FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

# Assignment of master's thesis

| | |
|---|---|
| **Title:** | Extending OntoUML Modelling Capabilities on the OpenPonk Platform |
| **Student:** | Bc. Marek Bělohoubek |
| **Supervisor:** | doc. Ing. Robert Pergl, Ph.D. |
| **Study program:** | Informatics |
| **Branch / specialization:** | Web and Software Engineering, specialization Software Engineering |
| **Department:** | Department of Software Engineering |
| **Validity:** | until the end of summer semester 2021/2022 |

## Instructions

The goal of this assignment is to extend OpenPonk's OntoUML modelling capabilities of OntoUML verifications and to implement detection of anti-patterns.

1. Acquaint yourself with the current state of OpenPonk.
2. Design and implement user interface for displaying results of OntoUML verifications.
3. Design and implement a framework for automated updating of OntoUML verification rules from OntoUML.org specifications.
4. Design and implement detection of OntoUML anti-patterns, including the user interface.
5. Extend the OntoUML.org portal with documentation of anti-patterns.
6. Document, test and demonstrate your work on a case study.

*Electronically approved by Ing. Michal Valenta, Ph.D. on 25 January 2021 in Prague.*

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

# Extending OntoUML Modelling Capabilities on the OpenPonk Platform

***Bc. Marek Bělohoubek***

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

May 6, 2021

# Acknowledgements

I would like to thank doc Ing. RobertPergl, Ph.D, my supervisor, for providing guidance and constructive criticism during creation of this work.

# Declaration

In Prague on May 6, 2021 . . . . . . . . . . . . . . . . . . . .

## Citation of this thesis

Bělohoubek, Marek. *Extending OntoUML Modelling Capabilities on the OpenPonk Platform.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

# Abstrakt

Tato práce se zaměřuje na rozšíření možností pro vytváření OntoUML modelů na platformě OpenPonk. Toto rozšíření je rozděleno do čtyř částí. Prvním rozšířením je grafické uživatelské rozhraní pro zobrazování výsledků verifikačního frameworku. Druhá část je prezenotvána novým frameworkem, sloužícím k automatické aktualizaci OunoUML verifikací. Třetím rozšířením je automatická detekce OntoUML anit-patternů. Poslední část se sestává z vybudování nové sekce portálu ontouml.org, obsahující dokumentaci k jednotlivým anti-patternům. V závěru práce je detekce anti-patternů demostrována na referenčním modelu.

**Klíčová slova** OntoUML, OpenPonk, Pharo, Anti-patterny, Aktualizační framework, Uživatelské rozhraní, Unified foundation ontology, Ontologický model, Konceptuální model

# Abstract

This work focuses on extending OntoUML modelling capabilities on the OpenPonk platform. This is done in four parts. First part of the expansion is graphical user interface for displaying results of the verification framework. Second part is represented by new framework, which is used for automatic updating of OntoUML verifications. Third part of the expansion is automatic detection of OntoUML anti-patterns. Last part consists of new section on portal ontouml.org, dedicated to anti-pattern documentation. End of this thesis focuses on demonstration of the anti-pattern detection using reference model.

**Keywords** OntoUML, OpenPonk, Pharo, Anti-patterns, Updating framework, User interface, Unified foundation ontology, Ontological model, Conceptual model

# Contents

# List of Figures

# Introduction

One of the main advantages of the OntoUML modelling languages lies in its methodologies, that allow modeller to check both syntax and semantics of his creations. Those methodologies are defined well enough, that it is possible to incorporate them into tools, that are already used for creation of OntoUML models. One such tool is platform OpenPonk developed by Czech Technical University in Prague using object oriented programming language Pharo.

In last few years this platform introduced its own verification framework used to check syntax of the OntoUML models. It performed its function well, but because it didn't have any dedicated graphical user interface, its usage wasn't very intuitive. Its other issues were more connected to the OntoUML language itself.

OntoUML is under constant development and because of this there can be major changes to its syntax. This created need for automatic process, that would update existing OntoUML verifications implemented inside OpenPonk and keep them up to date.

Lastly even though OpenPonk implemented verification of OntoUML syntax, there was no way to let the platform automatically validate semantics of the OntoUML model, through the anti-pattern detection.

This thesis aims to solve all of the aforementioned problems and its structurally divided into three parts. First part is called ”Review and analysis” and it introduces reader to all necessary theory needed for the practical parts.

Second part ”Design and implementation” and it explains how this thesis solved each of the problems mentioned above.

Third and last part is called ”Testing and documentation” and it focuses on unit tests for the new code, exemplifies anti-pattern detection on a reference model and mentions documentation for both anti-patterns and new code.

# Goals

Main goal of this master's thesis is to extend OntoUML modelling capabilities of OpenPonk platform by following functionalities: creation of graphical user interface for displaying verification results, development of updating framework and its integration into the existing verification framework and implementation of anti-pattern detection. All of the aforementioned extensions have to be both tested and documented.

First, it is necessary to review and analyse OntoUML language (both its basic concepts and anti-pattern definitions), OpenPonk platform as a whole (including Pharo programming language) and the existing OntoUML verification framework.

Second part is dedicated to design and implementation of the new functionalities. All of them had to be integrated into the existing verification framework with great emphasis on ease of use, modifiability and expandability. Reason for those requirements is constant development of both OntoUML modelling language and OpenPonk platform, which can result of major changes in parts of the verification framework.

Third, newly implemented code has to be verified using unit tests and both the new user interface and anti-pattern detection should be demonstrated on reference model.

Final part of this thesis is documentation. This should consist of three parts: documentation present in this thesis, documentation in the newly implemented code (OpenPonk platform allows any user to customise modelling environment according to his needs) and documentation if form of new section on portal ontouml.org dedicated to anti-patterns.

# Part I

# Review and analysis

Chapter 1

# OntoUML and OpenPonk review

Start of this chapter briefly talks about OntoUML modelling language, next two sections introduce reader to Pharo programming language and OpenPonk platform but the main focus is on the last section describing OntoUML verification framework, which was created and implemented as part of authors bachelor's thesis [1] for the OpenPonk platform and which was significantly extended as part of this work.

Please note that due to the size of the topic, anti-patterns are not discussed here, instead they have their own chapter, to increase clarity of this work.

## 1.1 OntoUML

Information contained in this section were obtained from following sources [2, 3, 4, 5, 6].

OntoUML modelling language was created to provide comprehensive set of ontological theories, that would cover all fundamental conceptual modelling and resolve many of the problems faced during conceptual modelling.

It is based on modal expansion of predicate logic called modal logic. Modality play a great importance in the language as it allows to model reality much more accurately. Another important part of the language are principles of identity, generalization and rigidity, that together with sortal and non-sortal classifications define all OntoUML stereotypes.

Significant part of anti-patterns presented in this thesis, is result of either misunderstanding of those core principles or misuse of stereotypes, it is therefore necessary to give the reader brief description of all aforementioned principles starting with modal logic.

### 1.1.1 Modal logic

Modal logic is expansion of predicate logic and thus it contains all predicate quantifiers (universal and existential) and logical operations (conjunction, disjunction, negation, implication and equivalence).

Modality itself allows us to express how the predicate formulas hold in different *worlds* - different space and time. This is done with help of two modal operators: possibility and necessity.

| | |
|---|---|
| **Possibility operator** | is represented by symbol $\Diamond$ and any statement beginning with it is true in *at least one* world. |
| **Necessity operator** | is represented by symbol $\Box$ and any statement beginning with it is true in *all* worlds. |

To exemplify, statement "In all words, every ship that has hole under it's waterline is sinking." would have following representation in modal logic:

$$\Box(\forall ship) : HasHole(ship) \Rightarrow IsSinking(ship)$$

### 1.1.2 Rigidity principle

Rigidity as a principle is closely tied to modality. It defines mutability of the type - it defines if individuals can become or cease to be instances of the type without having to altering their identity.

OntoUML distinguishes between four types of rigidity: rigid, anti-rigid, semi-rigid and non-rigid.

- *A **rigid** ($R_+$) type T is one that classifies its instances necessarily (in the modal sense), i.e., the instances of that type cannot cease to be so without ceasing to exist.* [3] In modal logic:

  $$R_+(T) = \Box(\forall x)(T(x) \Rightarrow \Box(T(x)))$$

- Type T is **anti-rigid** ($R_-$) when individual instantiates the type in some possible world, there *has to* be another world in which the individual exists but it's not instance of the anti-rigid type T. In modal logic:

  $$R_-(T) = \Box(\forall x)(T(x) \Rightarrow \Diamond(\neg T(x)))$$

- **Semi-rigid** ($R_\sim$) type T is rigid for some of it's instances and anti-rigid to others. In modal logic:

- **Non-rigidity** ($NR$) is logical negation of rigidity. It states that if individual instantiates non-rigid type T in one world, there *may* be another word in which the individual exists but it's not instance of the non-rigid type T. In modal logic:

  $$NR(T) = \Diamond(\exists x)(T(x) \land \Diamond(\neg T(x)))$$

### 1.1.3 Identity principle

Every entity in the real words has it's identity. Identity is something that differentiates particular entity from other entities of the same type. Identity has to be domain unique and immutable (it has to have the same value in all worlds).

To exemplify, let's look at type employee in relational database. Every employee has it's name, address and personal identification number. Which of those is it's identity? It cannot be the name as it doesn't guarantee uniqueness (there are many people called John Smith, but we can still differentiate between them).

Using the same logic rules out the address as it is neither domain unique nor immutable (multiple persons can live on the same address and the same person can move between multiple addresses during it's life).

This leaves us only with option of personal identification number, which (as its name suggests) is both unique and immutable, so in our case it represents the employee identity.

Example above demonstrates two things, first the importance of identity as a principle - if the employee had no identity we wouldn't be able to distinguish between different employees and thus for example we wouldn't be able pay them.

Second, defining identity for entities in particular domain isn't always as easy as was shown in the example. Look at entity painting for example the famous painting of Mona Lisa by Leonardo da Vinchi. We can say that it's identity lies in the shapes and colours of the image. But then if we paint a black line through the middle of the painting, it would by our definition lose its previous identity and become new entity.

### 1.1.4 Generalization

Much like modal logic is expanding on predicate logic, OntoUML is expanding on UML. This also means that many of the basic concepts of OntoUML are carried over from UML, generalization being one of them.

Generalizations are used to create hierarchy between types, defining subtypes that inherit all properties, methods and relations from subtypes. Multiple generalizations originating from the same supertype can be also grouped into one or more generalization sets.

Advantage of this grouping lies in the two meta-properties of generalization set. Those meta-properties are *disjoint* and *complete*. *Disjoint* states that there can be no individual that instantiates more than one of the subtypes in the generalization set, while *covering* states that any individual that instantiates supertype has to also instantiate one or more of the subtypes.

### 1.1.5 Sortals, non-sortals and aspects

Difference between *sortal* and *non-sortal* types is tied to the identity principle. *Sortal* types have identity principle and either have or require identity. Entities in source domain are represented via *sortal* types. Examples of sortals are kinds "car", "dog" and "tree", collective "deck of cards" ...

*Non-sortals* don't have identity principle and so they cannot have or provide identity. *Non-sortals* are used to model (usually abstract) concepts that are shared by multiple types in the same domain. Examples of non-sortals are role "customer", phases "healthy" and "ill" ...

*Aspects* represent features and properties of both *sortals* and *non-sortals*. They can be either structured/measurable in this case represented by *aspect* quality, or non-structured/non-measurable represented by *aspect* mode. Examples of aspects are qualities "price", "weight" and "length", modes "intention", "ability", "mood" ...

## 1.2 Pharo

Open-source programming language Pharo (downloadable from [7]) started as dialect of other object oriented programming language SmallTalk. It's authors describe it in a following way:

*Pharo is a pure object-oriented programming language and a powerful environment, focused on simplicity and immediate feedback (think IDE and OS rolled into one).* [8]

Pharo (documented mostly in its code, with additional documentation here [9]) has three main features that can be considered it's advantages. First, there is its syntax, that is compact enough to fit on a postcard as can be seen on its official summary here [10].

Second feature is the aforementioned combination of both IDE (integrated development environment) and OS (operation system), this results in live environment that allows programmer to look at any instance of any object in the environment, inspect it, interact with it or even rewrite its source code.

Lastly, pharo is purely object-oriented and dynamically typed language. Pure object-orientation means that every single element of Pharo from the elementary data types to the UI and developer tools is an object. This combined with the fact that checks for object types happen only during runtime (dynamic typing) allows Pharo to have simple, yet highly expressive syntax and programmers to create highly modular object-oriented code.

## 1.3 OpenPonk

*OpenPonk is a metamodeling platform and a modeling workbench implemented in the dynamic environment Pharo aimed at supporting activities surrounding* *software and business engineering such as modeling, execution, simulation, source code generation, etc.* [11]

OpenPonk (downloadable from [12]) is written in Pharo programming language and its currently developed by Centre For Conceptual Modeling and Implementation on Czech technical university in Prague.

It supports several modelling languages like: Business Objects Relation Modeling Object-Relation Diagrams (BORM ORD), Petri nets, UML class diagrams and most importantly OntoUML.

Following subsections look at most important parts of OpenPonk data model with emphasis on OntoUML profile and data model of OntoUML verification framework.

Information contained in this section and its subsections was obtained from [13, 1] and analysis of the OpenPonk source code.

### 1.3.1 Data model

In this section we look at classes from OpenPonk data model and OntoUML profile. We start with description of ancestors of most other classes discussed in this section classes `OPUMLMetaElement` and `OPUMLElement`.

We continue with classes that represent the model, entities and relations. Then we look at generalization, generalization set and we end this subsection by inspecting classes from the OntoUML profile.

#### 1.3.1.1 OPUMLMetaElement and OPUMLElement

Classes `OPUMLMetaElement` and `OPUMLElement` are important because they stand at the top of the hierarchy. Neither of them are directly instantiated during the model creation, but they define important parts of interface used by other classes.

`OPUMLMetaElement` is abstract class. It's methods `applyStereotype:`, `appliedStereotypes` and `oclIsKindOf` are the foundation for applying and retrieving stereotypes.

Abstract class `OPUMLElement` is direct subtype of `OPUMLMetaElement` and ancestor of all other `OPUML` classes discussed in this section. It is responsible for applying composition through methods `initializeSharedGeneralizations` and `initializeDirectGeneralizations`.

#### 1.3.1.2 OPUMLModel

Class `OPUMLModel` holds all entities, generalizations and relations in the model, making it the "data representation" of the conceptual model itself. It most important method is `packagedElements` which returns collection of all entities and relations in the model.

#### 1.3.1.3 OPUMLClass

Entities in from conceptual model are mapped to instances of `OPUMLClass`. Because `OPUMLClass` is subtype of `OPUMLElement` we can apply of stereotypes to it and create OntoUML entities.

#### 1.3.1.4 OPUMLAssociation

Instances of `OPUMLAssociation` are used to represent relations from conceptual model. Similarly to `OPUMLClass` we can also apply stereotypes to it and this allows it to represent all implemented OntoUML relations.

#### 1.3.1.5 OPUMLGeneralization

Generalizations from conceptual model are stored inside `OPUMLModel` mapped as instances of `OPUMLGeneralization`.

It has three notable methods. First two `general` and `specific` are used for access connected supertype and subtype respectively. Third important method is `generalizationSets`, it returns collection with all generalizations sets that contain this instance of `OPUMLGeneralization`.

#### 1.3.1.6 OPUMLGeneralizationSet

Generalization sets are represented by instances of `OPUMLGeneralizationSet`. References to the instances of `OPUMLGeneralization` are accessed through method `generalizations`.

It also holds two important properties accessible by methods `isCovering` (generalizations cover all possible options) and `isDisjoint` (instance of subtype defined by one generalization cannot be instance of subtype defined by any other generalization in the set).

#### 1.3.1.7 OntoUML stereotypes

Classes representing OntoUML stereotypes are subtypes of `OPUMLElement`. Name of each class starts with `OntoUML` and is followed by the name of the stereotype (for example `OntoUMLKind`).

Hierarchy for the stereotypes is created through composition with abstract ”stereotypes” like `OntoUMLSortal` standing at the top of the hierarchy.

### 1.3.2 OntoUML verification framework

OntoUML verification framework was created as by author of this master’s thesis as part of his bachelor’s thesis [1], that will be also main source for this subsection.

First let's look at class diagram (figure 1.1) of the entire verification framework (please note that it contains only one concrete subtype for `Verification` and `StereotypeVerification` for increased clarity).

As we can see from the diagram, verification framework consist of five main classes: `VerificationController`, `VerificationResults`, `Verification`, `StereotypeVerification` and `VerificationMessage`. Also there is one additional trait `TProfileGatherer`, which is special type of class, that can be applied through composition to other classes.

Following subsections are dedicated to the aforementioned classes and trait starting with `VerificationController`.

Please be aware that following sections describe state of the verification framework after it's creation and some part's might already changed. This text is being written after finishing the practical part of this thesis which has been already integrated to the current version of OpenPonk and OntoUML profile.

#### 1.3.2.1 VerificationController

`VerificationController` is used to both start and receive results of verifications. It is located in package `OntoUML-VerificationControllers` along with it's testes.

`VerificationController` has following methods:

**`verify:`** *Instance method*, runs all verifications on `OPUMLModel` passed in parameter and returns their results in form of `VerificationResults` instance.

**`verifySingleObject:`**[1] *Instance method*, applies all verifications to single object passed in parameter and stores their results into instances of `VerificationResults`.

**`getVerifications`** *Instance method*, that returns all verifications classes from `OntoUML-Verifications` package.

**`getVerifiedObjects`** *Instance method*, returns collection of `OPUMLElements` from the `OPUMLModel`, that was passed as parameter.

We would like to provide more information about the implementation of the `getVerifications` method. Verification framework was created with expansion in mind and this is can be best seen in the aforementioned method.

Method `getVerifications` takes advantage of ”living environment” provided by Pharo, that allows it to access source code of the verification framework during runtime. Using this feature `getVerifications` is able to load all verifications classes during the runtime and thus it is possible to insert new verification by simply adding them in to the appropriate package.

[1] `verifySingleObject:WithVerifications:WithModel:WithResults:`

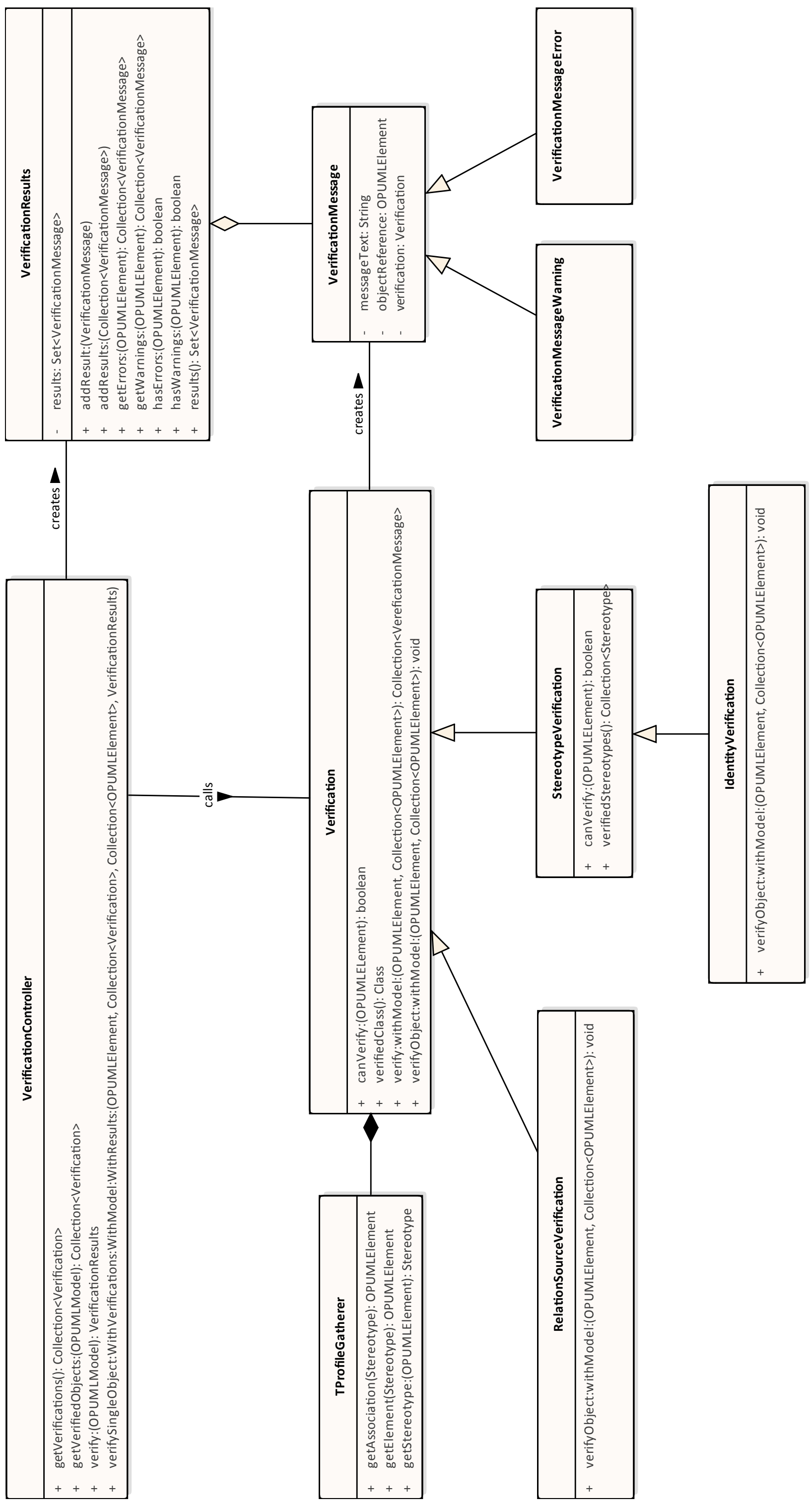

VerificationResults
- results: Set<VerificationMessage>
+ addResult:(VerificationMessage)
+ addResults:(Collection<VerificationMessage>)
+ getErrors:(OPUMLElement): Collection<VerificationMessage>
+ getWarnings:(OPUMLElement): Collection<VerificationMessage>
+ hasErrors:(OPUMLElement): boolean
+ hasWarnings:(OPUMLElement): boolean
+ results(): Set<VerificationMessage>
VerificationMessage
- messageText: String
- objectReference: OPUMLElement
- verification: Verification
VerificationMessageError
VerificationMessageWarning
creates
creates
VerificationController
+ getVerifications(): Collection<Verification>
+ getVerifiedObjects:(OPUMLModel): Collection<Verification>
+ verify:(OPUMLModel): VerificationResults
+ verifySingleObject:WithVerifications:WithModel:WithResults:(OPUMLElement, Collection<Verification>, Collection<OPUMLElement>, VerificationResults)
calls
Verification
+ canVerify:(OPUMLElement): boolean
+ verifiedClass(): Class
+ verify:withModel:(OPUMLElement, Collection<OPUMLElement>): Collection<VerificationMessage>
+ verifyObject:withModel:(OPUMLElement, Collection<OPUMLElement>): void
StereotypeVerification
+ canVerify:(OPUMLElement): boolean
+ verifiedStereotypes(): Collection<Stereotype>
IdentityVerification
+ verifyObject:withModel:(OPUMLElement, Collection<OPUMLElement>): void
TProfileGatherer
+ getAssociation(Stereotype): OPUMLElement
+ getElement(Stereotype): OPUMLElement
+ getStereotype:(OPUMLElement): Stereotype
RelationSourceVerification
+ verifyObject:withModel:(OPUMLElement, Collection<OPUMLElement>): void

Figure 1.1: Class model of the verification framework [1]

#### 1.3.2.2 VerificationResults

As its name suggests, class `VerificationResults` contains collection with outcomes of all applied verification. It's methods allow both adding and retrieving results, but following text contains descriptions only for the three most important accessors.

| | |
|---|---|
| **`results`** | *Instance method*, that returns collection with all stored verification results. |
| **`hasErrors:`** | *Instance method*, that looks if results contain at least one instance of `VerificationMessageError` for entity, relation or generalization passed as parameter. |
| **`hasWarnings:`** | *Instance method*, that looks if results contain at least one instance of `VerificationMessageWarning` for entity, relation or generalization passed as parameter. |

#### 1.3.2.3 VerificationMessage

`VerificationMessage` was created as abstract class to define uniform interface for negative results, i.e., all problems found by verifications in the verified model.

