SUPPLEMENTARY MATERIAL FOR THE PAPER

MANAGING DATA MODEL EVOLUTION IN THE CONTEXT OF WEB APIS

Catalog of Round-Trip Migration Scenarios

October 22, 2019

1 Introduction

In this technical report we present a catalog of round-trip migration scenarios. The scenarios are discussed in terms of the changes that can be observed on the instance level. To approach the problem systematically, the examples are based on an existing catalog for object-oriented (meta)model evolution as published by Herrmannsdoerfer et al. [1].

For this catalog we assume the use of an N4IDL migration runtime. On a more technical side, we actually used it to implement the scenario catalog in code.
Section 2 and 3 introduce notational aspects and the term of a *round-trip migration scenario*. In 4, we present instructions on how to use and understand this catalog. Section 5 forms the main body of the catalog and discusses the various scenarios in detail. Finally, in section 6 we present a selection of our main learning outcomes from the work on the scenarios.

2 Notation

In the following we will use a shorthand notation to refer to a round-trip migration. A round-trip migration of a data model instance in version 1 via version 2 is denoted as $\#1 \mapsto \#2 \mapsto \#1$.

Furthermore, we will make use of object-graph diagrams to visualize the effects of a round-trip migration. An example of such a visualization is given in Fig. 2.1. Such round-trip migration diagrams can usually be structured into four consecutive stages: *Original Object Graph*, *Migrated Object Graph*, *Modified Migrated Object Graph* and *Round Trip Object Graph*. They correspond to the (up to) 4 stages of a round-trip migration (with modification). The stages are illustrated in the form of lighter shaded rectangles with corresponding labels. They contain object graphs which depict the data model instances between the application of the different migration and modification functions. In case of a round-trip scenario without modification, the stage *Modified Migrated Object Graph* is omitted.

The black, solid arrows relate the different stages to the application of migrations. The dashed, black arrows indicate a modification from one object graph to the other. The light-green, dashed arrows indicate the trace links that were constructed based on the executed set of migrations and their in- and outputs.

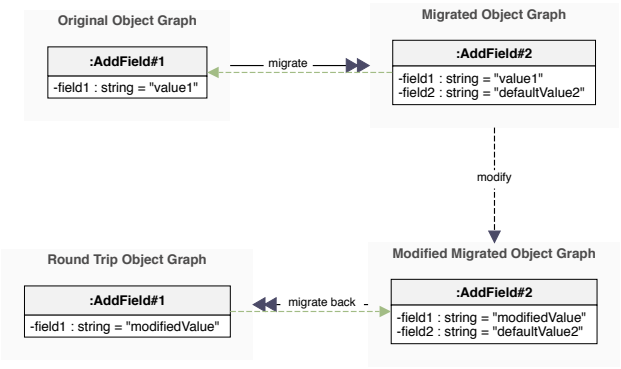


Figure 2.1: An example of a round-trip migration visualization.

Shortened migration listings

In some scenarios, the listings to illustrate the exemplary implementation of migrations is shortened for the sake of brevity. In these cases, the listings were reduced to demonstrate only the core ideas of a migration strategy. However, the full implementation can be found in the source code representation of the scenario catalog. All shortened listings are marked with an appropriate note.

3 Round-Trip Migration Scenarios

A round-trip migration scenario is defined by the following parameters:

- A data model difference D (e.g. *"add a field f to class C"*).
- Two versions of an exemplary data model M_1 and M_2 which differ in difference D.
- Implementations of migration functions f and g.
- An example instance $m_1 \in M_1$.

In many cases we will also consider a modification of the migrated instance. In these cases, a scenario is additionally specified by:

- A modification c_1 of the migrated instance $m_2 \in M_2$.
- A modification c_2 of the original instance $m_1 \in M_1$ which represents c_1 in terms of M_1 .

Given a model difference D, we may construct corresponding M_1 and M_2 as well as f and g. However, other parameters such as m_1, c_1 and c_2 need to be varied to cover all possible scenarios that stem from D. Furthermore, for each scenario we must consider both directions, resulting in the discussion of RTMs $M_1 \mapsto M_2 \mapsto M_1$ and $M_2 \mapsto M_1 \mapsto M_2$.

In this catalog, the general goal is to provide exemplary migrations f and g for each listed model difference D such that for any possible variation of m_1, c_1 and c_2, the migrations allow for a successful round-trip migration (with modification). The catalog is backed by an executable set of test suites, each representing one scenario. Such a suite makes assertions, assuring the above-mentioned properties of f and g when run as a test.

4 A Note on Completeness and How to Use This Catalog

When compiling a catalog round-trip scenarios, we may consider the concept of *completeness*. In other words, we ask the question whether *this catalog contains enough exemplary scenarios to compose a round-trip migration strategy for arbitrary model differences?* Simply put, the answer to this question is no. To be more precise, we must further define the term *completeness* in the context of round-trip migrations.

With regard to round-trip migrations, we understand the term *completeness* as follows: Given two data model versions, can the difference between the versions be decomposed into smaller (maybe even atomic) differences, for which fully implemented migration strategies can be found in the catalog? And further, can the migration strategies of those smaller differences be recomposed into a migration strategy that maintains correctness on a higher level? Authors of related fields (e.g. metamodel co-evolution) argue that such completeness is hard to achieve since higher level changes often entail migration strategies that differ from the simple concatenation of the strategies that fit their atomic constituents [1][2]. A simple example of this, is the model change of *renaming a field* (cf. scenario 1 Rename Field) as opposed to the consecutive *deletion of the old* and *creation of a new* field (cf. scenario 2). Based on this insight, we do not claim the completeness of our scenario catalog and assume a different standpoint.

Rather than completeness, we aim for a more practical use of the catalog. It was designed to serve as a knowledge base and guide for the implementation of round-trip migrations. While some scenarios may be applicable exactly as they are presented in this catalog, others may be more suitable as a demonstration of abstract aspects. As we will see later in our case study, in practice, it often helps to first consider related round-trip scenarios before implementing a concrete migration. Furthermore, we see our catalog as an initial listing of elementary data model changes, which in the future can be extended by additional and more complex scenarios (e.g. discovery of new scenarios during use, introduction of new modelling features in N4IDL, etc.).

5 Catalog

In this catalog, we will discuss the following list of scenarios in detail.

No.	Name	Description	trace links	modification detection	Page
1	Rename Field	The name of a field changes.			4
2	Create/Delete Field (functionally independent field)	A new functionally independent field is added/removed to/from a class of the data model.	✓		4
3	Create/Delete Field (functionally dependent field)	A field is removed from a class of the data model (or added respectively). The field is functionally dependent on other still-existing fields.	✓		5
4	Create/Delete Reference	A field of reference type is removed from a class of the data model (or added respectively).	✓		7
5	Declare Class as Abstract	A class is declared abstract/concrete.	✓		8
6	Add/Remove a Supertype	A new supertype is declared for a classifier (or removed respectively).	✓		10
7	Generalize/Specialize Field Type	The type of a field is generalized to a supertype or specialized to a subtype respectively.	✓		11
8	Change Field Multiplicity: 0..1 - 1	The multiplicity of a field is specialized from multiplicity `0..1` to `1` or generalized from `1` to `0..1` respectively.	✓	✓	13
9	Change Field Multiplicity: 0..n - 0..1	The multiplicity of a field is generalized from 0..1 to 0..n or specialized from 0..n to 0..1.	✓	✓	14
10	Change Field Multiplicity: 0..n - 1	The multiplicity of a field is specialized from multiplicity `0..n` to `1` or generalized from `1` to `0..n` respectively.	✓	✓	16
11	Pull Up / Push Down Field	A field is pulled up into a superclass (or pushed down respectively).			18
12	Split/Merge Type	Based on a specified criteria, instances of a type of one model version, translate to different (unrelated) types of the other model version.	✓		19
13	Specialize/Generalize Superclass	The supertype of a class is changed to one of the supertype's subclasses/superclasses.	✓		21
14	Extract/Inline Superclass	A new superclass is extracted from the set of fields of an existing type.	✓	✓	23
15	Fold/Unfold Superclass	A new superclass is declared for a type. Common fields of the superclass and the type are then removed from the type (folded into superclass).			24
16	Extract/Inline Subclass	A selection of fields is extracted into a new subclass (or inlined respectively).	✓	✓	25
17	Extract/Inline Class	A selection of fields is extracted into a new delegate class.			27
18	Fold/Unfold Class	A selection of fields is folded into an existing delegate class.			29
19	Collect Field over Reference	A field is collected/pushed over a reference.	✓	✓	30
20	Split/Merge Fields	A type is split by moving its fields to two new types and correspondingly replacing all references to it by references to the new types.			33

Scenario 1: Rename Field

The name of a field changes.

In the example model, the field `field1` is renamed to `field2`.

Data Models

```
export public class RenameField#1 {
        field1 : string
}
```

Version 1

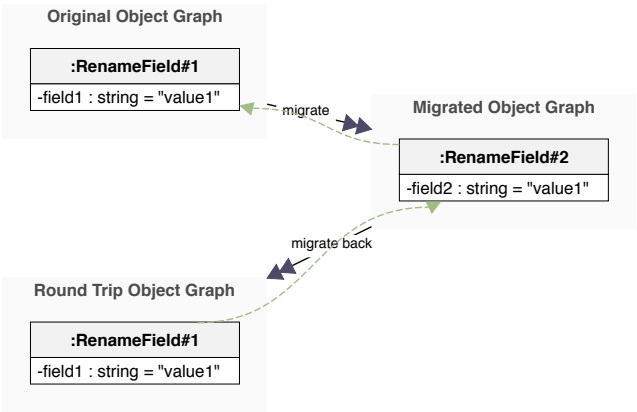
```
export public class RenameField#2 {
        field2 : string
}
```

Version 2

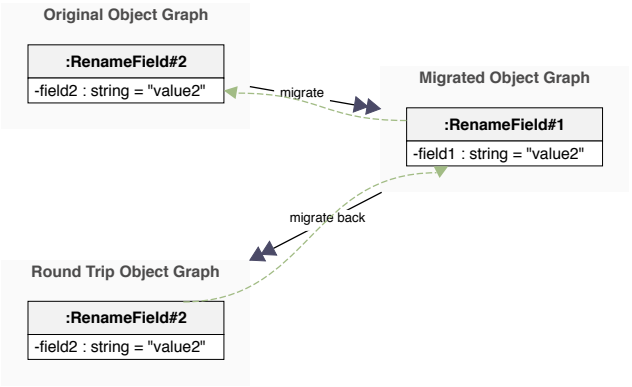
Discussion Since the the data model semantics of both versions are equal except for names, a round-trip migration can be performed without using any traceability features. Furthermore, any modification of `field1` will be reflected by `field2` and vice-versa.

Migrations

```
@Migration
export function migrate(o1 : RenameField#1) : RenameField#2 {
        let o2 = new RenameField#2();

        // renamed field
        o2.field2 = o1.field1;

        return o2;
}

@Migration
export function migrateBack(o2 : RenameField#2) : RenameField#1 {
        let o1 = new RenameField#1();

        // undo field renaming
        o1.field1 = o2.field2;

        return o1;
}
```



Round-Trip 1.1: $\#1 \mapsto \#2 \mapsto \#1$



Round-Trip 1.2: $\#2 \mapsto \#1 \mapsto \#2$

Scenario 2: Create/Delete Field (functionally independent field)

A new functionally independent field is added/removed to/from a class of the data model.

In this scenario we assume, that there do not exist any functional dependencies between `field1` and `field2` in version 2 of the data model.

Data Models

```
export public class AddField#1 {
        public field1 : string
}
```

Version 1

```
export public class AddField#2 {
        public field1 : string
        public field2 : string
}
```

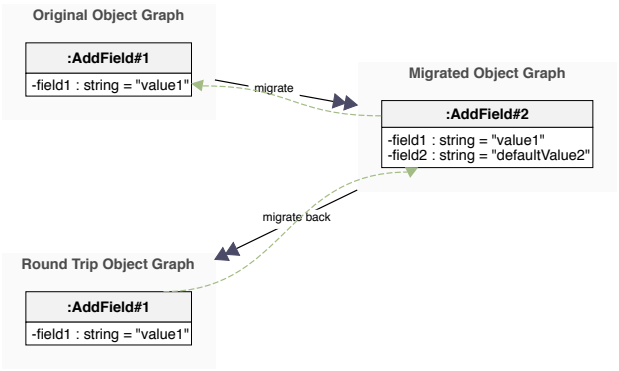
Version 2

Discussion In a $\#1 \mapsto \#2 \mapsto \#1$ RTM, `field2` is set to a default value since the original instance does not provide a value for this field on its own.

