FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Developing Normalized Systems Conceptual Modeler
Student:	Bc. Peter Uhnák
Supervisor:	Ing. Robert Pergl, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

The goal is to develop an advanced and detailed conceptual modeling tool for the data and flow models in Normalized Systems Theory. The tool must serve to explore the requirements for a state-of-the-art conceptual modeling environment for analysts and programmers working on Normalized Systems projects (from small- to large-scale, mission critical systems). The tool should focus on the modeling using graphical diagrams and support the inspection and comprehension of models as well as error detection and prevention. In combination with Normalized Systems Theory, the tool would offer the user a unique platform for modular conceptual models for information systems.

Key characteristics of the tool include:
- User friendly
- High flexibility
- Robustness
- Loosely coupled integration with the NS Prime Radiant, which functions as the repository and code generation environment for the conceptual modeling tool.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 9, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Developing Normalized Systems Conceptual Modeler

Bc. Peter Uhnák

Department of Software Engineering
Supervisor: Ing. Robert Pergl, Ph.D.

May 9, 2018

Declaration

Citation of this thesis

Uhnák, Peter. *Developing Normalized Systems Conceptual Modeler.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce se zaměřuje na tvorbu modelovacího a diagramovacího nástroje v platformě OpenPonk pro potřeby Normalizovaných Systémů a meta-systému Prime Radiant.

Stručně je představena teorie Normalizovaných Systémů, platformy OpenPonk a dalších nástrojů použitých při tvorbě modeláře.

Pro potřebu integrace mezi systémem Prime Radiant a implementovaným modelářem jsou analyzovány soubory XML obsahující deskripce Normalizovaných Systémů. Z analýzy je následně zkonstruován vhodný metamodel. Jak proces analýzy, tak i výstupný metamodel je detailně rozebrán.

Současná notace používaná pro Normalizované Systémy je obohacena, a mimo jiné čerpá z myšlenek notací jako je diagram databází, UML diagramy tříd či stavových strojů. Jsou probrány různé možnosti, návrhy a přístupy k diagramové tvorbě modelů Normalizovaných Systémů.

Dále je představena série validačních pravidel pro modely Normalizovaných Systémů. Téma samotného popisu takových pravidel je probrána, stejně jako i aktuální způsob implementace.

Neboť tato práce popisuje softwarový projekt, je modelář zanalyzován a zhodnocen z hlediska testování a testovatelnosti, stejně jako i dlouhodobé údržby.

Na závěr jsou shrunty přínosy této a práce a je navržena série kroků vhodných k budoucímu prozkoumání.

Klíčová slova

Normalizované Systémy, Prime Radiant, OpenPonk, modelování, diagramování, vizualizace, Pharo, Smalltalk

Abstract

This thesis explores the topic of providing modeling and diagramming support for Normalized Systems constructed in the Prime Radiant tool using the OpenPonk modeling platform.

The current state of the art of defining NS systems is summarized. The Normalized Systems theory, Prime Radiant, OpenPonk, and other framework used in the making of the modeler are introduced.

The thesis then continues by analyzing the artifacts produced by the Prime Radiant that hold the definitions of NS systems. These artifacts are reverse engineered and a metamodel is constructed; both the process, and the result is discussed.

Inspired by existing notations, such as Entity-Relationship diagrams, UML class diagrams, State Machine diagrams, and Flow diagrams, the existing NS diagramming notation is extended. It discusses various aspects not just of the notation itself, but also of the process of diagramming as a natural way to create the models. Several possibilities of forthcoming exploration are introduced.

Modeling rules and validations are introduced and explored as a way to guide a user through possible pitfalls, and to raise the quality of the defined models. The implemented rules, the current state, and an approach relevent from mid- and long-term perspective is explored and discussed at length.

As the main artifact of this thesis is a software project, testing, error tracking, and operations utilized in construction of the modeler are described.

Finally, the achievements of this project are evaluated, and the path forward summarized.

Keywords

Normalized Systems, Prime Radiant, OpenPonk, modeling, diagramming, visualizations, Pharo, Smalltalk

Contents

List of Figures

Introduction

Motivation

The Normalized Systems (NS) theory provides a comprehensive approach to constructing large-scale information systems based on the laws of entropy and thermodynamics. Such systems remaining flexible, modular, and evolvable even in the face of unbounded growth and changes. The NS theory provides a bottom-up approach, where it defines the design and properties of fine-grained modules that are directly mapped to a programming language.

Such bottom-up approach is in contrast to ontological approaches, where users define their systems in high-level abstract models, typically utilizing rich diagramming notations (e.g. UML, OntoUML, BPMN). But those approaches do not properly address the problem of translating the high-level concepts to real, executable systems. Thus there is a large conceptual gap between high-level ontologies and low-level implementation.

Currently, NS systems are designed in a meta-information system Prime Radiant, which is itself an NS system. There, the analysts and programmers describe system's technical properties using extensive and detailed textual forms. With such a strong focus on the low-level details, it can become hard for the users to maintain an overview of the entire system, thus often resulting in both technical and semantical errors. Users may choose to create paper drawings and sketches, but such approach is error-prone in itself, time-consuming, and not maintainable.

The Normalized Systems modeler is an attempt to categorically improve the situation.

The NS modeler makes a step towards unifying the low-level technical detail of Normalized Systems and the high-level conceptual perspective. It provides a diagramming editor where NS systems can be described through a visual notation similar to many existing modeling notations. High-level concepts and approaches are introduced for clarity, convenience, better understanding, and prevention of many potential errors. However, such concepts are still directly mapped to the low-level NS definitions – the modeler does not jump across the conceptual gap.

Furthermore, the modeler acts as an exploration ground for aspects related to the modeling and analysis of NS systems, such as the definition and implementation of validation rules.

Goals and Objectives

The goal is to develop an advanced and detailed conceptual modeling tool for the data and flow models in Normalized Systems Theory. The tool must serve to explore the requirements for a state-of-the-art conceptual modeling environment for analysts and programmers working on Normalized Systems projects (from small- to large-scale, mission-critical systems). The tool should focus on the modeling using graphical diagrams and support the inspection and comprehension of models as well as error detection and prevention. In combination with Normalized Systems Theory, the tool would offer the user a unique platform for modular conceptual models for information systems. Key characteristics of the tool include user-friendliness, high flexibility, robustness, loosely coupled integration with the NS Prime Radiant.

In addition to above goals, an internal set of goals exists in respect to OpenPonk. Namely the advancement of OpenPonk modeling platform, and possibly Pharo, and the Pharo ecosystem as well. It means that general (or generalizable) improvements and new features in the NS modeler should ideally be transferable back into OpenPonk to enrich it. I consider it essential, as OpenPonk is meant to provide a free foundation for building tools such as the NS modeler, and conduct modeling research, as is done in FIT CTU's *Centre For Conceptual Modelling and Implementation* (CCMI)[1] research group, under which OpenPonk is created. With such approach, additions made in one place can improve situation elsewhere and vice versa.

Structure of the Thesis

Part I *Review* focuses on review of theory and tooling utilized in this work.

Chapter 1 *Normalized Systems Theory* presents the an excerpt of the Normalized Systems Theory that is relevant and/or applicable to this work and the implementation of the modeler.

Chapter 2 *OpenPonk Modeling Platform* reviews the current state of the OpenPonk modeling platform, and how it advanced during the past years since its inception during my undergraduate studies.

Chapter 3 *Magritte* discusses the Magritte meta-description framework that is used in several places throughout this work.

[1] `https://ccmi.fit.cvut.cz/en/`

Part II *Analysis and Reverse Engineering of Metamodel* focuses on exploring and explaining the Normalized Systems metamodel as understood through the artifacts produced by Prime Radiant.

Chapter 4 *Normalized Systems Metamodel* describes the Normalized Systems metamodel and the NS modeler metamodel. The chapter focuses mostly on the technical details of it.

Chapter 5 *Metamodel Engineering* navigates through my reverse-engineering of XML files and other data artifacts produced by the Prime Radiant system, and though much of the metamodel in *Normalized Systems Metamodel* was engineered.

Part III discusses main technical aspects and features of the NS modeler.

Chapter *Rules and Validations* discusses in depth both theoretically and technically validation rules that are applied to Normalized Systems models.

Chapter *Diagram Editors* shows the current state of the diagramming aspect of component and flow modeling. A portion of the chapter is dedicated to exploring potential future expansions.

Chapter *Magritte Extensions* shows additional use cases and features built using the Magritte meta-description framework.

Part IV foucses on the Testing and Operations aspects of building tools such as the NS modeler and OpenPonk

Chapter *Error Tracking and Reporting* deals with detection of errors during the lifetime of the application and with their reproducibility.

Chapter *Testing, Continuous Integration and Deployment* summarizes the continuous testing efforts and explores various strategies applicable to the NS modeler and OpenPonk. A section is dedicated to Continuous Integration & Deployment.

Finally, in Chapter *Conclusion* the achievements of this thesis are summarized and a road forward is presented.

Part I

Review

Chapter 1

Normalized Systems Theory

In this chapter, several principles and concepts of Normalized Systems (NS) theory *[12]* are presented. The information presented is a fragment of the entire NS theory and by no means is meant to be exhaustive nor canonical.

Furthermore, the purpose of the modeler was not to construct NS systems, but rather to provide *another* way to describe such systems, and support the existing infrastructure – namely the Prime Radiant system.

1.1 High-level Overview

Normalized Systems (NS) is theoretical framework investigating modular structures under change. More specifically it aims to address and solve the problems and challenges associated with the unbounded growth of information systems.

NS observes coupling and ripple effects as a primary contributor to the ever-growing cost of development and maintenance of existing systems. A linear addition to such a system results in a superlinear impact on the system – the cost of every addition increases with the size of the system, not the size of the addition.

To address this problem, NS founds itself in the natural laws of entropy and thermodynamics used in physical sciences and engineering and applies them to the construction of software systems. Using these laws, it explores and defines properties that are necessary for a stable modular architecture (*Stable Modular Architecture*). It defines fine-grained building blocks that can be combined, changed, and modified while maintaining an impact on the system proportional to the size of the input (addition), rather than to the size of the (unbounded) system.

The combination and construction of such fine-grained and uniform building blocks is performed by an automated generation process called *expansion.* The expanders are subject to their own requirements – dimensions of evolvability (*Dimensions of Evolvability*; in fact, the expanders are *necessary* to construct a system with the essential evolvability properties. The expanders accompany the (expanded) system during its entire lifetime and allow for continuous regeneration (rejuvenation) as the system grows and changes.

1.2 Dimensions of Evolvability

All software systems, NS or not, have several orthogonal dimensions along which they can evolve. Any change in any of these dimensions has a potentially negative impact on the entire the entire system. Therefore it is critical to separate the dimensions so they can evolve independently.

NS theory identifies the following four dimensions of evolvability.

Mirrors (models) Real world concepts are *mirrored* in the software world via the construction of domain models. The mirrors are the fundamental core of a system, as they represent the targeted business. As the business evolves and changes, so must the mirrors.

It is necessary to express the models in some (software) form, but the representation itself should have no bearing on the model itself. Even expressed in different programming or modeling languages, they should maintain their essence.

The mirrors are described in a meta-information system Prime Radiant. However, the NS modeler aims at providing an alternative, or a complementary approach to Prime Radiant.

Technology (utilities) This dimension is concerned with the choice of technical stacks of the constructed systems. The concerns include selections of the user interface frameworks, database, logging, etc.

In traditional systems, the choice of technology is often critical, as it directly places limitations and possibilities on the remaining dimensions. For example, a team lead by database specialists may decide to go so far as to define the mirrors only in the database of their choice, which would be fundamentally different between a relational and a NoSQL database. In another team, the developers of a web application may choose a framework due to its current hype and vendor-lock themselves. A particularly dangerous situation considering the current proliferation of web frameworks. In both cases the technological choice impacts every other aspect, and a change to a different platform can result in a complete rewrite of the application without providing any new value to the business.

The ability to separate the technology dimension poses an immense value for growth and flexibility of a system.

Element Structure (skeletons) Skeletons are the shape and content of the fine-grained building elements. The element structure defines how various concerns ought to be combined, how the element should connect to other elements and communicate with them.

The structure must be flexible and powerful, as the expansion process combines the three other dimensions according to the structure descriptions.

Plugin Code (craftings) Living systems are always subject to evolution, change, and ever more complex requirements. The existing state of a system may not be flexible enough or advanced enough to accommodate quickly new changes in a clean, generic way. Quite the contrary, often a full understanding of a requirement comes from the implementation itself, as it forces to explore every possible scenario and variation.

To address the insufficient flexibility, the system must offer engineers the option to combine clean, generated code with their hand-crafted solutions.

At the same time, the evolution of other dimensions should not impact the injected code, and there should be a natural way for the custom craftings to slowly dissolve and become a normalized part of an expanded system.

NS systems achieve this by generating anchor points where an engineer can introduce custom code. To maintain evolvability of the other dimensions, it can *harvest* and separate the craftings. In this way, it is possible to regenerate the entire system due to e.g. a technology change while preserving the custom code – the expanders harvest code, regenerate the system, and apply the craftings where they belonged.

1.3 Stable Modular Architecture

This section presents four key theorems necessary for an architecture that is considered stable and evolvable in the scope of the NS theory.

Separation of Concerns Separation of Concerns is the separation tasks (individual steps) within a processing function. A processing function should handle only one task that is a potential *change driver*. A change driver in this context is a change that can occur independently of other steps within the processing function.

A low-level example violating separation of concerns is a processing function that reads a CSV file, extracts data for reporting and sends them via email. Such processing function contains at least three distinct concerns (change drivers) that can change independently.

A high-level example upholding the theorem is an enterprise service bus (ESB). Any service added has a fixed impact – only the integration with the ESB. Without the ESB, integration with all previously existing services may be required (the number of connections grows quadratically).

That theorem ties directly to the generally accepted concept of maintaining *high cohesion.* More importantly, it explicitly states what constitutes a highly cohesive element via the description of change drivers.

Data Version Transparency This theorem postulates that processing functions of data structures must be capable of accepting different versions of the same structure without being negatively affected by them.

Within object-oriented systems, this implies that functions should be provided with composite objects (*stamp coupling*) instead of separated primitive types (*data coupling*).

In a classical approach the latter is preferred, as the processing function is coupled only to the arguments that are provided and used by it. With a stamp coupling, such function is coupled to every property of the stamp, whether they are used or not.

From an evolvability point of view, any addition would result in an impact on the interface of the processing function. Thus stamp coupling is required for Data Version Transparency.

This theorem is provided in many non-oo systems. For example, a newly added attribute to an XML element is often ignored by a processing function which is not (yet) aware how it should handle it. Likewise, the query portions of an URL can be amended with additional key=value pairs without any impact on the functionality.

Action Version Transparency Similarly to Data Version Transparency, to achieve an evolvable system without combinatorial effects, it must be possible for actions (processing functions) to change without impacting processing functions that call the changed processing function.

In OO languages, this is achieved primarily via polymorphism – all instances of classes implementing the same interface can be used interchangeably. Thus a new version of a processing function, assuming the interface remains honored, can replace its previous version without impacting the callers.

Separation of State This theorem is concerned with the propagation of state through calls between processing functions. It mandates that it is not the processing function's concern to deal with the state of other processing functions.

An example in many OO languages is exception handling mechanism. After a change to a function, a new potential error state has been introduced or discovered, and the function fires a new exception. In such scenario, all calling functions are potentially impacted, as they should start handing the new exceptional behavior. Note that this is different from Action Version Transparency violation, as the interface remains to same.

The theorem thus requires separation (externalization) of state from the processing function.

An example with clearly defined separation of state is an assembly line. At each step of the assembly line a (Data Version Transparent) input is provided, the assembly step performs its action (processing function), and the constructed artifact is returned back to the assembly line. If an error occurred, then the artifact can be placed on a different line or removed entirely. At no point is an assembly step concerned with what the next step is doing, or what the previous step has done. It is the responsibility of the assembly line designer to place the steps in proper order. Similarly, in NS system it is the responsibility of a *workflow* to chain the individual actions as appropriate.

Returning to the exception handling mechanism, with Separation of State in mind, the calling function is no longer responsible for handling the error state. Instead, it has performed its job, passed data to the next step, and terminated. New error states arising in the called processing function no longer have any impact.

Chapter 2

OpenPonk Modeling Platform

OpenPonk modeling platform *[13]* is a free, open-source platform for developing tools for conceptual modeling such as diagramming, DSLs, model transformations, automatic layouting, validation, and more.

2.1 Overview

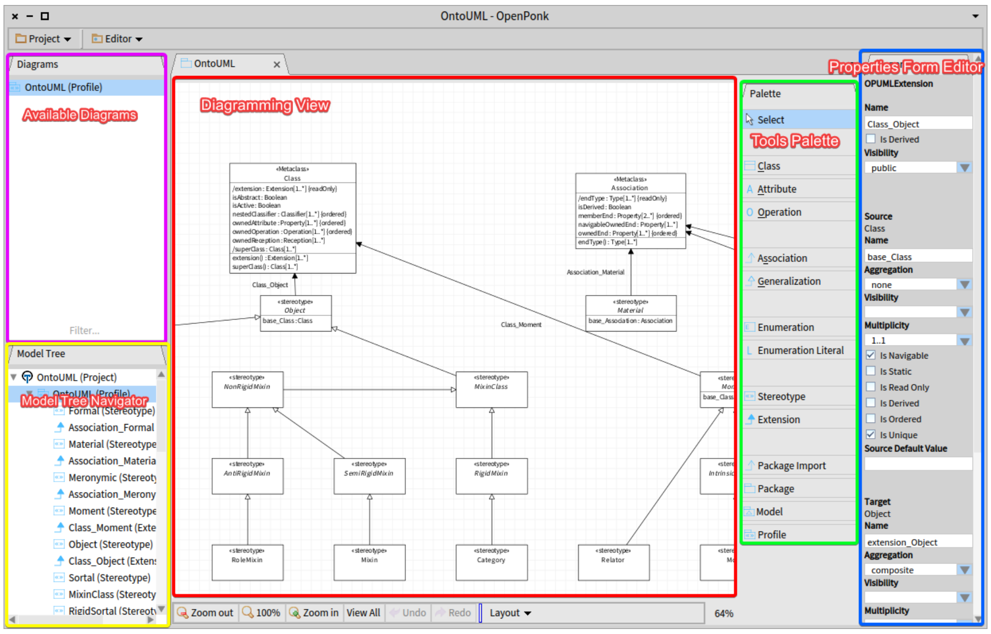


Figure 2.1: OpenPonk Workbench showing a portion of an OntoUML profile in a UML profile editor

Figure 2.1 shows OpenPonk Workbench, the main window of the modeling and diagramming environment. Each such workbench is tied to a single project, which in itself includes one or more models. The contained models do not have to be related or connected, nor do they need to share the same metamodel.

The individual subparts of the Workbench are as follows.

The left side of the Workbench contains *Diagrams* and *Model Tree* widgets. The former displays all diagrams available in a project and opens them. The latter shows a hierarchical tree of a subset of entities in the project's models and provides basic contextual actions. Needless to say, both widgets are under-utilized. Later in this work, some ideas about improving diagram organizations are presented.

The right-most widget is *Properties* form. This widget provides easy access to the essential aspects and properties of a currently selected entity. This widget has been improved in this thesis (see *Spec Properties Form Renderer* in *Magritte Extensions*).

The central part is occupied by the main diagramming interface. The *Diagramming View* itself can provide contextual controls for convenience, however the main controls are present in the *Tools Palette.* With both, users can create, modify, and view their models through an appropriate diagram notation.

The overall UI in its form started as a monolith, but it is slowly being replaced by a more modular approach.

2.2 Editors

To provide support for different models and notations, OpenPonk operates in terms of editor plugins. Every plugin is associated with a specific model and notation. For example Figure 2.1 shows a ClassEditor Plugin. The plugin is responsible for providing interaction between the model and non-diagramming components such as the Model Tree, or Workbench toolbar menus. It also determines the entry point for the Diagram Controller (see further).

2.3 Diagram Editor Architecture

The architecture of the diagram editor is a variation of the MVC pattern *[14]* (see Figure 2.2). A significant difference is the observability (or rather non-observability) of the model. OpenPonk can operate on foreign metamodel implementations – that is, a code base that is provided as-is without the option to modify it in any way. As a result, the instantiated model entities do not always announce their internal changes, an essential component of a classical MVC architecture.

Thus it is the responsibility of the controllers to be able to accommodate for it. It is a relevant problem in the Pharo environment, as users can do access model instances directly or through other means and interfaces. A smooth integration despite these challenges continues to be explored.

As for the MVC in this context:

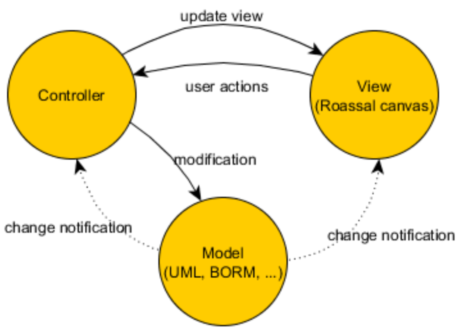


Figure 2.2: MVC architecture with limited communication from model

model Implementation of a particular metamodel that is being instantiated to create models, such as UML metamodel, BPMN, etc.

view OpenPonk uses Roassal visualization library *[15]* for the drawing and management of visual elements. For practical purposes, it may be convenient to introduce intermediate layers for a better notation representation. E.g., the UML editor and the NS component editor both use UML shapes library (see *Author's Contributions* in the chapter *Conclusion*).

controllers The controllers are responsible for mediating actions and changes between user, view, and model. Typically every metamodel concept that has a visual counterpart has a controller. In its current form, it can end up handling all matters related to an entity; a challenge that is continuously reexamined as new circumstances arise.

In addition to one controller per visual entity, a special *diagram controller* is required that manages other controllers and the overall interaction, including the description of a tool palette[1].

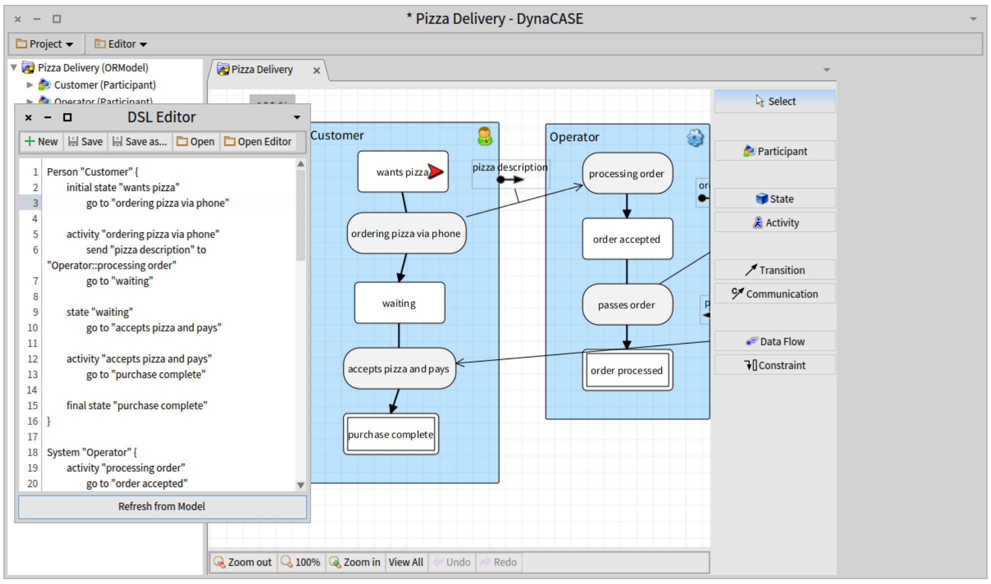
2.4 Orthogonal Perspectives

Although the Workbench provides the canonical approach to manipulate models, it is not the only one. As was already noted, the model can and is used independently of the Workbench. We demonstrate it on the following two examples.