All methods defined by it, are simple getters and setters for: reference to verified object, reference to instance of the verification function, short description of the problem and severity of the find.

Last mentioned method `severity` is implemented in subclasses responsible for representing errors and warnings `VerificationMessageError` and `VerificationMessageWarning`.

#### 1.3.2.4 Verification and SteretypeVerification

Abstract class `Verification` and its direct subclass `SteretypeVerification` define interface for all other verifications. Interface of class `Verification` consist of following four methods.

| | |
|---|---|
| **`verify:withModel:`** | *Class method*, that returns collection containing instances of `VerificationMessage` if the verification is applicable to the verified model and if it discovered problems in the model. Otherwise returns empty collection. |
| **`canVerify:`** | *Class method*, checks if the verification can be applied to the verified object, based on class of the verified object. |

| | |
|---|---|
| **`verifiedClass`** | *Class method*, returns class, to which this verification can be applied. |
| **`verifyObject:withModel:`** | *Instance method*, that is responsible for implementing the verification process, returns collection of `VerificationMessage`s (it may be empty). |

`SteretypeVerification` overrides method `canVerify:` and adds one more method `verifiedStereotype`.

| | |
|---|---|
| **`canVerify:`** | *Class method*, checks if the verification can be applied to the verified object, based on class and stereotype of the verified object. |
| **`verifiedStereotype`** | *Class method*, returns collection of stereotypes that can be verified by this verification. |

All other verifications are direct subclasses of either `Verification` or `SteretypeVerification` by overriding methods `verifyObject:withModel:`, `verifiedClass` and if needed also `verifiedStereotype`.

#### 1.3.2.5 TProfileGatherer

Lastly there is train `TProfileGatherer`, that contains three methods for working with stereotypes and provides them to other classes through composition.

| | |
|---|---|
| **`getStereotype:`** | *Class method*, that returns stereotype of entity or relation, that was provided as parameter. |
| **`getElement:`** | *Class method*, that returns entity, to which is the stereotype (provided as method parameter) applied. |
| **`getAssociation:`** | *Class method*, returning relation, to which the stereotype (provided as method parameter) is applied. |

Chapter 2

# Anti-patterns

Purpose of this chapter is to introduce anti-patterns as a concept and provide information for all OntoUML anti-patterns, that were implemented as part of this work.

First section explains basics of anti-pattern concept. It is followed by twenty sections of concrete anti-patterns. Each anti-pattern section contains it's definition, guide for refactoring it's occurences and example of one such occurrence.

Name of each anti-pattern section also contains it's abbreviation. Please note that for the clarity of this text we refer to all anti-pattern using those abbreviations.

Some sections contain definitions for additional OntoUML concepts, in those cases both refactoring guide and example of occurrence are in separate section named after the anti-pattern abbreviation.

This chapter was written using information from following sources [14, 15, 16], but the main source was [17] as it was at the time of creation of this work, only publication that contained definitions for all following anti-patterns (with most of them being defined there).

## 2.1 Binary relation between overlapping types (BinOver)

Name of this anti-pattern its definition, but before we can talk about it properly, it is necessary to understand the concepts of overlapping and disjoint sets of types.

### 2.1.1 Overlapping and disjoint sets

We will first start with the ”informal definitions” and examples that should help the reader to grasp the general concepts of *overlapping* and *disjoint* sets.

End of this subsection contains formal definitions for both *overlapping* and *disjoint* sets.

Set of types is considered *overlapping* when it's possible for single individual to instantiate all types in the set at the same time. Example for *overlapping* set would be set consisting of types brother, father and son as it is possible that single individual will instantiate the entire simultaneously.

Set of types is considered *disjoint* when it's impossible for single individual to instantiate more than one type from set at the same time. Example for *disjoint* set would be set consisting of types healthy, ill and dead as it is not possible for a single individual to instantiate more than one of those types.

***(Overlapping Set):*** *Let $W$ be a non-empty set of possible worlds, $w \in W$ be a specific world, $T$ the set of types, $t \in T$ be a particular type, $ext_w(t)$*[2] *the extension of a $t$ in world $w$ and $exists(w)$ the function that return all individuals that exists in a world $w$. A set of types is overlapping if there is at least one $w$, such that:* [17]

$$\forall t, t' \in T, \exists x, x \in exists(w) \wedge x \in ext_w(t) \wedge x \in ext_w(t')$$

**Definition (Disjoint Set):** Making the same conventions as in the previous definition, a set of types is disjoint, if for every $w$:

$$\forall t, t' \in T, t \neq t' \rightarrow \nexists x, x \in exists(w) \wedge x \in ext_w(t) \wedge x \in ext_w(t')$$

### 2.1.2 BinOver anti-pattern

Any binary relation between overlapping types leads to occurrence of BinOver anti-pattern. It is important to look out for such structures, because the overlap means that it's possible to have the same type and even same individual on both ends of the relation.

Sometimes we need to model structures where it is possible or even required to have the same type on both ends of the relation, but in most cases like this we don't want the possibility of the same individual on both ends.

Good example for this would be entity ”person” and formal association ”parentOf” (see figure 2.1). While we need to type ”person” to be both source and target of the association, it makes no sense for the same individual to be its own parent/child. In this case the solution would be to create OCL invariant and use it to enforce acyclicity of the association. Note that in other cases changing stereotype of the association may also be a possibility.

In case when the association connects two different but overlapping types, we may need to enforce disjointness of those types. To exemplify, look at the figure 2.2. It contains small model for a prison that contains only guards and prisoners. Both types ”guard” and ”prisoner” were classified as roles and both of them specialize kind ”person”. In addition to that we also have formal association ”guards” from role ”guard” to role ”prisoner”.

[2]We use the function $ext_w(t)$ as defined in [2]

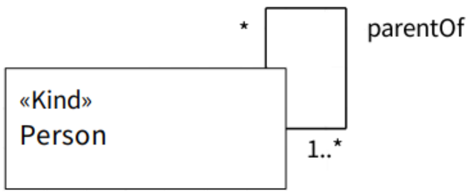


Figure 2.1: First example of binary relation between overlapping types

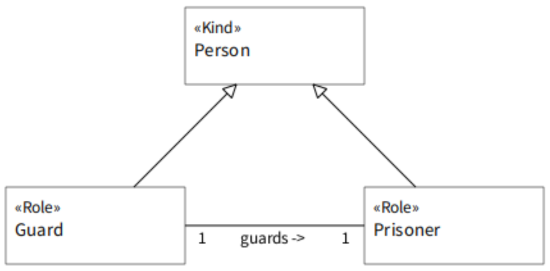


Figure 2.2: Second example of binary relation between overlapping types

All may look well, but after more thorough analysis, we can see that the model allows single person to be both guard and prisoner at once. Not only that but this ”prisoner guard” can also be guarding himself. In this case the problem lies in the generalization set and so does the fix. We have to simply set the meta-attribute *isDisjoint* on the generalization set to true and this will both prevent the same person to guard itself but it will also prevent the same person to be both guard and prisoner at the same time.

## 2.2 Deceiving intersection (DecInt)

Deceiving intersection anti-pattern aims to investigate if subtype (stereotyped as subkind, phase, role, mode or relator) with multiple generalizations to concrete types is case of intentional or derived subtyping and if its extension is not empty.

We can see the entire definition of DecInt anti-pattern above, but to fully understand it we will have to talk about concepts of concrete type, intentional and derived subtyping and lastly about empty extensions.

### 2.2.1 Concrete type

To be considered concrete, type has to fulfil two conditions. Its meta-attribute *isAbstract* has to be set to false and all generalizations set that aggregate generalizations to the concrete type must have their meta meta-attribute isCovering set to false. In another words, type is considered concrete, if it's possible to create instance of said type, that is not instance of one of its child types.

In figures bellow we can see three examples. In all three examples we have entity "person" stereotyped as kind which has meta-attribute *isAbstract* set to false. In first case (figure 2.3) we model that person can become a doctor or a teacher, but since person can also be neither doctor, nor teacher, we did set meta-attribute isCovering of the generalization set to false. This means that kind person concrete type in the first example.

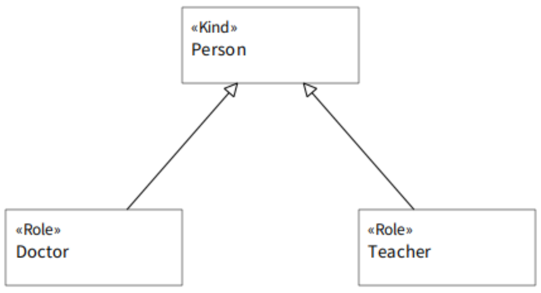


Figure 2.3: Concrete type - roles of person

For the second example (figure 2.4) we have again kind person, but this time we model life "states" of the person. In order to keep the model simple, we say that each person can be either alive or dead. Those two states are modelled as phases and because of that we have to set meta-attribute *isCovering* to true for their generalization set. Kind "person" can be now instantiated only if the instance also instantiates either phase alive or phase dead, thus in the second example kind "person" is not considered as concrete type.

Third example (figure 2.5) combines the first two together. Again, we have kind "person" and it is specialized by two generalization sets. First set aggregates generalizations for roles "doctor" and "teacher", while the second set contains generalizations for phases "alive" and "dead". Even though the generalization set for roles has its meta-attribute *isCovering* set to false and thus allows instantiation of person that is neither doctor nor teacher, we still have to look at the second generalization set that aggregates generalization for phases. Since this second generalization set has its meta-attribute *isCovering* set to true, it is not possible to instantiate person that is neither alive nor dead and this means that the kind "person" is not concrete type in the third example.

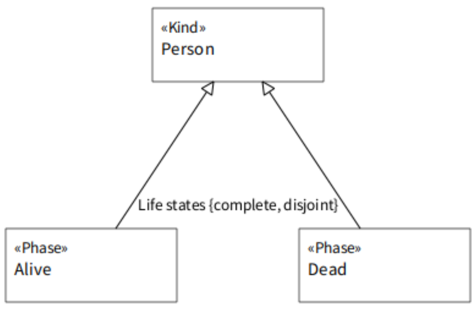


Figure 2.4: Concrete type - life states of person

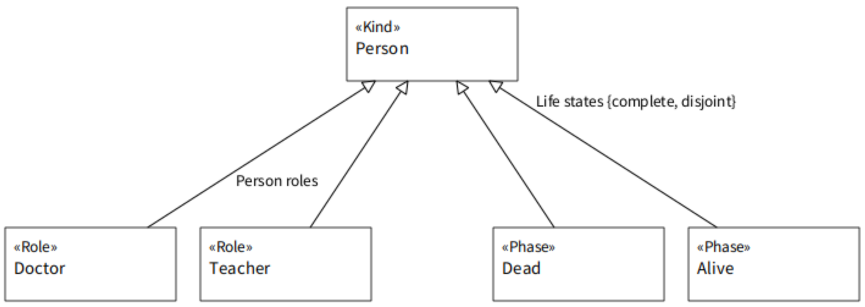


Figure 2.5: Concrete type - life states and roles of person

### 2.2.2 Intentional and derived subtyping

If we use generalization to characterize subtypes, by adding complementary characteristic, we would be using intentional subtyping. Other option is to use derived subtyping, i.e., use generalization to select subset of subtypes based on one or more properties of the parent type (characterization by restriction).

In figure 2.6 we can see usage of both subtyping methods. At the top of the generalization tree there is a ”2D object”, that has height and width. It is subtyped by ”3D object”, that adds depth thus it is case of intentional subtyping. Lastly the ”3D object” is further subtyped by ”sphere”, that restricts the shape of the object, making it an example of derived subtyping.

### 2.2.3 Empty extensions

By using multiple generalizations for the same type, we run into the hazard of creation empty extensions. This happens when two or more parents of the modelled type provide/inherit different identity principles or if they are

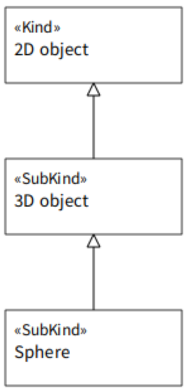


Figure 2.6: Example of intentional and derived subtyping

mutually disjoint due to their generalization set.

To exemplify let's look again on person and its live "state". Similarly, to example in section about concrete types, we say that each person can be either dead or alive and both of those states will be modelled as distinct phases.

Now let's assume that we are creating this model for a game or a fantasy story and we want to model new state "undead" as being both "alive" and "dead". In the figure 2.7 we have stereotyped it as phase and made it subtype of both "dead" and "aliv"e. Problem is that "dead" and "alive" are mutually exclusive (their generalization set is disjoint) and this consequently means that we cannot instantiate phase "undead" making it an empty extension.

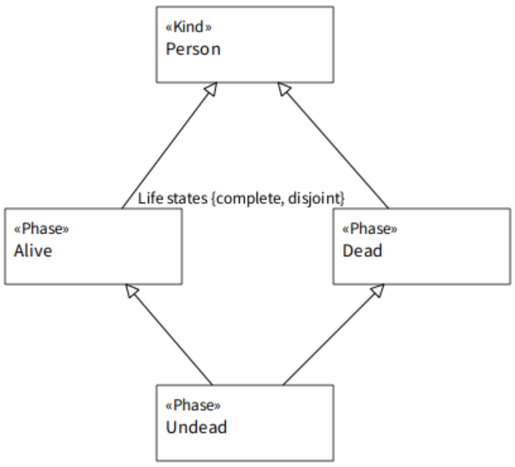


Figure 2.7: Example of empty extension

### 2.2.4 DecInt anti-pattern

Since we have already encountered the full definition of DecInt anti-pattern in the beginning of this section we will instead continue straight to the examples.

We want to create model (figure 2.8) for small company that sells its products to other companies and individuals. We have relator ”contract” with two mediations: one to kind ”company” and second to role ”customer”. Role ”customer” is further specialized by roles ”individual customer” and ”company customer”. Individual customer also specializes kind ”person”, while ”company customer” specializes kind ”company”.

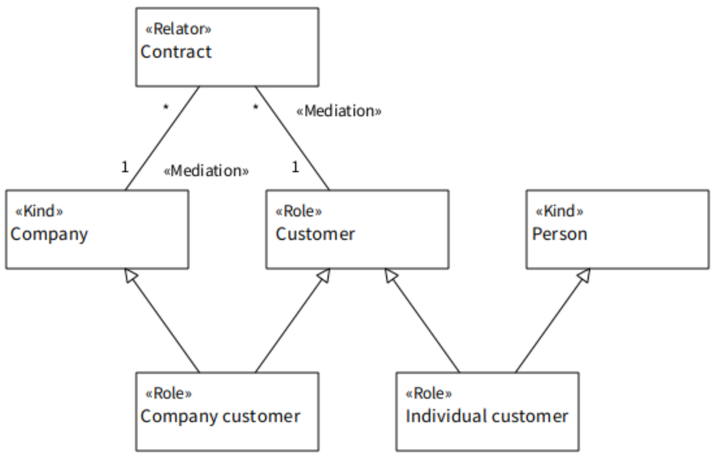


Figure 2.8: Example of decieving intersection occurrence

In case of this model both ”individual customer” and ”company customer” would cause DecInt anti-pattern to occur, as both of them have two generalization to concrete types (”person” and ”customer” for ”individual customer”; ”company” and ”customer” for ”company customer”) and there is no generalization set with isCovering meta-attribute set to true. (Also the role ”customer” hasn’t got any identity provider).

To fix this we should do two things. First, we should change stereotype of ”customer” from role to roleMixin. This would take care of both occurrences of DecInt anti-pattern since roleMixin is abstract by definition and thus any type stereotyped by it cannot be concrete type.

Second, we should also set meta-attribute isDistinct on generalization set that aggregates both generalizations to ”customer” to true. Optionally we should also make the aforementioned generalization set covering, if there are no other types of customer.

## 2.3 Relationally dependent phase (DepPhase)

Before we start with the description of the anti-pattern itself, it is crucial to about the difference between stereotypes phase and role.

### 2.3.1 Phase

Phase represents state of the object that can change in time and this state is directly connected to some intrinsic property of the entity. Changes in this intrinsic property can (and often will) lead to instantiation of new phase. For example, entity ”person” can be subtyped by phases ”child” and ”adult”, which are both based on property ”age”.

### 2.3.2 Role

Role on the other hand is used to specialize entities in relation context, thus its instantiation and destruction depends on existence of a connected relation. Example for this can be entity ”person” subtyped by roles ”employer” and ”employee” connected through relator ”contract” via mediations.

### 2.3.3 DepPhase anti-pattern

DepPhase anti-pattern occurs when we mix the concepts of role and phase. It is identified by phase being directly connected to other entity via mediation. Further analysis of this structure can lead to one of three following possibilities.

#### 2.3.3.1 Phase misused as role

First and probably most common possibility for occurrence of DepPhase anti-pattern, is using phase when we should have used the role.

In example 2.9 you can see simple model for small company. We have entity ”person” subtyped with ”employer” and ”employee” that are mediated through ”contract”. Both ”person” and ”contract” were correctly stereotyped as kind and relator respectively, but the modeller made a mistake and stereotyped both ”employer” and ”employee” as phase instead of role, thus creating instance of DepPhase anti-pattern for both phases.

#### 2.3.3.2 Relational dependency owned, but should be inherited

Second possibility is that suspicious part of model represents situation in which relational dependency should not be owned, but instead inherited by the phase.

For example, let’s suppose that we are creating model (figure 2.10) for a hospital. We have entity ”person” and we want to model that ”person” can be a ”patient” and each” patient” can be either ”healthy” or ”ill”. ”Person”

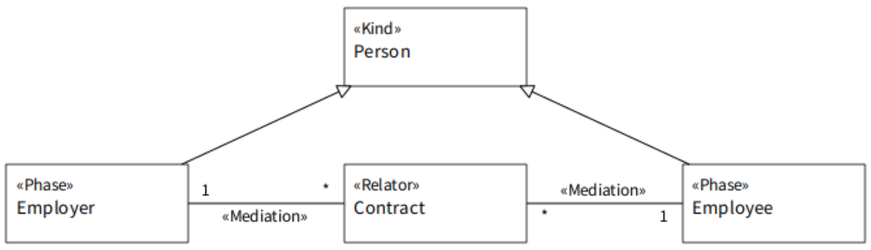


Figure 2.9: Example of phase misused as role

becomes ”patient” when it is registered in the hospital. As you can see in the model bellow, we have chosen to model this by subtyping kind ”person” with phases ”ill patient” and ”healthy patient” and connecting them to the ”patient card” relator via mediation.