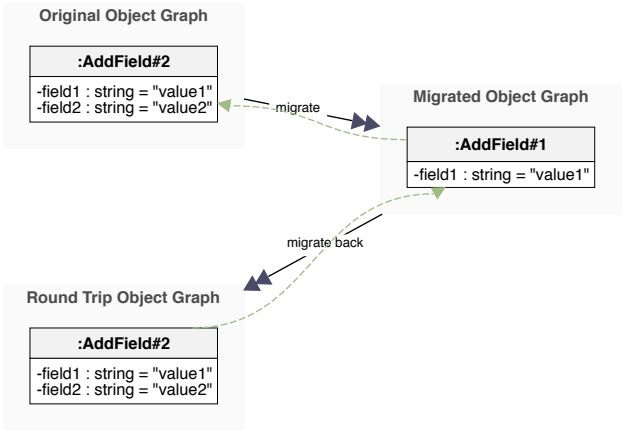
In a $\#2 \mapsto \#1 \mapsto \#2$ RTM, the migration leverages the support for trace links (cf. green links in the RTM graphs), and recovers the value of `field2` from the original instance. A simple copying of the previous value of `field2` is feasible in this scenario, since we assume a functional independence of `field1` and `field2`. Therefore, a potential modification of `field1` in version 2 cannot have any effect on `field2`.

Migrations

```
@Migration
export public function migrate(o1 : AddField#1) : AddField#2 {
        // obtain previous revision
        const previousRevision = context.getTrace(o1)[0] as AddField#2 ||Â {} as AddField#2;

        // instantiate an empty instance
        let o2 = new AddField#2();

        // transfer value for field 'field1'
        o2.field1 = o1.field1;
        // use previous value or a default value alternatively
        o2.field2 = previousRevision.field2 || "defaultValue2";

        return o2;
}

@Migration
export public function migrateBack(o2 : AddField#2) : AddField#1 {
        let o1 = new AddField#1();
        // transfer field value for 'field1'
        o1.field1 = o2.field1;

        return o1;
}
```



Round-Trip 2.1: $\#1 \mapsto \#2 \mapsto \#1$



Round-Trip 2.2: $\#1 \mapsto \#2 \mapsto \#1$

Scenario 3: Create/Delete Field (functionally dependent field)

A field is removed from a class of the data model (or added respectively). The field is functionally dependent on other still-existing fields.

In the example data model, version 1 declares a field `hereToStayTwice`. By an informal data invariant, it is specified to always hold a value that equals the concatenation of field `hereToStay` with itself. This implies a functional dependency between the fields. In version 2 of the same class, the field `hereToStayTwice` is removed.

Data Models

```
export public class CreateDeleteDependentField#1 {
        hereToStay : string
        /**
         * The value of this field should always be
         * 'hereToStay', but concatenated with
         * itself twice (e.g. 'A' -> 'A A').
         */
        hereToStayTwice : string
}
```

Version 1

```
export public class CreateDeleteDependentField#2 {
        hereToStay : string
}
```

Version 2

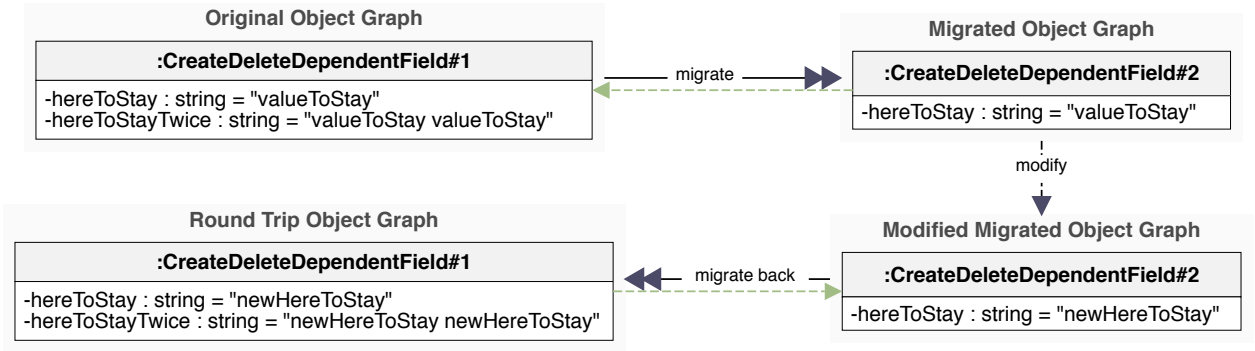
Discussion The migration from version 2 to 1 must consider the potentially new value of field `hereToStay` to compute the corresponding new value of `hereToStayTwice`. Although traceability would allow to access the previous value of `hereToStayTwice`, the migration cannot just copy it, since the functional dependency between `hereToStay` and `hereToStayTwice` may be violated.

For a $\#2 \mapsto \#1 \mapsto \#2$ RTM, the migration computes the missing value of field `hereToStayTwice` from field `hereToStay` in the original instance of version 2. Although `hereToStayTwice` is modified, this modification is not mapped back to the value of `hereToStay` in version 1.

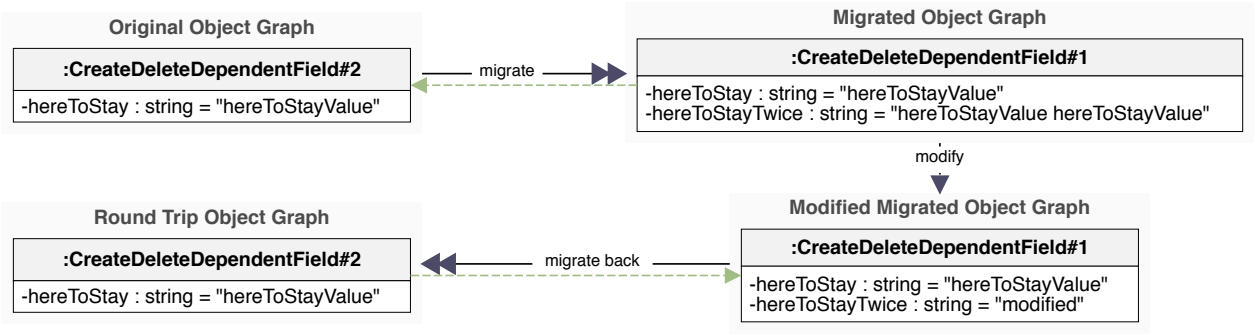
While intuitively this may not seem like a successful round-trip migration, it formally is one according to our initial definition. Nonetheless, the given example demonstrates an issue regarding functional dependencies between fields: A modification of only one of the instances of a redundant bit of information (e.g. the value of field `hereToStayTwice`) causes an inconsistent state, which may in turn result in unexpected behavior. However, this is not an issue with round-trip migrations specifically, but rather with data modeling in general.

Migrations

```
@Migration
export function migrate(o1 : CreateDeleteDependentField#1) : CreateDeleteDependentField#2 {
        let o2 = new CreateDeleteDependentField#2();

        // copy value for field 'hereToStay'
        o2.hereToStay = o1.hereToStay;

        return o2;
}

@Migration
export function migrateBack(o2 : CreateDeleteDependentField#2) : CreateDeleteDependentField#1 {
        let o1 = new CreateDeleteDependentField#1();

        o1.hereToStay = o2.hereToStay
        o1.hereToStayTwice = o2.hereToStay + "␣" + o2.hereToStay

        return o1;
}
```



Round-Trip 3.1: $\#1 \mapsto \#2 \mapsto \#1$ In version 2, the value of field `valueToStay` is modified.



Round-Trip 3.2: $\#2 \mapsto \#1 \mapsto \#2$ In version 1, the value of `hereToStayTwice` is modified.

Scenario 4: Create/Delete Reference

A field of reference type is removed from a class of the data model (or added respectively).

In the example data models, the reference to an instance of type `ReferencedElement` is removed in version 2.

Data Models

```
1  export public class CreateDeleteReference#1 {
2          reference : ReferencedElement
3
4          unrelated : string
5  }
6
7  export public class ReferencedElement#1 {
8          public value : string
9          constructor(@Spec spec : ~i~this) {}
10 }
```

Version 1

```
1  export public class CreateDeleteReference#2 {
2          unrelated : string
3  }
4
5
6  export public class ReferencedElement#2 {
7          public value : string
8          constructor(@Spec spec : ~i~this) {}
9  }
```

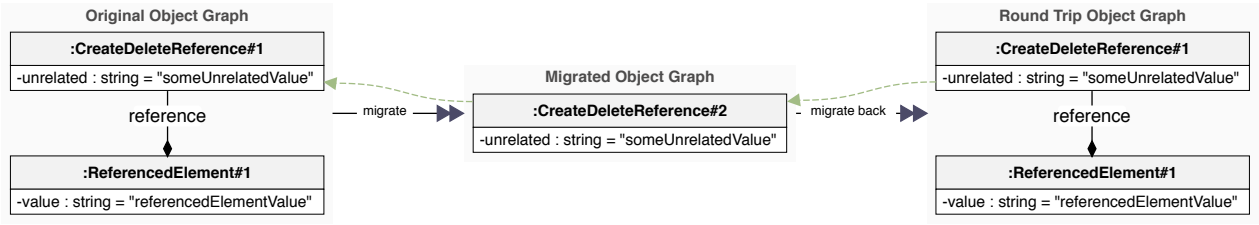
Version 2

Discussion This scenario exhibits similar properties to those of its counterpart for primitively typed fields in scenario 2 Create/Delete Field (functionally independent field) and 3 Create/Delete Field (functionally dependent field). It only differs in that the value which is recovered via trace links, is not of primitive type but an instance of type `ReferencedElement`.

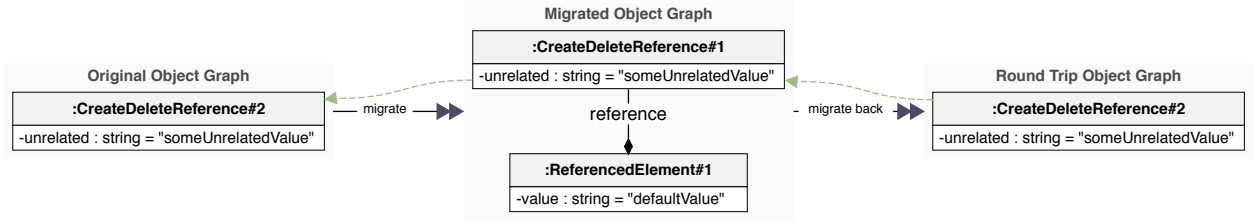
Analogously, this scenario poses the same challenges with regard to potential functional dependencies between different references (cf. discussions in scenario 2 and 3).

Migrations

```
@Migration
export function migrate(o1 : CreateDeleteReference#1) : CreateDeleteReference#2 {
        let o2 = new CreateDeleteReference#2();

        o2.unrelated = o1.unrelated;

        return o2;
}

@Migration
export function migrateBack(o2 : CreateDeleteReference#2) : CreateDeleteReference#1 {
        // obtain previous revision
         const previousRevision = context.getTrace(o2)[0] as CreateDeleteReference#1
                || {} as CreateDeleteReference#1;

        let o1 = new CreateDeleteReference#1();

        o1.unrelated = o2.unrelated;
        // choose a default value for 'reference' or recover previous value
        o1.reference = previousRevision.reference
                || new ReferencedElement#1({value: "defaultValue"});

        return o1;
}
```



Round-Trip 4.1: $\#1 \mapsto \#2 \mapsto \#1$



Round-Trip 4.2: $\#2 \mapsto \#1 \mapsto \#2$

Scenario 5: Declare Class as Abstract

A class is declared abstract/concrete.

In the exemplary data model of this scenario, the class `Value#1` is declared abstract in model version 2. The concrete subclass `SubValue` exists in both versions and is not changed from version 1 to 2.

Data Models

```
export public class MakeClassAbstract#1 {
        public field : Value
}

export public class Value#1 {
        public commonField : string
}

export public class SubValue#1 extends Value {
        public field1 : string
}
```

Version 1

```
export public class MakeClassAbstract#2 {
        public field : Value
}

export public abstract class Value#2 {
        public commonField : string
}

export public class SubValue#2 extends Value {
        public field1 : string
}
```

Version 2

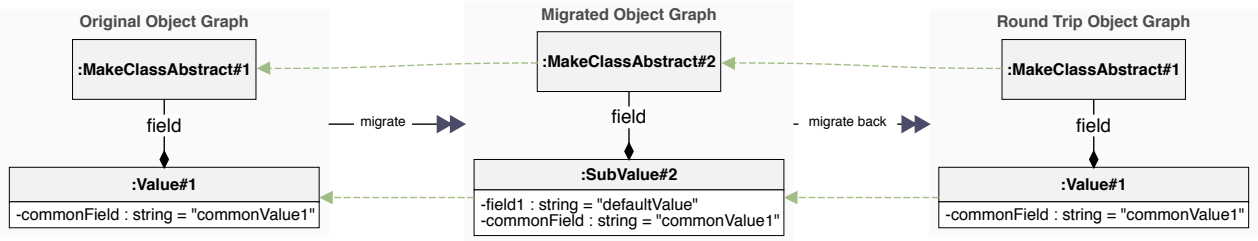
Discussion The core challenge of this scenario lies in the successful migration of instances of `Value#1` to one of its concrete subclasses in version 2. Instances of `SubValue` can be represented in both versions by the identical and correspondingly named types and are thus not of much interest. Concerning the migration of `Value#1` instances however, there are three main points that need to be addressed:

1. A migration must choose a concrete subclass in version 2, which `Value#1` instances are migrated to. Furthermore, default values must be chosen to compensate for missing information. In this example, the migration statically assumes that `Value#1` instances are represented as `SubValue#2` instances in version 2. Assuming there are multiple concrete subclasses to choose from however, it is also possible to make this choice at runtime (e.g. based on a designated field which indicates which type of version 2 to migrate to).
2. In order to successfully round-trip migrate `Value#1` instances, migrations must ensure that `SubValue1#2` instances that originate from a `Value#1` instance, maintain their original type in a round-trip. For instance, a `SubValue#2` instance must always be migrated back to a `Value#1` instance, if it was originally represented as such in version 1 (cf. Round-Trip 5.1). However, this case must be differentiable from `SubValue#2` instances that actually do stem from `SubValue#1` instances. By leveraging the support for traceability, this can be implemented by a runtime type check on the previous revision instance (cf. `migrateBackValue` in the migrations listing).

