, etc.</p>	<p>The Kernel element definitions cover:</p> <ul style="list-style-type: none"> • 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
<p>6.5.1.5 Scope and coverage</p> <p>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.</p>	<p>The Kernel can be extended to specific segments of the software industry by creating kernel extensions and specific practices.</p> <p>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.</p> <p>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</p>

	<p>waterfall lifecycles.</p> <p>See the lifecycle examples provided in Section C.1.3.</p>
<p>6.5.1.6 Extension</p> <p>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.</p> <p>a. The Kernel shall allow for project and organization specific extensions.</p> <p>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.</p>	<p>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.</p>

Table 18 – Mandatory Requirements (Language)

Requirement	Resolution
<p>6.5.2.1.1 MOF metamodel</p> <p>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.</p>	<p>The definition of the abstract syntax is based on MOF.</p>
<p>6.5.2.1.2 Static and operational semantics</p> <p>The Language shall have formal static and operational semantics defined in terms of the abstract syntax.</p>	<p>See Section 9.3 for the static semantics and section 9.5 for the dynamic semantics.</p>
<p>6.5.2.1.3 Graphical syntax</p> <p>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.</p>	<p>See Section 9.6 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.</p>
<p>6.5.2.1.4 Textual syntax</p> <p>The Language shall also have a textual concrete syntax that formally maps to the abstract syntax.</p>	<p>See Section 9.7 for the definition of the textual syntax.</p>
<p>6.5.2.1.5 SPEM 2.0 metamodel reuse</p> <p>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.</p>	<p>This is discussed in Annex B: Section B.2.</p>

<p>6.5.2.2.1 Ease of use</p> <p>The Language shall be designed to be easy to use for practitioners at different competency levels:</p> <ol style="list-style-type: none"> Those that have very little modeling experience and quickly and intuitively need to understand and learn how to use the Language. Intermediate users who are more advanced and willing to describe what kind of outcome they expect of their work. Advanced users that can work with all aspects of the Language to model their complete software endeavor. 	<p>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:</p> <ul style="list-style-type: none"> 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.
<p>6.5.2.2.2 Separation of views for practitioners and method engineers</p> <p>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.).</p> <p>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.</p>	<p>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.</p> <p>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.</p>
<p>6.5.2.2.3 Specification of kernel elements</p> <p>The Language shall have features for specifying Kernel elements, including:</p> <ol style="list-style-type: none"> Formal and informal descriptions of the content and meaning of an element. The relationship of the element of other elements. States the element may take over time and the events that cause transitions among those states. How the element is instantiated, including provisions for practice-specific tailoring of the element, and the basis for comparing different instantiations. Metrics defined to assess various attributes of the use of the element. 	<p>The language defines (amongst others) elements "Alpha" (see Section 9.3.1.1) and "Activity Space" (see Section 9.3.3.3) for specifying Kernel elements. The language include:</p> <ul style="list-style-type: none"> 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.
<p>6.5.2.2.4 Specification of practices</p>	<p>The language specification provides an element "Practice" (see Section 9.3.2.4) which is used and which relates to the Kernel</p>

<p>The Language shall have features for specifying practices in terms of Kernel elements, including:</p> <ol style="list-style-type: none"> Description of the particular cross-cutting concern addressed by the practice and the goal of the application of the practice. 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. Any work products required by and produced by the practice. 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.) Verification that the goal of the practice has been achieved in its application, particularly in terms of measurements of metrics defined for its elements. 	<p>elements in the following ways:</p> <ul style="list-style-type: none"> 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.
<p>6.5.2.2.5 Composition of practices</p> <p>The Language shall have features for the composition of practices, to describe existing and new methods, including:</p> <ol style="list-style-type: none"> Identifying the overall set of concerns addressed by composing the practices. 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). 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 	<p>The composition algebra allows for composition of practices.</p> <ul style="list-style-type: none"> a. Composed practices are in general not different from simple practices, so the concerns addressed by a composed practice can be 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.	
<p>6.5.2.2.6 Enactment of methods</p> <p>The semantic definition of the Language shall support the enactment by practitioners of methods defined in the Language, for the purposes of</p> <p>a. Tailoring the methods to be used on a project.</p> <p>b. Communicating and discussing practices and methods among the project team.</p> <p>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.</p> <p>d. Monitoring the progress of the project.</p> <p>e. Providing input for tool support for practitioners on the project.</p>	<ul style="list-style-type: none"> • 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 19 – Mandatory Requirements (Practices)

Requirement	Resolution
<p>6.5.3.1 Examples of Practices</p> <p>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.</p> <p>b. Submissions shall provide working examples to demonstrate the composing of practices into a method.</p> <p>c. Submissions shall provide working examples to demonstrate how a method can be enacted.</p> <p>d. Submission shall include a capability to demonstrate the operational execution of methods as a proof of concept.</p> <p>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:</p> <p>a. be described on its own, independent from any other practice</p> <p>b. be either explicitly defined as a continuous</p>	<p>A set of examples is described in Annex C:</p> <ul style="list-style-type: none"> • a. See Section C.1. • b. See Section C.2. • c. See Section C.3. • d. See Section C.3.

<p>activity or have a clear beginning and end states</p> <p>c. bring defined value to its stakeholders</p> <p>d. be assessable; in other words, its description must include criteria for its assessment when used</p> <p>e. include, whenever applicable, quantitative elements in its assessment criteria; correspondingly, the description must include suitable assessing metrics.</p>	
--	--

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 making 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 20 shows the various names considered for the Activity Spaces in the Customer Area of Concern.

Table 20 – 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 Satisfaction	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 21 shows the various names considered for the Activity Spaces in the solution Area of Concern.

Table 21 – 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 specific approaches to achieving the underlying state changes.
	Design the System	
Implement the System	Implement Software	There is more than just implementing the software involved 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 22 shows the various names considered for the Activity Spaces in the endeavor Area of Concern.