Figure 2.3 shows a Domain Specific Language (DSL) editor for BORM model. The editor is an independent application in the same Pharo runtime, but is not in any way connected to the OpenPonk Workbench. The editor only operates on the same BORM model instance as the Workbench. Thus it

[1] Palette tools typically only instantiate new controllers at a user-chosen place and moment.

is possible to edit the model in one tool and see the changes live in another one and vice versa.



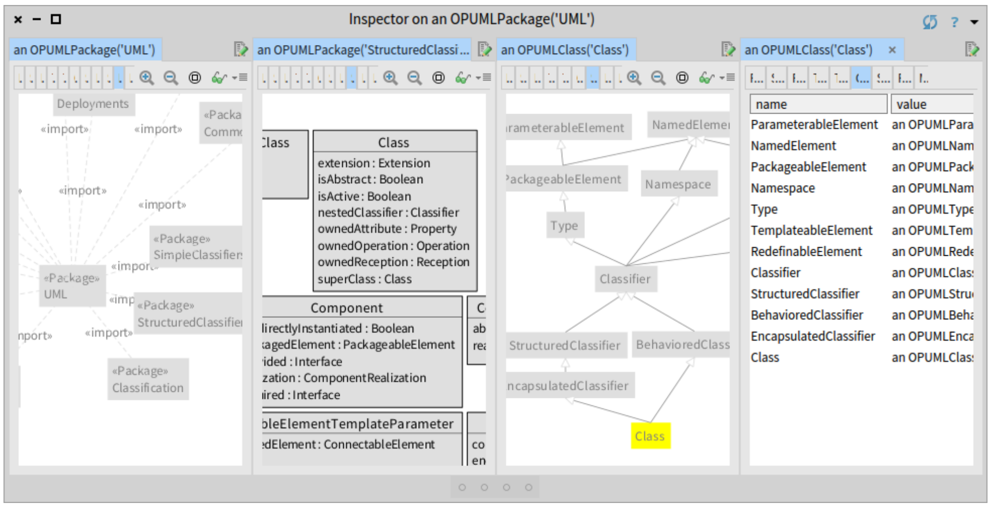
```
Person "Customer" {
    initial state "wants pizza"
        go to "ordering pizza via phone"

    activity "ordering pizza via phone"
        send "pizza description" to
"Operator::processing order"
        go to "waiting"

    state "waiting"
        go to "accepts pizza and pays"

    activity "accepts pizza and pays"
        go to "purchase complete"

    final state "purchase complete"
}

System "Operator" {
    activity "processing order"
        go to "order accepted"
```

Figure 2.3: DSL editor for a BORM model

In Figure 2.4 we can see four different perspectives on portions of the UML metamodel. Going from left to right they are: (1) package diagram of base UML packages, (2) class diagram of “StructuredClassifiers” package, but without relationships, (3) diagram of all superclasses of the UML *Class* entity, and (4) the same superclasses in a textual form. A user can click on any textual or visual entity to navigate into further detail or throughout the model.

Such scenario is not unique, it is very common to want to see the same model from a different perspective or summarize some concepts. All four perspectives and other such like that were created during my implementation of the UML metamodel, and were immensely helpful in my understanding of much of the arcane technical aspects of the UML specifications.

Ultimately, the objective of OpenPonk is not just a single diagramming interface, but a comprehensive environment for programmers, analysts, students, and researchers to explore and *play* with their models and understanding.


Inspector on an OPUMLPackage('UML')
an OPUMLPackage('UML')
an OPUMLPackage('StructuredClassi...
an OPUMLClass('Class')
an OPUMLClass('Class')
Deployments
«import»
«Package» SimpleClassifiers
«Package» UML
«Package» StructuredClassifiers
«Package» Classification
Class
extension : Extension
isAbstract : Boolean
isActive : Boolean
nestedClassifier : Classifier
ownedAttribute : Property
ownedOperation : Operation
ownedReception : Reception
superClass : Class
Component
irectlyInstantiated : Boolean
agedElement : PackageableElement
ided : Interface
zation : ComponentRealization
ired : Interface
bleElementTemplateParameter
edElement : ConnectableElement
rameterableElement
NamedElemen
ackageableElement
Namespace
Type
Classifier
StructuredClassifier
BehavioredClass
ncapsulatedClassifier
Class

name	value
ParameterableElement	an OPUMLPara
NamedElement	an OPUMLNam
PackageableElement	an OPUMLPack
Namespace	an OPUMLNam
Type	an OPUMLType
TemplateableElement	an OPUMLTem
RedefinableElement	an OPUMLRede
Classifier	an OPUMLClas
StructuredClassifier	an OPUMLStru
BehavioredClassifier	an OPUMLBeha
EncapsulatedClassifier	an OPUMLEnca
Class	an OPUMLClas

Figure 2.4: Perspectives on a UML “Class” metamodel entity

Chapter 3

Magritte

Magritte is a meta-data description framework *[8]* used extensively in both OpenPonk and the NS modeler. Magritte provides a system of descriptors through which it is possible to describe and manipulate individual properties of domain objects.

3.1 Descriptions

Figure 3.1 shows a diagram of an example situation and the relationships between the involved classes, their instances, and their descriptions. Listing 1 then shows the actual code used to describe the attributes from the same example.

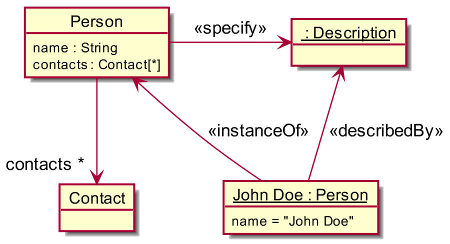


Figure 3.1: Person class specifies Descriptions that are used to describe Person's instances

Listing 1: Descriptions for name and contacts properties

```
Person>>descriptionName
 <magritteDescription>
 ^ MAStringDescription new
     accessor: #name;
     label: 'name';
     priority: 1;
     yourself.
```

(continues on next page)

(continued from previous page)

```
Person>>descriptionContacts
 <magritteDescription>
 ^ MAToManyRelationDescription new
     accessor: #contacts;
     label: 'contacts';
     priority: 2;
     classes: { Contact };
     yourself
```

All description methods are annotated with a *pragma*[1] *<magritteDescription>* and return an appropriate description object. Each description object is an instance of a particular type class.

In Listing 1 we see two such type classes[3]. *MAStringDescription* is a description of a string value, whilst *MAToManyRelationDescription* describes a collection-based relationship to instances of a *Field* class.

Figure 3.2 shows the hierarchy of all available Description types. Naturally, each description offers different API specific to its needs. *MAToManyRelationDescription* requires specification of classes it can accept, MASingleOptionDescription (typically represented with a droplist input) will need a list of permitted options from which the user can choose, etc.

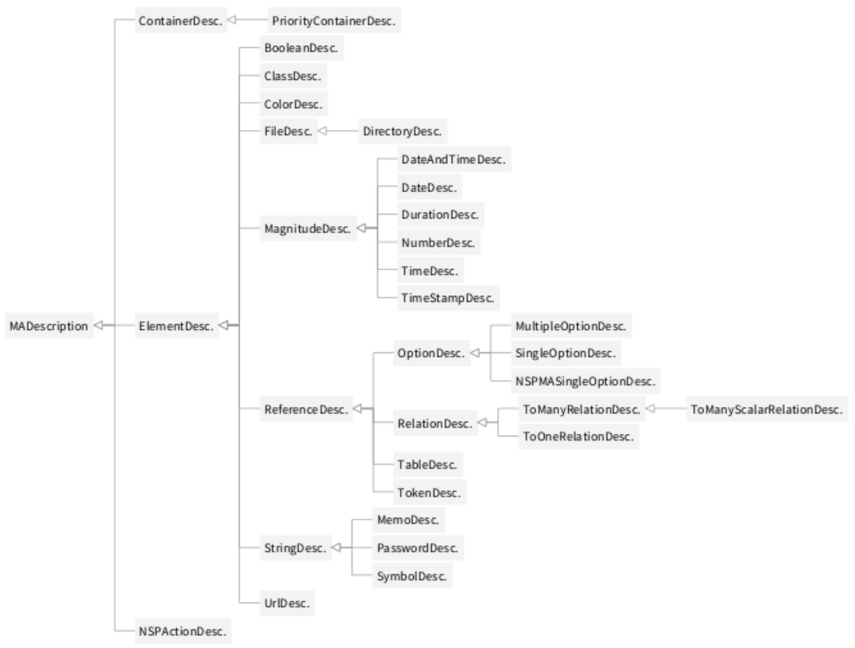


Figure 3.2: Hierarchy of Magritte Description types

[1] Pragmas are Pharo method annotation mechanism.
[3] For brevity I will omit Contact descriptions in this chapter.

3.2 Accessors

Magritte offers additional dimensions of flexibility. In the examples in Listing 1 the access to an instance variable was specified by a simple symbol (*accessor: #name*); this will use a getter (*name*) and a setter (*name:*) of the *Person* instance to read and write the real value. But this is not the only way how to specify the access path, as shown in Figure 3.3.

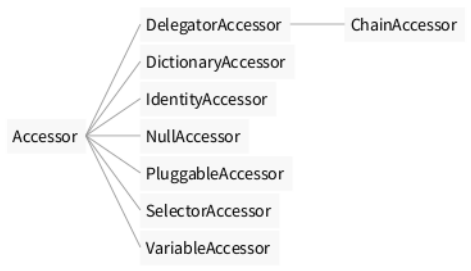


Figure 3.3: Hierarchy of Magritte accessors

Symbol used in the earlier examples is converted into a *SelectorAccessor*, but if no getter/setter is available, we could still utilize e.g., *VariableAccessor* which writes directly to the instance variables. Likewise, if the description is not defined directly on the object holding the data, a *PluggableAccessor* or a *DelegatorAccessor* can be used instead.

Listing 2: Using pluggable accessor to delegate

```
PersonProxy>>descriptionName
 <magritteDescription>
 | accessor |
 accessor := MAPluggableAccessor
     read: [ :proxy | proxy realObject name ]
     write: [ :proxy :newValue | proxy realObject name: newName ].
 ^ MAStringDescription new
     accessor: accessor;
     label: 'Name';
     priority: 1;
     beRequired;
     beNonEmpty;
     yourself
```

In Listing 2 a PluggableAccessor is used to read from and write to a different object (Person) than where the description is defined (PersonProxy). The option to delegate is particularly useful when the real object does not announce its internal changes, and we need to perform additional actions after the writing process.

3.3 Validations and Conditions

Specific description types by their nature limit the possible values that can be stored. However, a user will typically need a distinction even within the same type.

Listing 3: Additional conditions on #name description

```
Person>>descriptionName
<magritteDescription>
^ MAStringDescription new
   "..."
   beRequired;
   addCondition: [ :newName | newName first isUppercase ];
   addCondition: (MACondition selector: #isNotEmpty) labelled: 'The value is empty.'
   yourself
```

Listing 3 shows three different conditions/validations. The *beRequired* condition is a special condition provided for all description types, ensuring that a *nil* (empty) value cannot be set. The second condition is a *PluggableCondition*, where we can plug-in semantics of the test. The last condition is a simple condition which sends the provided selector to the value and expects a boolean return.

Additionally, conditions can be combined in boolean trees via additional condition classes show in Figure 3.4.

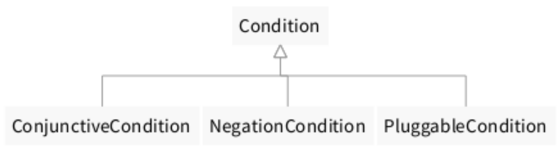


Figure 3.4: Available condition classes

3.4 Magritte Renderers

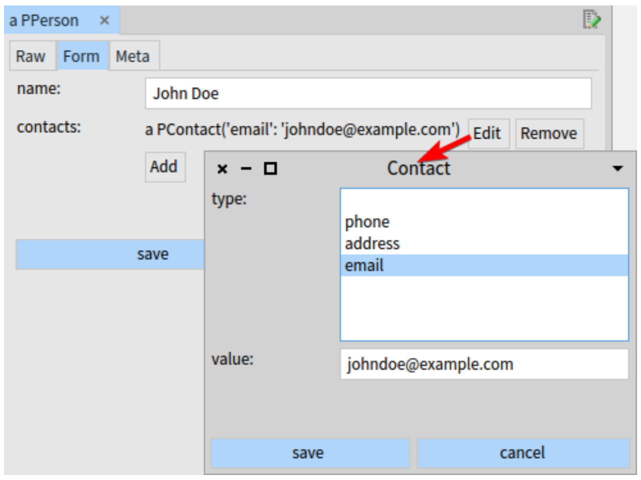
Magritte provides a Morphic[2] renderer, which is capable of generating an interactive form from the descriptions. For our *Person* example, such a form is shown in Figure 3.5. For contacts, which is a *ToManyRelation* description, buttons Add/Edit/Remove were added to manage the referenced elements. Here also comes to effect *priority* specified in the descriptions, as they determine the order of the entries in the form.

Another renderer specific for OpenPonk and the modeler is shown in chapter *Magritte Extensions*.

3.5 XML-Bindings Extension

XML-Bindings *[9]* is a Magritte extension that provides XML/Object mapping (X/O mapping) functionality – conversion between object graph and XML serialization and vice versa.

[2] Morphic is a low-level widget framework used by Pharo.



name: John Doe
contacts: a PContact('email': 'johndoe@example.com')
type: phone / address / email
value: johndoe@example.com

Figure 3.5: Magritte form with a subview

XML-Bindings extends Magritte description objects where it stores information about the strategy for storing and reading the description values.

Each class holding descriptions needs a class-side method which defines a XML element name under which instances of the class will be serialized (Listing 4).

Listing 4: Specification of XML element names

```
Person class>>xmlElementName
   ^ 'person'

Contact class>>xmlElementName
   ^ 'contact'
```

Then, in each description that we want to serialize, a strategy must be chosen (Listing 5).

Listing 5: Specifying serialization strategy

```
Person>>descriptionContacts
 <magritteDescription>
 ^ MAToManyRelationDescription new
    "..."
    beXmlElement

Person>>descriptionName
 <magritteDescription>
 ^ MAStringDescription new
    "..."
    beXmlChildAttribute;
    xmlAttributeName: 'value'
```

Serializing an object using these descriptions produces the XML document in Listing 6. Both *contact* attributes were produced via *beXmlAttribute*, and if needed a *beXMLElementCData* is available. The order of elements and attributes once again utilizes the *priority* property. In fact, XML-Binding can be a considered as yet another Magritte renderer.

Listing 6: Example XML produced by Magritte X/O

```
<person>
   <name value="John Doe"/>
   <contacts>
      <contact type="email" value="johndoe@example.com"/>
   </contacts>
</person>
```

To summarize, Magritte offers a systematic description of domain objects which are used by automatically generated user interfaces and reflective processing.

Part II

Analysis and Reverse Engineering of Metamodel

Chapter 4

Normalized Systems Metamodel

This chapter is a description of my contemporary understanding of the technical details of Normalized Systems metamodel.

My understanding is built primarily by reverse engineering XML files and other artifacts produced by the Prime Radiant (further described in chapter *Metamodel Engineering*). Of course I was already familiar with the theoretical perspective (see chapter *Normalized Systems Theory*), however here I focus on the precise *technical* details. For several NS entities I draw a comparison to UML *[10]* model entities and properties.

For better organization, the metamodel is here into three parts: component metamodel, application metamodel, and flow metamodel.

4.1 Component Metamodel

Component metamodel (Figure 4.1) is responsible for representing the structural aspect of NS models.

4.1.1 Primary Elements

A view limited to the primary entities and their relationships is shown in Figure 4.2.

The primary entities are as follow.

Component An organizational entity collocating related elements and their flows. Although it is comparable to a package, Component can contain Data Elements with different package names, but not other Components.

Data Element Main structural element encapsulating a concern. Data Element is expanded into a cluster of classes bearing the same name, each responsible for a particular aspect. Comparable to a UML Class or Entity in an ER diagram.

Field An attribute of a Data Element. Comparable to a UML Property (in the role of an attribute). It can be one of three categorical types.

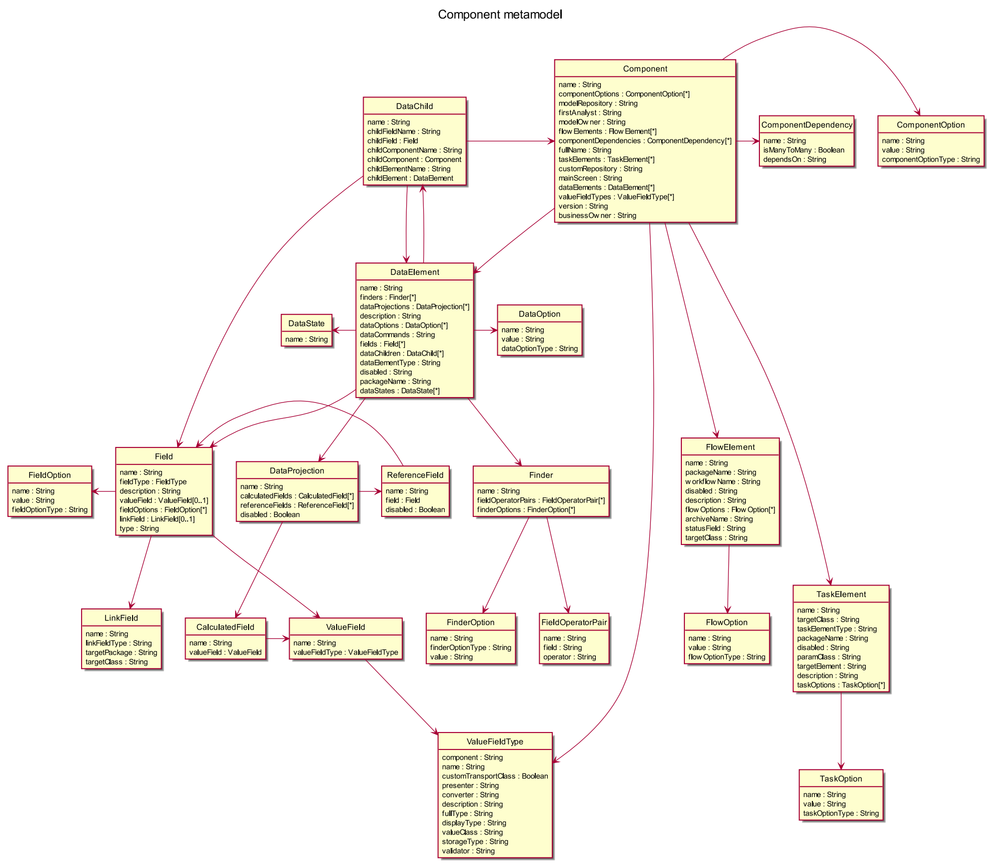


Figure 4.1: Component metamodel

A *value* field is a basic attribute with a specified type. A *calculated* field, comparable to UML *derived*, typically represents a value through a computational method – the value is *computed* on demand. Note that calculated field is represented as a Value Field, only the field type is different. Lastly, a *link* field marks the attribute as a relationship.

Link Field An extension of a Field that specifies another Data Element to which the Field's own Data Element is connected via a relationship. Link Field Types determine the cardinality:

- "one to many" (Ln01) and its inverse "from many to one" (Ln05)
- "many to many" (Ln03) and its inverse "from many to many" (Ln06),
- Ln02/Ln04 with the same semantics as Ln01/Ln05, but at runtime managed as low-level Java collections, and not at the conceptual level of NS software

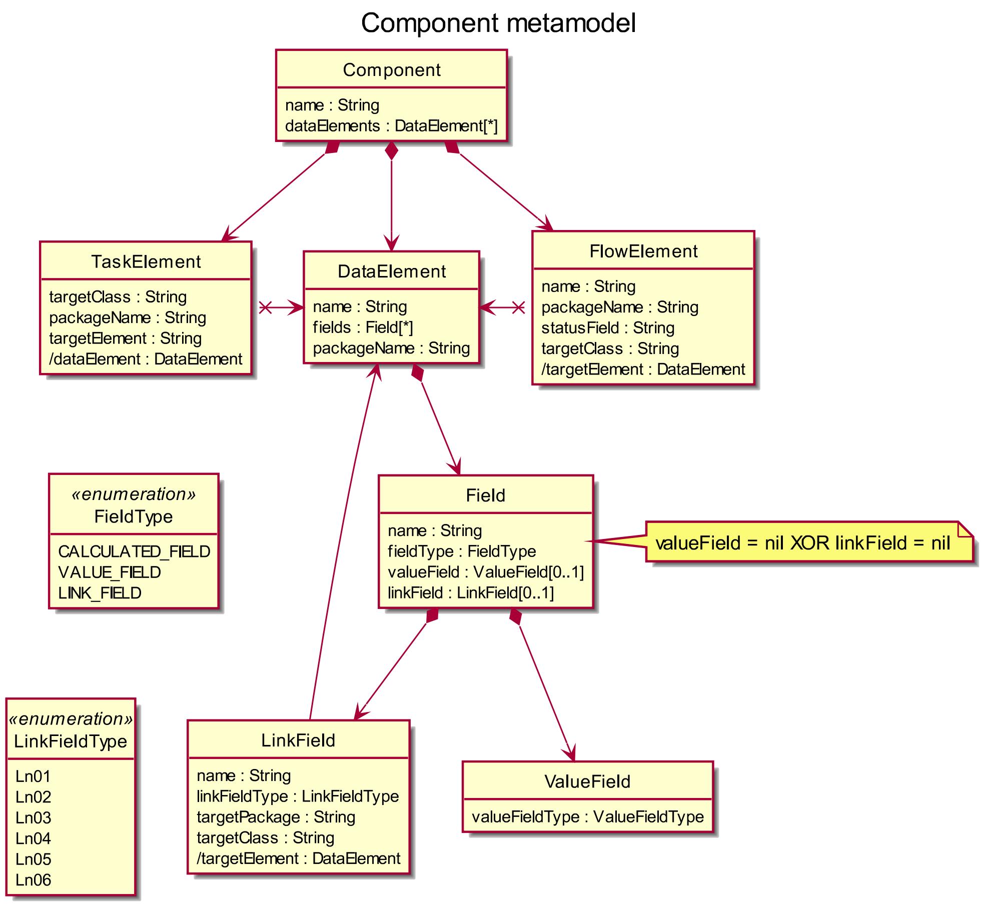


Figure 4.2: Component core

Note that the inverse relationship always requires the opposite (the "base" relationship) to be specified.

Flow Element and Task Element Discussed in section *Flow Metamodel*.

