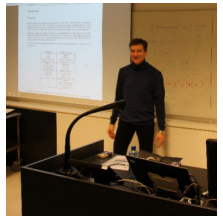
Høgskulen
på Vestlandet

Bidirectional Transformations, Part 1

Introduction, Motivation, and Concepts

Organization

- Online Lecture Mar 16, 17, 23, 24
  - Exercise Mar 17th → Mar 22nd
- Course Material can be found in Canvas‘ Moduls
- Who am I?
  - Practical Experience as a software engineer @ SAP
  - Became Prof. long time ago @ FHDW Hannover, Germany; I am also Prof II @ HVL
  - Research: Joint work with Patrick, Adrian, Yngve



2

Introduction

- Software (Re-)Engineering:
  - Engineer: gignere = produce, ingenium = invent
  - Re = repeat, renew
  - → Repeated Invention (of Software-Application)
- Becomes a continuous procedure
- Reengineering itself is trivial, but controlling *adjustments* in System *Y* after a change in System *X* is an issue
  - Goal of the lecture → *Synchronization of correlated IT-Applications*

3

Today little activity concerns the initial development of software solutions. Instead most activities are related to the maintenance and enhancing *existing* applications.
This **reengineering** is an ongoing cyclic procedure.
I will not talk much about reengineering itself, because little can be said about it (there are refactoring patterns, e.g. Martin Fowler, but that‘s it), but the control of depending adaption is important, e.g. changes in an object model infers changes in the data model and maybe vice versa.
The goal of the lecture is to analyse concepts and solutions when correlated IT-applications must be synchronized.
.

Introduction

- MDSE
  - Model Driven Software Engineering
- Correlated parts of an information system:
  - Object Model → Database Schema, User Interface
  - Use-Case → Activity diagram, Domain model
  - Can you give other examples of dependent SWE artifacts?
    - Test Cases + Class Models
    - Data in the database vs. database schema
- In all these cases, …
  - … consistency of different representations of the same concept must be kept in sync, i.e. …
  - … adaption in one place requires restoration in a different place, i.e. we require *global* consistency

4

We will see some other examples during the talk, and these examples are in most cases not about code generation: In MDSE there are dependencies between many **artifacts,** e.g. activity diagrams evolve from use-cases or user stories, furthermore the domain model (business terminology and its interrelations) is developed from use cases.

Whenever one artifact in a dependency changes, the dependent artifacts may have to be adapted → How can we cope with this ubiquitous problem?

→ A compound system is said be *globally* consistent, just like the two synchronized divers (two „systems“ which are in certain states that must correlate: jump – fall - dive).

Introduction

- Not only on the model level:
- There are also examples of updatable *data sources*, that require consistency restoration
  - Microservices: Overlapping (i.e. redundant) contents of databases must be kept consistent
  - Can you give other examples?
    - …?
- Thus there is a generalization of the goal:

1. How can we coordinate changes (on all levels), such that consistency of different representations of the same concept can permanently be in sync?
2. Can we provide a concept of a supporting maintenance framework?

5

On the previous slides model artefacts had to be kept in sync, but dependencies are not only on the model level: Since each microservice has its own databases, maintained data of different services may be redundant and hence overlap. I.e. changing the contents of a database requires the adaption of the contents of other databases!

E.g. a service A deals with goods ordering, service B with customer relationship management. Both maintain customer data. This situation also occurs after the merge of two companies (which may have the same business partner) or in the health care system, where general practitioners, hospitals and health insurance companies each possess a certain portion of person data, partly overlapping.

Contents

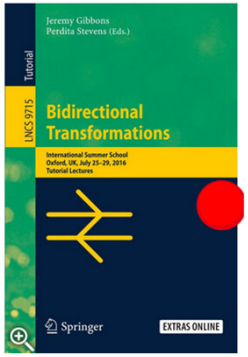
- Week 1: Motivation and Concepts
  - From Model Transformation to *Model Synchronization*
  - Model Spaces and their Interrelations
  - Bidirectional Transformations
    - Symmetric and Asymmetric Lenses
- Week 2: Algorithm for Consistency Restoration
  - Inter Model Constraints
  - Principles of Least Change and Least Surprise
  - Reduction to Single Models: Model Merge
  - Search-Based Repair

Literature and …

[1] Abou-Saleh, Cheney, Gibbons, McKinna, and Stevens: *Introduction to Bidirectional Transformations*

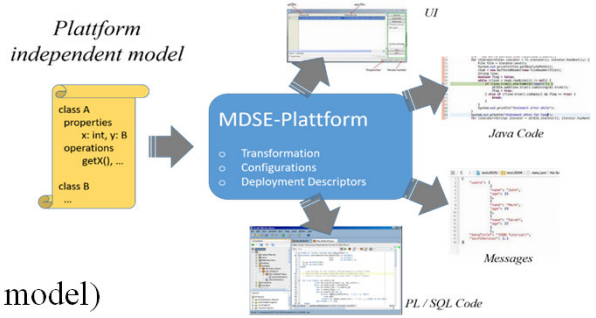
[2] Gibbons, Stevens: *Bidirectional Transformations,* International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures, Springer 2018