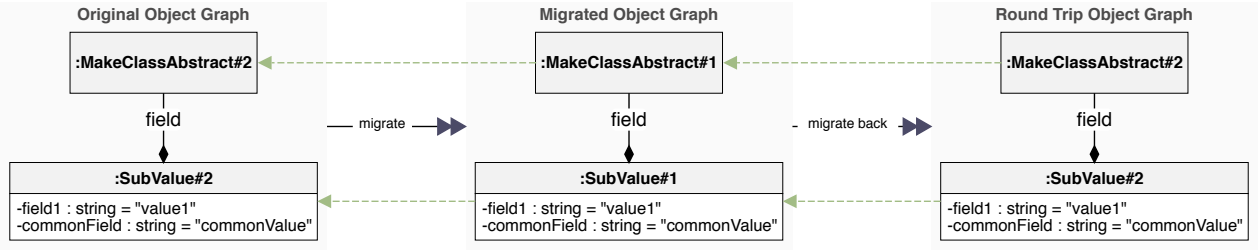
Migrations

```
@Migration
export function migrateValue(o1 : Value#1) : Value#2 {
        const previousRevision = context.getTrace(o1)[0] || {};

        const sv = new SubValue#2();
        // obtain value of field1 from previous revision or choose a default
        sv.field1 = (previousRevision as SubValue#2).field1 || "defaultValue";
        sv.commonField = o1.commonField;
        return sv;
}

@Migration
export function migrateBackValue(o2 : SubValue#2) : Value#1 {
        const previousRevision = context.getTrace(o2)[0] as Value#1
                || {} as Value#1;

        // If in a previous migration we were forced to migrate
        // from Value to SubValue, and 'field1' has not been modified,
        // migrate-back to Value
        if (context.getTrace(o2).length > 0
                && !(previousRevision instanceof SubValue#1)
                && !(context.isModified(o2, "field1"))) {
                const v = new Value#1();
                v.commonField = o2.commonField;
                return v;
        }

        // otherwise just migrate SubValue#2 to an instance of identical type SubValue#1
        const sv = new SubValue#1();
        sv.commonField = o2.commonField;
        sv.field1 = o2.field1;

        return sv;
}
```

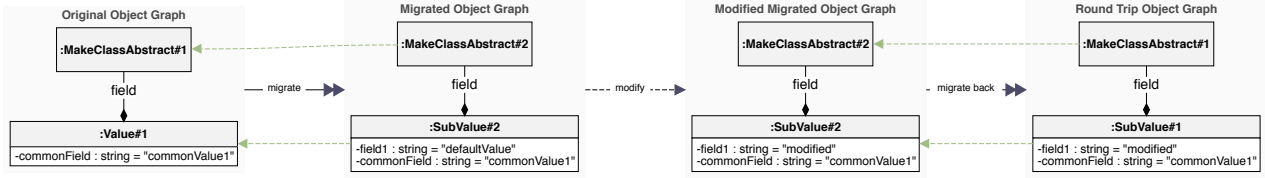
Shortened: For the full implementation of this scenario see the appended source code of the catalog.



Round-Trip 5.1: $\#1 \mapsto \#2 \mapsto \#1$ Instances of type `Value#1` are migrated to `SubValue#2` by choosing a default value for `field1`.



Round-Trip 5.2: $\#2 \mapsto \#1 \mapsto \#2$ An instance of `SubValue#2` is migrated via `SubValue#1` without any modifications in between.



Round-Trip 5.3: $\#1 \mapsto \#2 \mapsto \#1$ The field `field1` is modified in model version 2. As a consequence its type in the original version is changed to `SubValue#1` in order to represent the modification.

Scenario 6: Add/Remove a Supertype

A new supertype is declared for a classifier (or removed respectively).

In version 2 of the example data model, the class `AddSuperType#1` gains the supertype `SuperType#2`.

Data Models

```
export public class AddSuperType#1 {
        public ownedField : string
}
```

Version 1

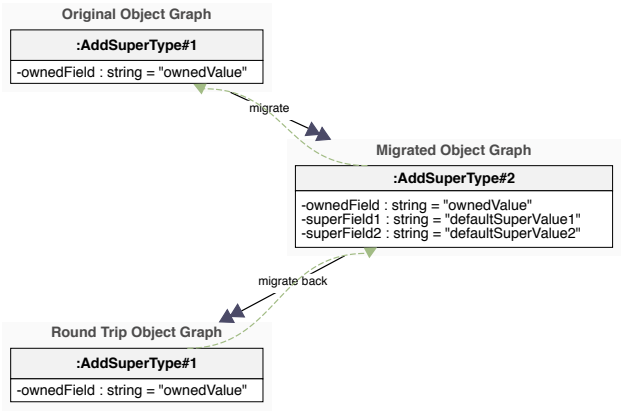
```
export public class AddSuperType#2
        extends SuperType {

        public ownedField : string
}

export public class SuperType#2 {
        public superField1 : string
        public superField2 : string
}
```

Version 2

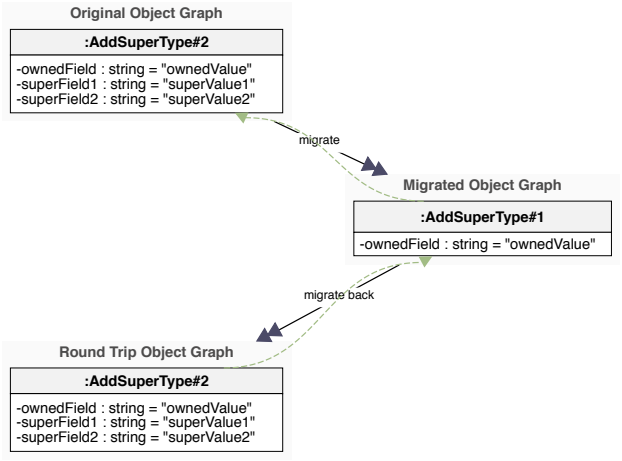
Discussion In version 2 of the data model, instances of type `AddSuperType#2` gain the fields `superField1` and `superField2` which are consumed from the new supertype `SuperType#2`. Therefore, this scenario exhibits the same properties as scenario 2 (Create/Delete Field) for each of the consumed fields. If, in the context of a concrete data model, the consumed fields maintain functional dependencies on existing fields in `AddSuperType#2`, the discussion of scenario 3 may also apply.

Migrations

```
@Migration
export function migrate(o1 : AddSuperType#1) : AddSuperType#2 {
        // obtain previous revision
        const previousRevision = context.getTrace(o1)[0] as AddSuperType#2
                || {} as AddSuperType#2;

        let o2 = new AddSuperType#2();

        o2.ownedField = o1.ownedField;
        o2.superField1 = previousRevision.superField1 || "defaultSuperValue1";
        o2.superField2 = previousRevision.superField2 || "defaultSuperValue2";

        return o2;
}

@Migration
export function migrateBack(o2 : AddSuperType#2) : AddSuperType#1 {
        let o1 = new AddSuperType#1();

        o1.ownedField = o2.ownedField;

        return o1;
}
```



Round-Trip 6.1: $\#1 \mapsto \#2 \mapsto \#1$ The migration chooses default values for the newly introduced fields `superField1` and `superField2` in version 2.



Round-Trip 6.2: $\#1 \mapsto \#2 \mapsto \#1$ Using trace links, the the original values of the new field `superField1` and `superField2` can be restored.

Scenario 7: Generalize/Specialize Field Type

The type of a field is generalized to a supertype or specialized to a subtype respectively.

In the example model, the type of field `field` is generalized from `FieldType` to the supertype `SuperFieldType`. The types `FieldType` and `SuperFieldType` remain unchanged from version 1 to version 2.

Data Models

```
export public class GeneralizeFieldType#1 {
        public field : FieldType
}

export public class SuperFieldType#1 {
        public generic : string
}

export public class FieldType#1
        extends SuperFieldType {

        public specific : string
}
```

Version 1

```
export public class GeneralizeFieldType#2 {
        public field : SuperFieldType
}

export public class SuperFieldType#2 {
        public generic : string
}

export public class FieldType#2
        extends SuperFieldType {

        public specific : string
}
```

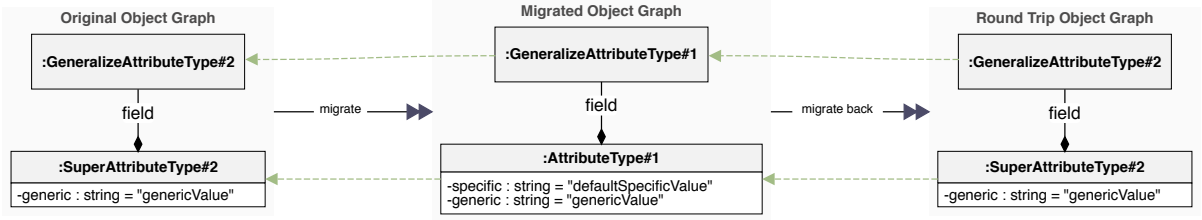
Version 2

Discussion This scenario exhibits similar properties to those of scenario 5 Declare Class as Abstract. This is due to the fact that this scenario requires the migration of an instance of more specific type to an instance of less specific type (cf. `FieldType` vs. `SuperFieldType`). The migrations need to make use of traceability information to differentiate between the different cases of `FieldType` instances in version 1 (cf. Fig. 7.1 vs Fig. 7.2). These are those that originally stem from a `FieldType` and those migrated from the supertype. Furthermore, a modification of field `specific` may further affect the migration strategy (cf. Fig. 7.3). See scenario 5 for a thorough discussion.

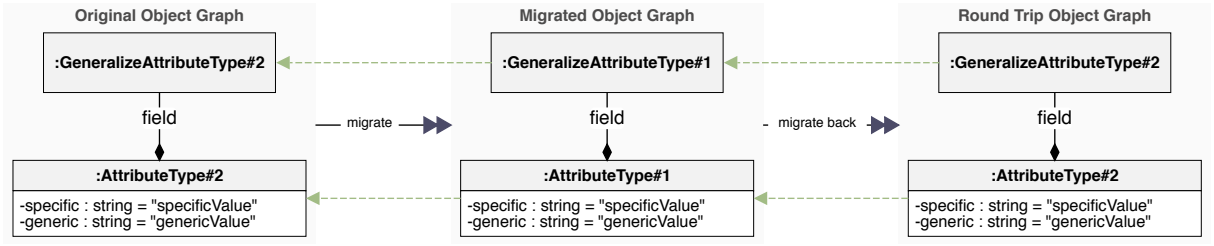
Migrations

```
@Migration
export function migrateBackSuperAttribute(a2 : SuperFieldType#2) : FieldType#1 {
        // obtain previous revision
        const previousRevision = context.getTrace(a2)[0] as FieldType#1 || {} as FieldType#1;

        const a1 = new FieldType#1();
        a1.generic = a2.generic;
        a1.specific = previousRevision.specific ||Â "defaultSpecificValue";

        return a1;
}

@Migration
export function migrateAttribute(a1 : FieldType#1) : SuperFieldType#2 {
        const previousRevision = (context.getTrace(a1)[0] || {}) as SuperFieldType#2;

        let a2 : SuperFieldType#2

        // If field has been of type SuperAttributeType but not
        // of AttributeType, and field 'specific' has not been changed
        if (previousRevision instanceof SuperFieldType#2
                && !(previousRevision instanceof FieldType#2)
                && !(context.isModified(a1, "specific"))) {
                // migrate back to SuperAttribute type
                a2 = new SuperFieldType#2();
        } else { // no previous revision, or a previous revision of type AttributeType
                a2 = new FieldType#2();
                (a2 as FieldType#2).specific = a1.specific;
        }

        // set generic field in any case
        a2.generic = a1.generic;

        return a2;
}
```

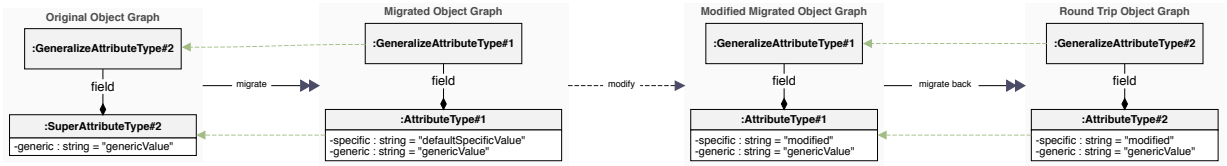
Shortened: For the full implementation of this scenario see the appended source code of the catalog.



Round-Trip 7.1: $\#2 \mapsto \#1 \mapsto \#2$ with initial field type `SuperFieldType`.