4.1.2 Non-Primary Elements

The remaining elements provide additional configuration (e.g., Data Options, Field Options) and support structure.

Finder Reifies a search query on a specified combination of fields using which Data Elements can be searched for. Crucial, as most relationships are stored as named strings. For example in Figure 4.2 both Flow Element and Link Field locate the target Data Element by lookup based on the specified *targetPackage* and *targetClass* String fields.

Data Child Marks an existing link field as a parent-child relationship. By default, this has no impact on the semantics of the relationship (comparable to UML Property aggregation), however it is possible to configure it so the lifecycle of the child element follows the parent (UML Property composition). Data Child is used primarily to provide hierarchical user interfaces (so-called *waterfall* screens).

Data Projection A virtual-like (or an aggregate) Data Element that selects several value and calculated fields of a Data Elements and possibly combines them with *calculated* fields of its own.

Data State A state in which a Data Element can be at runtime. The set of specified Data States determines what Flows can be designed in the model. However, at runtime the Data Element's state is not limited.

4.1.3 Other Elements

The component metamodel contains many more elements not described here such as Commands, Service Elements or Value Field Type specifications. At the time of writing, operating on these elements is not supported by the modeler, nor have I thoroughly familiarized myself with their meaning. Their support will be gradually added based on the demand for them; e.g., support for Value Field Types is already underway.

4.2 Application and Project Metamodel

4.2.1 Application Elements

Application metamodel (Figure 4.3) represents organizational information about the application, such as its metadata, components that are to be considered part of the application, and most importantly DataFlowTasks, which contain the behavioral aspect of NS models.

Application Description data for an NS application.

Data Flow Task and Workflow See *Flow Metamodel.*

Component *Not to be confused with Component from Component model.* Specifies component names and versions that are considered part of the application; i.e., the components upon which the application depends.

4.2.2 Project Elements

The Project metamodel (Figure 4.4) is used to collocate all models related to the same application. This is a required OpenPonk interface implemented by the *NSProject*. Also, the components and flows are aware of each other through the NSProject.

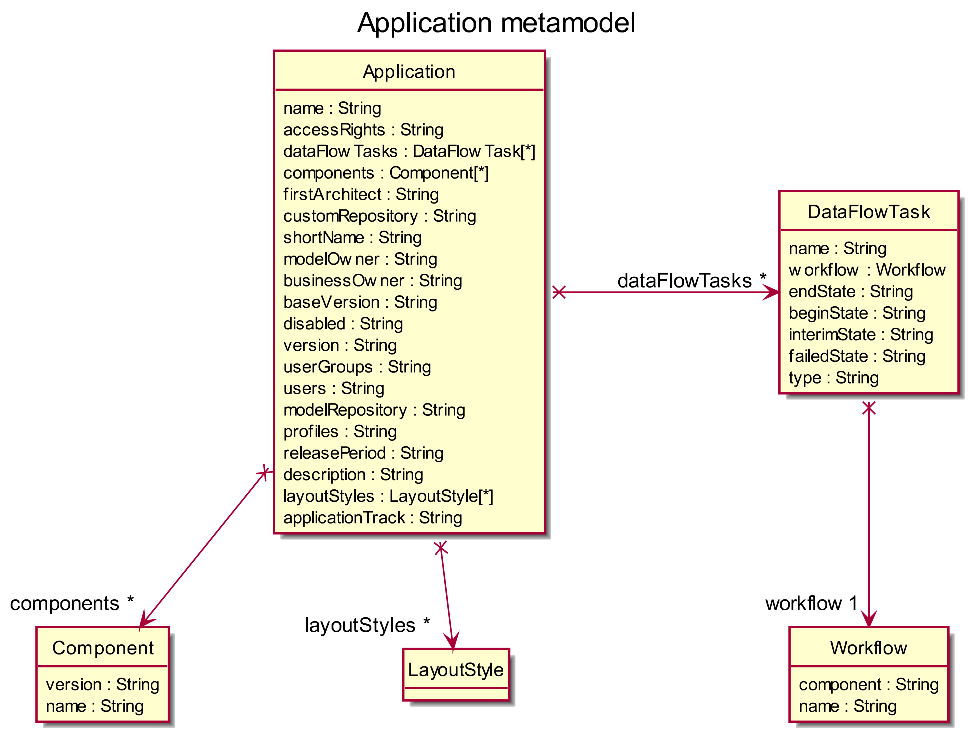


Figure 4.3: Application metamodel

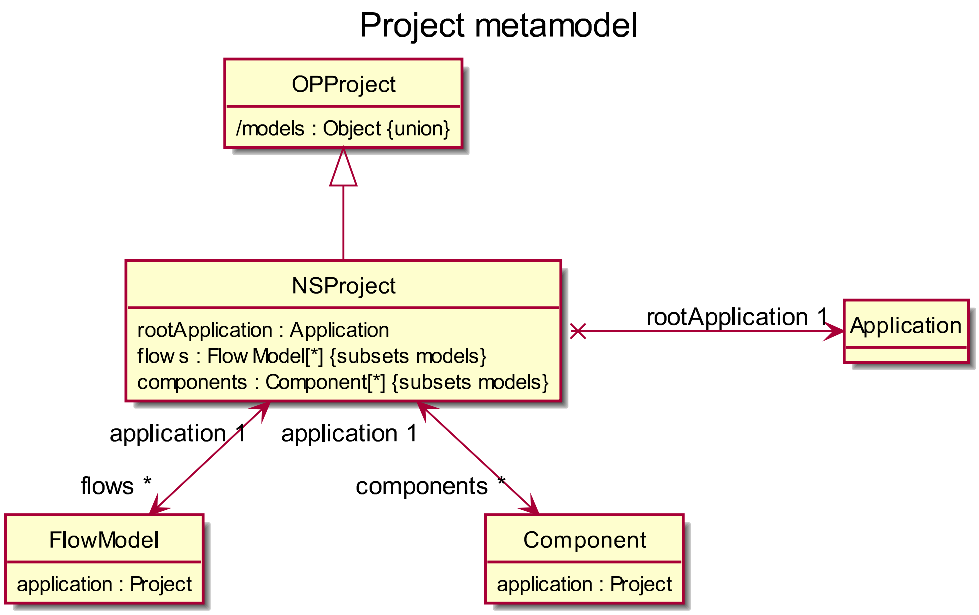


Figure 4.4: Project metamodel

OPProject OpenPonk base class representing a project tied to a single Workbench. It provides a collection of all models within the project.

NSProject OPProject subclass linking all primary models together – the Components (section *Component Metamodel*), Flows (section *Flow Metamodel*), and the root Application (subsection *Application Elements*). Via NSProject Flow is connected to its Data Element in the proper Component and vice versa.

4.3 Flow Metamodel

Flow metamodel (Figure 4.5) is a modeler-specific metamodel for flow models. The Application already contains the basic flow information in the form of DataFlowTasks. But in practice, a more natural representation is needed. Furthermore, The Application XML (represented in the Application metamodel) is not the only possible source of DataFlowTask descriptions. Thus a different representation is needed regardless. Likewise, Task Elements and Data States that are present in the Component and Data Element where the Flow is running should still be represented, even if no DataFlowTask is currently operating on them.

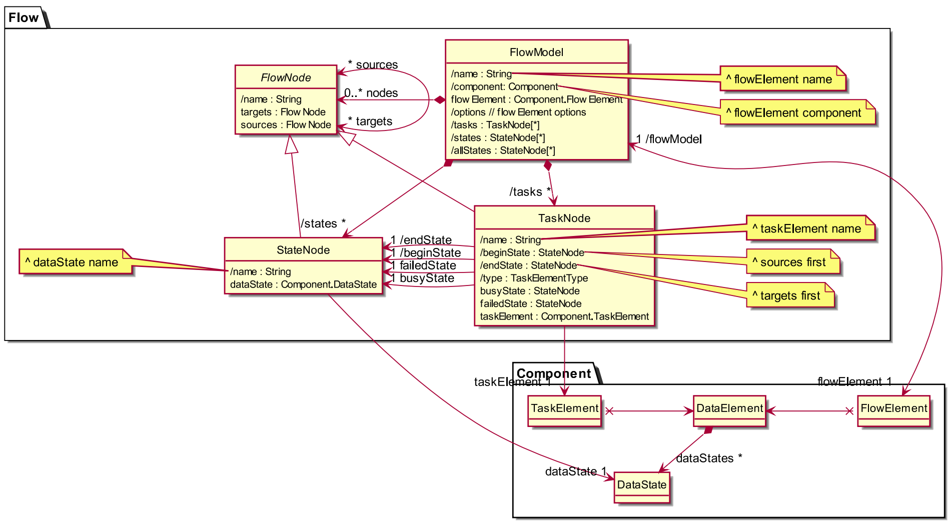


Figure 4.5: Flow metamodel

The Flow model is positioned orthogonally to the Component metamodel. Each concrete entity in the Flow metamodel is backed by a structural entity from the Component metamodel. The metamodel itself is capable of representing arbitrary graph structure, as all incoming and outgoing (sources, targets) links are multivalued. Note that targets and sources of one subclass always point to elements of the other subclass – State Nodes and Task Nodes alternate.

Flow Model Container for a Flow and its states and tasks. Operates on a Flow Element from a Component.

Flow Node Base class for connectable objects in the flow.

State Node Entity representing a state in which a Data Element can be; is linked to Data Element's Data State. State in the role of *busyState* is used when a Task is being performed. State in the role of *failedState* is used when Task execution has failed.

Task Node Entity representing a Task Element that is performed during a transition between states. The task node has references to a *busy* and *failed* states with abovementioned semantics. Figure 4.6 shows a single NS step of state processing.

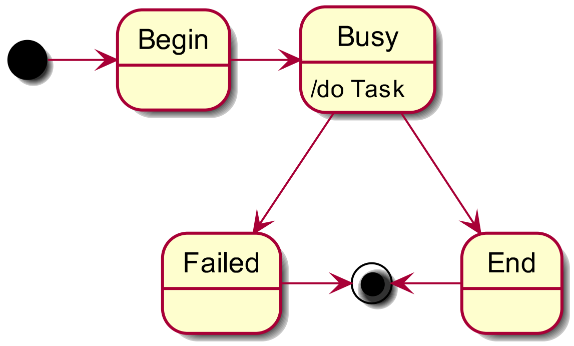


Figure 4.6: Basic NS flow step

4.3.1 Advanced Flows

The flow model in its current form is used to represent simple state-task-state steps. Although this is the most common usage, it is far from the full power of NS flows. Several extensions follow that are either in the development or are subject for future elaboration.

1. Busy and Failed States were likely (in my understanding) conceived as a technical detail for NS systems. However, they are sometimes used as regular *domain* states. For example, a task *InvoiceSender* branches into two domain states (Sent / Failed to Send) and uses its failed state for a genuinely unexpected situation (Invoice is invalid). But with the new approach, the "Failed to Send" state "Invoice is invalid" become merged. This is a work-in-progress feature.

2. The Failed State and the End State as described in the metamodel are not the only states where a Data Element can end after performing a Task. With *branching* Task Elements, the implementation of the Task returns the name of the next state. The modeler can represent it through multiple *targets* of the Task Node. Unfortunately, it cannot be expressed in Prime Radiant Data Flow Tasks while preserving the *branching* semantics.

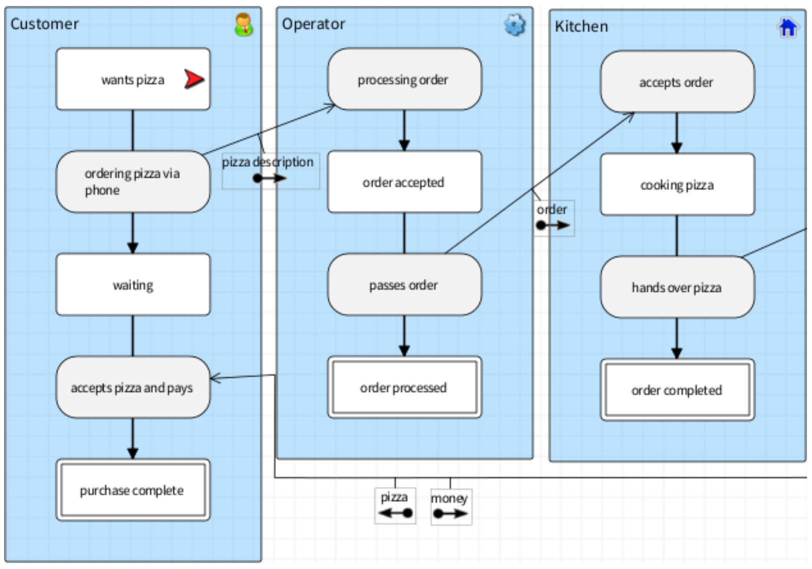


Figure 4.7: BORM diagram

3. A single Flow always operates only on a single Data Element. However, multiple Flows operating on different Data Elements collaborate through *updater* and *bridging* Task Elements.

   Supporting and visualizing such functionality was discussed only in passing, but Business Object Relation Modeling (BORM) *[11]* is an interesting direction of exploration due to its similarities to NS flows at both the metamodel, as well as diagramming level (see Figure 4.7).

   Participants (blue rectangles) are equivalent to NS flows, states (boxes) and activities (rounded boxes) alternate with the same semantics as states and tasks in NS flows, and the participants communicate via communications between activities in a similar fashion to updater/bridging communication between NS Task Elements.

4.4 Option Types

Many entities in the component metamodel can be provided with a collection of options. E.g. DataElement has DataOptions, Field has FlowOptions, etc. Figure 4.8 lists all known option types. Note that several options (e.g. *DataOptionTypes::hasDisplayName*) are marked with a *{value}* property. This property is marking types that require an Option to provide a specific string or numeric value. Non-marked options are considered binary flags. As there is no definite list to determine which options are binary and which are value-based, the information shown in the model is based on observations of existing components.

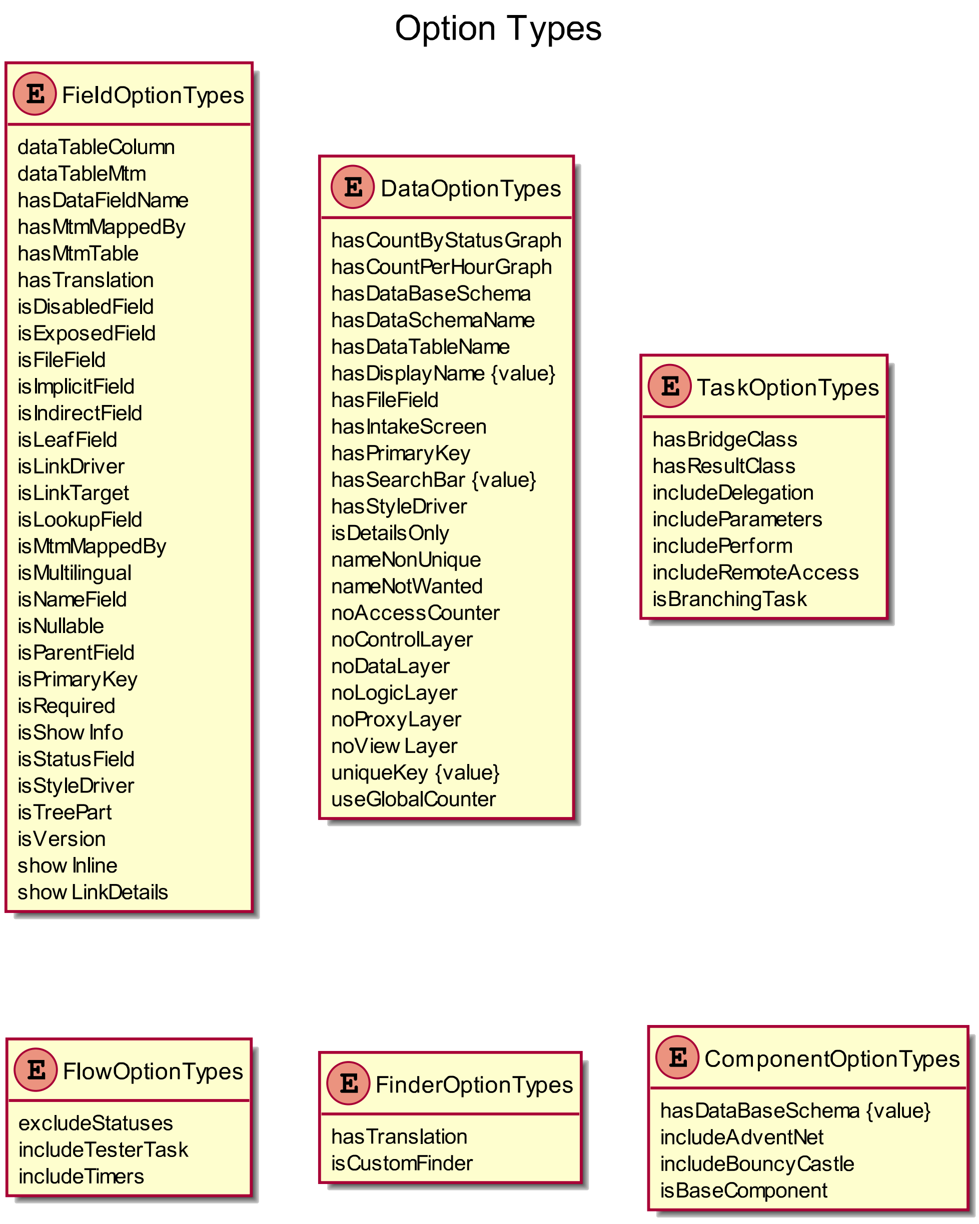


Figure 4.8: Option Types

Chapter 5

Metamodel Engineering

This chapter presents and overview of efforts and challenges associated with reverse-engineering artifacts provided by Prime Radiant, and engineering the current metamodel as described in *Normalized Systems Metamodel*.

5.1 XML files

The primary interchange artifacts produced by Prime Radiant are XML files containing descriptions of an NS system. The XML files are of two types. The Application XML file contains general metadata pertinent to the application, flow definitions, as well as listing of all components used in the application. The Component XML file contains the remaining model elements, such as Data Elements, Flow Elements, Task Elements, custom Value Types, Finders, etc.

Neither XML has a XSD schema[1] nor documentation describing their exact structure or meaning. Therefore an analysis of both XML files was required to infer a metamodel usable for representing and manipulating the models contained within.

5.2 XML Analysis

The approach to analysis and XML processing was constrained by several factors that all had to be addressed.

1. Already mentioned lack of technical information about the structure and relations within the XML documents, as well as my lack of knowledge of those relations.
2. Lack of schema support in Pharo's XML libraries.

[1] XSDs discussed in *X/O mapping and XSDs* were created much later, and are not yet used for metamodel regeneration.

3. Lack of dedicated XML/Object mapping (X/O mapping) libraries even in if I had XML schemas at my disposal.
4. Initial lack of access to Prime Radiant.
5. Incomplete XML files, as only a subset of metamodel is utilized by any NS system.
6. Ability to relatively quickly reanalyze XML files when new content is added or a mistake is discovered.

Particularly constraint (6) implicated either a fully or a semi-automated solution.

To resolve this, a new tool named *XML Magritte Generator* was created.

5.3 XML Magritte Generator

The responsibility of this tool is to take an arbitrary XML document, make an educated guess about the relationships and types within the model, generate classes and/or attributes in the document, and generate Magritte descriptions (see *Magritte*) for everything. Furthermore, it utilizes Magritte extension *XML Bindings*, which is capable of providing usable[2] X/O mapping when the Magritte descriptions and the classes containing them are properly annotated.

5.3.1 Analysis

The analysis part of the tool is demonstrated on a simple XML document in Listing 1.

Listing 1: Example XML document

```xml
<machine id="12">
    <version value="1.2.7" />
    <anyOnline>false</anyOnline>
    <nodes>
        <node id="node-1">
            <isOnline>false</isOnline>
            <address>
                <host>127.0.0.1</host>
                <port>80</port>
            </address>
        </node>
    </nodes>
</machine>
```

The analyzer performs a three-step classification to determine relationships and types for both XML attributes and elements.

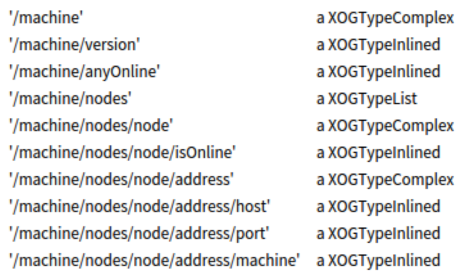
The first step decides what is the relationship between an element and its parent as seen in Listing 2 and Figure 5.1.

[2] This X/O mapper doesn't play well with arbitrary XMLs and extra steps were needed for the processing.

Listing 2: Element type classification

```
types := XOGTypeClassification new classificationFor: doc.
```

The result is a map between element's XPath and a Type.



```
'/machine'                                   a XOGTypeComplex
'/machine/version'                           a XOGTypeInlined
'/machine/anyOnline'                         a XOGTypeInlined
'/machine/nodes'                             a XOGTypeList
'/machine/nodes/node'                        a XOGTypeComplex
'/machine/nodes/node/isOnline'               a XOGTypeInlined
'/machine/nodes/node/address'                a XOGTypeComplex
'/machine/nodes/node/address/host'           a XOGTypeInlined
'/machine/nodes/node/address/port'           a XOGTypeInlined
'/machine/nodes/node/address/machine'        a XOGTypeInlined
```

Figure 5.1: Element types

The meaning of the element types is as follows.

Any The type has not been determined yet. *Any* is an internal type and does not appear in the output.

Complex Element should be represented by a Class because it contains a complex structure (other elements or multiple values).

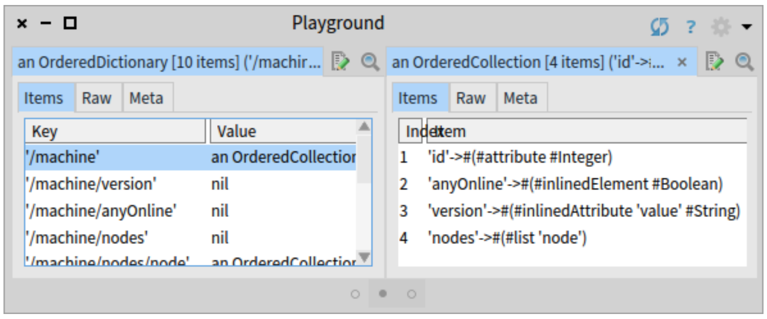
Inlined Element contains only string nodes. It will be represented as an attribute of the parent. Parent's type is expected to be *Complex*.

List Element's child elements are all of the same type and share the same name. The element therefore acts as a container for a list and will be represented as a collection-based attribute.

The type is checked against all elements with the same XPath and the type with the highest priority (Complex > List > Inlined > Any) is picked. Theoretically a situation could arise, where different types are actually needed. That has not been encountered in the analyzed XMLs, thus it is not given any further consideration.

Second step of the analysis is guessing the value types of element attributes and XML string nodes. At present only Boolean, Integer, Float, Number, and String value types are considered and in the same ascending priority. Naturally, this can lead to ambiguous definitions, such as is "1.0" being either a Float or a String. As the type depends on the target domain, it cannot be determined automatically, but the tool provides an option to manually override types.