… a tiny Java Demo, what we will do …



Considerations Beyond Model Transformation

Model Driven Software Engineering



- Goal: Overlapping Structures, e.g. …
  - OO Models
  - Data Models
  - UI Design
  - Data Exchange Formats
- … kept consistent with MDSE tools
- Idea of a System model (a.k.a. platform independent model)
- Vision of MDSE:

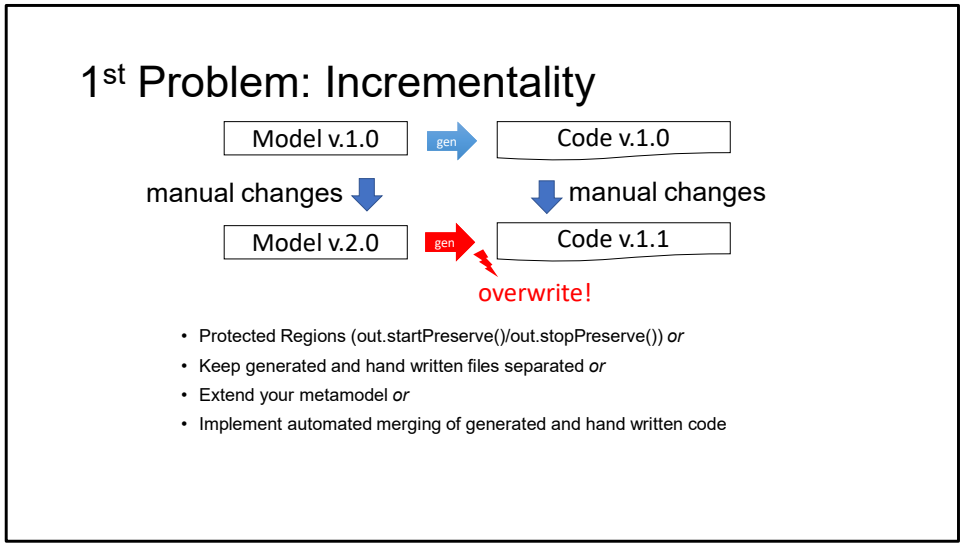
  The architectures and capabilities [of MDSE-landscapes] will produce a general class of highly dynamic and self-organizing systems that can act directly on domain knowledge and behave intelligently without having to be told how. . . . When systems do need to be modified, this is accomplished by altering the system model. This may be performed by domain experts who are not necessarily software specialists, or perhaps by the system itself, in many cases. *(J. D. Poole: Model-Driven Architecture: Vision, Standards And Emerging Technologies, **2001**)*
- Why is this still a vision?
  - …?

9

In MDSE the typical situation is the informational overlap of primary artifacts (e.g. a domain model of business terms) and its derived (generated, transformed) artifacts (e.g. the generated code) as you have seen it in EMF. That was the vision of MDSE: An IT-Systems evolution is driven by (the change of) domain knowledge **only**. If a requirement changes (e.g. a new kind of insurance policy is added) the adaption can be carried out by domain experts, who adapt the system model, and then (abracadabra) the system adapts automatically (!).

→ Thus, in a perfect world, a change in an IT-System must always be carried out in a unique primary artefact and all subordinated parts are re-generated

This is still a vision for several reasons: We will consider the problems on next slides.



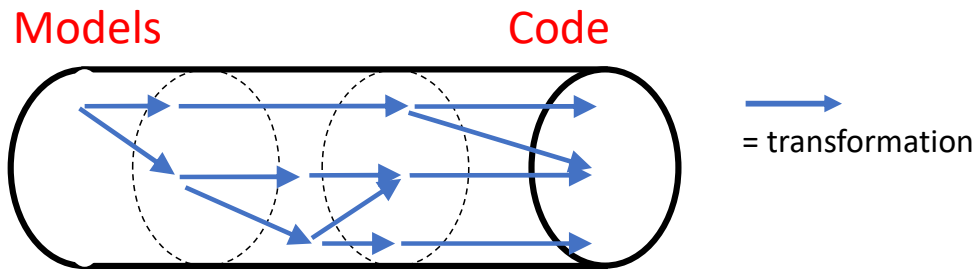
Models like class diagram undergo frequent updates. On the other hand, at each point in time there is code generated from the model. If after a model update, code is (re-)generated, this may overwrite manual changes in the generated code of the older model version. There are certain ways to cope with this issue:

- Define protected regions which shall not be touched by manual changes → portioning, constant complement (see later examples), @generated
- Manual changes are forbidden in generated files → complementation is carried out via subclassing
- Metamodel extension refers to the invention of special linguistic elements that specify manual changes
- Automated merges as in text-based repos are error-prone and do not always lead to a consistent state

2nd Problem: Multi-Stage Transformations

- PIM -> PSM[1] -> Code
  - Domain model -> Technical Class Diagram
  - Relations -> Associations or Aggregations
  - Multiple Inheritance -> …
  - Adding design patterns

  } -> Java Code



1 P(S/I)M = Platform (Specific / Independent) Model

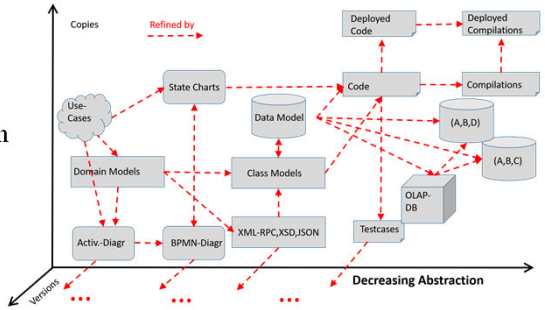
11

Also MDSE is not a one stage procedure: The source of a transformation step may already be the target of a previous step, e.g. a business model is derived from use-cases, a technical class model (with more design patterns) is generated from the business model, code is generated afterwards.
A concrete example, which often happens: A Use-Case yields a business model with multiple inheritance: TrafficLight inherits from ElectronicDevice, but in order to control the crossing, the TrafficLight also inherits from Observable, a general concept of objects where others may register for events (in this case the event that the traffic light switches to red telling a pedestrian traffic light to turn to green). Thus a *second* transformation must output a model, which avoids multiple inheritance for subsequent Java Code Generation.

This leads to our first variant of the depicted **wind tunnel**

3rd Problem: Models on the same level of abstraction

- State Chart + BPMN Diagram
- Data Model + Class Diagram
- Different, but overlapping class models
- And many more

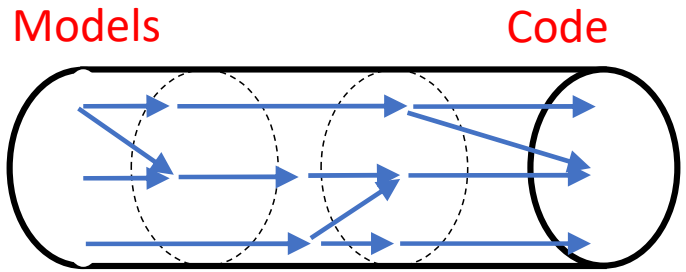


12

Models on the same level of abstraction both influence subordinate artefacts, e.g. a domain model is refined by a technical class models which dominates code generation. But domain models can also be enriched by adding process models (e.g. BPMN or State Charts), which also influence generated code. Then the question arises which is the real primary model.

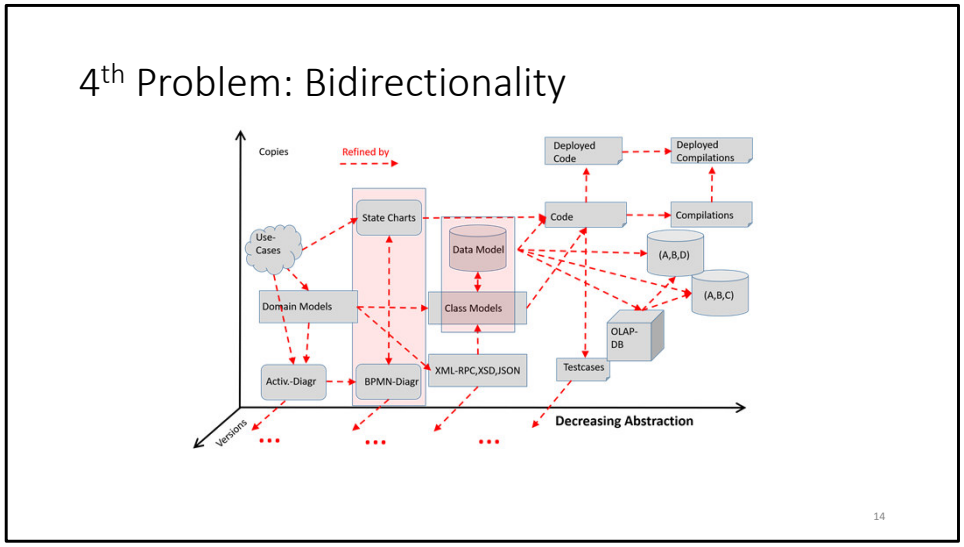
Consequences …

- Multi-Stage *and* Multi-Level Dependencies



13

So, there are different models at the same level of abstraction (the **wind tunnel** has no unique starting point)



In the figure, we see the refinement relation in both directions for state charts and BPMN-diagrams: Whereas state charts specify at which incoming events an object changes its state, the BPMN-diagram models the causality of activities and does not specify states explicitly. Vice versa the BPMN diagram also specifies interaction between objects of different types whereas state charts usually restrict to one object. Hence each of them refines the other. We can expect to have conflicting results if there are common generated artefacts.
Similarly, code is generated from class models (method headers, attributes, etc), but also from data models, e.g. stored procedures. However, if data model and class model are not in sync the call and the definition of stored procedure may be inconsistent.