Round-Trip 7.2: $\#2 \mapsto \#1 \mapsto \#2$ with initial field type `FieldType`.



Round-Trip 7.3: $\#2 \mapsto \#1 \mapsto \#2$ with a modification of field `specific`.

Scenario 8: Change Field Multiplicity: 0..1 - 1

The multiplicity of a field is specialized from multiplicity `0..1` to `1` or generalized from `1` to `0..1` respectively.

In the example data model, the mandator field `field` of model version 1 is declared optional in version 2.

Data Models

```
export public class GeneralizeAttributeOptional#1 {
        public field : string
}
```

Version 1

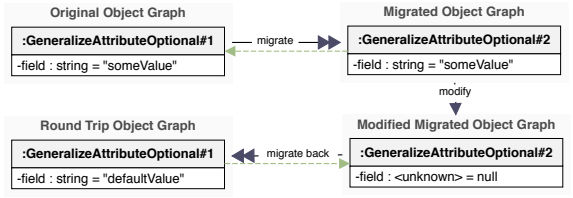
```
export public class GeneralizeAttributeOptional#2 {
        public field? : string
}
```

Version 2

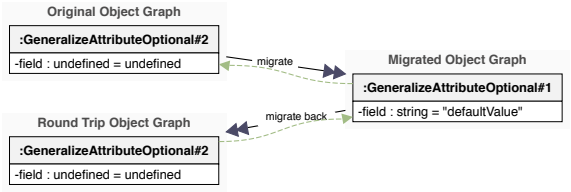
Discussion Model version 2 is more generic, as it allows for additional instances that hold a `null`-value for field `field`. To accommodate for this difference, a migration must choose a default value for `field` when migrating a `null`-value from model version 2 to 1. In a $\#2 \mapsto \#1 \mapsto \#2$ round-trip, it must further be assured, that a default value is not translated back into version 2 but rather detected and therefore mapped to the original `null`-value (cf. Round-Trip 8.1).

Migrations

```
@Migration
export function migrate(o1 : GeneralizeAttributeOptional#1) : GeneralizeAttributeOptional#2 {
        let o2 = new GeneralizeAttributeOptional#2();

        const previousRevision = context.getTrace(o1)[0] as GeneralizeAttributeOptional#2

        // If 'field' has not been modified, and a previous revision could
        // be obtained
        if (previousRevision != undefined
                && !context.isModified(o1, "field")) {
                // re-use the previous value for 'field'
                o2.field = previousRevision.field;
        } else {
                // otherwise copy over the value in 'field'
                o2.field = o1.field;
        }

        return o2;
}

@Migration
export function migrateBack(o2 : GeneralizeAttributeOptional#2) : GeneralizeAttributeOptional#1 {
        let o1 = new GeneralizeAttributeOptional#1();

        // use default value in case 'o2.field' is null
        o1.field = o2.field || "defaultValue";

        return o1;
}
```



Round-Trip 8.1: $\#1 \mapsto \#2 \mapsto \#1$ A modification in version 2 sets `field` to `null`. This results in a default value in version 1.



Round-Trip 8.2: $\#2 \mapsto \#1 \mapsto \#2$ In version 1, `field` is set to a default value, which is later not translated back to model version 2.

Scenario 9: Change Field Multiplicity: 0..n - 0..1

The multiplicity of a field is generalized from 0..1 to 0..n or specialized from 0..n to 0..1.

In the exemplary data model, the type of field `field` is changed from an optional reference to an instance of type `Element` to an array with element type `Element`.

Data Models

```
1 export public class GeneralizeAttributeOptional2Array#1 {
2         public field? : Element
3 }
4 export public class Element#1 {
5         public value : string
6 }
```

Version 1

```
1 export public class GeneralizeAttributeOptional2Array#2 {
2         public field : Array<Element>
3 }
4 export public class Element#2 {
5         public value : string
6 }
```

Version 2

Discussion The general migration strategy we propose for this scenario, is to introduce a mapping of the value of `field` to a specific index in the array in model version 2. For the sake of simplicity, we will assume for further discussion that this designated array index is always the first element (0). However, it may of course be any other index or even a dynamically chosen index.

For this scenario, there are two main observations to be made:

1. The migrations need to translate any change to the designated index in the array of model version 2 to version 1 and vice-versa. Since an array may be mutated using common operators such as insert and delete, we must consider the situation in which the array does not hold an element for that specific index. Therefore, this scenario can be seen as an instance of scenario 8 where we migrate between a mandatory and an optional field. This migration from version 2 to 1 can be implemented using the following mappings:

`field` in version 2	**`field` in version 1**
Empty Array	`field` is set to null.
Array without an element at the designated index.	`field` is set to null.
Array with an instance of `Element#2` at the designated index.	`field` is set to the migrated equivalent of the element at the designated index.

   Note that with this strategy we bind the value of `field` in version 1 to an index in the array of version 2, not a specific element. For instance, when a new element is inserted at index 0 of the array in version 2, this new element will replace the current value of `field` in version 1 (cf. Round-Trip 9.1). Another instance of this behavior is demonstrated in Round-Trip 9.4.

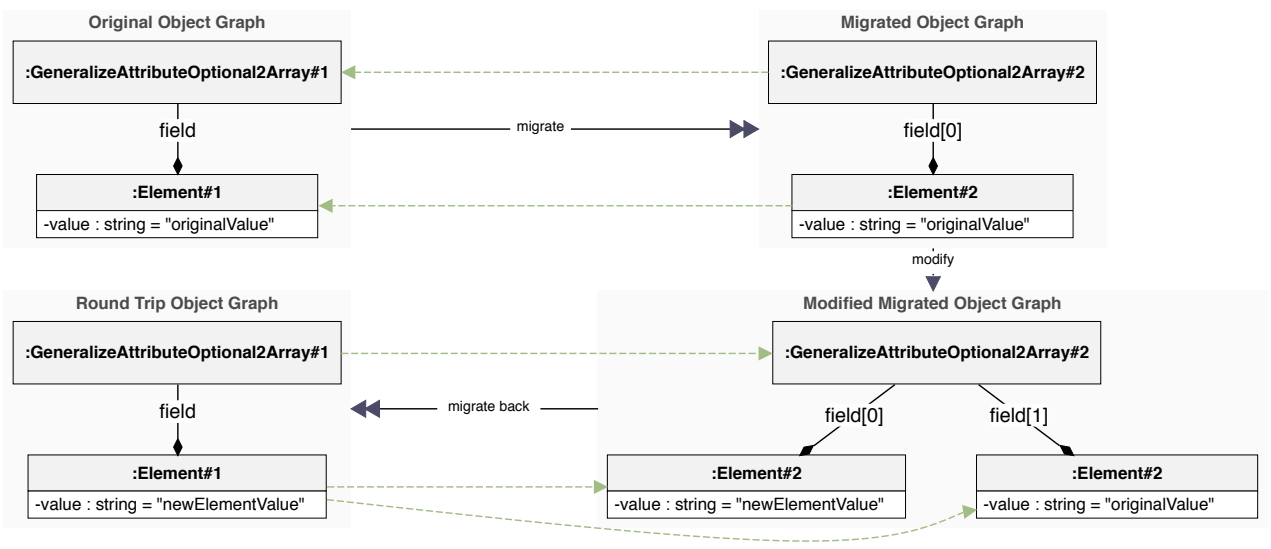
2. When round-trip migrating an instance of version 2 via version 1, a migration needs to update the designated array element with the potentially changed value of `field` from version 1. Apart from that, it needs to *restore all other array elements using trace links* (see line 17-19 in `migrate`).

The Round-Trips 9.2, 9.3 and 9.4 demonstrate how this migration strategy maps common modifications (set null, delete element, insert element) of `field` in version 1 and 2 to the other model version.

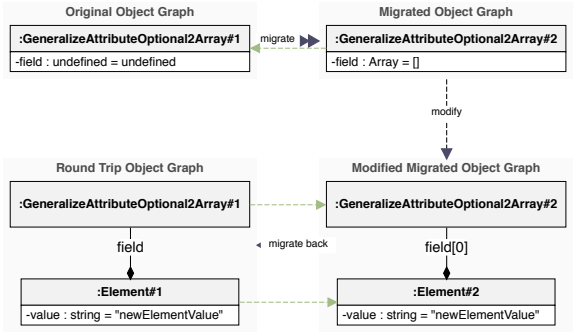
Migrations

```
@Migration
export function migrate(o1 : GeneralizeAttributeOptional2Array#1) : GeneralizeAttributeOptional2Array#2 {
        // obtain previous revision
        const previousRevision = context.getTrace(o1)[0] as GeneralizeAttributeOptional2Array#2
                || {} as GeneralizeAttributeOptional2Array#2;

        let o2 = new GeneralizeAttributeOptional2Array#2();

        let elements : Array<Element#2> = [];

        // if o1.field is present
        if (o1.field != null) {
                // add migrated field value as first array element
                elements.push(Migrations.copy(Element#2, o1.field));
        }

        // add all previousRevision elements, but the first one
        (previousRevision.field || []).slice(1)
                .forEach(e => elements.push(Migrations.copy(Element#2, e)));

        o2.field = elements;

        return o2;
}

@Migration
export function migrateBack(o2 : GeneralizeAttributeOptional2Array#2) : GeneralizeAttributeOptional2Array#1 {
        let o1 = new GeneralizeAttributeOptional2Array#1();

        // migrate the element at index 0, if not present set 'o1.field' to null
        o1.field = Migrations.migrateElementAt(o2.field, 0, null);

        return o1;
}
```

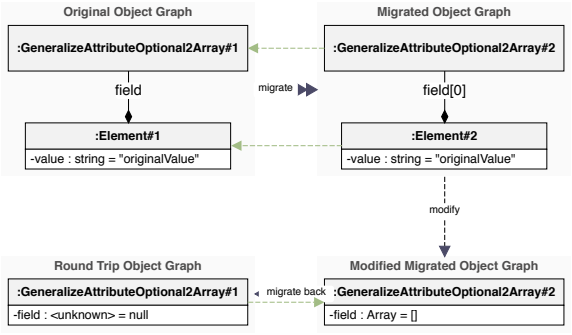
Shortened: For the full implementation of this scenario see the appended source code of the catalog.



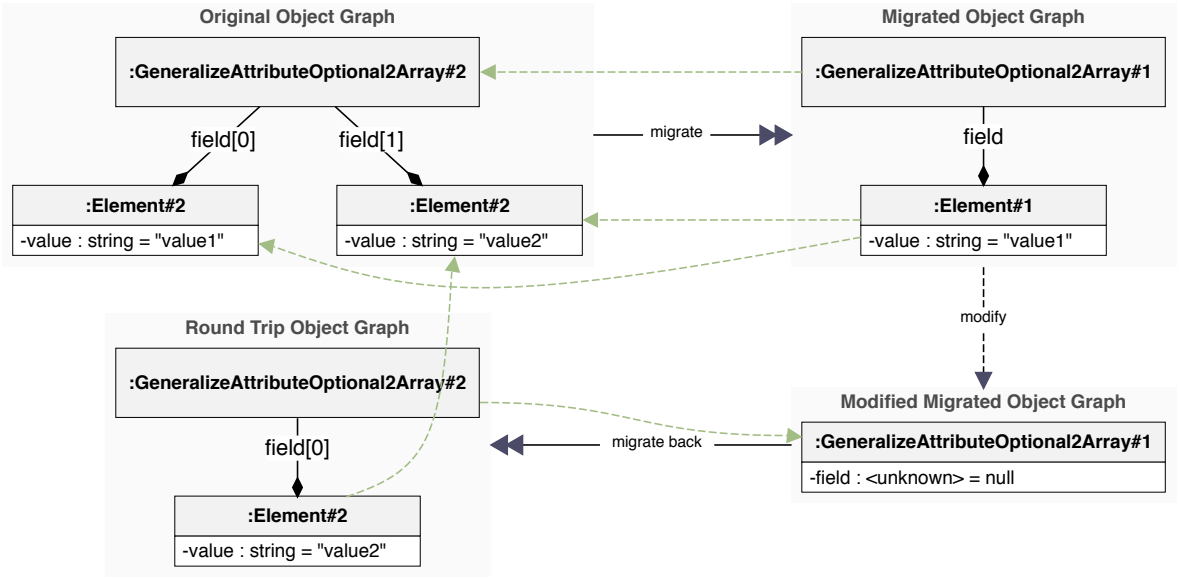
Round-Trip 9.1: $\#1 \mapsto \#2 \mapsto \#1$ In version 2, a new element is inserted into the array at index 0. As a consequence, the new element of the array replaces the value of `field` in version 1.



Round-Trip 9.2: $\#2 \mapsto \#1 \mapsto \#2$ An absent value of `field` in version 1 translates to an empty array in version 2. Adding a new element to the array in version 2, results in setting the value of `field` in version 1.



Round-Trip 9.3: $\#2 \mapsto \#1 \mapsto \#2$ Removing all elements from the array of version 2, translates to setting `field` in version 1 to null.



Round-Trip 9.4: $\#2 \mapsto \#1 \mapsto \#2$ Setting `field` to null in version 1, results in removing the corresponding element from the array of version 2.