The last step combines information from the previous two steps and creates another map that describes all complex types. Non-complex types do not require creation of classes, so they contain *nil* for description.



an OrderedDictionary [10 items] ('/machir...

Key	Value
'/machine'	an OrderedCollection
'/machine/version'	nil
'/machine/anyOnline'	nil
'/machine/nodes'	nil
'/machine/nodes/node'	an OrderedCollection

an OrderedCollection [4 items] ('id'->...

Index	Item
1	'id'->#(#attribute #Integer)
2	'anyOnline'->#(#inlinedElement #Boolean)
3	'version'->#(#inlinedAttribute 'value' #String)
4	'nodes'->#(#list 'node')

Figure 5.2: Complex type classification

As can be seen in Figure 5.2, some elements (here namely *<version value="1.2.7">*) have been inlined (*#(#inlinedAttribute 'value' #String)*) to form a single attribute *value* in the parent complex element. This is an intentional effort to reduce the number of model entities which would not bring any additional value to the domain. If new attributes or subelements were added, then regeneration of the model would properly create an independent class.

The result of the analysis also contains the *location* of an attribute or an element. It is necessary for accommodating accurate X/O mapping.

5.3.2 Generation

The generation of the metamodel is straight-forward. A class is generated for each *complex* type. A class attribute, accessors, and Magritte description is generated for each *analyzed*[3] attribute.

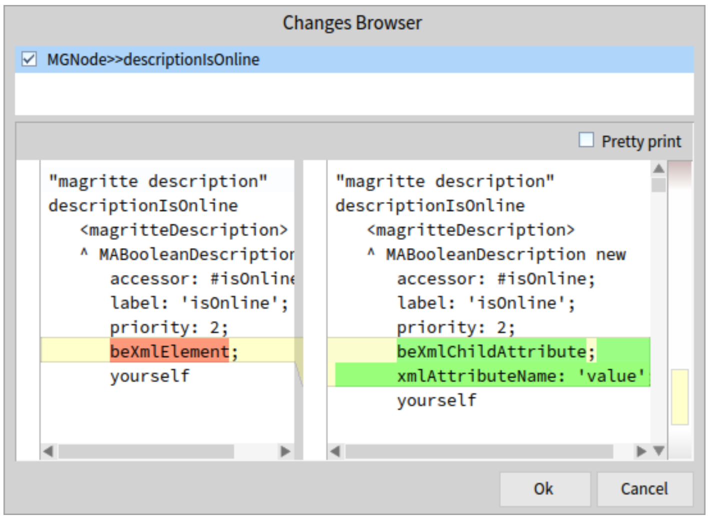
Listing 3: Generating the metamodel

```
"doc is <machine> XML from earlier examples"

gen := XOGStructureGenerator new.
"optional attributes, they have their default values"
gen packageName: 'MachineGenerated'.
gen classPrefix: 'MG'. "to avoid name conflicts"
gen rootClassName: 'MXObject'. "a class from which all the classes will inherit"

"run the generator; this will NOT create the code yet"
gen processDocument: doc.

"retrieve CBChangesSet, so additional modifications can be performed"
(CBChangesBrowser changes: gen changes refactoringChanges) open.
```

Generating the metamodel shows a simple code requesting the actual generation. To demonstrate earlier claim that elements with string nodes and elements with single attribute are identical, I've changed in the original XML example the element *<isOnline>false</isOnline>* to *<isOnline value="false"*

[3] I.e., attribute, element with string nodes content, or inlined element with a single attribute.

/> and regenerated the model. As Figure 5.3 shows, the only difference is change to the X/O mapper configuration.



```
"magritte description"
descriptionIsOnline
    <magritteDescription>
    ^ MABooleanDescriptior
        accessor: #isOnline
        label: 'isOnline';
        priority: 2;
        beXmlElement;
        yourself
```

```
"magritte description"
descriptionIsOnline
    <magritteDescription>
    ^ MABooleanDescription new
        accessor: #isOnline;
        label: 'isOnline';
        priority: 2;
        beXmlChildAttribute;
        xmlAttributeName: 'value'
        yourself
```

Figure 5.3: Diff after a small change to the XML

Using a code diff works well as a substitute for a fully-fledged model diff tooling. Furthermore, model diff would focus primarily on the domain, however in this instance knowing the precise technical difference is required, as custom overrides and code is continuously being added to the metamodel implementation as new understanding is gained.

5.4 Component Analysis

Despite its simplicity, The XML-Magritte-Generator tool has proven very effective. Even though a significant amount of code has been added throughout the NS development, it is still possible to perform regeneration, albeit increasingly more care is required when the tool attempts to alter existing code. For new entities whose support is only about to be introduced, the tool easily accommodates for most of the newly introduced metamodel code.

The main lacking feature of the generator is no support for horizontal references between entities. Entities often refer to each other via a combination of several attributes (e.g., package name and element name), or they contain an aggregated name that has to be expanded before the target model entity can be searched for.

This problem is side-stepped by a pair of transformations that happen after a model is materialized from XML (reading), and before it serialized back into the XML (writing). The reading transformation according to defined rules processes all model entities, connects them to a single, coherent model, and possibly transforms some values into a more object-appropriate form. The writing transformation performs the opposite – during its operation, the modeler interacts with the model primarily via its relationship links, thus it is necessary to *break apart* the links into a form that can be persisted, as well as adds or updates aggregate names that are used throughout the XML file.

The combination of X/O mapper taking care of the majority of issues and the bidirectional transformation handling special cases has so far been sufficient to fully manage the content of the both XML files.

5.5 X/O mapping and XSDs

The X/O mapping provided by Magritte's *XML-Binding* has its blind spots. Notably, the ordering of the elements is based on the priority of the Magritte descriptions. Unfortunately, this priority has been modified by the modeler for its own purposes and therefore no longer usable by the X/O mapper. Furthermore, the X/O removes missing (i.e. *nil* or empty) elements and attributes from the output. Finally, NS XML files use custom approach to represent nil (null) values.

Different element and attribute ordering or removing empty nodes is problematic not only semantically – Prime Radiant could misinterpret some omission, but it also produces textual changes even if the model remained unchanged[4] . This in turn confuses Version Control Systems (VCS) and make it hard, if not practically impossible, to trace any changes made in the XMLs.

To address these problems, I have created XSD documents for both the component and application XML.

- Each XSD defines the desired element ordering via its *<xs:sequence>* element.
- For attribute ordering, the order is based on the order definitions in the XSD itself – there's no explicit ordering element for attributes, as XML considers attribute ordering not significant.
- Nillable elements are represented as such with *nillable=true* (e.g. *<xs:element name="modelOwner" nillable="true">*).
- The attribute requirement is specified via the *use* attribute (*use="required"* vs *use="optional"*), although at the time of writing, all observed attributes were *required*.

Information in the XSDs is used in transformations applied on the XML documents produced by Magritte's X/O. *Ordering* transformation takes care of element/attribute reordering. *Missing* transformation ensures that missing *required* attributes are present, and that missing *nillable* elements are represented with *undefined="true"* attribute.

[4] The XMLs produced by both modeler and PR were consistent with themselves, but not with each other.

As a closing note on XSDs, most (if not all) information available in the XSD is similar to the information gained from the XML analysis used for model regeneration. Thus it is desirable to eventually update the generator to be able to process provided XSDs and skip (or complement) its own type and value type inference.

5.6 Workflow Analysis and Transformation

Workflows are represented in the Application XML file in the form of Data Flow Tasks. Both the XML code (Listing 4) as well as the model (Figure 5.4) seem simple on the surface.

Listing 4: DataFlowTask XML fragment

```xml
<application name="Tutorial App">
  <dataFlowTasks>
    <dataFlowTask name="InvoiceSender">
      <workflow component="tutorialComp" name="Invoice" />
      <endState name="Sent" />
      <beginState name="Initiated" />
      <interimState name="SendingBusy" />
      <failedState name="SendingFailed" />
    </dataFlowTask>
  </dataFlowTasks>
</application>
```

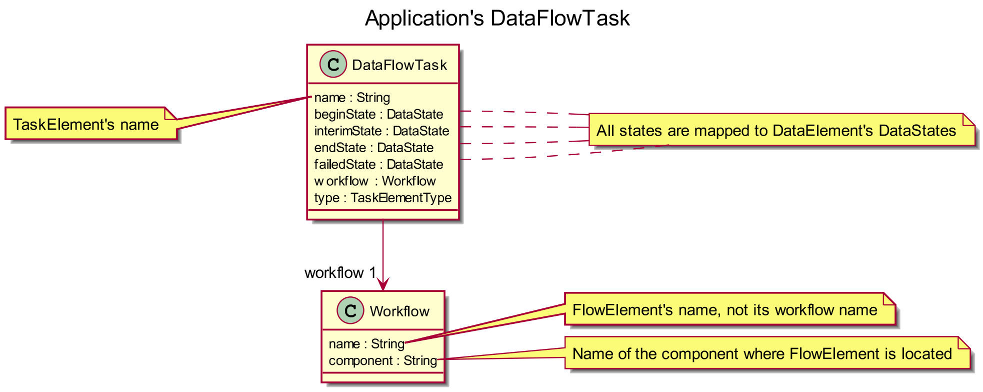


Figure 5.4: UML diagram of a DataFlowTask

Each *<dataFlowTask>* node represents a single step through a flow. That is, only going from *beginState* to *endState* and executing the task during the transition. To get the full flow, all DataFlowTasks with the same workflow must be grouped together.

The Data Flow Tasks, however, only reference by name the actual model entities that are involved in the flow – all of them part of the component metamodel (shown in Figure 5.5), as well as the Flow metamodel (shown in Figure 4.5 in the Chapter *Normalized Systems Metamodel*). As the location is effectively a horizontal reference, it is performed during the reading/writing transformation already described.

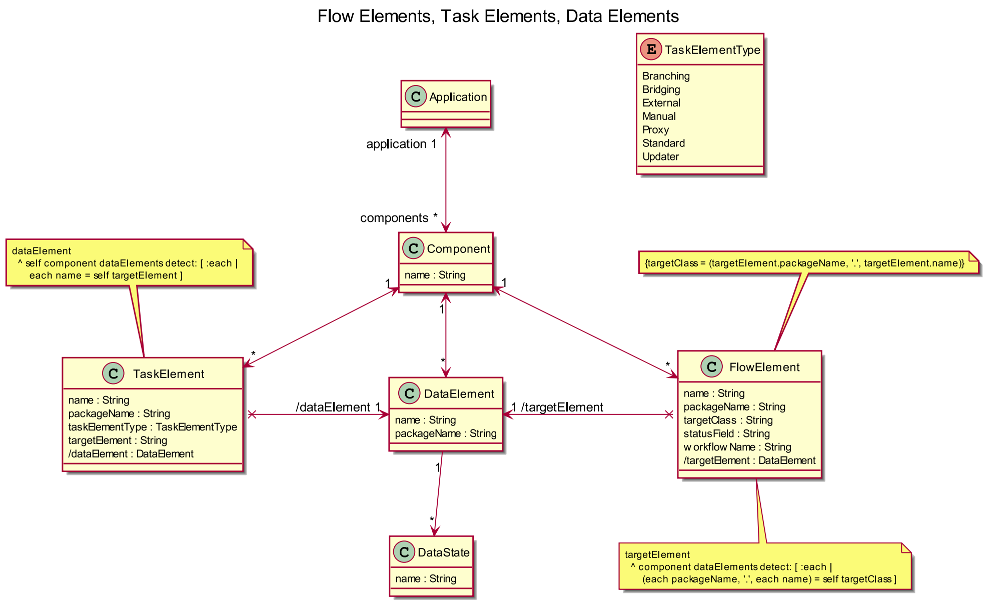


Figure 5.5: Flow Elements, Task Elements, Data Elements

In practice, this part of the model has proven to be the most problematic, as every single reference in the Data Flow Task can be missing. From the Component to the last Data State. Appropriate measures are slowly being taken to help users reasonably address such situations.

In summary, through a combination of detailed analysis, code generation and custom transformation it was possible to achieve near-exact (except for not yet supported content) match with the artifacts in Prime Radiant. However, I do not consider this situation satisfactory from a long-term perspective. A canonical source of truth for the metamodel definition should be provided to provide for a well engineered, and well maintainable interchange of data and representation of systems.

Chapter 6

Rules and Validations

6.1 Model Validations

Every experienced programmer is familiar with code validations and code. Whether in the form of compiler errors and warnings, external static analysis linters, or hints provided by an IDE. They all assist the programmer, help them find problems in their code, and even improve their code bases by suggesting refactoring changes clarifying their intent.

Such suggestions range in their value, scope, severity, and even the options to fix them.

Of course, not everything reported is an actual issue. A typical example in many programming languages is the usage of an assignment operator inside an if or a while statement: *while (a = queue.first()) { ... }*. A reasonable linter would report this as a *potential* problem, but it cannot decide whether it is a bug or not. Thus it is vital for any such warning to include helpful information so the user can address it adequately.

Such warnings are not limited to just code. The same principles, requirements, and demands apply to conceptual models as well. Code syntax is replaced with the metamodel structure, and unlike the code, the important of semantical correctness is increased.

This chapter presents model validations introduced in the NS modeler.

6.2 Validation Rules

At the time of writing, 38 validation rules are implemented. These rules range from checking simple typos in names, through warnings about missing elements, to structural validations across several model entities.

The table in Table 6.1 shows an excerpt[2] of the rules, they were chosen for they are referenced throughout this thesis. The *Class Name* column contains the name of the class implementing the rule. *Description* provides a detailed

[2] . The table is shortened for brevity, but the full list is readily available to NS modeler users.

description of the situation. The last column – *Example*, is generated by running tests on example cases to mimic what an actual user would see in their validation reports.

Table 6.1: Validation Rules

Class Name	Description	Group	Severity	Examples
AppBaseDependencies	Base dependencies should be added for an application.	Missing defaults	prError	Missing dependency "account". Missing dependency "utils". Missing dependency "validation". Missing dependency "workflow".
DataChildReference	Data childs should refer to existing components, elements, and fields.	References	error	Component unknownComponent is missing. Data Element UnknownElement in component unknownComponent is missing. Field UnknownElement::unknownField is missing.
DuplicateNames	No two elements in the same namespace (dataElements/taskElements/flowElements in a component, fields/finders in a dataElement, ...) can share the same name.	Naming	error	Invoice::fields::invoiceId is duplicate.
EmptyComponent	A component should not be empty.	Missing defaults	warning	Component testComp has no elements.
FlowElementDataElementNaming	There is no data element matching the flow element's name.	References	error	Data Element Invoice is missing for Flow Element Invoice.
PrimaryElementsWithFlows	Data Elements with flows should be in the Primary category.	Classification	warning	Data Element Invoice should be primary.
ReverseLinkField	A reverse link must be defined for Ln04 (Ln02), Ln05 (Ln01), and Ln06 (Ln03) link fields.	References	error	Ln05 link Customer::invoices is missing opposite Ln01 link.
TaskStatusElement	Each flow (i.e., primary element with a flow defined on it) should have a corresponding [Element]TaskStatus related to it.	References	prError	Data Element with flow Invoice is missing InvoiceTaskStatus element.

Group and *Severity* provide categorizations to organize and consume the rules. The severity translates as follows: *Error* will usually prevent an application from being built by the expanders. *Warning* may result in a problem during runtime, or it points to a technically correct, but unusual situation, such as mixing different package names within the same component. Finally *prError* is technically an *error* if a user tried to build an application directly from XMLs produced by the modeler, but if the user loads the XMLs into Prime Radiant, the system will automatically resolve them.

From a technical point of view, the modeler can be readily extended to automatically fix such prErrors. However, any such implementation has to weighed against duplication between the modeler and Prime Radiant. Apart from the double effort required, it creates a risk of behavioral misalignment, where each tool fixes the same issue in different, incompatible ways. This issue is a topic of ongoing discussion.

6.3 Describing Rules

The table Table 6.1 lists a wide range of complexity from simple "Data elements should start with a capital letter" rules that require nothing more than just the name to test, to more complex scenarios requiring the test of several model entities such as the *TaskStatusElement* rule.

However, just implementing a rule is rarely enough. We need a way to state *when* the rule should be applied, and *what* the rule *exactly* does. Both in a way that can be understood not just by the original author, but also by people not familiar with the codebase, but knowledgeable of the modeling (or modeled) domain, e.g., analysts.

6.3.1 Visual OCL

A possible approach that I have begun exploring is Visual OCL *[16]*. Visual OCL combines the technical precision and expressiveness of OCL and augments it with equal graphical representation. I will further use the Smalltalk language instead of the textual OCL language, as that is the language in which the rules are implemented, but the Visual OCL remains OCL-based.

What follows is a Visual OCL (VOCL) description of a TaskStatusElement rule. To better contextualize the VOCL, Figure 6.1 shows a fragment of the NS metamodel used and checked by the rule. We see that Data Elements are organized in Components, and that a Field that is a *link field* must have a *targetElement* pointing to the linked element. Finally, a Flow *can* be attached to a Data Element.

With the knowledge of the metamodel fragment, we can look at the VOCL in Figure 6.2. The *constraint* segment contains a Smalltalk code representation of the rule check. If the precondition is met (Data Element has a Flow), then the model graph is traversed and the required association is checked.

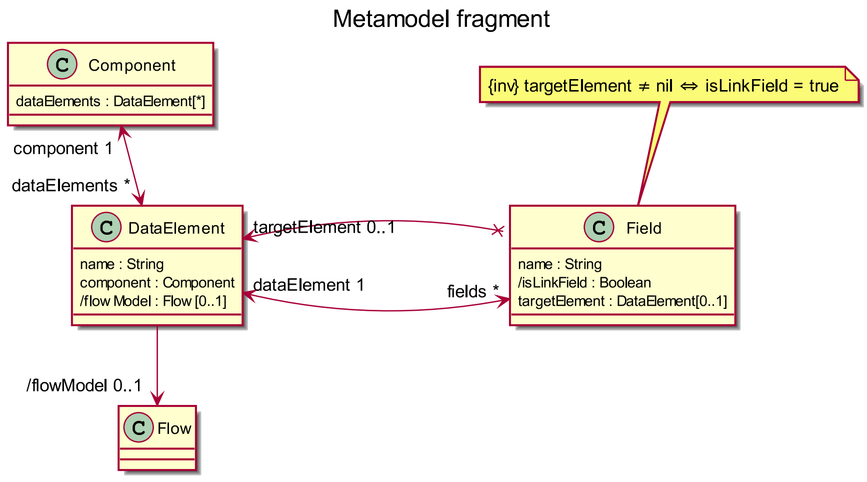


Figure 6.1: Metamodel fragment with elements used in a rule

The visual equivalent is expressed in the next two boxes of Figure 6.2, both using UML instance diagram notation. *Inv* (invariant) box contains instances of Data Element and Flow linked to each other, stating that these objects exist and are in a concrete (flowModel) relationship. *Implies* box contains an instance model of the desired outcome. There are two Data Elements in the same Component (they are both linked to the same instance). One Data Element has the same name as the one checked (self), but a "TaskStatus" suffix is added; its field is pointing back to the original checked Data Element just as required.

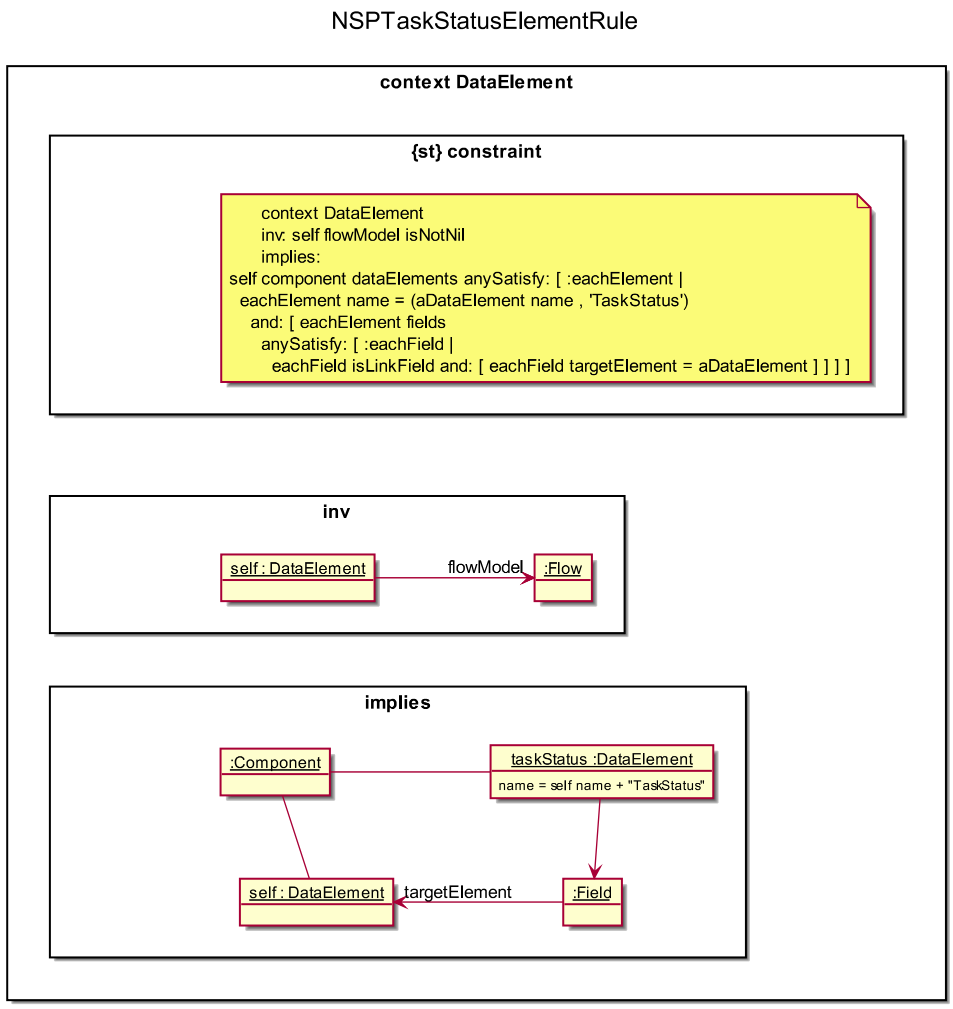
In this approach, we reverse the way we look at rules. Instead of describing how to test to rule, we show the desired outcome.

But to bring it into practice, a translation mechanism is required.

manual implementation The most straightforward approach is manually implementing every rule described by VOCL. But as with any manual implementation of a model-described structure, a real risk is introduced that the technical specifics will drift from the original intention, and the VOCL will become unmaintained.

code generation A more advanced approach is to create a code generator capable of transforming VOCL into code. At the moment it is unclear to me what effort would be required. A question opens up whether it would be possible to describe the rules in NS theory instead, as the rules are not technically complicated and are directly relevant to the NS metamodel. Then only the visual aspect of VOCL would remain.

validation engine Another possibility is a creation of validation engine that can directly consume the VOCL, or rather the underlying OCL. Many such engines already exist, but at least to some extent they are tied only to UML-based, and UML profile-base metamodels.



```
      context DataElement
      inv: self flowModel isNotNil
      implies:
self component dataElements anySatisfy: [ :eachElement |
  eachElement name = (aDataElement name , 'TaskStatus')
    and: [ eachElement fields
      anySatisfy: [ :eachField |
        eachField isLinkField and: [ eachField targetElement = aDataElement ] ] ] ]
```

Figure 6.2: Visual OCL on "Data Element with Flow should have a corresponding [Element]TaskStatus" rule

6.4 Renraku static analysis framework

In this section is an explanation of how are the rules currently written and tested.