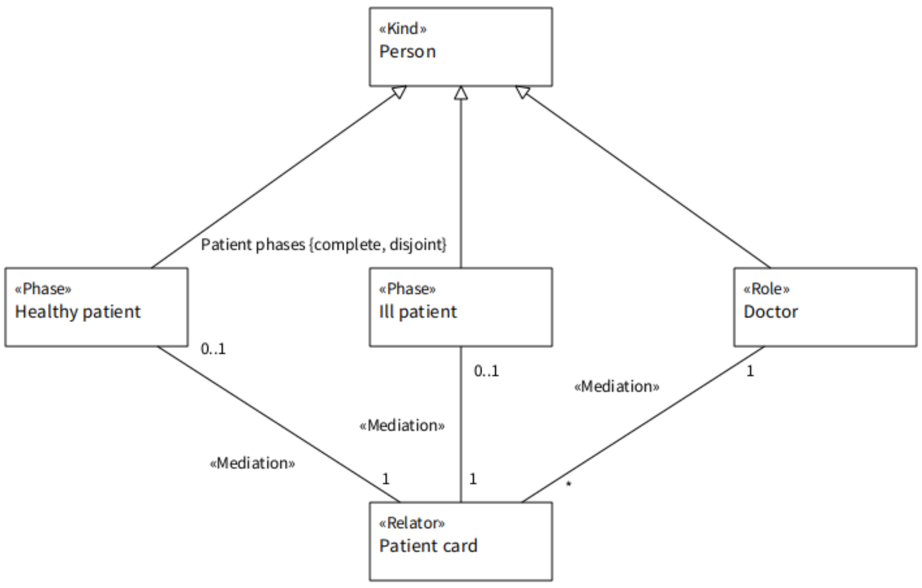


Figure 2.10: Example of relational dependecy, that is owned, but should be inherited instead

This is another occurrence of DepPhase anti-pattern and as you can see in the second diagram, we should have subtyped kind ”person” with role ”patient”, that is both connected to the ”patient card” relator via mediation, and further subtyped by phases ”ill” and ”healthy”.

#### 2.3.3.3 Phase characterized by intrinsic property and relation

Third possibility is that the phase is characterized by both change in an intrinsic property and creation of relational property.

This can be seen in the next example in which we model car and its possible states. We have entity ”car” stereotyped as kind, that subtyped by phases ”damaged” and ”operational”. Both phases are characterized by intrinsic properties of the ”car”, but for the ”car” to be considered legally ”operational” is also needs ”MOT” certificate from ”certification authority”, thus requiring relational property.

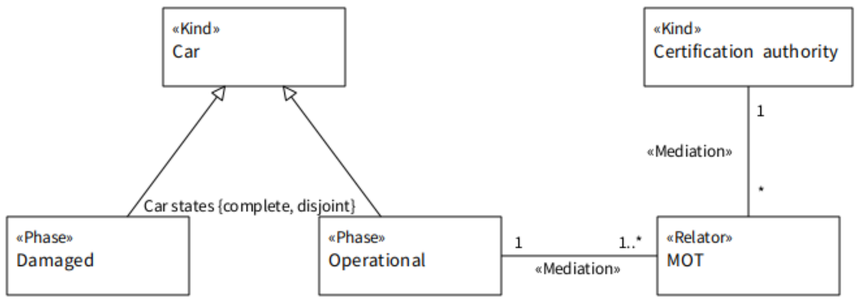


Figure 2.11: Example of phase characterised by both intrinsic property and relation

In this particular case, when phase is characterized by intrinsic property and relational property, DepPhase anti-pattern does not occur and we get a false alarm.

## 2.4 Free role specialization (FreeRole)

As we have discussed previously in section about DepPhase anti-pattern, role is relationally dependent type and as such it has to be connected to relator via mediation. It is possible to fulfil this dependency indirectly by specializing role or rolemixin that is connected to relator through mediation (again this specialization may be done indirectly through multiple ancestors).

Even though this indirect fulfilment is possible according to the OntoUML specification, such structures require increased attention, since they might indicate that the model is still missing some parts. There is however one more condition FreeRole anti-patter *does not* occur when the role and all its ancestors are indirectly connected to mediation through rolemixin ancestor.

There are four main reasons for the free role occurrences corresponding to four role specialization patterns: derived sub-role, intentional sub-role, material sub-role and role of role. Whenever we investigate a free role occurrence it is crucial to identify the reason / role specialization for the occurrence, since it also provides a solution for correcting the model.

Now to the reasons / role specialization patterns themselves.

### 2.4.1 Derived sub-role

The derived sub-role pattern applies to free-roles when they are instantiated according to a pre-determined set of conditions [17].

Informally we can think about derived sub-role as a ”phase of role”, since it creates structure similar to phase specialization. Note the emphasis on informality, unlike phase derived sub-role doesn’t require to be part of distinct generalization set.

Example for derived sub-role can be seen in figure 2.12 containing classification of drivers. Person becomes driver, when she obtains driving license from certification authority. Statistics also use terms inexperienced driver for drivers that have their driving license for less than two years, and experienced driver for others.

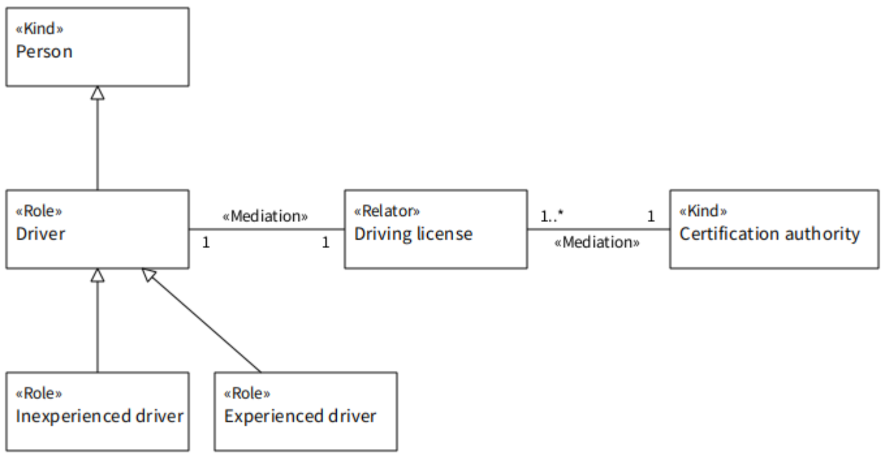


Figure 2.12: Example of derived sub-role

Stereotypes in the model were chosen as follows: both ”person” and ”certification authority” are kinds, ”driving license” is relator and all three ”drivers” are roles. According to the specification above both ”experienced driver” and ”inexperienced drive”r are instantiated according to set conditions (in our case time since obtaining their drivers license) and are examples of derived sub-role.

### 2.4.2 Role of role

This pattern is used when we need to define set of roles that require specific order in which they can be instantiated. Each role in the ”specification tree” has to have connection to independent relator to represent each ”step” in the role instantiation hierarchy.

To exemplify please refer to figure 2.13 that contains simplified model for athletes and Olympic representants. For the purposes of the professional sport, person becomes athlete when she has its own record in athletics ladder for the appropriate sport (maintained by athletics association). Athlete becomes representative after being chosen by Olympic committee from its country, based on its previous performances represented by his rank on the ladder.

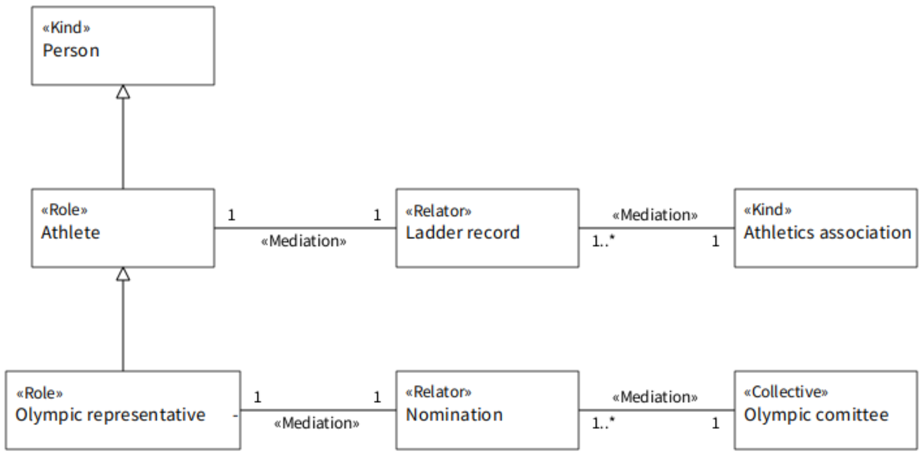


Figure 2.13: Example containing structure for role of role

Following the specification, both ”person” and ”athletics association” were stereotyped as kind, ”Olympic committee” was identified as collective (for the purposes of this model), ”ladder record” and ”nomination” were modelled as relators and finally ”athlete” and ”Olympic representative” were stereotyped as roles.

### 2.4.3 Intentional sub-role

Sometimes we need to model role that is supposed to represent special case in parent mediation in this case we use intentional sub-role pattern. In this case we create new relator that specializes relator in the parent relation and connect it to the sub-role.

Let’s suppose we are creating OntoUML model for a hospital (figure 2.14). We have analysed the domain and found out that there are three roles patient, doctor and x-ray operator. We know that person becomes patient when it had a medical checkup which has to be done by doctor, and that some checkups may be done on x-ray machine.

This has left us with model that has kind ”person” specialized by roles ”patient” and ”doctor” which is further specialized by role” x-ray operator”. There is also relator ”medical checkup” that is mediated by ”patient” and ”doctor”.

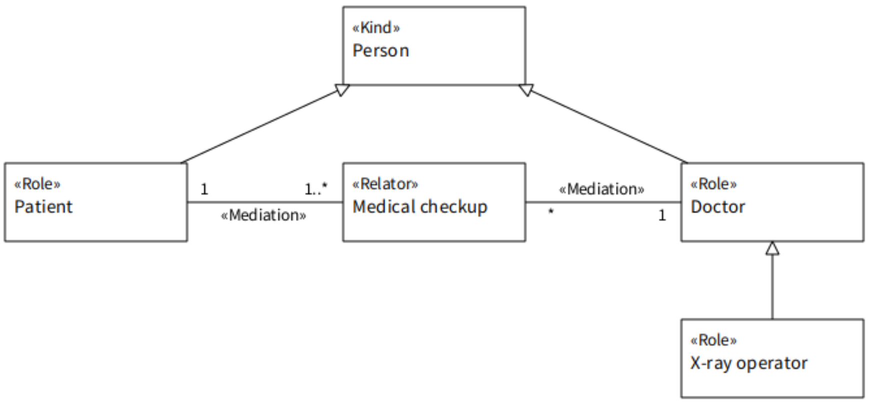


Figure 2.14: Example of intentional sub-role

Right now, we have created occurrence of free role anti-pattern as our model is still missing something. Answer lies in the relation between ”checkup” and ”x-ray operator” or rather lack of it. To resolve this, we should create relator ”x-ray checkup” which should specialize ”relator checkup” and should be connected via mediation to ”patient” and ”x-ray operator”.

Note that we still have multiple occurrences of BinOver anti-pattern, but correcting them is out of scope of this example.

### 2.4.4 Material sub-role

The material sub-role pattern is used when modeller needs to create sub-role which is defined by only a subset of relators defining parent role, but unlike in the case of intentional sub-role pattern the subset doesn’t add any additional characteristics.

Suppose that we are creating simplified model for patent approval process (figure 2.15). We start with kind ”technical solution” that is specialized by role ”patent application” that is further specialized by roles ”accepted patent” and ”rejected patent”. Relator ”application result” is connected to role ”patent application” and kind ”patent office” via mediation.

This would result in occurrence of free role anti-pattern. To avoid it we need to add two more relations in this case material associations between roles ”accepted pattern”/”rejected pattern” and kind ”patent office”.

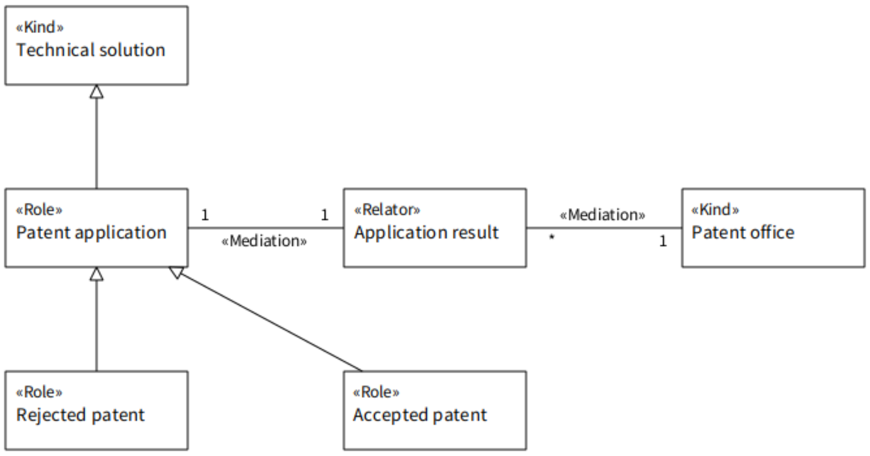


Figure 2.15: Example of material sub-role

## 2.5 Generalization set with mixed rigidity (GSRig)

As the name suggests this anti-pattern occurs when we have multiple types that follow different rigidity principles in the same generalization set. To understand why are such structures dangerous to model integrity, we must first talk about the principles behind the generalization set as concept.

### 2.5.1 Generalization set

Generalization sets are used to aggregate generalization to the same supertype and that follow a common specialization criterion. While first part of this definition is quite clear, the second part dealing with common specialization criterion might look a bit ominous, so we will explain it with an example.

Let's look at cars (figure 2.16). We can classify cars using their current price as either cheap or expensive. Alternatively, we can classify cars based on their mechanical state as either operational or damaged. Specialization criterion is used to define the reason why instance of type becomes instance of one of its subtypes, in other words why we classify concrete car as cheap or expensive using our first classification and why we classify concrete car as operational or damaged in our second classification. So, the specialization criterion would be price / mechanical state for the first/second classification respectively.

Another reason for using generalization sets lies in their meta-properties isCovering and isDisjoint. Those meta-properties allow us to define concrete types with additional subtypes and allow/forbid overlap in the subtype instantiation.

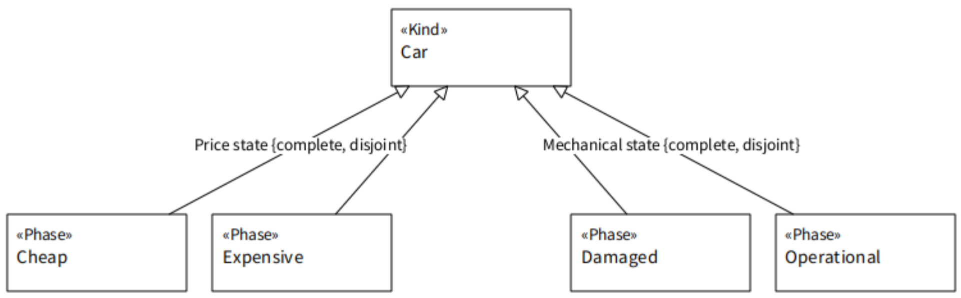

«Kind»
Car
Price state {complete, disjoint}
Mechanical state {complete, disjoint}
«Phase»
Cheap
«Phase»
Expensive
«Phase»
Damaged
«Phase»
Operational

Figure 2.16: Example of possible classifications for kind car

### 2.5.2 GSRig anti-pattern

In previous subsection we have talked about the principles behind the generalization set as concept and now we can finally look at the GSRig anti-pattern itself. Although we will focus mostly on finding, analysing and if needed correcting generalization sets with mixed rigidity, focus of this anti pattern is to identify generalization sets that aggregate generalizations using two or more specialization criteria at once.

The main reason for limiting further explanation only to the generalization sets with mixed rigidity is that the specialization criteria are closely connected to the domain semantics and the way we look at the domain. Which makes detection of this anti-pattern occurrences generally impossible without proper domain knowledge. Only exceptions to this problem are aforementioned generalization sets with mixed rigidity.

Any generalization set that contains subtypes that follow different rigidity principles is most likely aggregation of two or more generalization sets following different specialization criteria or contains one or more subtypes that have been incorrectly stereotyped. Notable exception to this ”rule” is generalization set that has mixin as supertype, since mixin by definition has to be specialized by both rigid and non-rigid types at the same type (see anti-pattern MixRig for more information).

To continue with our example about cars, let’s talk about car classification based on their mechanical states (damaged or operational) and body types. For the purposes of this explanation (figure 2.17) we will limit ourselves to sedans, hatchbacks and SUVs.

So right now, we have kind ”car”, phases ”damaged” and ”operational” and subkinds ”sedan”, ”hatchback” and ”SUV”. Let’s suppose that we made a mistake and decided to put all generalizations of the kind (all phases and subkinds) into one generalization set and made it disjoint.

Through this mistake we created occurrence of GSRig anti-pattern. Our model now allows a car to be either sedan, hatchback, SUV, damaged or oper-

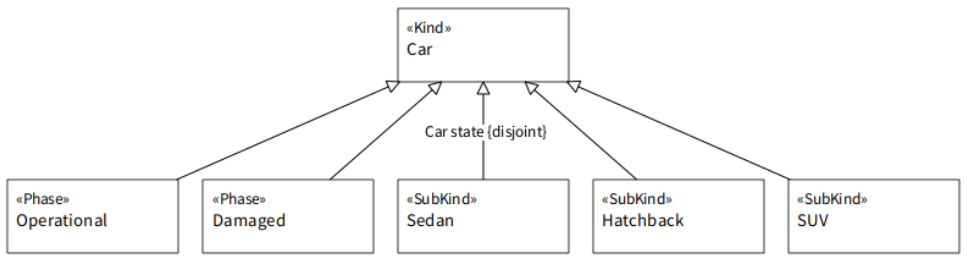


Figure 2.17: Example of generalization set with mixed rigidity

ational at one time, i.e., we cannot have car that is both sedan and operational or hatchback and damaged.

If we decided to remove operational and its generalization from the generalization set and make the set covering, we would get into even more trouble. As seen in the figure 2.18, we have now generalization set that is both *disjoint* and *covering*. This means that instance of "car" has to instantiate one of the subtypes from the generalization set and that it has to instantiate exactly one. But this means that phase "damaged" was now made rigid!

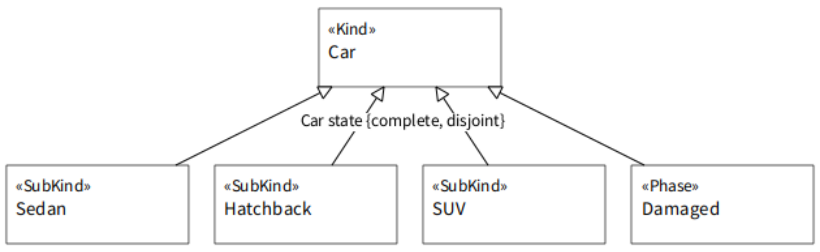


Figure 2.18: Example of rigidity forced to anti-rigid type

There are three ways to refactor GSRig occurrence. First, we should check if all subtypes are stereotyped properly. If not, we should change the stereotypes appropriately and this will either fix the problem or we will have to continue in our analysis of the generalization set.

Second option for resolving this problem is to look if the generalization couldn't be split. This would be the way to fix our example model. Phases "operational" and "damaged" would-be part of one generalization set and subkinds "sedan", "hatchback" and "SUV" would be part of second generalization set.

Third option is to create new direct parent for one or more anti-rigid subtypes. This might have to be followed with creating OCL restriction if the previous step resulted in creation of only one rigid subtype.

## 2.6 Heterogeneous collective (HetColl)

Anti-pattern HetColl is closely connected to the HomoFunc (Homogeneous functional complex) anti-pattern, as both of them are result of misunderstanding of differentiation between stereotype collective and functional complex.

### 2.6.1 Collective

Collective represents rigid concepts that provide identity principle for their instances. Its main characteristic is that all its parts are perceived in the same way from the collective's point of view, i.e., its internal structure is homogeneous.

Examples of collective are: deck of cards, forest, band.

### 2.6.2 Functional complex

Functional complex represents entity with multiple parts that play different roles as viewed by the complex itself, i.e., its internal structure is heterogenous. Similarly to collective, functional complex provides identity principle for its instances.

Examples of functional complex are: car (consisting from engine, chassis...), IT company staff (consisting of managers, programmers, support...), opera staff (consisting of singers, musicians, technical staff...).

### 2.6.3 HetColl anti-pattern

As was previously mentioned, HetColl anti-pattern occurs when the modeller confuses the difference between collective and functional complex and uses the former instead of the latter. This doesn't have to be a result of misunderstanding or lack of knowledge of the principles, but it can also appear as result of change of perception of the modelled entity.

It is important to know that the same entity can be correctly modelled as a collective in one model and in second model it will be correctly identified as functional complex. Example for this can be entity ”deck of cards” especially if we take a deck from card game such as Magic the Gathering.

From the manufacturers point of view ”deck of cards” will be modelled as collective, since all cards have the same dimensions and are printed using the same process.

From the players point of view ”deck of cards” will be functional complex since it contains cards representing creatures, lands, spells... and each of those categories has its own distinct set of properties and rules.

HetColl anti-pattern would occur if we modelled the deck from the point of view of the player and decided to stereotype it as collective. Such model can be seen in figure 2.19.