5th Problem: Reverse Engineering

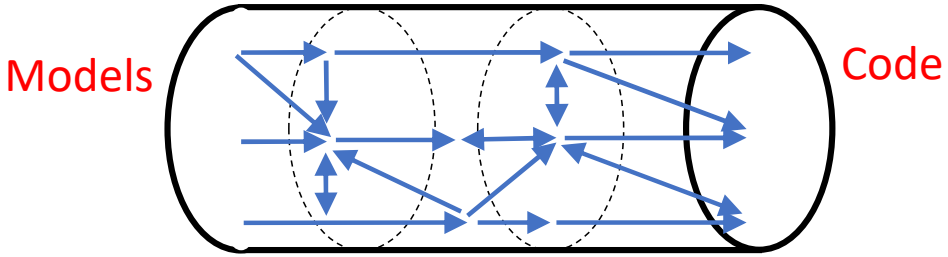
- It would be good to avoid enrichments in later stages, but in practice this is a dream:
  - If you make it possible to adapt subordinate models, *it will happen* without changing the primary model (believe me)
  - And if you forbid it, your developers will sooner or later start to cry:
  - Often, (major) changes are required in dependent artefacts, because developers have to hurry and do not have time to update the primary model (e.g. in late projects)

15

Remember: In a perfect world, a change in an IT-System must always be carried out in a unique primary artefact and all subordinated parts are re-generated. This paradigm can really no longer be carried out, because…

1) Subordinated models are opened for changes, as well, e.g. if in late projects you don‘t have time for a complicated way of to adaptation: Change the primary model, generate along several stages until e.g. the renaming of an attribute arrives in the code.
2) Enrichments in later stages are inevitable, also because of the different expertise of the development team, some working only in the modeling subteam, some others are in love with their code and don‘t care about these modeling freaks.
3) Sometimes we *must* adapt generated artefacts, because we do not agree with (the performance of) the generated code.

Summary of Problems



1. Incrementality
2. Multi-Stage Transformations
3. Bidirectional dependencies on the same level of abstraction
4. Unavoidable inconsistent change in later stages

- 1 and 2 solved by enrichment protection in advanced MDSE tools ✓
- 3 and 4 are the challenges

16

Consequences

- Disillusion:
  - An information system is a network of intensively interacting and (highly) dependent parts. Dependencies may be vertical or horizontal and even in the reverse direction.
  - Non-hierarchical transformation dependencies are not fully understood and hence controlling them (with tools) is a challenge.
- Consequence: We need to …
  - … formally describe requirements for consistency maintenance …
  - … and find operational procedures of supervising frameworks with clear semantics,
- … such that consistency between system parts can *successfully* be maintained[1]

1 Epsilon is a first approach and already includes some important conceptual aspects of the topic. A more practical paper about is Feldmann et al: *Managing inter-model inconsistencies in model-based systems engineering,* SoSym 2019

The text in the slide must not be explained further, but maybe we should comment on the picture with moles: It is taken from the game „Whack-a-Mole“, in which the strike on the head of one mole let‘s another one appear elsewhere. It is sometimes used in research papers as a metaphor for IT-System-Landscapes: Fixing an inconsistency in one component may cause a new one in another system.
Thus the overall goal is to have a super-ordinate controlling mechanism for **global consistency**.
In the sequel, we will investigate this goal for the binary case (two systems to be kept in sync). The multi-ary case is currently a subject of intensive research, e.g. in Patrick Stünkel’s work (“comprehensive systems”) and also in Tim’s PhD project.

