

Report

The FLUIDE Specification Languages with an Accompanying Method

Author(s)

Erik Gøsta Nilsson

Ketil Stølen



SINTEF IKT
SINTEF ICT

Address:
Postboks 124 Blindern
NO-0314 Oslo
NORWAY

Telephone:+47 73593000
Telefax:+47 22067350

postmottak.ikt@sintef.no
www.sintef.no
Enterprise /VAT No:
NO 948 007 029 MVA

Report

The FLUIDE Specification Languages with an Accompanying Method

KEYWORDS:
User interface
specification languages

Emergency response

VERSION
Final

DATE
2016-12-30

AUTHOR(S)
Erik Gøsta Nilsson
Ketil Stølen

CLIENT(S)
The EMERGENCY project supported by the Research
Council of Norway, p.nr. 187799/S10

CLIENT'S REF.
187799/S10

PROJECT NO.
90B261

NUMBER OF PAGES/APPENDICES:
136 incl. appendices

ABSTRACT

In this report we provide the syntax, semantics and pragmatics of the FLUIDE Specification Languages. First we provide an introduction to the FLUIDE Specification Languages, including their context, the rationale behind their design, important principles, and definitions of the main constructs in the languages. After the introduction we give detailed definitions of the languages. The detailed description includes their graphical syntax, their abstract syntax expressed in EBNF, as well as a natural language semantics. All these three aspects of the languages are illustrated by examples. Finally, we present the pragmatics of the languages through the parts of the FLUIDE Method giving guidance on how to use the FLUIDE Specification Languages.

PREPARED BY
Erik G. Nilsson

CHECKED BY
Jan Håvard Skjetne

APPROVED BY
Bjørn Skjellaug

REPORT NO. **ISBN**
A27972 9788214061505

CLASSIFICATION
Unrestricted

CLASSIFICATION THIS PAGE
Unrestricted

SIGNATURE

SIGNATURE

SIGNATURE

Table of contents

1	Introduction	6
2	The FLUIDE Specification Languages	7
2.1	Main Principles.....	7
2.2	Main Constructs in FLUIDE-A	9
2.3	Main Constructs in FLUIDE-D	12
3	The FLUIDE-A Language	15
3.1	Interactor	16
3.1.1	Graphical Syntax	16
3.1.2	Abstract Syntax	17
3.2	Basic Content Presenter.....	18
3.2.1	Graphical Syntax	18
3.2.2	Abstract Syntax	18
3.2.3	Semantics.....	20
3.2.4	Example	23
3.2.4.1	EBNF Specification	23
3.2.4.2	Semantics of the EBNF Specification	23
3.3	Aggregated Content Presenter	25
3.3.1	Graphical Syntax	25
3.3.2	Abstract Syntax	26
3.3.3	Semantics.....	27
3.3.4	Example	28
3.3.4.1	EBNF Specification	28
3.3.4.2	Semantics of the EBNF Specification	29
3.4	Task Supporter	32
3.4.1	Graphical Syntax	32
3.4.2	Abstract Syntax	32
3.4.3	Semantics.....	33
3.4.4	Example	34
3.4.4.1	EBNF Specification	34
3.4.4.2	Semantics of the EBNF Specification	34
3.5	Basic Work Supporter	35
3.5.1	Graphical Syntax	35
3.5.2	Abstract Syntax	35

3.5.3	Semantics.....	36
3.5.4	Example	38
3.5.4.1	EBNF Specification	39
3.5.4.2	Semantics of the EBNF Specification	40
3.6	Aggregated Work Supporter	41
3.6.1	Graphical Syntax	41
3.6.2	Abstract Syntax.....	41
3.6.3	Semantics.....	43
3.6.4	Example	45
3.6.4.1	EBNF Specification	45
3.6.4.2	Semantics of the EBNF Specification	46
3.7	Category Manager	48
3.7.1	Graphical Syntax	48
3.7.2	Abstract Syntax.....	48
3.7.3	Semantics.....	49
3.7.4	Example	50
3.7.4.1	EBNF Specification	50
3.7.4.2	Semantics of the EBNF Specification	50
4	The FLUIDE-D Language	51
4.1	Interactor Design	51
4.1.1	Graphical Syntax.....	51
4.1.2	Abstract Syntax.....	53
4.2	Views.....	54
4.2.1	Graphical Syntax	54
4.2.1.1	Decorational View	54
4.2.1.2	Layout Manager View	55
4.2.1.3	Content View	55
4.2.1.4	Content Integration View	57
4.2.1.5	Interactor Design View	58
4.2.1.6	Dialog navigation	58
4.2.1.7	Example	59
4.2.2	Abstract Syntax.....	59
4.2.2.1	Decorational View	62
4.2.2.2	Layout Manager View	62
4.2.2.3	Content View	62
4.2.2.4	Model Patterns Used in Content Views.....	66
4.2.2.5	Content Integration View	68
4.2.2.6	Interactor Design View	69
4.2.2.7	Dialog navigation	69

4.2.3	Semantics.....	70
4.2.3.1	Decorational View	73
4.2.3.2	Layout Manager View	74
4.2.3.3	Content View	75
4.2.3.4	Model Patterns Used in Content Views.....	89
4.2.3.5	Content Integration View	91
4.2.3.6	Interactor Design View	95
4.2.3.7	Dialog navigation	96
4.3	Basic Content Presenter Design.....	98
4.3.1	Graphical Syntax	98
4.3.2	Abstract Syntax.....	99
4.3.3	Semantics.....	101
4.3.4	Example	102
4.3.4.1	EBNF Specification	102
4.3.4.2	Semantics of the EBNF Specification	103
4.4	Aggregated Content Presenter Design	105
4.4.1	Graphical Syntax	105
4.4.2	Abstract Syntax.....	106
4.4.3	Semantics.....	107
4.4.4	Example	107
4.4.4.1	EBNF Specification	107
4.4.4.2	Semantics of the EBNF Specification	108
4.5	Task Supporter Design	109
4.5.1	Graphical Syntax	109
4.5.2	Abstract Syntax.....	109
4.5.3	Semantics.....	110
4.5.4	Example	111
4.5.4.1	EBNF Specification	111
4.5.4.2	Semantics of the EBNF Specification	111
4.6	Basic Work Supporter Design.....	113
4.6.1	Graphical Syntax	113
4.6.2	Abstract Syntax.....	113
4.6.3	Semantics.....	115
4.6.4	Example	115
4.6.4.1	EBNF Specification	115
4.6.4.2	Semantics of the EBNF Specification	116
4.7	Aggregated Work Supporter Design	117
4.7.1	Graphical Syntax	117
4.7.2	Abstract Syntax.....	117
4.7.3	Semantics.....	119

4.7.4	Example	119
4.7.4.1	EBNF Specification	119
4.7.4.2	Semantics of the EBNF Specification	120
4.8	Category Manager Design.....	122
4.8.1	Graphical Syntax	122
4.8.2	Abstract Syntax	122
4.8.3	Semantics.....	124
4.8.4	Example	124
4.8.4.1	EBNF Specification	124
4.8.4.2	Semantics of the EBNF Specification	124
5	The FLUIDE Method	126
5.1	Part 1 – Specifying user interfaces with FLUIDE-A.....	126
5.1.1	Step 1.1. Identify users/roles.....	126
5.1.2	Step 1.2. Identify work and task structure	127
5.1.3	Step 1.3. Determine tasks that need ICT support	127
5.1.4	Step 1.4. Determine information needs for task.....	127
5.1.5	Step 1.5. Choose, adapt or specify Content Presenter.....	127
5.1.6	Step 1.6. Consolidate Content Presenters.....	128
5.1.7	Step 1.7. Specify Task Supporters.....	129
5.1.8	Step 1.8. Specify Work Supporters.....	129
5.1.9	Step 1.9. Consolidate Work Supporters	129
5.1.10	Step 1.10. Specify Category Managers	129
5.2	Specifying user interfaces with FLUIDE-D	130
5.2.1	Step 2.1. Choose platforms, modalities and styles to use for each role	131
5.2.2	Step 2.2. Determine platforms, modalities and styles deviations for work	131
5.2.3	Step 2.3. Determine platforms, modalities and styles deviations for tasks	132
5.2.4	Step 2.4. Specify Content Presenter Designs.....	132
5.2.5	Step 2.5. Specify Task Supporter Designs	132
5.2.6	Step 2.6. Specify Work Supporter Designs	133
5.2.7	Step 2.7. Specify Category Manager Designs	133
	References.....	135

1 Introduction

We have developed the FLUIDE Framework to support development of flexible user interface supporting emergency responders. This framework contains a number of parts, as shown in Figure 1.1.

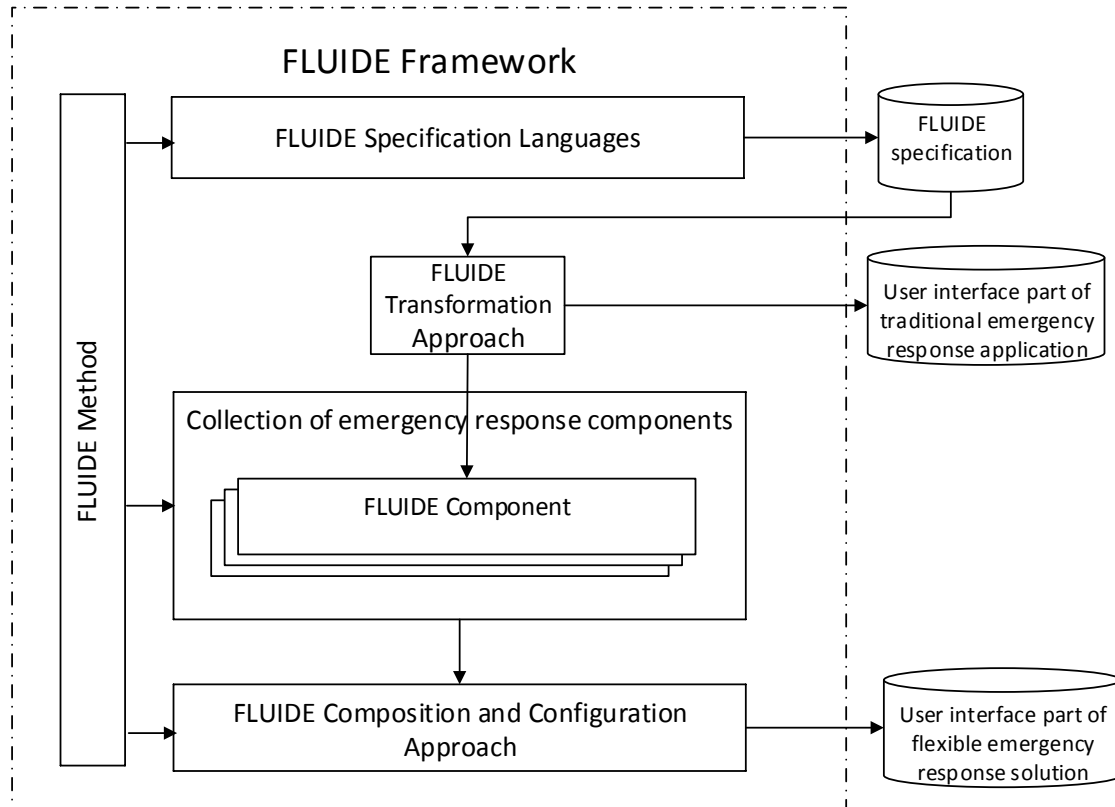


Figure 1.1. Overview of the FLUIDE Framework

The current version of the FLUIDE Framework is a first prototype. The most mature part of the FLUIDE Framework is the FLUIDE Specification Languages, denoted FLUIDE-A and FLUIDE-D. FLUIDE-A is used for expressing abstract user interface specifications, denoted *FLUIDE-A specifications*, while FLUIDE-D is used for expressing how FLUIDE-A specifications should be presented on different targets, denoted *FLUIDE-D specifications*. A target is an arbitrary combination of platform, type, style and modality used in a running user interface.

In this report, we provide a detailed definition of these two specification languages. When defining, describing and discussing languages, it is common to make a distinction between the *syntax*, *semantics* and *pragmatics* of the language at hand.

In the context of specification languages, as well as other artificial languages, the syntax describes the set of legal expressions. It is common to make a distinction between the *concrete* and *abstract* syntax of such languages. The concrete syntax may be textual, graphical, or even implicitly available by the user interface of a tool employed to express specifications in the language. The abstract syntax is usually expressed through a meta model and/or a definition in Extended Backus-Naur Form (EBNF). The concrete syntax of the FLUIDE Specification Languages is expressed using a graphical syntax. The abstract syntax is defined in EBNF, supported by concept models explaining how the constructs relate to each other in each of the languages. The graphical and abstract syntax for FLUIDE-A and FLUIDE-D is presented in Section 3 and 4 respectively, including examples illustrating how specifications are expressed in the graphical and abstract

syntax. The presentation of the syntax focuses on the different *constructs* in the languages. Before providing the details in Section 3 and 4, we present the rationale behind using the chosen constructs, definitions of the constructs, as well as other main principles in Section 2.

Semantics are concerned with the meaning of the legal sentences or expressions in a language. For specification languages, the semantics may be defined through different formal definitions which enables conducting formal reasoning and even proving different aspects of a language. Another approach is to define the semantics through implementing a compiler or an interpreter enabling expressions in a language to be executed. With this approach the semantics is defined through how it is executed. A third approach is to define a translation from the language at hand to a different language whose semantics is formally defined or well known. We have applied the third approach through defining the semantics of the FLUIDE Specification Languages as a set of production rules translating expressions in EBNF to English sentences. These production rules are defined for FLUIDE-A and FLUIDE-D in Section 3 and 4 respectively, including examples of the English sentences corresponding to specifications expressed in EBNF.

The pragmatics of languages deals with how the languages are used. For artificial languages, it is common to focus on how they *should* be used. A typical way of operationalizing this is through a *method*. In Section 5 we present the parts of the FLUIDE Method giving guidance on how to use the FLUIDE Specification Languages. There is one part for each of the two languages.

2 The FLUIDE Specification Languages

In this section, we provide an introduction to the FLUIDE Specification Languages. We start by outlining the main principles applied in the design of the languages. Then we present the main constructs in the languages, and related concepts used when describing them.

2.1 Main Principles

FLUIDE provides two specification languages: FLUIDE-A for expressing abstract (platform-independent) user interface (AUI) (Calvary et al., 2003), and FLUIDE-D for expressing concrete (platform-specific) user interface (CUI) specifications.

In Figure 2.1 we give a schematic overview of a FLUIDE-A specification.

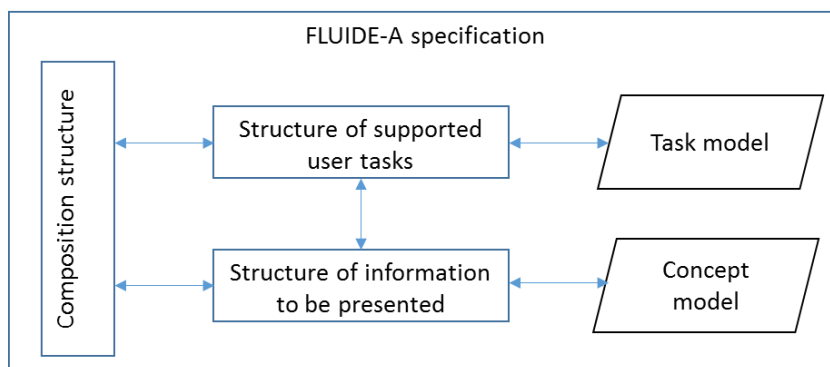


Figure 2.1. Content of FLUIDE-A specifications

Two FLUIDE-A constructs embed models expressing the intension of the user interface, one focusing on hierarchical and temporal structure among the tasks the user perform (task model) (Paternò, 1999; Wilson and Johnson, 1996) the other on the structure of the information to be presented (concept model) (OMG, 2008). Other constructs only specify compositions. Neither of the constructs offer any means for specifying

user interface components or controls, but possible aspects regarding the rendering of the intended user interfaces may be specified in annotations.

FLUIDE-D *designs* refine FLUIDE-A specifications for certain platforms, styles and modalities. These designs add a more concrete specification of the structure of the user interface through different *view* types. *Content Views* specify how the extent of a model fragment is presented. Figure 2.2 shows the connection between Content Views and specifications in FLUIDE-A and FLUIDE-D. Such views provide means for specifying quite advanced designs in a very compact way through exploiting a combination of user interface patterns Borchers (2001) and model patterns (Gamma et al., 1994).

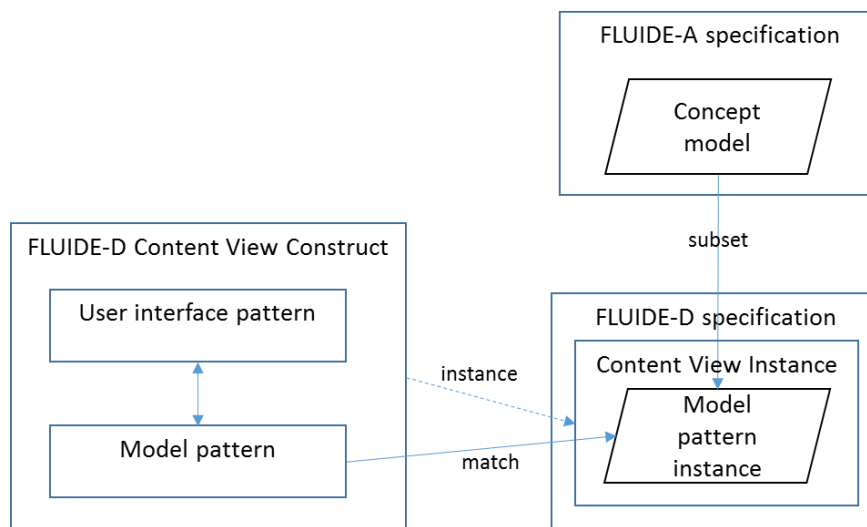


Figure 2.2. Content views in FLUIDE-D specifications

As FLUIDE-D is based on needs in the emergency response domain, the available views are ones considered particularly useful in this domain. The views provide versatility through being based on model patterns. This means that the views may be used to specify advanced user interfaces managing a wide variety of information as long as the information to be presented has a structure that matches the model patterns used in the view. For example the *Map Icons View* provides means for specifying an icon-based presentation of any type of information in a map user interface as long as the model follows a given structure (including providing locations). This view may thus just as well be used for presenting incident objects, resources, victims, important locations or risks. Thus, such views combine being specialized and powerful with regard to emergency response needs with being versatile with regard to the actual information they present.

FLUIDE uses a hybrid approach for domain support, as illustrated in Figure 2.3. All the constructs in FLUIDE-A, and the corresponding design constructs in FLUIDE-D are generic. FLUIDE does nevertheless provide specific support for the emergency response domain through view types in FLUIDE-D that support user interface patterns that are particularly useful in the emergency response domain. These views are constructs in FLUIDE-D, and thus they are a domain-specific part of FLUIDE-D. But as they reflect certain user interface patterns, and are named after these user interface patterns rather than after emergency response specific concepts, we consider them a library of emergency response user interface patterns. This is also supported by the versatility provided through the use of model patterns. Emergency response user interfaces also benefit from user interface designs that are not made specifically for the domain. Thus, FLUIDE-D contains generic views (reflecting user interface patterns and supporting model patterns) in addition to the domain-specific ones.

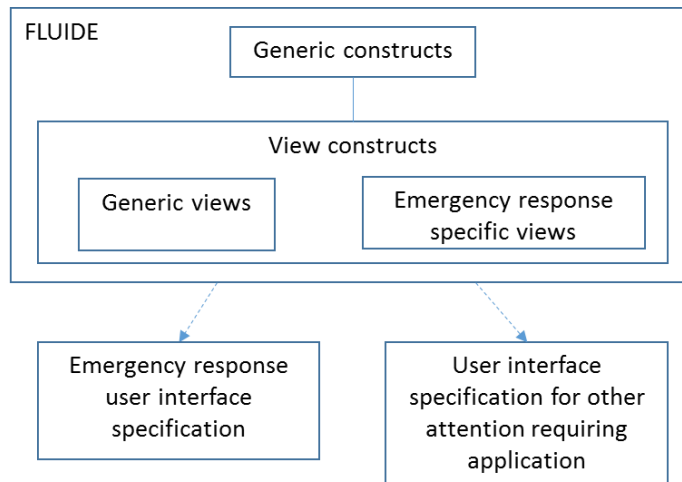


Figure 2.3. Domain support in FLUIDE

2.2 Main Constructs in FLUIDE-A

The user interface of an emergency response application must support the work performed by emergency responders. The four main language constructs in FLUIDE presented as rounded rectangles in Figure 2.4 support a natural breakdown of emergency response work, presented as ordinary rectangles.

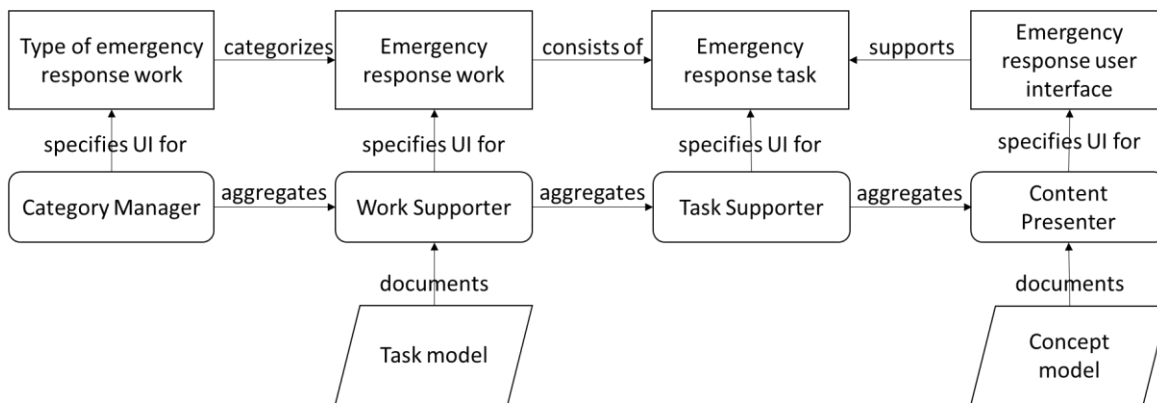


Figure 2.4. Overview of the main constructs in FLUIDE-A

We use the term *interactor construct* to refer to these constructs in FLUIDE-A and *interactor design construct* to refer to the corresponding constructs in FLUIDE-D. The CAMELEON glossary (CAMELEON, 2003) provides this definition of *interactor*:

Interactor: A computational abstraction that allows the rendering and manipulation of entities (domain concepts and/or tasks) that require input and output resources.

Based on this definition we define *interactor construct*:

By *interactor construct* we mean a language construct for specifying *interactor instances*, operationalized as one of the four main construct in FLUIDE-A, i.e. either *Content Presenter*, *Task Supporter*, *Work Supporter* or *Category Manager*.

When a systems developer uses the FLUIDE-A to specify the user interface of an application, the developer makes *instances* of the interactor constructs. We denote such specifications *interactor instances*.

By *interactor instance* we mean the concrete use of an interactor construct as part of a specification of a user interface.

Two of the interactor constructs, i.e. Content Presenter and Work Supporter have an associated domain model. As these models may be part of larger domain models we denote them *model fragments*.

By *model fragment* we mean a domain model or part thereof. We denote a model fragment expressed in a task modelling notation a *task model fragment*. We denote a model fragment expressed in a concept modelling notation a *concept model fragment*.

Emergency response work can be categorized with respect to responder types, responder roles and high level tasks, as well as combinations of these. In FLUIDE-A, the *Category Manager* construct supports the specification of a whole application, or some part of it if the application supports more than one category of functionality (Nilsson and Stølen, 2011).

By *Category Manager* we mean a construct that makes it possible to specify a placeholder for a part of a user interface supporting a category of functionality within a specific application domain, supporting different type of work related to the category.

A category of functionality supports certain work performed by emergency responders. Such work can be divided into tasks on different levels. These tasks may be categorized both in a hierarchical goals/means structure and through temporal constraints between sets of tasks. Such task structures are specified using the *Work Supporter* construct, which includes a task model to specify hierarchical and temporal structures.

By *Work Supporter* we mean a construct that makes it possible to define a placeholder for a part of a user interface supporting certain work within a specific application domain, as expressed in a connected task model fragment.

FLUIDE-A provides two kinds of Work Supporter constructs, i.e. *basic* and *aggregated*. Aggregated Work Supporters aggregate other Work Supporters (basic or aggregated). To be denoted *connected*, a task model fragment used in a Work Supporter must meet some constraints.

By *connected task model fragments* we mean a task model fragment containing at least one task, where all tasks in the task model fragment are part of the same hierarchical structure.

To be consistent with the vocabulary used in relation to concept model fragments below, we also introduce the term *task anchor*.

By *task anchor* we mean the root task of a connected task model fragment.

A user interface supporting one of the tasks in the task model of a Work Supporter needs to manage certain information content that is relevant for solving the task. The information needs of individual tasks are specified using the *Task Supporter* construct.

By *Task Supporter* we mean a construct that makes it possible to define a placeholder for a part of a user interface supporting a specific task in a specific application domain.

The task supported by a Task Supporter is usually part of at least one task model associated with a Work Supporter. How the information content used in a Task Supporter is further broken down and structured in a (part of a) user interface is specified using the *Content Presenter* construct.

By *Content Presenter* we mean a construct that makes it possible to define a placeholder for a part of a user interface within a specific application domain, presenting instances reflecting a connected concept model fragment.

FLUIDE-A provides two kinds of Content Presenter constructs, i.e. *basic* and *aggregated*. Aggregated Content Presenters aggregate other Content Presenters (basic or aggregated). The information to be presented by a Content Presenter is specified through a concept model where all entities¹ are connected through relations (directly or indirectly).

By *connected concept model fragment* we mean a concept model fragment containing at least one entity, where all entities are either directly or indirectly associated to all other entities in the model fragment.

A model fragment containing exactly one entity is connected. In a UML class model, a connected model fragment is a network, where the nodes are the entities and the edges are any of UMLs association types. Figure 2.5 (adapted from Nilsson (2010)) contains an example of a concept model containing three connected model fragments, indicated by the dashed outlines. As can be seen from the example, one of the model fragments contains a single entity with no associations to other entities.

¹ Some modelling notations use other terms. E.g., UML class diagrams use the term *class*. We use *entity* in its meaning as a natural language concept, and not restricted to the way it is used in Entity-Relationship modelling notation.

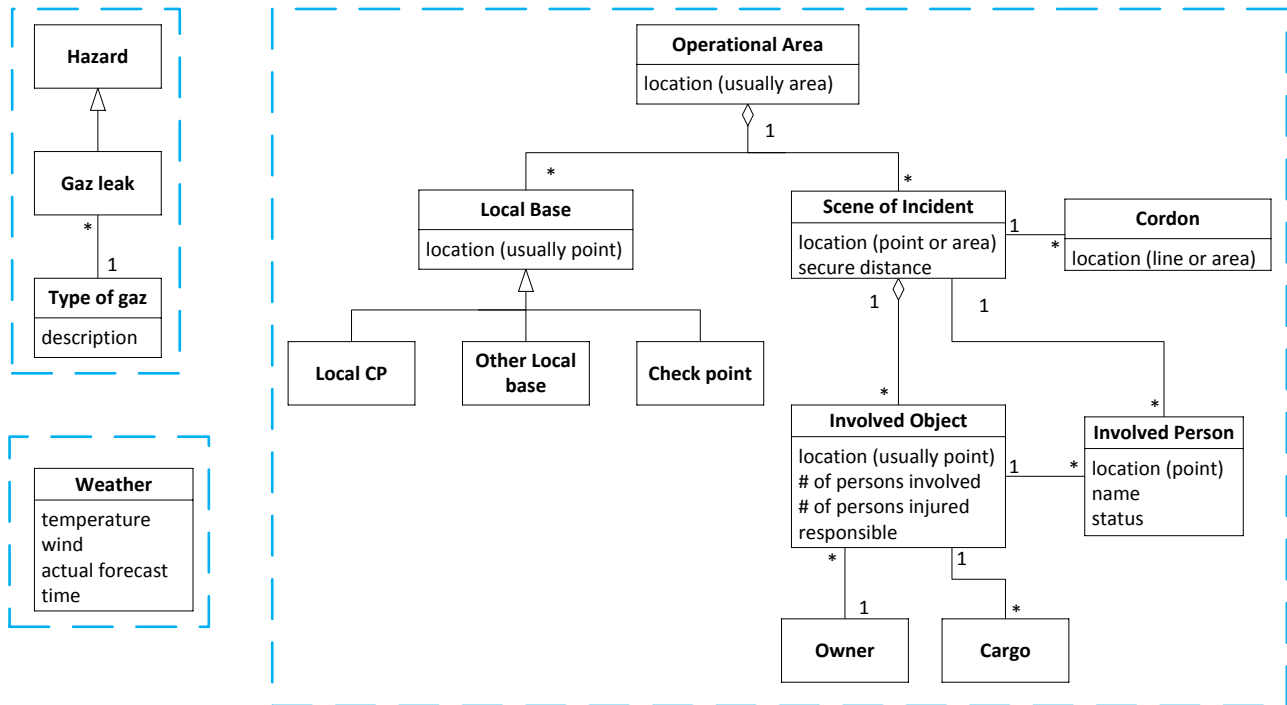


Figure 2.5 - Example model with three connected model fragments

A connected concept model fragment has an entry point, denoted an *entity anchor*.

By *entity anchor* we mean one specific entity in a connected concept model fragment that will be used to transform the model fragment to a hierarchical structure.

In order to determine the instances to present in a final user interface based on a Content Presenter at run-time, the connected concept model fragment (which in the general case is a network) must be transformed to a hierarchy with the anchor entity as the root. The anchor is also needed when composing Content Presenters into an Aggregated Content Presenter.

2.3 Main Constructs in FLUIDE-D

FLUIDE-D contains variants of the four main constructs in FLUIDE-A, using the same names with the suffix *Design*. The interactor design constructs in FLUIDE-D are used to specify which parts of the domain and task models that are to be included in a final user interface (FUI). The main role of FLUIDE-D is to provide the means for specifying enough additional information to the FLUIDE-A specification to enable a generic transformation mechanism to generate a final user interface. FLUIDE-D's core is the library of user interface patterns, operationalized in the *View* constructs. Views are used to specify how some part a FLUIDE-A specification is to be presented on a given user interface platform using certain modalities and user interface styles.

Below, we provide definition for the main constructs for specifying designs in FLUIDE-D.

By *interactor design construct* we mean a language construct for specifying *interactor design instances*, operationalized as one of the four main construct in FLUIDE-D, i.e. either *Content Presenter Design*, *Task Supporter Design*, *Work Supporter Design* or *Category Manager Design*.

By *interactor design instance* we mean the concrete use of an interactor design construct as part of the specification of the user interface for an application.

By *Category Manager Design* we mean a construct that makes it possible to specify how a Category Manager should be presented on a given user interface platform using certain user interface styles through wrapping selected designs for a subset of the member Work Supporters, as well as Content Presenters into a set of views.

By *Work Supporter Design* we mean a construct that makes it possible to specify how a Work Supporter should be presented on a given user interface platform using certain user interface styles through wrapping selected designs for a subset of the member Task and Work Supporters into a set of views.

By *Task Supporter Design* we mean a construct that makes it possible to specify how a Task Supporter should be presented on a given user interface platform using certain user interface styles through wrapping selected designs for a subset of the member Content Presenters into a set of views.

By *Content Presenter Design* we mean a construct that makes it possible to specify how a Content Presenter should be presented on a given user interface platform using certain user interface styles through wrapping one or more subsets of the connected concept model fragment into a set of views.

In the same way as FLUIDE-A, FLUIDE-D provides two kinds of Content Presenter Design and Work Supporter Design constructs, i.e. *basic* and *aggregated*. Aggregated Content Presenter Designs aggregate other Content Presenter Designs (basic or aggregated), while Aggregated Work Supporter Designs aggregate other Work Supporter Designs (basic or aggregated).

The functionality offered by a final user interface which is the result of applying a design will typically be restricted to CRUD (create, read, update, delete) and possible generic functionality for the user interface style. For example a map based user interfaces will have certain functionality for all information presented in a given way – like an information window that is displayed when an icon in the map is clicked.

In Figure 2.6 we give a schematic overview of the connection between the specification languages, user interface specifications made using the languages, and user interfaces generated from the specifications.

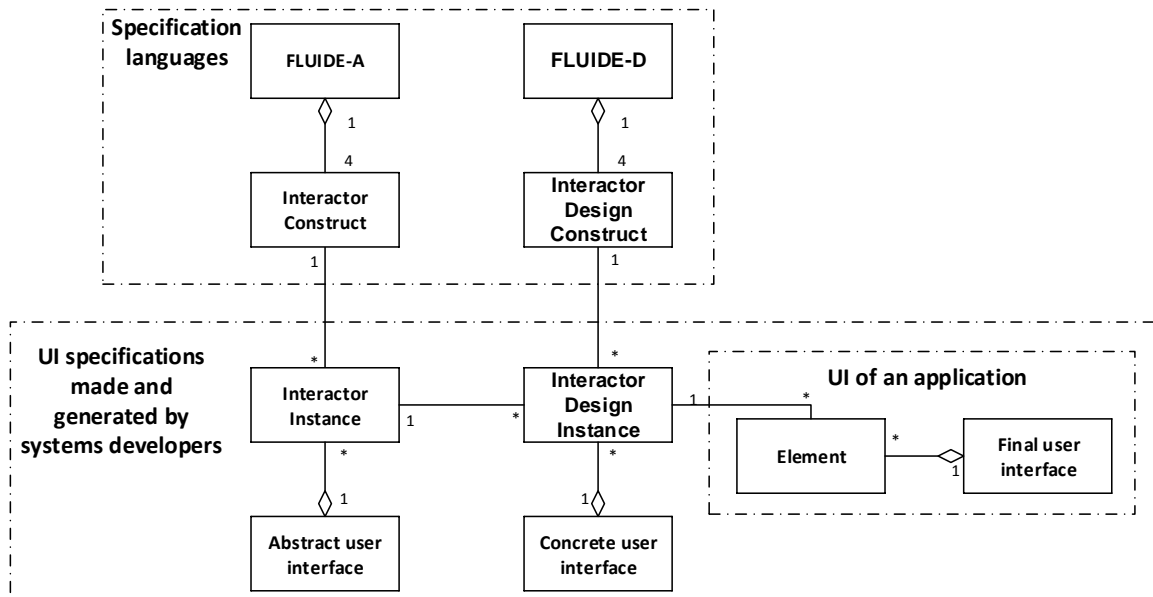


Figure 2.6. UML class diagram summarizing the FLUIDE specification languages and how they are connected and used

At the language level, FLUIDE-A and FLUIDE-D contain the interactor and interactor Design constructs just presented. When a systems developer specifies a user interface, the AUI is specified through making a set of interactor instances. The CUI contains a one or more interactor design instances for each interactor instance. The final user interface may be automatically generated from the CUI, and consists of a set of user interface elements on a given platform.

3 The FLUIDE-A Language

In this section, we present the syntax and semantics of the FLUIDE-A language. The syntax is presented both as a concrete syntax in the form of a graphical syntax, and as an abstract syntax expressed in Extended Backus-Naur Form (EBNF). The semantics is presented as a natural language semantics giving production rules which may be used to translate an expression in EBNF to one or more English sentences. For each interactor construct in FLUIDE-A we provide one example specification. The examples expressed using the graphical syntax are provided as part of the sections explaining the graphical syntax. The examples expressed in EBNF and the corresponding English sentences describing the semantics of the example are presented together directly after the definition of the semantics for the interactor construct at hand.