There are three conditions that must be met for HetColl to occur:

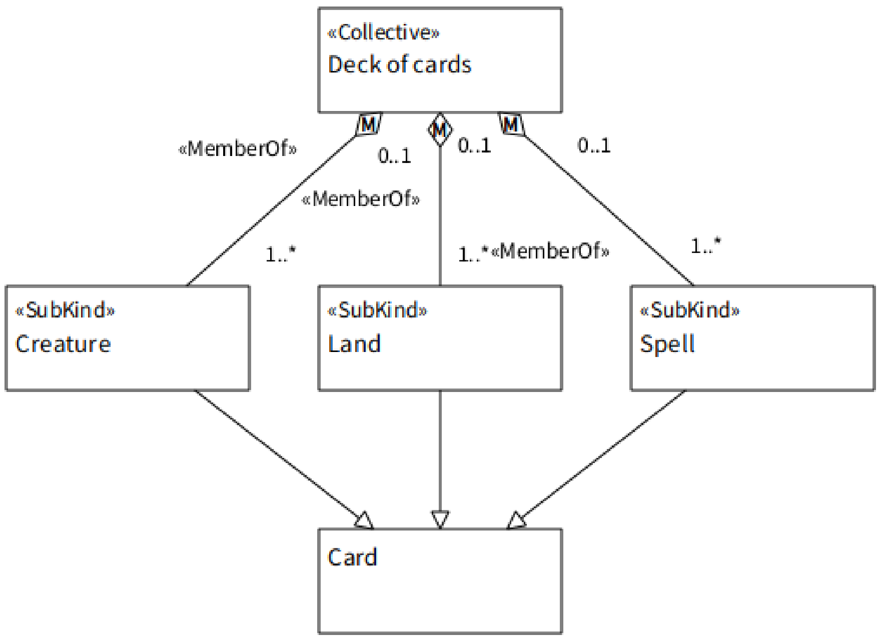


Figure 2.19: Example of heterogenous collective

1. Entity representing the whole has to be stereotyped as collection.
2. All parts must be instantiated directly by the whole.
3. All parts must be connected by memberOf association directly to the whole or one of its ancestors.

If all three conditions above are evaluated as true, modeller should check the collection and all its parts as it is probable, that the entity stereotyped as collective should be functional complex instead.

## 2.7 Homogeneous Functional Complex (HomoFunc)

HomoFunc anti-pattern occurs when modeller misidentifies collective and instead models it as functional complex. This differentiates HomoFunc anti-pattern from HetColl anti-pattern that deals with the same problem of misunderstanding differences between collective and functional complex, but focuses on analysis of the collective.

Due to this similarity, we have already covered differences between collective and functional complex in chapter about HetColl anti-pattern and this chapter will cover only parts that are specific to the HomoFunc anti-pattern.

### 2.7.1 HomoFunc anti-pattern

If we simplify the problem to its most basic form, we can say that HomoFunc anti-pattern occurs when there is functional complex that consist of multiple instances of single part, thus creating functional complex with homogeneous structure, which should be instantiated as collective instead.

You can see this in figure 2.20 in which we model entity ”deck of cards” from the point of view of the manufacturer, but the modeller didn't take this in mind and stereotyped the entity as a kind.

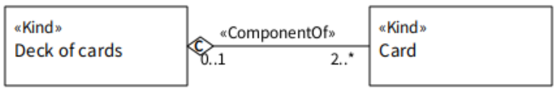


Figure 2.20: Example of homogenous functional complex

There are four conditions that are mandatory for HomoFunc to occur:

1. Only functional complexes may instantiate the Whole.
2. Only functional complexes may instantiate the Part.
3. Whole has only single direct or indirect Part.
4. Multiplicity of the partOf association has to be greater or equal to 2.

Only after all four conditions are true, HomoFunc anti-pattern occurs. In that case model should be analysed again and modeller should decide if the functional complex should be instead stereotyped as collective, or if additional parts should be added.

As we discussed in the chapter dealing with HetColl anti-pattern, we must always keep in mind the view with which we look at the domain, because it will determine we should model analysed entity either as functional complex or as collective. We have to remember this especially when the view changes, parts may be added or removed and this can lead to occurrence of HetColl or HomoFunc anti-patterns.

## 2.8 Imprecise abstraction (ImpAbs)

Anti-pattern ImpAbs is used to detect associations that connect generalization structures that are considered too permissive. This sort of connections usually means that the model allows for creation of unforeseen and unwanted instances.

ImpAbs anti-pattern requires occurs only when either of two following conditions hold:

- multiplicity on the source end of the association is greater or equal to 2 and source has at least 2 direct subtypes,
- multiplicity on the target end of the association is greater or equal to 2 and target has at least 2 direct subtypes.

As was said previously, making models too permissive will usually lead to unexpected consequences. Let's suppose we are creating a simple ontological model for a house. Each house consists of roof and multiple walls. For this example (figure 2.21), we assume that there are two kinds of a wall: bearing wall and partition wall, which are defined as follows: bearing walls hold full weight of the house, and therefore (for the purpose of this example) cannot have any windows or doors inside them, while partition walls can.

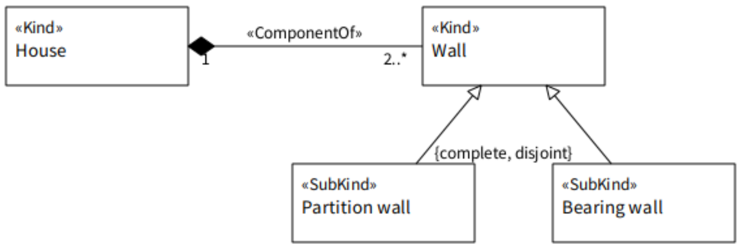


Figure 2.21: Example of imprecise abstraction

Full model can be seen bellow and on the first look everything looks fine. Problems start to appear when we analyze it. Due to our imprecise abstraction, we are permitted to build house that consist only from partition walls which would lead to instability and even to the collapse of the house. We are also permitted to build house that has only bearing walls, which would make it stable, but also impossible to get to, since bearing walls cannot have doors inside them.

There are three solutions for our problem (and to ImpAbs anti-pattern occurrences in general):

1. Set cardinality constrains using OCL invariant specification.
2. Set cardinality constrains by creating new associations between the set of subtypes on one end and the other end of original association. (In our case create associations between "bearing wall" and "partition wall" on one side and "house" on the other side).
3. Create association similarly to option 2, but also specify association meta-property values. For this we can use meta-properties isEssential, isInseparable, isImutableWhole, isImmutablePart, isShareable or isReadOnly.

Also, as Sales notes in his work [17], ImpAbs anti-pattern is most likely to show in cases when the association characterizing the occurrence is a part-whole relation (with the exception of memberOf association). modellers therefore should pay increased attention to those types of association, because as mentioned previously consequences for making a mistake can be quite surprising.

## 2.9 Mixin with the same identity (MixIden)

MixIden anti-pattern is part of ”classification anti-patterns” together with others such as HetColl and HomoFunc. All of those anti-patterns usually occur due to modellers lack of understanding of the OntoUML principles, or due to lack of attention to the view used to describe the model domain.

Principle that is being confused in case of MixIden anti-pattern is the difference between sortals and non-sortals. Instances of sortals (individuals) follow the same identity principle while non-sortals don’t [6].

You can see this in the model of the customer. Kind company is a sortal and thus all its subtypes follow the same identity principle in this case its company number. RoleMixin customer is a non-sortal since all of its subtypes don’t follow the same identity principle, in case of the ”company customer” it’s the aforementioned company number and in case of the ”individual customer” it’s the personal identification number.

MixIden anti-pattern occurs when modeller confuses the definition of non-sortals and models a structure in which all sortal children of non-sortal follow the same identity principle. This means that all sortal children get their identity directly or indirectly from the same ancestor, which can also be child of the same non-sortal. Note that although full name of MixIden is mixin with the same identity, its scope covers all non-sortals, not just mixin.

In figure 2.22 we can see OntoUML model for acquiring supplies by a shop. First there is a kind ”Company” representing shops business partners. Company can be either ”Manufacturer” or ”Distributor”, but both ”Manufacturer” and ”Distributor” can become ”Supplier”, by establishing ”Supply contract” with one of the shops ”Managers”. Note that for clarity of this example, we have omitted fulfilment of relational dependency for role ”Manager” in the presented model.

As we can see the modeller decided classify ”Supplier” as a rolemixin, but this results in occurrence of MixIden anti-pattern, since both ”Manufacturer” and ”Distributor” get their identity from the same source.

Correcting this mistake is bit more complicated. Any occurrence of MixIden can be fixed by changing the non-sortal that causes the problem to sortal appropriate to the rest of the model, but (like in this case) we might need to make additional changes to the model.

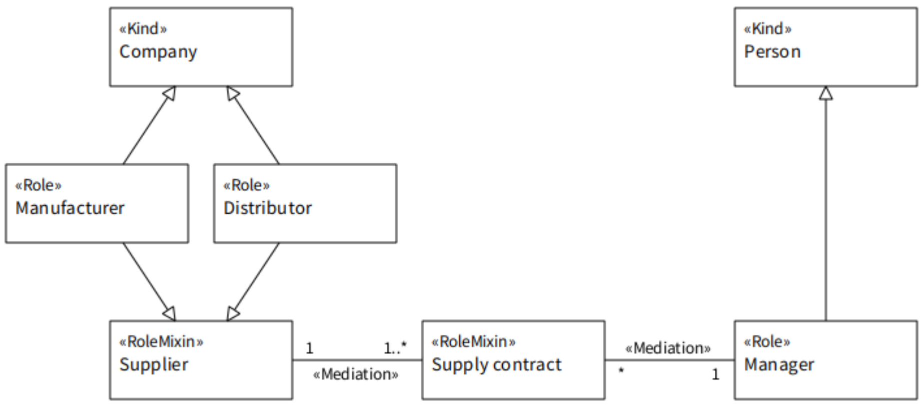


Figure 2.22: Example of mixin with the same identity

First we need to change stereotype of "Supplier" from rolemixin to role. Then we need to remove generalizations between "Supplier", "Manufacturer" and "Distributor". Since "Supplier" is now stereotyped as role it also requires identity, so we need to create generalization from "Supplier" to general "Company". Lastly we need to create OCL invariant to enforce that each "Supplier" has to also be "Manufacturer" or "Distributor".

## 2.10 Mixin with the same rigidity (MixRig)

As the name suggests MixRig anti-pattern occurrence is characterized by mixin with multiple children, that are all either rigid or non-rigid. Although this might look like other anti-patterns that cover misclassification analysed of entities (such as HetColl, HomoFunc, MixIden...) in case of MixRig it might be actually one of the mixin children that has been misclassified. But first we have to talk about mixin and the principle of rigidity.

### 2.10.1 Rigidity principle reminder

Rigidity is ontological meta-property, that defines mutability of the type. In other words, it defines if instances of particular type are always connected to the type or if their types can change.

There are four types of rigidity, but for purposes of MixRig anti-pattern and this reminder we will talk only about three:

1. **Rigid** – Individuals instantiate rigid in all possible scenarios (worlds). Rigid types define essential characteristics for their instances.

2. **Non-Rigid** – Non-rigidity is logical negation of rigidity. Type is non-rigid if at least one instance to which this type applies to in one world, is not applied by this type in another world.

3. **Semi-rigid** – Type is semi-rigid, if it is rigid for some instances and non-rigid to others. Stereotype mixin is semi-rigid.

### 2.10.2 MixRig anti-pattern

Mixin is semi-rigid type and as such it should be specialized by both rigid and non-rigid types. Analysis of mixin specialized only by rigid or non-rigid types leads to one of three possibilities:

1. **modeller has misclassified the type as mixin** and therefore the correction lies in finding proper classification of the type. Typical example for this is mixin specialized only by roles as seen in figure 2.23. In this case model can be fixed simply by changing mixin to rolemixin.

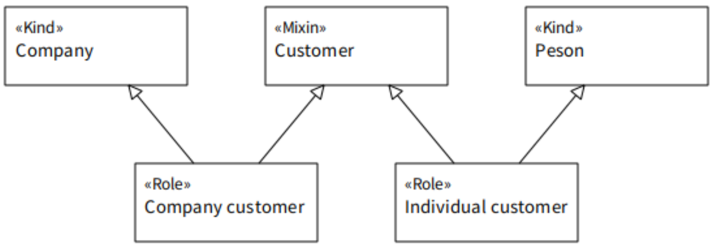


Figure 2.23: Example of mixin specialized only by roles

2. **modeller has misclassified one of the children.** Example for this can be seen in model [17]. Let's suppose that we are creating model (figure 2.24) for a transport company that needs to differentiate between legal and illegal goods. We can say that drugs are always considered illegal, but car is considered illegal only if it has been stolen. As we can see modeller made a mistake and specialized "Stolen" and "Legally owned" as subkinds instead of phases which leads to occurrence of MixRig.

3. **Model is incomplete.** This can typically happen during initial phase of the model creation when we are still identifying entities and associations between them. Nevertheless, this is as severe if not more than misclassification of either mixin or its children, as it means that we have omitted some part of the domain in our model.

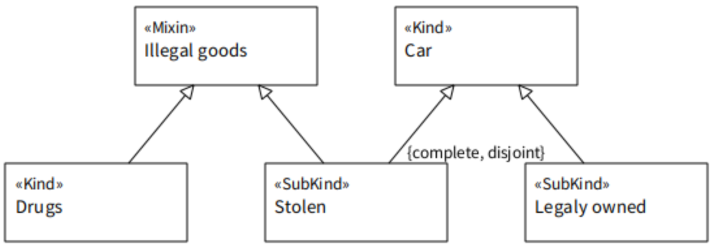


Figure 2.24: Example of mixin specialized only by rigid types

## 2.11 Multiple relational dependency (MultDep)

In OntoUML relational dependency is closely tied to classes stereotyped as role or roleMixin, but it is not exclusive to them. Others such as kind, collective, quantity, subkind, role, phase, category, mixin, roleMixin and phaseMixin can be also connected by mediation (characterizes relational dependency).

Even though there is no upper limit to how many relational dependencies (mediations) can be applied to single class, having more than one relational dependency can mean that there is a hidden complexity and even additional dependencies between the relational dependencies.

MultDep anti-pattern aims to identify classes with multiple relational dependencies and warn the modeller that he should carefully analyse both the class and all its dependencies.

Analysis itself should start with investigating the dependencies, notably if they are all mandatory or if some of them are optional. For each optional dependency modeller should create new role as a child of the original class and make the dependency mandatory for the new role.

After modeller creates all required roles, he shall look on the dependencies again and check if some of them require particular order in which they are established. Ordered dependencies will force modeller to create hierarchy of roles. For example, during each operation there is exactly one lead operator and he can have multiple doctors as assistants, but you have to first become a doctor, before you can be lead operator. This would be represented in the model by entity person, that is specialized by role doctor, that is further specialized by role lead operator as seen in the figure 2.25.

On the other hand, some dependencies don't require any particular ordering or cannot be ordered at all. Example for this can be roles singer and actor, as there is no requirement that you have to become a singer before you become an actor or vice versa.

Third step of the analysis is focused on search for "dependencies between dependencies". This happens if relator that formalizes the dependency is in relation with relator that formalizes different dependency.

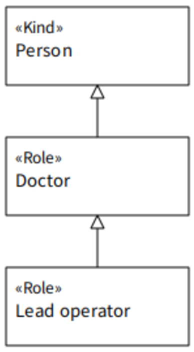


Figure 2.25: Example of ordered role dependencies

To exemplify this, we will refer to example provided as part of MultDep definition (see figure 2.26):

*A person becomes an undergraduate student when she enrolls in a major course at a university, e.g. "Computer Science" or "Philosophy". A unique number identifies each enrollment. Victor, a very curious and dedicated young man, decides to pursue, simultaneously, a major in "Philosophy" and "Computer Science". To do that, he would need to enroll two times at the university. After his enrollments, Victor wants to apply for "Logics 101" as a "Computer Science" student and apply for "Sociology 101" as a "Philosophy" major. To do that, each course application must not only identify "Victor" as the applying student, but also identify the particular enrollment he is using to apply.* [] ref tiago

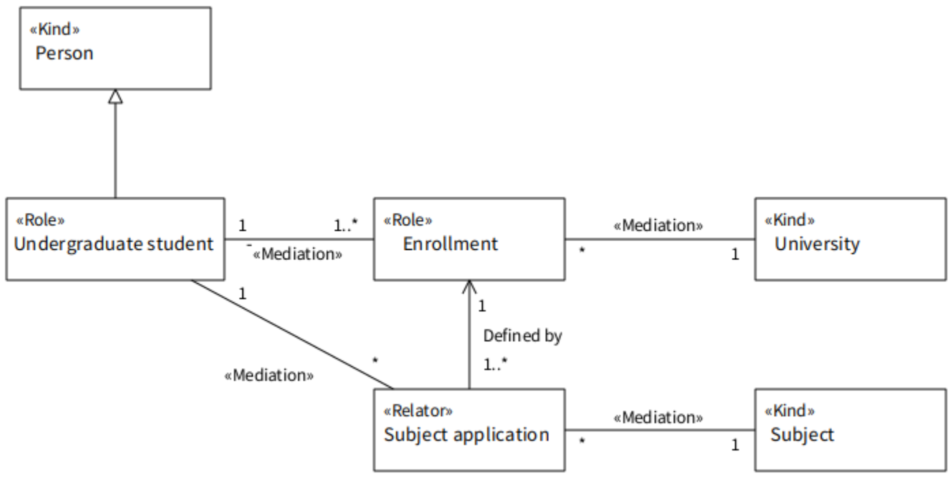


Figure 2.26: Example of "dependencies between dependencies"

Finally let's look at example of MultDep occurrence in model for a car rental company. Rental company has many cars. Each of them should have valid MOT to be considered operational. We also know that car is considered to be rented, when another company signs rental contract.

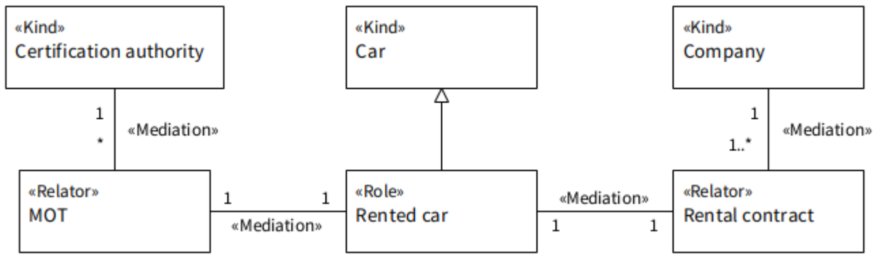


Figure 2.27: Example of multiple relational dependency

As we can see in figure 2.27, modeller decided to model role ”Rented car” mediated by both ”MOT” and ”Rental contract”, thus creating MultDep occurrence. This should be corrected by creating new phases ”Operational” and ”Damaged”. Mediation to ”MOT” should be removed from ”Rented car” and moved instead to phase ”Operational”.

## 2.12 Part composing overlapping wholes (PartOver)

Part over anti-pattern follows similar logic to BinOver anti-pattern, actually there are cases in which both anti-patterns overlap. Because of the similarity between the anti-patterns, reader is highly advised to look at BinOver anti-pattern first, as it contains definitions and explanation for concepts of overlapping and disjoint sets.

There are two conditions that have to be fulfilled in order to PartOver to occur. Both of them are rather obvious and can be summarized in a single sentence: part has to compose multiple wholes that have to be part of single overlapping set.

First condition states that the wholes have to overlap, i.e., it has to be possible for a single instance to instantiate all types in the set at the same time, while the second condition says that the sum of upper cardinalities of all part-whole associations has to be greater or equal to two, i.e., it has to be possible for a single instance of part to compose at least two wholes at once.

To both explain why is this structure potentially dangerous and to provide an example, we will create model for a deck of cards. In the figure 2.28 we can see collective ”deck of cards” (note that we look at the ”deck of cards” from the point of view of the manufacturer so all ”cards” have equal roles, otherwise we would get occurrence of HomoFunc anti-pattern).

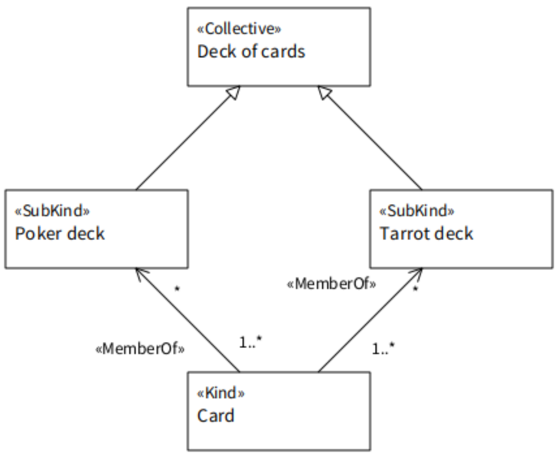


Figure 2.28: Example of part composing overlapping wholes

Deck of cards is specialized by subkinds ”poker deck” and ”tarot deck”. Last type in the model is kind ”card” which has two memberOf associations one to whole ”poker deck” and other to the ”tarot deck”. Upper bound for whole end of both associations is unlimited.

This model represents the most basic occurrence of PartOver anti-pattern. Main reason behind identifying such structures it their permissiveness, in other words, such structures may allow unexpected and often invalid (in terms of the domain) instances of whole.

PartOver anti-pattern is usually result of modellers mistake/oversight during creation of one of following intended interpretations:

- We want the whole to be **disjoint**, i.e., no individual can instantiate multiple whole types. This corresponds to our example. No card can be part of the poker deck and tarot deck at the same time. Solution for this is to simply make the whole types disjoint.

- We want the wholes to be **exclusive**, individual can instantiate multiple whole types at once, but each instance of the part can be connected only to one whole type at once. In our example this interpretation would be represented by ”mashup” deck (created by shuffling poker and tarot decks together). In this case we won’t need to make any changes to the generalization set, but we should create an OCL invariant to keep our previous logic that single card cannot be both poker and tarot card at once.