Scenario 10: Change Field Multiplicity: 0..n - 1

The multiplicity of a field is specialized from multiplicity `0..n` to `1` or generalized from `1` to `0..n` respectively.

In our example data model, the type of field `field` is changed from `Element` to `Array<Element>`.

Data Models

```
1 export public class GeneralizeAttributeArray#1 {
2         public field : Element
3 }
4 export public class Element#1 {
5         public value : string
6 }
```

Version 1

```
1 export public class GeneralizeAttributeArray#2 {
2         public field : Array<Element>
3 }
4 export public class Element#2 {
5         public value : string
6 }
```

Version 2

Discussion This scenario exhibits similar characteristics to those of scenario 9. Therefore, we propose an analogous migration strategy of introducing a mapping of the value of `field` in version 1 to a designated index in the corresponding array in model version 2.

As opposed to scenario 9 however, model version 1 does not allow to set `field` to null when the corresponding array index does not hold a value at migration time. Instead, we propose the use of default values. The following mapping between states of `field` in version 1 and 2 may then be used:

`field` in version 2	**`field` in version 1**
Empty Array	Default Value for `Element#1`
Array without an element at the designated index.	Default Value for `Element#1`
Array with an instance of `Element#2` at the designated index.	Migrated version of the instance.

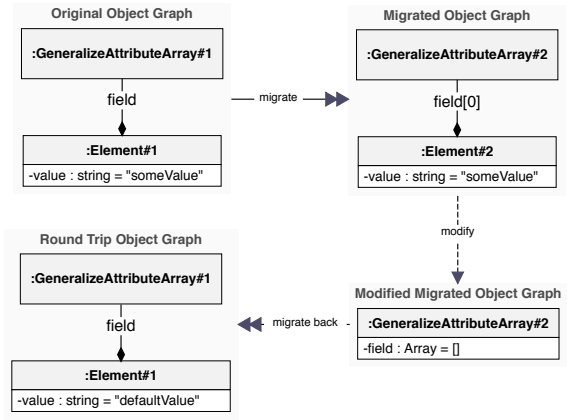
Similar to other scenarios in which we resort to default values, we need to make sure that a default instance for `field` is not translated back into version 2. Instead, we want to assure that we restore the original value of `field` using trace links. See line 11 to 14 in `migrate` for an exemplary implementation. Round-Trip 10.2 demonstrates such a situation.

Based on this migration strategy, we inherit a similar behavior as in scenario 9 (cf. Round-Trip 10.1 and 10.3). Overall however, we rate this scenario to be slightly more complex. Since in version 1 the field is mandatory, we rely on default values which in turn require the use of modification detection. This significantly increases the complexity of the migration code.

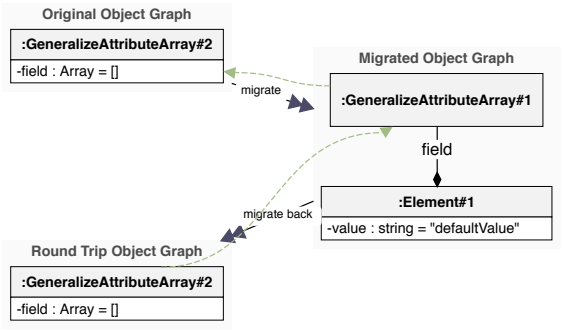
Migrations

```
1  @Migration
2  export function migrate(o1 : GeneralizeAttributeArray#1)
3          : GeneralizeAttributeArray#2 {
4          let o2 = new GeneralizeAttributeArray#2();
5
6          let elements : Array<Element#2> = [];
7
8          const previousRevision = context.getTrace(o1)[0] as GeneralizeAttributeArray#2;
9
10         // detect default value of 'o1.field'
11         if (previousRevision !== undefined
12                 && previousRevision.field.length == 0 && !context.isModified(o1, "field")) {
13                 // restore empty array, if default value in 'o1.field' remained unmodified
14                 elements = [];
15         } else {
16                 // add migrated field value as first array element
17                 elements.push(migrate(o1.field));
18
19                 // if a previous revision can be obtained
20                 if (previousRevision != undefined) {
21                         // add all previousRevision elements, but the first one
22                         previousRevision.field.slice(1)
23                                 .forEach(e => elements.push(Migrations.copy(Element#2, e)));
24                 }
25         }
26
27         // Finally assign the array of migrated elements to
28         // the migrated instance field 'field'
29         o2.field = elements;
30
31         return o2;
32 }
33 @Migration
34 export function migrateBack(o2 : GeneralizeAttributeArray#2) : GeneralizeAttributeArray#1 {
35         let o1 = new GeneralizeAttributeArray#1();
36
37         // migrate only the first element of the array
38         o1.field = Migrations.migrateElementAt(o2.field, 0, createDefaultElement());
39
40         return o1;
41 }
```

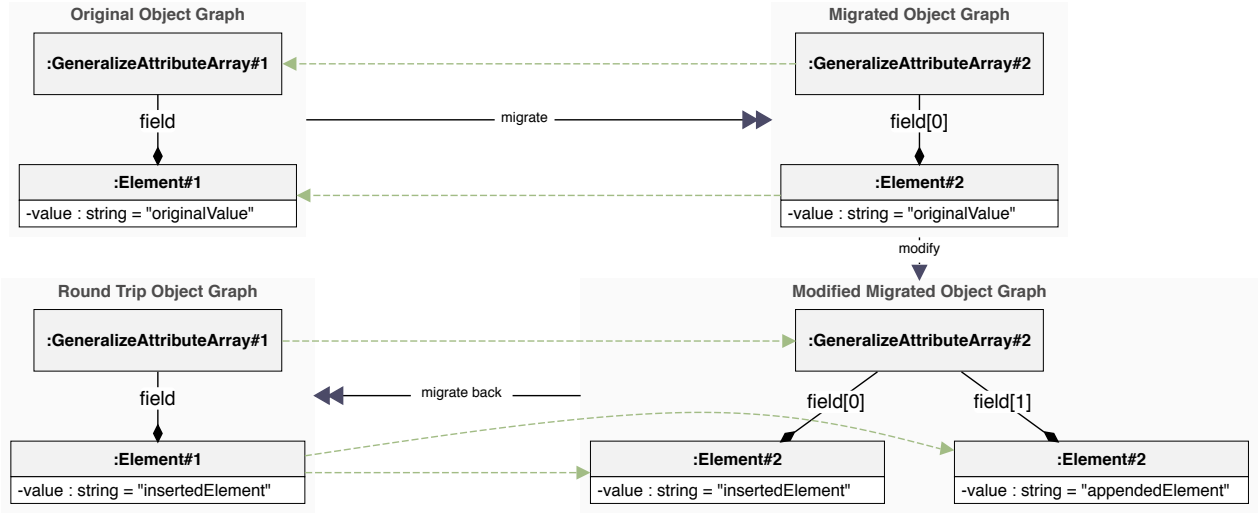
Shortened: For the full implementation of this scenario see the appended source code of the catalog.



Round-Trip 10.1: $\#1 \mapsto \#2 \mapsto \#1$ In version 2 the only element in array *field* is removed. This results in a default instance of `Element#1` in version 1.



Round-Trip 10.2: $\#2 \mapsto \#1 \mapsto \#2$ An empty array is migrated to a default instance of `Element#1`. When migrating back, the (unchanged) default value is recognized and the original empty array is restored.



Round-Trip 10.3: $\#1 \mapsto \#2 \mapsto \#1$ In version 2, a new element is inserted into the array at index 0. As a consequence, the new element of the array replaces the value of `field` in version 1.

Scenario 11: Pull Up / Push Down Field

A field is pulled up into a superclass (or pushed down respectively).

In our exemplary data model, the field `f` is pulled up into the supertype `SuperClass` in version 2 of the model.

Data Models

```
1 export public class SuperClass#1 {}
2 export public class PullUpFeature#1 extends SuperClass {
3         public f : string
4 }
```

Version 1

```
1 export public class SuperClass#2 {
2         public f : string
3 }
4 export public class PullUpFeature#2 extends SuperClass {}
```

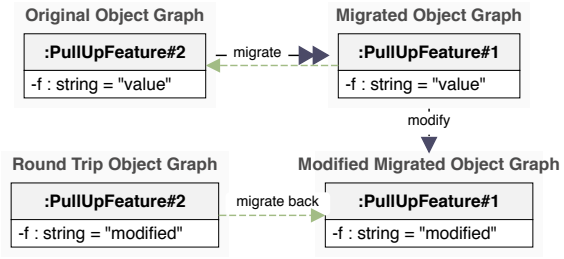
Version 2

Discussion On an instance level, the origin of field `f` (super field or local field) is not of importance since all fields of the supertype are consumed. Therefore we may migrate instances of `PullUpFeature` by copying.

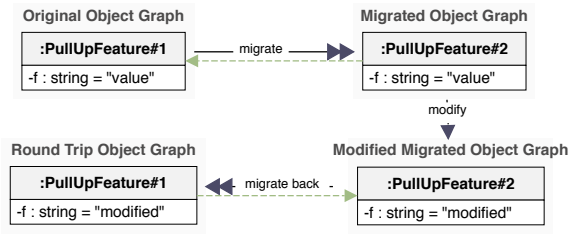
Another change that can be observed in this scenario, is that in model version 2, the class `SuperClass` gains the field `f`. This must be handled separately in the corresponding migration for type `SuperClass` (see scenarios 2, 3, 4 for a discussion).

Migrations

```
@Migration
export public function migratePullUpFeature(o1 : PullUpFeature#1) : PullUpFeature#2 {
        const o2 = new PullUpFeature#2();

        // simple copy the value of 'f'
        o2.f = o1.f;

        return o2;
}

@Migration
export public function migrateBackPullUpFeature(o2 : PullUpFeature#2) : PullUpFeature#1 {
        const o1 = new PullUpFeature#1();

        // simple copy the value of 'f'
        o1.f = o2.f;

        return o1;
}
```



Round-Trip 11.1: $\#2 \mapsto \#1 \mapsto \#2$ Instance of type `PullUpFeature` can simply be migrated by copying. Modifications equally apply in both model version.



Round-Trip 11.2: $\#1 \mapsto \#2 \mapsto \#1$ A round-trip in the other direction almost looks identical except for the type versions.

Scenario 12: Split/Merge Type

Based on a specified criteria, instances of a type of one model version, translate to different (unrelated) types of the other model version.

In the example data model, the type `Combined` indicates by a `type` field, which of the optional fields `intValue` and `stringValue` hold the actual data [1]. In other words, it models the concept of union types. In model version 2, `Combined#2` is split into two separate types. The referring class `SplitClass#2` furthermore specializes the type of its field `f`, as it now only allows instances of type `IntValue#2`.

Data Models

```
export public class SplitClass#1 {
        public f : Combined
}
export public enum CombinedType#1 { INT, STRING }
export public class Combined#1 {
        public type : CombinedType
        public stringValue? : string
        public intValue? : int
}
```

Version 1

```
export public class SplitClass#2 {
        public f : IntValue
}

export public class IntValue#2 {
        public intValue : int
}

export public class StringValue#2 {
        public stringValue : string
}
```

Version 2

[1] At this point we assume that the informal contract of the `type` field of `Combined#1` is not violated (e.g. `stringValue` is null even though `type` is `STRING`.

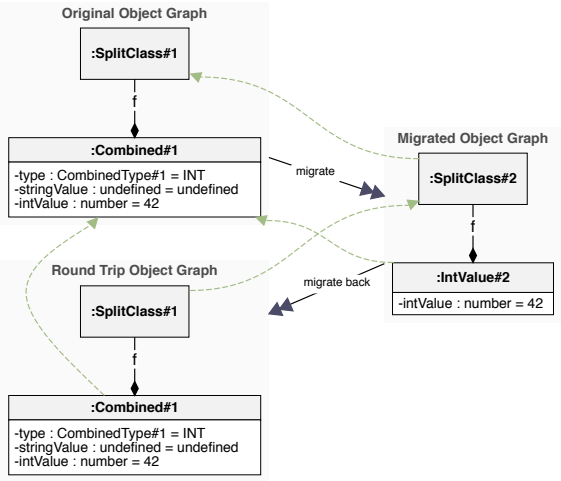
Discussion Since migrations operate on the instance level, we may evaluate the value of the `type` field during migration. Based on this information, we can decide whether to use the value in `intValue` of `Combined#1` (line 32) or whether we must provide a default value (line 29/30). Migrating instances of type `IntValue#2` back to `Combined#1` can be implemented by setting the `type` field to the corresponding `INT` literal (e.g. line 40).

Since we are dealing with default values, we must detect those and replace them with the previous revision if no modification can be detected (cf. line 16).