In the EBNF definition of the abstract syntax, terminal symbols are expressed using **bold** font, while non-terminal symbols are expressed using *italic* font. EBNF operators and brackets are expressed using normal font. As a convention, all names ending with the word "identifier" should be considered implicitly defined as alphanumeric identifiers/names for artefacts occurring in a specification written in FLUIDE-A. In the same way, data types like *string*, *integer*, and *picture* should be considered implicitly defined. Furthermore, *expression* should be considered implicitly defined as an arbitrary expression that is given at design-time or may be determined at run-time based on available information.

The semantics presented in the sections below give production rules which may be used to translate expressions following the abstract syntax into English sentences. The semantics is defined by a function $\llbracket \cdot \rrbracket$ that takes a FLUIDE-A expression as argument and returns a fragment of English prose. The production rules focus on the contents of the produced sentences, not details regarding formatting and grammar. Thus, line breaks, indents, bullets, etc. that are used in the English text in the examples below are not included in the rules. The same is the case for capitalization of identifiers, the use of correct verb forms for singular or plural nouns, as well as delimiters (commas, semicolons and conjunctions) between elements in the sentences produced from sets of elements in a specification. In the production rules, the EBNF expressions use the same formatting rules as in the EBNF definitions above, while the fixed English phrases are expressed using normal font.

As a convention, the semantics of all implicitly defined names in the EBNF (identifiers and expressions) should be considered implicitly defined as the identifiers/names for artefacts occurring in a specification written in FLUIDE-A. As a short hand notation in the production rules, the identifiers used for these elements in the EBNF definition are used to denote their semantics in the production rules. For example when *basic_content_presenter_identifier* occurs on the right hand side of the "="-sign in a production rule, it means the name used for a given presenter in a specification.

As the semantics of the control structures in EBNF (" $\{\dots\}$ ", " $[\dots]$ ", " $\dots|\dots$ ", etc.) is defined and well-known, we inherit this semantics in our production rule. Thus, we take it for given for example that $\llbracket \{\dots\} \rrbracket$ by definition equals $\{\llbracket \dots \rrbracket\}$. This also means that for example for elements in a specification that may occur zero or more times, the production rule (or the part of the production rule associated with the set) will only be applied if there actually are some elements in the set.

We start by giving the syntax and semantics for the interactor construct. This section contains the common graphical syntax used in FLUIDE-A. Thereafter, the syntax and semantics of the four interactor constructs in FLUIDE-A (Content Presenter, Task Supporter, Work Supporter and Category Manager) are given in separate sections for each of the constructs. The Content Presenters and the Work Supporters are presented in separate sections for the basic and aggregated variants. The Task Supporters and Category Managers are only available in one variant. When we present the constructs, we start with Basic Content Presenters, and move up in the aggregation hierarchy finishing with Category Managers.

3.1 Interactor

This section is primarily a definition of the common graphical syntax for FLUIDE-A. We also provide an EBNF definition of the abstract syntax for interactors. As there are no terminal symbols in the abstract syntax, the semantics for all parts of the EBNF definition of the interactor construct is given by the general rules for the semantics given in the introduction part of Section 3 above.

3.1.1 Graphical Syntax

In the graphical notation, all the interactor constructs in FLUIDE-A use the basic layout shown in Figure 3.1.

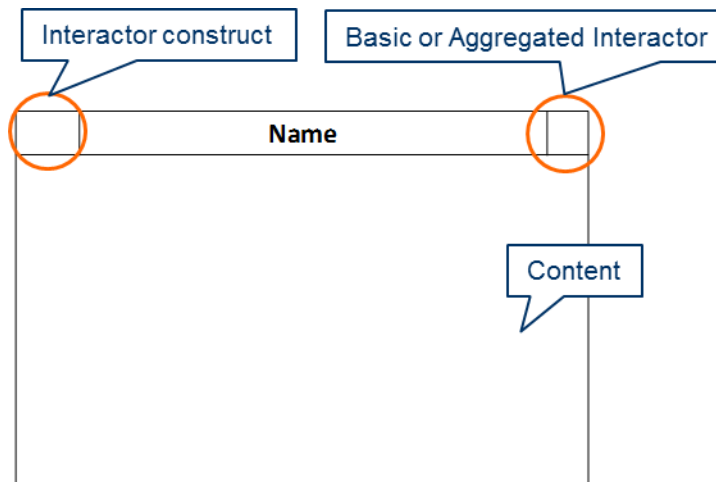


Figure 3.1 - Basic layout of the interactor constructs in FLUIDE-A

An interactor instance is represented as a rectangle with a top border resembling a window. The top border contains the name of the interactor instance, as well as an icon on the top left denoting interactor construct used and an icon on the top right denoting whether the instance is basic or aggregated. Instances of all constructs may be basic, while only Content Presenters and Work Supporters may be aggregated. The content part (canvas) underneath the top border is used for presenting the content of the interactor instance. The content is different for instances of the different constructs.

Table 3.1 shows the icons used for the four interactor constructs, while Table 3.2 shows the icons used to denote whether an interactor instance is basic or aggregated.

Table 3.1 – Icons used for the four interactor constructs





Interactor construct	Icon
Content Presenter	
Task Supporter	TS
Work Supporter	
Category Manager	CM

Table 3.2 – Icons used to denote whether an interactor instance is basic or aggregated

Interactor construct variant	Icon
Basic	
Aggregated	

3.1.2 Abstract Syntax

interactor = *content_presenter* | *task_supporter* | *work_supporter* | *category_manager*;

content_presenter = *basic_content_presenter* | *aggregated_content_presenter*;

work_supporter = *basic_work_supporter* | *aggregated_work_supporter*;

3.2 Basic Content Presenter

In this section, we provide the syntax and semantics of the basic variant of the Content Presenter construct.

3.2.1 Graphical Syntax

The concrete syntax of FLUIDE-A uses a subset of the UML class model syntax (extended with the anchor) to express the concept models of Basic Content Presenters. In the graphical notation, the concept model is located in the content part (canvas) of a Basic Content Presenter. Figure 3.2 gives an example of a Basic Content Presenter, with explanations of certain parts.

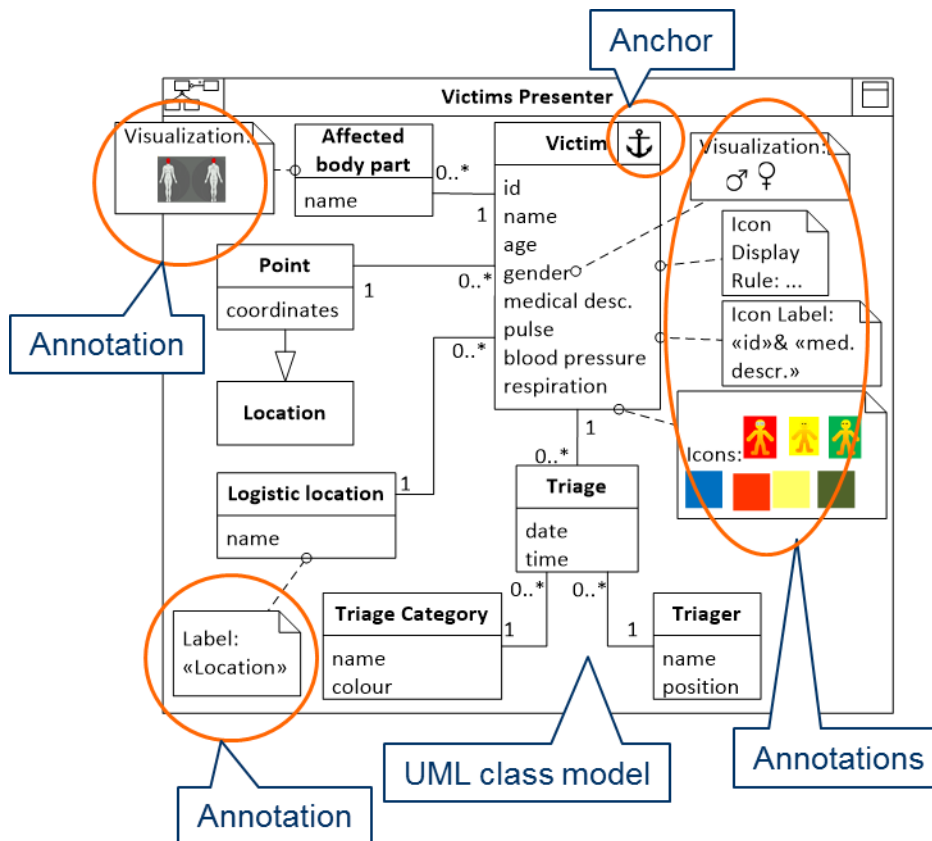


Figure 3.2 - A Basic Content Presenter in FLUIDE-A

There is always one anchor in the concept model of a Basic Content Presenter. Platform-independent visual properties are expressed using UML annotations. The annotations shown in Figure 3.2 specify icons, display rules, labels and visualizations.

3.2.2 Abstract Syntax

Figure 3.3 provides a concept model explaining the main concepts used when specifying a Basic Content Presenter.

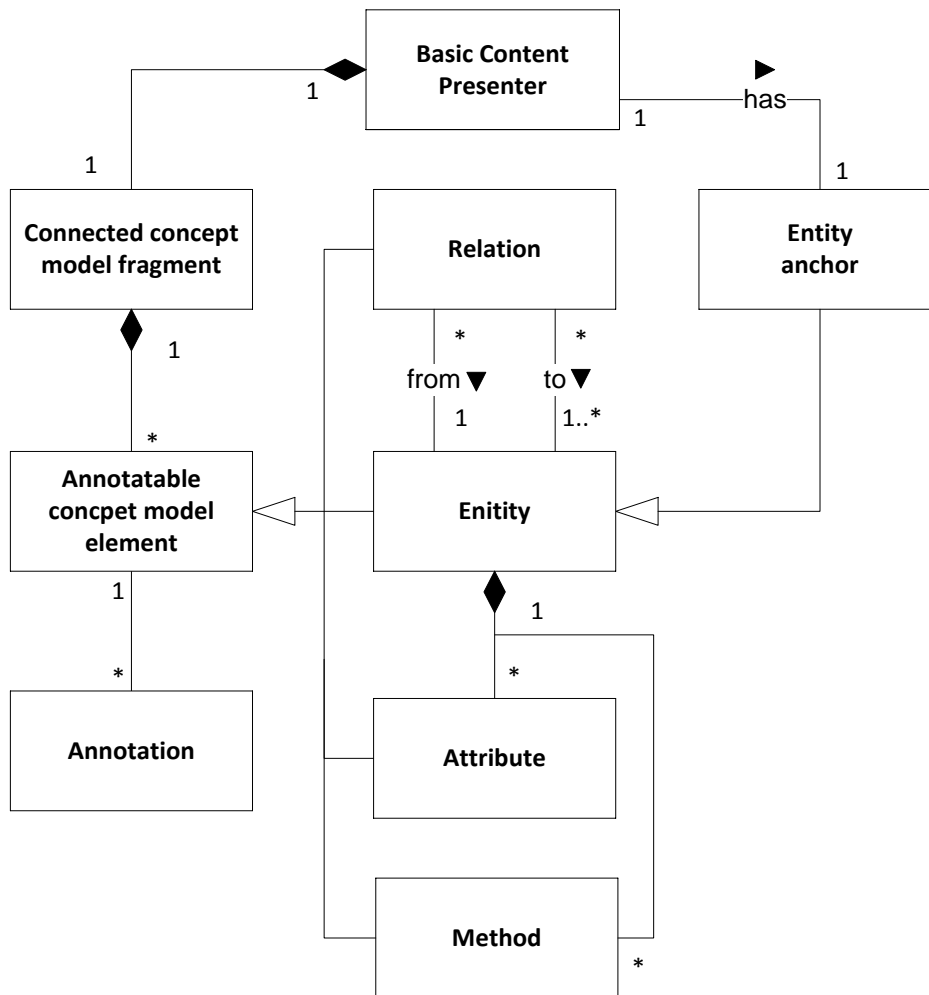


Figure 3.3 – Concept model describing the means for specifying Basic Content Presenters in FLUIDE-A

A Basic Content Presenter is mainly a definition of the connected concept model fragment. All members of such a fragment (entities, attributes, methods and relations) may have connected annotations, giving for example type, uniqueness, label, as well as domain information that is useful for choosing user interface controls automatically. The concept model in Figure 3.3 does not distinct between the three types of relations that may be specified in the EBNF.

```
basic_content_presenter =
    bcp(basic_content_presenter_identifier, connected_concept_model_fragment, anchor_entity);
```

```
connected_concept_model_fragment = ccmf({entity_with_attributes}-, {relation});
```

```
anchor_entity = entity_identifier;
```

```
entity_with_attributes = entwa(entity_identifier, {attribute}, {method}, {annotation});
```

```
relation = generalization | association | containment;
```

```
attribute = att(attribute_identifier, {annotation});
```

method = **met**(*method_identifier*, {*parameter*}, {*annotation*});

annotation = **ann**(*annotation_identifier*, *expression*);

parameter = **param**(*parameter_identifier*, *type-identifier*);

generalization = **gen**(*generalized_entity*, {*specialized_entity*}-, {*annotation*});

generalized_entity = *entity_identifier*;

specialized_entity = *entity_identifier*;

association =
 asso(*association_identifier*, (*from_entity*, *from_cardinality*), (*to_entity*, *to_cardinality*),
 {*annotation*});

from_entity = *entity_identifier*;

from_cardinality = *cardinality*;

cardinality = **one|many**;

to_entity = *entity_identifier*;

to_cardinality = *cardinality*;

containment = **cont**(*containment_identifier*, *from_entity*, {(*to_entity*, *to_cardinality*)}-, {*annotation*});

3.2.3 Semantics

[[*basic_content_presenter*]]=

[[**bcp**(*basic_content_presenter_identifier*, *connected_concept_model_fragment*, *anchor_entity*)]]

[[**bcp**(*basic_content_presenter_identifier*, *connected_concept_model_fragment*, *anchor_entity*)]] =

basic_content_presenter_identifier is a part of a user interface, consisting of [[*logical_unit*]].
basic_content_presenter_identifier presents instances from the extent of
[[*connected_concept_model_fragment*]]. The starting point for determining the extent of the concept
model is *anchor_entity*.

[[*logical_unit*]] =

a part of a "window", one "window", or a limited number of "windows" between which there exists
immediate mechanisms for easy navigation and visual connections

[[*connected_concept_model_fragment*]] =

[[**ccmf**(*{entity_with_attributes}*-,*{relation}*)]]

[[**ccmf**(*{entity_with_attributes}*-, *{relation}*)]] =

a concept model, containing the following main blocks of information: {*{entity_with_attributes}* }
 The user interface part also contains the following visual and behavioural connections: {*{relation}* }

[[*entity_with_attributes*]] =

[[**entwa**(*entity_identifier*, *{attribute}*, *{method}*, *{annotation}*)]]

[[**entwa**(*entity_identifier*, *{attribute}*, *{method}*, *{annotation}*)]] =

entity_identifier is presented as [[*logical_unit*]] guided by the information that {*{annotation}* }.
 Within the realms of the presentation of *entity_identifier*, visual (or other) means are used to present the values of {*{attribute}* } and {*{method}* }

[[*attribute*]] =

[[**att**(*attribute_identifier*, *{annotation}*)]]

[[**att**(*attribute_identifier*, *{annotation}*)]] =

attribute_identifier, guided by the information that {*{annotation}* }

[[*method*]] =

[[**met**(*method_identifier*, *{parameter}*, *{annotation}*)]]

[[**met**(*method_identifier*, *{parameter}*, *{annotation}*)]] =

method_identifier with the parameters {*{parameter}* }, guided by the information that {*{annotation}* }

[[*parameter*]] =

[[**param**(*parameter_identifier*, *type-identifier*)]]

[[**param**(*parameter_identifier*, *type-identifier*)]] =

parameter_identifier having the type *type-identifier*

[[*annotation*]] =

[[**ann**(*annotation_identifier*, *expression*)]]

[[**ann**(*annotation_identifier*, *expression*)]] =

annotation_identifier is *expression*

[[*relation*]] =

[[*generalization*]] | [[*association*]] | [[*containment*]]

[[*generalization*]] =

[[**gen**(*generalized_entity*, {*specialized_entity*}-, {*annotation*})]]

[[**gen**(*generalized_entity*, {*specialized_entity*}-, {*annotation*})]] =

generalized_entity, {*specialized_entity*} will usually be presented using the same means, guided by the information that { [[*annotation*]] }

[[*association*]] =

[[**asso**(*association_identifier*, (*from_entity*, *from_cardinality*), (*to_entity*, *to_cardinality*), {*annotation*})]]

[[**asso**(*association_identifier*, (*from_entity*, *from_cardinality*), (*to_entity*, *to_cardinality*), {*annotation*})]] =

A connection called *association_identifier* from [[*from_cardinality*]] of the presentation of *from_entity* to [[*to_cardinality*]] of the presentation of *to_entity*, guided by the information that { [[*annotation*]] }

[[*cardinality*]] =

one instance | any natural number of instances (including 0)

[[*containment*]] =

[[**cont**(*containment_identifier*, *from_entity*, {(*to_entity*, *to_cardinality*)}-, {*annotation*})]]

[[**cont**(*containment_identifier*, *from_entity*, {(*to_entity*, *to_cardinality*)}-, {*annotation*})]] =

The values in the presentation of { [[(*to_entity*, *to_cardinality*)]] } are determined by the value in the presentation of *from_entity*, guided by the information that { [[*annotation*]] }

[[(*to_entity*, *to_cardinality*)]] =

[[*to_cardinality*]] of *to_entity*

3.2.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Basic Content Presenters in FLUIDE-A. The example is a subset of the specification of the Basic Content Presenter in Figure 3.2.

3.2.4.1 EBNF Specification

```

bcp(Victim Presenter,
    ccmf(
        entwa(Victim,
            att(id,),
            att(name,),
            att(age,),
            att(gender, ann(Visualization, Icons(male, female))),
            /* specification of four more attributes omitted */
            /* no methods*/,
            /* annotations for the entity – one annotation omitted*/
            ann(Icon Label, value(id) & value(medical description)),
            ann(Icons, iconCollection(triage))
        ),
        entwa(Logistic location,
            att(name, ann(Label, "Location")),
            /* no methods*/,
            /* no annotations for the entity */
        ),
        entwa(Point,
            att(coordinates,),
            /* no methods*/,
            /* no annotations for the entity */
        ),
        entwa(Location,
            /* no attributes*/,
            /* no methods*/,
            /* no annotations for the entity */
        ),
        /* specification of four more entities omitted */
        asso(victimLogisticLocation, (Victim, many), (LogisticLocation, one), ),
        asso(victimLocation, (Victim, many), (Point, one), ),
        gen(Location, Point, )
        /* specification of four more relations omitted */
    ), /* end of ccmf specification */
    Victim /* anchor */
) /* end of bcp specification */

```

3.2.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 3.2.3 on the EBNF specification just presented results in the following English sentences:

Victim Presenter is a part of a user interface, consisting of a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual

connections. **Victim Presenter** presents instances from the extent of a concept model, containing the following main blocks of information:

Victim is presented as a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual connections guided by the information that Icon Label is "value(id) & value(medical description)" and Icons is iconCollection(triage). Within the realms of the presentation of **Victim**, visual (or other) means are used to present the values of

- id
- name
- age
- gender, guided by the information that Visualization is Icons(male, female)

Logistic location is presented as a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual connections. Within the realms of the presentation of **Logistic location**, visual (or other) means are used to present the values of

- name, guided by the information that Label is "Location"

Point is presented as a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual connections. Within the realms of the presentation of **Point**, visual (or other) means are used to present the values of

- coordinates

Location is presented as a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual connections.

The user interface part also contains the following visual and behavioural connections:

A connection called **victimLogisticLocation** from any natural number of instances (including 0) of the presentation of **Victim** to one instance of the presentation of **LogisticLocation**.

A connection called **victimLocation** from any natural number of instances (including 0) of the presentation of **Victim** to one instance of the presentation of **Point**.

Location and **Point** will usually be presented using the same means.

The starting point for determining the extent of the concept model is **Victim**.

3.3 Aggregated Content Presenter

In this section, we provide the syntax and semantics of the aggregated variant of the Content Presenter construct, i.e. Content Presenters that have other Content Presenters as children, also allowing relations between the child Content Presenters.

3.3.1 Graphical Syntax

Aggregated Content Presenters aggregate other Content Presenters (Basic or Aggregated). The connected concept model fragment of an Aggregated Content Presenter is specified indirectly, and thus expressed implicitly. The aggregated presenter inherits the model fragments of all its children. The relations in the aggregated presenter is operationalized in its implicit concept model by connecting the anchor entities of the concept models of the child presenters. Entities that are shared by a number of member presenters occur only once in the implicit concept model. The anchor of the aggregated presenter is specified by identifying which of the child presenters the anchor is inherited from. In the graphical notation, the child presenters are located in the content part of an Aggregated Content Presenter. Figure 3.4 gives an example of an Aggregated Content Presenter, with explanations of certain parts.

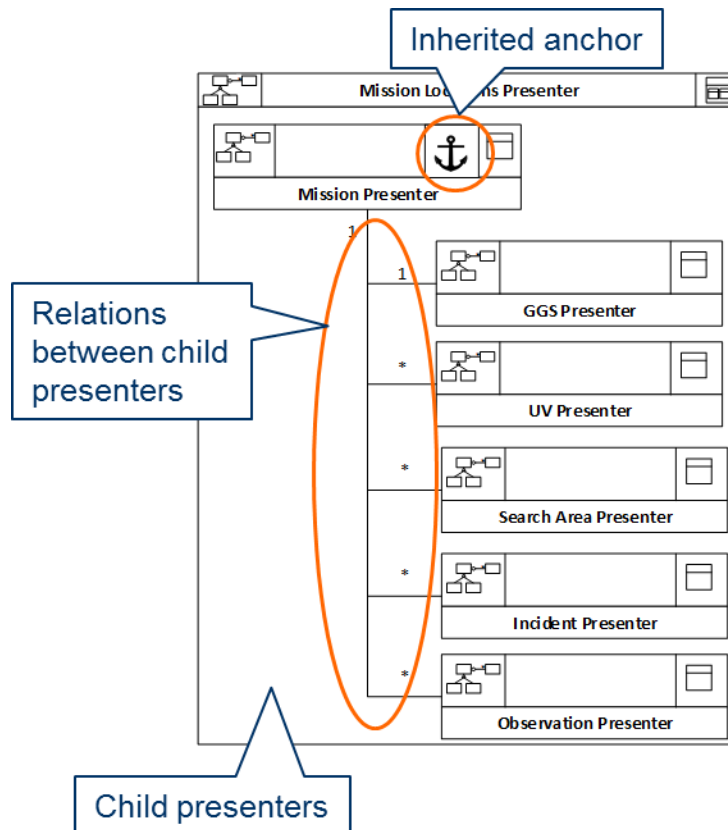


Figure 3.4 - An Aggregated Content Presenter in FLUIDE-A

Only the border part of the child presenters is shown in the aggregated one. The names of the child presenters are shown in their content part. The relations between the child presenters are expressed using the concrete syntax of relations in UML class models, including cardinalities. To ensure that the implicit concept model is connected, all child presenters must be connected through relations (directly or indirectly).

3.3.2 Abstract Syntax

Figure 3.5 provides a concept model explaining the main concepts used when specifying an Aggregated Content Presenter.

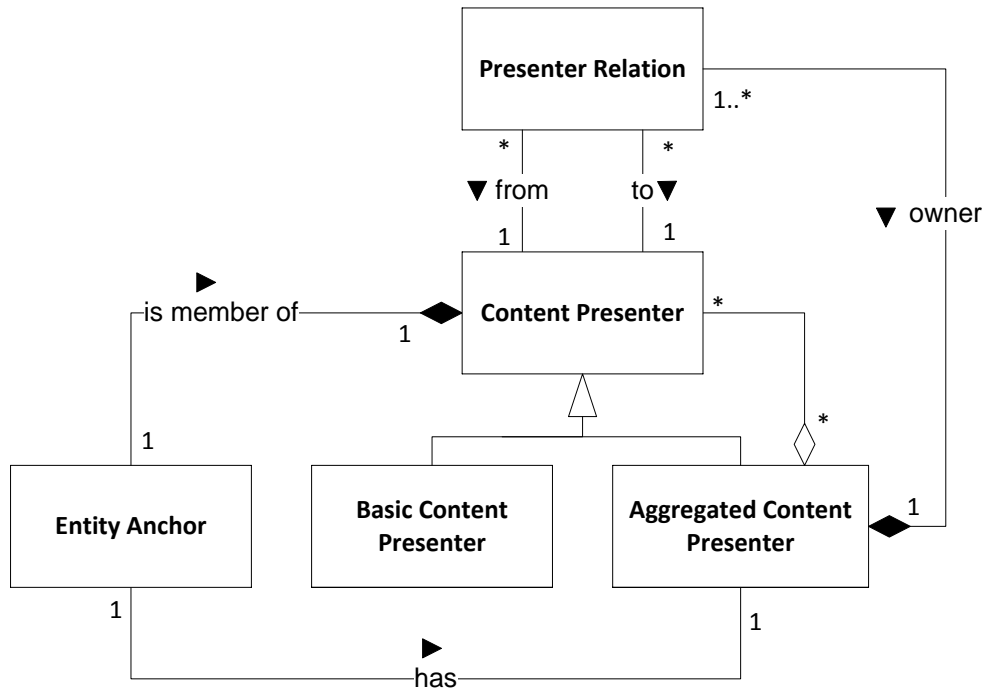


Figure 3.5 - Concept model describing the means for specifying Aggregated Content Presenters in FLUIDE-A

An Aggregated Content Presenter contains references to a number of Content Presenters that are either basic or aggregated. It also identifies the anchor for the Aggregated Content Presenter through identifying the Content Presenter from which the anchor is inherited. The Aggregated Content Presenter also has a number of Presenter Relations connecting pairs of member presenters. The concept model in Figure 3.5 does not distinct between the two types of presenter relations that may be specified in the EBNF.

```

aggregated_content_presenter =
    acp(aggregated_content_presenter_identifier, anchor_entity, {(content_presenter)}-,
        {presenter_relation}-);
  
```

```

anchor_entity = entity_identifier;
  
```

```

content_presenter = aggregated_content_presenter | basic_content_presenter;
  
```

```

presenter_relation = presenter_association | presenter_containment;
  
```

```

presenter_association = presasso( (from_anchor, from_cardinality), (to_anchor, to_cardinality));
  
```

```

from_anchor = entity_identifier;
  
```

```

from_cardinality = cardinality;
  
```

```

cardinality = one|many;
  
```

to_anchor = *entity_identifier*;

to_cardinality = *cardinality*;

presenter_containment = **prescont**(*from_anchor*, {(*to_anchor*, *to_cardinality*)}-);

3.3.3 Semantics

[[*aggregated_content_presenter*]] =

[[**acp**(*aggregated_content_presenter_identifier*, *anchor_entity*, {(*content_presenter*)}-,
{*presenter_relation*}-)]]

[[**acp**(*aggregated_content_presenter_identifier*, *anchor_entity*, {(*content_presenter*)}-,
{*presenter_relation*}-)]] =

aggregated_content_presenter_identifier is a part of a user interface, consisting of [[*logical_unit*]].
aggregated_content_presenter_identifier presents instances from the extent of a concept model,
containing the following main blocks of information:

{ /* These brackets represent the set of child presenters */

{ /* These brackets represent the set of entities for each of the child presenter */

[[*entity_with_attributes*]]

}

}

The user interface part also contains the following visual and behavioural connections:

{ /* These brackets represent the set of child presenters */

{ /* These brackets represent the set of relations for each of the child presenter */

[[*relation*]]

}

}

In addition, there is:

{ [[*presenter_relation*]]}

The starting point for determining the extent of the concept model is *anchor_entity*.

[[*logical_unit*]] =

a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual connections

[[*cardinality*]] =

one instance | any natural number of instances (including 0)

[[*presenter_relation*]] =

[[*presenter_association*]] | [[*presenter_containment*]]

[[*presenter_association*]] =

[[**presasso**((*from_anchor*, *from_cardinality*), (*to_anchor*, *to_cardinality*))]]

[[**presasso**((*from_anchor*, *from_cardinality*), (*to_anchor*, *to_cardinality*))]] =

A connection from [[*from_cardinality*]] of the presentation of *from_anchor* to [[*to_cardinality*]] of the presentation of *to_anchor*

[[*presenter_containment*]]=

[[**prescont**(*from_anchor*, {(*to_anchor*, *to_cardinality*)}-)]]

[[**prescont**(*from_anchor*, {(*to_anchor*, *to_cardinality*)}-)] =

The values in the presentation of [[*to_cardinality*]] of *to_anchor* } are determined by the value in the presentation of *from_anchor*

3.3.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Aggregated Content Presenters in FLUIDE-A. The example is a subset of the specification of the Aggregated Content Presenter in Figure 3.4.

3.3.4.1 EBNF Specification

To enable sufficiently rich semantic description, a copy of the whole specification of the member presenters are included in the specifications of the aggregated presenter.

```

acp(Mission Locations Presenter, Mission,
    bcp(Mission Presenter,
        ccmf(
            entwa(Mission,
                att(name,),
                att(id,),
                att(description, ann(Label, "Purpose")),

```

```

        /* specification of three more attributes omitted */
        /* no methods*/,
        /* annotation for the entity */
        ann(Colouring rule, <expression>
    ),
    entwa(Observation,
        att(description,)
        /* no methods*/,
        /* no annotation for the entity */
    ),
    entwa(Search Area,
        att(type,)
        /* no methods*/,
        /* no annotation for the entity */
    ),
    /* specification of four more entities omitted */
    asso(missionObservations, (Mission, one), (Observation, many), ),
    asso(missionSearchArea, (Mission, one), (Search Area, many), )
    /* specification of three more relations omitted */
    ), /* end of ccmf specification */
Mission /* anchor */
), /* end of Mission Presenter bcp specification */

bcp(GGS Presenter,
    ccmf(
        entwa(Generic Ground Station,
            /* no attributes */
            /* no methods*/,
            /* annotation for the entity */
            ann(Icon, iconCollection(GGS))
        ),
        entwa(Point,
            att(coordinates,)
            /* no methods*/,
            /* no annotation for the entity */
        )
        asso(GGSLocation, (Generic Ground Station, one), (Point, one), ),
    ), /* end of ccmf specification */
Generic Ground Station /* anchor */
), /* end of GGS Presenter bcp specification */
/* specification of three more bcps omitted */
/* presenter relations */
presasso((Mission, one), (Generic Ground Station, one)),
/* specification of four more presenter relations omitted */
) /* end of acp specification */

```

3.3.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 3.3.3 on the EBNF specification just presented results in the following English sentences:

Mission Locations Presenter is a part of a user interface, consisting of a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual connections. **Mission Locations Presenter** presents instances from the extent of a concept model, containing the following main blocks of information:

Mission is presented as a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual connections guided by the information that Colouring rule is <expression>. Within the realms of the presentation of **Mission**, visual (or other) means are used to present the values of

- name
- id
- description, guided by the information that Label is " Purpose"

Observation is presented as a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual connections. Within the realms of the presentation of **Observation**, visual (or other) means are used to present the values of

- description

Search Area is presented as a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual connections. Within the realms of the presentation of **Search Area**, visual (or other) means are used to present the values of

- type

Generic Ground Station is presented as a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual connections guided by the information that Icon is iconCollection(GGS).

Point is presented as a part of a "window", one "window", or a limited number of "windows" between which there exists immediate mechanisms for easy navigation and visual connections. Within the realms of the presentation of **Point**, visual (or other) means are used to present the values of

- coordinates

The user interface part also contains the following visual and behavioural connections:

A connection called **missionObservations** from one instance of the presentation of **Mission** to any natural number of instances (including 0) of the presentation of **Observation**.

A connection called **missionSearchArea** from one instance of the presentation of **Mission** to any natural number of instances (including 0) of the presentation of **Search Area**.

A connection called **GGSLocation** from one instance of the presentation of **Generic Ground Station** one instance of the presentation of **Point**.

In addition, there is:

*A connection from one instance of the presentation of **Mission** to one instance of the presentation of **Generic Ground Station**.*

*The starting point for determining the extent of the concept model is **Mission**.*

3.4 Task Supporter

In this section, we provide the syntax and semantics of the Task Supporter construct. Task Supporters are only provided in a basic variant.

3.4.1 Graphical Syntax

The Task Supporter construct is used to specify the information that is needed for solving one specific task, through identifying one or more Content Presenters that manages the information that is needed to support performing the task. A Task Supporter thus aggregates a number of Content Presenters (Basic or Aggregated). Unlike a Content Presenter, a Task Supporter does not have a connected concept model fragment. Thus it does not have an anchor, and it is not possible to specify any relations between its child presenters. In the graphical notation, the child presenters are located in the content part of a Task Supporter. Figure 3.6 gives an example of a Task Supporter, with explanations of certain parts.

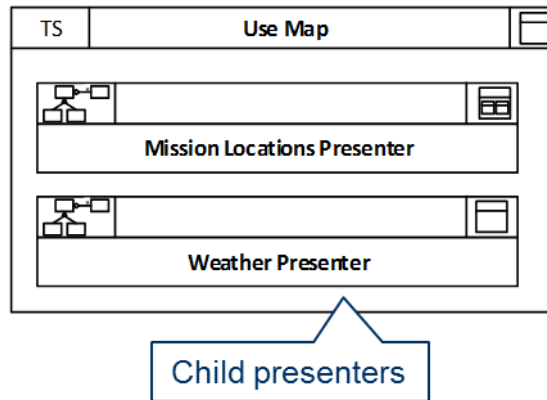


Figure 3.6 - A Task Supporter in FLUIDE-A

Only the border part of the child presenters is shown in the Task Supporter. The names of the child presenters are shown in their content part. In the example in Figure 3.6, one of the children is a Basic and the other is an Aggregated Content Presenter.

3.4.2 Abstract Syntax

Figure 3.7 provides a concept model explaining the main concepts used when specifying a Task Supporter.

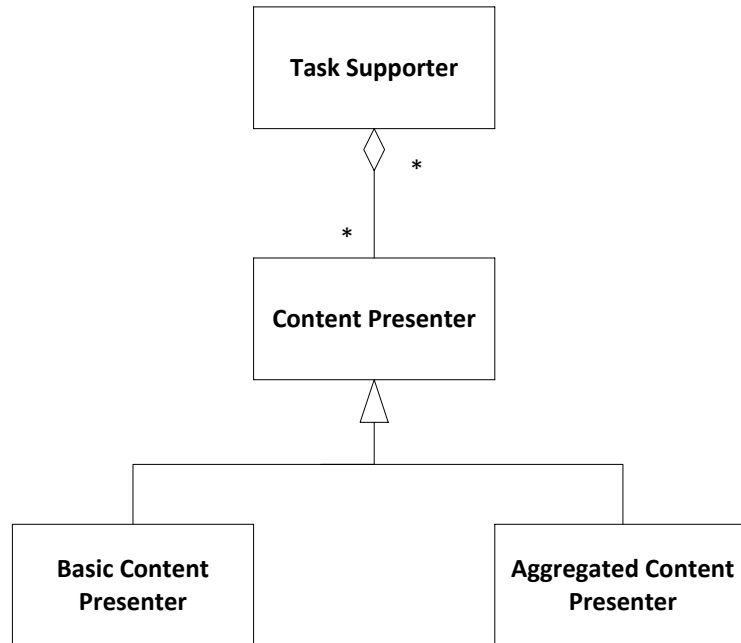


Figure 3.7 - Concept model describing the means for specifying Task Supporters in FLUIDE-A

As can be seen from the EBNF definition below, the specification of a Task Supporter includes the name of the task the Task Supporter supports.