The rules are written in the Renraku *[17]* framework. Renraku, among others, powers Pharo's internal static code analysis checker. I have chosen Renraku for its simplicity and familarity; it provides only the necessary facilities required for custom analysis and does not force its users to commit to an extensive framework.

The basic metamodel of Renraku is shown in Figure 6.3. A *Rule* is given a *Target* (e.g., a Data Element instance) and produces a number of *Critiques* for the checked Target. Each critique describes a single problem with the Target. As multiple checks in the same context are performed inside some rules, the number of produced Critiques can be higher.

An example of such situation is the *AppBaseDependencies* rule, where a critique is created for every missing dependency.

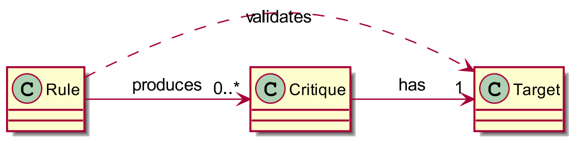


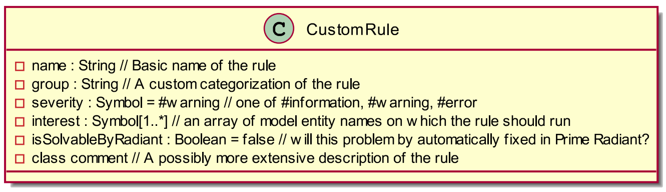
Figure 6.3: Renraku Metamodel

6.5 Rule Implementation

Each validation rule is encompassed in a *NSPAbstractModelRule* subclass. Each subclass is responsible for providing all the necessary metadata information about the rule (Figure 6.4), the actual code of the rule, as well as the information provided to a user.

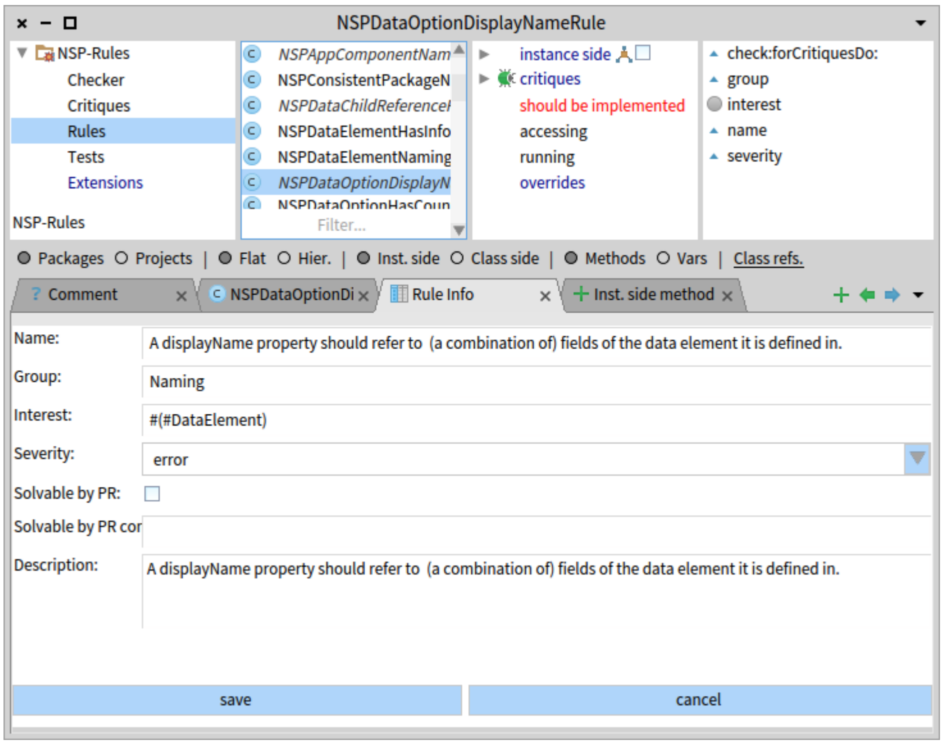
6.5.1 Metadata

Table 6.1 shows the metadata attributes that a rule class must provide.



name : String // Basic name of the rule
group : String // A custom categorization of the rule
severity : Symbol = #warning // one of #information, #warning, #error
interest : Symbol[1..*] // an array of model entity names on which the rule should run
isSolvableByRadiant : Boolean = false // will this problem by automatically fixed in Prime Radiant?
class comment // A possibly more extensive description of the rule

Figure 6.4: CustomRule



Name:	A displayName property should refer to (a combination of) fields of the data element it is defined in.
Group:	Naming
Interest:	#(#DataElement)
Severity:	error
Solvable by PR:	☐
Solvable by PR cor	
Description:	A displayName property should refer to (a combination of) fields of the data element it is defined in.

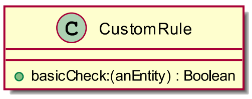
Figure 6.5: Rules Metadata Editor

Listing 1: List of Model entities to which the rule applies

```
NSPGeneralNamingRule>>interest
    ^ #(DataElement Field StateNode TaskNode)
```

As the metadata is entered manually, I have extended Pharo's code editor/browser with a custom editor shown in Figure 6.5. The data in the form is exchanged directly with the source code – changes in the form change the underlying source code and vice versa. A more technical perspective is offered in *Magritte Extensions*.

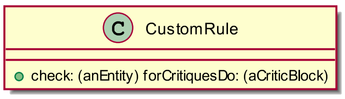
6.5.2 Basic Check



A basic check can be added by implementing *basicCheck:* method. The method should return *true* when the rule was *violated*. An argument of the method is a model entity whose type depends on whatever is returned by the *interest* metadata method.

```
NSPEmptyComponentRule>>basicCheck: aComponent
    ^ aComponent dataElements isEmpty
```

6.5.3 Complex Rule



When the rule should produce multiple different critiques, *check:forCritiquesDo:* should
In this method, every time we see a violation, we call back the critic block and give it a new *Critique* intance.

```
NSPAppBaseDependenciesRule>>check: anApplication forCritiquesDo: aCriticBlock
    self baseDependencies
        do: [ :each |
            anApplication components
                detect: [ :comp | comp name = each ]
                ifNone: [ aCriticBlock cull: (self critiqueFor: anApplication about:␣
↪each) ] ]
```

6.5.4 Critiques

The result of a validation check is a subinstance of *ReAbstractCritique*. This object contains additional information telling the user what precisely went wrong.

The default critique is *ReTrivialCritique*, which apart from a reference to the infringing model entity contains a *tinyHint* description of the problem. Examples in Table 6.1 are provided this way.

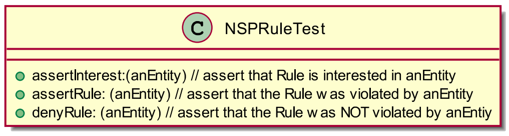
In Listing 2, a critique is provided with a human-friendly description of the issue.

Listing 2: critiqueFor: implementation

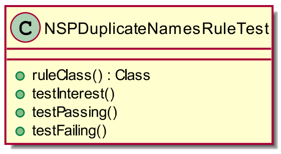
```
critiqueFor: aField
    ^ self
        critiqueFor: aField
        about:
            'Type "' , aField type , '" for field ' , aField dataElement name , '::' ,
↪ aField name
                , ' is not supported'
```

6.5.5 Testing a Rule

Every rule *must* have tests. They are not only used to verify that the rule operates correctly, but the test cases are used to create reports, such as the examples in Table 6.1. To create a test, subclass *NSPRuleTest*; the name of the test class follows [RuleClass]Test pattern. By subclassing and proper name, the system automatically tests basic metadata (name, group, rationale/description). Furthermore, it provides custom assert methods.


NSPRuleTest
assertInterest:(anEntity) // assert that Rule is interested in anEntity
assertRule: (anEntity) // assert that the Rule was violated by anEntity
denyRule: (anEntity) // assert that the Rule was NOT violated by anEntiy

Each rule must have at least the following tests and methods:



- *ruleClass* returns the class of the rule we are testing

```
NSPDuplicateNamesRuleTest>>ruleClass
   ^ NSPDuplicateNamesRule
```

- *testInterest* checks all entities that are expected to go into the rule

```
NSPDuplicateNamesRuleTest>>testInterest
  self assertInterest: NSPComponent new.
  self assertInterest: NSPDataElement new.
  self assertInterest: NSPFlowModel new
```

- *testPassing* checks a correct (valid) situation (*denyRule:*)

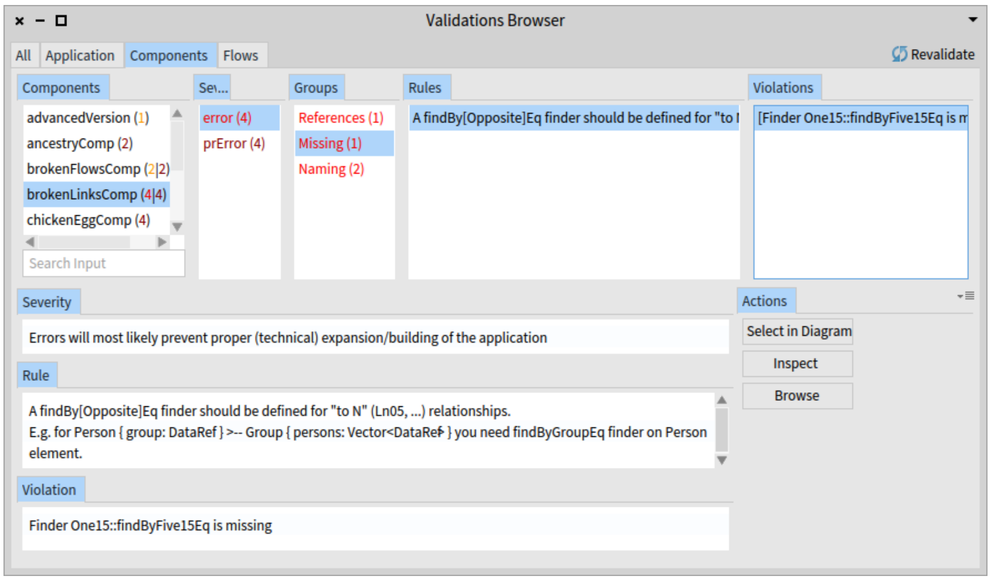
```
NSPDuplicateNamesRuleTest>>testPassing
   | namespace |
   namespace := NSPDataElement new.
   namespace fields add: (NSPField new name: 'hello').
   namespace fields add: (NSPField new name: 'world').
   self denyRule: namespace
```

- *testFailing* that checks an invalid situation (*assertRule:*) * the *critiques* attribute is automatically populated from *assertRule* to minimize the perceived complexity

```
testFailing
   | namespace |
   namespace := NSPDataElement new name: 'El'.
   namespace fields add: (NSPField new name: 'hello').
   namespace fields add: (NSPField new name: 'hello').
   self assertRule: namespace.
   self assert: critiques first tinyHint equals: 'El::fields::hello'
```

6.6 User Tools

An essential part of the validation is not just checking the rules, but presenting the results to the user. The first interface is Validations Browser, shown in Figure 6.6. The browser displays all generated critiques to the user organized by all the available dimensions, such as the source component, severity, group, rule types, etc. Additional tabs provide different groups or context. Some actions are available to regular users (*Select in Diagram*), as well as to modeler developers (e.g., *inspecting* the object internals, or *browsing* the source code class of the rule). Action to automatically fix problems where possible is planned for future releases.



Errors will most likely prevent proper (technical) expansion/building of the application

A findBy[Opposite]Eq finder should be defined for "to N" (Ln05, ...) relationships.
E.g. for Person { group: DataRef } >-- Group { persons: Vector<DataRef> } you need findByGroupEq finder on Person element.

Finder One15::findByFive15Eq is missing

Figure 6.6: Validations Browser

Other place where users can see the issues is directly in the diagram (Figure 6.7).

The warning icon in the top-left corner of the diagram applies to the entire component and opens Validation Browser scoped on it. The Warning Icons in the Data Elements open a smaller validation browser (Figure 6.8) which displays critiques only for the Data Element and its subentities (fields, data childs, etc.).

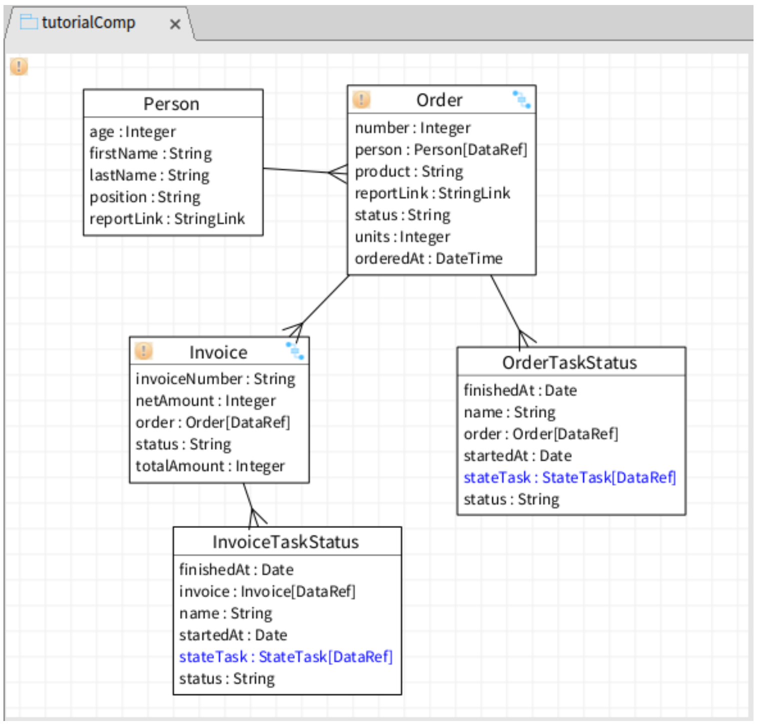
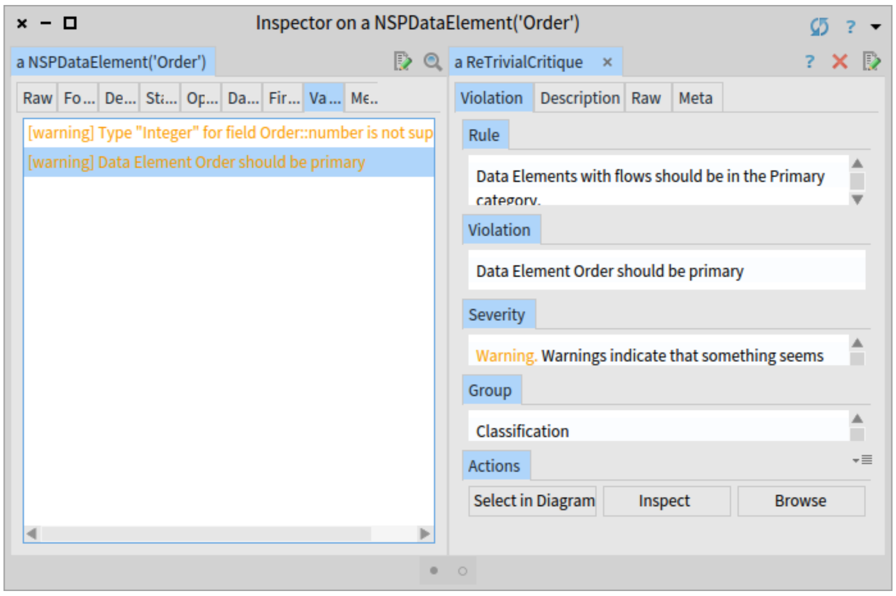


Figure 6.7: Validation Warning Icons in Component Diagram



[warning] Type "Integer" for field Order::number is not sup
[warning] Data Element Order should be primary

Rule
Data Elements with flows should be in the Primary category.

Violation
Data Element Order should be primary

Severity
Warning. Warnings indicate that something seems

Group
Classification

Actions

Figure 6.8: Validations Inspector on a single Data Element

Part III

Implementation

Chapter 7

Diagram Editors

7.1 Component Diagramming

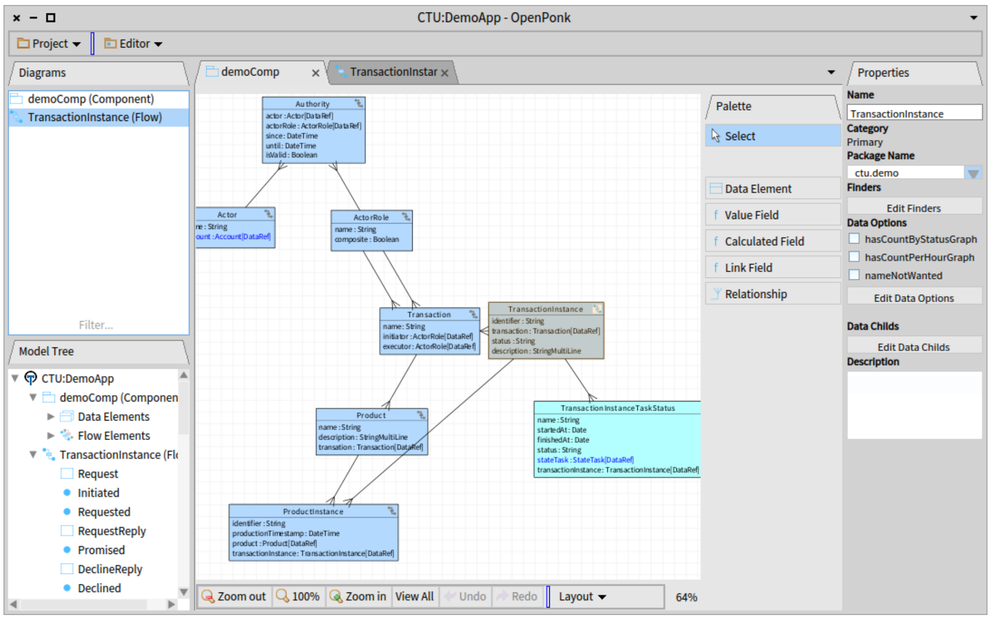


Figure 7.1: Component Editor

Component editor (see Editor in *OpenPonk Modeling Platform*) is responsible for modeling the structural aspect of an NS model. An example of a diagram of a component model is shown in Figure 7.1. The visualization is based on Entity-Relationship database notation that is typically used to represent NS components. The inspiration for additional notation has been drawn from UML class diagrams based on similar understanding as was described in Chapter *Normalized Systems Metamodel*.

All primary structural aspects – Data Elements, Fields (of all types), and Relationships are supported.

7.1.1 Link Fields and Relationships

From an NS perspective, a separate *relationship* concept doesn't exist. A relationship is merely a link field pointing to another Data Element. However, in the diagram we provide an explicit representation using a line between the concerning elements. Note that the real link field is still shown inside the appropriate Data Element box (e.g., *order* field in *OrderTaskStatus* element in Figure 7.2).

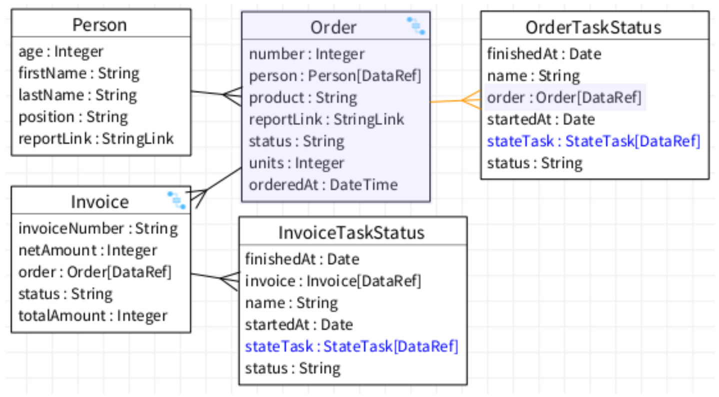


Figure 7.2: Link fields and relationships in a component diagram

The relationship lines in Figure 7.2 are not labeled and end at the borders of the related elements; a typical approach used by many diagramming tools[2]. To ease the identification, both the target Data Element and source Field are highlighted when a user selects a relationship line (e.g. *Order -< OrderTaskStatus::order*); the same applies when a user selects the source Field.

Not all link fields are representable with relationship lines. Figure 7.2 shows two such fields: *InvoiceTaskStatus::stateTask* and *OrderTaskStatus::stateTask*. Although both are link fields, neither references a Data Element in the current component. Instead, they point to *remote* components, such as another user component in the application, or in this case, to one of the base components. As this gives them a special standing, they are shown in different color.

7.1.2 Data Elements and Flows

One of the recommendations for building NS applications is creating Flows only on *Primary* Data Elements. In the modeler, this is addressed by offering *create flow* or *open flow* action only on Primary elements, as shown in Figure 7.3.

[2] Some tools support layouting where the line ends precisely at the position of the source/target field, but that approach has downsides of its own.

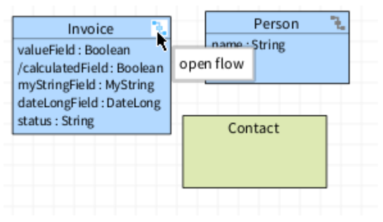


Figure 7.3: Action opening or creating flows on Primary data elements

This seemingly simple one-click action addresses a series of problems that a user could inadvertently create. As noted, the action is only offered for Primary Data Elements (see *PrimaryElementsWithFlows* in Table 6.1), creates the necessary Flow Element, links it to the correct Data Element (FlowElementDataElementNaming rule), and populates Flow Element attributes with proper content. All in a single click.

7.1.3 Errors and Warnings

To provide more localized information about potential problems, error/warning icons are shown on an entity with failing validation (Figure 7.4). Displaying the icons is optional so as not to distract the user.