- We want the whole to be **partially exclusive**, i.e., individuals can instantiate some whole types simultaneously while others should be disjoint. Expanding on our previous example with ”mashup” deck, we will now add collective bridge deck specializing deck of card and as whole end for another member of association to the kind card. Now we have situation when card can be part of both poker and bridge deck, but not tarot deck. Again, the solution would be to create OCL invariant to define the exclusive subsets.

## 2.13 Relation composition (RelComp)

Significant part of anti-patterns discussed in this work was created to identify overly permissive structures to warn modeller about unforeseen consequences and to provide help with deciding if and how to refactor/correct them.

RelComp anti-pattern is another representative of this category. It is defined by two distinct associations: association $A$ that connects type *sourceA* to type *targetA* and association $B$ that connects type *sourceB* to type *targetB*.

There are two additional conditions that need to be fulfilled. Note that for this explanation we suppose that association A is the on that connects "ancestor types" (second condition contains additional explanation).

First condition defines minimal cardinality of association $A$ on *targetA* end. Concretely lower bound has to be greater than zero and upper bound has to be greater than one. Second condition is bit more complicated. It is fulfilled when one or both of following sub-conditions are fulfilled:

- *SourceA* is equal to or ancestor of both *SourceB* and *TargetB*. In predicate logic:

$$(SourceA = SourceB \vee ancestorOf(SourceA, SourceB)) \wedge \\ (SourceA = TargetB \vee ancestorOf(SourceA, TargetB))$$

- *TargetA* is equal to or ancestor of both *SourceB* and *TargetB*. In predicate logic:

$$(TargetA = SourceB \vee ancestorOf(TargetA, SourceB)) \wedge \\ (TargetA = TargetB \vee ancestorOf(TargetA, TargetB))$$

Fulfilling both primary conditions described above would lead to occurrence of RelComp anti-pattern. This does not mean that the model contains mistake or that its incorrect, it only urges the creator of the model to analyse the structure and think about the intended form of composition.

OntoUML defines five types of composition based on the way in which the instantiation of association $B$ depends on instantiation of association $A$:

- ***Definition (Existential Composition)****: for every distinct individuals x, y, if x is related to y through B, it implies that x and y are related to at least one common individual through relation A.* [17]

- ***Definition (Right Universal Composition)**: for every distinct individuals x, y, if x is related to y through B, it implies that all z that is connected to x, through A, is also connected to y, through A.* [17]
- ***Definition (Left Universal Composition)**: for every distinct individuals x, y, if x is related to y through B, it implies that all z that is connected to y, through A, is also connected to z, through A.* [17]
- ***Definition (Forbidden Composition)**: for every two distinct individuals x, y, if x is related to y through B, it implies that x and y are connected to no individual in common through A.* [17]
- ***Definition (Custom Existential Composition)**: for every distinct individuals x, y, if x is related to y through B, it implies that x and y are connected to [less than / more than / exactly] n common individuals through A.* [17]

Refactoring of the RelComp anti-pattern consists of identification of the desired type of composition and creation of OCL invariant that enforces it.

Figure 2.29 represents OntoUML model of graph and it's also example of RelComp occurrence. Each graph consists of at least one "graph element". There are two kinds of "graph elements": vertexes and edges. Edges connect either one or two vertexes and this is represented by formal association "connects".

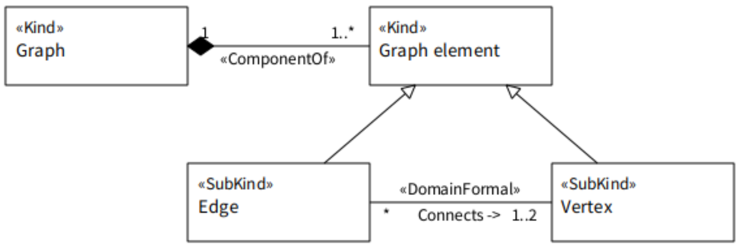


Figure 2.29: Example of relation composition

Such model can be used to describe simple graphs and can form a basis for further description of graph theory in OntoUML. Problem is that the model also contains a hidden catch, it allows single edge to connect two different graphs.

This might be an issue (it depends on our definition of graph) and if we decide that it's indeed an unwanted consequence we should identify our desired composition type and create appropriate OCL invariant to enforce it.

## 2.14 Relator mediating overlapping types (RelOver)

Logic behind RelOver anti-pattern is almost identical to PartOver anti-pattern and there is also overlap with BinOver anti-pattern. Since most of the definitions and explanations have been provided in sections dealing with aforementioned anti-patterns, we are not going to repeat them here and instead focus on explaining the differences and providing an example.

Main difference between PartOver and RelOver anti-patterns is directly connected to their names. PartOver focuses on overlapping wholes, while RelOver focuses on identifying relators that are mediated by overlapping types.

RelOver anti-pattern has two conditions that have to be fulfilled for it to occur. First one is similar to PartOver and it states that the types mediating common relator have to be part of overlapping set, i.e., it has to be possible for single individual to instantiate all mediated types at same time.

Second condition was created to limit the number of "false alarms" of RelOver anti-pattern. The total sum of upper bounds of all mediations between common relator and the set of overlapping types on the relator side has to be greater than two.

This is essential because OntoUML definition of relator states that each relator has to mediate at least two distinct individuals and without the second condition above we would need to analyse many occurrences of RelOver, just to find that significant part of them were "false alarms".

Similarly to PartOver anti-pattern RelOver usually occurs when modeller wants the mediated types to be disjoint, exclusive or partially exclusive. For more explanation about the intended interpretations, please read the PartOver section.

Now let's look at a typical example of RelOver anti-pattern as can be seen in figure 2.30. Here we have simple model for crime investigation namely interrogation of suspects. Suppose that have been told that each interrogation is done by one suspect and one investigator and results in interrogation report.

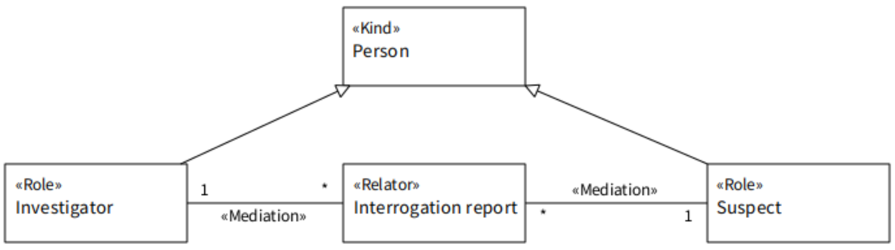


Figure 2.30: Example of relator mediating overlapping types

We have correctly determined that the interrogation report should be stereotyped as relator, while "suspect" and "investigator" should be roles mediated by the interrogation report. Since role is anti-rigid sortal and as such it requires an identity, we have also created kind ”person” and specialized it by both roles.

This wouldn’t be enough to cause RelOver occurrence, but since we know that single investigator may (and most probably will) be part of multiple interrogations as might be the suspect we need to set the upper cardinalities of the mediations at the relator end to * (unlimited).

Problem is that we have now created model that permits single person to not only be both investigator and suspect, but also create interrogation reports with itself. Luckily the solution is to just create new generalization set, add both generalizations to it and set it’s meta property *isDisjoint* to true.

## 2.15 Relator mediating rigid types (RelRig)

Mediations are usually used to connect relator on one side with role/roleMixin (anti-rigid types) on the other side. Although this is the most common model structure OntoUML also allows other types to be part of mediation.

RelRig anti-pattern is defined by mediation connecting relator with rigid type (category, collective, kind, quantity or subkind). This structure is syntactically valid, but modeler should pay increased attention to it, as it may indicate that improper stereotype was selected for the rigid type or event that the model is missing some parts.

Analysis of RelRig occurrence should therefore start with verifying rigidity of the mediated type, continue with checking if the mediation is really mandatory and end with determining the direction of the relational dependency.

When verifying the mediated type (further referred also as <Type>) rigidity, we have to determine if the type is rigid by asking ourselves if every individual that at some point instantiates <Type> had to be created as one and cannot cease to be instance of the <Type> during the individual’s existence. Or we ask ourselves following question to determine anti-rigidity: ”Can an individual that was not created as <Type> to become one, or an individual that is already an instance of <Type> cease to be it and still exists?”. [17]

If we discover that the mediated type is anti-rigid, we have to change its stereotype based on it’s sortal/non-sortal property to either role/roleMixin respectively and we may end in our analysis (we may also need to create new identity provider for the role).

Next step is to check if the relational dependency is mandatory for the mediated type. If we determine this is not the case, we should create new role/roleMixin and add/change specializations accordingly, otherwise we need to continue to the third step of our analysis.

Third and last step of our analysis is aimed at determination of the dependency direction. Since we have already determined that the mediation relation is mandatory, we have only three possible cases of the dependency direction:

1. **Dependency is bidirectional**: (both relator and mediated type are dependent on each other) modeler has to set isReadOnly meta-property on the mediated type end to true.
2. **Dependency from relator to mediated type**: in this case RelRig occurrence is a false alarm.
3. **Dependency from mediated type to relator**: both mediation association and the mediated type have been incorrectly stereotyped. Modeller should change the mediation to characterization and the stereotype of the mediated type to mode.

Figure 2.31 contains simplified model for ordering food from restaurant. We have kind ”person” that is specialized by role ”customer”. There is also kind ”restaurant” and relator ”food order” that is connected through mediations to both ”customer” and ”restaurant”.

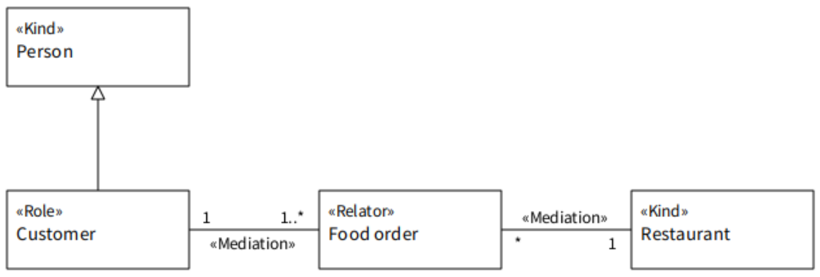


Figure 2.31: Example of relator mediating rigid types

RelRig occurrence is connection between kind ”restaurant” and relator ”food order”. Therefore, we have to start our analysis at the first step, by verifying rigidity of the ”restaurant”. In this case we quickly realize that ”restaurant” is not rigid – restaurant may branch out and include shop with local grocery or it might change into catering company.

Kind ”restaurant” should be stereotyped as role instead. Since role doesn’t have provide identity, we also need to create kind ”company” and add generalization between it and role ”restaurant” as its subtype.

## 2.16 Relation specialization (RelSpec)

Structures that characterize RelSpec anti-pattern are similar to structures that characterize RelComp anti-pattern, indeed there are several special cases that are both occurrences of RelSpec and RelComp.

Basic structure of RelSpec anti-pattern is characterized by two distinct relations: association A that connects types *SourceA* and *TargetA*, and association *B* that connects types *SourceB* and *TargetB*.

There are two additional conditions that need to be fulfilled before RelSpecs occurs. Note that for the purposes of this explanation, we assume that association a contains ”ancestor types”, i.e., that *SourceB* is descendant/equal to either *SourceA* or *TargetA* and the same has to be true for *TargetB*.

First condition is fairly simple as it states that associations A and B have to be distinct, but the second condition has almost identical structure to second condition of RelComp anti-pattern and as such it consist of two sub-conditions. At least one of the following sub-conditions has to be evaluated as true to fulfil the second condition.

- *SourceA* is equal to or ancestor of *SourceB* and *TargetA* is equal to or ancestor of *TargetB*. In predicate logic:

$$(SourceA = SourceB \lor ancestorOf(SourceA, SourceB)) \land \\ (TargetA = TargetB \lor ancestorOf(TargetA, TargetB))$$

- *TargetA* is equal to or ancestor of *SourceB* and *SourceA* is equal to or ancestor of *TargetB*. In predicate logic:

$$(TargetA = SourceB \lor ancestorOf(TargetA, SourceB)) \land \\ (SourceA = TargetB \lor ancestorOf(SourceA, TargetB))$$

Correcting occurrence of RelSpec anti-pattern depends on type of restriction for the instantiations of associations $A$ and $B$. There are four types of restrictions: subsetting, redefinition, association disjointness and specialization.

**Subsetting** happens when being related through association $B$ implies being related to association $A$. Examples of subsetting are associations ”sibling of” and ”sister of”. Every sisters is also a sibling but not all siblings are sisters.

**Redefining** happens when following condition holds for each instance of *SourceB*: ”individuals connected to instance of *SourceB* through association $B$ are the same individuals that are connected through association $A$”. In other words, instance of *SourceB* is connected to other individual either through both $A$ and $B$, or not at all.

Continuing with our previous example, let’s assume that the *SourceA* is kind ”person”, *SourceB* is subkind ”girl”, association $A$ is ”child of” and association $B$ is ”daughter of”. Association ”daughter of” redefines association ”child of”, because both ”daughter of” and ”child of” always define the same set of parents. By creating association ”daughter of” we have increased complexity of the model but we haven’t provided any new knowledge. Model for this example can be seen in figure 2.32.

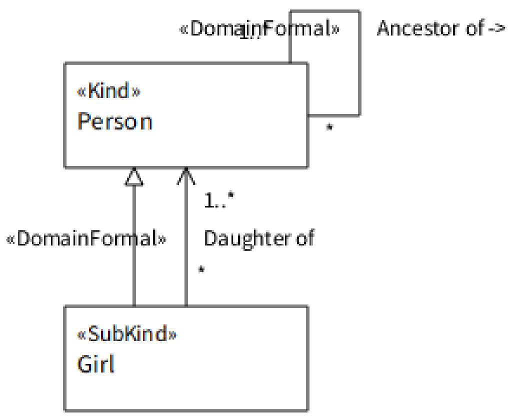


Figure 2.32: Example of relation specialization - redefining

**Disjointness** happens when being related through association *B* implies not being related through association *A*. Let's use math to exemplify, by looking at relations greater than and lesser than. We can say that number X is greater than Y, if and only if, X is also not lesser or equal to X.

**Specialization** is similar to subsetting, but it also represents intentional relation between associations *A* and *B*, by forcing inheritance of all properties from association *A* to association *B*, i.e. marking that association *B* is subtype of *A*.

Refactoring of first three restriction types (subsetting, redefining and disjointness) is done by creation of OCL invariant and enforcing the desired partition type, while refactoring to enforce specialization is even easier, since we only have to create generalization from subtype *B* to supertype A.

## 2.17 Repeatable relator instances (RepRel)

Purpose of RepRel anti-pattern is to help modeller with applying internal uniqueness constraint. This concept comes from Object Role Modelling (ORM) approach. In ORM uniqueness constraint is applied to limit duplication of identical combinations for a single predicate.

OntoUML unlike ORM is not fact-oriented language and instead of using predicates to define roles for entities, it uses relators and mediations. This might be an issue in some models, since OntoUML normally doesn't set any limits to duplicate instances of relator, that mediate the same individuals. Uniqueness constraint has to be therefore enforced by creation of OCL invariant for either current relator type or historical relator type.

Historical relator is little bit closer to the ORM view of the world, because it represents "database table" that contains single record for each relator existence. Note that existence is not the same as instantiation.

To exemplify, employee has work contract with his employer. Instance of work contract is always time limited, it has start and end, even though the end date might not be defined during the contract creation. Instance of historical relator ”work contract” would represent the contract form its start till its end, but it wouldn’t cease to exist when the contract expires. This way we can represent the history of all instances of ”work contract” between the same employer and employee.

Concurrent relator on the other hand represents truth maker of existing relation. The moment the relation ceases to exist so does the appropriate relator instance. Continuing with our previous example, representing ”work contract” by concurrent relator would give us information about all work contracts between employer and its employees that are currently active, but it wouldn’t allow us to represent their history.

As was mentioned previously uniqueness constraint is enforced via OCL invariant for both relator types. Historical relator might also require creation of new phases to represent states in which the relator instance was ”active” or ”expired”.

We have already presented why was RepRel anti-pattern defined, we have discussed and provided examples for both concurrent and historical relators, but we have yet to properly define the RepRel anti-pattern itself.

Structure of RepRel occurrence is characterized by relator mediated by at least two other types. There are also two additional conditions that need to be fulfilled:

1. Each mediation that directly or indirectly (through ancestor) connects relator to other type has to have its upper bound cardinality on relator end greater than one.

2. There has to be at least one mediation connected directly to the relator, that characterizes the RepRel occurrence.

Figure 2.33 contains example of RepRel occurrence. We have simple model for marriage. There is kind ”person” specialized by disjoint roles ”husband” and ”wife”. Relator ”marriage” is connected to both roles via mediations. We have also set upper bound cardinality on relator end in both mediations to unlimited.

Now we have model that permits any person to be in marriage multiple times at the same time. Not only that it is also possible for two persons to be in marriage multiple times with each other. This is definitely an error on the modelers side and needs to be fixed.

This particular example can be corrected in three ways based on the intended outcome:

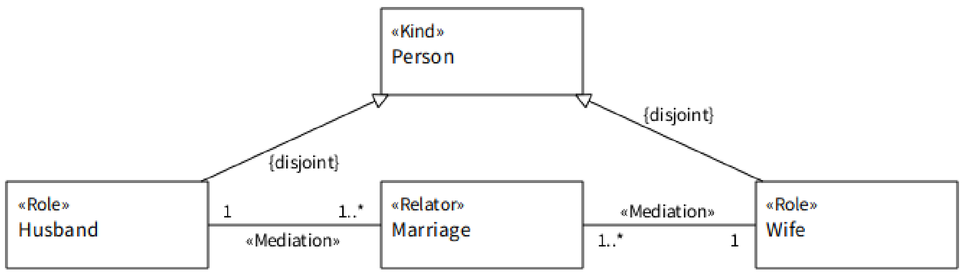


Figure 2.33: Example of repeatable relator instances

1. Setting upper bound cardinality of the mediations on the relator end to one, would result concurrent relator, that allows each person to be in at most one marriage at the same time.

2. Creating an OCL invariant that limits duplicate instances of marriage between the same two people, would be the right decision if we wanted ”marriage” to be a concurrent relator and model polygamy.

3. Adding phases ”active” and ”ended”, and creating an appropriate OCL invariant (that can either permit or deny polygamy), would result in model with historical relator, model which acts as ”register” of marriages.

## 2.18 Undefined formal association (UndefFormal)

Before we can define UndefFormal anti-pattern, we need to first talk about difference between association stereotype formal and OntoUML concept of formal relations.

OntoUML defines two kinds of associations material and formal. Formal relations simply connect two individuals, while material relations are defined by their truth-maker, entity that is created as part of the material relation and that acts as seal between the two connected individuals.

This means that relations stereotyped as characterisation, componentOf, derivation, memberOf and many others are also considered formal relations. Of course the next logical question is why have stereotype formal?

Full name of the formal (sometimes called domainFormal) is Domain Comparative Formal Relation (from now on referred as DCFR). Such relations are characterised by qualities of the connected types. Examples of this are qualities ”age” in relation ”olderThan”, ”weight” in relation ”heavierThan” and ”size” in relation ”smallerThan”.

Creation of DCFR association between two individuals that don't have such quality leads to UndefFormal occurrence. This is caused by either of two possible causes: our ontology is missing qualities used for deriving of the relation or we have stereotyped the relation incorrectly as DCFR.

Refactoring of such structure is dependant on the reason for UndefFormal occurrence. If our relation is indeed DCFR, we have to either add qualities to the connected individual set the relation as derived, and create OCL invariant with definition of the derivation rule.

If we decide that our relations isn't DCFR after all, we need to change the stereotype to the correct one - this heavily depends on context of the UndefFormal occurrence, and in some cases it may also lead into creation of additional entities.

Lastly we may conclude that our relation is formal, but there is no OntoUML stereotype that suits it. In this case we don't need to refactor anything, but this decision has to be backed by a proper analysis. Modellers often use formal stereotype in most cases when it's not clear for them which stereotype should be used, since they know that the model will be syntactically valid.

In figure 2.34 we have example of UndefFormal occurrence. Modeller wanted to create simple model for keg of gunpowder. Both "Keg" and "Gunpowder" were stereotyped as kinds, and modeller decided to connect them with (domain) formal relation. This is a typical example of UndefFormal occurrence and in this case, it indicates two problems.

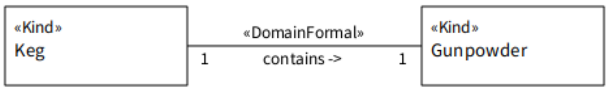


Figure 2.34: Example of undefined formal association

First we should change stereotype of the formal relation from domain formal to containment. This will lead us to the second problem – "Gunpowder" was incorrectly stereotyped. Instead of kind it should be stereotyped as quantity. After both changes we will end with valid model with no anti-pattern occurrences.

## 2.19 Undefined phase partition (UndefPhase)

OntoUML defines phase as anti-rigid subtypes instantiated by changes in intrinsic properties (more on phases section about DepPhase anti-pattern above). It is therefore logical that any type that acts as supertype of phase should have at least one such property.

UndefPhase anti-pattern (similarly to UndefFormal anti-pattern) aims to identify every phase, whose supertype is lacking any intrinsic property (modelled either directly as property or indirectly as connected mode/quality) and notify modeller about their existence.

There are three refactoring options, but before we can talk about them we need to first look at derived and intentional partitioning.