$task_supporter = ts(task_supporter_identifier, task_identifier, \{content_presenter_identifier\}-);$

$content_presenter_identifier =$
 $aggregated_content_presenter_identifier \mid basic_content_presenter_identifier;$

3.4.3 Semantics

$\llbracket task_supporter \rrbracket =$

$\llbracket ts(task_supporter_identifier, task_identifier, \{(content_presenter_identifier)\}-) \rrbracket$

$\llbracket ts(task_supporter_identifier, task_identifier, \{(content_presenter_identifier)\}-) \rrbracket =$

$task_supporter_identifier$ is a part of a user interface supporting the user task $task_identifier$.
 $task_supporter_identifier$ contains the user interface parts $\{content_presenter_identifier\}$. These user interface parts have no specific connections.

With these production rules, only the identifiers (the names) of the Content Presenters that are member of the Task Supporter are included in the resulting sentences. To investigate the semantics of the corresponding presenters, the production rules for the presenters must be used. The reason for this solution is that a Task Supporter (in contrast to an Aggregated Content Presenter) does not change the semantics of the member presenters.

3.4.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Task Supporters in FLUIDE-A. The example is the specification of the Task Supporter in Figure 3.6.

3.4.4.1 EBNF Specification

ts(User Map, Use Map, Mission Location Presenter, Weather Presenter)

3.4.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 3.4.3 on the EBNF specification just presented results in the following English sentences:

User Map is a part of a user interface supporting the user task **User Map**. *User Map* contains the user interface parts **Mission Location Presenter** and **Weather Presenter**. These user interface parts have no specific connections.

3.5 Basic Work Supporter

In this section, we provide the syntax and semantics of the basic variant of the Work Supporter construct.

3.5.1 Graphical Syntax

The work that is supported by a Work Supporter is specified through a task model giving the hierarchical and temporal structures of the work, as well as which tasks that are supported by which user interface (if any). The concrete syntax of FLUIDE-A uses a neutral hierarchical task model syntax to express the hierarchical structure. The temporal structure is expressed using operators. The graphical notation used for operators is adopted from a subset of the temporal relationships used in ConcurrentTaskTrees (CTT) notation (Paternò, 1999). In the graphical notation, the task model is located in the content part of a Basic Work Supporter. Figure 3.8 gives an example of a Basic Work Supporter, with explanations of certain parts.

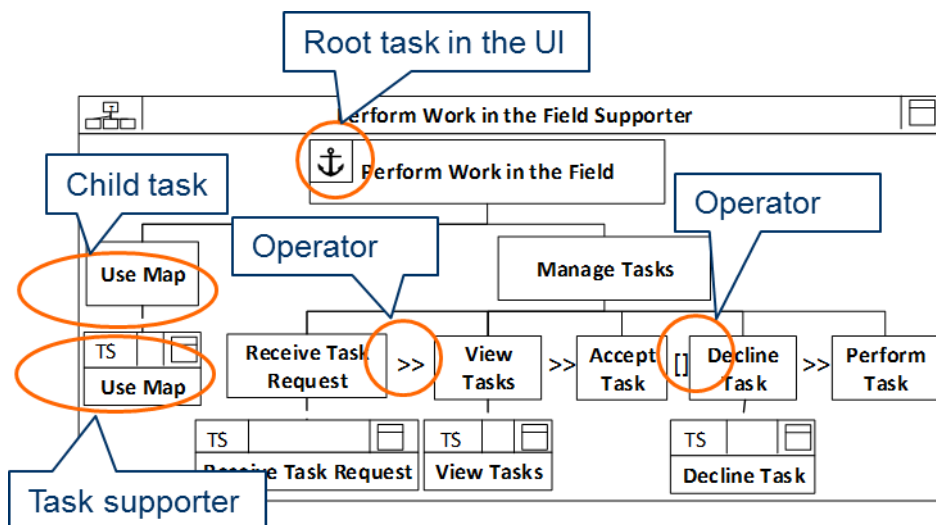


Figure 3.8 - A Basic Work Supporter in FLUIDE-A

There are eight tasks organized in three levels in the task model in the *Perform Work in the Field Supporter*, of which four of the leaf tasks have Task Supporters. Tasks with children may also have Tasks Supporters. Only the border part of the Task Supporters is shown in the Work Supporter, and their names are shown in their content part. The task model contains some operators: “>>” indicates sequence in task performance, while “[]” indicates a choice between tasks. The anchor is always located at the root task.

3.5.2 Abstract Syntax

Figure 3.9 provides a concept model explaining the main concepts used when specifying a Basic Work Supporter.

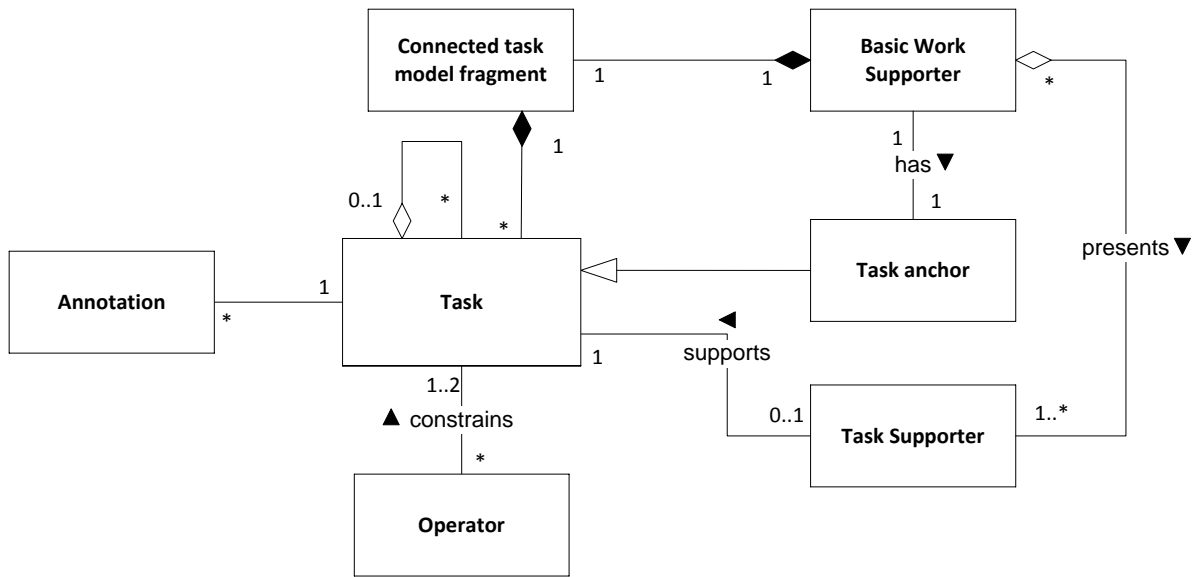


Figure 3.9 - Concept model describing the means for specifying Basic Work Supporters in FLUIDE-A

A Basic Work Supporter is mainly a definition of the connected task model fragment. Tasks may have connected annotations, giving for example headings and colours. As given by the relation from Basic Work Supporter and Task Supporter in Figure 3.9, as well as in the EBNF definition below, at least one of the tasks in the task model of a Basic Work Supporter instance must include a reference to a Task Supporter.

```
basic_work_supporter =
    bws(basic_work_supporter_identifier, connected_task_model_fragment, anchor_task,
        {task_supporter}-);
```

```
connected_task_model_fragment = ctmf({task}-, {operator});
```

```
anchor_task = task_identifier;
```

```
task = ta(task_identifier, {child_task}, {annotation});
```

```
child_task = task_identifier;
```

```
annotation = ann(name, expression);
```

```
operator = op(operator_type, from_operand_task, [to_operand_task]);
```

```
operator_type = [] | [> | >> | |> | *];
```

```
from_operand_task = task_identifier;
```

```
to_operand_task = task_identifier;
```

3.5.3 Semantics

```
[[ basic_work_supporter ]] =
```

[[**bws**(*basic_work_supporter_identifier*, *connected_task_model_fragment*, *anchor_task*,
{*task_supporter*}-)]]

[[**bws**(*basic_work_supporter_identifier*, *connected_task_model_fragment*, *anchor_task*,
{*task_supporter*}-)]] =

basic_work_supporter_identifier is a part of a user interface supporting the following user tasks through a set of Task Supporters (which present the parts of the user interface supporting each task):

{

The user task *task_identifier* is supported the Task Supporter *task_supporter_identifier*

}

The tasks presented by *basic_work_supporter_identifier* have a hierarchical structure. The task *anchor_task* is the root of this hierarchy. [[*connected_task_model_fragment*]].

[[*connected_task_model_fragment*]] =

[[**ctmf**(*{task}*-, *{operator}*)]]

[[**ctmf**(*{task}*-, *{operator}*)]] =

/* These brackets are for describing the hierarchical structure among the tasks */

[[*task*]]

}

Furthermore,

/* These brackets are for describing operators */

[[*operator*]]

}

[[*task*]] =

[[**ta**(*task_identifier*, {*child_task*}, {*annotation*})]]

/* There are two production rules for task, depending on whether the task has any children or not */

[[**ta**(*task_identifier*, {*child_task*}-, {*annotation*})]] =

task_identifier has the child tasks {*child_task*}, and its presentation is guided by the information that
{ [[*annotation*]]

$\llbracket \mathbf{ta}(task_identifier, \emptyset, \{annotation\}) \rrbracket =$

The presentation of $task_identifier$ is guided by the information that $\{\llbracket annotation \rrbracket\}$

$\llbracket annotation \rrbracket =$

$\llbracket \mathbf{ann}(name, expression) \rrbracket$

$\llbracket \mathbf{ann}(name, expression) \rrbracket =$

$name$ is $expression$

$\llbracket operator \rrbracket =$

$\llbracket \mathbf{op}(operator_type, from_operand_task, [to_operand_task]) \rrbracket$

/ There is one production rules for each operator type */*

$\llbracket \mathbf{op}([], from_operand_task, to_operand_task) \rrbracket =$

either $from_operand_task$ or $to_operand_task$ is performed

$\llbracket \mathbf{op}(>, from_operand_task, to_operand_task) \rrbracket =$

$to_operand_task$ deactivates $from_operand_task$

$\llbracket \mathbf{op}(>>, from_operand_task, to_operand_task) \rrbracket =$

$from_operand_task$ is performed before $to_operand_task$ is performed

$\llbracket \mathbf{op}(|>, from_operand_task, to_operand_task) \rrbracket =$

$to_operand_task$ may interrupt and deactivates $from_operand_task$. $from_operand_task$ is reactivated once $to_operand_task$ is completed

$\llbracket \mathbf{op}(*, from_operand_task) \rrbracket =$

$from_operand_task$ is performed a number of times

3.5.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Basic Work Supporters in FLUIDE-A. The example is a subset of the specification of the Basic Work Supporter in Figure 3.8.

3.5.4.1 EBNF Specification

To enable sufficiently rich semantic description, a copy of the whole specification of the member Task Supporters are included in the specifications of the Basic Work Supporter.

```

bws(Perform Work in the Field Supporter,
  ctmf(
    ta(Perform Work in the Field,
      /* child tasks: */
      Use Map, Manage Tasks,
      /* no annotations for the task */
    )
    ta(Use Map,
      /* no child tasks */,
      /* no annotations for the task */
    )
    ta(Manage Tasks,
      /* child tasks: */
      Receive Task Request, View Tasks, Accept Task, Decline Task, Perform Task,
      /* no annotations for the task */
    )
    ta(Receive Task Request,
      /* no child tasks */,
      /* no annotations for the task */
    )
    ta(View Tasks,
      /* no child tasks */,
      /* no annotations for the task */
    )
    ta(Accept Task,
      /* no child tasks */,
      /* no annotations for the task */
    )
    ta(Decline Task,
      /* no child tasks */,
      /* no annotations for the task */
    )
    /* specification of Perform Task omitted */
    /* operators: */
    op(>>, Receive Task Request, View Tasks),
    op(>>, View Tasks, Accept Task),
    op([], Accept Task, Decline Task),
    /* specification of one operator omitted */
  ), /* end of ctmf specification */
  Perform Work in the Field, /* anchor */
  /* Task Supporters: */
  ts(User Map, Use Map, Scene of Incident Presenter, Local Bases Presenter, Zones Presenter,
  Resources Presenter, Task for Resources Presenter),
  ts(Receive Task Request, Receive Task Request, Task for Resources Presenter),
  ts(View Tasks, View Tasks, Task for Resources Presenter, Task for Resources Presenter, Task for
  Resources Presenter),

```


ts(Decline Task, Decline Task, Task for Resources Presenter)
) /* end of bws specification */

3.5.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 3.5.3 on the EBNF specification just presented results in the following English sentences:

Perform Work in the Field Supporter is a part of a user interface supporting the following user tasks through a set of Task Supporters (which present the parts of the user interface supporting each task):

*The user task User Map is supported the Task Supporter **User Map**.*

*The user task Receive Task Request is supported the Task Supporter **Receive Task Request**.*

*The user task View Tasks is supported the Task Supporter **View Tasks**.*

*The user task Decline Task is supported the Task Supporter **Decline Task**.*

*The tasks presented by **Perform Work in the Field Supporter** have a hierarchical structure. The task Perform Work in the Field is the root of this hierarchy. Perform Work in the Field has the child tasks Use Map and Manage Tasks. Manage Tasks has the child tasks Receive Task Request, View Tasks, Accept Task, Decline Task, and Perform Task.*

Furthermore, Receive Task Request is performed before View Tasks is performed, View Tasks is performed before Accept Task is performed, and either Accept Task or Decline Task is performed.

3.6 Aggregated Work Supporter

In this section, we provide the syntax and semantics of the aggregated variant of the Work Supporter construct, i.e. Work Supporters that have other Work Supporters as children, also allowing operators between the child Work Supporters.

3.6.1 Graphical Syntax

Aggregated Work Supporters aggregate other Work Supporters (Basic or Aggregated). An Aggregated Work Supporter must add exactly one task (possibly with a Task Supporter) on the level above the member supporters. The task model of an Aggregated Work Supporter is specified indirectly, and thus expressed implicitly. The aggregated supporter inherits the task models of all its children. In the implicit task model, the root tasks of all the child supporters become direct children of the root task in aggregated supporter. In the graphical notation, the root task (and its Task Supporter if it has one) as well as its child supporters are located in the content part of an Aggregated Work Supporter. Figure 3.10 gives an example of an Aggregated Work Supporter, with explanations of certain parts.

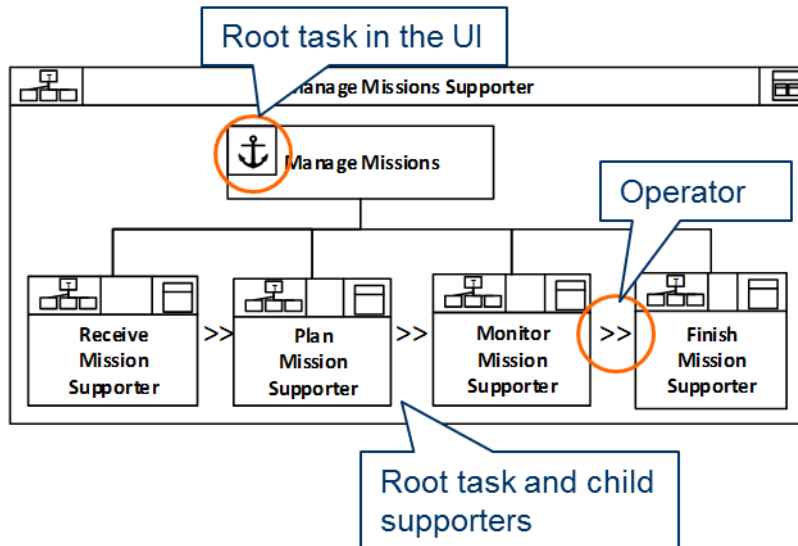


Figure 3.10 - An Aggregated Work Supporter in FLUIDE-A

Only the border part of the child supporters is shown in the aggregated one. The names of the child supporters are shown in their content part. The operators specified in the aggregated supporter operate between the root tasks of the involved child supporters in the implicit task model.

3.6.2 Abstract Syntax

Figure 3.11 provides a concept model explaining the main concepts used when specifying an Aggregated Work Supporter.

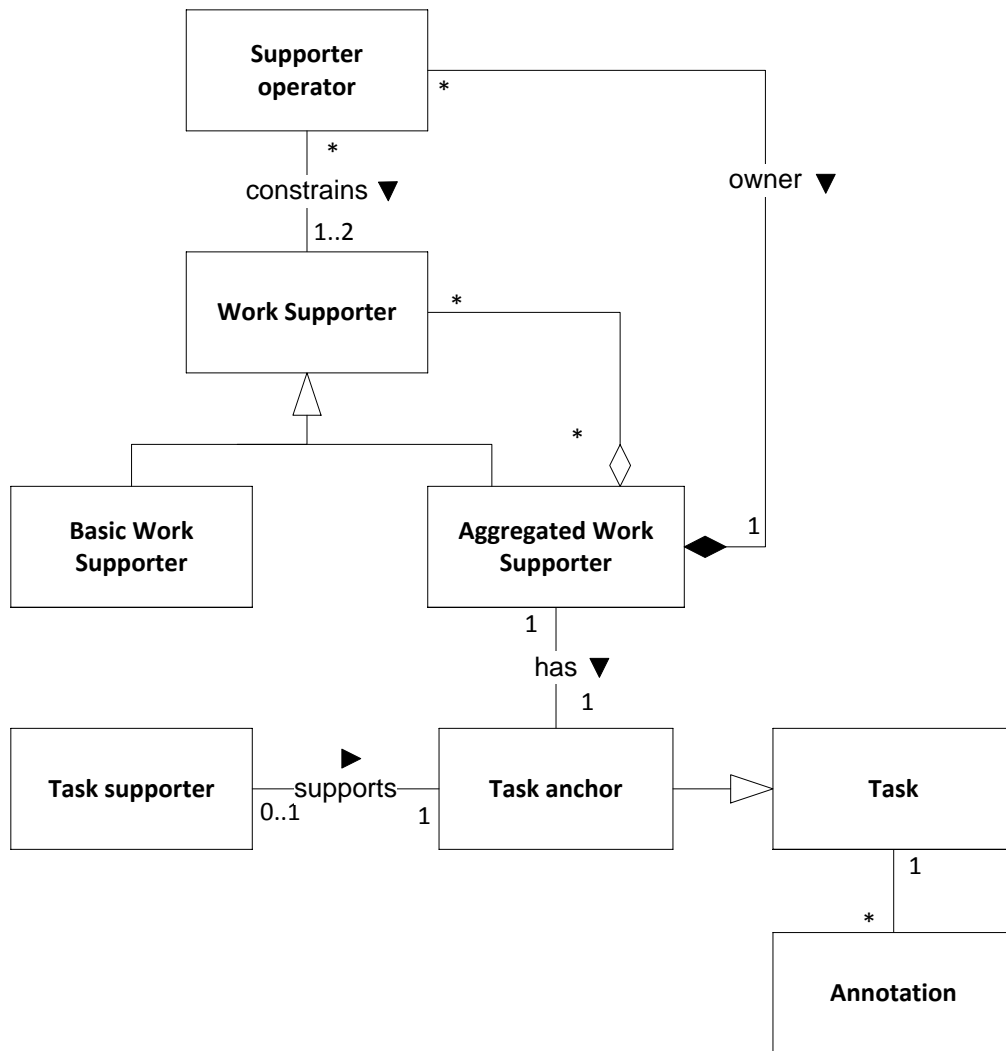


Figure 3.11 - Concept model describing the means for specifying Aggregated Work Supporters in FLUIDE-A

An Aggregated Work Supporter contains references to a number of Work Supporter that are either basic or aggregated. It also adds a task anchor that may have a Task Supporter and a set of annotations. The Aggregated Work Supporter may also have a number of Supporter Operators giving temporal restrictions on the performance of one or more of the tasks of the member supporters.

```

aggregated_work_supporter =
  aws(aggregated_work_supporter_identifier, task_anchor,
    {(work_supporter, presenter_anchor)}-, {supporter_operator});

task_anchor = tanc(anchor_task, [anchor_task_supporter], {annotation});

anchor_task = task_identifier;

anchor_task_supporter = task_supporter_identifier;

annotation = ann(name, expression);
  
```

work_supporter = *aggregated_work_supporter* | *basic_work_supporter*;

presenter_anchor = *task_identifier*;

supporter_operator =
supop(*operator_type*,
 (*from_operand_work_supporter*, *from_operand_presenter_anchor*),
 [(*to_operand_work_supporter*, *to_operand_presenter_anchor*)]);

operator_type = [] | [*>* | *>>* | *>**];

from_operand_work_supporter = *work_supporter_identifie*r;

from_operand_presenter_anchor = *task_identifie*r;

to_operand_work_supporter = *work_supporter_identifie*r;

to_operand_presenter_anchor = *task_identifie*r;

3.6.3 Semantics

[[*aggregated_work_supporter*]] =

[[**aws**(*aggregated_work_supporter_identifie*r, *task_anchor*,
 {(*work_supporter*, *presenter_anchor*)}-, {*supporter_operator*})]]

[[**aws**(*aggregated_work_supporter_identifie*r, *task_anchor*,
 {(*work_supporter*, *presenter_anchor*)}-, {*supporter_operator*})]] =

*aggregated_work_supporter_identifie*r is a part of a user interface supporting the following user tasks through a set of Task Supporters (which present the parts of the user interface supporting each task):

/* If there is a presenter for the task anchor, there must be a separate sentence for this */

[The user task *anchor_task* is supported through the Task Supporter *anchor_task_supporter*]

{ /* These brackets represent the set of child presenters (recursively for the aggregated ones) */

{ /* These brackets represent the set of Task Supporters for each of the child presenter */

The user task *task_identifie*r is supported through the Task Supporter
*task_supporter_identifie*r

}

}

The tasks presented by *aggregated_work_supporter_identifier* have a hierarchical structure. The task *anchor_task* is the root of this hierarchy.

/ First a description of the task anchor */*

`[[task_anchor]]`

/ The anchors of each aggregated presenters are children of the anchor task */*

anchor_task has the child tasks `{presenter_anchor}`

/ Each of these child tasks (and their children) are described */*

`{ /* These brackets represent the set of child presenters (recursively for the aggregated ones) */`

`{ /* These brackets represent the set of tasks within the connected_task_model_fragment of each child presenter */`

`[[task]]`

`}`

`}`

Furthermore,

`{ [[presenter_operator]]` */* if no presenter_operators are given, this will produce nothing */*

`{ /* These brackets represent the set of child presenters (recursively for the aggregated ones) */`

`{ /* These brackets represent the set of tasks within the connected_task_model_fragment of each child presenter */`

`[[operator]]`

`}`

`}`

`[[task_anchor]]=`

`[[tanc(anchor_task, [anchor_task_supporter], {annotation})]]`

`[[tanc(anchor_task, [anchor_task_supporter], {annotation})]]=`

The presentation of *anchor_task* is guided by the information that `{ [[annotation]]`.

`[[annotation]]=`

[[**ann**(*name*, *expression*)]]

[[**ann**(*name*, *expression*)]] =

name is *expression*

[[*supporter_operator*]] =

[[**supop**(*operator_type*,
(*from_operand_work_supporter*, *from_operand_presenter_anchor*),
(*to_operand_work_supporter*, *to_operand_presenter_anchor*))]]

/* There is one production rules for each operator type */

[[**supop** ([],(*from_operand_work_supporter*, *from_operand_presenter_anchor*),
(*to_operand_work_supporter*, *to_operand_presenter_anchor*))]]=

either *from_operand_presenter_anchor* or *to_operand_presenter_anchor* is performed

[[**supop** ([>,(*from_operand_work_supporter*, *from_operand_presenter_anchor*),
(*to_operand_work_supporter*, *to_operand_presenter_anchor*))]]=

to_operand_presenter_anchor deactivates *from_operand_presenter_anchor*

[[**supop** (>>,(*from_operand_work_supporter*, *from_operand_presenter_anchor*),
(*to_operand_work_supporter*, *to_operand_presenter_anchor*))]]=

from_operand_presenter_anchor is performed before *to_operand_presenter_anchor* is performed

[[**supop** (|>,(*from_operand_work_supporter*, *from_operand_presenter_anchor*),
(*to_operand_work_supporter*, *to_operand_presenter_anchor*))]]=

to_operand_presenter_anchor may interrupt and deactivate *from_operand_presenter_anchor*.
from_operand_presenter_anchor is reactivated once *to_presenter_anchor* is completed

[[**supop** (*,(*from_operand_work_supporter*, *from_operand_presenter_anchor*))]]=

from_operand_presenter_anchor is performed a number of times

3.6.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Aggregated Work Supporters in FLUIDE-A. The example is a subset of the specification of the Aggregated Work Supporter in Figure 3.10.

3.6.4.1 EBNF Specification

To enable sufficiently rich semantic description, a copy of the whole specification of the member Work and Task Supporters are included in the specifications of the Aggregated Work Supporter.

```

aws(Manage Missions Supporter,
  anc(Manage Missions,
    /* no Task Supporter for the anchor task */
    /* no annotations for the anchor task */
  ),
  /* child Work Supporters: */
  (
    bws(Receive Mission Supporter,
      ctmf(
        ta(Receive Mission,
          /* no child tasks: */
          /* no annotations for the task */
        ),
        /* no operators */
      ), /* end of ctmf specification */
      Receive Mission, /* anchor */
      /* Task Supporters: */
      ts(Receive Mission, Receive Mission, Message Reader),
    ), /* end of bcp specification */
    Receive Mission /* presenter anchor */
  ),
  /* specification of Plan Mission Supporter and Monitor Mission Supporter omitted */
  (bws(Finish Mission Supporter,
    ctmf(
      ta(Finish Mission,
        /* no child tasks: */
        /* no annotations for the task */
      ),
      /* no operators */
    ), /* end of ctmf specification */
    Receive Mission, /* anchor */
    /* Task Supporters: */
    ts(Assess Mission, Assess Mission, Mission Presenter, Mission Presenter),
  ), /* end of bcp specification */
  Finish Mission /* presenter anchor */
),
/* supporter operators: */
op(>>,(Receive Mission Supporter, Receive Mission), (Plan Mission Supporter, Plan Mission)),
op(>>,(Plan Mission Supporter, Plan Mission),
  (Monitor Mission Supporter, Monitor Mission)),
op(>>,(Monitor Mission Supporter, Monitor Mission),
  (Finish Mission Supporter, Finish Mission))
) /* end of aws specification */

```

3.6.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 3.6.3 on the EBNF specification just presented results in the following English sentences (contents from the parts of the EBNF specification that are omitted is indicated in [brackets]):

***Manage Missions Supporter** is a part of a user interface supporting the following user tasks through a set of Task Supporters (which present the parts of the user interface supporting each task):*

*The user task Receive Mission is supported the Task Supporter **Receive Mission**.*

[eleven more Task Supporters from the two omitted child Work Supporters]

*The user task Assess Mission is supported the Task Supporter **Assess Mission**.*

*The tasks presented by **Manage Missions Supporter** have a hierarchical structure. The task Manage Missions is the root of this hierarchy. Manage Missions has the child tasks Receive Mission, [two more tasks] and Finish Mission. [the rest of the task hierarchy].*

Furthermore, Receive Mission is performed before Plan Mission is performed. Plan Mission is performed before Monitor Mission is performed. Monitor Mission is performed before Finish Mission is performed. [three more operators from the task models in the child Work Supporters].

3.7 Category Manager

In this section, we provide the syntax and semantics of the Category Manager construct. Category Managers are only provided in a basic variant.

3.7.1 Graphical Syntax

The Category Manager construct is used to group user interfaces supporting a category of functionality, like *common operational picture*, *triage* and *resource management*. Such a category may coincide with an application, but one application may support more than one category of functionality. A Category Manager primarily aggregates a number of Work Supporters (Basic or Aggregated). It may also aggregate Content Presenters (Basic or Aggregated). A Category Manager does not have an anchor, and it is not possible to specify any relations between its child supporters/presenters. A Category Manager aggregating only Content Presenters is semantically equal to a Task Supporter aggregating the same Content Presenters. In the graphical notation, the child supporters/presenters are located in the content part of a Category Manager. Figure 3.12 gives an example of a Category Manager, with explanations of certain parts.

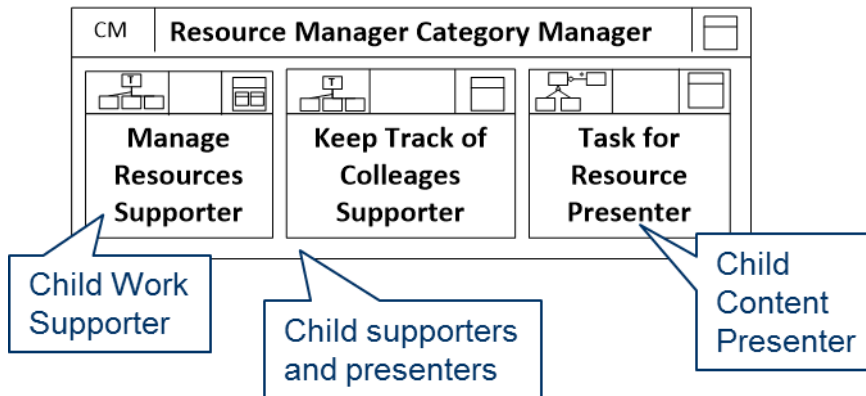


Figure 3.12 - A Category Manager in FLUIDE-A

Only the border part of the child supporters/presenters is shown in the Category Manager. The names of the child supporters/presenters are shown in their content part. In the example in Figure 3.12, there are three children, a Basic Work Supporter, an Aggregated Work Supporter, as well as a Basic Content Presenter.

3.7.2 Abstract Syntax

Figure 3.13 provides a concept model explaining the main concepts used when specifying a Category Manager.

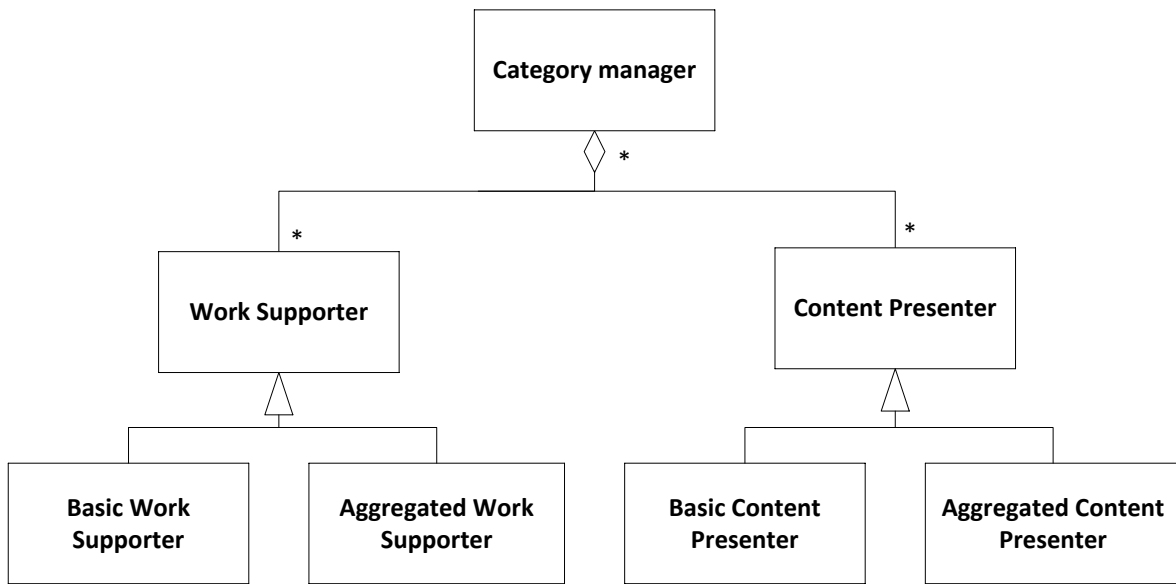


Figure 3.13 - Concept model describing the means for specifying Category Managers in FLUIDE-A

In the EBNF below, there is neither a requirement to have Work Supporter nor Content Presenter members of a Category Manager, but to have neither is not meaningful. To have both types of members is allowed.

category_manager =
cm(*category_manager_identifier*, {*work_supporter_identifier*}, {*content_presenter_identifier*});

work_supporter_identifier =
aggregated_work_supporter_identifier | *basic_work_supporter_identifier*;

content_presenter_identifier =
aggregated_content_presenter_identifier | *basic_content_presenter_identifier*;

3.7.3 Semantics

[[*category_manager*]] =

[[**cm**(*category_manager_identifier*, {*work_supporter_identifier*}, {*content_presenter_identifier*})]]

[[**cm**(*category_manager_identifier*, {*work_supporter_identifier*}, {*content_presenter_identifier*})]] =

category_manager_identifier is a part of a user interface supporting a category of functionality.
category_manager_identifier contains the user interface parts {*work_supporter_identifier*} as well as {*content_presenter_identifier*}. These user interface parts have no specific connections.

With these production rules, only the identifiers (the names) of the Work Supporters and Content Presenters that are member of the Category Manager are included in the resulting sentences. To investigate the semantics of the corresponding supporters and presenters, the production rules for the supporters and presenters must be used. The reason for this solution is that a Category Manager (in contrast to an Aggregated Work Supporter and Aggregated Content Presenter) does not change the semantics of the member supporters and presenters.

3.7.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Category Managers in FLUIDE-A. The example is the specification of the Category Manager in Figure 3.12.

3.7.4.1 EBNF Specification

cm(Resource Manager Category Manager, Manage Resources Supporter, Keep Track of Colleagues Supporter, Task for Resource Presenter)

3.7.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 3.7.3 on the EBNF specification just presented results in the following English sentences:

*Resource Manager Category Manager is a part of a user interface supporting a category of functionality. Resource Manager Category Manager contains the user interface parts **Manage Resources Supporter** and **Keep Track of Colleagues Supporter**, as well as **Task for Resource Presenter**. These user interface parts have no specific connections.*

4 The FLUIDE-D Language

In this section, we present the syntax and semantics of the FLUIDE-D language. The syntax is presented both as a concrete syntax in the form of a graphical syntax, and as an abstract syntax expressed in Extended Backus-Naur Form (EBNF). The semantics is presented as a natural language semantics giving production rules which may be used to translate an expression in EBNF to one or more English sentences. For each interactor construct in FLUIDE-D, we provide one example specification, each being a design for the corresponding interactor instance provided in the examples in Section 3. The examples expressed using the graphical syntax are provided as part of the sections explaining the graphical syntax. The examples expressed in EBNF and the corresponding English sentences describing the semantics of the example are presented together directly after the definition of the semantics for the interactor design construct at hand. We use the same typographical and other conventions we use in the description of FLUIDE-A, as described in the beginning of Section 3.

We start by giving the syntax and semantics for the common parts. This includes the interactor design construct, as well as the different view types. These sections contain the common graphical syntax used in FLUIDE-D. Even though this part contains definitions of abstract syntax and semantics, we do not provide examples for the common parts. Instead, examples expressed in EBNF and the corresponding English sentences for these parts are embedded in the examples provided for the individual interactor design constructs.

After the common parts are presented, the syntax and semantics of the four interactor constructs in FLUIDE-D (Content Presenter Design, Task Supporter Design, Work Supporter Design and Category Manager Design) are given in separate sections for each of the constructs. The Content Presenter Designs and the Work Supporter Designs are presented in separate sections for the basic and aggregated variants. The Task Supporter Designs and Category Manager Designs are only available in one variant. When we present the constructs, we start with Basic Content Presenter Designs, and move up in the aggregation hierarchy finishing with Category Manager Designs.

4.1 Interactor Design

In this section, we provide a definition of the common graphical syntax for FLUIDE-D. We also provide an EBNF definition of the abstract syntax for interactor designs. As there are no terminal symbols in the abstract syntax, the semantics for all parts of the EBNF definition of the interactor design construct is given by the general rules for the semantics given in the introduction part of Section 3 above.

4.1.1 Graphical Syntax

In the graphical notation, all the interactor constructs in FLUIDE-D use the basic layout in shown in Figure 4.1.