Table 22 – 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 64. 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.

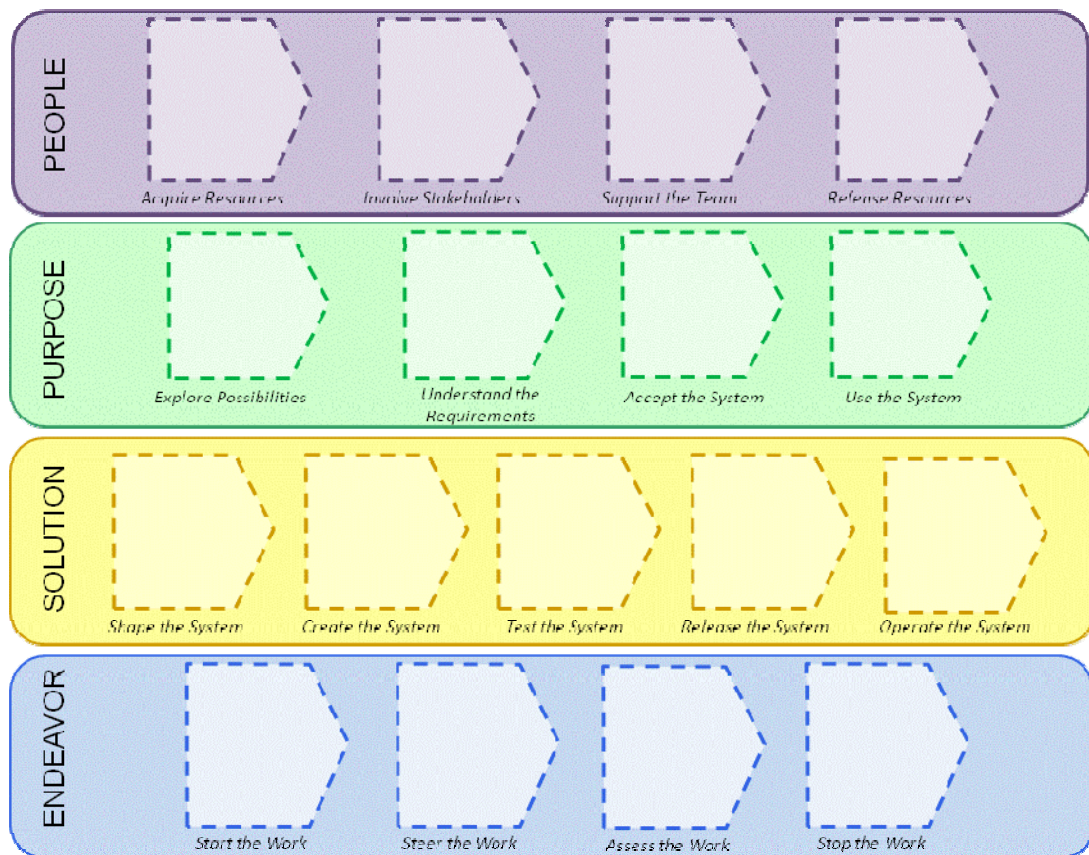


Figure 64 – Alternative Set of Activity Spaces using four Areas of Concern

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 than the one presented here.

B.2 SPEM 2.0

<This will be provided as an Annex update for the March meeting.>

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

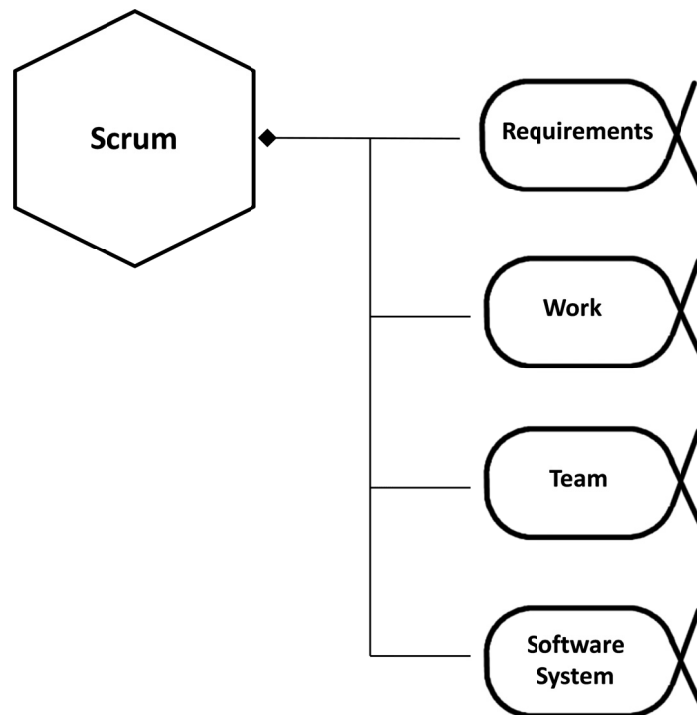


Figure 65 – Scrum practice

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

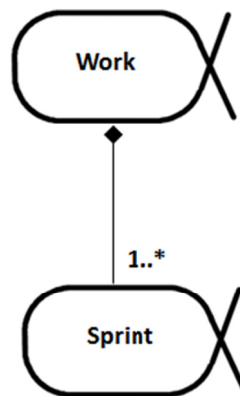


Figure 66 – Sprint sub-alpha of Work

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. Here we have adopted the states of the Requirements alpha but introduced Scrum-specific checkpoints (see the Textual syntax example).