### 2.19.1 Derived partition

Derived partition as its name suggest is derived from intrinsic properties of parent for all phases in the partition. Example for this can be seen in figure 2.35. Here we have kind ”Person” with quality ”Age” and derived partition consisting of phases ”Baby”, ”Child”, ”Teenager”, ”Adult” and ”Senior”. All of the aforementioned phases (and their transitions) have been derived based on value of quality ”Age”.

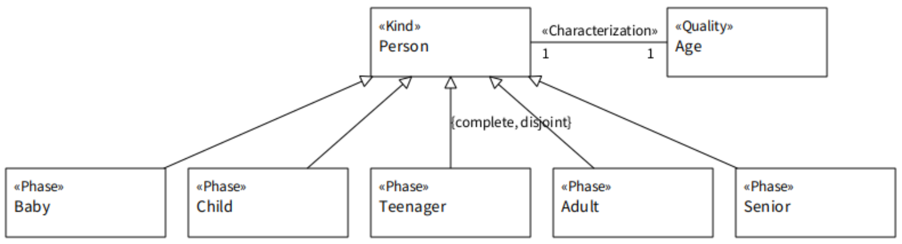


Figure 2.35: Example of derived partition

### 2.19.2 Intentional partition

Intentional partition still requires intrinsic property or properties, but those properties don’t have to be owned by a common parent. Some or even all of them can be connected directly to the phases themselves and thus can appear and disappear. Example of intentional partition is simple classification of ship states (figure 2.36). We can say that any ship can be either afloat or sinking and that ship is sinking when it has hole under waterline, thus the ship state is determined by existence or absence of hole under waterline.

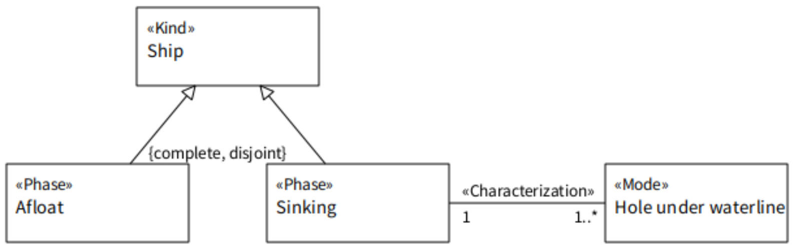


Figure 2.36: Example of intentional partition

### 2.19.3 UndefPhase anti-pattern

As was mentioned previously there are three refactoring options: creation of derived partition, creation of intentional partition and changing stereotype of the phase.

If we intend to create derived partition, we need to make sure that there is common supertype with intrinsic property (represented as property, quality or mode). Additionally we might need to create an OCL invariant to define transitions between the phases.

If we intend to create intentional partition the process is even simple. All we have to do is to make sure that every phase is characterised by at least one mode or quality (it's possible to let one phase without mode or quality, this is called partition through exclusion).

Last option for refactoring is used when we decide that the types were stereotyped as phases incorrectly. Usually this means that they should be stereotyped as roles instead, but this depends on the current context.

We don't provide explicit example of UndefPhase occurrence as it can be easily obtained from either figure 2.35 or 2.36 by removing the quality/mode.

## 2.20 Whole composed of overlapping parts (WholeOver)

Last anti-pattern discussed in this work is also last representant of "Over" anti-patterns. It follows similar logic as RelOver, PartOver and BinOver anti-patterns. It's closes connection is to the PartOver anti-pattern, since they both deal with overlapping types in part-whole relations, but each is dealing with overlap on different end.

Contrary to what the shortcut suggest, WholeOver anti-pattern focuses on identification of overlapping parts. Since we already covered all definitions in sections about BinOver, PartOver and RelOver, we won't repeat them here again. Instead, we will present WholeOver specific constraints and examples.

There are two conditions that need to be fulfilled before we can talk about WholeOver occurrence:

1. Total sum of upper bound cardinality for all part-whole relations on part ends has to be greater or equal to two.
2. Parts have to be part of overlapping set.

Example of WholeOver anti-pattern occurrence can be seen in figure 2.37. It contains fragment of OntoUML model describing an orchestra. Model itself consist of kind "orchestra" that represents the whole for multiple part-whole relations with roles "flutist", "soloist" and "violinist", that are specializations of kind "person". Note that for greater clarity of this example, we have omitted all types and associations that would be required to fulfil relational dependencies of aforementioned roles.

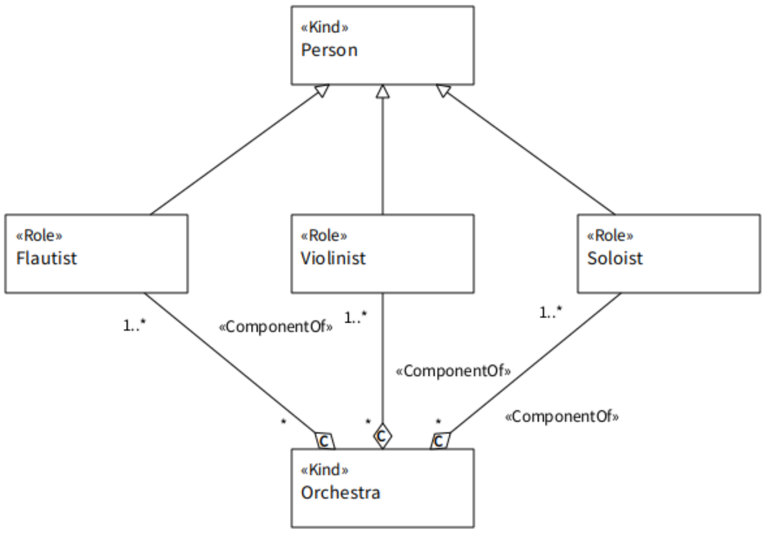


Figure 2.37: Example of whole composed of overlapping types

Orchestra consist of multiple flautists, violinists and sometimes there can be multiple soloists. This leads to occurrence of WholeOver anti-patter, which can be corrected similarly to PartOver/RelOver anti-patterns in three different ways:

1. using OCL invariant to enforce exclusivity,
2. using OCL invariant to enforce partial exclusivity (this would be used to correct our example),
3. removing overlap between the parts by forcing disjoitness.

# Part II

# Design and implementation

Chapter 3

# User interface for verification framework

This chapter describes both design process and final implementation of user interface (UI) for verification framework.

This chapter consists of three sections. First section talks about designs, while second describes the initial implementation and its shortcomings. Lastly thirds sections focuses on the final implementation and it's challenges.

## 3.1 Design of the UI

Even though the user interface was the second part of this thesis to be implemented, preparations for it started during creation of the verification framework in form of classes `VerificationResults` and `VerificationMessage`. But before we get into the initial implementation and resulting problems, we have to first talk about the design itself.

User interface for verification framework has to fulfil three functions:

1. *provide* user a way to start the verification that will be both simple and quick,

2. *display* the results in separate window, that contains all necessary information in one place,

3. *highlight* all model elements (entities, relations and generalizations), for which either errors or warnings have been found. Also selecting verification result should change highlight of the connected element.

Each of the following three subsections discussed design choices made to incorporate the aforementioned functions.

### 3.1.1 UI for starting verification process

Designing interface for the first function (starting the verification process) was fairly straightforward. It was decided to expand ”Diagram” drop-down menu at the in the top toolbar of OpenPonk editor.

Initially there was going to be only one button for starting the whole verification process including detection of anti-patterns, but after further analysis, it was decided to split it into two buttons – one for applying verifications and one for detection of anti-patterns.

In the latter part of implementation process we have also decided to add third button for removing all highlights from model, otherwise they would stay active even after closing the results window.

### 3.1.2 Results window

In previous section we have talked about the way to start the verification process. It equally important to display the results.

It was decided that the best way to do this, would be to create new window containing table with multiple columns. Each row would contain single verification result (represented in the data model as`VerificationMessage`) and columns would correspond to its properties.

Results window should also allow user to inspect selected row directly through built-in Pharo inspector and selecting row in the table should also highlight the corresponding element in the model window.

Last step of the design was deciding which properties would be displayed and how will they be ordered. Following list contains all displayed properties in order from left to right.

1. *Severity* – warning for anti-patterns, error for verifications.
2. *Object* – reference to the verified element
3. *Reason* – contents of property `messageText`.
4. *Documentation reference* – link to the documentation at ontouml.org portal.

## 3.2 Initial implementation

Even though this implementation wasn't entirely successful, most of its code was used in the final implementation. We will therefore start our description with class `VerificationUI` that was created as part of the initial implementation and completely unchanged stayed as part of final implementation.

This subsection will be then followed by brief description of results window implemented using library Spec and resulting problems.

### 3.2.1 VerificationUI

Previously verification process was started by sending message `verify:` with the verified model as a parameter to the instance of `VerificationController`.

As we discussed in the design section, our goal was to create button in the drop-down menu ”Diagram” that would both start the process and after it finishes passed its results to the results window.

This required creation of new subclass of `ComposablePresenter` (from library Spec-Core) called `VerificationUI`. Following list contains descriptions of its most important methods (please note that some methods in the list are referenced using abbreviations, full names are in the footnote):

**`changeElementColorTo:`**[3] *Class method*, that is responsible of changing color of single element. It’s parameters are color and controller of the element.

**`removeHighlights:`**[4] *Class method*, responsible for removing highlights from all elements in the model. Its parameter is diagram-Controller for the of the verified model.

**`toolbarMenu:`** *Class method*, that checks if the currently displayed model uses OntoUML profile. This is needed to prevent verification framework UI from showing up in models from different languages.

**`toolbarMenuItemFor:`** *Class method*, that defines buttons ”verify model”, ”detect anti-patterns” and ”remove highlights”.

**`updateElementStyle:`**[5] *Class method*, that checks if results contain errors or warnings for the updated element and changes its color accordingly. It takes two parameters instance of `VerificationResults` and controller of the updated element.

**`updateElements:`**[6] *Class method*, that first updates all verified elements by calling `updateElementStyle:` for each of them and then opens results window. It has two parameters instance of `VerificationResults` and controller of the verified model.

Buttons ”verify model” and ”detect anti-patterns” defined by method `toolbarMenuItemFor:` send messages directly to `VerificationController` and pass its results to `updateElements:`[5].

[3]`changeElementColorTo:using:`

[4]`removeHighlightsFromElementsIn:`

[5]`updateVerifiedElementStyle:using:`

[5]`updateVerifiedElements:using:`

Because `VerificationController` contained only methods for gathering and applying verifications, we had to add two additional methods:

**`getAntiPatterns`**[7] *Instance method*, that gathers all anti-pattern verifications located in `OntoUML-VerificationFramework`.

**`detectAntipatternsIn:`** *Instance method*, that applies anti-pattern verifications on model passed in parameter.

### 3.2.2 Spec2 results window

Initial implementation of results window was created using library Spec (we also tried its newer version Spec2, which can be downloaded from [18]). This library was chosen because it is used through most of the OpenPonk UI and therefore it would be easier to make the new results window math the style of OpenPonk.

The goal was to create results window with multicolumn table. Documentation for such element was rather scarce, so in the end it was decided to use class `MultiColumnListModel`. But with it lied the first problem, each row in this multicolumn list consist of several cells and each cell is represented by a single string.

It was therefore needed to separately obtain each displayed property from `VerificationMessage` and one by one convert them to string, then those values had to be in correct order put into collection representing row and all rows had to be put into additional collection along with table header to make contents of the table.

`MultiColumnListModel` was proving to be a bit clumsy, but the main problem came during implementation of row inspection and highlighting element corresponding to the selected row. Since all rows were just collections of strings, they didn't contain any reference to the `VerificationMessage` and therefore no reference to the verified element.

Those problems could be partially resolved, but it would result in fragile code that would be very hard to maintain. So in the end it was decided to abandon Spec and instead use resources build in to the Pharo itself.

## 3.3 Final implementation

In previous sections we have described class `VerificationUI`, changes to class `VerificationController` and implementation of results window using library Spec2.

We have also discussed problems with the Spec2, which lead to creation of second and final implementation. But before we start with its description,

[7] getAntiPatternVerifications

it is necessary to mention, that both `VerificationUI` and changes to class `VerificationController` have been kept over from the initial implementation.

### 3.3.1 Results window using inspector tool

Problems with the previous implementation forced us to look for another solution. We have found it in the tool which has been built into the Pharo from the very start. It's called *Inspector*.

This tool allows inspection of any class or its instance in current Pharo image. It is also possible to customize contents of the inspection window by implementing instance method `gtInspectorTableIn:` in the inspected class.

Since verification process returns its findings in form of instance of class `VerificationResults` that holds individual `VerificationMessages`, we had to create two more UI classes. First let's look at class `VerificationMessageUI` which has only one method:

**`getObjectReferenceName:`** *Class method*, that returns formatted name of referenced object. It has one parameter, which is instance of `VerificationMessage`.

Second class `VerificationResultsUI` has much more methods, twelve of them to be exact. To increase clarity of the following list we have omitted six of them, because they are just simple getters and setters for properties `verificationResults`, `controller` and `highlights`. Please note that some methods are referred using abbreviations, due to their name being too long. Their full names are in the footnotes.

**`changeElementColor:`**[8] *Instance method*, responsible for changing color for single element. It's parameters are element and color.

**`createUIFor:`**[9] *Class method*, parametric constructor that takes instance of `VerificationResults` and collection of UI controllers for all elements (classes, generalizations and associations) in the model as parameters.

**`getElementColor:`** *Instance method*, that returns stroke colour of element, that was passed in parameter.

**`gtInspectorTableIn:`** *Instance method*, defines custom inspection page with multicolumn table, containing all verification results.

[8] `changeElementColor:toColor:`

[9] `createUIFor:withControllers:`

**`highlightElement:`** *Instance method*, responsible for highlighting selected element. It first checks if it is already highlighted and stops if it is. Otherwise it changes color of all highlighted elements to the previous one, saves color of the selected element and then highlights it. Parameter of this method is the element that should be highlighted.

**`notifySelectChanged:`** *Instance method*, acting as event handler responsible for calling `highlightElement:` whenever selection changes.

Even this implementation started with some problems, the most notable one was acquiring currently selected row in the table. There were multiple tries to solve this problem, but in the end it was solved by creating new class of called `GLMTableSelectablePresentation`, that expands parent class with the methods necessary for reliably acquiring the selected row.

Expansion itself consist of adding collection for `selection:` handlers, method for registration of those handlers and finally overriding of method `selection:` to include call to all registered handlers. Code of this override can be seen bellow.

```
selection: anObject
  super selection: anObject.
  onSelectHandlers do: [ :handler |
     handler notifySelectChanged: self selection ].
```

Chapter 4

# Updating framework

Even though it is not the biggest part of this thesis, it was definitely the most challenging one. Updating framework itself is extension of verification framework and can be split into three parts characterised by affected classes: updating process itself represented by `OntoUMLVerificationUpdater`, parsing OntoUML definitions using `OntoUMLStereotypeYaml` and lastly changes made to updated verification classes.

But before we can delve into describing implementation of each of those parts, we first need to discuss design of the updating framework as a whole.

## 4.1 Design of the updating framework

Significant part of verification classes matches one or more properties of the verified element against collection of permitted values. Good examples for this are `AllowedSupertypeVerification` and `AllowedSubtypeVerification`.

Both of them inspect generalization and check if the stereotype of supertype/subtype is permitted based on the stereotype of the other type. This is done through dictionary with keys represented by OntoUML class stereotypes and values mapped to collections of allowed class stereotypes.

All those values have been hard coded into the verification classes. Ordinarily this would be just a small issue, but since OntoUML is being continuously developed and expanded, it was necessary to create user friendly process that would automatically apply changes in the OntoUML specification directly to the the verification classes.

Rather than saving all OntoUML specifications into separate files and loading them each time user wants to verify a model, we have decided to try meta-programming, using code reflection and through it allow verifications to regenerate their own source code. This split the work on the framework to the three aforementioned parts: parsing and transforming OntoUML specification, defining and controlling the updating process and updating the verification classes themselves.

Following three sections each delve into both design and implementation for each of the parts and each section is named after the representative class.

## 4.2 OntoUMLStereotypeYaml

First task of the updating framework is to download and parse OntoUML specification for stereotypes. This specification is currently located in Github repository together with the source files for the ontouml.org portal. Each OntoUML stereotype has its own specification file written in YAML.

*YAML™ (rhymes with "camel") is a human-friendly, cross language, Unicode based data serialization language designed around the common native data structures of agile programming languages.* [19]

This meant that each file containing specification for a single stereotype had to be parsed and transformed into form suitable for further use by updating framework. Parsing itself is done in `OntoUMLVerificationUpdater`, but its result are still not suitable for use by updating framework.

We have therefore created class `OntoUMLStereotypeYaml`, that encapsulates the parsed document, defines basic interface for accessing the raw result and also implements parsing corrections (there are some minor differences between internal stereotype names in OpenPonk and in specification files).

There are two additional subclasses `OntoUMLClassStereotypeYaml` and `OntoUMLAssociationStereotypeYaml`. Each of them defines methods for accessing properties for the parsed class/association stereotype specification.

Here is the list of methods of class `OntoUMLClassStereotypeYaml` that allow access to similarly named properties:

| | |
|---|---|
| **`abstract`** | specifies if the stereotypes is abstract, possible values are true, false or undefined. |
| **`dependency`** | specifies if the stereotype is relationally dependant. |
| **`forbiddenAssociations`** | specifies collection of forbidden relations. |
| **`identityPrinciple`** | defines how/if stereotype follows identity principle. |
| **`name`** | specifies name of the class stereotype. |
| **`providesIdentity`** | defines if stereotype provides identity for its subtypes. |
| **`rigidity`** | specifies rigidity of the stereotype. |
| **`subtypes`** | specifies collection of allowed subtypes. |
| **`supertypes`** | specifies collection of allowed supertypes. |

Relationship stereotype specifications have a slightly different format (represented by `OntoUMLAssociationStereotypeYaml`) and include several structured properties, most notably properties for both source and target end. We have decided to create method for accessing the ends as a structure and their ”subproperty” `allowed` due to its importance for the updated verifications.

`OntoUMLAssociationStereotypeYaml` provides following methods.

| | |
|---|---|
| **`allowedSources`** | specifies collection of allowed stereotypes at the source end. |
| **`allowedTargets`** | specifies collection of allowed stereotypes at the target end. |
| **`binaryProperties`** | structured property, that specifies reflexivity, transitivity, symmetry and cyclicity. |
| **`directed`** | specifies if the relation is directed or directionless. |
| **`name`** | specifies name of the relation stereotype. |
| **`sourceEnd`** | structured property, that specifies values for lower and upper multiplicity bounds, `readOnly` mate-property and list of allowed stereotypes for the source end. |
| **`targetEnd`** | structured property, that specifies values for lower and upper multiplicity bounds, `readOnly` mate-property and list of allowed stereotypes for the target end. |

Every single method had to be tested, but this lead to a problem. Parsed specification is fairly complicated structure and it would require test data of similar complexity. It was therefore decided to create ”`Mock`” subclass for both `OntoUMLClassStereotypeYaml` and `OntoUMLAssociationStereotypeYaml`.

These ”`Mock`” classes define setters for all properties that don’t already have them. This allows for massive simplification of test data and creation of clear unit tests.

## 4.3 OntoUMLVerificationUpdater

`OntoUMLVerificationUpdater` is the hearth of the updating framework. It is responsible for control of the entire updating process and as such contains single method that starts the entire process, methods for downloading and parsing specification from source files, transforming the parsed results using `OntoUMLClassStereotypeYaml` and finally calling method `updateFrom:` for every single verification.

First we’ll discuss how are the specifications downloaded and parsed in separate subsection and then describe execution of the update itself.

### 4.3.1 Specification loading and parsing

We have previously mentioned that the specifications are located in Github repository together with other source files for the ontouml.org portal. Each stereotype has its own separate specification and since the repository is structured mostly according to OntoUML stereotype hierarchy, each of the specifications is located in its own directory and may or may not share some part of its path with specifications for other stereotypes.

Hardcoding references to for such structure into the updating framework would be highly inefficient and it would lead to unmanageable code. We have therefore created file called `updatingIndex.yaml`. This file is located at the very top of the repository and contains paths to all stereotype specifications.

First step of the loading process is to download and parse this index. Parsing itself is done using class `PPYAMLGrammar`, which is part of `PetitParser` framework. This open-source framework can be downloaded from here [20].

After loading and parsing of the index, we need to load and parse all stereotype specification mentioned in the index. Parsing is again done using `PPYAMLGrammar`, but this time we have to encapsulate each parsed specification either into instance of `OntoUMLClassStereotypeYaml` or into instance of `OntoUMLAssociationStereotypeYaml` based on the type of the specification.

Resulting collection is then passed to the updated verifications, but we'll look at it in more detail in the next subsection. Before that here is list of most important methods used to loading and parsing of both `updatingIndex.yaml` and stereotype specifications. Some methods are referenced using abbreviations, full names are in footnotes.

| | |
|---|---|
| **`loadClassSpec:`**[10] | *Instance method*, responsible for downloading single class specification. Takes url of the specification as a parameter. |
| **`loadClassSpecs`**[11] | *Instance method*, responsible for downloading all class specifications. |
| **`loadDocRef`**[12] | *Instance method*, responsible for loading collection of references to documentation, that is located at portal ontouml.org. |
| **`loadIndex`** | *Instance method*, responsible for both downloading and parsing `updatingIndex.yaml`. |

[10] `loadClassSpecification:`

[11] `loadClassSpecifications`

[12] `loadDocumentationReferences`

| | |
|---|---|
| **`loadRelSpec:`**[13] | *Instance method*, responsible for downloading single relation specification. Takes url of the specification as a parameter. |
| **`loadRelSpecs`**[14] | *Instance method*, responsible for downloading all relation specifications. |
| **`parseClassYaml:`** | *Instance method*, responsible for parsing and transforming specification of single class stereotype. Takes contents of YAML document as parameter. |
| **`parseRelYaml:`**[15] | *Instance method*, responsible for parsing and transforming specification of single relationship stereotypes. Takes contents of YAML document as parameter.. |
| **`parseYaml:`** | *Instance method*, responsible for parsing all YAML documents. Takes contents of YAML document as parameter. |

### 4.3.2 Managing updating process

So far we have described how `OntoUMLVerificationUpdater` handles downloading, parsing and transforming of stereotype specifications. All that is left is to discuss how does it handle updating itself.