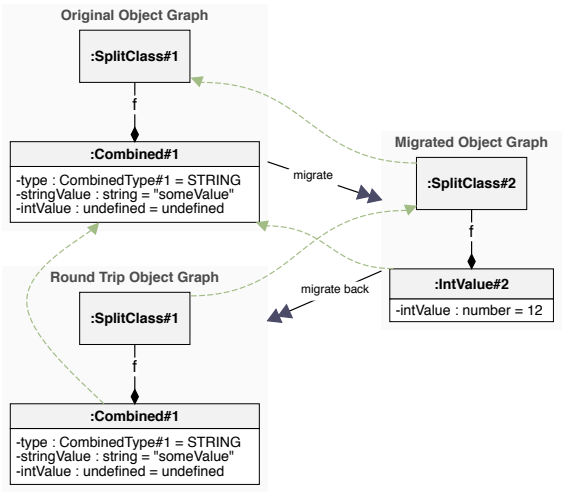
Migrations

```
1  @Migration
2  function migrate(sc1 : SplitClass#1) : SplitClass#2 {
3          const sc2 = new SplitClass#2();
4
5          // delegate migration of 'f'
6          sc2.f = migrate(sc1.f);
7
8          return sc2;
9  }
10
11 @Migration
12 function migrateBack(sc2 : SplitClass#2) : SplitClass#1 {
13         const sc1 = new SplitClass#1();
14         const previousRevision = context.getTrace(sc2)[0] as SplitClass#1;
15
16         if (previousRevision && !context.isModified(sc2.f)) {
17                 sc1.f = copy(Combined#1, previousRevision.f);
18         } else {
19                 // migrate instance of IntValue to Combined by delegation
20                 sc1.f = migrate(sc2.f);
21         }
22         return sc1;
23 }
24
25 @Migration
26 function migrateIntValue(combined : Combined#1) : IntValue#2 {
27         const iv = new IntValue#2();
28         if (combined.type != CombinedType#1.INT) {
29                 // choose default value if 'combined' is of wrong type
30                 iv.intValue = 12;
31         } else {
32                 iv.intValue = combined.intValue;
33         }
34         return iv;
35 }
36
37 @Migration
38 function migrateBackIntValue(iv : IntValue#2) : Combined#1 {
39         const c = new Combined#1();
40         c.type = CombinedType#1.INT;
41         c.intValue = iv.intValue;
42         return c;
43 }
```

Shortened: For the full implementation of this scenario see the appended source code of the catalog.



Round-Trip 12.1: $\#1 \mapsto \#2 \mapsto \#1$ In case the instance of `Combined#1` represents an integer value, the migration can be performed without the use of any default values.



Round-Trip 12.2: $\#1 \mapsto \#2 \mapsto \#1$ If the instance of `Combined#1` does not represent an integer value, the migration strategy makes use of default values in model version 2.

Scenario 13: Specialize/Generalize Superclass

The supertype of a class is changed to one of the supertype's subclasses/superclasses.

In the example data model, the supertype of class `SpecializeSuperType` is specialized to `SuperType` from `SuperSuperType` in version 2 of the model.

Data Models

```
1  export public class SuperSuperType#1 {
2          // no fields
3  }
4
5  export public class SuperType#1 extends SuperSuperType {
6          public superField : string
7  }
8
9  export public class SpecializeSuperType#1
10         extends SuperSuperType {
11         // Inheriting all fields of SuperSuperType,
12         // therefore this type does not have
13         // any fields in version 1.
14 }
```

Version 1

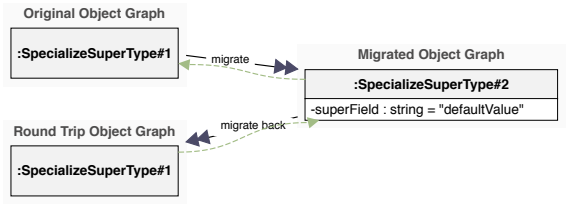
```
1  export public class SuperSuperType#2 {
2          // no fields
3  }
4
5  export public class SuperType#2 extends SuperSuperType {
6          public superField : string
7  }
8
9  export public class SpecializeSuperType#2
10         extends SuperType {
11         // Inheriting all fields of HighestSuperType,
12         // therefore this type inherits field 'superField'
13 }
```

Version 2

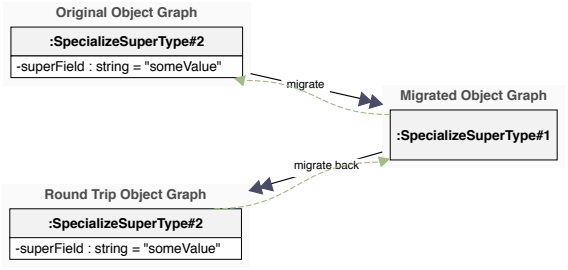
Discussion Since the specialization/generalization of the supertype can be seen as the removal and the addition of two unrelated supertypes, the migration strategy of this scenario is similar to that of scenario 6 Add/Remove a Supertype. Therefore, the actually migrated change is the creation/deletion of fields. Depending on potential functional dependencies between existing fields and fields that are introduced by the changed supertype, the scenarios 2, 3, 4 are then applicable.

Migrations

```
@Migration
function migrate(sst1 : SpecializeSuperType#1) : SpecializeSuperType#2 {
        const sst2 = new SpecializeSuperType#2();
        const previousRevision = context.getTrace(sst1)[0] as SpecializeSuperType#2;

        if (previousRevision) {
                // re-use previous revision 'superField' value, if present
                sst2.superField = previousRevision.superField;
        } else {
                // otherwise, choose a default value for 'superField'
                sst2.superField = "defaultValue";
        }

        return sst2;
}

@Migration
function migrateBack(sst2 : SpecializeSuperType#2) : SpecializeSuperType#1 {
        // empty type, nothing to migrate
        return new SpecializeSuperType#1();
}
```



Round-Trip 13.1: $\#1 \mapsto \#2 \mapsto \#1$ For the newly introduced field `superField`, a default value is chosen.



Round-Trip 13.2: $\#2 \mapsto \#1 \mapsto \#2$ If a previous revision can be obtained via trace links, the original value of `superField` is restored.

Scenario 14: Extract/Inline Superclass

A new superclass is extracted from the set of fields of an existing type.

In the example data model, in version 2 the field `genericField` is extracted into the new superclass `SuperClass#2`.

Data Models

```
export public class ExtractSuperClass#1 {
        public specificField : string
        public genericField : string
}
```

Version 1

```
export public class SuperClass#2 {
        public genericField : string
}

export public class ExtractSuperClass#2 extends SuperClass {
        public specificField : string
}
```

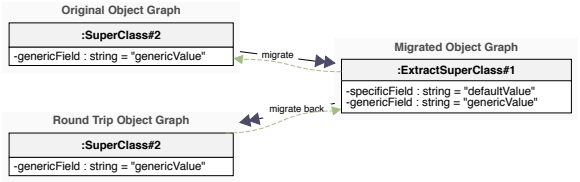
Version 2

Discussion Similar to scenario 11 Pull Up / Push Down Field, the migration for this scenario can be performed by copying, since the data model change is not visible on an instance level. However, by extracting a class, a new class is added to the data model. Therefore, we must also consider the migration of `SuperType#2` instances back to model version 1 (cf. Round-Trip 14.1 and 14.3). With the exemplary migration strategy below, we propose a similar solution as in scenario 5 *Declare class as abstract*.

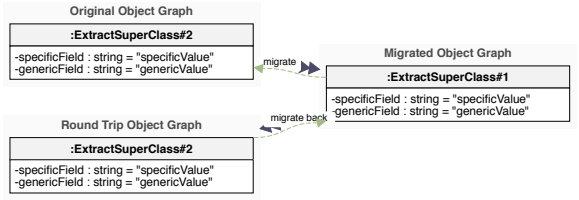
Alternatively, we can avoid the migration of `SuperType#2` instances, by additionally *declaring the extracted superclass as abstract*. As a consequence, the data model change of this scenario becomes fully transparent from an instance perspective.

Migrations

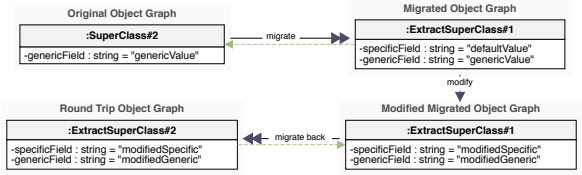
```
@Migration
function migrate(e1 : ExtractSuperClass#1) : SuperClass#2 {
        const previousRevision = context.getTrace(e1)[0] as SuperClass#2;

        // if a previous revision can be obtained
        if (previousRevision &&
                // and e1 has originally been an instance of SuperClass ...
                (previousRevision instanceof SuperClass#2) &&
                !(previousRevision instanceof ExtractSuperClass#2) &&
                // and e1.specificField has not been modified (is default)
                !(context.isModified(e1, "specificField"))) {

                // ... migrate back to a SuperClass instance
                const s = new SuperClass#2();
                s.genericField = e1.genericField;
                return s;
        }

        // otherwise copy all values over to a new instance
        // of type ExtractSuperClass#2
        return copy(ExtractSuperClass#2, e1);
}

@Migration
function migrateBack(e2 : ExtractSuperClass#2) : ExtractSuperClass#1 {
        // copy all values over to new instance of type ExtractSuperClass#1
        return copy(ExtractSuperClass#1, e2);
}

@Migration
function migrateSuperClass(s : SuperClass#2) : ExtractSuperClass#1 {
        // migrate instances of SuperClass#2 back to ExtractSuperClass#1
        // as there is no other means to represent them in model version 1
        const e = new ExtractSuperClass#1();

        e.genericField = s.genericField;
        e.specificField = "defaultValue";

        return e;
}
```



Round-Trip 14.1: $\#2 \mapsto \#1 \mapsto \#2$ An instance of `SuperType#2` is migrated to an instance of `ExtractSuperClass` in version 1. For the additional field `specificField`, a default value must be chosen.



Round-Trip 14.2: $\#1 \mapsto \#2 \mapsto \#1$ On an instance-level, moving fields into a superclass is not visible.



Round-Trip 14.3: $\#2 \mapsto \#1 \mapsto \#2$ A modification of `specificField` of an `ExtractSuperClass#1` instance that originally stems from a `SuperType#2` instance, is mapped back to an instance of `ExtractSuperClass` in version 2.

Scenario 15: Fold/Unfold Superclass

A new superclass is declared for a type. Common fields of the superclass and the type are then removed from the type (folded into superclass).

In our exemplary data model, the class `FoldSuperType` gains the new supertype `SuperClass`. As a consequence its fields `f1` and `f2` can be folded into `SuperClass`.

Data Models

```
1 export public class SuperClass#1 {
2         public f1 : string
3         public f2 : string
4 }
5 export public class FoldSuperClass#1 {
6         public f1 : string
7         public f2 : string
8 }
```

Version 1

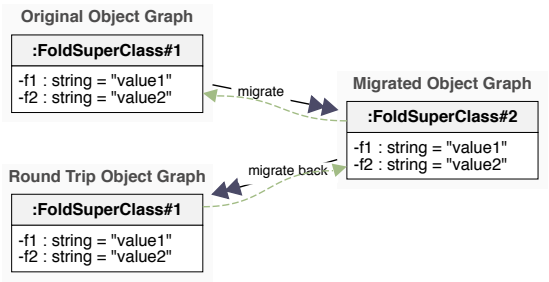
```
1 export public class SuperClass#2 {
2         public f1 : string
3         public f2 : string
4 }
5 export public class FoldSuperClass#2 extends SuperClass {}
```

Version 2

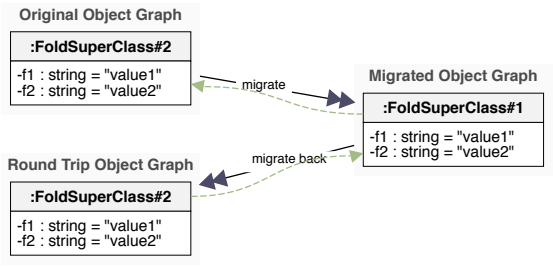
Discussion This scenario is closely related to scenario 14 Extract Super Class. It only differs in the fact that the superclass already existed in model version 1. Therefore, there is no need to further consider a migration of `SuperClass#2` instances. Since the changes to the class hierarchy are transparent on the instance level, a migration-by-copying strategy can be deployed.

Migrations

```
@Migration
function migrateFoldSuperClass(f : FoldSuperClass#1) : FoldSuperClass#2 {
        const fs2 = new FoldSuperClass#2();

        // simply copy over values of 'f1' and 'f2'
        fs2.f1 = f.f1;
        fs2.f2 = f.f2;

        return fs2;
}

@Migration
function migrateBackFoldSuperClass(f : FoldSuperClass#2) : FoldSuperClass#1 {
        const fs1 = new FoldSuperClass#1();

        // simply copy over values of 'f1' and 'f2'
        fs1.f1 = f.f1;
        fs1.f2 = f.f2;

        return fs1;
}
```



Round-Trip 15.1: $\#1 \mapsto \#2 \mapsto \#1$ On an instance-level, the changes in the class hierarchy are transparent.



Round-Trip 15.2: $\#2 \mapsto \#1 \mapsto \#2$ Apart from the concrete type versions, this round-trip equals the other direction.

Scenario 16: Extract/Inline Subclass

A selection of fields is extracted into a new subclass (or inlined respectively).

In our example data model, the field `specificField` is extracted into a new subclass `SubA#2` of type `A`. The field `genericField` remains part of the original type `A`. The class `ExtractSubClass` only serves as a container that holds a reference to an instance of type `A`.