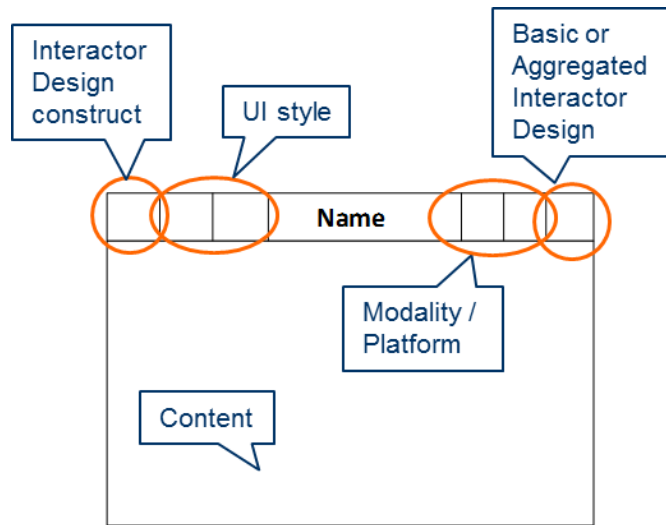


Figure 4.1 - Basic layout of the interactor design constructs in FLUIDE-D

An interactor design instance in FLUIDE-D looks similar to an interactor instance in FLUIDE-D, and is represented as a rectangle with a top border resembling a window. The top border contains the name of the interactor design instance, as well as an icon on the top left denoting interactor design construct used and an icon on the top right denoting whether the instance is basic or aggregated. Instances of all constructs may be basic, while only Content Presenter Designs and Work Supporter Designs may be aggregated. The content part (canvas) underneath the top border is used for presenting the content of the interactor design instance. The content is different for instances of the different constructs. Compared to interactor instances in FLUIDE-D, the interactor design instances have some additional icons in the top border. On the top left (to the right of the construct icon), there should be one or more icons specifying the user interface style(s) used in the design. On the right (to the left of the basic/aggregated icon), there should be one or more icons specifying the modality/platform combination(s) used in the design.

Table 4.1 shows the icons used for the four interactor design constructs. As can be seen, these icons resemble the icons used in FLUIDE-A, but the FLUIDE-D versions have a "window" heading to indicate that the specifications are less abstract. The icons used to denote whether an interactor design instance is basic or aggregated are the same as in FLUIDE-A (see Table 3.2 above). Table 4.2 shows the icons used for the available user interface styles, while Table 4.3 shows the icons used for the available modalities/platforms.

Table 4.1 – Icons used for the four interactor design constructs





Interactor design construct	Icon
Content Presenter Design	
Task Supporter Design	
Work Supporter Design	
Category Manager Design	

Table 4.2 – Icons used for user interface styles












User interface style	Icon
Forms based	
List based	
Icons based	
Map based	
Graph based	
Multimedia based	

Table 4.3 – Icons used for user platforms/modalities

Modality / Platform	Icon
PC / laptop with mouse and keyboard	
Mobile device with touch	
Table top	
Augmented reality	
Audio interaction	

4.1.2 Abstract Syntax

interactor_design =

content_presenter_design | *task_supporter_design* | *work_supporter_design* | *category_manager_design*;

content_presenter_design =

basic_content_presenter_design | *aggregated_content_presenter_design*;

work_supporter_design = *basic_work_supporter_design* | *aggregated_work_supporter_design*;

4.2 Views

The interactor design instances in FLUIDE-D have similar content as the interactor instances in FLUIDE-A, except that the content in the interactor design instances is wrapped in views. Below, we present the five different view types available in FLUIDE-D including their available layout mechanisms.

In this section, we present the common graphical syntax for views FLUIDE-D. We also provide an EBNF definition of the abstract syntax for the different view types, as well as production rules defining a translation from EBNF expressions to English sentences.

As mentioned above, we do not provide examples of using the abstract syntax and production rules for the semantics. Examples of views expressed in EBNF and the corresponding English sentences are embedded in the examples provided for the interactor design constructs.

4.2.1 Graphical Syntax

In this section, we present the graphical syntax for each of the five view types in FLUIDE-D, as well as the syntax used for specifying dialog navigation. The corresponding abstract syntax defined in EBNF is given for the five view types and the model patterns in Section 4.2.2 below, while semantics for the same is defined in Section 4.2.3.

4.2.1.1 Decorational View

A Decorational View is used to specify different containers with a visual appearance. There are three types of such views, i.e. a Border, Window and Loosely connected windows. The former represents a visual border within a user interface, while the two others represent the outermost visual part of one or more dialogs. In the graphical notation, these views resemble the concrete user interface element they represent. A Border type view is represented as a solid rectangle with its optional heading shown on the outline of the rectangle. A Window type view is represented as an abstract window, with its optional heading shown in the top border, and a close button on the top right. A stacking look is used for loosely connected windows/dialogs. Figure 4.2 shows the graphical notation for the three types of Decorational Views.

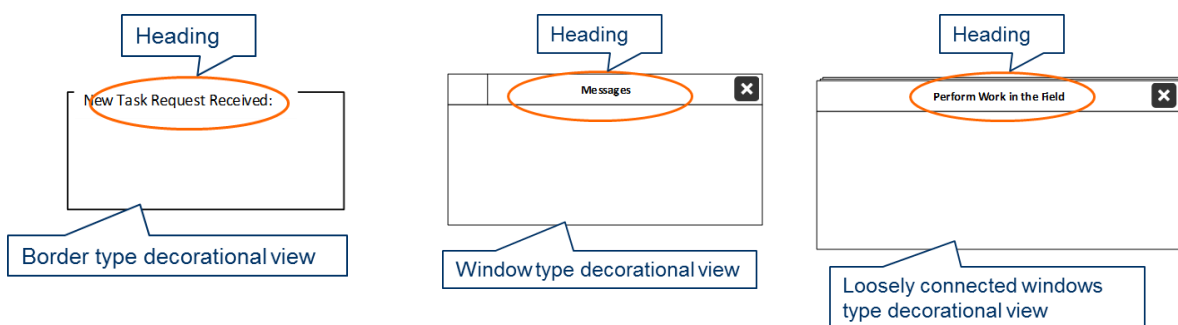


Figure 4.2 - Decorational Views in FLUIDE-D

The different view types offer a set of options for how the layout of their content is organized. Decorational Views offer four layout methods: percentage, relative, managed and automatic. When *percentage layout* is used, the location and size of the view's children are given as percentage values of the parent view. When *relative layout* is used, the location and size of the view's children are given through rules for how they should be placed relative to their parent or a sibling, using expressions like "parent top left filled" and "sibling right". In the graphical notation, *relative layout* is specified using fat solid arrows pointing from the view being laid out to the view it connects to (see Figure 4.7 for an illustration). When *managed layout* is used, the layout is determined by one or more child views of the type Layout Manager View. When *automatic layout* is used, the location and

size of the view's children are determined by a layout algorithm provided by the view. In the graphical notation, the layout method is indicated by an icon on the top right of the view. Table 4.4 shows the icons used for the different layout methods in the graphical notation.

Table 4.4 – Icons used for layout methods

Layout method	Icon
Percentage	<div style="display: flex; flex-direction: column; align-items: center;"> <div style="display: flex; gap: 5px;">□ □</div> <div style="display: flex; gap: 5px;">□ %</div> </div>
Relative	<div style="display: flex; flex-direction: column; align-items: center;"> <div style="display: flex; gap: 5px;">□ □</div> <div style="display: flex; gap: 5px;">□ R</div> </div>
Managed	<div style="display: flex; flex-direction: column; align-items: center;"> <div style="display: flex; gap: 5px;">□ □</div> <div style="display: flex; gap: 5px;">□ M</div> </div>
Automatic	<div style="display: flex; flex-direction: column; align-items: center;"> <div style="display: flex; gap: 5px;">□ □</div> <div style="display: flex; gap: 5px;">□ A</div> </div>

If the sibling rules used in a view with *relative layout* do not refer to which sibling it should be placed relative to, as well as for *managed* and *automatic layout*, the sequence in which sibling elements are placed in the interactor design is important for the layout. The sequence may also be influenced by annotations from the corresponding FLUIDE-A specification.

4.2.1.2 Layout Manager View

Layout Manager Views are not given names, and are shown using dashed lines (to indicate that they are usually not visible). The arrows on the dashed line specify whether the children are organized horizontally or vertically. Figure 4.3 shows the graphical notation for Layout Manager Views (a vertical one with a horizontal child).

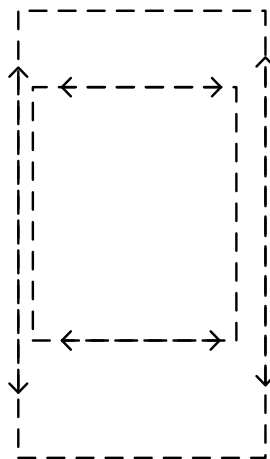


Figure 4.3 - Layout Manager View in FLUIDE-D

As Layout Manager Views always use a *managed layout* method there are no icon for layout method for such views.

4.2.1.3 Content View

A Content View presents instances of one or more entities, and may only be used as part of Basic Content Presenters. In the graphical notation, Content Views are represented as a solid rectangle. On the top of the

rectangle, UML stereotype notation is used to denote the view type before its name. A 1 or * on the top left denotes whether the view presents one or a number of instances of the anchor entity. Content views usually use *automatic layout*, but a layout method icon is still used. Figure 4.4 shows the graphical notation for Content Views.

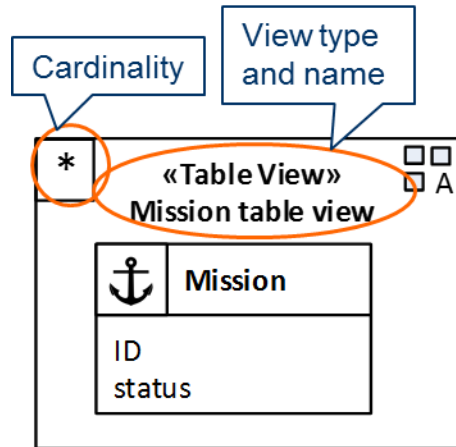


Figure 4.4 - Content View in FLUIDE-D

All Content Views impose restrictions on the concept model fragment they may present, expressed in FLUIDE-D as a model pattern fitting the user interface pattern supported by the Content View type. The model pattern for the generic Content View used in Figure 4.4 (Table View) must contain one entity (possibly with subtypes) that determines the rows in the table (*Mission*). It may also include related entities, as long as the cardinality on the side of the related entity is one. By requiring that the concept model fragment to be presented in a view follow a certain model pattern, transforming the view to a final user interface is a manageable and predictable process that may be automated. Because the structure of the concept model fragment is known, it is possible to define an algorithm for doing the transformation that is independent of the actual entities, attributes and relations used in the concept model fragment.

Content Views may be divided in two groups: generic and domain-specific. Generic Content Views are views that are not tailored for the emergency response domain. Example of such views in FLUIDE-D are:

- Single Instance View – for viewing one instance of one entity type in a forms-based user interface
- Owner + Members View – for viewing details of one instance of one entity type in a forms-based user interface and a list of a related entity type in a table
- List + Details View – for viewing multiple instances of one entity type in a list, and details about one of these instances in a forms-based user interface
- List View – for viewing multiple instances of one attribute of one entity type as a list
- Table View – for viewing multiple instances of one entity type in a table able to show multiple attributes
- Icon List View – for viewing multiple instances of one entity type as a list of icons
- Icon Table View – for viewing multiple instances of one entity type as a grid of icons
- Sensor Feeds View – for viewing a sensor feed graphically
- Media-player View – for viewing images and videos

Domain-specific Content Views are views that are tailored for needs in the emergency response domain. Despite being tailored for such needs, these views may also be used when specifying user interfaces in related domains or domains with overlapping needs. Examples of such views in FLUIDE-D are:

- The views used for specifying the contents of the ribbon sub categories and the ribbon ticker and ribbon buttons (Nilsson and Stølen, 2016a). These views include Ribbon Sub Category Single Entity View, Ribbon Category Overview View, Ribbon Ticker Category View, and Ribbon Button View.
- The views used for specifying maps with icons and other graphics representing emergency response objects on a map (Nilsson and Stølen, 2016a; Nilsson and Stølen, 2016b). These views include Map Icons View, Map Outline View, and Map Icons with Details Dialog View.
- The Body Parts Visualization View used in the eTriage part of the (Nilsson and Stølen, 2016a) for visualizing affected body parts of a victim being triaged.

The distinction between generic and domain-specific views is neither reflected in the graphical nor the abstract syntax.

4.2.1.4 Content Integration View

Content Integration Views integrate related content from different interactor design instances. Content Integration Views require that their member designs use specific Content or Content Integration Views. Occurrences of all interactor design constructs may be children as long as they contain views having the required type. In the graphical notation, Content Integration Views are represented as a solid rectangle. On the top of the rectangle, UML stereotype notation is used to denote the view type before its name. Content Integration Views do not have any cardinality. Such views usually use *automatic layout*, but a layout method icon is still used. Figure 4.5 shows the graphical notation for Content Views.

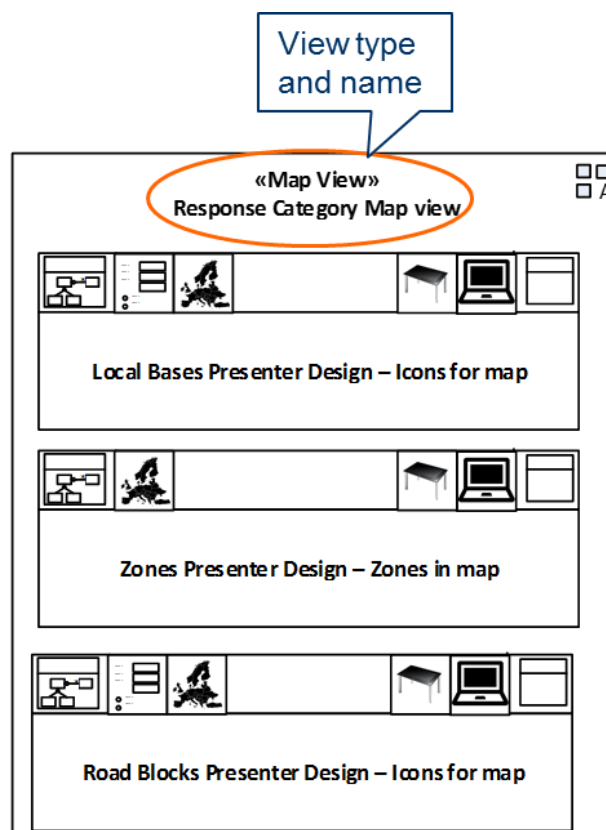


Figure 4.5 - Content Integration View in FLUIDE-D

The connection between Content Views and Content Integration Views may be illustrated through the views for map-based visualization. Map View is a Content Integration View that among other may have Map Icons Views and Map Outline Views, as well as other Map Views as members. If a presenter design using Map Icons Views or Map Outline Views is member of a Map View, either directly or one or more times among its parents, the icons and outlines are shown on the map provided by the Map View highest up in the hierarchy. If a Map Icons (or Outline) View does not have a parent providing a Map View, it will provide its own map. All Content Integration Views in FLUIDE-D are domain-specific. Examples of such views in FLUIDE-D are:

- The views used for specifying a ribbon category, a set of ribbon buttons and the ribbon ticker (Nilsson and Stølen, 2016a). These views include Ribbon Category View, Ribbon Contents View, Ribbon Ticker View, and Ribbon View.
- The Map View discussed above.
- A view combining ribbons and maps.

The available Content and Content Integration Views make up the FLUIDE library of emergency response user interface patterns.

4.2.1.5 Interactor Design View

As occurrences of interactors (except Basic Content Presenter Designs) may include references to other interactor occurrences, interactor design occurrences may contain references to other interactor design occurrences. As the content of interactor design occurrences are views, member interactor design occurrences are formally defined as views in FLUIDE-D. We denote such views *Interactor Design Views*. The graphical syntax is similar to references to interactor instances in FLUIDE-A specifications. The view shown in Figure 4.5 contains three Interactor Design Views, more precisely three Basic Content Presenter Design Views.

In the abstract syntax of the different interactor design constructs, references to Interactor Design Views are usually referred to by the name of the design construct with the term *view* added. For example, in an Aggregated Content Presenter Design, child designs are referred to as (or rather wrapped in) *Content Presenter Design Views*.

4.2.1.6 Dialog navigation

A dialog navigation specification is shown as a dashed-lined arrow with a growing size. The type of dialog navigation (open, show, hide, close, or return) is shown as text on the arrow, as illustrated in Figure 4.6.

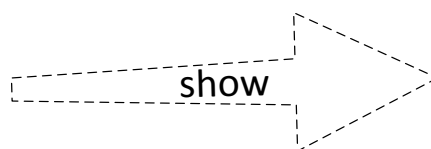


Figure 4.6 – Dialog navigation in FLUIDE-D

Such arrows point from a part of an interactor design specification to a reference to another interactor design specification (using the same syntax as for members of aggregated interactor designs) or another view in the interactor design. The small end indicates which element of the user interface that triggers the dialog navigation, i.e. the user interface representation of the view, button, entity or attribute that the small end starts from. The point of the arrow indicates which design or view that is the target for the dialog navigation. If the navigation type is *return*, there is no target, as this specifies navigation back to the dialog from which the dialog being specified was opened.

Figure 4.17 provides an example where dialog navigation is specified in the context of a view that is part of an Aggregated Work Supporter Design.

4.2.1.7 Example

To illustrate some of the view types and layout mechanisms we present the *Incident Presenter Design* shown in Figure 4.7.

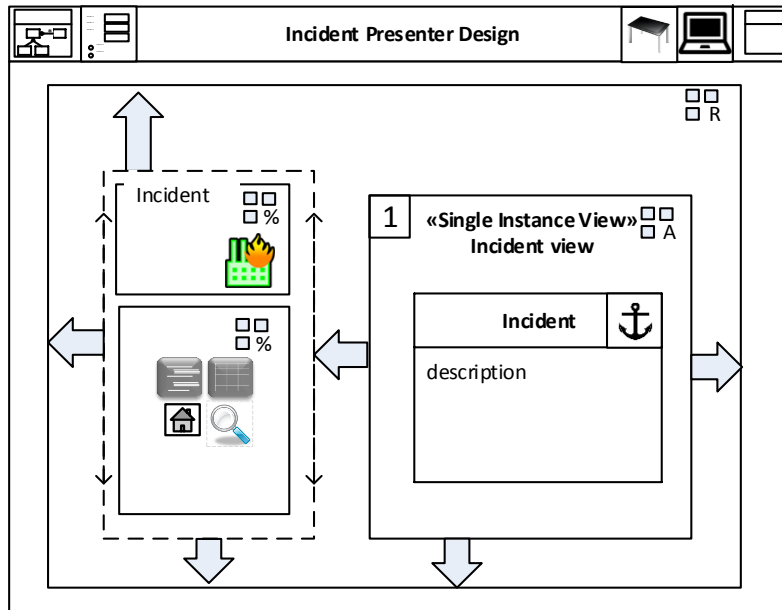


Figure 4.7 - Example showing different views and layout mechanisms in FLUIDE-D

The Basic Content Presenter in Figure 4.17 has one view as its top level child. This is a Border type Decorational View using a *relative layout* mechanism. Therefore, all the child views of this view have fat arrows specifying how their layout relate to their parent and sibling views. This will also ensure that the child views will resize properly if the outmost view is resized. The border view has two child views. To the left, there is a vertical Layout Manager View, which itself has two Border type Decorational Views as children. Both these views use *percentage layout* method. To the right, there is a generic Content View (a Single Instance View called *Incident view*) using an *automatic layout* method.

4.2.2 Abstract Syntax

In this section, we present the abstract syntax defined in EBNF for each of the five view types in FLUIDE-D, as well as the model patterns used in the Content Views and the syntax used for specifying dialog navigation. The semantics for the same is defined in Section 4.2.3.

Figure 4.8 provides a concept model giving an overview of the different view types and explaining the main concepts used when specifying views.

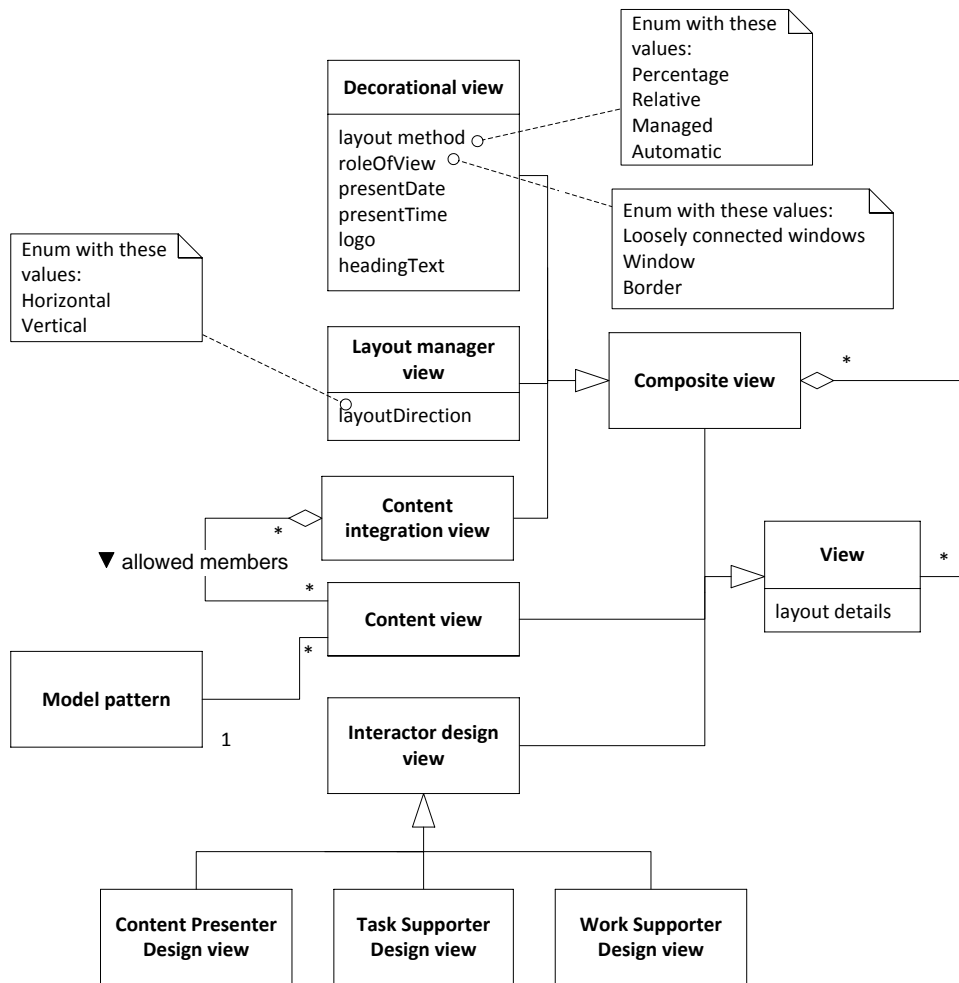


Figure 4.8 - Concept model describing the means for specifying views in FLUIDE-D

As shown in Figure 4.8, the view types may be divided into views that are composite and views that are not. Composite views may have other views (composite or not) as children. All view types except *Content Views* and *Interactor Design Views* are composite.

Before addressing the five view types (and the model patterns and dialog navigation), we provide definitions of some parts of the abstract syntax that are used by all or most of the view types as well as by the interactor design constructs.

view = *composite_view* | *content_view* | *interactor_design_view*;

composite_view = *decorational_view* | *layout_manager_view* | *content_integration_view*;

Layout details may be used by all view types in the role as child view to a parent having *percentage* or *relative* layout method.

layout_details = *coordinates* | *relative_layout_details*;

coordinates = **coord**(*x*, *y*, *width*, *height*);

x = *percentage*;

y = *percentage*;

width = *percentage*;

height = *percentage*;

The *x*, *y*, *width* and *height* values are expressed relative to the parent.

relative_layout_details = **top left parent** | **filled parent** | **top filled parent** | **sibling right** | **sibling left** | **sibling over** | **sibling under**;

Visual elements may be used by all view types, but attributes and annotations may only be used in the specification of Content Views. *Image* may be used for static logos as well as icons. *Text* may be used both for headings and labels.

visual_elements =
ves(*image*], [*text*] , [*border_colour*], {*graphics*}, {*button*});

image = *static_picture* | *picture_attribute*;

static_picture = *picture*;

picture_attribute = *attribute_identifier*;

text = *static_string* | *text_attribute*;

static_string = *string*;

text_attribute = *attribute_identifier*;

border_colour = *colour_spec*;

colour_spec = *static_colour* | **use annotation**;

static_colour = *colour*;

graphics = **gr**(*shape*, *colour_spec*);

shape = **circle** | **triange** | **square**;

button = **btn**(*button_identifier*, [*button_image*], [*button_text*]);

button_image = *picture*;

button_text = *string*;

4.2.2.1 Decorational View

decorational_view =
dv(*decorational_view_identifier*, *role_of_decorational_view*, [**present date**], [**present time**],
[*visual_elements*], {*dialog_navigation*}, *layout_method*, [*layout_details*], {*child_view*});

role_of_decorational_view = **loosely connected windows** | **window** | **border**;

layout_method = **percentage** | **relative** | **managed** | **automatic**;

child_view = *view*;

4.2.2.2 Layout Manager View

layout_manager_view =
lmv(*layout_manager_view_identifier*, *layout_direction*, [*visual_elements*], {*dialog_navigation*},
[*layout_details*], {*child_view*});

layout_direction = **horizontal** | **vertical**;

child_view = *view*;

If the visual elements contain a specification of a heading, this will be ignored.

4.2.2.3 Content View

content_view = *3D_icons_imposed_on_camera_view* | *body_parts_visualization_view* |
browser_for_composite_view | *browser_for_one_to_many_hierarchy_view* | *icon_list_view* | |
icon_table_view | *icon_table_subtypes_view* | *list_details_view* | *list_view* | *map_icons_view* |
map_icons_with_details_dialog_view | *map_multi_line_view* | *map_outline_view* | *media_player_view* |
multi_instance_view | *owner_member_view* | *ribbon_button_view* | *ribbon_category_overview_view* |
ribbon_sub_category_single_entity_view | *ribbon_sub_category_subtyped_single_entity_view* |
ribbon_sub_categories_categorized_single_entity_view |
ribbon_sub_categories_categorized_subtyped_single_entity_view | *ribbon_ticker_category_view* |
sensor_feeds_view | *single_instance_view* | *single_instance_with_proposed_standard_text_view* | *table_view*
| *tree_for_composite_view* | *tree_for_one_to_many_hierarchy_view*;

3D_icons_imposed_on_camera_view =
3Diiocv(*3D_icons_imposed_on_camera_view_identifier*, *main_entity_identifier*,
point_localized_extended_single_entity, [*visual_elements*], {*dialog_navigation*}, [*layout_details*]);

body_parts_visualization_view =
bpvv(*body_parts_visualization_view_identifier*, *single_entity*, [*visual_elements*],
{*dialog_navigation*}, [*layout_details*]);

browser_for_composite_view =
bfcv(*browser_for_composite_view_identifier*, *composite*, [*visual_elements*], {*dialog_navigation*},
[*layout_details*]);

```
browser_for_one_to_many_hierarchy_view =  
    bftmv(browser_for_one_to_many_hierarchy_view_identifier, one_to_many_hierarchy,  
        [visual_elements], {dialog_navigation}, [layout_details]);
```

```
icon_list_view =  
    ilv(icon_list_view_identifier, single_entity, [visual_elements], {dialog_navigation},  
        [layout_details]);
```

```
icon_table_view =  
    itv(icon_table_view_identifier, categorized_single_entity, [visual_elements], {dialog_navigation},  
        [layout_details]);
```

```
icon_table_subtypes_view =  
    itsv(icon_table_subtypes_view_identifier, subtyped_single_entity, [visual_elements],  
        {dialog_navigation}, [layout_details]);
```

```
list_details_view =  
    ldv(list_details_view_identifier, main_entity_identifier, entity_with_list_attributes,  
        entity_with_details_attributes, [visual_elements], {dialog_navigation}, [layout_details]);
```

entity_with_list_attributes and *entity_with_details_attributes* must have the same *main_entity*.

```
entity_with_list_attributes = extended_single_entity;
```

```
entity_with_details_attributes = extended_single_entity;
```

```
list_view =  
    lv(list_view_identifier, extended_single_entity, [visual_elements], {dialog_navigation},  
        [layout_details]);
```

```
map_icons_view =  
    micv(map_icons_view_identifier, main_entity_identifier, point_localized_extended_single_entity,  
        map_buttons, [visual_elements], {dialog_navigation}, [layout_details]);
```

```
map_buttons = mb([include mode button], [include center button], [include projector button]);
```

```
map_icons_with_details_dialog_view =  
    miwddv(map_icons_with_details_dialog_view_identifier, main_entity_identifier,  
        point_localized_extended_single_entity, map_buttons, [visual_elements], {dialog_navigation},  
        [layout_details]);
```



```
map_multi_line_view =  
    mmlv(map_multi_line_view_identifier, single_entity_with_point_collection, map_buttons,  
        [visual_elements], {dialog_navigation}, [layout_details]);
```

```
map_outline_view =  
    mov(map_outline_view_identifier, area_localized_categorized_single_entity, map_buttons,  
        [visual_elements], {dialog_navigation}, [layout_details]);
```

```
media_player_view =  
    mpv(media_player_view_identifier, {single_attribute_from_single_entity}, [visual_elements],  
        {dialog_navigation}, [layout_details]);
```

```
multi_instance_view =  
    miv(multi_instance_view_identifier, extended_single_entity, [visual_elements],  
        {dialog_navigation}, [layout_details]);
```

```
owner_member_view =  
    omv(owner_member_view_identifier, extended_owner_entity_with_member, [visual_elements],  
        {dialog_navigation}, [layout_details]);
```

This user interface pattern is sometimes called master/details.

```
ribbon_button_view =  
    rbv(ribbon_button_view_identifier, single_entity, [visual_elements], {dialog_navigation},  
        [layout_details]);
```

```
ribbon_category_overview_view =  
    rcov(ribbon_category_overview_view_identifier, single_entity, ribbon_category_buttons,  
        [visual_elements], {dialog_navigation}, [layout_details]);
```

```
ribbon_category_buttons = rcb([include tree button], [include home button], [include table button],  
    [include search button]);
```

```
ribbon_sub_category_single_entity_view =  
    rscsev(ribbon_sub_category_single_entity_view_identifier, single_entity, [include add button],  
        [visual_elements], {dialog_navigation}, [layout_details]);
```

```
ribbon_sub_category_subtyped_single_entity_view =  
  rscssev(ribbon_sub_category_subtyped_single_entity_view_identifier, subtyped_single_entity,  
  [include add button], [visual_elements], {dialog_navigation}, [layout_details]);
```

```
ribbon_sub_categories_categorized_single_entity_view =  
  rscssev(ribbon_sub_categories_categorized_single_entity_view_identifier,  
  categorized_single_entity, [include add button], [visual_elements], {dialog_navigation},  
  [layout_details]);
```

```
ribbon_sub_categories_categorized_subtyped_single_entity_view =  
  rscssev(ribbon_sub_categories_categorized_subtyped_single_entity_view_identifier,  
  categorized_subtyped_single_entity, [include add button], [visual_elements], {dialog_navigation},  
  [layout_details]);
```

```
ribbon_ticker_category_view =  
  rtcv(ribbon_ticker_category_view_identifier, single_entity, [visual_elements], {dialog_navigation},  
  [layout_details]);
```

```
sensor_feeds_view =  
  sfv(sensor_feeds_view_identifier, sensor_feed | typed_sensor_feed, [visual_elements],  
  {dialog_navigation}, [layout_details]);
```

```
sensor_feed = value_collection;
```

```
typed_sensor_feed = typed_value_collection;
```

```
single_instance_view =  
  siv(single_instance_view_identifier, extended_single_entity, [visual_elements], {dialog_navigation},  
  [layout_details]);
```

```
single_instance_with_proposed_standard_text_view =  
  siwpsv(single_instance_with_proposed_standard_text_view_identifier, extended_single_entity,  
  attribute_for_standard_texts, {standard_text}, [visual_elements], {dialog_navigation},  
  [layout_details]);
```

```
attribute_for_standard_texts = attribute_identifier;
```

```
standard_text = string;
```

```
table_view = tv(table_view_identifier, extended_single_entity, [visual_elements], {dialog_navigation},  
  [layout_details]);
```

```
tree_for_composite_view =  
    tfcv(tree_for_composite_view_identifier, composite, [visual_elements], {dialog_navigation},  
    [layout_details]);
```

```
tree_for_one_to_many_hierarchy_view =  
    tfootmv(tree_for_one_to_many_hierarchy_view_identifier, one_to_many_hierarchy,  
    [visual_elements], {dialog_navigation}, [layout_details]);
```

4.2.2.4 Model Patterns Used in Content Views

```
categorized_single_entity = cse(main_entity, categorizer);
```

```
main_entity = entity_design;
```

```
categorizer = categorizing_entity | categorizing_attribute;
```

```
categorizing_entity = entity_identifier;
```

```
categorizing_attribute = attribute_identifier;
```

```
composite = comp(composite_entity, component_entity, leaf_entity);
```

```
composite_entity = entity_identifier;
```

```
component_entity = entity_identifier;
```

```
leaf_entity = entity_identifier;
```

```
one_to_many_hierarchy = entity_without_members | otmh(owner_entity, one_to_many_hierarchy);
```

```
entity_without_members = entity_identifier;
```

```
owner_entity = entity_identifier;
```

```
extended_owner_entity_with_member = eoewm(the_extended_single_entity, member_entity);
```

```
the_extended_single_entity = extended_single_entity;
```

```
single_entity = se(main_entity);
```

```
single_attribute_from_single_entity = (entity_identifier, attribute_design);
```

```
subtyped_single_entity = sse(main_entity, {subtype});
```

```
extended_single_entity = ese(main_entity, {subtype}, {one_related});
```

```
categorized_subtyped_single_entity = csse(the_subtyped_single_entity, categorizer);
```

the_subtyped_single_entity = *subtyped_single_entity*;

point_localized_extended_single_entity =
plse(*the_extended_single_entity*, *point_localizer*);

area_localized_categorized_single_entity =
alcse(*main_entity_identifiser*, *area_localizer*, *categorizer*);

single_entity_with_point_collection =
sewpc(*main_entity_identifiser*, *point_collection_provider*, {*subtype*}, {*one_related*});

point_collection_provider may be the same entity as *main_entity_identifiser*.

main_entity_identifiser = *entity_identifiser*;

point_localizer = *point_localizing_entity* | *point_localizing_attribute*;

point_localizing_entity = *entity_identifiser*;

point_localizing_attribute = *attribute_identifiser*;

area_localizer = *area_localizing_entity* | *area_localizing_attribute*;

area_localizing_entity = *entity_identifiser*;

area_localizing_attribute = *attribute_identifiser*;

point_collection_provider = *entity_identifiser*;

member_entity = *entity_design*;

subtype = *entity_design*;

one_related = *entity_design*;

entity_design = **ed**(*entity_identifiser*, {*attribute_design*}, {*method_design*}, {*annotation_design*});

attribute_design = **ad**(*attribute_identifiser*, {*annotation_design*});

method_design = **md**(*method_identifiser*, {*annotation_design* });

annotation_design = *annotation_identifiser*;

value_collection = **vc**(*collection*, *value_provider*);

collection = *entity_design*;

```
value_provider = entity_design;
```

```
typed_value_collection = tv(collection, value_provider, value_type);
```

```
value_type = entity_design;
```

4.2.2.5 Content Integration View

```
combined_map_and_ribbon_view =  
    cmrv(combined_map_and_ribbon_view_identifier, view_for_map, view_for_ribbon,  
        [visual_elements], {dialog_navigation}, [layout_details]);
```

```
view_for_map = interactor_design_view;
```

```
view_for_ribbon = interactor_design_view;
```

```
map_view =  
    mv(map_view_identifier, {view_for_child_map}, [visual_elements], {dialog_navigation},  
        [layout_details]);
```

```
view_for_child_map = interactor_design_view;
```

```
ribbon_buttons_view =  
    rbv(ribbon_buttons_view_identifier, {view_for_ribbon_button}, [visual_elements],  
        {dialog_navigation}, [layout_details]);
```

```
view_for_ribbon_button = interactor_design_view;
```

```
ribbon_categories_view =  
    rcsv(ribbon_categories_view_identifier, {view_for_ribbon_category}, [visual_elements],  
        {dialog_navigation}, [layout_details]);
```

```
view_for_ribbon_category = interactor_design_view;
```

```
ribbon_category_view =  
    rcv(ribbon_category_view_identifier, view_for_overview, views_for_categories, [visual_elements],  
        {dialog_navigation}, [layout_details]);
```

```
view_for_overview = basic_content_presenter_design_view;
```

```
views_for_categories = views_for_sub_category / view_for_sub_categories;
```

```
views_for_sub_category = { basic_content_presenter_design_view };
```

```
view_for_sub_categories = basic_content_presenter_design_view;
```

```
ribbon_contents_view =  
    rctv(ribbon_contents_view_identifier, view_for_ribbon_categories, view_for_ribbon_buttons,  
        [visual_elements], {dialog_navigation}, [layout_details]);
```

```
view_for_ribbon_categories = interactor_design_view;
```

```
view_for_ribbon_buttons = interactor_design_view;
```

```
ribbon_ticker_view =  
    rtv(ribbon_ticker_view_identifier, {view_for_ribbon_ticker_category}, [visual_elements],  
        {dialog_navigation}, [layout_details]);
```

```
view_for_ribbon_ticker_category = interactor_design_view;
```

```
ribbon_view =  
    rv(ribbon_view_identifier, view_for_ribbon_contents, view_for_ribbon_ticker, [visual_elements],  
        {dialog_navigation}, [layout_details]);
```

```
view_for_ribbon_contents = interactor_design_view;
```

```
view_for_ribbon_ticker = interactor_design_view;
```

4.2.2.6 Interactor Design View

As Category Manager Designs may not be members of any other interactor designs there is no need for a Category Manager Design View construct.

```
interactor_design_view =  
    content_presenter_design_view | task_supporter_design_view | work_supporter_design_view;
```

```
content_presenter_design_view =  
    cpdv(content_presenter_design_identifier, [layout_details]);
```

```
task_supporter_design_view = tsdv(task_supporter_design_identifier, [layout_details]);
```

```
work_supporter_design_view = wsdv(work_supporter_design_identifier, [layout_details]);
```

4.2.2.7 Dialog navigation

Dialog navigation may have different sources, if no source is provided, the source is the view in which the dialog navigation is specified. The target is either another view or return.

```
dialog_navigation =  
    dn(dialog_navigation_identifier, navigation_type, [navigation_source], [navigation_target]);
```

navigation_type =
open | **show** | **hide** | **close** | **return** ;

navigation_source =
entity_identifier |(*attribute_identifier*, *entity_identifier*) | *button_identifier*;

navigation_target =
view_identifier;

4.2.3 Semantics

In this section, we present the production rules defining the semantics for each of the five view types in FLUIDE-D, as well as the model patterns used in the Content Views and dialog navigation. Before addressing the five view types (and the model patterns and dialog navigation), we provide the semantics for the parts of the abstract syntax that are used by all or most of the view types as well as by the interactor design constructs. There are also some part of some constructs that are used in more than one view. The semantics of these are only specified in the first construct in which they are used.