This can be best explained by its only *class method* `updateVerifications`. Please note that there are two methods called `updateVerifications` one is *class method* and it's code can be seen directly bellow this paragraph and the second *instance method* will be described later.

```
updateVerifications
    | updater |
    updater := self new.
    updater classSpecifications: updater loadClassSpecifications.
    updater relationshipSpecifications:
        updater loadRelationshipSpecifications.
    updater loadDocumentationReferences.
    updater updateReferences.
    updater updateVerifications.
```

From the code above we can see the entire process. First it creates new instance of `OntoUMLVerificationUpdater`, then it loads, parses and transforms class and relationship specifications, which are then saved into internal

[13] `loadRelationshipSpecification:`
[14] `loadRelationshipSpecifications`
[15] `parseRelationshipYaml:`

collections. It also parses and saves all references to OntoUML documentation.

Now to the last two methods `updateReferences` and `updateReferences`. Both of them are fairly simple with similar implementation, but both are crucial to the updating framework. Following list contains descriptions for them and most important accessing methods. Some methods are referred using abbreviations, their full names are in the footnotes.

| | |
|---|---|
| **`updateReferences`** | *Instance method*, that forces update of constants in class `OntoUMLDotOrgReferences`. |
| **`updateVerifications`** | *Instance method*, that forces update of constants in all verification classes. |
| **`classDocReferences`**[16] | *Instance method*, that provides access to collection of loaded references to class documentation, located on portal *ontouml.org*. |
| **`classSpecifications`** | *Instance method*, that provides access to collection of `OntoUMLClassStereotypeYaml`s encapsulating specifications for class stereotypes. |
| **`relDocReferences`**[17] | *Instance method*, that provides access to collection of loaded references to relationship documentation, located on portal *ontouml.org*. |
| **`relSpecifications`**[18] | *Instance method*, that provides access to collection of `OntoUMLAssociationStereotypeYaml`s encapsulating specifications for relationship stereotypes. |

List above contains several accessing methods, that are used by updated classes. Putting all stereotype specifications into one collection would only increase overhead, because each updated class would have to filter between class and relationship specifications.

Passing only the correct collection to the updated class wouldn't help that much. `OntoUMLVerificationUpdater` would have to implement logic for distinguishing between different types of updated classes (class/relationship verifications).

Access methods allow us to use *double dispatch*, i.e., pass reference for `OntoUMLVerificationUpdater` to the updated class and let the class select the correct collection of stereotype specifications. *Double dispatch* also simplifies future expansions of the updating framework.

[16]`classDocumentationReferences`
[17]`relationshipDocumentationReferences`
[18]`relationshipSpecifications`

*Instance method* `updateVerifications` implements one half of the *double dispatch* pattern, other half is implemented in all updated classes. Following source code contain implementation of the aforementioned method.

```
updateVerifications
    self getVerifications do: [ :verification |
        verification updateFrom: self ].
```

## 4.4 Changes to verification framework

Previous sections talked about updating framework and classes from which it consists. This section describes changes and additions to the verification framework, that were needed to connect both frameworks together.

We will start our description with changes to the structure of packages that form the verification framework and continue with newly created traits `TClassUpdater` and `TCodeInjectionChecker`. Next we will discuss creation of class `OntoUMLDotOrgReferences`, which will be followed by description of changes done to `Verification` itself. This section will be finished with example: implementation of updating methods by `AllowedSubtypesVerification`.

### 4.4.1 Restructuring of the verification framework

Verification framework was originally split into five separate packages, which proved to be impractical. Therefore as part of development of updating framework, we have merged all five packages into one.

Each package is now represented by tag applied to all classes that used to be part of the original packages. Same principle bas been applied for all new classes created as part of this thesis, so for example all UI classes have been marked by tag `UI`, all verifications are now tagged as `verification-classes` and so on.

This lead to changes in `VerificationController`, most notably in its method `getVerifications` originally loaded all verifications from their own separate package.

### 4.4.2 New traits

Before we can talk about `OntoUMLDotOrgReferences` and `Verification`, it is necessary to describe two new traits created for the updating framework. Those traits are `TClassUpdater` and `TCodeInjectionChecker`. They are applied through composition to all updated classes.

Let's start with the `TClassUpdater`. It contains single *class method* called `updateMessage:withClassification:`. As we can see from its source code bellow, it takes two parameters `aMessageCode` (contains code of the newly created method) and `aClassification` (name of protocol, under which it will be saved) and compiles new method into the trait / composing class.

```
updateMessage: aMessageCode withClassification: aClassification
   self class compile: aMessageCode classified: aClassification.
```

Trait `TCodeInjectionChecker` is used in conjunction with `TClassUpdater`. It contains two methods, that are called `checkStringForCodeInjection:` and `checkUrlForCodeInjection:`. Both of them are used to prevent generated code from injecting and executing malicious code.

### 4.4.3 OntoUMLDotOrgReferences

Part of the updating framework are also references to documentation on portal *ontouml.org* and their updating. This means two things. First, that it would be necessary to expand `VerificationMessage` by adding methods for getting and setting documentation reference and second, that new class for holding all references to portal *ontouml.org* would have to be created.

Expansion of the `VerificationMessage` class added new property called `documentationReference` and its getter and setter. Since verifications don't have their own documentation (they are result of class and relationship specifications), most of the documentation references would be determined by the verified object.

We have therefore expanded getter for `documentationReference`. If first checks if the values is not nil (not empty) and if it is empty, loads documentation reference for the verified object into the property before it returns its value as normal.

This leads to us to class `OntoUMLDotOrgReferences`. It holds all references for documentation on class and relationship stereotypes. Each reference is represented by its own method named after the referenced stereotype and the url itself is hardcoded. Normally this would lead to code that would be difficult to manage, but because this class was created as part of the updating framework, all reference methods were generated using the updating process and can be simply regenerated after every change in the documentation.

Following list contain the most important methods implemented by class `OntoUMLDotOrgReferences`. Note that this list is not complete due to sheer number of the methods and due to the fact, that several methods are obtained from traits `TClassUpdater` and `TCodeInjectionChecker`. Some methods are referred using abbreviations, their full names are in the footnotes.

| | |
|---|---|
| **`getDefRefFor:`**[19] | *Class method*, that takes class or relation as parameter and returns url to documentation of applied stereotype applied to class/relation. |

[19] `getDefinitionRefereceFor:`

**getDefRefForSterCode:**[20] *Class method*, that takes key-value pair consisting of stereotype name and documentation reference as a parameter. It returns string containing with source code for method with hardcoded documentation reference.

**updateFrom:** *Class method*, that has one parameter, which is instance of `OntoUMLVerificationUpdater` and regenerates all methods with documentation references.

`OntoUMLDotOrgReferences` also contains methods for all references to stereotype documentation (and in future to documentation of anti-patterns).

Those methods are named in following way `getDefinitionRefereceFor` followed by name of the referenced stereotype.

### 4.4.4 Updating verification classes

Next step in the development of the updating framework was expansion of the `Verification` abstract class. This meant adding following three new methods (some of them are referred using abbreviations, their full names are in the footnotes):

**expandAllowedClStIn:from:**[21] *Class method*, that takes two parameters: collection of strings (stereotype names) and instance of `OntoUMLVerificationUpdater`. It iterates through the collection and if necessary expands subcolections defined by values "functional complex" or '*'.

**expandClSt:from:**[22] *Class method*, takes string containing class stereotype name, functional complex or '*' (symbol for all stereotypes), if necessary expands it using second parameter (instance of `OntoUMLVerificationUpdater`) and returns collection of strings.

**updateFrom:** *Class method*, that takes one parameter instance of `OntoUMLVerificationUpdater`. Subclasses that want to included in the updating process need to override this methods.

All three methods are meant to be used by subclasses of `Verification`. Method `updateFrom:` was kept intentionally empty instead of making it abstract and because of this `OntoUMLVerificationUpdater` can send this message to all verifications and update only those that override it with own their specific code. Example of such implementation is in the next subsection.

[20] `getDefinitionReferenceForStereotypeCode:`

[21] `expandAllowedClassStereotypesIn:from:`

[22] `expandClassStereotype:from:`

### 4.4.5 Example implementation of AllowedSubtypesVerification

Until now we have described every principle behind the updating framework, from loading, parsing and transforming specifications to changes made to class `Verification`. But there one thing is still missing – example for the implementation of the updating methods in non-abstract class.

Therefore we will look at class `AllowedSubtypesVerification`, starting with implementation of method `updateFrom:`.

```
updateFrom: aVerificationUpdater
  | updatedMessageCode |
  updatedMessageCode :=
     self getUpdatedMessageCode: aVerificationUpdater.

  aVerificationUpdater classSpecifications
    do: [ :classSpecification |
      self setValidSubtypesFor: classSpecification name
        fromCollection: classSpecification subtypes. ].

  self updateMessage: updatedMessageCode
    withClassification: 'constants'.
```

As we can see implementation of method `updateFrom:` does three things. First it call method `getUpdatedMessageCode:`, that returns string with source code for method `validSubtypes` which builds stereotype dictionary from Pharo associations (key-value pairs) using class specifications contained in `OntoUMLVerificationUpdater`.

Next block of code (re)generates methods, that contain aforementioned Pharo associations with class stereotype as key and collection of allowed subtypes as value.

Last block of code uses variable `updatedMessageCode`, that contains code obtained in the first step, to (re)generate method `validSubtypes`.

All updated verification contain similar implementation of `updateFrom:`, first calling another method to get the new code and then (re)generating it.

Before the end of this section (and chapter) let's look at two more code snippets first is shortened implementation of method `validSubtypes` (we have omitted several lines, which build parts of the dictionary).

```
validSubtypes
    | result |
    result := Dictionary new.
    result add: self validSubtypesForKind.
    result add: self validSubtypesForRoleMixin.
    ^ result.
```

Second code snippet contains method `getUpdatedMessageCode:`, which is used to generate string containing source code of the aforementioned method `validSubtypes` showed above (we have removed all comment lines from this code for this example).

```
getUpdatedMessageCode: aVerificationUpdater
    | validSubtypesCode |
    validSubtypesCode := 'validSubtypes
    | result |
    result := Dictionary new.'.

    aVerificationUpdater classSpecifications
        do: [ :classSpecification |
            self checkStringForCodeInjection:
               classSpecification name.

    validSubtypesCode := validSubtypesCode ,
        (String with: Character cr),
        'result add: self validSubtypesFor' ,
            classSpecification name, '.' ].

    validSubtypesCode := validSubtypesCode ,
        (String with: Character cr),
        '^ result.'.

    ^ validSubtypesCode.
```

As we can see in the generating code above, every time we include any value outside of the prepared code we need to check if it does not contain unexpected code, which would cause code injection.

Chapter 5

# Design and implementation of anti-pattern verifications

Even though the implementation of anti-pattern verification/detection represents the biggest part of this thesis (in the terms of analysis, implementation and documentation), this design and implementation chapter will be shorter.

Sizeable part of the design was done during the development of the verification framework, which was created by author as his bachelor's thesis. Verification framework was conceived with possibility of anti-pattern extension in mind and this greatly reduced time needed for design and implementation of the anti-pattern verifications themselves.

This chapter is divided into two sections. First section contains brief design descriptions for each implemented anti-pattern and second explains general principles behind anti-pattern implementation.

## 5.1 Design of anti-pattern verifications

This section is split into twenty subsections one for each implemented anti-pattern. Each subsection contains brief description of algorithm used for detection of the discussed anti-pattern. Please note that we refer to all anti-patterns using their abbreviations.

### 5.1.1 Design of BinOver anti-pattern verification

Occurrence of anti-pattern BinOver is characterised by relation between two overlapping types *source* and *target*. Since BinOver detects overlap for all relations it was one of the hardest anti-patterns to implement, as it required the most precise detection of overlapping types.

Types $A$ and $B$ are considered to be overlapping when they fulfil at least one of the following conditions:

1. $A$ is equal to $B$.

2. $A$ is ancestor of $B$ or $B$ is ancestor of $A$.

3. $A$ and $B$ are sortals, have the same identity provider and there is no generalization set, that makes them disjoint.

4. $A$ and $B$ are both relators or modes with common ancestor and there is no generalization set, that makes them disjoint.

5. $A$ and $B$ are mixins, that either have at least one common sortal descendant, or have common ancestor and none of their subtypes are sortals.

This means that we had to implement methods for obtaining identity providers, ancestor trees and using those calculate ”identity paths” for both types $A$ and $B$, before we could check the overlap.

It was also necessary to check for generalization sets in the common parts of the ancestor tree that have meta-property `isDistinct` set to true.

## 5.1.2 Design of DecInt anti-pattern verification

DecInt anti-pattern occurs when type specializes two or more concrete types. Anti-pattern itself has just three additional conditions:

1. Generalization between type and its parent must be syntactically valid.

2. Parent type mustn't be *abstract*.

3. All generalization sets, that have parent type as supertype must have meta-property *isCovering* set to false.

We have decided to omit the first additional condition in our implementation. This will result in slightly more detected occurrences, but in this case we think its better to warn the modeller using both dedicated verification and this anti-pattern.

Latter two additional conditions are part of single method that returns all concrete parents for the tested type.

## 5.1.3 Design of DepPhase anti-pattern verification

DepPhase anti-pattern occurs when there is a phase directly connected to relation stereotyped as mediation.

Due to simplicity of the rule, we have decided to implement it using only one method that simply checks if the phase is connected to at least one mediation.

### 5.1.4 Design of FreeRole anti-pattern verification

FreeRole anti-pattern occurs when there is a role, that has its mediation dependency fulfilled indirectly by ancestral role or if is mediation dependency is unfulfilled.

Therefore we have to start our verification of role by getting all its ancestors stereotyped as role. Then get all mediations that are directly connected to the role or one of the ancestral roles. After that we have to check if the inspected role is directly connected to at least one mediation and if not, check the ancestral roles in the same way.

Again we have decided to omit part of this anti-pattern - complete check for fulfilment of mediation dependency. This is covered by existing verification called `RoleMediationDependencyVerification`.

### 5.1.5 Design of GSRig anti-pattern verification

GSRig anti-pattern occurs when single generalization set with rigid supertype contains subtypes that follow different rigidity principles.

During design we have decided to implement it using four methods. First method checks if the supertype is rigid. Then we have two methods one for finding rigid subtypes and one for finding anti-rigid subtypes. Last method manages the entire process and returns the final result.

### 5.1.6 Design of HetColl anti-pattern verification

HetColl anti-pattern occurs when *"collective"* has multiple different parts. Design of the anti-pattern verification is very simple, it contains two methods, one that checks if the *"collective"* has more than one part (either directly, or indirectly through its ancestors) and second one that checks if the type is *"collective"*.

Reason why we are talking about *"collective"* is because this anti-pattern is applicable to following stereotypes:

- collective,
- subkind, phase or role, that have collective as their ancestor (direct or indirect),
- category, mixin, phasemixin or rolemixin, whose children are all *"collectives"*.

### 5.1.7 Design of HomoFunc anti-pattern verification

HomoFunc anti-pattern occurs when functional complex has multiple parts and all parts are of the same type.

We have split the detection into four methods. First method represents the detection process, call all other methods and returns findings. Second and third methods are responsible for verifying that the inspected type is actually functional complex. Last method is needed for obtaining sortal descendants of the inspected type.

### 5.1.8 Design of ImpAbs anti-pattern verification

ImpAbs anti-pattern occurs when at least one end of a relation has its lover bound multiplicity grater or equal to two and the connected type has at least two subtypes.

ImpAbs anti-pattern verification was split into three methods. One method for obtaining lower bound multiplicity, another method for counting subtypes of the inspected type and lastly method, that encapsulates the process and returns its findings.

### 5.1.9 Design of MixIden anti-pattern verification

MixIden anti-pattern occurs when children of type sterotyped as category, mixin, phasemixin or rolemixin follow the same identity principle.

This calls for four methods. First method to manage the detection process, second method for obtaining identity providers of a type, third method for checking if stereotype provides identity and last method for detecting identity overlap.

### 5.1.10 Design of MixRig anti-pattern verification

MixRig anti-pattern occurs when type stereotyped as mixin has only rigid or non-rigid subtypes (those subtypes can be either direct or indirect if all subtypes between the ancestor mixin and descendant type are stereotyped as mixin).

Detection of the anti-pattern occurrence is relatively easy and thus it could be handled by one method, but obtaining all subtypes of the mixin and child mixins had to be handled by separate recursive method.

### 5.1.11 Design of MultDep anti-pattern verification

MultDep anti-pattern occurs when type (other than relator) is mediated by at least two distinct relators. There is an exception to this rule – if one or more connected relators are direct or indirect child of other connected relator, MultDep *does not* occur.

We have split the detection into three methods. One method to manage the process and two for checking if the is hierarchy between the relators.

### 5.1.12 Design of PartOver anti-pattern verification

PartOver anti-pattern occurs when part composes two or more overlapping wholes. OntoUML defines four different path-whole relations: subQuantityOf, componentOf, memberOf and subCollectionOf.

Since part-whole relations limit possible stereotypes that could be overlapping it wasn't necessary to design and implement full overlap detection.

This means that detection of PartOver can be split into just five methods. First it is necessary to get all part-whole relations in which the verified type acts as part. After that we have to check upper multiplicity for all connected part-whole relation. Next we need to check for both regular and mixin overlap. Lastly we have to create method that encapsulates the entire process.

### 5.1.13 Design of RelComp anti-pattern verification

RelComp anti-pattern occurrence is characterised by two different relations: relation $A$, that must have multiplicity bounds set on its target end to at least one for the lower bound and at least two for the upper bound, and $B$. Additionally source or target of relation $A$ has to be ancestor or equal to source and target of relation $B$.

Detection of the RelComp has to be therefore split into four methods: one to represent the detection process, another that checks if one type is ancestor or equal to second type, next method has to check the required cardinality and the last one for that would apply previously mentioned methods on all associations connected to verified type.

### 5.1.14 Design of RelOver anti-pattern verification

RelOver anti-pattern occurs when single relator mediates multiple overlapping types. Anti-pattern RelOver is similar to PartOver anti-pattern.

Because of this we were able to reuse design of the PartOver anti-pattern verification, with the only changes being in the implementation of method representing the detection process and method responsible for obtaining mediated types.

### 5.1.15 Design of RelRig anti-pattern verification

RelRig anti-pattern occurs when relator mediates one or more rigid types and their ends are not marked as read-only.

Detection of RelRig is simple enough to be handled by single method, that collects all mediations connected to the relator and checks if at least one of the mediated ends is rigid and not read-only.

### 5.1.16 Design of RelSpecs anti-pattern verification

RelSpecs anti-pattern occurrence is characterised by two different relations *A* and *B*. Additionally one of the following conditions has to be fulfilled:

- *source* of *A* has to be ancestor of or equal to *source* of B and *target* of *A* has to be ancestor of or equal to *target* of B;
- *target* of *A* has to be ancestor of or equal to *source* of B and *source* of *A* has to be ancestor of or equal to *target* of B.

During design we have decided to implement it using three methods. First method to check if one type is ancestor or equal to second type. Second method to represent the detection itself and third method to apply previous methods to all relations connected to verified type.

### 5.1.17 Design of RepRel anti-pattern verification

RepRel anti-pattern occurs when single relator is connected to multiple mediations and all of those mediations have upper bound multiplicity on the relator end greater or equal to two.

Its anti-pattern verification has three main methods, one method to obtain all mediations for relator and its ancestors, another mediation for getting upper bound multiplicity at the relator end from a mediation and last one to represent the detection itself.

### 5.1.18 Design of UndefFormal anti-pattern verification

UndefFormal anti-pattern occurs when formal relation connects types, that don't own/inherit any qualities or attributes.

We had to implement three methods. First to check if type owns or inherits any attributes, second to check if type owns or inherits any qualities and third that contains the detection algorithm.

### 5.1.19 Design of UndefPhase anti-pattern verification

UndefPhase anti-pattern occurs when common parent of all phases in a partition does not own or inherit any qualities, modes or attributes.

This is another case of anti-pattern detection in which we decided to make a small deviation from the specification. Instead of running detection on generalization sets (which define partitions), we are looking for occurrence of UndefPhase from each phase.

Reason for this difference between specification and implementation lies in the way in which OpenPonk model stores generalization sets which is slightly different to the way in which verification framework obtains all elements for verification / anti-pattern detection.

Using phases to detect occurrence might produce few more false alarms, but all of those are special cases of intentional partitioning (see part Review and analysis, chapter Anti-patterns, section UndefPhase).

Otherwise design and implementation are almost identical to UndefFormal anti-pattern verification.

### 5.1.20 Design of WholeOver anti-pattern verification

WholeOver anti-pattern occurs when whole is composed of two or more overlapping parts. Is almost identical to PartOver only difference between them is that WholeOver looks at whole ends of part-whole relations while PartOver looks at the part ends.