Graphical syntax

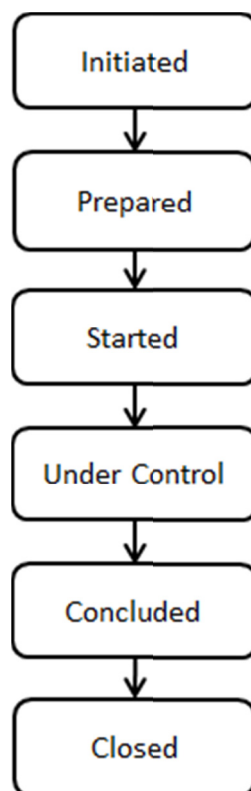


Figure 67 – The states of the Sprint sub-alpha

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 Initiated {"The work has been requested."
            checks {
                item c1 {"Product Owner presents ordered Product
                    Backlog items to the Development Team."}
            }
        }
        state Prepared {"All pre-conditions for starting the work have been
        met."
            checks {
                item c1 {"Entire Scrum Team collaborates on understanding
                    the work of the Sprint"}
                item c2 {"Development Team decides how it will build this
                    functionality into a "Done" product Increment
                    during the Sprint"}
                item c3 {"Scrum Team crafts a Sprint Goal"}
            }
        state Started {"The work is proceeding."
            checks {
                item c1 {"A new Sprint starts immediately after the
                    conclusion of the previous Sprint"}
            }
        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

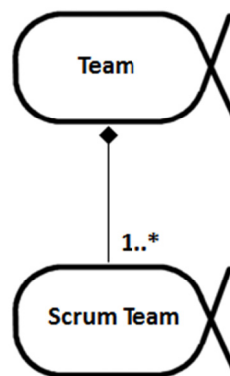


Figure 68 – Scrum Team

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

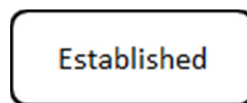


Figure 69 – The states of the Scrum Team sub-alpha

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 and Sprint 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]
- "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

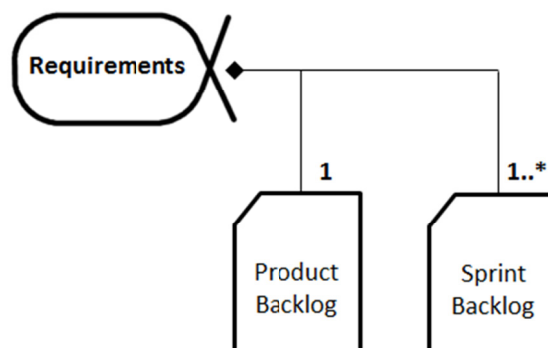


Figure 70 – Product Backlog

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...)"

    has {
        state NotOrdered
        state Ordered
    }
}
```

```

    transition NotOrdered -> Ordered
  }
}

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...)"

  has {
    state Planned
    state Assigned
    state Done
    transition Planned -> Assigned
    transition Assigned -> Done
  }
}

```

C.1.1.3.2 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

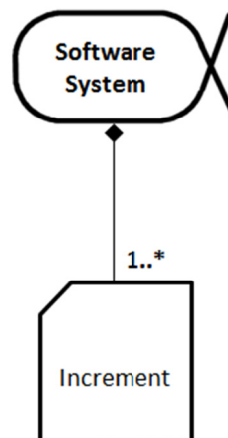


Figure 71 – Increment

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...)"

  has {

```

```

state BeingDeveloped
state Done
state Released
transition BeingDeveloped -> Done
transition Done -> Released
}