Data Models

```
export public class A#1 {
        public specificField : string
        public genericField : string
}

export public class ExtractSubClass#1 {
        public f1 : A
}
```

Version 1

```
export public class A#2 {
        public genericField : string
}

export public class SubA#2 extends A {
        public specificField : string
}

export public class ExtractSubClass#2 {
        public f1 : A
}
```

Version 2

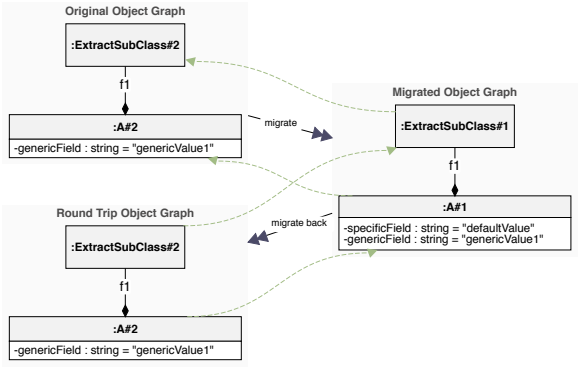
Discussion Since references to instances of type `A#2` can always refer to an instance of type `SubA#2` as well, migrating from model version 1 to 2, can simply be implemented by migrating all instances of `A#1` to instances of `SubA#2`.

For $\#2 \mapsto \#1 \mapsto \#2$ round-trips however, we must consider the case of direct instances of `A#2` (cf. Round-Trip 16.1 and 16.2). Similar to scenario 5 Declare Class as Abstract, we need to make use of traceability features in order to detect `A#1` instances with default values that have previously been `A#2` instances (line 7-14 in `migrateA`).

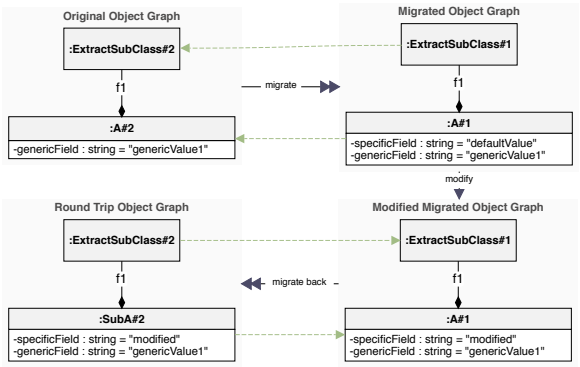
Migrations

```
@Migration
function migrateA(a : A#1) : A#2 {
        const previousRevision = context.getTrace(a)[0] as A#2;

        let a2 : A#2;

        // if previous revision exists, is of type SubA...
        if (previousRevision instanceof SubA#2 ||
                // ..or 'specificField' has been modified
                context.isModified(a, "specificField")) {

                // migrate-back to an instance of SubA
                a2 = new SubA#2();
                (a2 as SubA#2).specificField = a.specificField;
        } else {
                // otherwise migrate-back to A
                a2 = new A#2();
        }

        // copy over value of 'genericField'
        a2.genericField = a.genericField;

        return a2;
}

@Migration
function migrateBackA(a : A#2) : A#1 {
        const a1 = new A#1();

        a1.genericField = a.genericField;
        // use default value for missing field 'specificField'
        a1.specificField = "defaultValue";

        return a1;
}

@Migration
function migrateBackSubA(a : SubA#2) : A#1 {
        const a1 = new A#1();

        a1.genericField = a.genericField;
        a1.specificField = a.specificField;

        return a1;
}
```

Shortened: For the full implementation of this scenario see the appended source code of the catalog.



Round-Trip 16.1: $\#2 \mapsto \#1 \mapsto \#2$ An instance of A#2 is migrated to A#1 using a default value. When migrating back however, the default value is detected and the instance is again represented as A#2



Round-Trip 16.2: $\#2 \mapsto \#1 \mapsto \#2$ An instance of A#2 is migrated to A#1 using a default value. After a modification of field `specificField`, the instance is migrated back to SubA#2 to represent the modification in version 2.

Scenario 17: Extract/Inline Class

A selection of fields is extracted into a new delegate class.

In the example data model, the field `f` is moved from class `ExtractClass` to class `DelegateClass`. Instead, `ExtractClass` holds a mandatory reference to an instance of `DelegateClass` in version 2 of the data model.

Data Models

```
1 export public class ExtractClass#1 {
2         public f : string
3 }
```

Version 1

```
1 export public class ExtractClass#2 {
2         public delegate : DelegateClass
3 }
4 export public class DelegateClass#2 {
5         public f : string
6 }
```

Version 2

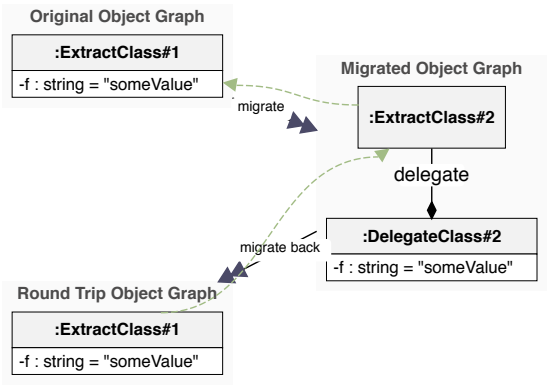
Discussion Since the field `delegate` is mandatory, a migration can collect the value of field `f` via the `delegate` reference (line 17 in `migrateBackExtractClass`). This means, that version 2 of the data model is semantically equivalent to model version 1 and no traceability features are required to successfully migrate instances.

In model version 2, instances of the new class `DelegateClass` may occur. In our exemplary migration implementation, we do not implement the migration of such `DelegateClass` instances back to model version 1, since we assume that those only occur in combination with `ExtractClass` instances. In a concrete case, this assumption may be invalid when there exists another use of `DelegateClass` instances. Depending on the nature of such, further scenarios must then be consulted.

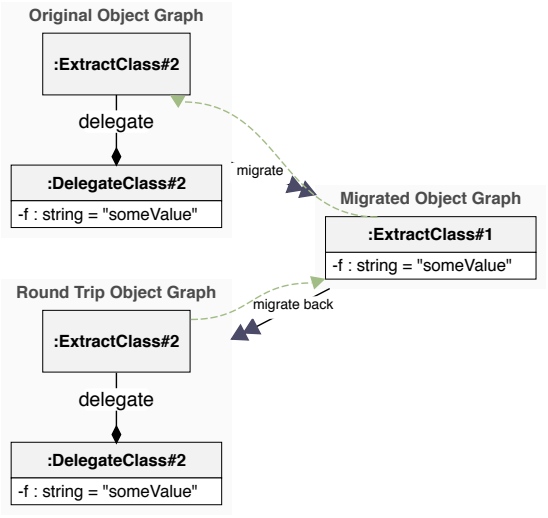
A modification of field `f` can simply be mapped in both directions, by applying it to the corresponding field of `DelegateClass` or `ExtractClass` respectively.

Migrations

```
1  @Migration
2  function migrateExtractClass(ec : ExtractClass#1) : ExtractClass#2 {
3          const ec2 = new ExtractClass#2();
4
5          // create new delegate class to hold value of 'field''
6          ec2.delegate = new DelegateClass#2();
7          ec2.delegate.f = ec.f;
8
9          return ec2;
10 }
11
12 @Migration
13 function migrateBackExtractClass(ec : ExtractClass#2) : ExtractClass#1 {
14         const ec1 = new ExtractClass#1();
15
16         // collect value of 'field' via 'delegate''
17         ec1.f = ec.delegate.f;
18
19         return ec1;
20 }
```



Round-Trip 17.1: $\#1 \mapsto \#2 \mapsto \#1$ The value of field `f` is moved into a new `DelegateClass` instance.



Round-Trip 17.2: $\#2 \mapsto \#1 \mapsto \#2$ The value of field `f` in model version 1 is collected from the `DelegateClass` instance of model version 2.

Scenario 18: Fold/Unfold Class

A selection of fields is folded into an existing delegate class.

In the example data model, the field `f` is folded into class `OtherClass`. Instead, `FoldClass` holds a reference to delegate class `OtherClass` in version 2 of the data model.

Data Models

```
export public class OtherClass#1 {
        public f : string
}

export public class FoldClass#1 {
        public f : string
}
```

Version 1

```
export public class OtherClass#2 {
        public f : string
}

export public class FoldClass#2 {
        public delegate : OtherClass
}
```

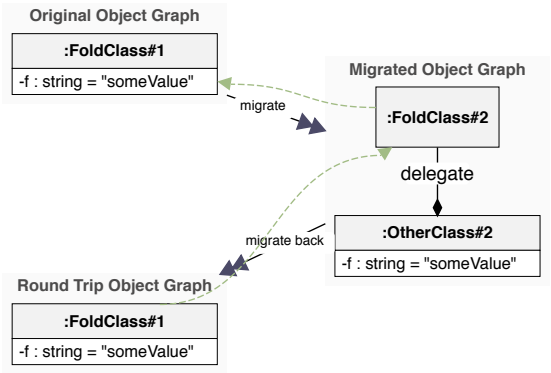
Version 2

Discussion Similar to scenario 17 Extract/Inline Class, the two data model versions in the scenario represent semantically equivalent models. Thus, a round-trip migration can be performed without the use of any traceability features (see scenario 17 for further discussion).

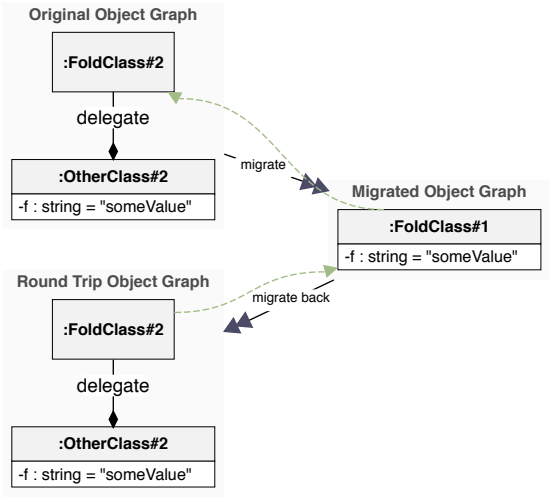
Since the class into which the fields are folded, already existed in version 1 of the model, we do not have to further consider the migration of `OtherClass` instances.

Migrations

```
@Migration
function migrateFoldClass(fc : FoldClass#1) : FoldClass#2 {
        const fc2 = new FoldClass#2();

        fc2.delegate = new OtherClass#2();
        // move value of 'f' to delegate instance
        fc2.delegate.f = fc.f;

        return fc2;
}
@Migration
function migrateBackFoldClass(fc : FoldClass#2) : FoldClass#1 {
        const fc1 = new FoldClass#1();

        // collect value of 'f' from delegate instance
        fc1.f = fc.delegate.f;

        return fc1;
}
```



Round-Trip 18.1: $\#1 \mapsto \#2 \mapsto \#1$ The value of field `f` is moved to delegate class `OtherClass#2`.



Round-Trip 18.2: $\#2 \mapsto \#1 \mapsto \#2$ The value of field `f` is collected from delegate class `OtherClass#2`.

Scenario 19: Collect Field over Reference

A field is collected/pushed over a reference.

In our example data model, `field` is collected from `SuperClass` via field `reference` into the class `CollectField`. As a consequence, `field` of `CollectField#2` assumes the multiplicity (0..1) of field `reference`.

Data Models

```
export public class CollectField#1 {
        public reference? : SourceClass
}
export public class SourceClass#1 {
        public field : string
        public someOtherField : string
}
```

Version 1

```
export public class CollectField#2 {
        public reference? : SourceClass
        public field? : string
}

export public class SourceClass#2 {
        public someOtherField : string
}
```

Version 2

Discussion In a migration strategy for this scenario, we mainly need to decide on a mapping between potential `null`-values for the optional fields `CollectField#1.reference`, CollectField#2.reference and `CollectField#2.field`. Specifically, for a migration of `CollectField` from model version 2 to 1 we need to consider a trade-off between default values and loss of information. In our exemplary migration strategy, we propose the following mapping:

Version 2		**Version 1**
`CollectField#2.reference`	`CollectField#2.field`	
`null`	value	Set `reference` to an instance of `SourceClass` and choose a default value for `SourceClass#1.someOtherField`.
value	`null`	Migrate the instance of `SourceClass#2` of field `reference` to version 1 by choosing a default value for `SourceClass#1.field`.
value	value	Migrate the instance of `SourceClass#2` of field `reference` to version 1 using the value in `field`.
`null`	`null`	Set `CollectField#1.reference` to `null`.

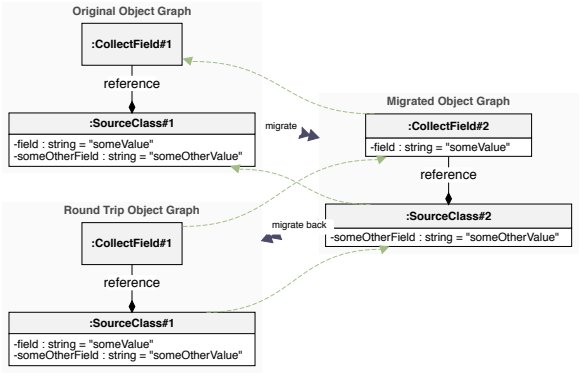
Round-Trip 19.1, 19.2, 19.3 and 19.4 demonstrate how this strategy affects different cases. In general, we aim to minimize the use of default values while also minimizing the loss of information (e.g. `field` is set to a value in one version, but the change is not visible in the other version).