[[*layout_details*]] =

[[*coordinates*]] | [[*relative_layout_details*]]

[[*coordinates*]] =

[[**coord**(*x*, *y*, *width*, *height*)]]

[[**coord**(*x*, *y*, *width*, *height*)]] =

is located *x* percent into the horizontal, and *y* percent into the the vertical extent of the parent view. The width is *width* percent of the width of the partent view. The high is *height* percent of the hight of the partent view.

[[*relative_layout_details*]] =

[[**top left parent**]] | [[**filled parent**]] | [[**top filled parent**]] | [[**sibling right**]] | [[**sibling left**]] | [[**sibling over**]] | [[**sibling under**]]

[[**top left parent**]] =

is located at the top left of its partent view. Its width and height will be determined automatically.

[[**filled parent**]] =

uses all available space in the partent view. Its vertical location and height will be determined automatically.

[[**top filled parent**]] =

uses all available space in the top part of the parent view. Its height will be determined automatically.

[[**sibling right**]] =

is located to the right of its prior sibling view. Its width and height will be determined automatically.

[[**sibling left**]] =

is located to the left of its prior sibling view. Its width and height will be determined automatically.

[[**sibling over**]] =

is located over its prior sibling view. Its width and height will be determined automatically.

[[**sibling under**]] =

is located under its prior sibling view. Its width and height will be determined automatically.

[[*visual_elements*]] =

[[**ves**([[*image*]], [[*text*]], [[*border_colour*]], {*graphics*}, {*button*})]]

[[**ves**([[*image*]], [[*text*]], [[*border_colour*]], {*graphics*}, {*button*})]] =

[[*image*]] [[*text*]] [[*border_colour*]] { [[*graphics*]] } { [[*button*]]

[[*image*]] =

[[*static_picture*]] | [[*picture_attribute*]]

[[*static_picture*]] =

A static picture is shown or used as icon.

[[*picture_attribute*]] =

A dynamic picture provided by *attribute_identifier* is shown or used as icon.

[[*text*]] =

[[*static_string*]] | [[*text_attribute*]]

[[*static_string*]] =

The text *string* is used as heading or label.

[[*picture_attribute*]] =

A dynamic text provided by *attribute_identifier* is used as heading or label.

[[*border_colour*]] =

The colour of the border is [[*colour_spec*]].

[[*colour_spec*]] =

[[*static_colour*]] | [[**use annotation**]]

[[*static_colour*]] =

colour

[[**use annotation**]] =

determined by an annotation

[[*graphics*]] =

[[**gr**(*shape*, *colour_spec*)]]

[[**gr**(*shape*, *colour_spec*)]] =

A [[*shape*]] which colour is [[*colour_spec*]].

[[*shape*]] =

[[**circle**]] | [[**triange**]] | [[**square**]]

[[**circle**]] =

visual circle

[[**triange**]] =

visual triangle

[[**square**]]=

visual square

[[*button*]]=

[[**btn**(*button_identifier*, [*button_image*], [*button_text*)]]]

[[**btn**(*button_identifier*, [*button_image*], [*button_text*)]] =

The button *button_identifier* [with the image [[*button_image*]]] [and the text [[*button_text*]]].

4.2.3.1 Decorational View

[[*decorational_view*]] =

[[**dv**(*decorational_view_identifier*, *role_of_decorational_view*, [**present date**], [**present time**], [*visual_elements*], {*dialog_navigation*}, *layout_method*, [*layout_details*], {*child_view*}-)]]

[[**dv**(*decorational_view_identifier*, *role_of_decorational_view*, [**present date**], [**present time**], [*visual_elements*], {*dialog_navigation*}, *layout_method*, [*layout_details*], {*child_view*}-)]] =

The Decorational View *decorational_view_identifier* which represents [[*role_of_decorational_view*]] in which a following content are presented:

[[[**present date**]]] [[[**present time**]]] [[[*visual_elements*]]] { [[*child_view*]] }

The layout of this content is [[*layout_method*]].

[In the context of its parent view, *decorational_view_identifier* [[*layout_details*]].]

decorational_view_identifier is the source for { [[*dialog_navigation*]] }.

[[*role_of_decorational_view*]] =

[[**loosely connected windows**]] | [[**window**]] | [[**border**]]

[[**loosely connected windows**]] =

a set of loosely connected windows or a number of full screen renderings

[[**windows**]] =

a window or a full screen rendering

[[**border**]] =

a visually marked border, usually a rectangle

[[**present date**]] =

The current date.

[[**present time**]] =

The current time.

[[*child_view*]] =

[[*view*]].

[[*layout_method*]] =

[[**percentage**]] | [[**relative**]] | [[**managed**]] | [[**automatic**]]

[[**percentage**]] =

given as percentage values of *decorational_view_identifier* in the child views

[[**relative**]] =

given by rules in the child views for how they should be placed relative to their parent or a sibling view

[[**managed**]] =

given through one or more child Layout Manager Views

[[**automatic**]] =

determined by a layout algorithm provided by *decorational_view_identifier*

4.2.3.2 Layout Manager View

[[*layout_manager_view*]] =

[[**Imv**(*layout_manager_view_identifier*, *layout_direction*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*], {*child_view*}-)]]

[[**Imv**(*layout_manager_view_identifier*, *layout_direction*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*], {*child_view*}-)]] =

The Layout Manager *layout_manager_view_identifier* which is invisible and presents:

[[[*visual_elements*]] { [[*child_view*]}]

This content is presented [[*layout_direction*]].

[In the context of its parent view, *layout_manager_view_identifier* [[*layout_details*]].]

layout_manager_view_identifier is the source for { [[*dialog_navigation*]]}.

[[*layout_direction*]] =

[[horizontal]] | [[vertical]]

[[horizontal]] =

side by side horizontally

[[vertical]] =

over/under each other vertically

4.2.3.3 Content View

[[3D_icons_imposed_on_camera_view]] =

[[**3Diocv**(*3D_icons_imposed_on_camera_view_identifier*, *main_entity_identifier*, *point_localized_extended_single_entity*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]]]

[[**3Diocv**(*3D_icons_imposed_on_camera_view_identifier*, *main_entity_identifier*, *point_localized_extended_single_entity*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]] =

The 3D Icons Imposed on Camera View *3D_icons_imposed_on_camera_view_identifier* provides an augmented reality presentation where 3D icons are imposed on a real time camera image from a mobile device. The 3D icons represent one or more instances of *main_entity_identifier*. The icons may include a presentation of [[*the_extended_single_entity*]].

The location of the 3D icons are determined by [[*point_localizer*]].

[*3D_icons_imposed_on_camera_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *3D_icons_imposed_on_camera_view* [[*layout_details*].]

3D_icons_imposed_on_camera_view_identifier is the source for {[[*dialog_navigation*]]}.

[[*body_parts_visualization_view*]] =

[[**bpvv**(*body_parts_visualization_view_identifier*, *single_entity*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]]]

[[**bpvv**(*body_parts_visualization_view_identifier*, *single_entity*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]] =

The Body Parts Visualization View *body_parts_visualization_view_identifier* provides a graphical presentation of one or more body parts that are affected. The affected body parts are given by [[*main_entity*]].

[*body_parts_visualization_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *body_parts_visualization_view_identifier* `[[layout_details]]`.]

body_parts_visualization_view_identifier is the source for `{[[dialog_navigation]]}`.

`[[browser_for_composite_view]]` =

`[[bfcv(browser_for_composite_view_identifier, composite, [visual_elements], {dialog_navigation}, [layout_details])]]`

`[[bfcv(browser_for_composite_view_identifier, composite, [visual_elements], {dialog_navigation}, [layout_details])]]` =

The Browser for Composite View *browser_for_composite_view_identifier* provides a browser in which each level of a composite structure is presented in a list. Such a list contains one or more instances of `[[component_entity]]`. If an item in the list is selected, details about this entity is shown in a details pane. If the selected list item is an instance of `[[composite_entity]]`, its children are shown in an additional list view. If the selected list item is an instance of `[[leaf_entity]]`, all list views reflecting lower levels in the hierarchy are closed. There are no restrictions with regard to the number of levels, and thus the number of lists in the view.

browser_for_composite_view_identifier also includes `[[visual_elements]]`.

[In the context of its parent view, *browser_for_composite_view_identifier* `[[layout_details]]`.]

browser_for_composite_view_identifier is the source for `{[[dialog_navigation]]}`.

`[[browser_for_one_to_many_hierarchy_view]]` =

`[[bftmv(browser_for_one_to_many_hierarchy_view_identifier, one_to_many_hierarchy, [visual_elements], {dialog_navigation}, [layout_details])]]`

`[[bftmv(browser_for_one_to_many_hierarchy_view_identifier, one_to_many_hierarchy, [visual_elements], {dialog_navigation}, [layout_details])]]` =

The Browser for One to Many Hierarchy View *browser_for_one_to_many_hierarchy_view_identifier* provides a browser in which each level of a tree of entities connected through one to many relations is presented in a list. Such a list contains one or more instances of the entity at a given level. If an item in the list is selected, details about this entity is shown in a details pane. If the entity that the selected list item is an instance of has a related entity at the level below, its children are shown in an additional list view. If the entity that the selected list item is an instance of does not have related entity at the level below, all list views reflecting lower levels in the hierarchy are closed. The hierarchy of entities that may be presented in the browser consists of:

`[[one_to_many_hierarchy]]`

browser_for_one_to_many_hierarchy_view_identifier also includes `[[visual_elements]]`.

[In the context of its parent view, *browser_for_one_to_many_hierarchy_view_identifier* `[[layout_details]]`.]

browser_for_one_to_many_hierarchy_view_identifier is the source for `{[[dialog_navigation]]}`.

`[[icon_list_view]] =`

`[[ilv(icon_list_view_identifier, single_entity, [[visual_elements]], {dialog_navigation}, [[layout_details]])]]`

`[[ilv(icon_list_view_identifier, single_entity, [[visual_elements]], {dialog_navigation}, [[layout_details]])]]` =

The Icon List View *icon_list_view_identifier* provides an icon-based presentation of one or more instances of `[[main_entity]]` in a list.

icon_list_view_identifier also includes `[[visual_elements]]`.

[In the context of its parent view, *icon_table_view_identifier* `[[layout_details]]`.]

icon_list_view_identifier is the source for `{[[dialog_navigation]]}`.

`[[icon_table_view]] =`

`[[itv(icon_table_view_identifier, categorized_single_entity, [[visual_elements]], [[layout_details]])]]`

`[[itv(icon_table_view_identifier, categorized_single_entity, [[visual_elements]], {dialog_navigation}, [[layout_details]])]]` =

The Icon Table View *icon_table_view_identifier* provides an icon-based presentation of one or more instances of `[[main_entity]]`. The icons are organized in groups given by `[[categorizer]]`.

icon_table_view_identifier also includes `[[visual_elements]]`.

[In the context of its parent view, *icon_table_view_identifier* `[[layout_details]]`.]

icon_table_view_identifier is the source for `{[[dialog_navigation]]}`.

`[[icon_table_subtypes_view]] =`

`[[itsv(icon_table_subtypes_view_identifier, subtyped_single_entity, [[visual_elements]], {dialog_navigation}, [[layout_details]])]]`

[[**itsv**(*icon_table_subtypes_view_identifier*, *subtyped_single_entity*, [visual_elements], {dialog_navigation}, [layout_details])]] =

The Icon Table Subtypes View *icon_table_subtypes_view_identifier* provides an icon-based presentation of one or more instances of [[*subtyped_single_entity*]]. The subtypes are usually presented by different icons.

[*icon_table_subtypes_view_identifier* also includes [[visual_elements]].

[In the context of its parent view, *icon_table_subtypes_view_identifier* [[layout_details].]

icon_table_subtypes_view_identifier is the source for {[[dialog_navigation]].

[[*list_details_view*]] =

[[**ldv**(*list_details_view_identifier*, *main_entity_identifier*, *entity_with_list_attributes*, *entity_with_details_attributes*, [visual_elements], {dialog_navigation}, [layout_details])]]

[[**ldv**(*list_details_view_identifier*, *main_entity_identifier*, *entity_with_list_attributes*, *entity_with_details_attributes*, [visual_elements], {dialog_navigation}, [layout_details])]] =

The List + Details View *list_details_view_identifier* provides two synchronized presentations of *main_entity_identifier*. One presents a list of one or more instances of [[*entity_with_list_attributes*]]. The other gives a forms-based presentation of the details of [[*entity_with_details_attributes*]] from the instance represented by the item selected in the list.

[*list_details_view_identifier* also includes [[visual_elements]].

[In the context of its parent view, *list_details_view_identifier* [[layout_details].]

list_details_view_identifier is the source for {[[dialog_navigation]].

[[*list_view*]] =

[[**lv**(*list_view_identifier*, *extended_single_entity*, [visual_elements], {dialog_navigation}, [layout_details])]]

[[**lv**(*list_view_identifier*, *extended_single_entity*, [visual_elements], {dialog_navigation}, [layout_details])]] =

The List View *list_view_identifier* provides a simple list-based presentation (using a single user interface control) of one or more instances of [[*extended_single_entity*]].

[*list_view_identifier* also includes [[visual_elements]].

[In the context of its parent view, *list_view_identifier* `[[layout_details]]`.]

list_view_identifier is the source for `{[[dialog_navigation]]`.

`[[map_icons_view]]` =

`[[micv(map_icons_view_identifier, main_entity_identifier, point_localized_extended_single_entity, map_buttons, [visual_elements], {dialog_navigation}, [layout_details])]]`

`[[micv(map_icons_view_identifier, main_entity_identifier, point_localized_extended_single_entity, map_buttons, [visual_elements], {dialog_navigation}, [layout_details])]]` =

The Map Icons View *map_icons_view_identifier* provides a map-based presentation of icons representing one or more instances of *main_entity_identifier*. The icons may include a presentation of `[[the_extended_single_entity]]`. The location of the icons are determined by `[[point_localizer]]`. If *map_icons_view_identifier* does not have a parent providing a Map View, it will provide its own map.

map_icons_view_identifier includes `[[map_buttons]]`

`[[map_icons_view_identifier]` also includes `[[visual_elements]]`.

[In the context of its parent view, *map_icons_view_identifier* `[[layout_details]]`.]

map_icons_view_identifier is the source for `{[[dialog_navigation]]`.

`[[map_buttons]]` =

`[[mb([include mode button], [include center button], [include projector button])]]`

`[[mb([include mode button], [include center button], [include projector button])]]` =

`[[[include mode button]] [[include center button]] [[include projector button]]`

`[[include mode button]]` =

a button for accessing functionality for adjusting the presentation mode of the map

`[[include center button]]` =

a button for centering the map

`[[include projector button]]` =

a button for opening an augmented reality presentation of the contents in the map

[[*map_icons_with_details_dialog_view*]] =

[[**miwddv**(*map_icons_with_details_dialog_view_identifier*, *main_entity_identifier*,
point_localized_extended_single_entity, *map_buttons*, [*visual_elements*], {*dialog_navigation*},
[*layout_details*])]]

[[**miwddv**(*map_icons_with_details_dialog_view_identifier*, *main_entity_identifier*,
point_localized_extended_single_entity, *map_buttons*, [*visual_elements*], {*dialog_navigation*},
[*layout_details*)]] =

The Map Icons with Details Dialog View *map_icons_with_details_dialog_view_identifier* provides a map-based presentation of icons representing one or more instances of *main_entity_identifier*. The icons may include a presentation of [[*the_extended_single_entity*]]. The location of the icons are determined by [[*point_localizer*]]. The icons provide functionality for separate forms-based presentations of [[*the_extended_single_entity*]] of selected instances. If *map_icons_with_details_dialog_view_identifier* does not have a parent providing a Map View, it will provide its own map.

map_icons_with_details_dialog_view_identifier includes [[*map_buttons*]]

[*map_icons_with_details_dialog_view_identifier* also includes [[*visual_elements*]]].

[In the context of its parent view, *map_icons_with_details_dialog_view_identifier* [[*layout_details*]].]

map_icons_with_details_dialog_view_identifier is the source for {{*dialog_navigation* }}.

[[*map_multi_line_view*]] =

[[**mmlv**(*map_multi_line_view_identifier*, *single_entity_with_point_collection*, *map_buttons*,
[*visual_elements*], {*dialog_navigation*}, [*layout_details*])]]

[[**mmlv**(*map_multi_line_view_identifier*, *single_entity_with_point_collection*, *map_buttons*,
[*visual_elements*], {*dialog_navigation*}, [*layout_details*])] =

The Map Multi-line View *map_multi_line_view_identifier* provides a map-based presentation of a set of line segments related to *main_entity_identifier*. The points defining the line segments are obtained from *point_collection_provider*. If *map_multi_line_view_identifier* does not have a parent providing a Map View, it will provide its own map.

map_multi_line_view_identifier includes [[*map_buttons*]]

[*map_multi_line_view_identifier* also includes [[*visual_elements*]]].

[In the context of its parent view, *map_multi_line_view_identifier* [[*layout_details*]].]

map_multi_line_view_identifier is the source for {{*dialog_navigation* }}.

[[*map_outline_view*]] =

[[**mov**(*map_outline_view_identifier*, *area_localized_categorized_single_entity*, *map_buttons*,
[*visual_elements*], {*dialog_navigation*}, [*layout_details*])]]

[[**mov**(*map_outline_view_identifier*, *area_localized_categorized_single_entity*, *map_buttons*,
[*visual_elements*], {*dialog_navigation*}, [*layout_details*])]] =

The Map Outline View *map_outline_view_identifier* provides a map-based presentation of outlines representing one or more instances of *main_entity_identifier*. The visual appearance of the outlines may be influenced by [[*categorizer*]]. The location of the outlines are determined by [[*area_localizer*]]. If *map_outline_view_identifier* does not have a parent providing a Map View, it will provide its own map.

map_outline_view_identifier includes [[*map_buttons*]]

[*map_outline_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent *map_outline_view_identifier* [[*layout_details*]].]

map_outline_view_identifier is the source for {[[*dialog_navigation*]]}.

[[*media_player_view*]] =

[[**mpv**(*media_player_view_identifier*, {*single_attribute_from_single_entity*}, [*visual_elements*],
{*dialog_navigation*}, [*layout_details*])]]

[[**mpv**(*media_player_view_identifier*, {*single_attribute_from_single_entity*}, [*visual_elements*],
{*dialog_navigation*}, [*layout_details*])]] =

The Media-player View *media_player_view_identifier* presents media contents from one instance of *entity_identifier*. The media contents are identified by {[[*attribute_design*]]}.

[*media_player_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *media_player_view_identifier* [[*layout_details*]].]

media_player_view_identifier is the source for {[[*dialog_navigation*]]}.

[[*multi_instance_view*]] =

[[**miv**(*multi_instance_view_identifier*, *extended_single_entity*, [*visual_elements*],
{*dialog_navigation*}, [*layout_details*])]]

[[**miv**(*multi_instance_view_identifier*, *extended_single_entity*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]] =

The Multi-instance View *multi_instance_view_identifier* provides a forms-based presentation of one or more instances of [[*extended_single_entity*]] using separate user interface controls for each attribute.

[*multi_instance_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *multi_instance_view_identifier* [[*layout_details*].]

multi_instance_view_identifier is the source for {[[*dialog_navigation*]]}.

[[*owner_member_view*]] =

[[**omv**(*owner_member_view_identifier*, *extended_owner_entity_with_member*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]]

[[**omv**(*owner_member_view_identifier*, *extended_owner_entity_with_member*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]] =

The Owner + Member View *owner_member_view_identifier* provides a master/details forms-based presentation. The master part presents one instance of [[*the_extended_single_entity*]]. The details part presents one or more instances of the many-related [[*member_entity*]].

[*owner_member_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *owner_member_view_identifier* [[*layout_details*].]

owner_member_view_identifier is the source for {[[*dialog_navigation*]]}.

[[*ribbon_button_view*]] =

[[**rbv**(*ribbon_button_view_identifier*, *single_entity*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]]

[[**rbv**(*ribbon_button_view_identifier*, *single_entity*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]] =

The Ribbon Button View *ribbon_button_view_identifier* provides a button intended to be part of a ribbon. The button may show the name of *entity_identifier* as well as [[*main_entity*]] as part of the button. The button also provides access to a ribbon category.

[*ribbon_button_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *ribbon_button_view_identifier* [[*layout_details*].]

ribbon_button_view_identifier is the source for `[[dialog_navigation]]`.

`[[ribbon_category_overview_view]]` =

`[[rcov(ribbon_category_overview_view_identifier, single_entity, ribbon_category_buttons, [visual_elements], { dialog_navigation }, [layout_details])]]`

`[[rcov(ribbon_category_overview_view_identifier, single_entity, ribbon_category_buttons, [visual_elements], { dialog_navigation }, [layout_details])]]` =

The Ribbon Category Overview View *ribbon_category_overview_view_identifier* provides overview information about a ribbon category. This includes the name of *entity_identifier* as well as `[[main_entity]]`.

ribbon_category_overview_view_identifier includes `[[ribbon_category_buttons]]`

ribbon_category_overview_view_identifier also includes `[[visual_elements]]`.

[In the context of its parent view, *ribbon_category_overview_view_identifier* `[[layout_details]]`.]

ribbon_category_overview_view_identifier is the source for `[[dialog_navigation]]`.

`[[ribbon_category_buttons]]` =

`[[rcb([include tree button], [include home button], [include table button], [include search button])]]`

`[[rcb([include tree button], [include home button], [include table button], [include search button])]]` =

`[[[include tree button]] [[include home button]] [[include table button]] [[include search button]]]]`

`[[include tree button]]` =

a button for opening a presentation of the instances of the category in a tree view

`[[include home button]]` =

a button for going back to the ribbon buttons

`[[include table button]]` =

a button for opening a presentation of the instances of the category in a table view

`[[include search button]]` =

a button for accessing the search function

[[*ribbon_sub_category_single_entity_view*]] =

[[**rscsev**(*ribbon_sub_category_single_entity_view_identifier*, *single_entity*, [**include add button**], [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]]]

[[**rscsev**(*ribbon_sub_category_single_entity_view_identifier*, *single_entity*, [**include add button**], [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]]]

The Ribbon Sub Category Single Entity View *ribbon_sub_category_single_entity_view_identifier* provides an icon-based presentation of one sub category of a ribbon as a grid or table of icons. The icons represent one or more instances of [[*main_entity*]].

[*ribbon_sub_category_single_entity_view_identifier* includes [[**include add button**]].

[*ribbon_sub_category_single_entity_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *ribbon_sub_category_single_entity_view_identifier* [[*layout_details*]].]

ribbon_sub_category_single_entity_view_identifier is the source for {{ [[*dialog_navigation*]].

[[**include add button**]] =

a button for adding instances of the entity presented in the ribbon category

[[*ribbon_sub_category_subtyped_single_entity_view*]] =

[[**rscssev**(*ribbon_sub_category_subtyped_single_entity_view_identifier*, *subtyped_single_entity*, [**include add button**], [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]]]

[[**rscssev**(*ribbon_sub_category_subtyped_single_entity_view_identifier*, *subtyped_single_entity*, [**include add button**], [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]]]

The Ribbon Sub Category Subtyped Single Entity View *ribbon_sub_category_subtyped_single_entity_view_identifier* provides an icon-based presentation of one sub category of a ribbon as a grid or table of icons. The icons represent one or more instances of [[*subtyped_single_entity*]]. The subtypes are usually presented by different icons.

[*ribbon_sub_category_subtyped_single_entity_view_identifier* includes [[**include add button**]].

[*ribbon_sub_category_subtyped_single_entity_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *ribbon_sub_category_subtyped_single_entity_view_identifier* `[[layout_details]].`]

ribbon_sub_category_subtyped_single_entity_view_identifier is the source for `{{dialog_navigation}}`.

`[[ribbon_sub_categories_categorized_single_entity_view]] =`

`[[rscssev(ribbon_sub_categories_categorized_single_entity_view_identifier, categorized_single_entity, [include add button], [visual_elements], {dialog_navigation}, [layout_details])]]`

`[[rscssev(ribbon_sub_categories_categorized_single_entity_view_identifier, categorized_single_entity, [include add button], [visual_elements], {dialog_navigation}, [layout_details])]]` =

The Ribbon Sub Categories Categorized Single Entity View *ribbon_sub_categories_categorized_single_entity_view_identifier* provides an icon-based presentation of a ribbon category as a grid or table of icons divided into sub categories. The icons represent one or more instances of `[[main_entity]]`. The sub categories are determined by `[[categorizer]]`.

ribbon_sub_categories_categorized_single_entity_view_identifier includes `[[include add button]]`.

ribbon_sub_categories_categorized_single_entity_view_identifier also includes `[[visual_elements]]`.

[In the context of its parent view, *ribbon_sub_categories_categorized_single_entity_view_identifier* `[[layout_details]].`]

ribbon_sub_categories_categorized_single_entity_view_identifier is the source for `{{dialog_navigation}}`.

`[[ribbon_sub_categories_categorized_subtyped_single_entity_view]] =`

`[[rscssev(ribbon_sub_categories_categorized_subtyped_single_entity_view_identifier, categorized_subtyped_single_entity, [include add button], [visual_elements], {dialog_navigation}, [layout_details])]]`

`[[rscssev(ribbon_sub_categories_categorized_subtyped_single_entity_view_identifier, categorized_subtyped_single_entity, [include add button], [visual_elements], {dialog_navigation}, [layout_details])]]` =

The Ribbon Sub Categories Categorized Subtyped Single Entity View *ribbon_sub_categories_categorized_subtyped_single_entity_view_identifier* provides an icon-based presentation of a ribbon category as a grid or table of icons divided into sub categories. The icons

represent one or more instances of `[[the_subtyped_single_entity]]`. The sub categories are determined by `[[categorizer]]`. Within each sub category the subtypes are usually presented by different icons.

`[[ribbon_sub_categories_categorized_subtyped_single_entity_view_identifier]]` includes `[[include add button]]`.

`[[ribbon_sub_categories_categorized_subtyped_single_entity_view_identifier]]` also includes `[[visual_elements]]`.

[In the context of its parent view, `ribbon_sub_categories_categorized_subtyped_single_entity_view_identifier` `[[layout_details]]`.]

`ribbon_sub_categories_categorized_subtyped_single_entity_view_identifier` is the source for `{[[dialog_navigation]]`.

`[[ribbon_ticker_category_view]]` =

`[[rtcv(ribbon_ticker_category_view_identifier, single_entity, [visual_elements], {dialog_navigation}, [layout_details])]]`

`[[rtcv(ribbon_ticker_category_view_identifier, single_entity, [visual_elements], {dialog_navigation}, [layout_details])]]` =

The Ribbon Ticker Category View `ribbon_ticker_category_view_identifier` presents key information from one ribbon category as part of a ribbon ticker. The key information that is presented is `[[main_entity]]`.

`[[ribbon_ticker_category_view_identifier]]` also includes `[[visual_elements]]`.

[In the context of its parent view, `ribbon_ticker_category_view_identifier` `[[layout_details]]`.]

`ribbon_ticker_category_view_identifier` is the source for `{[[dialog_navigation]]`.

`[[sensor_feeds_view]]` =

`[[sfv(sensor_feeds_view_identifier, sensor_feed | typed_sensor_feed, [visual_elements], {dialog_navigation}, [layout_details])]]`

`[[sfv(sensor_feeds_view_identifier, sensor_feed | typed_sensor_feed, [visual_elements], {dialog_navigation}, [layout_details])]]` =

The Sensor Feeds View `sensor_feeds_view_identifier` provides a graphical presentation of a set of sensor values from one or more sensors. `[[sensor_feed]]` | `[[typed_sensor_feed]]`

`[[sensor_feeds_view_identifier]]` also includes `[[visual_elements]]`.

[In the context of its parent view, *sensor_feeds_view_identifier* [[*layout_details*].]

sensor_feeds_view_identifier is the source for {[*dialog_navigation*]}.

[[*sensor_feed*]] =

The sensors are determined by [[*collection*]]. The sensor values to present are determined by [[*value_provider*]].

[[*typed_sensor_feed*]] =

The sensors are determined by [[*collection*]]. The sensor type is given by [[*value_type*]]. The sensor values to present are determined by [[*value_provider*]].

[[*single_instance_view*]] =

[[**si**(*single_instance_view_identifier*, *extended_single_entity*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]]]

[[**si**(*single_instance_view_identifier*, *extended_single_entity*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]] =

The Single Instance View *single_instance_view_identifier* which provides a forms-based presentation of one instance of [[*extended_single_entity*]] using separate user interface controls for each attribute.

[*single_instance_view_identifier* also includes [[*visual_elements*]]].

[In the context of its parent view, *single_instance_view_identifier* [[*layout_details*].]

single_instance_view_identifier is the source for {[*dialog_navigation*]}.

[[*single_instance_with_proposed_standard_text_view*]] =

[[**siwpstv**(*single_instance_with_proposed_standard_text_view_identifier*, *extended_single_entity*, *attribute_for_standard_texts*, {*standard_text*}, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]]]

[[**siwpstv**(*single_instance_with_proposed_standard_text_view_identifier*, *extended_single_entity*, *attribute_for_standard_texts*, {*standard_text*}, [*visual_elements*], {*dialog_navigation*}, [*layout_details*)]] =

The Single Instance with Proposed Standard Text View *single_instance_with_proposed_standard_text_view_identifier* provides a forms-based presentation of one instance of [[*extended_single_entity*]] using separate user interface controls for each attribute. Values for the attribute *attribute_for_standard_texts* may be chosen from a list containing the values

{`[[standard_text]]`}. Access to this list is provided from a button with the label "Choose standard attribute_for_standard_texts text".

`[[single_instance_with_proposed_standard_text_view_identifier]]` also includes `[[visual_elements]]`.

[In the context of its parent view, `single_instance_with_proposed_standard_text_view_identifier` `[[layout_details]]`.]

`single_instance_with_proposed_standard_text_view_identifier` is the source for `{[[dialog_navigation]]}`.

`[[table_view]] =`

`[[tv(table_view_identifier, extended_single_entity, [visual_elements], {dialog_navigation}, [layout_details])]]`

`[[tv(table_view_identifier, extended_single_entity, [visual_elements], {dialog_navigation}, [layout_details])]]` =

The Table View `table_view_identifier` provides a list-based presentation (using separate user interface controls for each attribute) of one or more instances of `[[extended_single_entity]]`.

`[[table_view_identifier]]` also includes `[[visual_elements]]`.

[In the context of its parent view, `table_view_identifier` `[[layout_details]]`.]

`table_view_identifier` is the source for `{[[dialog_navigation]]`.

`[[tree_for_composite_view]] =`

`[[tfcv(tree_for_composite_view_identifier, composite, [visual_elements], {dialog_navigation}, [layout_details])]]`

`[[tfcv(tree_for_composite_view_identifier, composite, [visual_elements], {dialog_navigation}, [layout_details])]]` =

The Tree for Composite View `tree_for_composite_view_identifier` provides a visual tree in which instances of a whole composite structure is presented. Each level in the tree contains one or more instances of `[[component_entity]]`. If an item in the tree is an instance of `[[composite_entity]]`, its children may be viewed in the tree by expanding the item. If an item in the tree is an instance of `[[leaf_entity]]`, it may not be expanded. There are no restrictions with regard to the number of levels, and thus the depth of the tree.

`[[tree_for_composite_view_identifier]]` also includes `[[visual_elements]]`.

[In the context of its parent view, `tree_for_composite_view_identifier` `[[layout_details]]`.]

tree_for_composite_view_identifier is the source for $\{\{\textit{dialog_navigation}\}\}$.

$\{\{\textit{tree_for_one_to_many_hierarchy_view}\}\} =$

$\{\{\textit{tftmv}(\textit{tree_for_one_to_many_hierarchy_view_identifier}, \textit{one_to_many_hierarchy},$
 $\{\textit{visual_elements}\}, \{\textit{dialog_navigation}\}, \{\textit{layout_details}\})\}\}$

$\{\{\textit{tftmv}(\textit{tree_for_one_to_many_hierarchy_view_identifier}, \textit{one_to_many_hierarchy}, \{\textit{visual_elements}\},$
 $\{\textit{dialog_navigation}\}, \{\textit{layout_details}\})\}\} =$

The Tree for One to Many Hierarchy View *tree_for_one_to_many_hierarchy_view_identifier* provides a visual tree in which instances of all levels of a hierarchy of one to many related entities is presented. Each level in the tree contains one or more instances of the entity at a given level. If the entity that an item in the tree is an instance of has a related entity at the level below, its children may be viewed in the tree by expanding the item. If the entity that an item in the tree is an instance of does not have related entity at the level below, it may not be expanded. The hierarchy of entities that may be presented in the tree consists of:

$\{\{\textit{one_to_many_hierarchy}\}\}$

tree_for_one_to_many_hierarchy_view_identifier also includes $\{\{\textit{visual_elements}\}\}$.