```

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

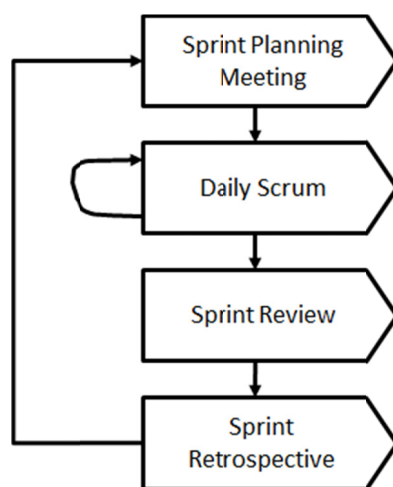


Figure 72 – Scrum activities

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

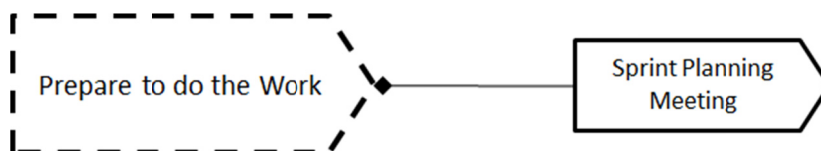


Figure 73 – Sprint Planning Meeting

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

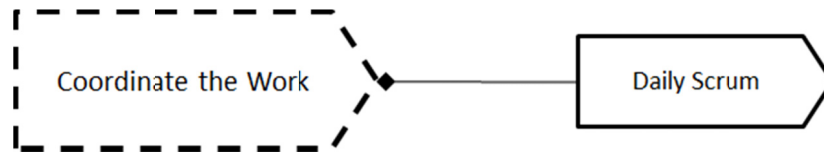


Figure 74 – Daily Scrum

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

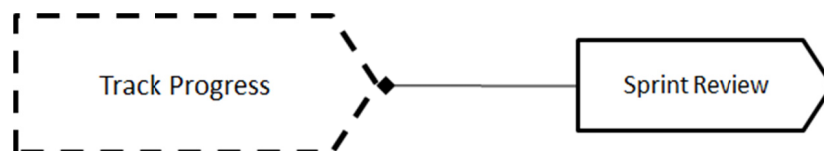


Figure 75 – Sprint Review

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

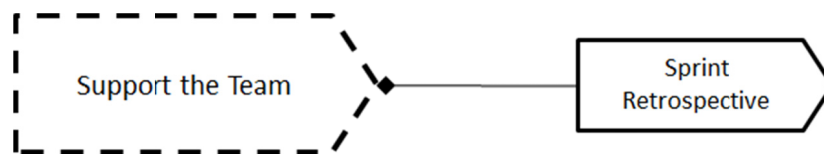


Figure 76 – Sprint Retrospective

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

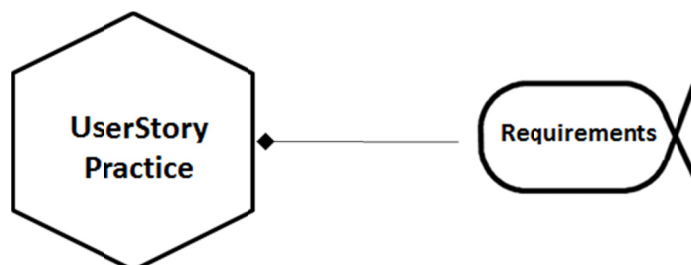


Figure 77 – User Story practice

C.1.2.2 Work Products

C.1.2.2.1 User Story

Graphical syntax

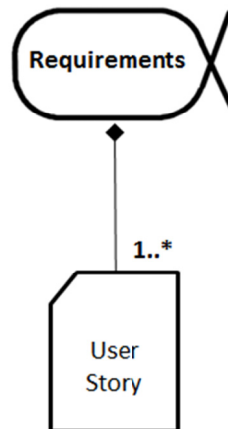


Figure 78 – User Story

C.1.2.3 Activities

C.1.2.3.1 Write User Story

Graphical syntax

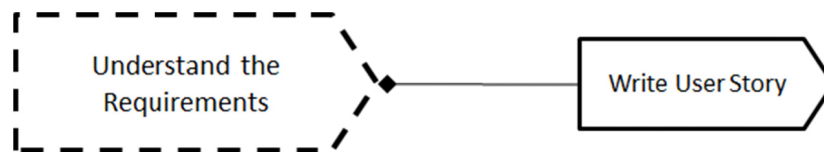


Figure 79 – Write User Story

C.1.3 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 80.

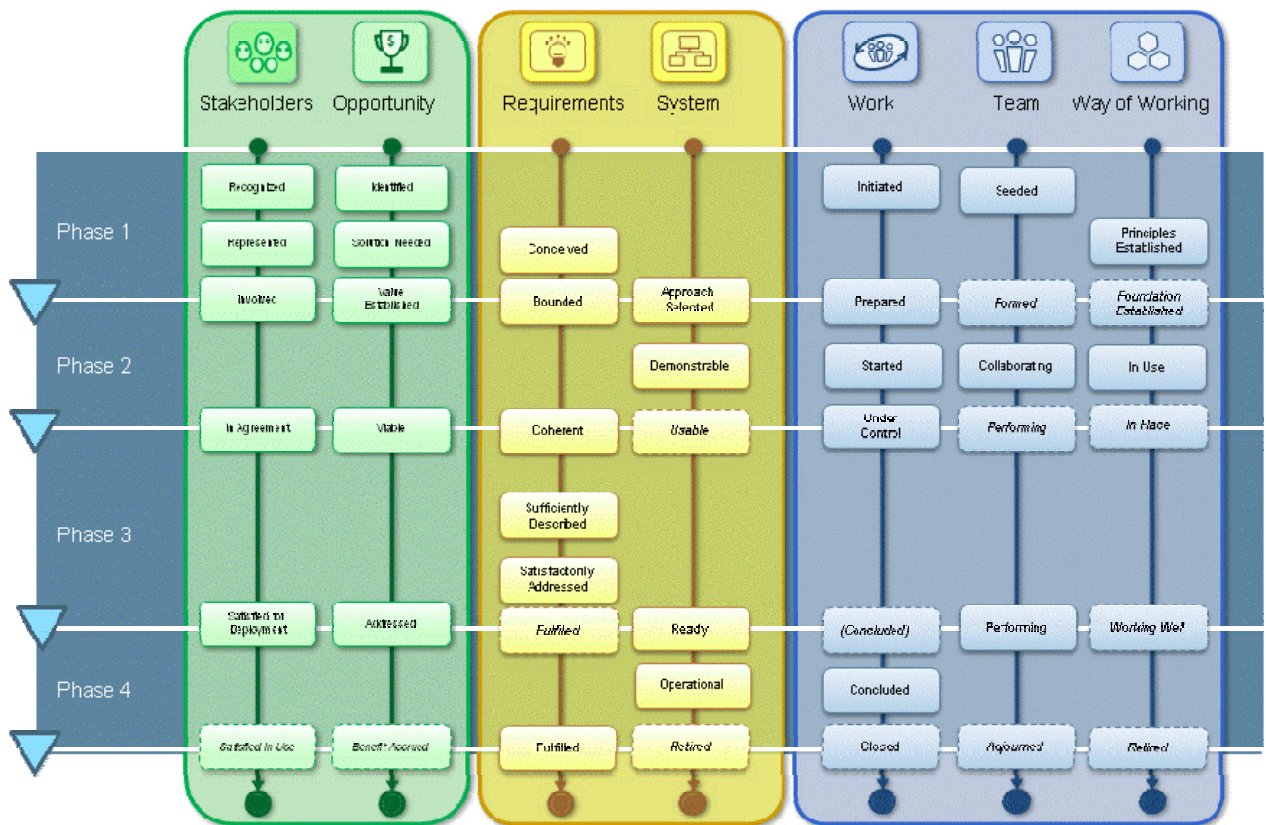


Figure 80 – Lifecycle template

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.1.3.1 The Unified Process Lifecycle

An illustration of the Unified Process Lifecycle is shown in Figure 81. 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 81, the milestones are represented by the blue inverted triangles but the names are suppressed to keep things simple.