From Model Transformations to Model Synchronisations

18

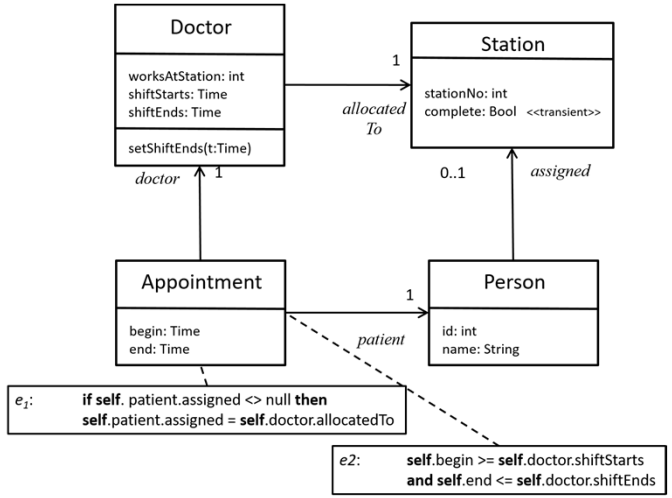
Model Space

- A *Model Space* $\mathcal{m}$ consists of …
  - Models (of a specific type) and …
  - Updates $A \rightarrow A'$ (changes) of model A leading to new model A'.
- An update may either describe, …
  - … which parts of A are removed and what is added, or …
  - … it is an edit log, which records a modeler‘s actions, when changing A to A‘.
- Examples of Model Spaces:
  - All UML-Class Models (together with updates)
  - All possible DB Schemas (formalized e.g. in an appropriate modeling language)
  - All object diagrams for a given class model M
  - *State Charts, Use Cases , BPMN Diagrams, others see Slide 14*
- Any $A \in \mathcal{m}$ is called a *model*
- On the next slide we consider the most important examples of a model space: Instances typed over a class model

19

This is an informal description of Model Spaces (there are more theoretical ones, which we omit here). In order to cope with changes between models as discussed before it is important to have in mind that there are two types of members in a model space: Models **and** Updates

An important Model Space

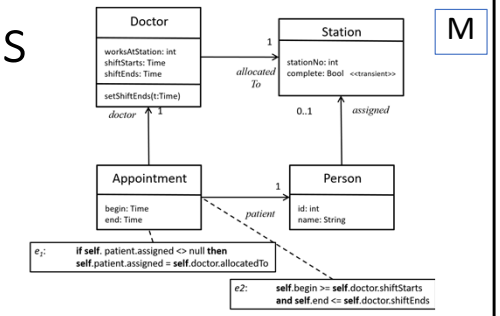


20

Let‘s consider a simple class models as the one shown above. We always have to distinguish between the modeled classes, associations and attributes on the one hand and *additional* **constraints** on the other hand. Constraints may be built-in into UML (multiplicities or annotation like <<transient>>) or are non-UML constraints like e_1, e_2, which have to be formulated in an extra language (e.g. OCL, the Object Constraint Language).

Model Space of (Typed) Instances

Here are two possible instances (times are all a.m.) A_1 and A_2 (all doctors work 9 am–5 pm at station 1)



(A_1, τ_1)	Appointment	Doctor	Person
	1 (10:00 - 11:00)	Carlsen	Comey (Station 1)
	2 (10:00 - 11:00)	Caruana	Hoover (no station assigned)
	3 (11:00 - 12:00)	Caruana	Comey (Station 1)

(A_2, τ_2)	Appointment	Doctor	Person
	1 (08:00 - 09:00)	Carlsen	Comey (Station 1)
	2 (10:00 - 11:00)	Carlsen	Hoover (no station assigned)

← Two instances (two elements of model space)

21

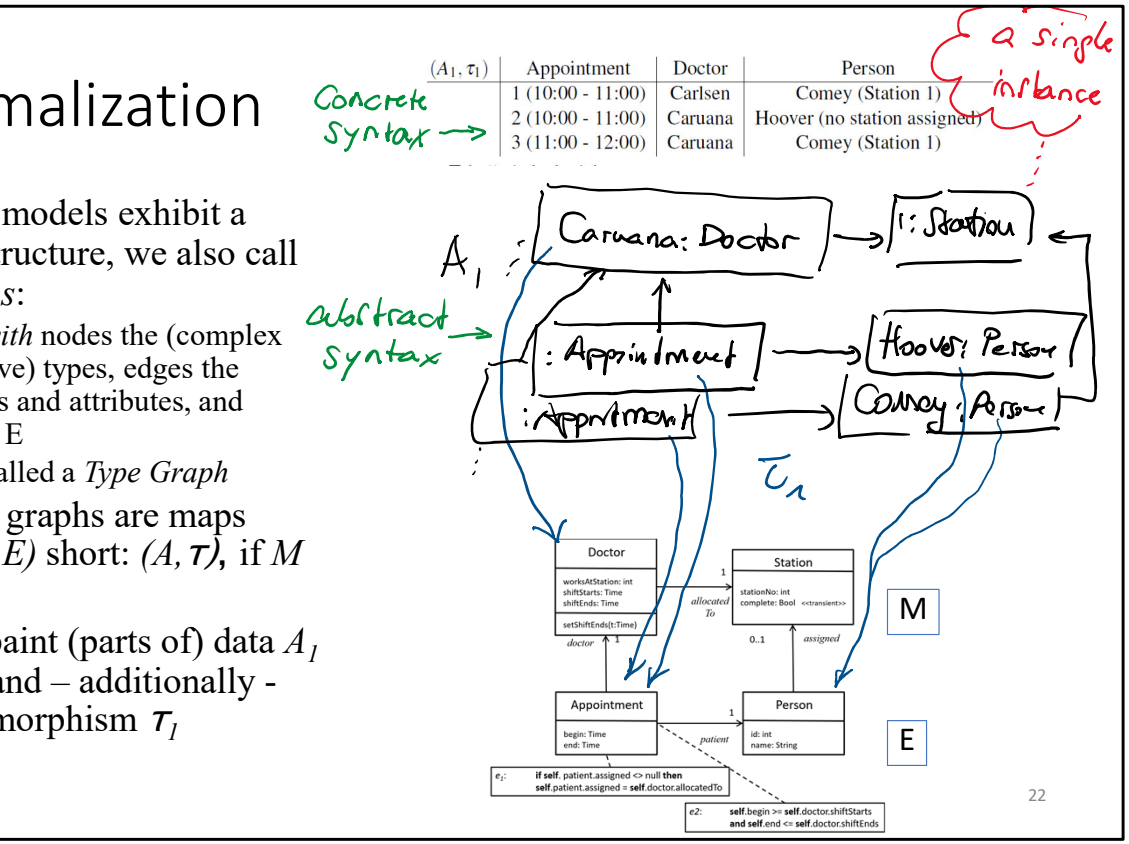
For a model M like the one in the top right corner, we call data that is typed over the model, a (typed) instance of model M. Often in object-oriented languages one single object is also called an instance, but in our case we may have many objects together with references between them, which make up *one* instance.