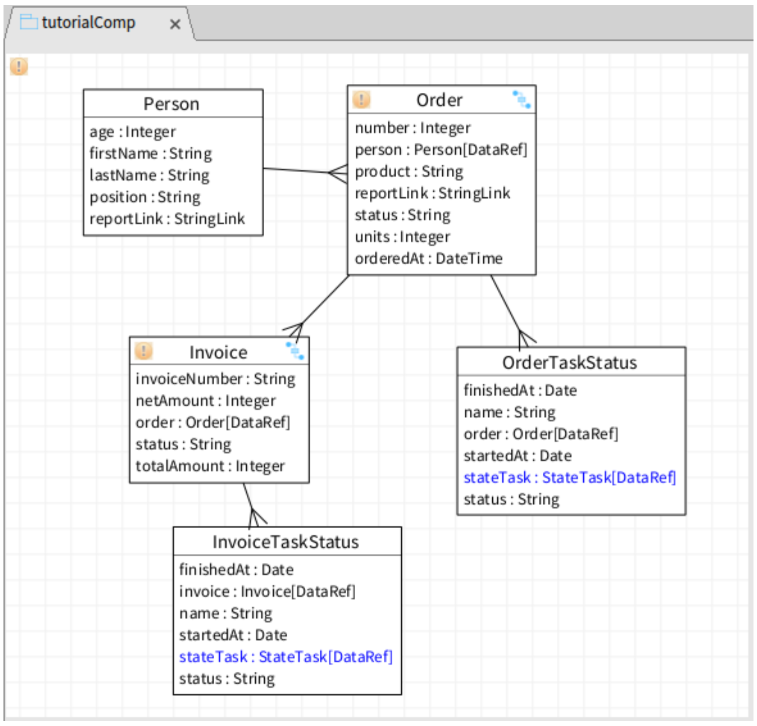
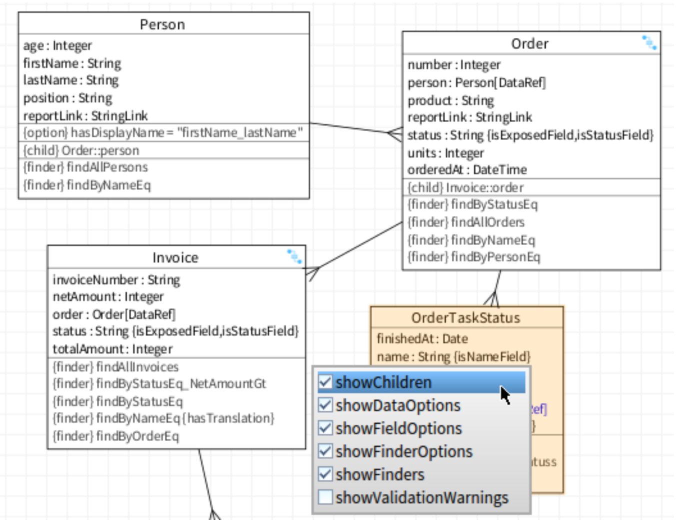


Figure 7.4: Validation Warning Icons in a Component Diagram

7.1.4 Secondary Diagram Content

Not all information is visible in the diagram by default. Showing everything would clutter the diagram and overwhelm the user with noise. Some additional properties, such as finders and data or field options are available on demand (see Figure 7.5).



```
Person
age : Integer
firstName : String
lastName : String
position : String
reportLink : StringLink
{option} hasDisplayName = "firstName_lastName"
{child} Order::person
{finder} findAllPersons
{finder} findByNameEq
```

```
Order
number : Integer
person : Person[DataRef]
product : String
reportLink : StringLink
status : String {isExposedField,isStatusField}
units : Integer
orderedAt : DateTime
{child} Invoice::order
{finder} findByStatusEq
{finder} findAllOrders
{finder} findByNameEq
{finder} findByPersonEq
```

```
Invoice
invoiceNumber : String
netAmount : Integer
order : Order[DataRef]
status : String {isExposedField,isStatusField}
totalAmount : Integer
{finder} findAllInvoices
{finder} findByStatusEq_NetAmountGt
{finder} findByStatusEq
{finder} findByNameEq {hasTranslation}
{finder} findByOrderEq
```

```
OrderTaskStatus
finishedAt : Date
name : String {isNameField}
```

- ✓ showChildren
- ✓ showDataOptions
- ✓ showFieldOptions
- ✓ showFinderOptions
- ✓ showFinders
- ☐ showValidationWarnings

Figure 7.5: Showing additional properties of the model

7.1.5 Future Concepts

The current state of the component diagram is far from complete. Many entities are created using form-based interfaces while a diagramming-based approach could be better.

7.1.5.1 Merged Bi-directional Relationships

Link fields of all types (see *Primary Elements*) are modeled and diagrammed in the same manner – Link Field is created, it is provided with a target, and a particular link field type. However, the inverse types (Ln04, Ln05, Ln06) require the "base" type to exist and point in the other direction. This creates a potential point of failure where user neglects or accidentally removes the base side (see ReverseLinkField rule). If that is the case, then we can merge the visual lines, thus reducing the total number of edges and improving both layouting and aesthetics *[2]*.

Furthermore, instead of letting the user manually define the inverse link, a simple *add inverse link* checkbox (possibly with a configurable field name) can be provided for the base link. This approach always ensures that: (1) the base link exists, (2) the inverse link is always of the correct type (because it is updated according to the base type), and (3) removing the base link will ensure removal of the opposite link.

7.1.6 Data Child Link Field Property

In the same perspective, we can view Data Child entities. Although it may not be directly evident, they always operate on a pre-existing link field.

Thus we add a *is data child* checkbox on the base link. Removal of the link once again removes the Data Child. Combined, they prevent the user fcreating yet another problem (see DataChildReference rule). Technically, the already existing (form-based) UI of the modeler guards the creational part of the problem, but the user has to think about what they are doing, instead of *seeing* what they are doing.

7.1.7 Projections

Data Projections are defined on Data Elements and filter (project) the Data Element's fields. Additionally, they can contain new calculated fields. We can view Projections as a more limited kind of a Data Element and show them as such in the diagram. But care must be taken concerning clutter in the diagram and disrupting the layout. Likewise, it must be easy for the user to choose whether they want to see them or not and even conditionally show them just for specific Data Elements.

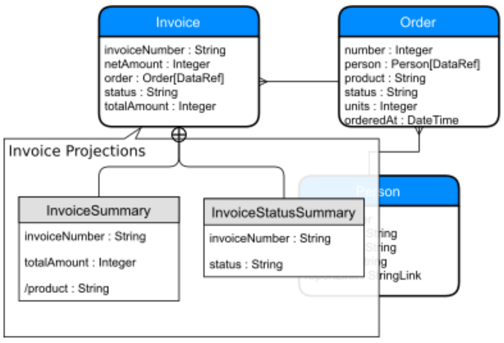


Figure 7.6: A concept of Data Projection presentation

Figure 7.6 shows a potential approach. Instead of adding new entities directly to the diagram and overlapping existing elements or disrupting the layout, they are placed in a box layered on top. The box demarks where the Projections are, to what Data Element they belong, and the box can be easily toggled on or off.

7.2 Flow Diagramming

Figure 7.7 shows a diagram of a simple flow. States are represented by black dots, tasks are represented by rounded boxes, transitions between them rep-

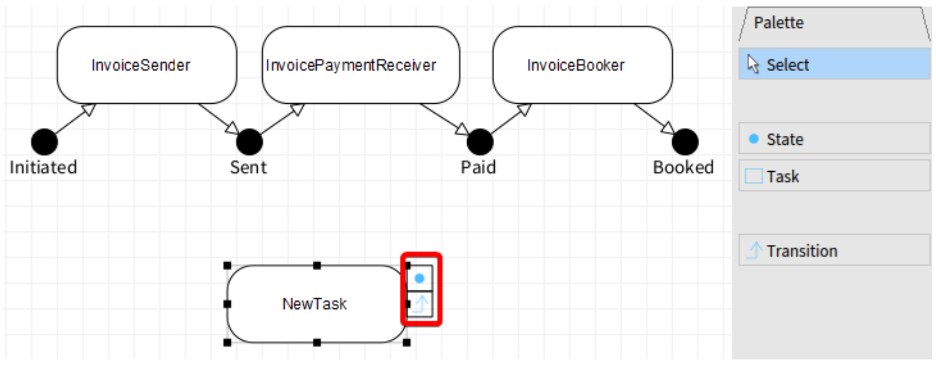


Figure 7.7: A selected task with two available actions.

resented by arrowed lines indicating the direction.

The figure also shows a new task that has just been added and selected. Directly in the diagram are offered actions (highlighted in red) that a user can perform. The top action creates a new state and adds a *Transition* from the selected task to a new state, thus merging several actions into one. Similarly, the bottom action adds a Transition to a target state. It is still more practical than the palette, as the source of the transition is already specified.

When we proceed with an addition of a new entity or a connection, the modeler performs basic validation on whether an entity can be added inside another, or whether the user is trying to connect compatible objects. Flows are governed by two elementary rules. The first rule permits creation of transitions only from a task to a state (or vice versa) as shown in Figure 7.8, but not from task to another task (Figure 7.9) or from state to another state.

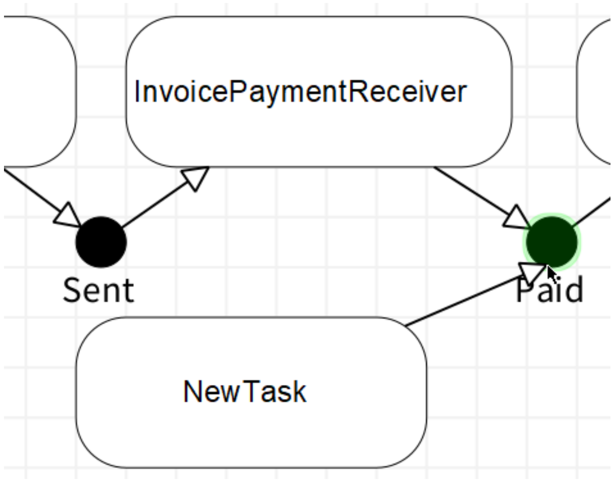


Figure 7.8: A state is accepting transition from a task

The second rule allows only a single outgoing transition from a state or a flow. In Figure 7.10 the same task no longer offers any actions, because it

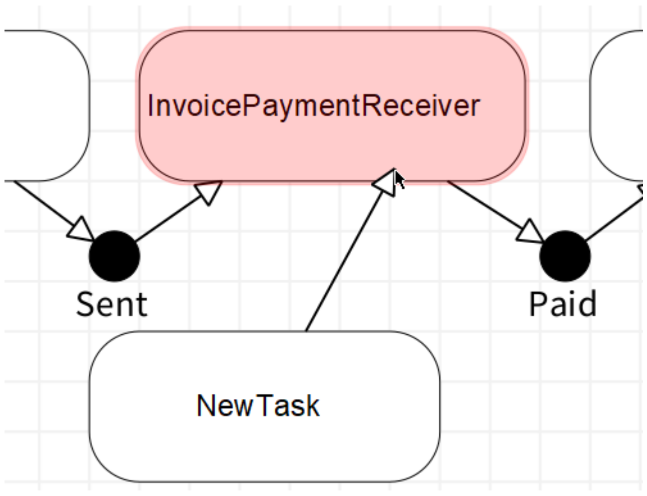


Figure 7.9: A state is denying transition from another state

already has an outgoing transition. Of course, if the user attempted to add transition via Palette, they will be met with the same visual denial.

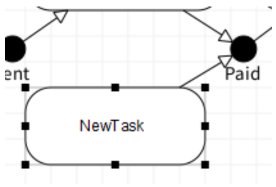


Figure 7.10: Selected task no longer offers any actions

7.2.1 Future Concepts

Future concepts for the flow modeling and diagramming are discussed as part of the Flow Metamodel section of *Normalized Systems Metamodel* chapter.

7.3 Diagram Views and Partitioning

A common concern for larger models of any kind is viewing just the right amount of information. Even component diagrams with a few dozen Data Elements are, or can become overwhelming; especially considering NS systems unlimited growth.

At the moment, partitioning at the level of the model is achieved by specifying aspects such as package name or Taxonomy. Alas, no first-class partitioning is available for the entire model in a similar manner as Data Projection is for Data Element.

How to address this in the model is well outside of the scope of this work, but for the diagrams, three approaches seem appropriate[3].

Static partitioning Such partitioning can be implemented across existing axes of similarity. For example, the user could choose what Taxonomy of Elements they want to see – e.g., only Primary and History, or only Taxonomy, or all Elements with a particular package name. Another example would be to hide all fields and focus only on elements and their relationships. These scenarios are static because they are explicitly implemented when a particular use case was foreseen or needed. A user cannot modify them and is bound by their behavior. A benefit is that they can cover the most common scenarios, and they can be provided via easy-to-use interfaces, as a modeler developer can design an appropriate place for them. Both the earlier presented user-configurable showing of Children/DataOptions/. . . , as well as the concept of a layered view of Projections fall into this category.

Stored partitioning Diagrams are created by manually selecting model entities. For example, a user of a UML modeling tool drags a couple of UML classes from a large model tree onto the diagram. Then the user proceeds with dragging a subset of attributes for each, and perhaps a subset of relationships between them. This approach is very flexible, as the user has full control over the information presented. At the same time, constructing each diagram from scratch is time-consuming. But when new entities are added to the model, the diagram remains the same even if the new entities are relevant in the context presented by the diagram. This makes it the user's responsibility to be aware of any modifications and to make amends to the diagram. In a collaborative environment, it is an easy way how to end up with misleading, or out-of-date diagrams. Nevertheless, this approach is very desirable. OpenPonk does not support this functionality on the surface (i.e., UI), but the underlying code is mostly ready.

Dynamic partitioning This approach tries to achieve best of both worlds; providing full flexibility for the user while ensuring the content is always up-to-date. A user is provided with a query language of sorts[4], where they can describe in abstract terms (metamodel terms) what should be shown. For example, a sentence like:

```
component dataElements select: [ :el |
   (el fields select: #isLinkField) anySatisfy: [ :field |
      field targetElement name = 'Person' ] ]
```

[3] The chosen naming is mine, as I have not encountered a proper naming elsewhere, even though some or all approaches are used in other tools.

[4] Whether the language is visual, textual, or described via UI form screens is not relevant here.

would show all Data Elements that are linking to the *Person* Data Element.

As the content is described by evaluable predicates, any addition or removal in the model is automatically propagated to the diagram without the need for user involvement. Reifying these predicates could lead to reusable and combinable diagram descriptions. The viability of this strategy remains a subject of future exploration.

Chapter 8

Magritte Extensions

Chapter *Magritte* contained a general discussion of Magritte. Here I presented in an illustrative detail two use cases for both OpenPonk and the NS modeler.

8.1 Spec Properties Form Renderer

OpenPonk has a *Properties* widget showing essential properties of a selected model entity (as shown in Figure 8.1).

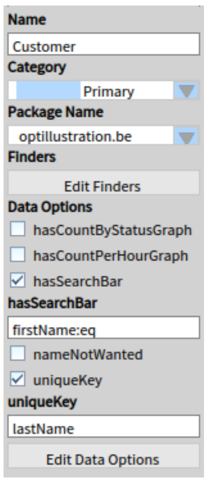

Name
Customer
Category
Primary
Package Name
optillustration.be
Finders
Edit Finders
Data Options
☐ hasCountByStatusGraph
☐ hasCountPerHourGraph
✓ hasSearchBar
hasSearchBar
firstName:eq
☐ nameNotWanted
✓ uniqueKey
uniqueKey
lastName
Edit Data Options

Figure 8.1: Properties widget showing options for a Data Element

Initially, this toolbar was created imperatively. In fact, one of the oldest editors created for OpenPonk (BORM) still uses this imperative construction. It was nothing more than a thin layer on top of one of the UI frameworks available (Listing 1.

Listing 1: Imperative construction of forms

```
BormProcessNodeController>>buildEditorForm: aForm
   (aForm addText: 'Name')
      text: self model name;
      whenTextIsAccepted: [ :newValue | self model name: newValue asString ].
   (aForm addDroplist: 'Submodel')
      items: {nil} , self bormModels;
      displayBlock: [ :m | m ifNil: '' ifNotNil: #name ];
      setSelectedItem: self model submodel;
      whenSelectedItemChanged: [ :newValue | self model submodel: newValue ]
```

Such approach, although sufficient to provide basic functionality, is hardly a good long-term solution. It couples the code to a concrete UI framework and is not reusable in any other context. It creates a non-uniform code, as the widgets tend to have slightly different API, e.g. *text:* and *setSelectedItem:* for writing, and *whenTextIsAccepted:* and *whenSelectedItemChanged:* for reading. Furthermore, it promotes mixing several widgets inside a single method, as there is no natural approach to separate the code.

To address all these issues, Magritte was chosen as a natural alternative. Each form field is now described by an appropriate Magritte description (Listing 2). This approach separates the data from UI, *forces* separation of every description and unifies read/write access via Magritte's accessors.

Note that at the moment controllers mostly use PluggableAccessor. However, they are planned to be replaced with a custom DelegateAccessor.

Listing 2: Rewritten buildEditorForm: with Magritte

```
BormProcessNodeController>>descriptionName
 <magritteDescription>
 ^ MAStringDescription new
     accessor: (MAPluggableAccessor read: [ :ctrl | ctrl model name ] write: [ :ctrl␣
↪:newValue | ctrl model name: newValue ];
     label: 'Name';
     priority: 1;
     yourself.

BormProcessNodeController>>descriptionSubmodel
 <magritteDescription>
 ^ MASingleOptionDescription new
     accessor: (MAPluggableAccessor read: [ :ctrl | ctrl model submodel ] write: [␣
↪:ctrl :newValue | ctrl model submodel: newValue ];
     label: 'Submodel';
     options: self bormModels;
     priority: 2;
     yourself
```

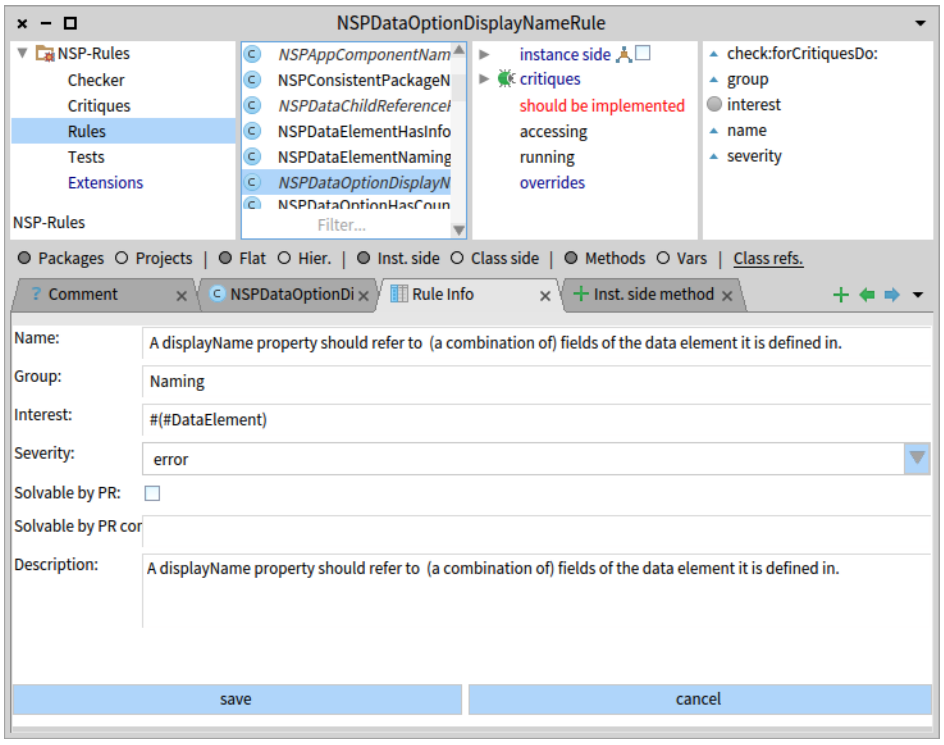
With the imperative construction replaced by declarative descriptions, we still need to preserve the original behavior and look of the properties toolbar. Thus a new renderer was created to transform the descriptions into the original Spec API calls. With this approach, a gradual transition is possible, as both the old and the new method can be used at the same time from different controllers even within the same OpenPonk editor.

Compared to the Morphic renderer (shown in Figure 3.5), this Form renderer cannot (at the moment) display Relationship descriptions. This has been a partially intentional decision, as such relationships (e.g., between a DataElement and its Fields) should be managed via the diagramming interface.

On the other hand, it contains two new features. (1) Action descriptions performing arbitrary actions that are visualized as buttons. (2) Support for (sub)container descriptions (*MAContrainer* and *MAPriorityContainer*). With containers, descriptions can be grouped, and more importantly, they can be dynamically generated. For example all the individual Data Options (from *hasCountByStatusGraph* to *uniqueKey*) shown in Figure 8.1 are in fact dynamically generated based on the current modeler configuration (which options should be available by default), options that were already specified for the selected Data Element, and whether such options are boolean or string based. Recall that there were over 20 Data Options Types (Figure 4.8), yet the properties form shows only five of them.

8.2 Calypso Extensions

In chapter *Rules and Validations* I have shown a simple editor for manipulating Rules metadata, once again shown in Figure 8.2.



Name:	A displayName property should refer to (a combination of) fields of the data element it is defined in.
Group:	Naming
Interest:	#(#DataElement)
Severity:	error
Solvable by PR:	☐
Solvable by PR cor	
Description:	A displayName property should refer to (a combination of) fields of the data element it is defined in.

Figure 8.2: Rules metadata editor

This editor utilizes the meta-capabilities of Pharo, extensibility of the newly developed Calypso *[18]* browser[1], the flexibility of Magritte descriptions, and composability of other Pharo UI components.

The individual descriptions for the editor are stored in a separate class whose instance has a reference to the rule class we are currently editing. But instead of (or in addition to) accessing values of an object, the description's accessor operates directly on the source code (Listing 3) by recompiling new or existing methods, or even removing ones when they are not warranted.

Listing 3: Simplified code of the rule descriptions

```
NSPClyRuleInfoDescription>>descriptionName
<magritteDescription>
^ MAStringDescription new
    accessor:
    (MAPluggableAccessor
        read: [ :obj | obj ruleClass new name ]
        write: [ :obj :newValue | obj ruleClass compile: ('name<n><t>^ <1p>'␣
↪expandMacrosWith: newValue) ]);
    label: 'Name';
    priority: 1;
    yourself

NSPClyRuleInfoDescription>>descriptionSeverity
<magritteDescription>
^ MASingleOptionDescription new
    options: #(error warning information prError);
    morphClass: MADropListMorph;
    accessor:
    (MAPluggableAccessor
        read: [ :obj | obj ruleClass new name ]
        write: [ :obj :newValue |
        newValue = #warning
            ifTrue: [ obj ruleClass removeSelector: #severity ]
            ifFalse: [ obj ruleClass compile: ('severity<n><t>^ #<1s>'␣
↪expandMacrosWith: newValue) ] ]);
    label: 'Severity';
    priority: 4;
    yourself
```

This use case was initially a demonstrative prototype, but has proven to be quite useful.

Other uses of Magritte are present, and many additional user interfaces were developed for Pharo. To limit the extent of this thesis, they are not discussed in further detail and are only sporadically mentioned in relevant context.

[1] Calypso will be the default code editor for Pharo 7.

Part IV

Testing and Operations

Chapter 9

Error Tracking and Reporting

9.1 Error Tracking

Writing code without any errors or mistake is indeed a dream of every programmer, and many practices, such as Test Driven Development (TDD) are attempting the lower the number of errors produced and improve the quality of code and product. Nevertheless, errors are present in possibly every conceivable piece of software. Therefore it is essential to consider what to do when errors occur during *operations*.

One of many possible ways is automated error reporting. Whenever a bug occurs, information about the failure is automatically sent to a location where the developers can review the issues, learn from them, and address them in future releases. This is also a strategy chosen for the NS modeler, and later for OpenPonk itself.

The initial attempt was a simple collection of error stack traces using Pharo's ShoreLine Reporter *[3]* sent to a private AWS S3 bucket. After over a month in production, this solution proved to be grossly insufficient – a very problematic situation considering the limited options available for Pharo. The main problem was not the collection itself, although just an error stack trace by itself is insufficient. But instead the consumption and processing of the captured information, which is perhaps even more important. Hand-analyzing files individually without any overview or organization was neither practical nor efficient, and eventually, I was forced to give up on this solution.