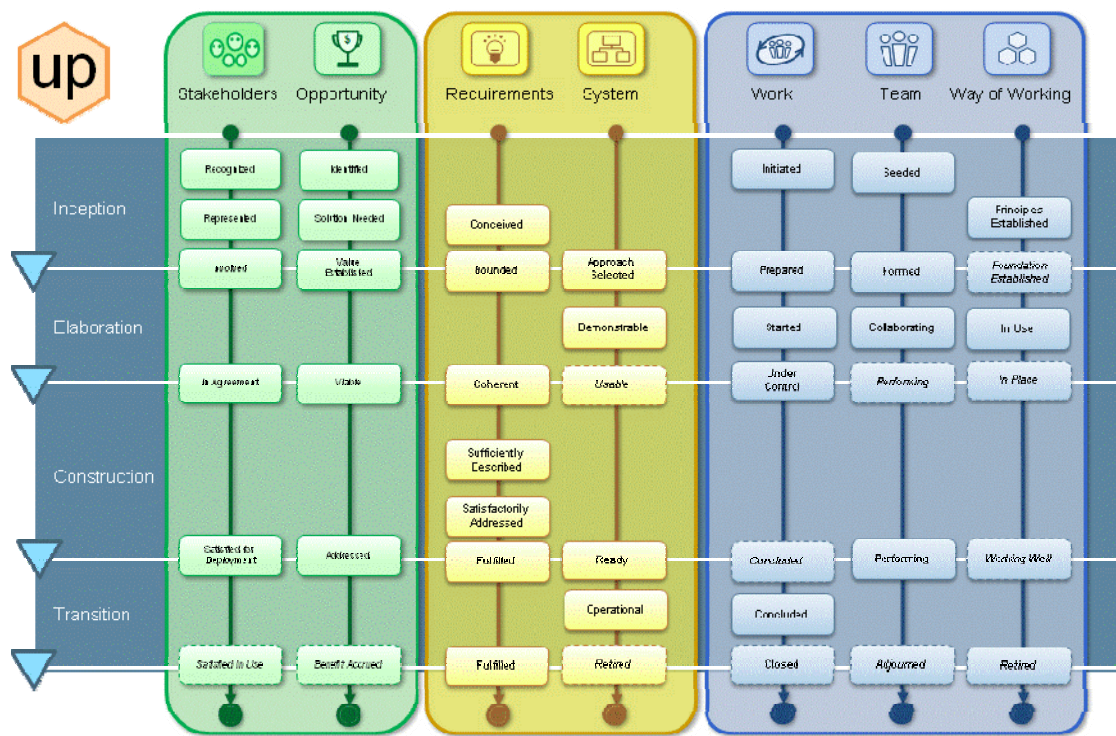


Figure 81 – The Unified Process lifecycle

C.1.3.2 The Waterfall Lifecycle

An illustration of a Waterfall Lifecycle is shown in Figure 82. 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.

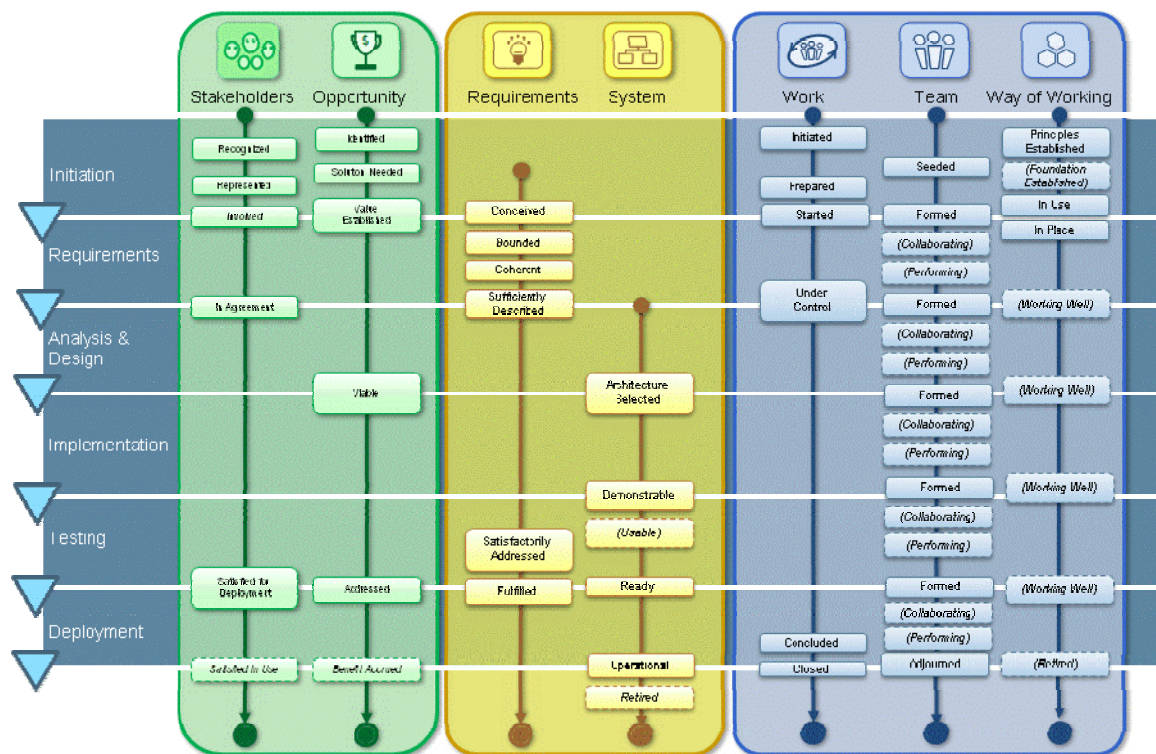


Figure 82 – A Waterfall lifecycle

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.1.3.3 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 83 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.