Note that not only the instance structure, e.g. A_1, is considered, but its typing is additionally mentioned. The typing is a map, which assigns to each element in an instance (e.g. „Carlsen“) its type („Doctor“.) This map is called τ above. Thus we have two maps τ_1: A_1 → M and τ_2: A_2 → M.

Note that two instances are shown, which may or may not be present in the system at a certain point in time. Instances are also called *snapshots* to stress this time dependency.

Formalization

- Since class models exhibit a graphical structure, we also call them *graphs*:
  - Graph M *with* nodes the (complex and primitive) types, edges the associations and attributes, and *constraints* E
  - M is also called a *Type Graph*
- Thus typed graphs are maps $\tau: A \rightarrow (M,E)$ short: (A,τ), if M is known
- Ex.: Let's paint (parts of) data A_1 as a graph and – additionally - the typing morphism τ_1



M is called a type graph, because it provides the typing for instances.
An instance A (e.g A_1) can also be depicted as a graph, i.e. an UML object diagram. We call this representation the *abstract syntax*.
The table is (one possible) representation of A in *concrete syntax.*
The mapping τ is then the typing and the pair (A,τ) is called a typed graph, if the map is τ: A → M. We write τ: A → (M,E) if we want to stress the existence of constraints.

Abstract syntax is machine readable and hence ready for the use in algorithms, whereas concrete syntax is better readable for humans.

Constraints

- Notation $(A, \tau) \models e$, if an instance A satisfies constraint e
- We have

  $(A_1, \tau_1) \models e_1$ and $(A_1, \tau_1) \models e_2$

  but

  $(A_2, \tau_2) \models e_1$ and $(A_2, \tau_2) \not\models e_2$
- … in the hospital example
- Let E be the set of all constraints on M and (A, τ) be an instance, then

  $(A, \tau) \models E$ *(short:* $A \models E$*)*

  if $(A, \tau) \models e$ for all $e \in E$.
  - I.e., in the example, $E = \{e_1, e_2\}$, $A_1 \models E$, but $A_2 \not\models E$
- *We obtain two model spaces:*

1. ***TypedGraphs$^{?}$(M,E) are all instances typed over M***
2. ***TypedGraphs$^{!}$(M,E) are those instances A for which*** $A \models E$

23

The validity of a certain constraint *e* depends **not** only on the structure of A. One also needs to consider the typing τ. Thus satisfaction is written $(A, \tau) \models e$. However, if the typing is known, we also write $A \models e$. In the hospital example $(A_2, \tau_2) \not\models e_2$ because Carlsen starts his work at 0900, whereas the first appointment starts at 0800.

The notation for typed graphs shall symbolize that a constraint may or may not be satisfied and that this is a question („?“), whereas in TypedGraphs$^{!}$(M,E) we know (!) that all constraints are satisfied.