Neverthless, *a* solution was still required. So I have reversed my criteria and started looking for solutions that were addressing the second part of the problem – the consumption of the data. As Pharo did not have any support in this area, my search criteria had to accommodate for it. I was looking for a self-hosted (to negate/minimize any costs), open-source, feature-rich solution capable of integration with Pharo.

From several alternatives I have concluded on Sentry error tracking platform *[4]*. The platform is, as required, feature-rich, open-source, easily self-hosted, and most critically provides well-documented SDK for implementing bindings for a custom language. The last requirement was the most important, as no platform had native support for Pharo, nor any bindings existed in Pharo community.

Thus parallel to the development of the modeler I have created *pharo-sentry [5]* project, which provided the necessary bindings to integrate Pharo and Sentry.

9.2 Pharo-Sentry

Pharo-Sentry is composed of three parts.

9.2.1 Sentry-Core

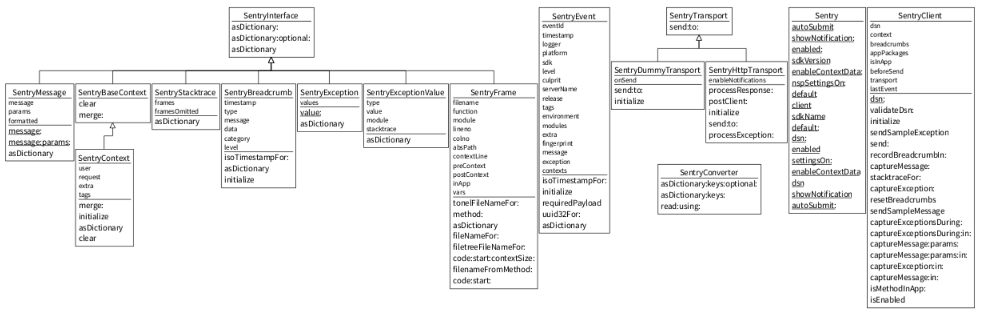
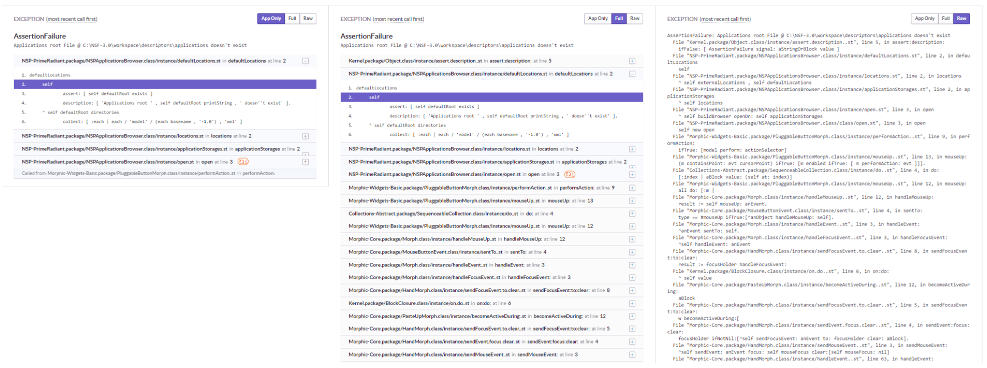


Figure 9.1: The current model of pharo-sentry

The first part is the core functionality, model of which is shown in Figure 9.1. The core is responsible for representing captured events, serializing them to correct Sentry SDK forms, and dispatching them to a configured server.

I have implemented bindings for most of the functionality that Sentry offers. Notably:

- serialization of exception/stack traces/individual trace frames
  - Compared to stack traces provided by ShoreLine, these can contain variable values (if enabled), location and code context, and distinguishing between OpenPonk/NSM frames and Pharo frames, which further improves reading the reports (see Figure 9.2 for comparison).
- breadcrumbs
  - Logging of application events that are sent when a problem occurs, thus providing a trail of events that happened before the error.
- messages



EXCEPTION (most recent call first)

App Only | Full | Raw

AssertionFailure

```
Applications root File @ C:\NSF-3.0\workspace\descriptors\applications doesn't exist
```

NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/defaultLocations.st in defaultLocations at line 2

```
1. defaultLocations
2.     self
3.         assert: [ self defaultRoot exists ]
4.         description: [ 'Applications root ' , self defaultRoot printString , ' doesn''t exist' ].
5.     ^ self defaultRoot directories
6.         collect: [ :each | each / 'model' / (each basename , '-1.0') , 'xml' ]
```

NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/locations.st in locations at line 2

NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/applicationStorages.st in applicationStorages at line 2

NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/open.st in open at line 3

Called from: Morphic-Widgets-Basic.package/PluggableButtonMorph.class/instance/performAction..st in performAction:

EXCEPTION (most recent call first)

App Only | Full | Raw

AssertionFailure

```
Applications root File @ C:\NSF-3.0\workspace\descriptors\applications doesn't exist
```

Kernel.package/Object.class/instance/assert.description..st in assert:description: at line 5

NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/defaultLocations.st in defaultLocations at line 2

```
1. defaultLocations
2.     self
3.         assert: [ self defaultRoot exists ]
4.         description: [ 'Applications root ' , self defaultRoot printString , ' doesn''t exist' ].
5.     ^ self defaultRoot directories
6.         collect: [ :each | each / 'model' / (each basename , '-1.0') , 'xml' ]
```

NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/locations.st in locations at line 2

NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/applicationStorages.st in applicationStorages at line 2

NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/open.st in open at line 3

Morphic-Widgets-Basic.package/PluggableButtonMorph.class/instance/performAction..st in performAction: at line 9

Morphic-Widgets-Basic.package/PluggableButtonMorph.class/instance/mouseUp..st in mouseUp: at line 13

Collections-Abstract.package/SequenceableCollection.class/instance/do..st in do: at line 4

Morphic-Widgets-Basic.package/PluggableButtonMorph.class/instance/mouseUp..st in mouseUp: at line 12

Morphic-Core.package/Morph.class/instance/handleMouseUp..st in handleMouseUp: at line 12

Morphic-Core.package/MouseButtonEvent.class/instance/sentTo..st in sentTo: at line 4

Morphic-Core.package/Morph.class/instance/handleEvent..st in handleEvent: at line 3

Morphic-Core.package/Morph.class/instance/handleFocusEvent..st in handleFocusEvent: at line 3

Morphic-Core.package/HandMorph.class/instance/sendFocusEvent.to.clear..st in sendFocusEvent:to:clear: at line 8

Kernel.package/BlockClosure.class/instance/on.do..st in on:do: at line 6

Morphic-Core.package/PasteUpMorph.class/instance/becomeActiveDuring..st in becomeActiveDuring: at line 12

Morphic-Core.package/HandMorph.class/instance/sendFocusEvent.to.clear..st in sendFocusEvent:to:clear: at line 5

Morphic-Core.package/HandMorph.class/instance/sendEvent.focus.clear..st in sendEvent:focus:clear: at line 4

Morphic-Core.package/HandMorph.class/instance/sendMouseEvent..st in sendMouseEvent: at line 3

EXCEPTION (most recent call first)

App Only | Full | Raw

```
AssertionFailure: Applications root File @ C:\NSF-3.0\workspace\descriptors\applications doesn't exist
  File "Kernel.package/Object.class/instance/assert.description..st", line 5, in assert:description:
    ifFalse: [ AssertionFailure signal: aStringOrBlock value ]
  File "NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/defaultLocations.st", line 2, in defau
ltLocations
    self
  File "NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/locations.st", line 2, in locations
    ^ self externalLocations , self defaultLocations
  File "NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/applicationStorages.st", line 2, in ap
plicationStorages
    ^ self locations
  File "NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/instance/open.st", line 3, in open
    ^ self buildBrowser openOn: self applicationStorages
  File "NSP-PrimeRadiant.package/NSPApplicationsBrowser.class/class/open.st", line 3, in open
    self new open
  File "Morphic-Widgets-Basic.package/PluggableButtonMorph.class/instance/performAction..st", line 9, in perf
ormAction:
    ifTrue: [model perform: actionSelector]
  File "Morphic-Widgets-Basic.package/PluggableButtonMorph.class/instance/mouseUp..st", line 13, in mouseUp:
    (m containsPoint: evt cursorPoint) ifTrue: [m enabled ifTrue: [ m performAction: evt ]]].
  File "Collections-Abstract.package/SequenceableCollection.class/instance/do..st", line 4, in do:
    [:index | aBlock value: (self at: index)]
  File "Morphic-Widgets-Basic.package/PluggableButtonMorph.class/instance/mouseUp..st", line 12, in mouseUp:
    all do: [:m |
  File "Morphic-Core.package/Morph.class/instance/handleMouseUp..st", line 12, in handleMouseUp:
    result := self mouseUp: anEvent.
  File "Morphic-Core.package/MouseButtonEvent.class/instance/sentTo..st", line 4, in sentTo:
    type == #mouseUp ifTrue:[^anObject handleMouseUp: self].
  File "Morphic-Core.package/Morph.class/instance/handleEvent..st", line 3, in handleEvent:
    ^anEvent sentTo: self.
  File "Morphic-Core.package/Morph.class/instance/handleFocusEvent..st", line 3, in handleFocusEvent:
    ^self handleEvent: anEvent
  File "Morphic-Core.package/HandMorph.class/instance/sendFocusEvent.to.clear..st", line 8, in sendFocusEven
t:to:clear:
    result := focusHolder handleFocusEvent:
  File "Kernel.package/BlockClosure.class/instance/on.do..st", line 6, in on:do:
    ^ self value
  File "Morphic-Core.package/PasteUpMorph.class/instance/becomeActiveDuring..st", line 12, in becomeActiveDur
ing:
    aBlock
  File "Morphic-Core.package/HandMorph.class/instance/sendFocusEvent.to.clear..st", line 5, in sendFocusEven
t:to:clear:
    w becomeActiveDuring:[
  File "Morphic-Core.package/HandMorph.class/instance/sendEvent.focus.clear..st", line 4, in sendEvent:focus:
clear:
    focusHolder ifNotNil:[^self sendFocusEvent: anEvent to: focusHolder clear: aBlock].
  File "Morphic-Core.package/HandMorph.class/instance/sendMouseEvent..st", line 3, in sendMouseEvent:
    ^self sendEvent: anEvent focus: self mouseFocus clear:[self mouseFocus: nil]
  File "Morphic-Core.package/HandMorph.class/instance/handleEvent..st", line 63, in handleEvent:
```

Figure 9.2: Comparison of Stacktrace views in Sentry UI (App Only, Full, Raw)

  - An arbitrary content sent to the server, typically notification messages or warnings.

- context and event attributes

  - A very valuable functionality with which additional information such as library versions, modeler version, user information, error level, etc. can be attached to all events.

Attaching version information, in particular, has proven to be especially valuable, as it makes it obvious whether a particular problem has resurfaced, or just merely some user is running an older version of the modeler.

A major remaining feature that is still due to be implemented are so-called Releases. Releases provide first-class tracking of individual versions and automatically integrate with Git versioning. However, the current tracking of the modeler version via context information has been good enough solution so far, which enabled to refocus effort back onto the modeler itself.

9.2.2 Sentry-Beacon

The second part of the pharo-sentry project is integration with logging framework Beacon *[6]*.

Pharo-sentry provides a separate logger (Listing 1) which dispatches events to Sentry.

Listing 1: SentryLogger

```
SentryLogger start.
"send a Sentry Message event"
StringSignal emit: 'test'.
SentryLogger stop.
```

(continues on next page)

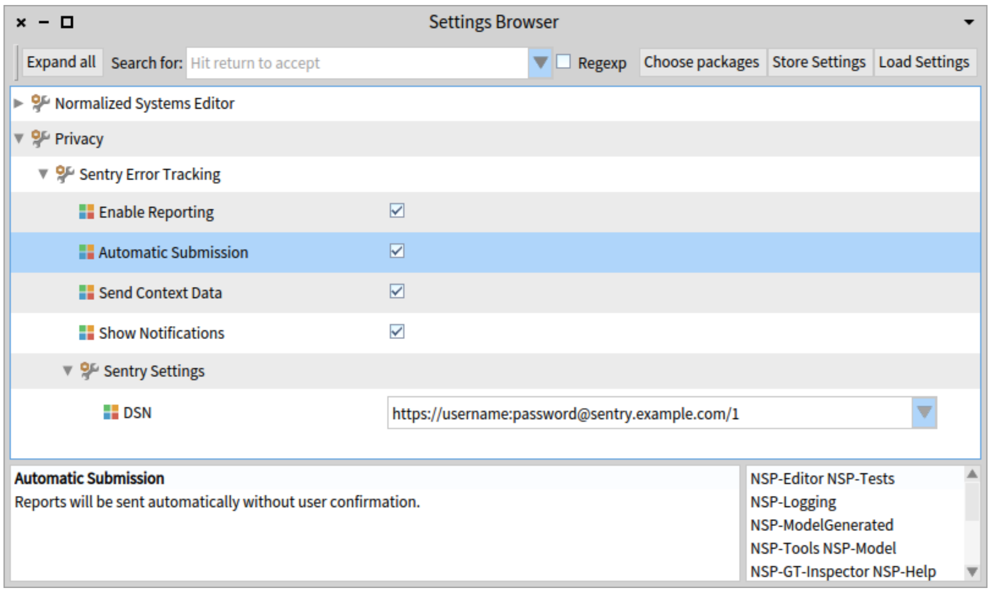
(continued from previous page)

```
SentryLogger new runDuring: [
    "send a Sentry Exception event"
    [ 1/0 ] on: Exception do: [ :ex | ex emit ]
].
```

Furthermore, serializers are provided for all standard Beacon signals (Exception, String, Wrapper, ThisContext). Therefore it is possible to send signals collected from other Loggers as well.

9.2.3 System Integration and Debugger

The final part of pharo-sentry is integration with Pharo itself and its tooling.



Normalized Systems Editor
Privacy
Sentry Error Tracking
Enable Reporting ✓
Automatic Submission ✓
Send Context Data ✓
Show Notifications ✓
Sentry Settings
DSN https://username:password@sentry.example.com/1

Automatic Submission
Reports will be sent automatically without user confirmation.

NSP-Editor NSP-Tests
NSP-Logging
NSP-ModelGenerated
NSP-Tools NSP-Model
NSP-GT-Inspector NSP-Help

Figure 9.3: Settings configurable during runtime

Several configuration options are available that a user can modify (Figure 9.3). Particularly *Send Context Data* is an important option, as enabling it will send variable and model data to report server(s). Such data can potentially contain sensitive information. Thus it is vital to ensure that a user has control over what is being sent. For example, the default releases of the modeler intended for NS developers have this option disabled, but we enabled it in in releases intended for students and university courses.

Lastly, pharo-sentry integrates directly with Pharo debugger. When any runtime exception happens, they are dispatched to sentry if *Automatic Submission* is enabled. When it is not enabled, a *Report* button is added to Debugger window for a user to send the exception manually (Figure 9.4). With enabled submission, the button states that the exception was already reported (Figure 9.5).

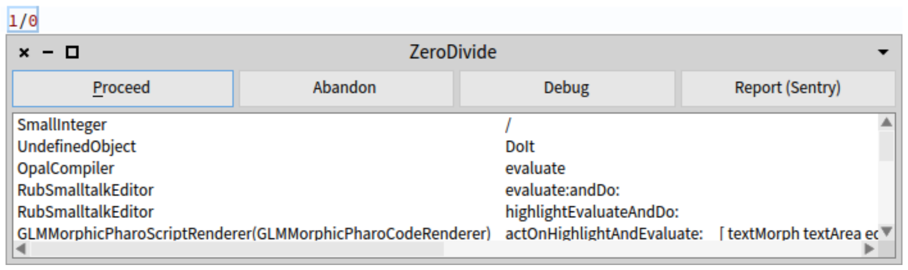

```
1/0
ZeroDivide
Proceed | Abandon | Debug | Report (Sentry)
SmallInteger                                              /
UndefinedObject                                           DoIt
OpalCompiler                                              evaluate
RubSmalltalkEditor                                        evaluate:andDo:
RubSmalltalkEditor                                        highlightEvaluateAndDo:
GLMMorphicPharoScriptRenderer(GLMMorphicPharoCodeRenderer) actOnHighlightAndEvaluate:   [ textMorph textArea ec
```

Figure 9.4: *Report* button in Debugger

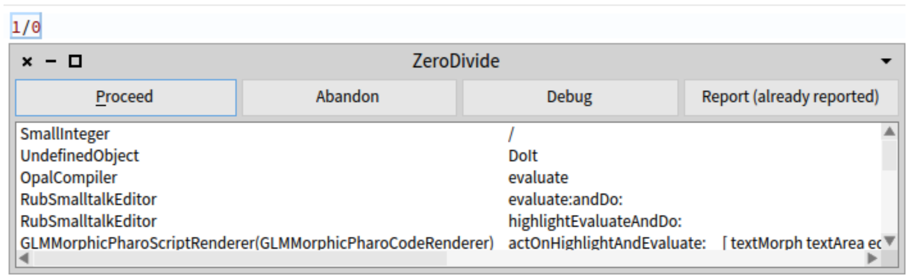

```
1/0
ZeroDivide
Proceed | Abandon | Debug | Report (already reported)
SmallInteger                                              /
UndefinedObject                                           DoIt
OpalCompiler                                              evaluate
RubSmalltalkEditor                                        evaluate:andDo:
RubSmalltalkEditor                                        highlightEvaluateAndDo:
GLMMorphicPharoScriptRenderer(GLMMorphicPharoCodeRenderer) actOnHighlightAndEvaluate:   [ textMorph textArea ec
```

Figure 9.5: *Report* button if the event was already submitted

9.3 Error and Model Capture

9.3.1 Serializing Modeler State

A practical issue for debugging is handling state, and addressing problems that occurred in one place, even though the source of the problem lied in an entirely different part of the application. A typical scenario in many languages is module A setting a variable of some object to *null*. The setting itself works as expected, but when module B, which assumes for it to be always non-null, tries to access it, an error is encountered an exception raised. Most importantly those two events (writing to a variable, and reading from it) can happen in separate time events, and thus analyzing the exception stack is fruitless, as the incriminating code is not part of the stack. To ease debugging in such cases, a full application state logging and reporting was introduced. When an exception is raised, the entire state of all objects related to the opened application (individual instances of all model entities, all NS/OpenPonk objects responsible for the runtime of the modeler is serialized. The serialization is performed using Fuel *[7]* which can serialize and compress tens of thousands of objects in a fraction of a second. The data is then sent to a private S3 bucket. To avoid organization issues, the payload is marked and paired with an appropriate Sentry event/issue.

9.3.2 Submitting User Models

Many problems encountered in the past months were not related to issues in the modeler, but rather to the content of XML files. This stems mostly from lack of documentation of the metamodel and what possible combination of entities was permitted. This problematic is illustrated by the following situations.

Flow Elements should be always tied to Data Elements. In fact, neither the modeler nor Prime Radiant permits the creation of a Flow Element without associating it to an existing Data Element. However Prime Radiant (and until recently the modeler[1]) does not prevent the user from deleting the Data Element. When a user then tries to open an application with such a component, a problem is raised, as the modeler attempts to construct the model graph, but is missing a mandatory component.

In some cases, such as missing Data States (very common scenario), the modeler automatically creates missing model entities. But in the case of the mentioned missing Data Element, no such default action can be performed, as the resolution is ambiguous. Perhaps the user meant to delete the Flow Element, but forgot. Or the Data Element was removed by accident and should be added back to the component. Or perhaps the Data Element was renamed, and so should be the Flow Element. Other such problems will undoubtedly arise, especially in less utilized entities of the metamodel. For Prime Radiant, the detection of such issues usually occurs during expansion, but the modeler requires a more nuanced and a user-friendlier approach.

To ease the detection of such issues, when *Context Data* is enabled, the application and component files are serialized and reported in the same fashion as the application state serialization. This is paired with a Sentry Event and uploaded to S3.

[1] As of yet the modeler permits renaming of Data Element without renaming the Flow Element, resulting in the same problem.

Chapter 10

Testing, Continuous Integration and Deployment

10.1 Code Testing

This section summarizes all efforts related to direct code testing, whether in the form of unit tests, integration tests, or end-to-end tests.

10.1.1 General Summary

The modeler version 1.18.0 (latest at the time of writing) contains a total of 686 tests.

Broken down by individual packages with tests, the result is shown in Table 10.1.1.

Package	Package Test Coverage	Total Coverage
NSP-Editor	49.23	51.69
NSP-Model	19.59	69.08
NSP-GT-Inspector	37.5	37.5
NSP-Rules	61.02	71.51
NSP-Serialization	84.03	86.55
NSP-Browsers	57.25	57.25
NSP-Settings	19.4	41.79

Package test coverage is computed based only on tests within the same package, while *Total* coverage is based on tests from all packages. All numbers are percentages. For *NSP-Model* package in particular, the numbers differ significantly, as most of the code has been generated (by a tested generator), and only manually added code was explicitly tested. Of course, most packages are heavily utilizing the model (e.g., Rules), so they all contribute significantly to the total coverage.

Most packages (probably all) have the coverage somewhat deflated by experimental code, generated code, hard to test UI code, and support code I have created to ease or guide some of my programming (typically scripts to explore, inspect, ...). However, manually removing these pieces of code for the code coverage aggregation was an undue burden.

As an example, if we look only on the code of Rules themselves, and ignore UI and some of my experiments, the coverage jumps to 93.72%, nicely shown in Figure 10.1.

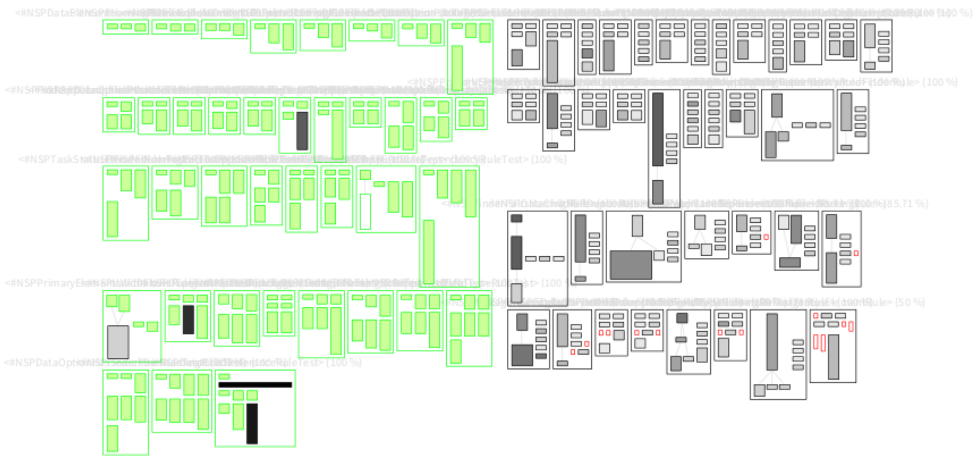


Figure 10.1: Code coverage visualization of NS rules.

10.1.2 Testing Strategies

An important aspect of testing is developing strategies for approaching code with specific characteristics.