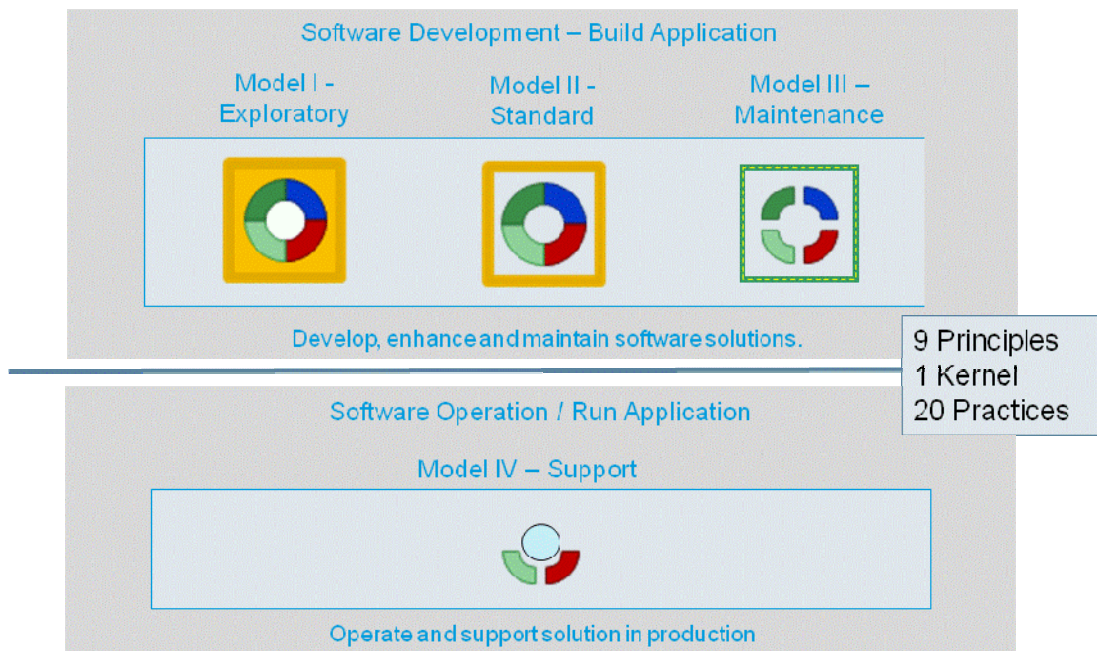


Figure 83 – Different types of development need different methods and lifecycles

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 84. 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 85, Figure 86, Figure 87 and Figure 88.

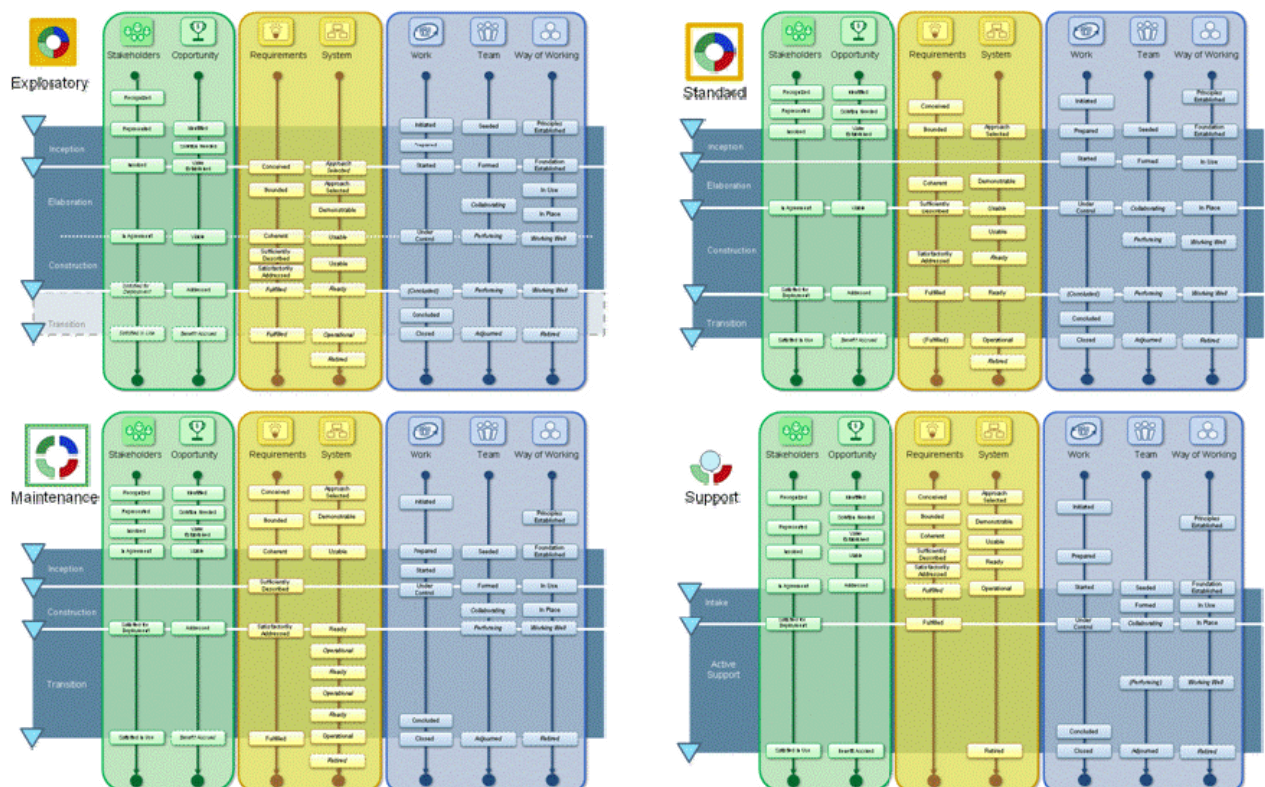


Figure 84 – Four complementary lifecycles to support application development

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.