[In the context of its parent view, *tree_for_one_to_many_hierarchy_view_identifier* $\{\{\textit{layout_details}\}\}$.]

tree_for_one_to_many_hierarchy_view_identifier is the source for $\{\{\textit{dialog_navigation}\}\}$.

4.2.3.4 Model Patterns Used in Content Views²

$\{\{\textit{one_to_many_hierarchy}\}\} =$

$\{\{\textit{entity_without_members}\}\}$. $\{\{\textit{entity_without_members}\}\}$ does not have any related entities.
 $\{\{\textit{otmh}(\textit{owner_entity}, \textit{one_to_many_hierarchy})\}\}$

$\{\{\textit{otmh}(\textit{owner_entity}, \textit{one_to_many_hierarchy})\}\} =$

$\{\{\textit{owner_entity}\}\}$. $\{\{\textit{owner_entity}\}\}$ has the one to many related entity $\{\{\textit{one_to_many_hierarchy}\}\}$

$\{\{\textit{extended_single_entity}\}\} =$

$\{\{\textit{ese}(\textit{main_entity}, \{\textit{subtype}\}, \{\textit{one_related}\})\}\}$

$\{\{\textit{ese}(\textit{main_entity}, \{\textit{subtype}\}, \{\textit{one_related}\})\}\} =$

² Some of the EBNF expressions in Section 4.2.2.4 are not included in this section as the definitions of the semantics of the views using these model patterns exploit the semantics of individual elements of the model patterns rather than the semantics of the complete model pattern.

[[*main_entity*]] { [[*subtype*]] } { [[*one_related*]]

[[*subtyped_single_entity*]] =

[[**sse**(*main_entity*, {*subtype*})]]

[[**sse**(*main_entity*, {*subtype*})]] =

[[*main_entity*]] { [[*subtype*]]

[[*entity_design*]] =

[[**ed**(*entity_identifier*, {*attribute_design*}, {*method_design*}, {*annotation_design*})]]

[[**ed**(*entity_identifier*, {*attribute_design*}, {*method_design*}, {*annotation_design*})]] =

{ [[*attribute_design*]]} and { [[*method_design*]]} from *entity_identifier*, taking { [[*annotation_design*]]} into account

[[*attribute_design*]] =

[[**ad**(*attribute_identifier*, {*annotation_design*})]]

[[**ad**(*attribute_identifier*, {*annotation_design*})]] =

attribute_identifier, taking { [[*annotation_design*]]} into account

[[*method_design*]] =

[[**md**(*method_identifier*, {*annotation_design* })]]

[[**md**(*method_identifier*, {*annotation_design* })]] =

method_identifier, taking { [[*annotation_design*]]} into account

[[*categorizer*]] =

the one-related entity [[*categorizing_entity*]] | by the attribute [[*categorizing_attribute*]]

[[*point_localizer*]] =

the one-related entity [[*point_localizing_entity*]] | by the attribute [[*point_localizing_attribute*]]

[[*area_localizer*]] =

the one-related entity [[*area_localizing_entity*]] | by the attribute [[*area_localizing_attribute*]]

4.2.3.5 Content Integration View

[[*combined_map_and_ribbon_view*]] =

[[**cmarv**(*combined_map_and_ribbon_view_identifier*, *view_for_map*, *view_for_ribbon*,
[*visual_elements*], {*dialog_navigation*}, [*layout_details*)]]

[[**cmarv**(*combined_map_and_ribbon_view_identifier*, *view_for_map*, *view_for_ribbon*, [*visual_elements*],
{*dialog_navigation*}, [*layout_details*)]] =

The Combined Map and Ribbon View *combined_map_and_ribbon_view_identifier* puts together one interactor design containing a Map View with another interactor design containing a Ribbon View, together making up a user interface with the map and the ribbon working together. The map part is provided by:

[[*view_for_map*]]

The ribbon part is provided by:

[[*view_for_ribbon*]]

[*combined_map_and_ribbon_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *combined_map_and_ribbon_view_identifier* [[*layout_details*]].]

combined_map_and_ribbon_view_identifier is the source for {[*dialog_navigation*]}.

[[*map_view*]] =

[[**mv**(*map_view_identifier*, {*view_for_child_map*}, [*visual_elements*], {*dialog_navigation*},
[*layout_details*)]]

[[**mv**(*map_view_identifier*, {*view_for_child_map*}, [*visual_elements*], {*dialog_navigation*},
[*layout_details*)]] =

The Map View *map_view_identifier* provides a map for presenting various map overlays. The overlays are provided by the child interactor designs, which must contain either another Map View, a Map Icons View, a Map Icons with Details Dialog View, a Map Outline View, or a Map Multi Line View. Such views are provided by:

{[[*view_for_child_map*]]}

The overlays are shown on the map provided by the Map View highest up in the hierarchy.

[*map_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *map_view_identifier* [[*layout_details*]].]

map_view_identifier is the source for `{[[dialog_navigation]]}`.

`[[ribbon_buttons_view]]` =

`[[rbsv(ribbon_buttons_view_identifier, {view_for_ribbon_button}, [visual_elements],
{dialog_navigation}, [layout_details])]]`

`[[rbsv(ribbon_buttons_view_identifier, {view_for_ribbon_button}, [visual_elements], {dialog_navigation},
[layout_details])]]` =

The Ribbon Buttons View *ribbon_buttons_view_identifier* provides the button part of the top level of a ribbon, through putting together a set of interactor designs containing Ribbon Button Views. The ribbon buttons are provided by:

`{[[view_for_ribbon_button]]}`

ribbon_buttons_view_identifier also includes `[[visual_elements]]`.

[In the context of its parent view, *ribbon_buttons_view_identifier* `[[layout_details]].`]

ribbon_buttons_view_identifier is the source for `{[[dialog_navigation]]}`.

`[[ribbon_categories_view]]` =

`[[rcsv(ribbon_categories_view_identifier, {view_for_ribbon_category}, [visual_elements],
{dialog_navigation}, [layout_details])]]`

`[[rcsv(ribbon_categories_view_identifier, {view_for_ribbon_category}, [visual_elements],
{dialog_navigation}, [layout_details])]]` =

The Ribbon Categories View *ribbon_categories_view_identifier* puts together a set of a set of interactor designs containing Ribbon Category Views. The coupling is functional rather than visual. The ribbon categories are provided by:

`{[[view_for_ribbon_category]]}`

ribbon_categories_view_identifier also includes `[[visual_elements]]`.

[In the context of its parent view, *ribbon_categories_view_identifier* `[[layout_details]].`]

ribbon_categories_view_identifier is the source for `{[[dialog_navigation]]}`.

`[[ribbon_category_view]]` =

[[**rcv**(*ribbon_category_view_identifier*, *view_for_overview*, *views_for_categories*,
[*visual_elements*], { *dialog_navigation* }, [*layout_details*])]]

[[**rcv**(*ribbon_category_view_identifier*, *view_for_overview*, *views_for_categories*, [*visual_elements*],
{ *dialog_navigation* }, [*layout_details*])]] =

The Ribbon Category View *ribbon_category_view_identifier* provides the presentation of one ribbon category, consisting of an overview part and a set of sub categories. The overview part is given by a Basic Content Presenter Design containing a Ribbon Category Overview View, i.e.:

[[*view_for_overview*]]

The sub categories are given by [[*views_for_categories*]].

[*ribbon_category_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *ribbon_category_view_identifier* [[*layout_details*].]

ribbon_category_view_identifier is the source for [[*dialog_navigation*]].

[[*views_for_categories*]] =

[[*views_for_sub_category*]] | [[*view_for_sub_categories*]]

[[*views_for_sub_category*]] =

a number of Basic Content Presenter Designs each containing either a Ribbon Sub Category Single Entity View or a Ribbon Sub Category Subtyped Single Entity View, i.e.:

{ [[*basic_content_presenter_design_view*]]}

[[*view_for_sub_categories*]] =

a Basic Content Presenter Design containing either a Ribbon Sub Categories Categorized Single Entity View or a Ribbon Sub Categories Categorized Subtyped Single Entity View, i.e.:

[[*basic_content_presenter_design_view*]]

[[*ribbon_contents_view*]] =

[[**rctv**(*ribbon_contents_view_identifier*, *view_for_ribbon_categories*, *view_for_ribbon_buttons*,
[*visual_elements*], { *dialog_navigation* }, [*layout_details*])]]

[[**rctv**(*ribbon_contents_view_identifier*, *view_for_ribbon_categories*, *view_for_ribbon_buttons*,
[*visual_elements*], { *dialog_navigation* }, [*layout_details*])]] =

The Ribbon Contents View *ribbon_contents_view_identifier* puts together one interactor design containing a Ribbon Categories View with another interactor design containing a Ribbon Buttons View. The coupling is functional rather than visual, and provides the functionality navigating between single ribbon buttons and corresponding ribbon categories and up again. The ribbon categories part is provided by:

[[*view_for_ribbon_categories*]]

The ribbon buttons part is provided by:

[[*view_for_ribbon_buttons*]]

[*ribbon_contents_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *ribbon_contents_view_identifier* [[*layout_details*]].]

ribbon_contents_view_identifier is the source for {[*dialog_navigation*]}.

[[*ribbon_ticker_view*]] =

[[**rtv**(*ribbon_ticker_view_identifier*, {*view_for_ribbon_ticker_category*}, [*visual_elements*], {*dialog_navigation*}, [*layout_details*])]]

[[**rtv**(*ribbon_ticker_view_identifier*, {*view_for_ribbon_ticker_category*}, [*visual_elements*], {*dialog_navigation*}, [*layout_details*])]] =

The Ribbon Ticker View *ribbon_ticker_view_identifier* puts together a set of a set of interactor designs containing Ribbon Ticker Category Views, making up the ticker part of the top level of a ribbon. The ribbon ticker categories are provided by:

{[[*view_for_ribbon_ticker_category*]]}

[*ribbon_ticker_view_identifier* also includes [[*visual_elements*]].

[In the context of its parent view, *ribbon_ticker_view_identifier* [[*layout_details*]].]

ribbon_ticker_view_identifier is the source for {[*dialog_navigation*]}.

[[*ribbon_view*]] =

[[**rv**(*ribbon_view_identifier*, *view_for_ribbon_contents*, *view_for_ribbon_ticker*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*])]]

[[**rv**(*ribbon_view_identifier*, *view_for_ribbon_contents*, *view_for_ribbon_ticker*, [*visual_elements*], {*dialog_navigation*}, [*layout_details*])]] =

The Ribbon View *ribbon_view_identifier* puts together one interactor design containing a Ribbon Contents View with another interactor design containing a Ribbon Ticker View, together making up a complete ribbon. The ribbon contents part is provided by:

`[[view_for_ribbon_contents]]`

The ribbon ticker part is provided by:

`[[view_for_ribbon_ticker]]`

ribbon_view_identifier also includes `[[visual_elements]]`.

[In the context of its parent view, *ribbon_view_identifier* `[[layout_details]].`]

ribbon_view_identifier is the source for `{[[dialog_navigation]]}`.

4.2.3.6 Interactor Design View

`[[content_presenter_design_view]]` =

`[[cpdv(content_presenter_design_identifier, [layout_details])]]`

`[[cpdv(content_presenter_design_identifier, [layout_details])]]` =

The entire contents of the Content Presenter Design *content_presenter_design_identifier*.

[In the context of its parent view, *content_presenter_design_identifier* `[[layout_details]].`]

`[[task_supporter_design_view]]` =

`[[tsdv(task_supporter_design_identifier, [layout_details])]]`

`[[tsdv(task_supporter_design_identifier, [layout_details])]]` =

The entire contents of the Task Supporter Design *task_supporter_design_identifier*.

[In the context of its parent view, *task_supporter_design_identifier* `[[layout_details]].`]

`[[work_supporter_design_view]]` =

`[[wsdv(work_supporter_design_identifier, [layout_details])]]`

`[[wsdv(work_supporter_design_identifier, [layout_details])]]` =

The entire contents of the Work Supporter Design *work_supporter_design_identifier*.

[In the context of its parent view, *work_supporter_design_identifier* *layout_details*.]

With these production rules, only the identifiers (the names) of the interactor designs being referenced are included in the resulting sentences. To investigate the semantics of the corresponding interactor designs, their production rules must be used.

4.2.3.7 Dialog navigation

{*dialog_navigation*} =

dn(dialog_navigation_identifier, navigation_type, [navigation_source], [navigation_target])

dn(dialog_navigation_identifier, navigation_type, [navigation_source], [navigation_target]) =

a dialog navigation by which *navigation_source* the view *navigation_type* *navigation_target*

navigation_source =

the entity *entity_identifier* in | the attribute *attribute_identifier* from the entity *entity_identifier* in | the button *button_identifier* in

navigation_target =

view_identifier

navigation_type =

open | **show** | **hide** | **close** | **return**

open =

opens

show =

shows

hide =

hides

close =

closes

return =

navigates back to the dialog from which the view was opened

4.3 Basic Content Presenter Design

In this section, we provide the syntax and semantics of the basic variant of the Content Presenter Design construct.

4.3.1 Graphical Syntax

A Content Presenter Design identifies which parts of the connected concept model fragment of the corresponding Content Presenter in FLUIDE-A which should be included in the design. The concept model fragment in the design is thus a subset of the FLUIDE-A counterpart, and both entities and attributes may be omitted. Relations are omitted implicitly if at least one of the entities they connect is omitted. Annotations on model elements that are omitted are automatically omitted in the design. In addition, it is also possible to omit annotations on model elements that are not omitted. When model elements are omitted, the resulting concept model fragment in the design may seem not be connected. This is not a problem, as the original model fragment (which is connected) is used when the model extent is determined at run-time. This also means that even the anchor entity may be omitted in the design. The concrete syntax for expressing the model fragments is the same as in FLUIDE-A. In the graphical notation, the concept model is located in one or more Content Views. Relations going between entities presented in different views are not shown to avoid visual clutter. All the views (and their children) are located in the content part of a Basic Content Presenter Design. Figure 4.9 gives an example of a Basic Content Presenter design, with explanations of certain parts.

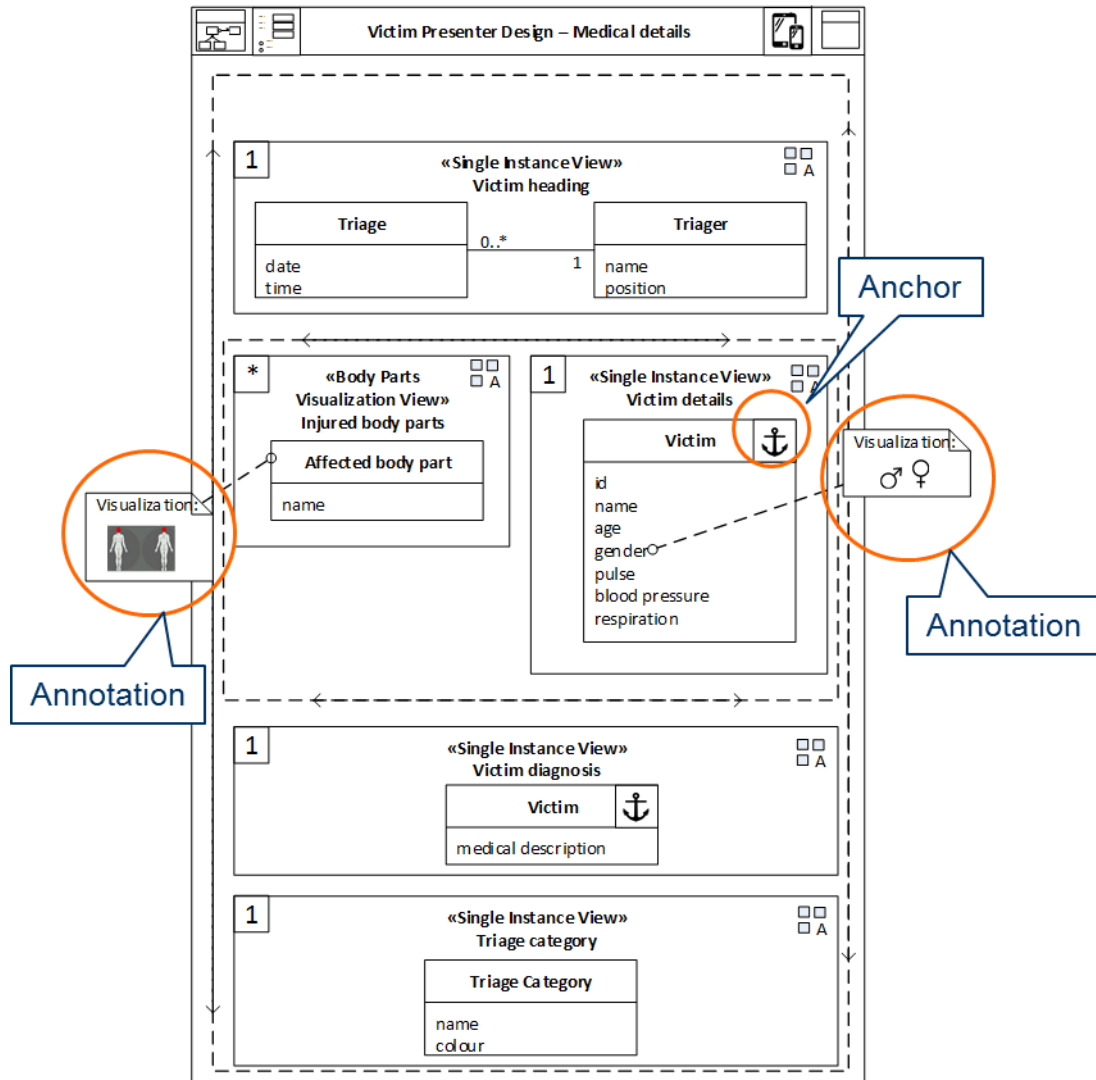


Figure 4.9 - A Basic Content Presenter Design in FLUIDE-D

The corresponding FLUIDE-A specification of the design shown in Figure 4.9 is the Basic Content Presenter shown in Figure 3.2. As can be seen in Figure 4.9, the concept model fragment may be distributed between different Content Views, which may have different cardinality. The cardinality in the views must correspond to the cardinality of the entities presented in the view, as seen from the anchor entity. For example the *Injured body parts* view in Figure 4.9 has many cardinality, because there is a relation between the *Victim* and *Affected body part* entities, having many cardinality on the *Affected body part* side. Note also that only the relevant annotations are included in the design.

4.3.2 Abstract Syntax

Figure 4.10 provides a concept model explaining the main concepts used when specifying a Basic Content Presenter Design.

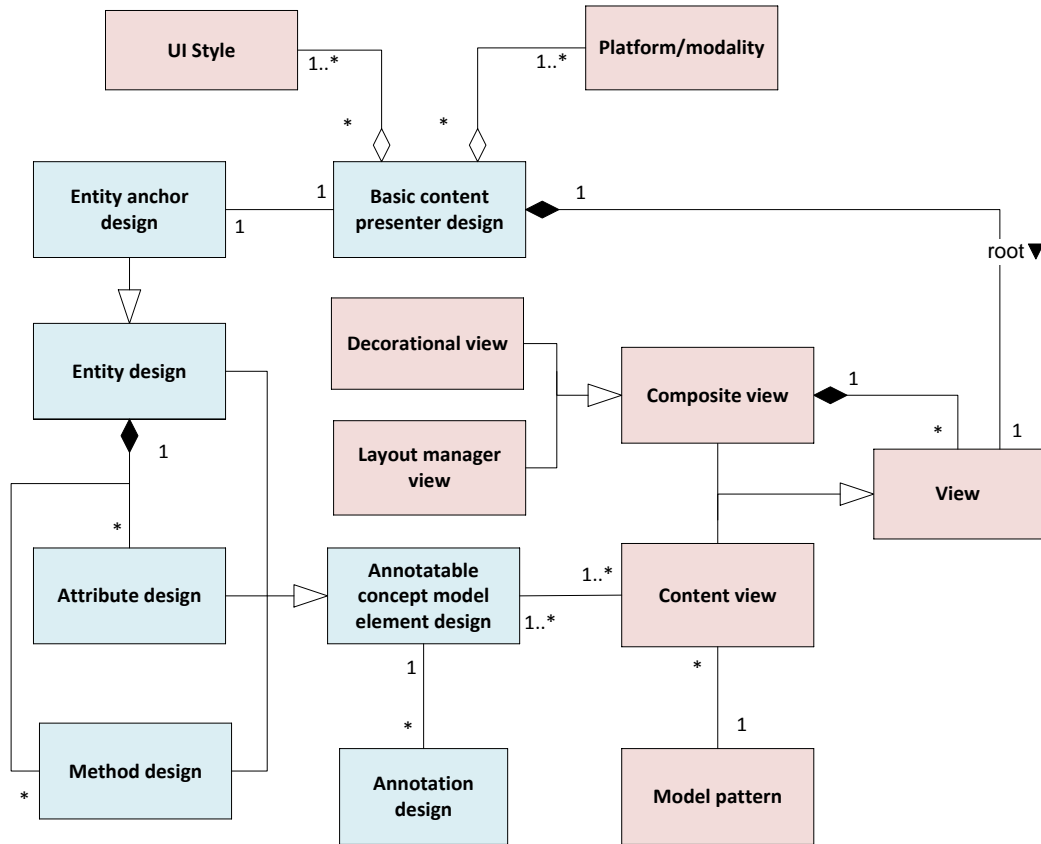


Figure 4.10 - Concept model describing the means for specifying Basic Content Presenter Designs in FLUIDE-D

The concept model in Figure 4.10 has two parts. The concepts drawn in light blue contain a structure mirroring the structure of the FLUIDE-A concept model for *Basic Content Presenter*, while the concepts drawn in light pink represents the means for specifying the design of each part of the mirrored structure, plus additional mechanisms for specifying decoration and structure that is needed in a design. The light pink part contains a subset of the corresponding concept model for views (see Figure 4.8).

When specifying a Basic Content Presenter Design using EBNF, most of the specification is expressed using the View constructs defined in Section 4.2.2. This means that only the root view is included in the EBNF definition of Basic Content Presenter Design. It also means that all concepts in the light blue part except Basic Content Presenter Design and Entity Anchor Design are specified as part of Content Views. Thus Content Views also specify which subset of the connected concept model from FLUIDE-A which is included in the design.

```

basic_content_presenter_design = bcpd(basic_content_presenter_design_identifier, {UI_style}-,
    {platform_modality}-, basic_content_presenter_identifier, anchor_entity_design, root_view);
  
```

The *basic_content_presenter_identifier* is a reference to the corresponding specification in FLUIDE-A.

UI_style = **forms based | list based | icons based | map based | graph based | multimedia based;**

platform_modality = **PC with mouse and keyboard | mobile device with touch | table top with touch | augmented reality | audio interaction;**

anchor_entity_design = *entity_identifier*;

root_view = *view*;

4.3.3 Semantics

[[*basic_content_presenter_design*]] =

[[**bcpd**(*basic_content_presenter_design_identifier*, {*UI_style*}-, {*platform_modality*}-, *basic_content_presenter_identifier*, *anchor_entity_design*, *root_view*)]] =

[[**bcpd**(*basic_content_presenter_design_identifier*, {*UI_style*}-, {*platform_modality*}-, *basic_content_presenter_identifier*, *anchor_entity_design*, *root_view*)]] =

basic_content_presenter_design_identifier expresses how the Basic Content Presenter *basic_content_presenter_identifier* should be rendered on { [[*platform_modality*]] } using { [[*UI_style*]] } as presentation styles. *basic_content_presenter_design_identifier* presents these parts:

[[*root_view*]]

The starting point for determining the instances to present in these parts is *anchor_entity_design*.

[[*platform_modality*]] =

[[**PC with mouse and keyboard**]] | [[**mobile device with touch**]] | [[**table top with touch**]]|
[[**augmented reality**]]| [[**audio interaction**]]

[[**PC with mouse and keyboard**]] =

a PC or similar using mouse and keyboard as interaction devices

[[**mobile device with touch**]] =

a mobile phone, tablet or similar mobile device using touch as interaction means

[[**table top with touch**]] =

a table top or similar device using touch as interaction means

[[**augmented reality**]] =

a mobile phone, tablet or similar mobile device using augmented reality techniques

[[**audio interaction**]] =

a platform supporting sound output, spoken commands and/or dictation as interaction means

[[*UI_style*]] =

[[forms based]] | [[list based]] | [[icons based]] | [[map based]] | [[graph based]] | [[multimedia based]]

[[forms based]] =

traditional forms-based presentation, typically exploiting one user interface control per attribute that is presented

[[list based]] =

list-based presentation, showing a number of instances in a list box or in a more complex list using separate user interface controls for each attribute that is presented

[[icons based]] =

icon-based presentation, showing instances as icons

[[map based]] =

map-based presentation, showing instances as overlays on a map background

[[graph based]] =

graph-based presentation, visualising attribute values using graphical techniques

[[multimedia based]] =

multi media-based presentation, presenting attribute values using rich media techniques like images, sound and video

4.3.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Basic Content Presenter Design in FLUIDE-D. The example is a subset of the specification of the Basic Content Presenter Design in Figure 4.9.

4.3.4.1 EBNF Specification

```
bcpd(Victim Presenter Design – Medical details,  
  forms based,  
  PC with mouse and keyboard,  
  Victim Presenter,  
  Victim,  
  /* root view is a layout manager:*/  
  lmv(triage layout,  
    vertical,  
    /* no visual elements*/,  
    /* no dialog navigation*/,  
    /* no layout details*/,
```

```

/* child views */
/* single instance view: */
siv(Victim heading,
    /* model pattern instance: */
    ese(
        /* main entity: */
        ed(Triage,
            /* included attributes: */
            ad(date, /* no annotations for the attribute design */),
            ad(time, /* no annotations for the attribute design */),
            /* no methods */,
            /* no annotations for the entity design */,
            ) /* end of main entity design specification */,
        /* no sub types */,
        /* one-related entity: */
        ed(Triager,
            /* included attributes: */
            ad(name, /* no annotations for the attribute design */),
            ad(position, /* no annotations for the attribute design */),
            /* no methods */,
            /* no annotations for the entity design */,
            ) /* end of one-related entity design specification */
        ) /* end of model pattern specification */,
        /* no visual elements*/,
        /* no dialog navigation*/,
        /* no layout details*/
    ) /* end single instance view specification */,
/* layout manager view: */
lmv(medical details layout,
    vertical,
    /* no visual elements*/,
    /* no dialog navigation*/,
    /* no layout details*/,
    /* child views (two content views) omitted */
    ) /* end of layout manager specification */
/* two single instance views omitted */
) /* end of layout manager specification */
) /* end of bcpd specification */

```

4.3.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 4.3.3 on the EBNF specification just presented results in the following English sentences:

Victim Presenter Design – Medical details expresses how the Basic Content Presenter **Victim Presenter** should be rendered on a mobile phone, tablet or similar mobile device using touch as interaction means using traditional forms-based presentation, typically exploiting one user interface control per attribute that is presented as presentation style. *Victim Presenter Design – Medical details* presents these parts:

The Layout Manager triage layout which is invisible and presents:

*The Single Instance View **Victim heading** which provides a forms-based presentation of one instance of date and time from Triage and name and position from Triager using separate user interface controls for each attribute.*

*The Layout Manager **medical details layout** which is invisible and presents:*

[two omitted Content Views]

This content is presented side by side horizontally.

[two omitted single instance views]

This content is presented over/under each other vertically.

4.4 Aggregated Content Presenter Design

In this section, we provide the syntax and semantics of the aggregated variant of the Content Presenter Design construct, i.e. Content Presenter Designs that have other Content Presenter Designs as children of their views.

4.4.1 Graphical Syntax

Aggregated Content Presenter Designs aggregate other Content Presenter Designs (Basic or Aggregated). The implicit concept model fragment of the design is determined in the same way as the connected concept model fragment of the corresponding presenter in FLUIDE-A. Model elements that are to be omitted in the design must be omitted in the child Content Presenter Designs. In addition, some child presenter designs (including the one containing the anchor) may be omitted. The relations in the corresponding Aggregated Content Presenter in FLUIDE-A may not be explicitly omitted, and is not shown in the design to avoid visual clutter. In the graphical notation, the child presenters are often located in Content Integration Views. One such view may contain more than one child presenter, sometimes in different compartments. All the views (and their children) are located in the content part of an Aggregated Content Presenter Design. Figure 4.11 gives an example of an Aggregated Content Presenter Design, with explanations of certain parts. As a shorthand notation in Figure 4.11, in subsequent figures providing examples for the other interactor design constructs, and in the text explaining the graphical notation, we refer to the references to member interactor design as *interactor designs*, even though they formally (and in the EBNF) are *interactor design views*.

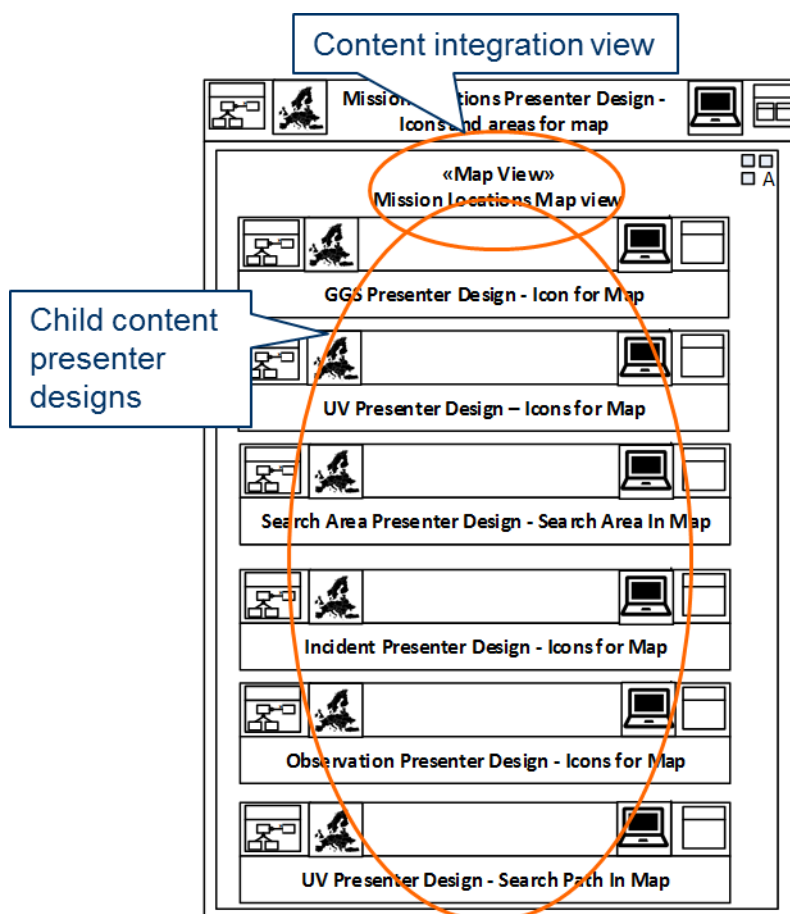


Figure 4.11 - An Aggregated Content Presenter Design in FLUIDE-D

The corresponding FLUIDE-A specification of the design shown in Figure 4.11 is the Aggregated Content Presenter shown in Figure 3.4. Only the border part of the child presenter designs is shown in the aggregated one. The names of the child presenter designs are shown in their content part. The same is the case for the anchor of the presenter design containing the anchor (this presenter design is omitted in the Aggregated Content Presenter Design in Figure 4.11). In this example, there are two different designs for *UV Presenter* in the FLUIDE-A specification. This is similar to locating attributes of the same entity in different Content Views in a Basic Content Presenter Design (as is done in the example in Figure 4.9). An aggregated presenter may contain more than one Content Integration View.

4.4.2 Abstract Syntax

Figure 4.12 provides a concept model explaining the main concepts used when specifying an Aggregated Content Presenter Design.

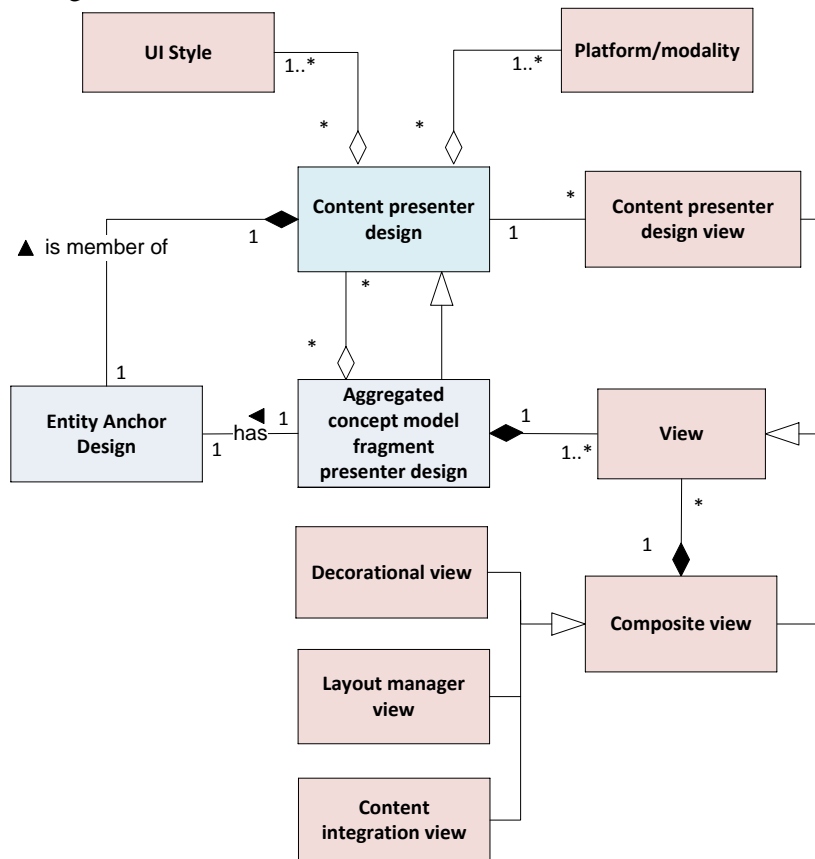


Figure 4.12 - Concept model describing the means for specifying Aggregated Content Presenter Designs in FLUIDE-D

The concept model in Figure 4.12 has two parts. The concepts drawn in light blue contain a structure mirroring part of the structure of the FLUIDE-A concept model for *Aggregated Content Presenter*, while concepts drawn in light pink represent the means for specifying the design of each part of the mirrored structure, plus additional mechanisms for specifying decoration and structure that is needed in a design. The light pink part contains a subset of the corresponding concept model for views (see Figure 4.8).