One more example

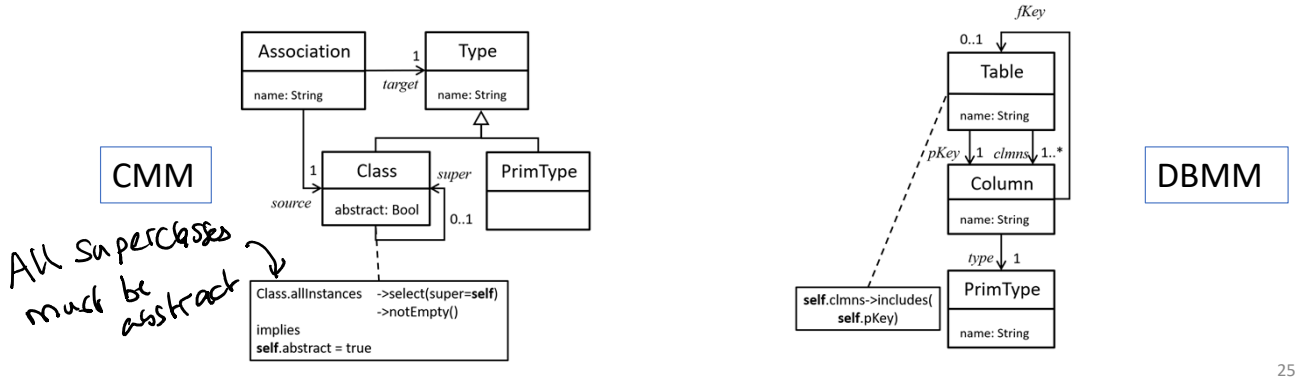
- Let $\mathcal{M}_1$ be the model space of all class models and $\mathcal{M}_2$ be the model space of all data models
- Why are elements of $\mathcal{M}_{1/2}$ also “typed graphs”?
- What are the respective “model”s?
- Answers: see next slide
  - …
  - …

24

Class Models must obey certain rules, namely the ones defined in its metamodel. The figure shows an excerpt of a sample metamodel for class diagrams, in which certain constraints are integrated. Can you guess what the constraint in the left figure means? Well, it says that each superclass must be abstract.
In the same way there is (again a part of) a metamodel for data models on the bottom right. What is the meaning of the constraint? Clearly, it say that the primary key column must appear in the list of all columns of a table.
Keep in mind for later that the two metamodels show certain **commonalities**, e.g. they both specify the existence of primitive types.
The next page show a similar example, but now “one level below”. Models are class models and their instances are snapshots of runtime objects.

One more example

- Let $\mathcal{M}_1$ be the model space of all class models and $\mathcal{M}_2$ be the model space of all data models
- Why are elements of $\mathcal{M}_{1/2}$ also "typed graphs"?
- What are the respective "model"s?
- Answers: Models for instances (i.e. class models) are *Metamodels CMM* and *DBMM,* i.e.
  - $\mathcal{M}_1$ = TypedGraphs^{I}(CMM)
  - $\mathcal{M}_2$ = TypedGraphs^{I}(DBMM)



25

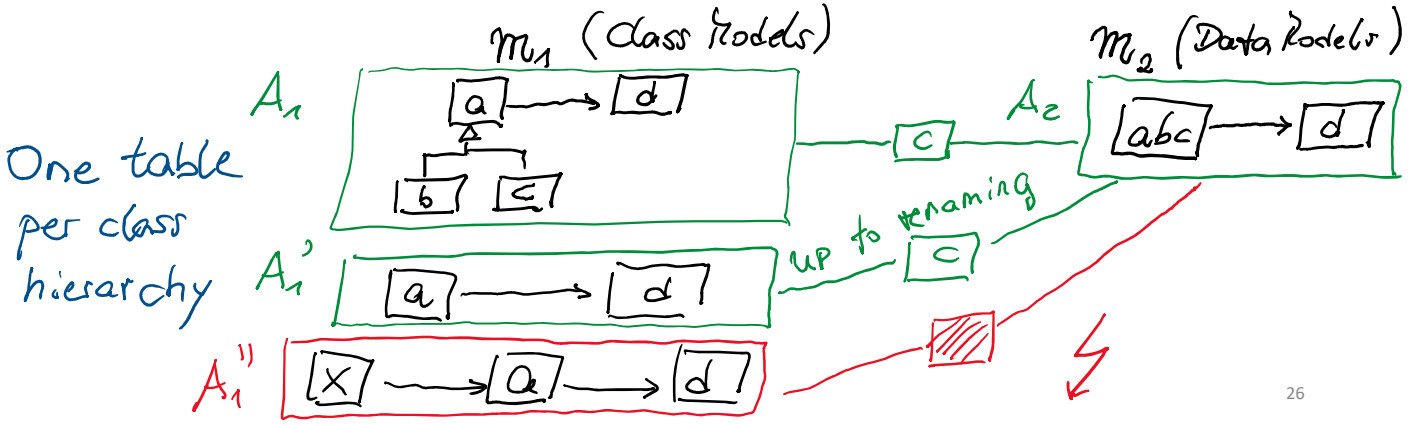
Class Models must obey certain rules, for instance the ones defined in its metamodel. The figure shows an excerpt of a sample metamodel for class diagrams, in which certain constraints are integrated. Can you guess what the constraint in the left figure means? Well, it says that each superclass must be abstract. In the same way there is (again a part of) a metamodel for data models on the bottom right. What is the meaning of the constraint? Clearly, it say that the primary key column must appear in the list of all columns of a table.
Keep in mind for later that the two metamodels show certain **commonalities**, e.g. they both specify the existence of primitive types.
The next page show a similar example, but now "one level below": models are class models and their instances are snapshots of runtime objects.