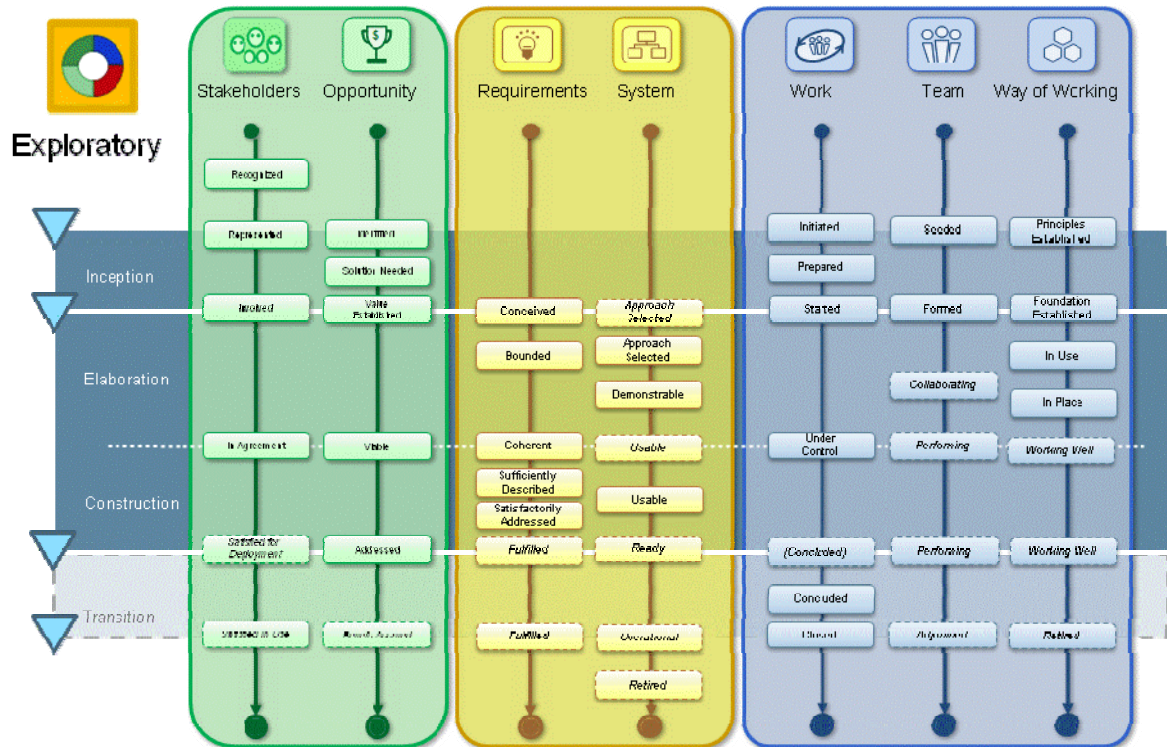


Figure 85 – The Exploratory lifecycle

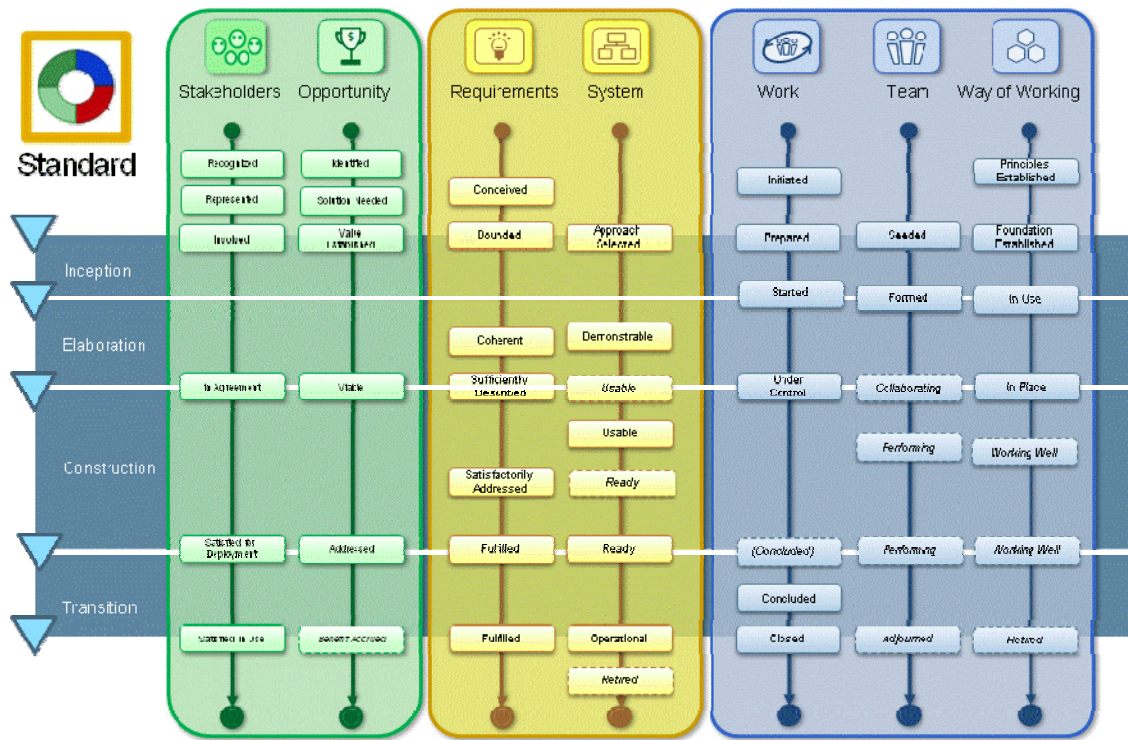


Figure 86 – The Standard lifecycle

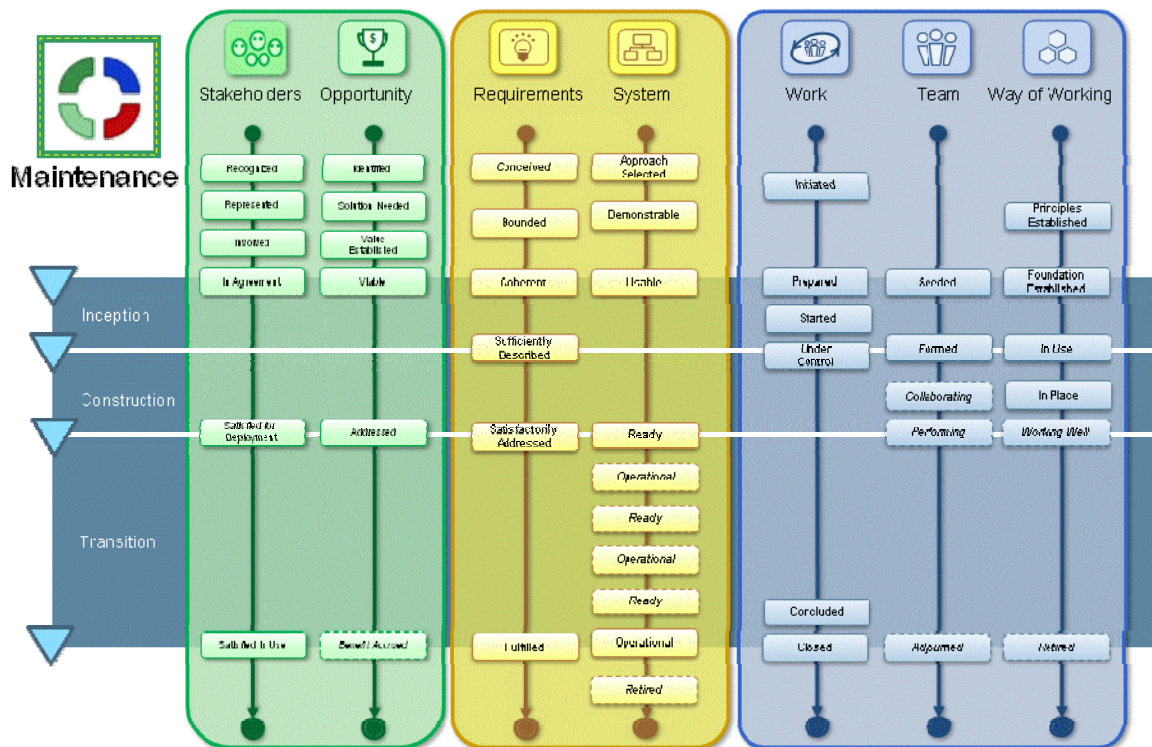


Figure 87 – The Maintenance lifecycle