When specifying an Aggregated Content Presenter Design using EBNF, most of the specification is expressed using the View constructs defined in Section 4.2.2. This means that only the first level of the child views is included in the EBNF definition of Aggregated Content Presenter Design.

```
aggregated_content_presenter_design = acpd(aggregated_content_presenter_design_identifier, {UI_style}-
, {platform_modality}-, aggregated_content_presenter_identifier, entity_anchor_design,
{child_view}-);
```

UI_style = **forms based** | **list based** | **icons based** | **map based** | **graph based** | **multimedia based**;

platform_modality = **PC with mouse and keyboard** | **mobile device with touch** | **table top with touch** | **augmented reality** | **audio interaction**;

entity_anchor_design = *entity_identifier*;

child_view = *view*;

4.4.3 Semantics

[[*aggregated_content_presenter_design*]] =

[[**acpd**(*aggregated_content_presenter_design_identifier*, {*UI_style*}-, {*platform_modality*}-, *aggregated_content_presenter_identifier*, *entity_anchor_design*, {*child_view*-})]]

[[**acpd**(*aggregated_content_presenter_design_identifier*, {*UI_style*}-, {*platform_modality*}-, *aggregated_content_presenter_identifier*, *entity_anchor_design*, {*child_view*-})]] =

aggregated_content_presenter_design_identifier expresses how the Aggregated Content Presenter *aggregated_content_presenter_identifier* should be rendered on {[[*platform_modality*]]} using {[[*UI_style*]]} as presentation styles. *aggregated_content_presenter_design_identifier* presents these parts:

{[[*child_view*]]}

The starting point for determining the instances to present in these parts is *entity_anchor_design*.

4.4.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Aggregated Content Presenter Design in FLUIDE-D. The example is a subset of the specification of the Aggregated Content Presenter Design in Figure 4.11.

4.4.4.1 EBNF Specification

```
acpd (Mission Locations Presenter Design – Icons and areas for map,
map based,
PC with mouse and keyboard,
Mission Locations Presenter,
Mission,
/* One child view (a Map View):*/
mv(Mission Locations Map view,
/* pointers to six child views (Content Presenter Design Views) */
cpdv(GGS Presenter Design – Icons for Map, /* no layout details*/),
cpdv(UV Presenter Design – Icons for Map, /* no layout details*/),
cpdv(Search Area Presenter Design – Search Area In Map, /* no layout details*/),
```

```
cpdv(Incident Presenter Design – Icons for Map, /* no layout details*/),
cpdv(Observation Presenter Design – Icons for Map, /* no layout details*/),
cpdv(UV Presenter Design –Search Path In Map, /* no layout details*/),
/* no visual elements*/,
/* no dialog navigation*/,
/* no layout details*/
) /* end of Map View specification */
) /* end of acpd specification */
```

4.4.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 4.4.3 on the EBNF specification just presented results in the following English sentences:

Mission Locations Presenter Design – Icons and areas for map expresses how the Aggregated Content Presenter *Mission Locations Presenter* should be rendered on a PC or similar using mouse and keyboard as interaction devices using map-based presentation, showing instances as overlays on a map background as presentation style. *Mission Locations Presenter Design – Icons and areas for map* presents these parts:

The Map View Mission Locations Map view provides a map for presenting various map overlays. The overlays are provided by the child interactor designs, which must contain either another Map View, a Map Icons View, a Map Icons with Details Dialog View, a Map Outline View, or a Map Multi Line View. Such views are provided by:

The entire contents of the GGS Presenter Design – Icons for Map.

The entire contents of the UV Presenter Design – Icons for Map.

The entire contents of the Search Area Presenter Design – Search Area In Map.

The entire contents of the Incident Presenter Design – Icons for Map.

The entire contents of the Observation Presenter Design – Icons for Map.

The entire contents of the UV Presenter Design –Search Path In Map.

The overlays are shown on the map provided by the Map View highest up in the hierarchy.

The starting point for determining the instances to present in these parts is Mission.

4.5 Task Supporter Design

In this section, we provide the syntax and semantics of the Task Supporter Design construct. Task Supporter Designs are only provided in a basic variant.

4.5.1 Graphical Syntax

As the relations in an Aggregated Content Presenter Design is not shown, the only difference in the graphical notation of the content part of a Task Supporter Design and an Aggregated Content Presenter Design is that there may not be any anchor in a Task Supporter Design. Semantically, the difference is the same as for the corresponding FLUIDE-A interactors, i.e. that the content of all children of an Aggregated Content Presenter Design is determined together based on the implicit concept model fragment and the anchor, while the content of each child of a Task Supporter Design is determined independently of each other based on their separate concept model fragments and anchors. Figure 4.13 gives an example of a Task Supporter Design, with explanations of certain parts.

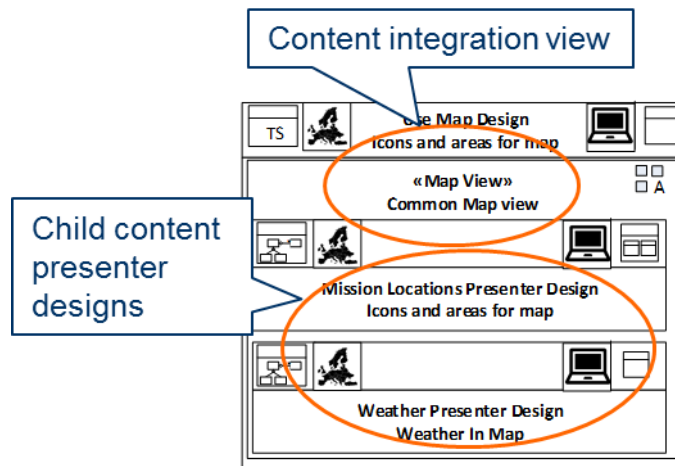


Figure 4.13 - A Task Supporter Design in FLUIDE-D

The corresponding FLUIDE-A specification of the design shown in Figure 4.13 is the Task Supporter shown in Figure 3.6. Only the border part of the child presenter designs is shown in the Task Supporter Design. The names of the child presenter designs are shown in their content part. In the example in Figure 4.13, one of the children is a Basic and the other is an Aggregated Content Presenter Design.

4.5.2 Abstract Syntax

Figure 4.14 provides a concept model explaining the main concepts used when specifying a Task Supporter Design.

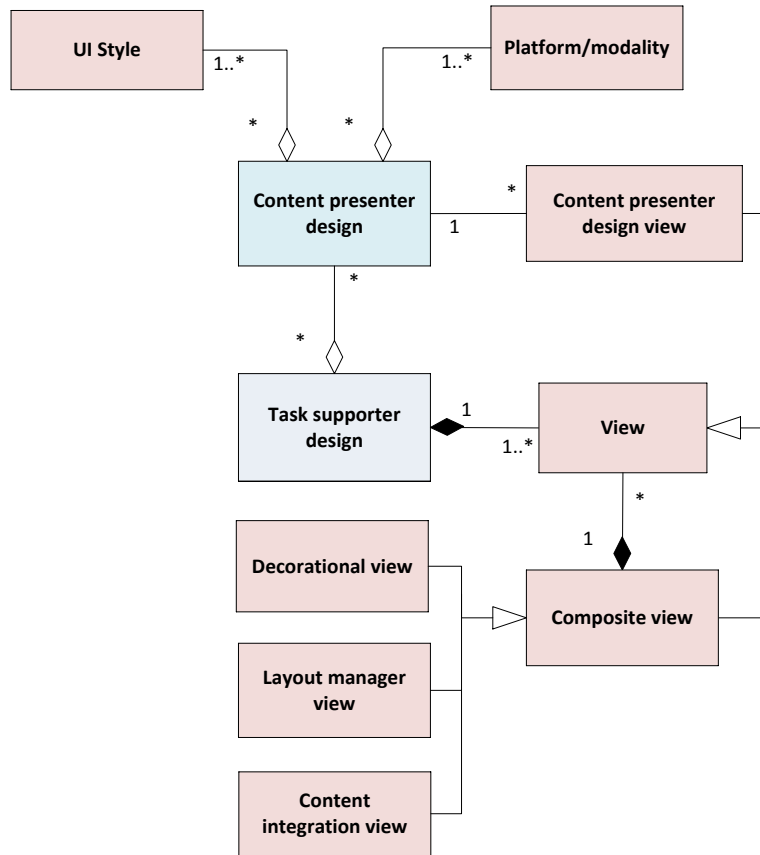


Figure 4.14 - Concept model describing the means for representing Task Supporter Designs in FLUIDE-D

The concept model in Figure 4.14 has two parts. The concepts drawn in light blue contain a structure mirroring part of the structure of the FLUIDE-A concept model for *Task Supporter* while the concepts drawn in light pink represent the means for specifying the design of each part of the mirrored structure, plus additional mechanisms for specifying decoration and structure that is needed in a design. The light pink part contains a subset of the corresponding concept model for views (see Figure 4.8).

When specifying a Task Supporters using EBNF, most of the specification is expressed using the View constructs defined in Section 4.2.2. This means that only the first level of the child views is included in the EBNF definition of Task Support Design.

```

task_supporter_design = tsd(task_supporter_design_identifier, {UI_style}-, {platform_modality}-,
    task_supporter_identifier, {child_view}-);
  
```

UI_style = forms based | list based | icons based | map based | graph based | multimedia based;

platform_modality = PC with mouse and keyboard | mobile device with touch | table top with touch | augmented reality | audio interaction;

child_view = view;

4.5.3 Semantics

$\llbracket \text{task_supporter_design} \rrbracket =$

```
[[ tsd(task_supporter_design_identifier, {UI_style}-, {platform_modality}-,
task_supporter_identifier, {child_view}-) ]]
```

```
[[ tsd(task_supporter_design_identifier, {UI_style}-, {platform_modality}-, task_supporter_identifier,
{child_view}-) ]] =
```

task_supporter_design_identifier expresses how the Task Supporter *task_supporter_identifier* should be rendered on {*platform_modality*} using {*UI_style*} as presentation styles. *task_supporter_design_identifier* presents these parts:

```
{child_view}
```

The instances to present are determined separately for each part by the Content Presenter Designs used in the parts.

4.5.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Task Supporter Design in FLUIDE-D. The example is the specification of the Task Supporter Design in Figure 4.13.

4.5.4.1 EBNF Specification

```
tsd(Use Map Design – Icons and Areas for Map,
map based,
PC with mouse and keyboard,
Use Map,
/* One child view (a Map View):*/
mv(Common Map view,
/* pointers to two child views (Content Presenter Design Views) */
cpdv(Mission Locations Presenter Design – Icons and areas for map,
/* no layout details*/),
cpdv(Weather Presenter Design –Weather in Map, /* no layout details*/),
/* no visual elements*/,
/* no dialog navigation*/,
/* no layout details*/
) /* end of Map View specification */
) /* end of tsd specification */
```

4.5.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 4.5.3 on the EBNF specification just presented results in the following English sentences:

Use Map Design – Icons and Areas for Map expresses how the Task Supporter *Use Map* should be rendered on a PC or similar using mouse and keyboard as interaction devices using map-based presentation, showing instances as overlays on a map background as presentation style. *Use Map Design – Icons and Areas for Map* presents these parts:

*The Map View **Common Map view** provides a map for presenting various map overlays. The overlays are provided by the child interactor designs, which must contain either another Map View, a Map Icons*

View, a Map Icons with Details Dialog View, a Map Outline View, or a Map Multi Line View. Such views are provided by:

*The entire contents of the **Mission Locations Presenter Design – Icons and areas for map.***

*The entire contents of the **Weather Presenter Design –Weather in Map.***

The overlays are shown on the map provided by the Map View highest up in the hierarchy.

The instances to present are determined separately for each part by the Content Presenter Designs used in the parts.

4.6 Basic Work Supporter Design

In this section, we provide the syntax and semantics of the basic variant of the Work Supporter Design construct.

4.6.1 Graphical Syntax

In a Basic Work Supporter in FLUIDE-A, the tasks in the task model may or may not have a connected Task Supporter. In the corresponding design in FLUIDE-D, only the designs for tasks with Task Supporters are relevant to include. Thus, the tasks are not shown in the design, only the Task Supporter Designs chosen to be included are shown. Naturally, the operators are not shown in the design, neither. There are three types of choices to make when specifying a Basic Work Supporter Design. The first is which Task Supporters from the corresponding Work Supporter to include designs for. The second is which Task Supporter Designs to use for these (if more than one is available). The third is which view(s) to use for wrapping the Task Supporter Designs. The graphical notation of the content part of a Basic Work Supporter Design is similar to the one used in Task Supporter Designs and Aggregated Content Presenter Designs. The main difference is that the children of a Basic Work Supporter Design must be Task Supporter Designs. It is also allowed to include only parts of child Task Supporter Design. In these cases, these Task Supporter Designs are represented by some of their Content Presenter Design children. Figure 4.15 gives an example of a Basic Work Supporter Design, with explanations of certain parts.

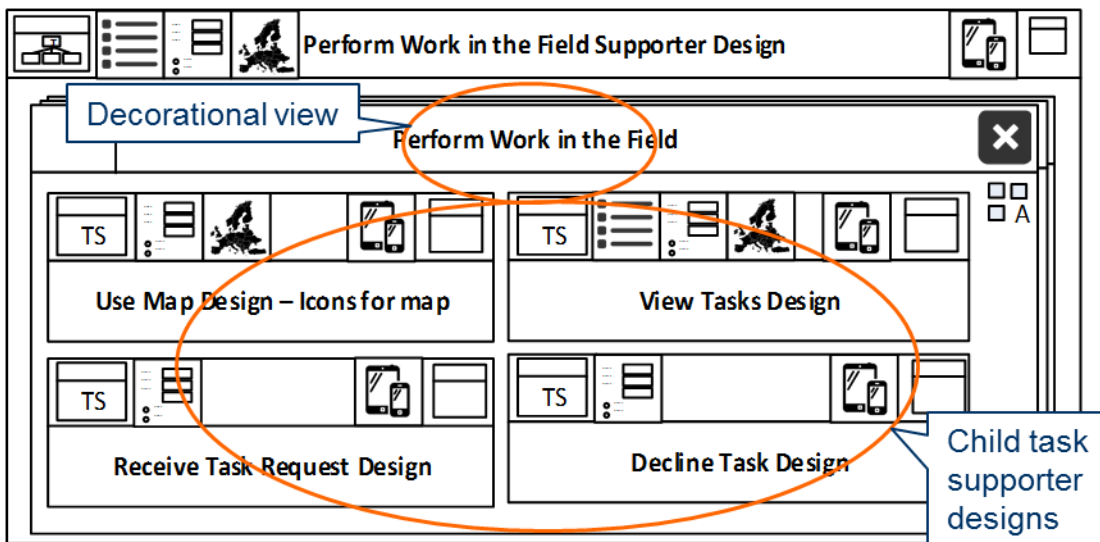


Figure 4.15 - A Basic Work Supporter Design in FLUIDE-D

The corresponding FLUIDE-A specification of the design shown in Figure 4.15 is the Basic Work Supporter shown in Figure 3.8. Only the border part of the Task Supporter Designs is shown in the Work Supporter Design, and their names are shown in their content part.

4.6.2 Abstract Syntax

Figure 4.16 provides a concept model explaining the main concepts used when specifying a Basic Work Supporter Design.

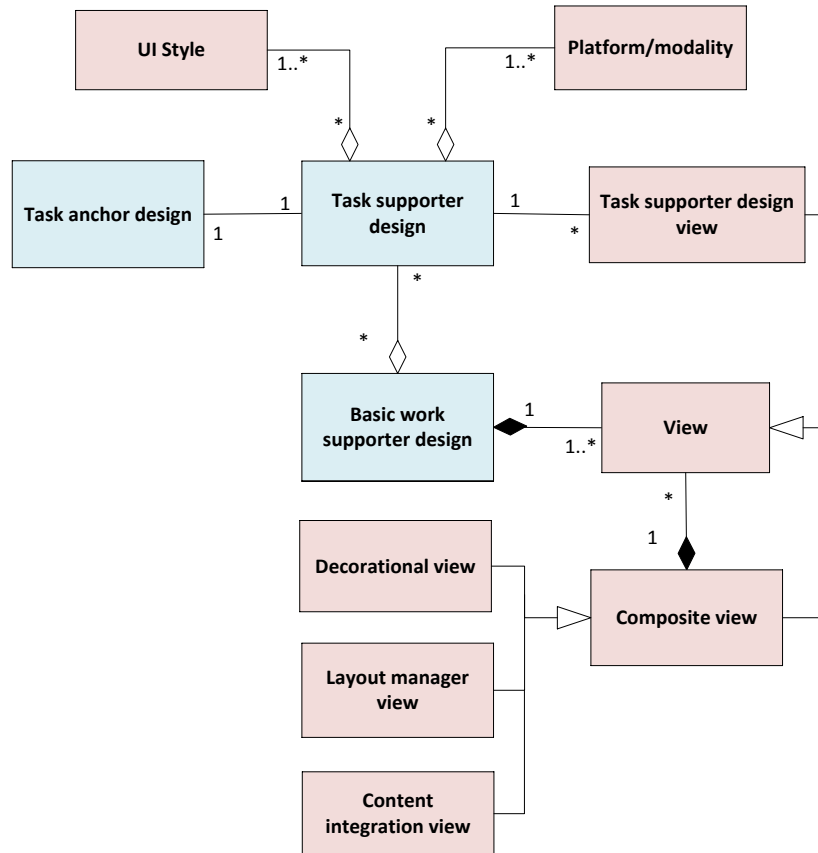


Figure 4.16 - Concept model describing the means for specifying Basic Work Supporter Designs in FLUIDE-D

The concept model in Figure 4.16 has two parts. The concepts drawn in light blue contain a structure mirroring part of the structure of the FLUIDE-A concept model for *Basic Work Supporter* while the concepts drawn in light pink represent the means for specifying the design of each part of the mirrored structure, plus additional mechanisms for specifying decoration and structure that is needed in a design. The light pink part contains a subset of the corresponding concept model for views (see Figure 4.8).

When specifying a Basic Work Supporter Design using EBNF, most of the specification is expressed using the View constructs defined in Section 4.2.2. This means that only the first level of the child views is included in the EBNF definition of Basic Work Support Design. To simplify the concept model, only Task Supporter Design View is included as an Interactor Design View sub type that may be used, even though also Concept Presenter Design Views may be used as representatives for Task Supporter Design Views. As the task model is not shown in the graphical syntax of Basic Work Supporter Designs, it is not included in the EBNF definition neither, except for the anchor task.

```
basic_work_supporter_design = bwsd(basic_work_supporter_design_identifier, {UI_style}-,
    {platform_modality}-, basic_work_supporter_idenfifier, anchor_task_design, {child_view}-);
```

UI_style = **forms based** | **list based** | **icons based** | **map based** | **graph based** | **multimedia based**;

platform_modality = **PC with mouse and keyboard** | **mobile device with touch** | **table top with touch** | **augmented reality** | **audio interaction**;

```
anchor_task_design = task_identifier;
```

```
child_view = view;
```

4.6.3 Semantics

```
[[ basic_work_supporter_design ]] =
```

```
[[ bwsd(basic_work_supporter_design_identifier, {UI_style}-, {platform_modality}-,
basic_work_supporter_identifier, anchor_task_design, {child_view}-) ]]
```

```
[[ bwsd(basic_work_supporter_design_identifier, {UI_style}-, {platform_modality}-,
basic_work_supporter_identifier, anchor_task_design, {child_view}-) ]]
```

basic_work_supporter_design_identifier expresses how the Basic Work Supporter *basic_work_supporter_identifier* (supporting a task model having *anchor_task_design* as root) should be rendered on `{platform_modality}` using `{UI_style}` as presentation styles. *basic_work_supporter_design_identifier* presents these parts:

```
{[[ child_view ]]}
```

The instances to present are determined separately for each part by the Task Supporter Designs and Content Presenter Designs used in the parts.

4.6.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Basic Work Supporter Design in FLUIDE-D. The example is the specification of the Basic Work Supporter Design in Figure 4.15.

4.6.4.1 EBNF Specification

```
bwsd(Perform Work in the Field Supporter Design,
list based, forms based, map based,
mobile device with touch,
Perform Work in the Field Supporter,
Perform Work in the Field,
/* One child view (a Decorational View):*
dv(Windows supporting Perform Work in the Field,
loosely connected windows,
/* no date */,
/* no time */,
/* visual elements: */
ves(/* no image */,
/* heading text: */
"Perform Work in the Field",
/* no border colour */,
/* no graphics */,
/* no buttons */,
), /* end of visual elements */
/* no dialog navigation*/,
/* layout method: */
```

```
    automatic,  
    /* no layout details*/,  
    /* pointers to four child views (all Task Supporter Design View):*/  
    tsdv(Use Map Design – Icons for map, /* no layout details*/),  
    tsdv(Receive Task Request Design, /* no layout details*/),  
    tsdv(View Task Design, /* no layout details*/),  
    tsdv(Decline Task Design, /* no layout details*/),  
    ) /* end of Decorational View specification */  
 ) /* end of bwsd specification */
```

4.6.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 4.6.3 on the EBNF specification just presented results in the following English sentences:

***Perform Work in the Field Supporter Design** expresses how the Basic Work Supporter **Perform Work in the Field Supporter** (supporting a task model having Perform Work in the Field as root) should be rendered on a mobile phone, tablet or similar mobile device using touch as interaction means using list-based presentation, showing a number of instances in a list box or in a more complex list using separate user interface controls for each attribute that is presented, traditional forms-based presentation, typically exploiting one user interface control per attribute that is presented, as well as map-based presentation, showing instances as overlays on a map background as presentation styles. **Perform Work in the Field Supporter Design** presents these parts:*

*The Decorational View **Windows supporting Perform Work in the Field** which represents a set of loosely connected windows or a number of full screen renderings in which a following content are presented:*

The text "Perform Work in the Field" is used as heading or label.

*The entire contents of the **Use Map Design – Icons for map**.*

*The entire contents of the **Receive Task Request Design**.*

*The entire contents of the **View Task Design**.*

*The entire contents of the **Decline Task Design**.*

*The layout of this content is determined by a layout algorithm provided by **Windows supporting Perform Work in the Field**.*

The instances to present are determined separately for each part by the Task Supporter Designs used in the parts.

4.7 Aggregated Work Supporter Design

In this section, we provide the syntax and semantics of the aggregated variant of the Work Supporter Design construct, i.e. Work Supporter Designs that have other Work Supporter Designs as children of their views.

4.7.1 Graphical Syntax

The graphical notation of the content part of an Aggregated Work Supporter Design is similar to the one used in Basic Work Supporter Designs, Task Supporter Designs and Aggregated Content Presenter Designs. As for the Basic Work Supporter Designs, the operators are not shown. Figure 4.17 gives an example of an Aggregated Work Supporter Design, with explanations of certain parts.

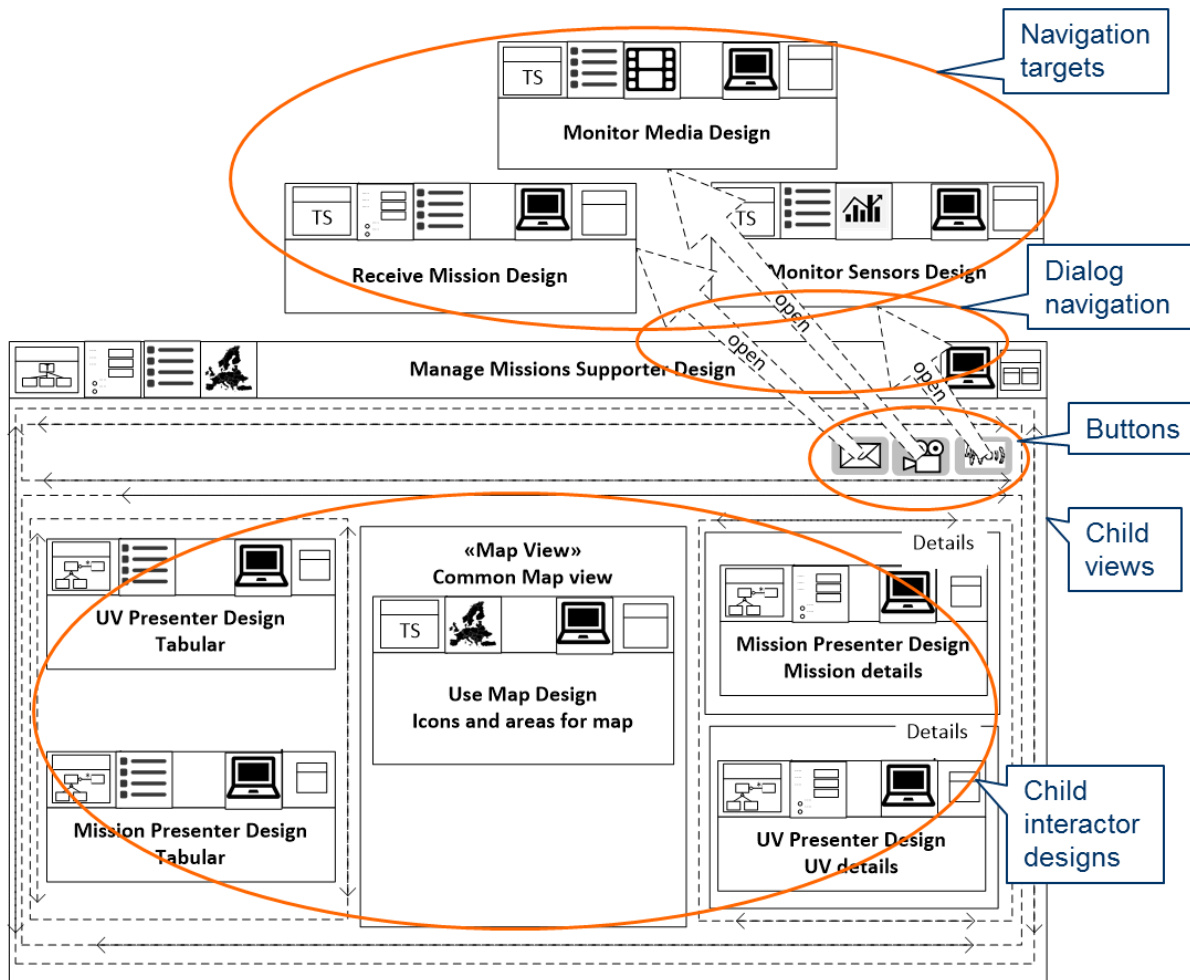


Figure 4.17 - An Aggregated Work Supporter in FLUIDE-D

The corresponding FLUIDE-A specification of the design shown in Figure 4.17 is the Aggregated Work Supporter shown in Figure 3.10. Only the border part of the child supporter designs is shown in the aggregated one. The names of the child supporter designs are shown in their content part. The specification in Figure 4.17 also illustrates the use of buttons and dialog navigation.

4.7.2 Abstract Syntax

Figure 4.18 provides a concept model explaining the main concepts used when specifying an Aggregated Work Supporter Design.

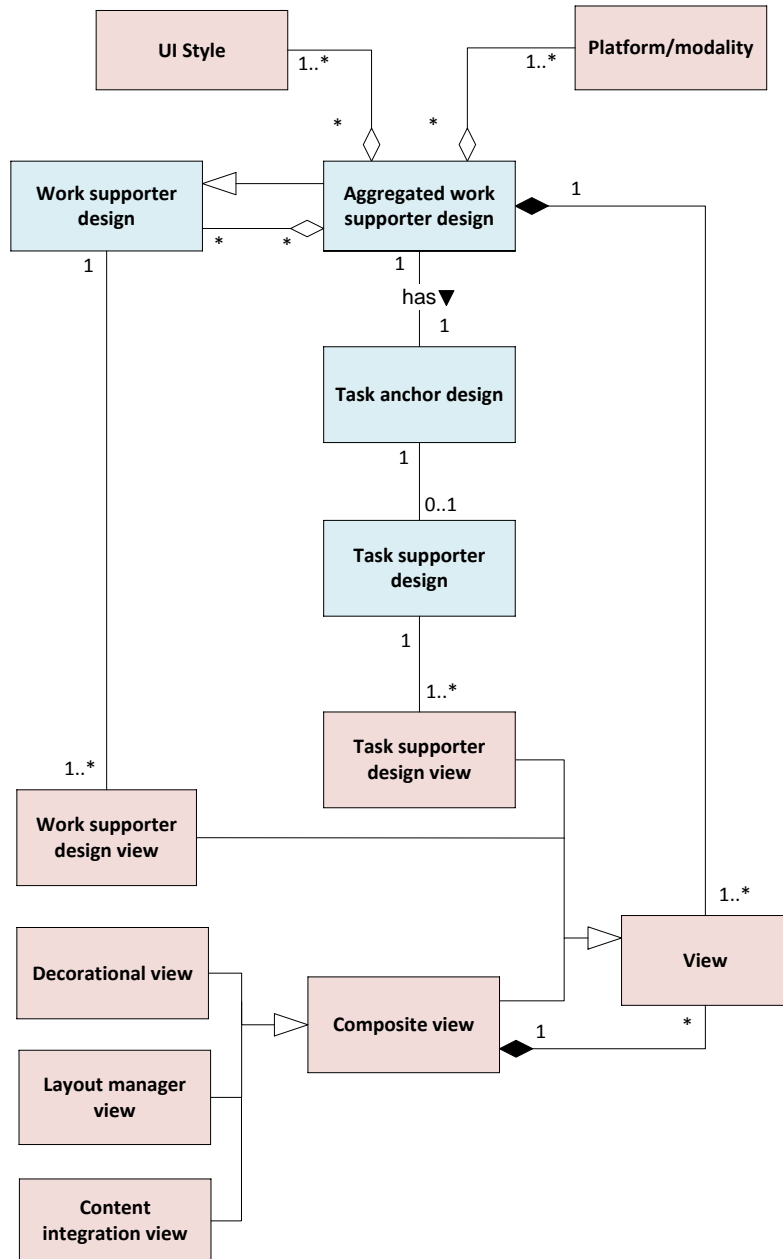


Figure 4.18 - Concept model describing the means for specifying Aggregated Work Supporter Designs in FLUIDE-D

The concept model in Figure 4.18 has two parts. The concepts drawn in light blue contain a structure mirroring part of the structure of the FLUIDE-A concept model for *Aggregated Work Supporter* while the concepts drawn in light pink represent the means for specifying the design of each part of the mirrored structure, plus additional mechanisms for specifying decoration and structure that is needed in a design. The light pink part contains a subset of the corresponding concept model for views (see Figure 4.8).

When specifying an Aggregated Work Supporter Design using EBNF, most of the specification is expressed using the View constructs defined in Section 4.2.2. This means that only the first level of the child views is included in the EBNF definition of Aggregated Work Support Design. To simplify the concept model, only Task Supporter Design View and Work Supporter Design View are included as the Interactor Design View

sub types that may be used, even though also Concept Presenter Design Views may be used as representatives for Work or Task Supporter Design Views.

```
aggregated_work_supporter_design = awsd(aggregated_work_supporter_design_identifier, {UI_style}-,
    {platform_modality}-, aggregated_work_supporter_identifier, anchor_task_design, {child_view}-);
```

UI_style = **forms based** | **list based** | **icons based** | **map based** | **graph based** | **multimedia based**;

platform_modality = **PC with mouse and keyboard** | **mobile device with touch** | **table top with touch** | **augmented reality** | **audio interaction**;

anchor_task_design = *task_identifier*;

child_view = *view*;

4.7.3 Semantics

$\llbracket aggregated_work_supporter_design \rrbracket =$

$\llbracket awsd(aggregated_work_supporter_design_identifier, \{UI_style\}-, \{platform_modality\}-, aggregated_work_supporter_identifier, anchor_task_design, \{child_view\}-) \rrbracket$

$\llbracket awsd(aggregated_work_supporter_design_identifier, \{UI_style\}-, \{platform_modality\}-, aggregated_work_supporter_identifier, anchor_task_design, \{child_view\}-) \rrbracket =$

aggregated_work_supporter_design_identifier expresses how the Aggregated Work Supporter *aggregated_work_supporter_identifier* (supporting a task model having *anchor_task_design* as root) should be rendered on $\llbracket platform_modality \rrbracket$ using $\llbracket UI_style \rrbracket$ as presentation styles. *aggregated_work_supporter_design_identifier* presents these parts:

$\{\llbracket child_view \rrbracket\}$

The instances to present are determined separately for each part by the Work Supporter Designs, Task Supporter Designs and Content Presenter Designs used in the parts.

4.7.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Aggregated Work Supporter Design in FLUIDE-D. The example is a subset of the specification of the Aggregated Work Supporter Design in Figure 4.17.

4.7.4.1 EBNF Specification

```
awsd(Manage Missions Supporter Design,
    forms based, list based, map based,
    PC with mouse and keyboard,
    Manage Missions Supporter,
    Manage Missions,
    /* One child view (a Layout Manager View):*/
    lmv(manage missions overall layout,
        vertical,
```



```

/* no visual elements*/,
/* no dialog navigation*/,
/* no layout details*/,
/* two child views (Layout Manager Views) */
lmv(buttons layout,
    horizontal,
    /* three buttons as visual elements*/
    ves(
        /* no image, text, border colour or graphics*/,,,,
        /* the buttons*/
        btn(message button, letterImage, /* no button text*/),
        btn(media button, cameraImage, /* no button text*/),
        btn(sensor button, sensorImage, /* no button text*/)
    ), /* end of visual elements */
    /* dialog navigation for the buttons: */
    dn(open message reader, open, message button, Receive Mission Design),
    dn(open media viewer, open, media button, Monitor Media Design),
    dn(open sensor viewer, open, sensor button, Monitor Sensor Design)
    /* no layout details*/,
    /* no child views */
), /* end of second layout manager specification */
lmv(content and content integration views layout,
    horizontal,
    /* no visual elements*/,
    /* no dialog navigation*/,
    /* no layout details*/,
    /* three child views (two layout managers with interactor design view children
    and one Map View) omitted */
    ) /* end of third layout manager specification */
) /* end of first layout manager specification */
) /* end of awsd specification */

```

4.7.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 4.7.3 on the EBNF specification just presented results in the following English sentences:

Manage Missions Supporter Design expresses how the Aggregated Work Supporter *Manage Missions Supporter* (supporting a task model having Manage Missions as root) should be rendered on a PC or similar using mouse and keyboard as interaction devices using traditional forms-based presentation, typically exploiting one user interface control per attribute that is presented, list-based presentation, showing a number of instances in a list box or in a more complex list using separate user interface controls for each attribute that is presented, as well as map-based presentation, showing instances as overlays on a map background as presentation styles. *Manage Missions Supporter Design* presents these parts:

The Layout Manager manage missions overall layout which is invisible and presents:

The Layout Manager buttons layout which is invisible and presents:

The button message button with the image letterImage.

The button media button with the image cameraImage.

The button sensor button with the image sensorImage.

This content is presented side by side horizontally.

***Manage missions overall layout** is the source for a dialog navigation by which the button message button in the view opens Receive Mission Design, a dialog navigation by which the button media button in the view opens Monitor Media Design, and a dialog navigation by which the button sensor button in the view opens Monitor Sensor Design.*

*The Layout Manager **content and content integration views layout** which is invisible and presents:*

*The Layout Manager **buttons layout** which is invisible and presents:*

[two omitted Layout Manager Views, each with two Interactor Design View children, as well as one omitted Map View]

This content is presented side by side horizontally.

This content is presented over/under each other vertically.

The instances to present are determined separately for each part by the Work Supporter Designs, Task Supporter Designs and Content Presenter Designs used in the parts.

4.8 Category Manager Design

In this section, we provide the syntax and semantics of the Category Manager Design construct. Category Manager Designs are only provided in a basic variant.

4.8.1 Graphical Syntax

The graphical notation of the content part of a Category Manager Design is similar to the one used in Aggregated Work Supporter Designs. The main difference is that the children of an Aggregated Work Supporter Design may be both Work Supporter Designs (basic or aggregated) and Content Presenter Designs (basic or aggregated). Semantically, the difference is the same as for the corresponding FLUIDE-A interactors, among other that the implicit task model provides a connection between all children of an Aggregated Work Supporter Design, while each child of a Category Manager Design is independent of each other. Figure 4.19 gives an example of a Category Manager Design, with explanations of certain parts.