Due to this design and implementation of both anti-pattern verification differs only in few details (mainly relation ends accessed during the detection).

## 5.2 Implementation of anti-pattern verifications

Previous section discussed design of each anti-pattern verification. In this section we are going to describe thei implementation as part of the verification framework.

First of all, we need to explain why are we calling those classes anti-pattern verifications instead of using more appropriate name – anti-pattern detectors. This is result of their integration into the verification framework. All classes created for detecting anti-pattern occurrences are direct or indirect (through class `StereotypeVerification`) subclasses of `Verification`.

This means that every anti-pattern verification must implement following methods:

**`verifiedClass`** *Class method*, that specifies class, to which can be this verification applied.

**`verifiedSterotype`** *Class method*, that specifies collection of stereotypes, that can be inspected by this verification. Only subclasses of `StereotypeVerification` implement this method.

**`verifyObject:withModel:`** *Instance method*, responsible for performing the verification/detection itself. Its parameters are verified object (method *canVerify:* makes sure it will have correct class and stereotype) and verified model.

Rest of the implementation is either specific to each anti-pattern verification, or it's provided by the verification framework (for more information see chapter OntoUML and OpenPonk review, section OpenPonk).

## Part III

# Documentation and testing

Chapter 6

# Testing

This chapter is split into two sections. First sections describes design, implementation and problems with unit test, while second section briefly talks about tests on real models.

## 6.1 Unit tests

No matter how good we thing the design and implementation of any program are, it sis always necessary to create (unit) test to both prove that it works as intended and to help maintaining its integrity during future updates and expansions.

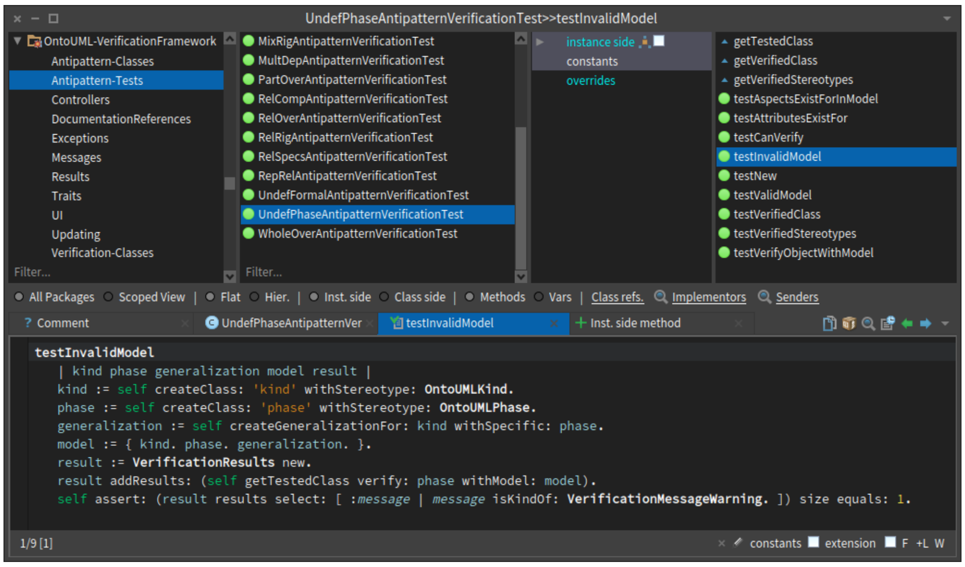
Luckily for us, Pharo provides unit test framework that is both robust and easy to use. With its help we were able to create unit tests for most methods written as part of this thesis.

Even classes that have higher amount of methods without concrete unit test (such as `BinOverAntipatternVerification`), have at least one test that covers all methods at once.

Since all anti-pattern verifications are subclasses of either `Verification` or `StereotypeVerification`, classes containing their unit test had to specialize either `AbstractVerificationTest` or `StereotypeVerificationTest` in similar way. This means that every method inherited from original verifications had already its own unit test prepared and in most cases we had to only prepare data for the test. Example of this can be seen in figure 6.1

Creating unit test for methods generated by updating framework proved to be much bigger challenge. Tests that cover code generation were split into two types. First type validates code of the generated methods without actually compiling them. Second type validates generation itself, i.e., test that we can use appropriate methods to generate/regenerate methods.

In addition all generated methods have to be checked, to be sure that they do what they are supposed to do. This presented another problem number



```
testInvalidModel
   | kind phase generalization model result |
   kind := self createClass: 'kind' withStereotype: OntoUMLKind.
   phase := self createClass: 'phase' withStereotype: OntoUMLPhase.
   generalization := self createGeneralizationFor: kind withSpecific: phase.
   model := { kind. phase. generalization. }.
   result := VerificationResults new.
   result addResults: (self getTestedClass verify: phase withModel: model).
   self assert: (result results select: [ :message | message isKindOf: VerificationMessageWarning. ]) size equals: 1.
```

Figure 6.1: Unit tests for class `UndefPhaseAntipatternVerification`

of the tested methods is not known before runtime. Therefore all generated methods would have to have generated unit test (this would be a bit impractical) or we would have to create single test for each group of generated methods and call test them using a cycle.

We have chosen the latter option. If the test fails we won't know which method caused it immediately (it would require additional analysis), but since all methods covered by one test are generated by the same code, there is a very good chance that all of them are malformed.

However, we run into one more problem – how to call multiple methods without knowing their names before runtime? Or solution lies again in use of meta-programming, concretely in reflection. Following code snippet shows implementation of unit test for methods `validSubtypesFor*`, where `*` should is replaced by name of class stereotype, for which was the method built.

```
testValidSubtypesFor
  | methodList |
  methodList := (AllowedSubtypesVerification class localMethods
    collect: [ :method | method selector ])
      select: [ :methodName |
        methodName matchesRegex: 'validSubtypesFor.*' ].
  self assert: methodList isNotEmpty.
  methodList do: [ :methodName | self assert:
    (((OntoUMLDotOrgReferences class >> (methodName asSymbol))
      valueWithReceiver: nil arguments: #())
        findString: 'http://') equals: 0. ].
```

Code above was split into three blocks for better reading. First block uses reflection to collect all `validSubtypesFor*` methods. Second block checks that the list is not empty and the third block uses meta-programming to call each tested methods using string with its name.

## 6.2 Tests on real models

In addition to the unit test, we have also tested detecting anti-patterns in real models. Model shown in figure 6.2 was the biggest one with well over hundred elements (50 types, 59 relations and 27 generalizations). This model (along with other models used for testing) was created as part of commercial project and therefore we cannot provide much additional information about it, except that it was chosen for its considerable size and complexity.

We have applied both OntoUML verifications and anti-pattern detection to this model and to our pleasant surprise run time for both processes was around one second. OntoUML verifications found two errors, both of them were result of inverting source and target of a mediation.

Anti-pattern detection discovered seventeen anti-pattern occurrences, including one occurrence of ImpAbs, three cases of MixIden, one instance of MultDep, four occurrences of RelRig, four cases of RelSpecs and finally four instances of RepRel.

It is worth noting that this model was obtained during its creation, but as can be seen from the number of verification errors (only two minor ones), its creators focused on its from the beginning correctness. Nevertheless number of detected anti-patterns show, that there are still some problems with its semantics.

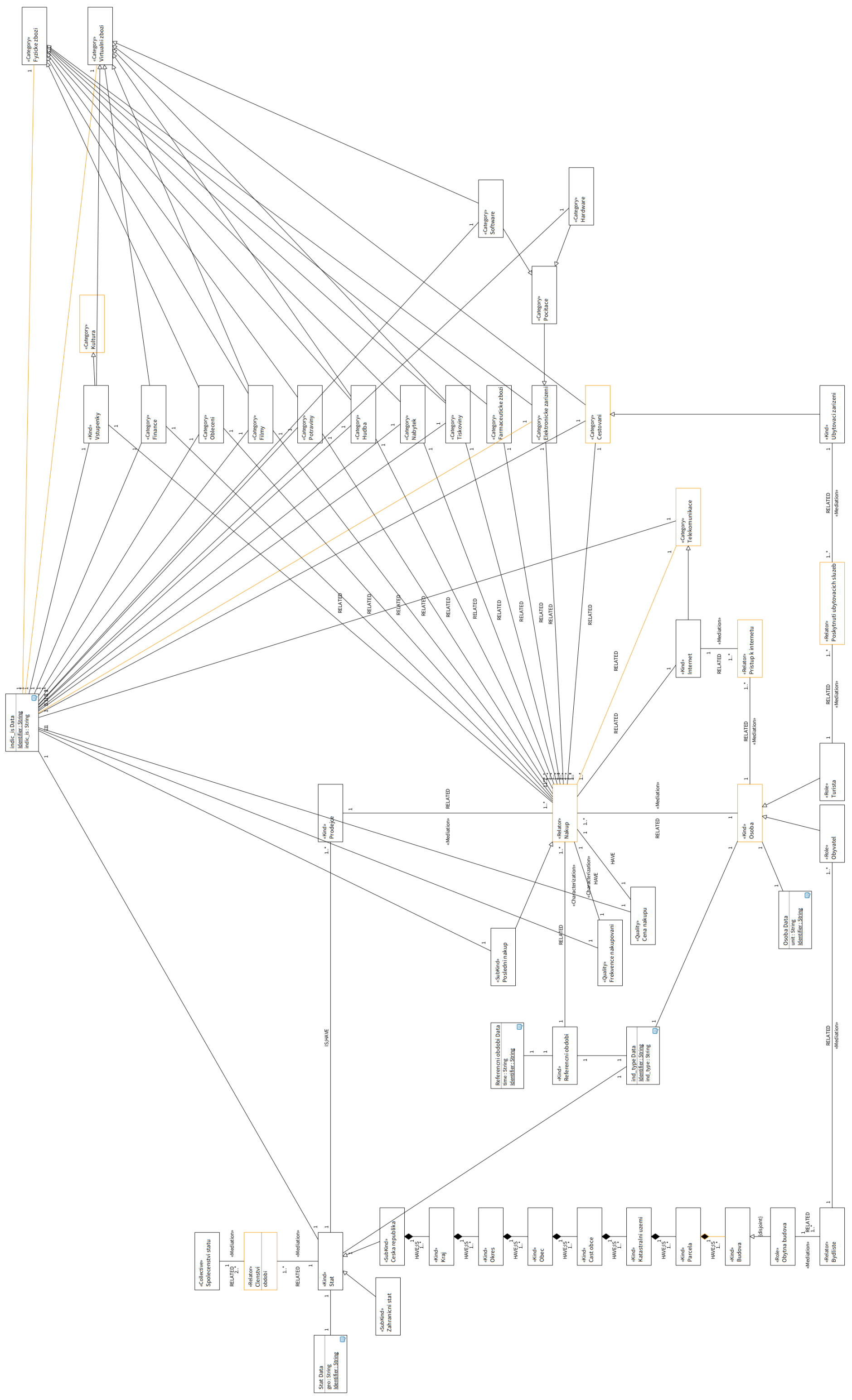


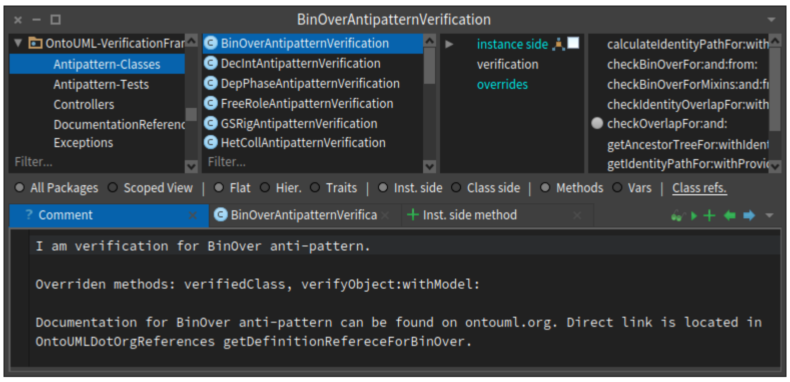
Figure 6.2: Real life reference model after anti-pattern detection

Chapter **7**

# Documentation

Documentation for practical part of this thesis consists of three parts. First part is this text, which contains explanation of for both design and implementation of the practical part.

Second part is located directly in the implementation itself in form of comments for both classes and methods. This can be seen in figure 7.1 which contains comments for class `BinOverAntipatternVerification`.


BinOverAntipatternVerification
OntoUML-VerificationFrar
Antipattern-Classes
Antipattern-Tests
Controllers
DocumentationReferenc
Exceptions
Filter...
BinOverAntipatternVerification
DecIntAntipatternVerification
DepPhaseAntipatternVerification
FreeRoleAntipatternVerification
GSRigAntipatternVerification
HetCollAntipatternVerification
Filter...
instance side
verification
overrides
calculateIdentityPathFor:with
checkBinOverFor:and:from:
checkBinOverForMixins:and:fr
checkIdentityOverlapFor:with
checkOverlapFor:and:
getAncestorTreeFor:withIdent
getIdentityPathFor:withProvic
All Packages
Scoped View
Flat
Hier.
Traits
Inst. side
Class side
Methods
Vars
Class refs.
Comment
BinOverAntipatternVerifica
Inst. side method

```
I am verification for BinOver anti-pattern.

Overriden methods: verifiedClass, verifyObject:withModel:

Documentation for BinOver anti-pattern can be found on ontouml.org. Direct link is located in
OntoUMLDotOrgReferences getDefinitionRefereceForBinOver.
```

Figure 7.1: Documentation for `BinOverAntipatternVerification`

Last part of the documentation is located on portal *ontouml.org*. It consists of twenty pages, one for each anti-pattern. Those pages were created using anti-pattern summary tables form [17].

Format of the tables had to be changed, diagrams had to be extracted form the original text and in some cases, we also had to draw new diagrams with examples.

There are also plans for expansion of the anti-pattern section outside scope of this thesis in form of additional examples taken from this thesis and creation of educational clips that will provide additional explanation of the anti-pattern problematic.

# Conclusion

Goal of this master's thesis was extending OntoUML modelling capabilities of OpenPonk platform with three new functionalities: graphical user interface for displaying verification results, updating framework (including integration into the existing verification framework) and OntoUML anti-pattern detection.

First, we reviewed and analysed OntoUML modelling language, OpenPonk platform, OntoUML verification framework and there was a full chapter dedicated to OntoUML anti-patterns which provided both explanation and examples for each anti-pattern.

Next part focused on design and implementation of aforementioned functionalities, which were successfully integrated into existing verification framework. During the design we put emphasis on ease of use (in case of the user interface), modifiability and expandability. This was done to provide stable and long lasting additions to the OpenPonk platform.

Third part of this thesis was aimed at verifying the new functionalities using unit test for most of the internal classes and reference model for demonstration of anti-pattern detection and user interface.

Last part was dedicated to documentation, which is located at three separate places. First part of the documentation is represented by this thesis, second part is in the newly implemented code and the third part is located in new section of ontouml.org dedicated to anti-pattern.

With this I have fulfilled all goals set at the start of this thesis, but there are still many ways for OpenPonk expansion. There are additional anti-patterns that were out of scope of this thesis, because they would require major overhaul of the verification framework.

There are also OntoUML patterns, that represents structures for typical use cases and their addition would help both new and experienced modellers. Lastly there is possibility for optimalization of the verification framework as a whole. So far it was created with focus on expandability and even though there were no problems with the performance yet, there is still room for improvement.

# Bibliography

1. BĚLOHOUBEK, Marek. *OntoUML Models Verification for the OpenPonk plat-form.* 2019. B.S. thesis. Czech Technical University in Prague, Faculty of Information Technology.
2. GUIZZARDI, Giancarlo. *Ontological foundations for structural conceptual models.* Telematica Instituut / CTIT, 2005. ISBN 90-75176-81-3. PhD thesis. University of Twente.
3. GUIZZARDI, Giancarlo; FONSECA, Claudenir M.; BENEVIDES, Alessander Botti; ALMEIDA, João Paulo A.; PORELLO, Daniele; SALES, Tiago Prince. Endurant Types in Ontology-Driven Conceptual Modeling: Towards OntoUML 2.0. In: TRUJILLO, Juan C.; DAVIS, Karen C.; DU, Xiaoyong; LI, Zhanhuai; LING, Tok Wang; LI, Guoliang; LEE, Mong Li (eds.). *Conceptual Modeling.* Cham: Springer International Publishing, 2018, pp. 136–150. ISBN 978-3-030-00847-5.
4. GUIZZARDI, Giancarlo; WAGNER, Gerd. A Unified Foundational Ontology and some Applications of it in Business Modeling. In: *CAiSE Workshops (3).* 2004, pp. 129–143.
5. ONTOUML COMMUNITY. *OntoUML community portal* [online]. 2018 [visited on 2021-05-04]. Available from: `ontouml.org`.
6. PERGL, Robert; SALES, Tiago Prince; RYBOLA, Zdeněk. Towards OntoUML for Software Engineering: From Domain Ontology to Implementation Model. In: CUZZOCREA, Alfredo; MAABOUT, Sofian (eds.). *Model and Data Engineering.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 249–263. ISBN 978-3-642-41366-7.
7. PHARO COMMUNITY. *Pharo* [online]. 2021. Version 8.0. Available also from: `https://pharo.org/download`.
8. PHARO COMMUNITY. *pharo.org* [online] [visited on 2021-05-04]. Available from: `https://pharo.org/`.

9. PHARO COMMUNITY. *Pharo wiki* [online] [visited on 2021-05-04]. Available from: `https://github.com/pharo-open-documentation/pharo-wiki`.
10. PHARO COMMUNITY. *PharoCheatSheet* [online]. [N.d.] [visited on 2021-05-04]. Available from: `http://files.pharo.org/media/pharoCheatSheet.pdf`.
11. UHNÁK, Peter; BLIZNIČENKO, Jan. *OpenPonk modeling platform* [online]. 2020 [visited on 2021-05-04]. Available from: `https://openponk.org/`.
12. CENTRE FOR CONCEPTUAL MODELLING AND IMPLEMENTATION. *OpenPonk* [online]. 2010. Version 1.0.0. Available also from: `https://openponk.org`.
13. UHNÁK, Peter; PERGL, Robert. The OpenPonk modeling platform. In: *IWST*. 2016, p. 14.
14. GUIZZARDI, Giancarlo. Ontological patterns, anti-patterns and pattern languages for next-generation conceptual modeling. In: *International Conference on Conceptual Modeling*. 2014, pp. 13–27.
15. GUIZZARDI, Giancarlo; SALES, Tiago Prince. Detection, Simulation and Elimination of Semantic Anti-patterns in Ontology-Driven Conceptual Models. In: YU, Eric; DOBBIE, Gillian; JARKE, Matthias; PURAO, Sandeep (eds.). *Conceptual Modeling*. Cham: Springer International Publishing, 2014, pp. 363–376. ISBN 978-3-319-12206-9.
16. SALES, Tiago Prince; BARCELOS, Pedro Paulo Favato; GUIZZARDI, Giancarlo. Identification of semantic anti-patterns in ontology-driven conceptual modeling via visual simulation. In: *4th International Workshop on Ontology-Driven Information Systems (ODISE 2012)*. 2012.
17. SALES, Tiago Prince. *Ontology Validation for Managers*. 2014. PhD thesis. Free University of Bozen-Bolzano.
18. PHARO COMMUNITY. *Spec2* [online]. 2021. Version 0.8.12. Available also from: `https://github.com/pharo-spec/Spec`.
19. BEN-KIKI, Oren; EVANS, Clark; NET, Ingy döt. *The Official YAML Web Site* [online]. 2009-10-01 [visited on 2021-05-04]. Available from: `https://yaml.org/`.
20. MOOSETECHNOLOGY. *PetitParser* [online]. 2010. Available also from: `https://github.com/moosetechnology/PetitParser`.

Appendix A

# Acronyms

| | |
|---|---|
| **IDE** | Integrated development environment |
| **OS** | Operating System |
| **ORM** | Object Role Modelling |
| **UI** | User interface |
| **BinOver** | Binary relation between overlapping types |
| **DecInt** | Deceiving intersection |
| **DepPhase** | Relationally dependent phase () |
| **FreeRole** | Free role specialization |
| **GSRigGSRig** | Generalization set with mixed rigidity |
| **HetColl** | Heterogeneous collective |
| **HomoFunc** | Homogeneous Functional Complex |
| **ImpAbs** | Imprecise abstraction |
| **MixIden** | Mixin with the same identity |
| **MixRig** | Mixin with the same rigidity |
| **MultDep** | Multiple relational dependency |
| **PartOver** | Part composing overlapping wholes |
| **RelComp** | Relation composition |
| **RelOver** | Relator mediating overlapping types |

| | |
|---|---|
| **RelRig** | Relator mediating rigid types |
| **RelSpec** | Relation specialization |
| **RepRel** | Repeatable relator instances |
| **UndefFormal** | Undefined formal association |
| **UndefPhase** | Undefined phase partition |
| **WholeOver** | Whole composed of overlapping parts |

Appendix **B**

# Contents of enclosed CD

- `readme.txt` ........ File describing structure of enclosed CD
- `EXE` ........ Directory with executables
  - `OpenPonk` ........ Directory with OpenPonk and verification framework
  - `Ontouml.org` ........ Directory with source files for ontouml.org
- `TEXT` ........ Directory containing this text and its source codes
  - `DP_Assignment.pdf` ........ Assignment of this thesis
  - `DP_Bělohoubek_Marek_2021.pdf` ........ Master's thesis in PDF format
  - `DP_Latex` ........ Directory with LaTeX source files for the thesis
    - `Figures` ........ Directory with figures used in this thesis