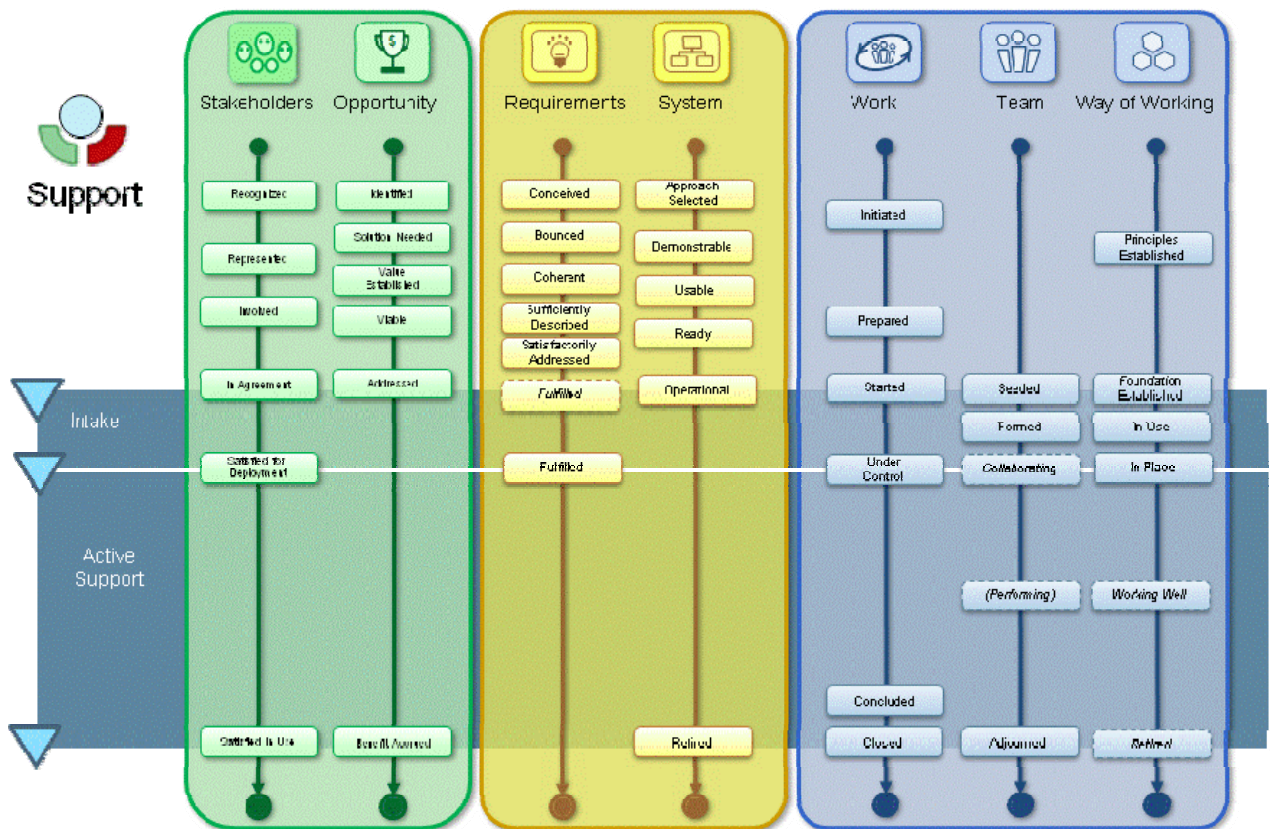


Figure 88 – The Support lifecycle

C.2 Composing Practices into Methods

<This will be provided as an Annex update for the March meeting.>

C.3 Enactment of Methods

<This will be provided as an Annex update for the March meeting.>