One of the examples are once again Rules, for which I've created a systematic testing approach (see :doc:rules:). With a system, it is both easy to write tests, as I already know what is needed to test, as well as easy to verify that the tests are present[4].

Another example is controllers, which are responsible for managing the diagram visualizations and mediating user interactions and mediating data changes between models, visualized information, and associated widgets. Controllers code inherently performs a lot of manipulations on both the models and visualizations, often intertwined, which makes testing it particularly problematic, as all sides must be fully initialized. As it was short-sighted to leave such critical piece of code almost untested, another strategy was formed. The controllers (both in OpenPonk and later in NS modeler) were refactored, model and visualization interaction and construction was separated to a large degree, and I have started utilizing mocking framework Mocketry *[19]* to achieve fur-

[4] In practice Rules are usually written in TDD style, so tests are written first.

ther separation. With this approach, I've managed to increase code coverage from almost zero to the current ~80% (and for Component Controllers ~90%)[5].

Another approach is, quite naturally, separation of problematic code. By separation I do not mean separation of concerns, but rather moving the code hiding within a well-tested module somewhere else. The intent is to make problems visible and obvious. Point in case, most browser-based user interfaces did not have any automated tests and instead were tested mostly by-hand. Because they were part of a larger package, their poor testing (or lack thereof) was easily overlooked – it was only a single (or couple) classes. Thus from a high-level perspective it was not ideal, but it was passable. Further addressing just a single browser would result in very specific tests that would not be applicable elsewhere. But by moving browsers from many different packages together, it became *literally* a glaring issue (Figure 10.2). Furthermore, by moving them together, patterns and commonalities between them emerged, and it was possible to start addressing the problems in a consistent manner (Figure 10.3).

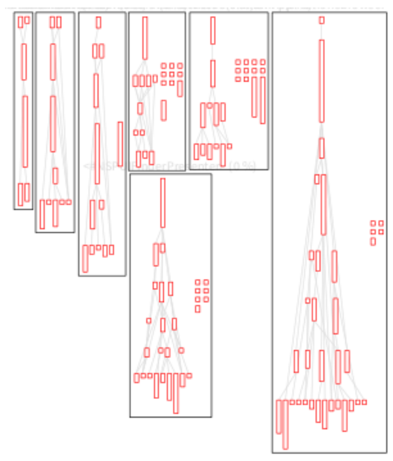


Figure 10.2: Untested browsers

That previous text discussed *automated* testing. On top of it, before every release some time is devoted to manual QA to ensure that at least the most common operations go correctly. It is desirable to also automate this as much as possible, inspired by, e.g., Selenium used for website testing. But no such tool exists as of yet for Pharo.

10.2 Scenarios Testing

A non-automated approach that I am utilizing is scenarios testing. A *scenario* is an NS application or component that contains a particular configuration of elements and entities.

[5] Controllers are part of *NSP-Editor* package.

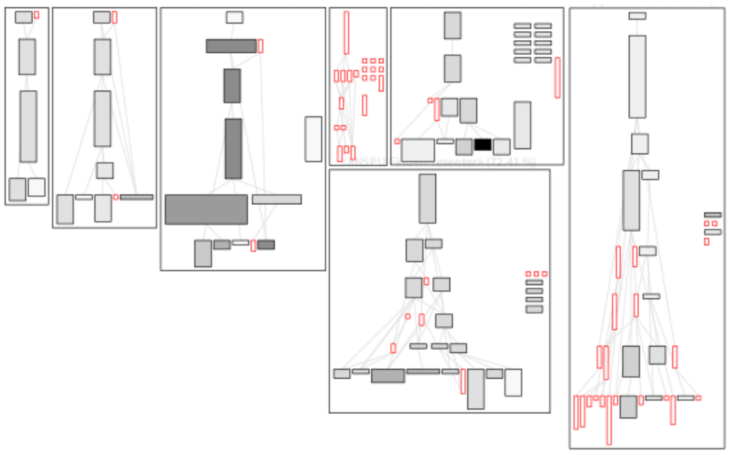
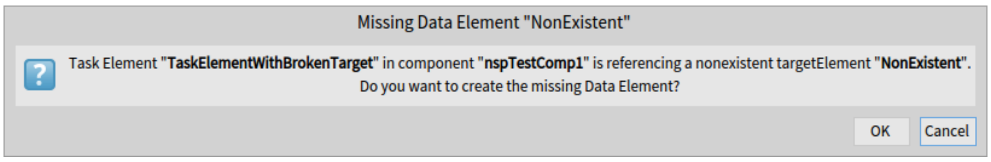


Figure 10.3: Addressing browser testability

These scenarios are useful in several ways.

Firstly, they can be used to represent a problematic model state (whether application or component model state). Such scenarios are typically created directly in Prime Radiant, as the modeler has not yet addressed the problem. Several such examples were mentioned in the Flow Metamodel section of *Normalized Systems Metamodel* chapter.


Missing Data Element "NonExistent"

Task Element "**TaskElementWithBrokenTarget**" in component "**nspTestComp1**" is referencing a nonexistent targetElement "**NonExistent**".
Do you want to create the missing Data Element?

Figure 10.4: Opening an application with Task Element missing a Data Element

The second usage is a visual/manual inspection of how the modeler behaves in edge scenarios (Figure 10.4), or overviewing all aspects of a certain feature such as local (Figure 10.5) and remote (Figure 10.6) link fields.

Another usage is exploration and communication. Figure 10.7 shows a flow that contains several prohibited situations (e.g., TaskXY has two different end states). Unlike the modeler, Prime Radiant permits the creation of such flow, even though it will fail during expansion. Note that in principle Task having multiple different end states does make sense. However Prime Radiant cannot process such semantics at the moment. Thus although it is a bug at the moment, we are discussing how to turn it into a feature.

At the moment there are 14 scenarios in two applications, both having in total 11 components and 3 flows. On top of this are other applications that serve different purposes. An application with a significant amount of Data

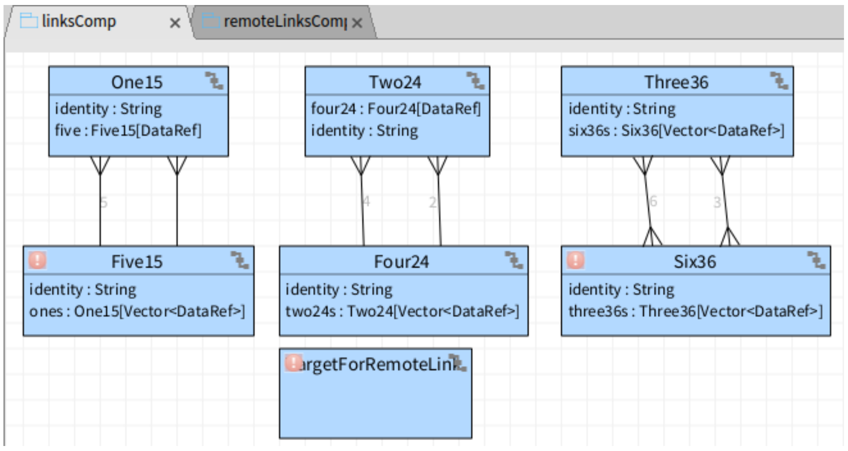


Figure 10.5: Component with variations of local linkFields

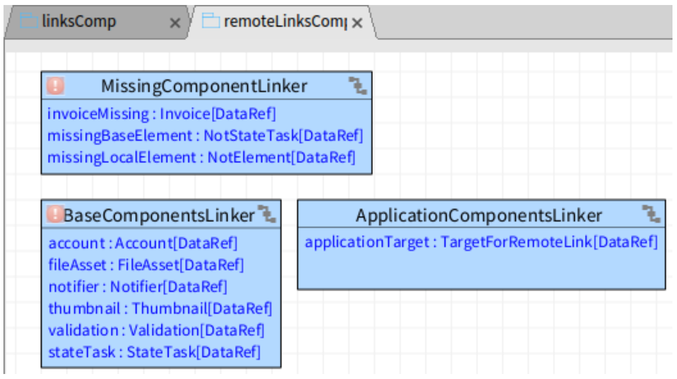
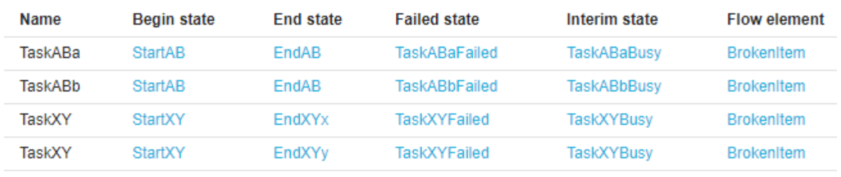

linksComp
remoteLinksComp
MissingComponentLinker
invoiceMissing : Invoice[DataRef]
missingBaseElement : NotStateTask[DataRef]
missingLocalElement : NotElement[DataRef]
BaseComponentsLinker
account : Account[DataRef]
fileAsset : FileAsset[DataRef]
notifier : Notifier[DataRef]
thumbnail : Thumbnail[DataRef]
validation : Validation[DataRef]
stateTask : StateTask[DataRef]
ApplicationComponentsLinker
applicationTarget : TargetForRemoteLink[DataRef]

Figure 10.6: Component with variations of remote linkFields



Name	Begin state	End state	Failed state	Interim state	Flow element
TaskABa	StartAB	EndAB	TaskABaFailed	TaskABaBusy	BrokenItem
TaskABb	StartAB	EndAB	TaskABbFailed	TaskABbBusy	BrokenItem
TaskXY	StartXY	EndXYx	TaskXYFailed	TaskXYBusy	BrokenItem
TaskXY	StartXY	EndXYy	TaskXYFailed	TaskXYBusy	BrokenItem

Figure 10.7: TaskXY has two different end states, and Tasks ABa and ABb share the same begin state

Elements and entities to observe performance, typical (tutorial) applications, a “playground” application to experiment with behavior, and more.

10.3 Continuous Integration & Deployment

OpenPonk and all its projects make use of automated testing and Continuous Integration.

The standard pipeline used by OpenPonk and all open-source projects I have written for Pharo utilize GitHub as source code repository, Travis for automated code testing, Coveralls for watching the code coverage trajectory[1], and for OpenPonk specifically, private CCMi server to host build artifacts. This stack serves well and posits no cost as all mentioned services are provided for free for public open-source projects.

NS modeler, however, is not a public project; therefore an alternative solution was required. I chose to host the code in a private repository on a GitLab *[20]* instance provided by my faculty. It naturally followed to seek Continuous Integration (CI) server well integrated with GitLab, namely GitLab CI/CD *[21]*.

Just like GitLab itself, GitLab CI/CD can be self-hosted, which I took full advantage of[2] and hosted it on a private AWS EC2 instance. Direct access to the server was surprisingly valuable, as it significantly reduced the effort required to debug a pipeline. Likewise, the cost was negligent, ~$5/month for an all-purpose server usable for other purposes, compared to $69/month (or $129/month for 2 concurrent jobs) for just Travis.

Only for Coveralls I have failed to find any reasonable alternative. Attempt to collect code coverage data and process them manually failed with the same problems as already mentioned ShoreLine stacks.

However, code coverage is only a single metric of many that are worth watching. The ideal scenario is a fully-fledged Pharo support for some mature code quality service such as SonarQube *[22]*. That is, however, a significant amount of effort, and well outside of the scope of this work.

[1] Whether the overall code coverage is increasing or decreasing.

[2] At a later point, my faculty began providing GitLab CI/CD runner.

Conclusion

Achieved Results

This section summarizes the achievements in respect to the goals defined in the introduction (see *Goals and Objectives*).

Primary Goals

The goal is…

… modeling tool for the data and flow models in NS Theory. The tool provides full support for all primary aspects of data and flow models.

In addition, the tool is continuously adding and improving support for auxiliary elements of both models.

… serve to explore the requirements of users working on NS projects During the entire development, I have maintained a close collaboration with the target user base.

We were able to effectively and efficiently explore needs and requirements for the current state of the modeler.

The tool should focus on the…

… modeling using graphical diagrams All the primary elements can be modeled using the canonical notation (ER diagrams, Flow diagrams).

The modeler provides non-graphical modeling for some auxiliary elements, however the chapter *Diagram Editors* explores the options of providing additional support for graphical modeling of such elements.

… support the inspection and comprehension of models The primary approach to inspection and comprehension is provided via the graphical diagrams. Additional UI explorers, browsers, and inspectors are available.

An ontological inspection is currently being discussed but falls outside the scope of this thesis.

. . . as well as error detection and prevention. The modeler provides many ways to prevent users from creating errors, whether it is in the diagramming interface, or in form-based editors.

Many validation rules (doc:*rules*) were created to detect errors in models created both by the modeler and by the Prime Radiant.

A discussion is provided for the comprehension of the rules by analysts, as well as a potential technology-neutral representation.

Key characteristics of the tool include. . .

. . . user friendliness A close collaboration with the users provided an important feedback to improve aspects of the user interface and the user experience.

The modeler offers the user higher-level tools to construct their models while shielding them by resolving technical details of the model on their behalf. Where it is not possible, various tools provide relevant feedback for the actions the user is performing.

Depending on the context where the user is operating, different ways to access the same functionality is provided for users' comfort.

. . . high flexibility During the collaboration, it was possible to introduce new functionality in very short cycles.

Likewise, it was possible to address technical problems quickly end effectively.

. . . robustness The modeler is tested with different approaches (*Testing, Continuous Integration and Deployment*) to assure a certain level of quality and functionality of the modeler.

If the modeler fails, mechanisms for comprehensive error detection (*Error Tracking and Reporting*) are provided, so problems can be assessed and resolved effectively.

. . . loosely coupled integration with the NS Prime Radiant. The modeler exchanges information with the Prime Radiant via external XML files with a stable format. Thus the modeler is not reliant on the current state and interfaces of the Prime Radiant.

Data retrieved directly from Prime Radiant are stored directly in the modeler, therefore there is no coupling for users. Updates are performed only by the modeler programmer. This data includes option types and contents of base components.

The modeler was field-tested by students of the University of Antwerp in their Normalized Systems course. The students were able to easily construct both their data and flow models, and provide analysis based on their construction.

The modeler is also regularly used by some of the NS developers.

Secondary Goals

The development of the NS modeler forced many improvements in the OpenPonk platform, which in turn can improve other editors developed for OpenPonk. Several new libraries were created to assist with the development of OpenPonk and Normalized Systems. The section *Author's Contributions* summarizes all projects created and/or related to OpenPonk or the NS modeler.

Future Work

The NS modeler is planned to be developed for the coming years, so all future plans and concepts discussed throughout the thesis are very relevant. A summary of them follows.

growing modeling support The discussed metamodel (see chapter *Normalized Systems Metamodel*) continues to grow as more and more parts are being utilized by the modeler.

extended diagramming support Many improvements and additions were discussed for the diagramming notation (*Diagram Editors*).

rules comprehension A better approach to defining rules, possibly even outside the modeler (*Rules and Validations*).

richer integration between PR and the NS modeler A progressive enhancement[1] base integration between the modeler and PR is of interest, e.g., a user could request expansion from the modeler, or full model validation from Prime Radiant.

modeler becoming NS-compliant Although the modeler is not an information system, growth challenges will eventually befall it, assuming that unbounded growth of the modeler is even desirable. Nevertheless, it is worth exploring what would be the challenges associated with normalizing such a graphical tool.

[1] Progressive enhancement is a term used in web design where a core functionality is always provided to every user, but if their browser support novel functionality, the web will automatically offer improved functionality.

Author's Contributions

This work focused primarily on the NS modeler. However since its inception, a lot of parallel effort is going into the underlying platform OpenPonk, Pharo, and its ecosystem, without which the modeler wouldn't be possible in its present scope.

This section briefly summarizes all libraries I have created for OpenPonk, as well as other projects I have created that either directly or indirectly support OpenPonk and/or the modeler; some of which were already described in greater detail in this thesis.

Pharo

Since the inception of OpenPonk, I have been an active contributor to Pharo, and to many libraries and tools in the Pharo ecosystem. My primary contributions were quite naturally in libraries directly relevant to OpenPonk, namely the Roassal visualization library used to provide the diagramming support, and the Spec UI framework.

The contributions include code contributions, debugging, detailed bug reports, extensive discussions, and providing help to new and existing developers alike (Pharo mailing list (2000+ mails), StackOverflow (top 5% contributor to UML, top 10% to Smalltalk and Pharo)).

OpenPonk libraries

Projects under the OpenPonk umbrella.

OpenPonk OpenPonk platform core; extensions and additions for other libraries used by OpenPonk, such as Roassal and Spec.

UML-Metamodel UML 2.5 metamodel implementation.

uml-bootstrap-generator Self-bootstrapping UML 2.5 metamodel generator based on UML specifications.

XMI & UML-XMI XMI (XML Metadata Interchange) reader/writer for UML 2.5.

UML shapes A Roassal abstraction layer for management of hierarchical visual structures, and inspired by UML Diagram Definition/Diagram Interchange specifications. Used by both UML editor, and NS modeler.

uml-profiles & OntoUML profile UML-Metamodel extensions for creating UML profiles and automatically generating their implementations. UML profile for OntoUML.

synchronized-links Utility for creating self-synchronizing bidirectional references between objects. Used by both UML-Metamodel and NS modeler metamodel.

class-editor UML class diagram editor and UML profile editor.

DEMO editor DEMO user-interactive simulator (puppeteered theater simulation).

fsm-editor Finite State Machine editor

borm-editor BORM editor, custom DSL language for BORM models

Pharo libraries

Other projects I have created to either directly or indirectly support OpenPonk and/or Pharo itself.

xmi-analyzer Analysis of XMI-like documents and generation of their equivalent domain code. Predates UML-Metamodel/UML-XMI.

xml-dom-visitor Extension of XMLParser to help with hand-processing of arcane XML files.

xml-magritte-generator (see chapter *Metamodel Engineering*) Generator of domain objects augmented with Magritte support, based on analysis of XML documents.

IconFactory Library for managing Icons inside Pharo.

metalinks-toolkit Library for management of MetaLinks. (MetaLinks are Pharo mechanism for transparent, non-intrusive injection of custom code to other code at runtime.)

live-instance-viewer (presented in *[1]*) Experimental live capture and instance visualization of runtime object structures using MetaLinks. (Live UML instance visualizations.)

git-migration Utility for migrating Pharo code from its custom versioning format to git. Originally written to migrate OpenPonk to git. Continues to be used by many Pharo users.

tonel-migration Utility for migrating Pharo code to new textual representation by performing a deep git history rewrite.

git-fast-writer Support library for git-migration & tonel-migration converting commands and rewrites to Git fast-import format.

file-dialog New file picker dialog for Pharo. To be included in Pharo 7 release.

pharo-trello (to be released) Trello API implementation in Pharo. Originally written to help manage trello tasks during NS modeler development.

pharo-sentry (see chapter *Error Tracking and Reporting*) Sentry error tracking SDK for Pharo.

pharo-changes-builder Wrapper for better management of interactive code generation. Used by all my projects that generate any code (including the NS modeler).

Bibliography

[1] Peter Uhnák and Robert Pergl. Ad-hoc Runtime Object Structure Visualizations with MetaLinks. In *IWST'17: Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, 1–10. ACM Press, 2017. doi:10.1145/3139903.3139912.

[2] Peter Uhnák. Layouting of Diagrams in the DynaCASE Tool. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology. 2016.

[3] Tommaso Dal Sasso. ShoreLine [software]. 2014. [Cited 2018-05-01] Available from: http://shoreline.inf.usi.ch/.

[4] Functional Software, Inc. Sentry [online]. [Cited 2018-05-01] Available from: https://sentry.io.

[5] Peter Uhnák. Pharo-sentry [software]. 2017. [Cited 2018-05-01] Available from: https://github.com/peteruhnak/pharo-sentry.

[6] Feenk. Beacon [software]. http://www.humane-assessment.com/blog/beacon. [Cited 2018-05-01] Available from: http://www.humane-assessment.com/blog/beacon.

[7] Martín Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. Fuel: a fast general purpose object graph serializer: FUEL: A FAST GENERAL PURPOSE OBJECT GRAPH SERIALIZER. *Software: Practice and Experience*, 44(4):433–453, April 2014. doi:10.1002/spe.2136.

[8] Lukas Renggli. Magritte - Meta-Described Web Application Development. 2008.

[9] Norbert Hartl. XML-Bindings [software]. 2010. [Cited 2018-05-01] Available from: https://github.com/magritte-metamodel/XML-Bindings/.

[10] OMG. Unified Modeling Language (UML) v2.5. March 2015. [Cited 2018-05-01] Available from: http://www.omg.org/spec/UML/2.5.

[11] Martin Podloucký and Robert Pergl. Towards Formal Foundations for BORM ORD Validation and Simulation. In *16th International Conference on Enterprise Information Systems*, 315–322. SCITEPRESS - Science and and Technology Publications, 2014. doi:10.5220/0004897603150322.

[12] Herwig Mannaert, Jan Verelst, and Peter De Bruyn. *Normalized Systems Theory: From Foundations for Evolvable Software toward a General Theory for Evolvable Design*. Koppa Digitale Media, 2016. ISBN 978-90-77160-09-1. OCLC: 1021375288.

[13] Peter Uhnák and Robert Pergl. The OpenPonk modeling platform. In *IWST'16: Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, 1–11. ACM Press, 2016. doi:10.1145/2991041.2991055.

[14] Stephen T. Pope and Glenn E. Krasner. A Cookbook for Using Model-View-Controller User Interface Pradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1988.

[15] Alexandre Bergel. *Agile Visualization*. Alexandre Bergel, 2016. ISBN 978-1-365-31409-4.

[16] Christiane Kiesner, Gabriele Taentzer, and Jessica Winkelmann. Visual OCL: A Visual Notation of the Object Constraint Language. 2002. Available from: http://www.user.tu-berlin.de/o.runge/tfs/projekte/vocl/gKTW02.pdf.

[17] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. Renraku: the One Static Analysis Model to Rule Them All. In *IWST'17: Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, 1–10. ACM Press, 2017. doi:10.1145/3139903.3139919.

[18] Denis Kudriashov. Calypso - Pharo system browser [software]. [Cited 2018-05-01] https://github.com/dionisiydk/Calypso.

[19] Denis Kudriashov. Mocketry [software]. [Cited 2018-05-01] Available from: https://github.com/dionisiydk/Mocketry.

[20] GitLab. GitLab [online]. https://about.gitlab.com/. [Cited 2018-05-01] Available from: https://about.gitlab.com/.

[21] GitLab. GitLab CI & CD [online]. https://about.gitlab.com/features/gitlab-ci-cd/. [Cited 2018-05-01] Available from: https://about.gitlab.com/features/gitlab-ci-cd/.

[22] SonarSource S.A. SonarQube [online]. [Cited 2018-05-01] Available from: https://sonarqube.org.

Appendix A

Acronyms

NS	Normalized Systems
PR	Prime Radiant
OP	OpenPonk
ER	Entity-Relationship (ER diagram, ER model)
UML	Unified Modeling Language
OntoUML	Ontological UML
BPMN	Business Process Model and Notation
BORM	Business Object Relation Modeling

Appendix B

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
nsp-windows-1.18.0.zip ........... all-in-one (modeler+source) archive
text ....................................... the thesis text directory
  thesis.pdf ......................... the thesis text in PDF format
```