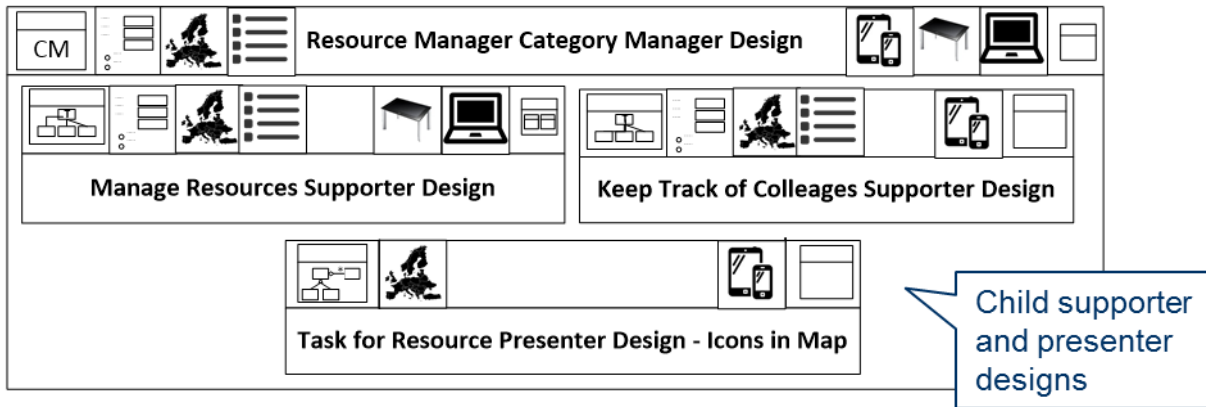


Figure 4.19 - A Category Manager Design in FLUIDE-D

The corresponding FLUIDE-A specification of the design shown in Figure 4.19 is the Category Manager shown in Figure 3.12. Only the border part of the child supporter/presenter designs is shown in the Category Manager Design. The names of the child designs are shown in their content part. In the example in Figure 4.19, the children are a Basic Work Supporter Design, an Aggregated Work Supporter Design, as well as a Basic Content Presenter Design.

4.8.2 Abstract Syntax

Figure 4.20 provides a concept model explaining the main concepts used when specifying a Category Manager Design.

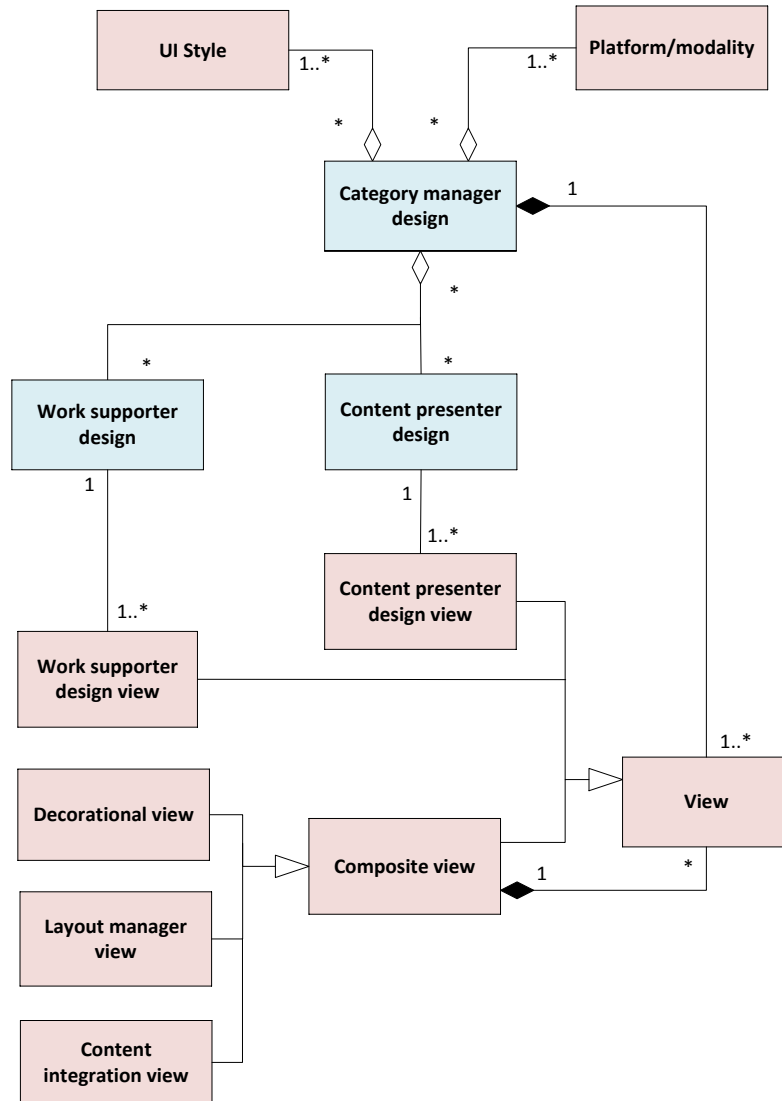


Figure 4.20 - Concept model describing the means for specifying Category Manager Designs in FLUIDE-D

The concept model in Figure 4.20 has two parts. The concepts drawn in light blue contain a structure mirroring part of the structure of the FLUIDE-A concept model for *Category Manager* while the concepts drawn in light pink represent the means for specifying the design of each part of the mirrored structure, plus additional mechanisms for specifying decoration and structure that is needed in a design. The light pink part contains a subset of the corresponding concept model for views (see Figure 4.8).

When specifying a Category Manager Design using EBNF, most of the specification is expressed using the View constructs defined in Section 4.2.2. This means that only the first level of the child views is included in the EBNF definition of Category Manager Design. To simplify the concept model, only Content Presenter Design View and Work Supporter Design View are included as the Interactor Design View sub types that may be used, even though also Task Supporter Design Views may be used as representatives for Work Supporter Design Views.

```
category_manager_design = cmd(category_manager_design_identifier, {UI_style}-, {platform_modality}-,
category_manager_identifier, {child_view}-);
```

UI_style = **forms based** | **list based** | **icons based** | **map based** | **graph based** | **multimedia based**;

platform_modality = **PC with mouse and keyboard** | **mobile device with touch** | **table top with touch** | **augmented reality** | **audio interaction**;

child_view = *view*;

4.8.3 Semantics

$\llbracket \text{category_manager_design} \rrbracket =$

$\llbracket \text{cmd}(\text{category_manager_design_identifier}, \{UI_style\}\text{-}, \{platform_modality\}\text{-}, \text{category_manager_identifier}, \{child_view\}\text{-}) \rrbracket$

$\llbracket \text{cmd}(\text{category_manager_design_identifier}, \{UI_style\}\text{-}, \{platform_modality\}\text{-}, \text{category_manager_identifier}, \{child_view\}\text{-}) \rrbracket =$

category_manager_design_identifier expresses how the Category Manager *category_manager_identifier* should be rendered on $\llbracket platform_modality \rrbracket$ using $\llbracket UI_style \rrbracket$ as presentation styles. *category_manager_design_identifier* presents these parts:

$\{\llbracket child_view \rrbracket\}$

The instances to present are determined separately for each part by the Work Supporter Designs, Task Supporter Designs and Content Presenter Designs used in the parts.

4.8.4 Example

In this section, we provide an example of using the abstract syntax (EBNF definitions) and the production rules defining the semantics for Category Manager Design in FLUIDE-D. The example is the specification of the Category Manager Design in Figure 4.19.

4.8.4.1 EBNF Specification

```
cmd(Resource Manager Category Manager Design,
  forms based, map based, list based,
  PC with mouse and keyboard, mobile device with touch, table top with touch,
  Resource Manager Category Manager,
  /* Three interactor design view (of different type) child views:*/
  wsdv(Manager Resources Supporter Design, /* no layout details*/),
  wsdv(Keep Track of Colleagues Supporter Design, /* no layout details*/),
  cpdv(Task for Resources Design – Icons in Map, /* no layout details*/)
) /* end of cmd specification */
```

4.8.4.2 Semantics of the EBNF Specification

Applying the production rules from Section 4.8.3 on the EBNF specification just presented results in the following English sentences:

Resource Manager Category Manager Design expresses how the Category Manager **Resource Manager Category Manager** should be rendered on a PC or similar using mouse and keyboard as interaction devices, a mobile phone, tablet or similar mobile device using touch as interaction means, or a table top or similar device using touch as interaction means using traditional forms-based presentation, typically exploiting one user interface control per attribute that is presented, map-based presentation, showing instances as overlays on a map background, as well as list-based presentation, showing a number of instances in a list box or in a more complex list using separate user interface controls for each attribute that is presented as presentation styles. **Resource Manager Category Manager Design** presents these parts:

*The entire contents of the Work Supporter Design **Manager Resources Supporter Design**.*

*The entire contents of the Work Supporter Design **Keep Track of Colleagues Supporter Design**.*

*The entire contents of the Content Presenter Design **Task for Resources Design – Icons in Map**.*

The instances to present are determined separately for each part by the Work Supporter Designs, Task Supporter Designs and Content Presenter Designs used in the parts.

5 The FLUIDE Method

In this section, we give an overview of the parts of the FLUIDE Method that support using the FLUIDE Specification Languages, i.e. Part 1 focusing on FLUIDE-A and Part 2 focusing on FLUIDE-D.

5.1 Part 1 – Specifying user interfaces with FLUIDE-A

Figure 5.1 shows Part 1 of the method. This part suggests how to develop FLUIDE-A specifications. The solid arrows and numbering of the steps give their sequence. The vertical dimension indicates which part of the language that is used, given by the parallelograms (skewed rectangles) in the centre of Figure 5.1.

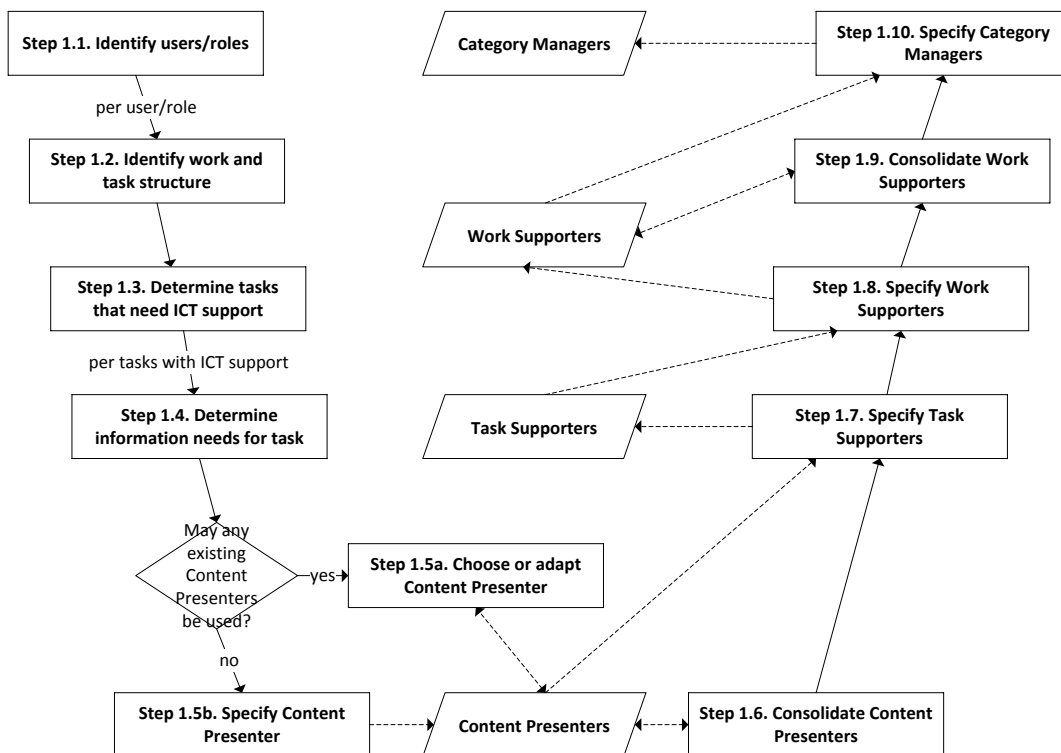


Figure 5.1. Part 1 of the FLUIDE Method

The steps on the left hand side depict a top down process starting with users, roles, work and tasks, and ending up with specifying Content Presenters. In these steps, there is a focus on specifying the task and concept models, and only Content Presenters are specified, adapted or selected. Some of these steps are performed iteratively per user/role or task. The steps on the right hand side depict a bottom up process when specifying the higher level interactor instances, to make sure the content of these specifications are specified before used. In Steps 1.6 and 1.9, Content Presenters and Work Supporters are consolidated. By this we mean a walkthrough of all such interactor instances to determine or review the connections between basic and aggregated instances of these constructs.

Below we describe each of the steps in Part 1 of the FLUIDE Method.

5.1.1 Step 1.1. Identify users/roles

When a user interface is specified, it is of paramount importance to know who are going to use it. In an emergency response setting, the same personnel may have different tasks and responsibilities in different operations. The user interfaces supporting different sets of tasks may vary. This means that the same

personnel may need different user interfaces in different operations, depending on their role in the operation at hand. Thus, it is often more important to identify the roles than the positions and actual users that may fill the roles.

Steps 1.2-1.5 should be performed for each of the identified users/roles.

5.1.2 Step 1.2. Identify work and task structure

This step consists of performing a traditional task modelling activity (Wilson and Johnson, 1996; Paternò, 1999). The results should be documented in the notation used in Work Supporters in FLUIDE-A.

In many cases, there exist appropriate task models in the organization that develops a user interface / a user interface is developed for. Such task models may be reused as they are or they may be used as a basis for adaptation to new requirements. Before doing this, it should be considered whether adapting the existing models will require more or less resources than specifying new models from scratch. There may both be a need for adjusting any existing task model and complementing them. In both cases there may also be a need for specifying brand new task models from scratch.

In Step 1.2, task models are adapted or made, but the Work Supporters embedding these task models are specified in Steps 1.8 and 1.9.

5.1.3 Step 1.3. Determine tasks that need ICT support

A task modelling activity may vary quite much with regard to how much it focuses on the role of ICT support. One extremity is to focus only on the tasks performed by the users in a real-world setting independently of any available ICT support. Another extremity is to make a task model for an existing or envisioned ICT solution, where the tasks focus on and relate to different parts of the ICT solution. Step 1.3 consists of identifying which of the tasks from Step 1.2 that indeed should be supported by an ICT solution. These tasks are the candidates for having Task Supporters, which are specified in Step 1.7. In Step 1.3, the candidate are just identified, so that the information needs can be further investigated in Step 1.4.

Steps 1.4-1.5 should be performed for each task that is identified having a need for ICT support.

5.1.4 Step 1.4. Determine information needs for task

This step to a large extent consists of performing a traditional concept modelling activity (OMG, 2008). The results should be documented as UML class models.

In many cases, there exist appropriate concept models in the organization that develops a user interface / a user interface is developed for. Such concept models may be reused as they are or they may be used as a basis for adaptation to new requirements. Before doing this, it should be considered whether adapting the existing models will require more or less resources than specifying new models from scratch. There may both be a need for adjusting any existing concept model and complementing them. In both cases there may also be a need for specifying brand new concept models from scratch.

In Step 1.4, concept models are adapted or made, but the Content Presenters embedding these concept models are specified in Steps 1.5 and 1.6.

5.1.5 Step 1.5. Choose, adapt or specify Content Presenter

This step is performed for each combination of roles from Step 1.1 and tasks needing ICT support from Step 1.3. Thus, when an arbitrary performance of Step 1.5 is conducted, a number of Content Presenters may

already have been specified. Therefore, and as can be seen in Figure 5.1, Step 1.5 consists of a choice and two sub steps, denoted Steps 1.5a and 1.5b.

The choice consists of determining whether any Content Presenters that are specified already fit to the information needs for the task at hand. If this is the case, *Step 1.5a. Choose or adapt Content Presenter* should be conducted for each Content Presenter that is identified. This sub step consists of investigating candidate Content Presenters, and for the ones that fit just note this down. For the ones that fit only partly, it should be considered whether it is possible to extend them in a way that makes them fit the needs of the task(s) to which the candidate presenters are specified. Such an extension may involve adding entities, attributes, methods and relations to the concept model of existing Content Presenters, as well as adjusting or adding annotations. Possible adjustments should not include changing the anchor. If this is not possible, the candidates should be abandoned for reuse, and Step 1.5b should be performed.

If none of the Content Presenters that are specified already fit to the information needs for the task at hand, or if the ones chosen or adapted in Step 1.5a do not cover all the information needs, *Step 1.5b. Specify Content Presenter* should be conducted one or more time. In this sub step a Content Presenter is specified from a concept model made in Step 1.4. This primarily consists of identifying the anchor and adding annotations.

To reduce the number of Content Presenters, and to avoid having redundant specifications, it should be a goal to conduct Step 1.5a whenever possible, and Step 1.5b only when needed. Even though the Content Presenters are to be specified on the basis of the information needs connected to each task, it should be a goal to make them task independent. Through this, the Content Presenters may be exploited in many tasks.

In Step 1.5, it is primarily Basic Content Presenters that are specified. The aggregated variant is primarily specified in Step 1.6.

5.1.6 Step 1.6. Consolidate Content Presenters

The information needs for a number of tasks may overlap, either because one task needs a subset of the information needed by another or because a number of tasks need different subsets of a larger connected concept model. Clear overlaps and extensions should ideally be handled in Step 1.5, but it may still be valuable to view the Content Presenters from Step 1.5, which are mainly occurrences of the basic variant, together and compare them to identify overlapping and redundant presenters. Thus, Step 1.6 consist of a walkthrough of all the Content Presenters to identify the best possible structure among them.

One particular focus in Step 1.6 is to determine whether some of the Basic Content Presenters may be specified as aggregations of a number of other Basic or Aggregated Content Presenters. On the other hand, the Basic Content Presenters should not be disintegrated down to a too low level. One might envision that doing Step 1.6 too thoroughly could result in a set of Basic Content Presenters containing exactly one entity each. To avoid this, a main principle for specifying Basic Content Presenters is that occurrences should be useful in their own right or be exploited in a number of aggregated occurrences. Furthermore, one should keep in mind that an Aggregated Content Presenter has an implicit concept model fragment, and that this model fragment needs to be connected. If the implicit concept model fragment is not connected, the aggregation should be done when specifying Task Supporters in Step 1.7.

When Steps 1.5 and 1.6 are performed, it is important to keep track of the connections to the tasks the Content Presenters specify the information need for.

5.1.7 Step 1.7. Specify Task Supporters

When this step is performed, all needed Content Presenters should have been specified (Step 1.5) and consolidated (Step 1.6). Step 1.7 consists of specifying Task Supporters for the tasks identified in Step 1.4. If the connections to the tasks are kept when the Content Presenters are specified and consolidated (Steps 1.5 and 1.6), Step 1.7 is quite simple. If not, it may involve some considerations to identify the Content Presenters that are most appropriate to aggregate into each Task Supporter being specified. Task supporters are the "glue" between task and concept models.

When specifying Task Supporters, it is also important to keep in mind the difference between Task Supporters and Aggregated Content Presenters, particularly that they aggregate Content Presenters in fundamentally different ways. When aggregating one or more Content Presenters into an Aggregated Content Presenter, the connected concept model fragments of the Content Presenters being aggregated are also implicitly merged through their anchors. This is not the case for Task Supporters. A Task Supporter may thus aggregate a number of Content Presenters with associated model fragments that are independent of each other.

5.1.8 Step 1.8. Specify Work Supporters

When this step is performed, all needed Task Supporters should have been specified (Step 1.7). Step 1.8 consists of specifying Work Supporters from the task models made in Step 1.2, as well as coupling Task Supporters (specified in Step 1.7) to the tasks that need ICT support (identified in Step 1.3). In Step 1.8 it is primarily Basic Work Supporters that are specified. The aggregated variant is primarily specified in Step 1.9.

In the same way as the connection between Task Supporters and Content Presenters, one Task Supporter may be used in different Work Supporters, but as Task Supporters may not aggregate other Task Supporters the structure is simpler.

5.1.9 Step 1.9. Consolidate Work Supporters

The task models identified in Step 1.2 are made iteratively for different roles and users. This may result in different task models, and thus corresponding Work Supporters (specified in Step 1.8) that overlap to a certain degree. Such overlaps may have been handled in Step 1.2 and/or Step 1.8, but it may still be valuable to view the Work Supporters from Step 1.8, which are mainly the basic variant, together and compared them to identify overlapping and redundant supporters. Thus, Step 1.9 consist of a walkthrough of all the Work Supporters to identify the best possible structure among them.

One particular focus in Step 1.9 is to determine whether some of the Basic Work Supporters may be specified as aggregations of a number of other Basic or Aggregated Work Supporters. On the other hand, the Basic Work Supporters should not be disintegrated down to a too low level. One might envision that doing Step 1.9 too thoroughly could result in a set of Basic Work Supporters containing exactly one task each. To avoid this, a main principle for specifying Basic Work Supporters is that occurrences should be useful in their own right or be exploited in a number of aggregated occurrences. Furthermore, one should keep in mind that an Aggregated Work Supporter has an implicit task model fragment, and it is important to make sure that the hierarchical structure from the one task added in an Aggregated Work Supporter to the anchors of its member Work Supporters make sense. If not, the aggregation should be done when specifying Category Managers in Step 1.10.

5.1.10 Step 1.10. Specify Category Managers

When this step is performed, all Work Supporters and Content Presenter should have been specified in the prior steps. But while the aggregation of Content Presenters into Task Supporters, as well as the aggregation

of Task Supporters into Work Supporters to a large extent are given by Steps 1.1-1.5, there is no results from earlier steps that direct which Category Managers that should be specified.

A number of approaches may be used to choose which Category Managers to use. These approaches may well give similar results, and could thus be combined. Before sketching some possible approaches, it should be emphasized that a Category Manager should specify a user interface supporting a category of functionality. It is also important to keep in mind that Category Managers primarily aggregate Work Supporters. They may also aggregate Content Presenters, but this option is to be used to supplement the Category Managers if the Work Supporters that are aggregated do not support all aspect of some category of functionality, as well as in cases where the user interfaces and/or the task structures are so simple that Task and Work Supporters are not needed/used.

Based on this, a good starting point to choose which Category Managers to use is to look into related Work Supporters having related task models. One approach for finding such Work Supporters is to consider Work Supporters for one or a number of related roles/users. Another approach is to consider the information that is presented, as this may overlap across roles and users. A third approach is to use the categories of functionality from Nilsson and Stølen (2011) as a starting point or inspiration, i.e.:

1. Operational picture
2. Incident details
3. Logging
4. Information services
5. Resource management
6. Actions and plans
7. Transmission
8. Monitoring
9. Automatic reasoning
10. Communication management
11. Special interaction mechanisms

Furthermore, there should be a limited number of Category Managers. If the number of chosen Category Managers exceed 15, the choice may be reconsidered.

With an appropriate set of chosen Category Managers, determining which interactor occurrences (Work Supporters and Content Presenters) that should be part of the Category Managers is normally a manageable task, particularly if a grouping of Work Supporters is used when choosing Category Managers.

When specifying Category Managers, it is also important to keep in mind the difference between Category Managers and Aggregated Work Supporters, particularly that they aggregate Work Supporters in fundamentally different ways. When aggregating one or more Work Supporters into an Aggregated Work Supporter, the task model fragments of the Work Supporters being aggregated are also implicitly merged through their anchors. This is not the case for Category Managers. A Category Managers may thus aggregate a number Work Supporters with associated task models that are independent of each other.

A Category Manager aggregating only Content Presenters is semantically identical to a Task Supporter aggregating the same Content Presenters.

5.2 Specifying user interfaces with FLUIDE-D

Figure 5.2 shows Part 2 of the method. This part suggests how to develop FLUIDE-D specifications.

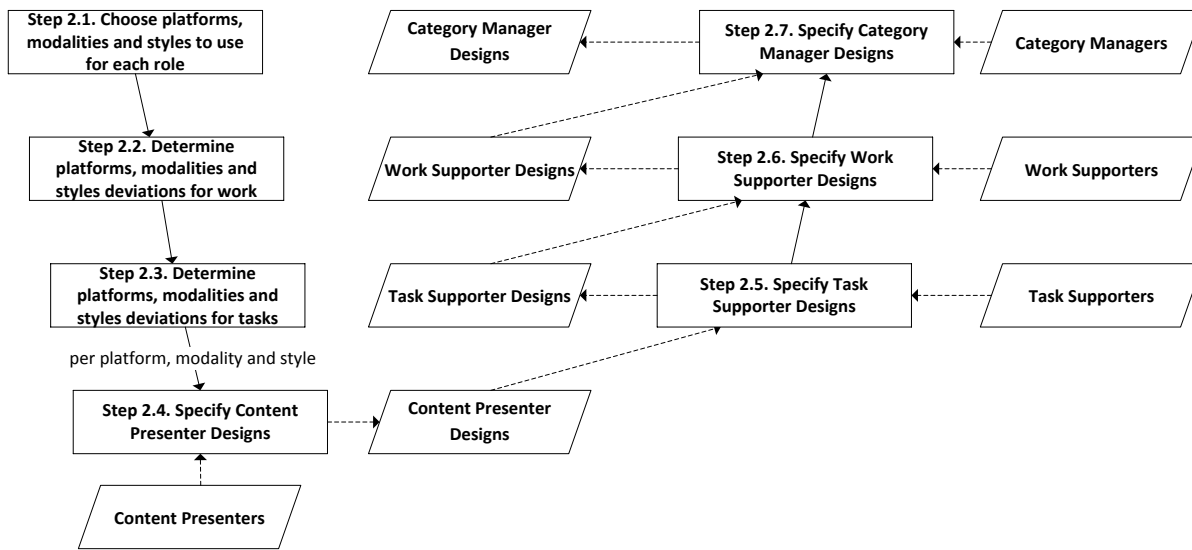


Figure 5.2. Part 2 of the FLUIDE Method

The steps on the left hand side depict a top down process starting with identifying the platforms/modalities as well as the user interface styles to use for each role. These choices are then adjusted by finding out any deviations to these choices in particular work or tasks, ending up with specifying the necessary Content Presenter Designs for the platform, modality and style choices for the different task and work structure.

The steps on the right hand side depict a bottom up process when specifying the higher level interactor design instances, to make sure the content of these specifications are specified before used. For Basic Content Presenter Designs, it is important to choose which part of the concept model to include in the design, and which views to use for wrapping different part of the model. For the other interactor design constructs, there are three choices to make. The first is which child interactors from the corresponding FLUIDE-A specification to include designs for. The second is which designs to use for these (if more than one is available). The third is which views to use for wrapping the child designs.

Below we describe each of the steps in Part 2 of the FLUIDE Method.

5.2.1 Step 2.1. Choose platforms, modalities and styles to use for each role

Each interactor design occurrence must be aimed at a specific target (an arbitrary combination of platform, type, style and modality used in a running user interface). In Step 2.1 the main targets are chosen for each role/user that are to exploit the user interface being specified. As these choices may both be restricted and augmented in the consecutive steps, this choice should reflect the default targets to use for each role. Despite the choices being a default, they may well include more than one target for each role.

5.2.2 Step 2.2. Determine platforms, modalities and styles deviations for work

In this step, each part of the work for the different roles, as expressed in the task models related to the Work Supporters specified in Part 1 should be considered to identify any deviations from the default choices made in Step 2.1. Such deviations may include both limiting and augmenting the choice of targets to use. Limiting should typically be done if some kind of work is only expected to be supported by a subset of the default targets. Augmenting should typically be done if some kind of work is expected to be conducted using platforms, modalities and styles that are not among the default. The choice of not including these targets among the default ones is natural if such deviations are identified for a minority of the kinds of work supported by the specification.

5.2.3 Step 2.3. Determine platforms, modalities and styles deviations for tasks

Step 2.3 is similar to Step 2.2, except that the individual tasks are considered instead of the task models. The deviations may either be from the default choice from Step 2.1 or from the deviations identified in Step 2.2 for the work that a given task is part of. Step 2.3 may include limiting and augmenting the choice of targets to use in the same way as in Step 2.2. Such deviations may include both limiting and augmenting the choice of targets to use. Limiting should typically be done if some tasks are only expected to be supported by a subset of the default targets. Augmenting should typically be done if some tasks are expected to be conducted using platforms, modalities and styles that are not among the default. The choice of not including these targets among the default ones is natural if such deviations are identified for a minority of the tasks supported by the specification. Similar considerations may be conducted towards the kinds of work whose task model the tasks with deviating targets are part of.

5.2.4 Step 2.4. Specify Content Presenter Designs

Step 2.4 should be performed for each combination of platform, modality and style.

Normally, Content Presenter Designs should use the same targets as the tasks needing the information presented in the Content Presenter Designs. Some Content Presenter Designs may use only a subset of the platforms, modalities and styles identified for these tasks. Content Presenter Designs should not exploit additional targets that are not identified for these tasks.

There may be a need for a number of designs for each Content Presenter in the FLUIDE-A specification. This is the case if different tasks needing the information presented in the Content Presenter Design are to exploit different targets. It may also be the case if different tasks needing the information presented in some Content Presenter Design use the same targets but still need to have the content presented in different ways.

Given the needs of the tasks and the chosen targets, specifying each Content Presenter Design consists of wrapping one or more subsets of the connected concept model fragment of the corresponding Content Presenter into a hierarchy of views. For Basic Content Presenter Designs such a hierarchy must include at least one Content View. For Aggregated Content Presenter Designs, the hierarchy will usually include at least one Content Integration View. When specifying an Aggregated Content Presenter Design, it is also possible to omit the designs for one or more of the member presenters from the corresponding FLUIDE-A specification. The possible choices of Content Views/Content Integration Views to exploit may be dependent of the chosen target for a given design. The choice of Content Integration View to use is also influenced by the Content (Integration) Views used in the children.

5.2.5 Step 2.5. Specify Task Supporter Designs

Even though the aggregation structure specifying which Content Presenters that are member of a Task Supporter is given from the FLUIDE-A specification, there are still three main choices that must be made when specifying a Task Supporter Design.

First, it is possible to omit the design for some of the child Content Presenters. For Aggregated Content Presenters that are member of a Task Supporter, it is possible to include the designs for only some of the Aggregated Content Presenters' children (recursively). On the other hand, it is not possible to add designs for presenters that are not part of the Task Supporter or any of its children.

Second, there may exist more than one design for the child presenters whose designs should indeed be included in a Task Supporter Design. Only one of these may be used in the Task Supporter Design. This choice may be given by the target(s) for the Task Supporter Design, but in cases with a number of designs for the same target(s), the most appropriate one should be used.

Third, the chosen child presenter designs must be wrapped into a hierarchy of views. This hierarchy will usually include at least one Content Integration View. The possible choices of Content Integration Views to exploit may be dependent of the chosen target(s) for a given design. The choice of Content Integration View to use is also influenced by the Content (Integration) Views used in the children.

Normally, the targets for a Task Supporter Design is determined in Steps 2.1-2.3. Still, it should be checked whether the Task Supporter Design has any child presenter designs deviating from the choices in Steps 2.1-2.3. If this is the case, it should first be asserted that the child presenter designs indeed are appropriate for the Task Supporter Design. If this is not the case, the Task Supporter Design should be adjusted. If the children indeed are appropriate, the targets for the Task Supporter Design should be extended so that the sum of its children's targets are included.

5.2.6 Step 2.6. Specify Work Supporter Designs

Even though the aggregation structure specifying which Task Supporters that are member of a Basic Work Supporter, as well as the Task and Work Supporters that are members of an Aggregated Work Supporter are given from the FLUIDE-A specification, there are still three main choices that must be made when specifying a Work Supporter Design.

First, it is possible to omit the design for some of the child Task and Work Supporters. Furthermore, it is possible to include the designs for only some of the Task and Work Supporters' children (recursively). On the other hand, it is not possible to add designs for supporters and presenters that are not part of the Work Supporter or any of its children.

Second, there may exist more than one design for the child supporters and presenters whose designs should indeed be included in a Work Supporter Design. Only one of these may be used in the Work Supporter Design. This choice may be given by the target(s) for the Work Supporter Design, but in cases with a number of designs for the same target(s), the most appropriate one should be used.

Third, the chosen child supporters and presenter designs must be wrapped into a hierarchy of views. This hierarchy may include one or more Content Integration Views. The possible choices of Content Integration Views to exploit may be dependent of the chosen target(s) for a given design. The choice of Content Integration View to use is also influenced by the Content (Integration) Views used in the children.

Normally, the targets for the Work Supporter Design is determined in Steps 2.1-2.2. Still, it should be checked whether the Work Supporter Design has any child supporter or presenter designs deviating from the choices in Steps 2.1-2.2. If this is the case, it should first be asserted that the child supporter and presenter designs indeed are appropriate for the Work Supporter Design. If this is not the case, the Work Supporter Design should be adjusted. If the children indeed are appropriate, the targets for the Work Supporter Design should be extended so that the sum of its children's targets are included.

5.2.7 Step 2.7. Specify Category Manager Designs

Even though the aggregation structure specifying which Work Supporter and Content Presenters that are member of a Category Manager Design is given from the FLUIDE-A specification, there are still three main choices that must be made when specifying a Category Manager Design.

First, it is possible to omit the design for some of the child Work Supporters and Content Presenters. Furthermore, it is possible to include the designs for only some of the Work Supporters' and Content

Presenters' children (recursively). On the other hand, it is not possible to add designs for supporters and presenters that are not part of the Category Manager or any of its children.

Second, there may exist more than one design for the child supporters and presenters whose designs should indeed be included in a Category Manager Design. Only one of these may be used in the Category Manager Design. This choice may be given by the target(s) for the Category Manager Design, but in cases with a number of designs for the same target(s), the most appropriate one should be used.

Third, the chosen child supporters and presenter designs must be wrapped into a hierarchy of views. This hierarchy may include one or more Content Integration Views. The possible choices of Content Integration Views to exploit may be dependent of the chosen target(s) for a given design. The choice of Content Integration View to use is also influenced by the Content (Integration) Views used in the children.

Normally, the targets for the Category Manager Design is determined in Step 2.1. Still, it should be checked whether the Category Manager Design has any child supporter or presenter designs deviating from the choices in Step 2.1. If this is the case, it should first be asserted that the child supporter and presenter designs indeed are appropriate for the Category Manager Design. If this is not the case, the Category Manager Design should be adjusted. If the children indeed are appropriate, the targets for the Category Manager Design should be extended so that the sum of its children's targets are included.

Specifying a design for a Category Manager that only aggregates Content Presenters is similar to specifying a design for a Task Supporter (Step 2.5).

References

Borchers, J: *A Pattern Approach to Interaction Design* 2001; John Wiley & Sons.

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L. and Vanderdonckt, J.: *A unifying reference framework for multi-target user interfaces*; *Interacting with Computers* 15(3) 2003, p. 289-308; Elsevier/Oxford Journals.

CAMELEON: *CAMELEON glossary*; Deliverable 1.1 Companion in the CAMELEON project, 2003; Available at: <http://giove.isti.cnr.it/projects/cameleon/glossary.html>

Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object Oriented Software* 1994; Addison-Wesley.

Nilsson, E.G.: *Tasks and information models for local leaders at the TYR training exercise*; SINTEF Report A16007, 2010a; ISBN 978-82-14-04479-9.

Nilsson, E.G. and Stølen, K.: *Generic functionality in user interfaces for emergency response*; Proceedings of the 23rd Australian Computer-Human Interaction Conference (OZCHI'11) 2011; ACM.

Nilsson, E.G. and Stølen, K.: *A case-based assessment of the FLUIDE framework for specifying emergency response user interfaces*; Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '16) 2016a; ACM.

Nilsson, E.G. and Stølen, K.: *The FLUIDE framework for specifying emergency response user interfaces employed to a search and rescue case*; Proceedings of the 13th International ISCRAM Conference 2016b.

OMG: *OMG unified modeling language (OMG UML), superstructure, V2.1.2*; 2008; Available at: <http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/>

Paternò, F.: *Model-based design and evaluation of interactive applications* 1999; Springer.

Wilson, A. and Johnson, P.: *Bridging the generation gap: from work tasks to user interface designs*; Proceedings of Computer-Aided Design of User Interfaces (CADUI) 1996; Presses Universitaires de Namur.



Technology for a better society

www.sintef.no