The overall challenge this scenario poses, is the combinations of multiplicities in `SourceClass` as well as of field `reference`. Since both fields of `SourceClass` are mandatory, they are coupled. More specifically, we cannot only represent the presence of one of the fields. In model version 2 however, the fields are decoupled in that we may represent the presence of only one of `field` and `someOtherField`. In our migration strategy, we propose to compensate for this semantic difference, by the use of default values.

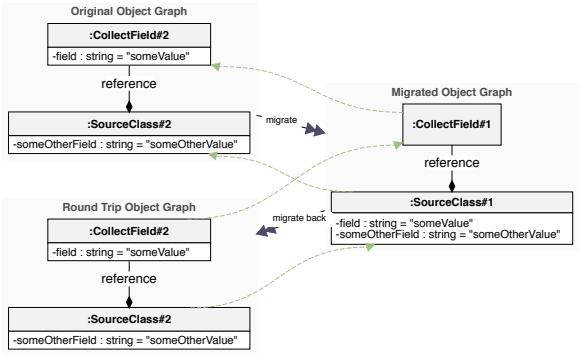
Migrations

```
@Migration function migrateCollectField(cf : CollectField#1) : CollectField#2 {
        const cf2 = new CollectField#2();

        // migrate value of 'field', if 'reference' is present
        cf2.field = cf.reference == null ? null : cf.reference.field;

        if (cf.reference == null) {
                cf2.reference = null;
        } else {
                const previousRevision = context.getTrace(cf)[0] as CollectField#2;
                // If in the previous revision 'reference' was null
                // and 'reference.someOtherField' holds the unmodified default value...
                if (previousRevision &&
                        previousRevision.reference == null &&
                        !context.isModified(cf.reference, "someOtherField")) {
                        // ... migrate back to 'reference' being null
                        cf2.reference = null;
                } else {
                        // otherwise, 'someOtherField' holds new information (changed))
                        cf2.reference = migrate(cf.reference);
                }
        }

        return cf2;
}

@Migration
function migrateBackSourceClass(s : SourceClass#2, fieldValue : string) : SourceClass#1 {

        const sc1 = new SourceClass#1();

        // migrate 'someOtherField'
        sc1.someOtherField = s.someOtherField;
        // use given value for 'field' or a default value
        sc1.field = fieldValue || "defaultValue";

        return sc1;
}

@Migration function migrateBackCollectField(cf : CollectField#2) : CollectField#1 {
        const cf1 = new CollectField#1();

        if (cf.reference != null) {
                cf1.reference = migrate(cf.reference, cf.field);
        } else {
                if (cf.field != null) {
                        cf1.reference = new SourceClass#1();
                        cf1.reference.field = cf.field;
                        cf1.reference.someOtherField = "someOtherDefaultValue";
                } else {
                        cf1.reference = null;
                }
        }

        return cf1;
}
```

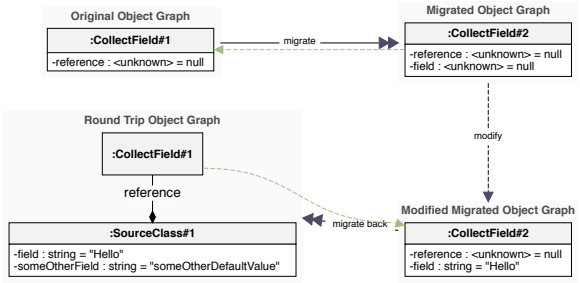
Shortened: For the full implementation of this scenario see the appended source code of the catalog.



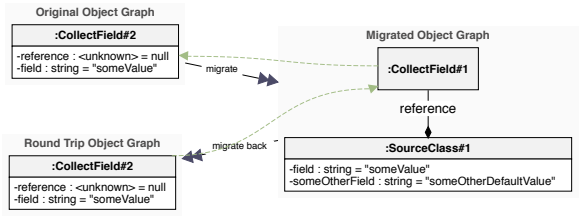
Round-Trip 19.1: $\#1 \mapsto \#2 \mapsto \#1$ The value of `field` is collected into class `CollectField` in model version 2.



Round-Trip 19.2: $\#2 \mapsto \#1 \mapsto \#2$ Via `reference`, `field` is pushed back to class `SourceClass` in model version 1.



Round-Trip 19.3: $\#1 \mapsto \#2 \mapsto \#1$ A `null`-value for `reference` in model-version 1 translates to `null`-values for both `field` and `reference` in model version 2. A modification sets a value for `field` only. As a consequence, a default value is chosen for `someOtherField` in version 1 of the model.



Round-Trip 19.4: $\#2 \mapsto \#1 \mapsto \#2$ The use of a `SourceClass` default instance is detected when migrating back to model version 2.

Scenario 20: Split/Merge Fields

A type is split by moving its fields to two new types and correspondingly replacing all references to it by references to the new types.

In the example data model, in version 2 the type X is split into Y and Z. Its field a is moved to Y and its field b is moved to Z. The type `SplitField#1` holds a reference to an X instance in version 1 and to corresponding instances of Y and Z in version 2.

Data Models

```
export public class SplitFields#1 {
        public x : X
}

export public class X#1 {
        public a : string
        public b : string
}
```

Version 1

```
export public class SplitFields#2 {
        public y : Y
        public z : Z
}
export public class Y#2 {
        public a: string
}

export public class Z#2 {
        public b: string
}
```

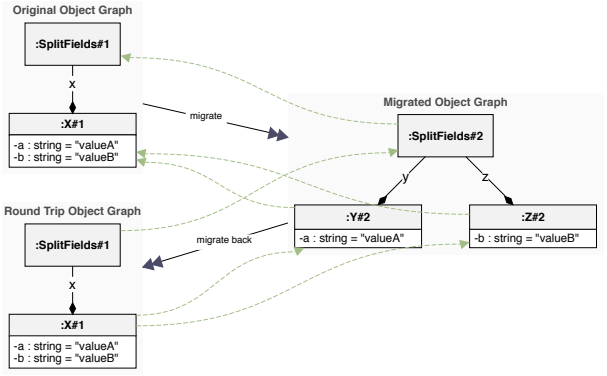
Version 2

Discussion The one-to-one correspondence of an instance of X and instances of Y and Z implies a semantic equivalence between the data model versions. Therefore, we may deploy a migrations strategy which does not use any traceability features. Furthermore, our proposed migration strategy leverages the support for multiple migration parameters and return types in N4IDL. This also becomes apparent in the Round-Trip 20.1 and 20.2.

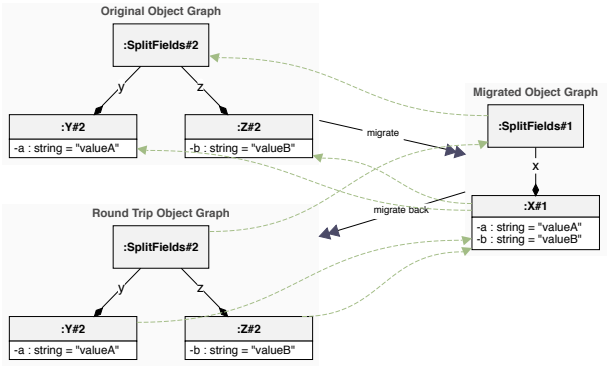
Due to the one-to-one correspondence of the fields in type X and the fields in Y and Z, any modifications of the fields a and b, can directly be mapped to the other version.

Migrations

```
@Migration function migrateSplitFields(sf : SplitFields#1) : SplitFields#2 {
        const sf2 = new SplitFields#2();

        // migrating X#1 by splitting it into Y#2, Z#2
        const yAndZ = migrate(sf.x);
        sf2.y = yAndZ.y;
        sf2.z = yAndZ.z;

        return sf2;
}

@Migration function migrateBackSplitFields(sf : SplitFields#2) : SplitFields#1 {
        const sf1 = new SplitFields#1();

        // migrate back to X#1 based on Y#2, Z#2
        sf1.x = migrate(sf.y, sf.z);

        return sf1;
}

@Migration function migrateYZ(y : Y#2, z : Z#2) : X#1 {
        const x = new X#1();

        x.a = y.a;
        x.b = z.b;

        return x;
}

@Migration function migrateX(x : X#1) : ~Object with {y : Y#2, z : Z#2} {
        const y = new Y#2();
        const z = new Z#2();

        // copy over the values of field 'a' and 'b' to
        // the instances of X and Z respectively
        y.a = x.a;
        z.b = x.b;

        return {y: y, z: z};
}
```



Round-Trip 20.1: $\#1 \mapsto \#2 \mapsto \#1$ The fields a and b are distributed between the instances of Y and Z in model version 2.



Round-Trip 20.2: $\#1 \mapsto \#2 \mapsto \#1$ Due to the use of migrations with multiple parameters and return types, the instances of Y and Z both link back to the same instance of X in version 1.

6 Learning Outcomes

During our work on the catalog, we were able to identify a selection of recurring problems that must be solved when implementing round-trip migrations. In the following, we discuss some of these problems by reformulating them independently from the concrete cases in which they appear.

Using Default Values

Many of the represented migration strategies leverage the use of default values. This is usually required when the target model version requires some sort of information to be available, while the source model version allows for the omission of said information. In round-trip migrations this imposes a challenge: When migrating back to the original model version, we must be able to detect default values so that we can avoid to introduce redundant information into the original model instance. Furthermore, our criteria for successful RTMs without modification require that default values do not appear in the round-trip migrated instance. To address this issue, we recommend the use of modification detection in order to detect unmodified default values and replace them with their original representation (e.g. an absent optional field). In case a modification introduces instance data that is equal to the corresponding default values, we assume an explicit user intent and therefore migrate such changes back to the original model version. For these cases, the criteria for successful RTMs with modification apply. Therefore, using default values in migrations always entails the need for modification detection in order to distinguish explicit user intent from implicitly-set default values.

Information Redundancy

Redundancy of information imposes another challenge. More specifically, let us consider an original model version that models a certain bit of information using a single construct (e.g. a single field). Let us further consider another model version in which this bit of information is to be found in two different places (e.g. two separate fields). Informally, a functional dependency exists between these two sites and modifications must always equally apply to both sites. However, in the concrete case this may not always hold true. Therefore, the implementation of a migration usually encodes a precedence between the two redundant sites of information. As a consequence, if a modification updates the information inconsistently, a round-trip migration may dismiss the change and prioritize differently. While in data modeling, redundancy is usually undesired, it may still exist and it is important to note this implication for round-trip migrations.

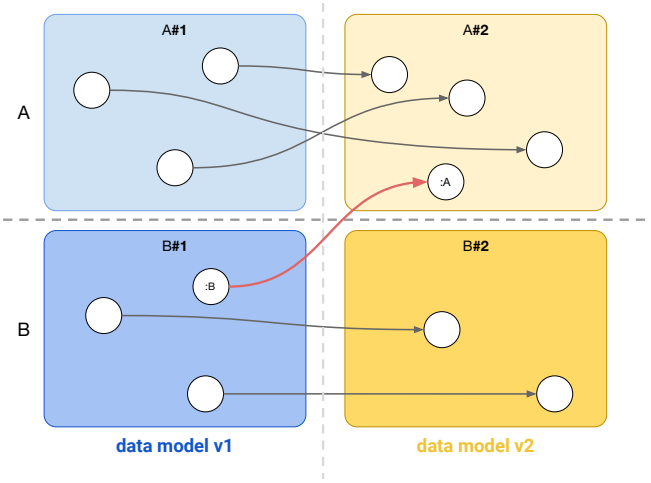


Figure 6.1: Schematic illustration of type A and B in different versions with corresponding representative objects in their instance sets (white nodes). The arrows demonstrate the mapping of objects that is imposed by a translation layer. Most objects are migrated within type boundaries (e.g. object of type A#1 to object of type A#2). The marked objects `:B` and `:A` however, are migrated across type boundaries. In these cases, the same system entity is represented as B in model version 1 and as A in model version 2.

Migration Across Type Boundaries

Finally, another recurring challenge is the migration across type boundaries. In some cases, it is required to migrate objects of a type A of the first model version to a type B of the second model version. This is to be differentiated from a simple type renaming, since we further assume that the types A and B exist in both of the model versions. However, some of the objects in the corresponding instance sets of A and B conceptually need to be moved to the other set during migration: The objects are migrated across type boundaries. An illustration of this idea is given in Figure 6.1. In migrations that face this issue, we must differentiate between objects that originally stem from the same type (e.g. B#1 to B#2) and objects that originally stem from a different type (e.g. A#1 to B#2). In order to fulfill our criteria of successful RTMs in these cases, we have found the use of trace links and runtime type checks to be an effective measure. More specifically, using trace links and runtime type checks we can differentiate the two types of objects and apply according migration strategies.

References

[1] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, “An extensive catalog of operators for the coupled evolution of metamodels and models,” in *International Conference on Software Language Engineering*. Springer, 2010, pp. 163–182.

[2] T. Kehrer, “Calculation and propagation of model changes based on user-level edit operations: a foundation for version and variant management in model-driven engineering,” Ph.D. dissertation, 2015.