Consistency Relation

- Consistency maintenance of correlated system components requires to understand the interplay of two or more model spaces
- Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be model spaces, *a Consistency Relation* is a binary relation on objects of $\mathcal{M}_1$ *and* $\mathcal{M}_2$ i.e.

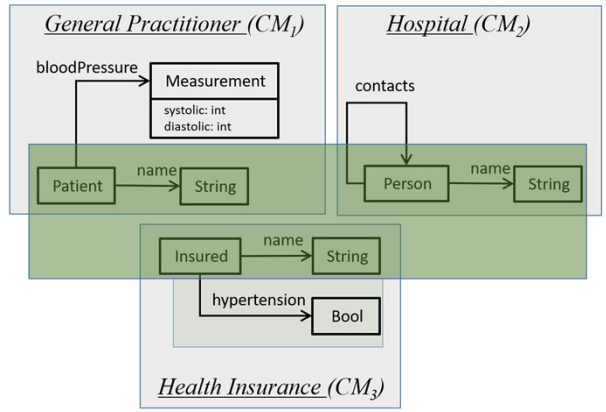
$$C \subseteq \mathcal{M}_1 \times \mathcal{M}_2$$

- $(A_1, A_2) \in C$ means that A_1 and A_2 are consistent.
- Example (OR-Mapping):



Without further ways of defining consistency, the consistency relation is just the set of all consistent pairs. All pairs not in C are inconsistent. The figure shows the situation of an OR Mapping in which a class model is reflected in the database by having one table per class hierarchy, i.e. all attributes of classes a, b, c in A_1 are stored in one table (called abc in A_2). The rule "One Table per class Hierarchy" yields consistence of the pairs (A_1, A_2) and (up to maybe a different name for the table abc) (A'_1, A_2), but **not** (A''_1, A_2), because there is no table in A_2 for class x in A''_1.

Model Spaces of Instances and Consistency Variants



- systolic/diastolic = min/max
- hypertension: sys > 140, dia > 80

Patients with hypertension (i.e. patient.bloodpressure > (140,80)) must have at least one contact person!

Consider Model Spaces $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$ the typed instances over CM_1, CM_2, CM_3

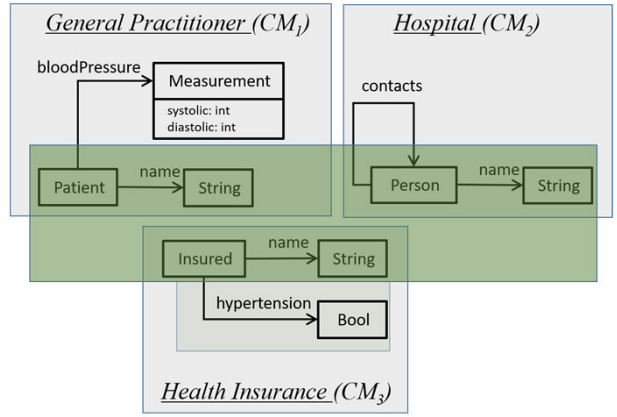
- I.e. $\mathcal{M}_{1/2/3}$ = TypedGraphs$^!$($CM_{1/2/3}$)
- Consistency between models of $\mathcal{M}_1$, $\mathcal{M}_2$ due to person names of same real-world person
- Consistency between models of $\mathcal{M}_1$, $\mathcal{M}_3$ due to person names and hypertension info
- Consistency between models of $\mathcal{M}_2$, $\mathcal{M}_3$ due to person names and red constraint

27

We see here an example of three model spaces. Attention: What you see are not elements of the model spaces. You see the controlling class models which specify the elements of the spaces, namely the instances typed over $CM_{1/2/3}$.
Again attention: We still call $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$ ***model*** space, although their elements are typed ***instances*** over $CM_{1/2/3}$.
In the sequel we often get rid of the distinction between *model* and *instance* and **call everything a *model*** (Adrian will be happy …).

Note that we have to distinguish between **one** legal instance w.r.t. a single model, e.g. a patient without name may not be allowed as an instance typed over CM_1, and **three** legal instances w.r.t. to the *inter model constraint* (colored in red). In the sequel, we will analyse how to express comprehensive validity of instances (A_1,A_2,A_3) in $\mathcal{M}_1$ x $\mathcal{M}_2$ x $\mathcal{M}_3$

Informational (A-)Symmetry



Patients with hypertension (i.e. patient.bloodpressure > (140,80)) must have at least one contact person!

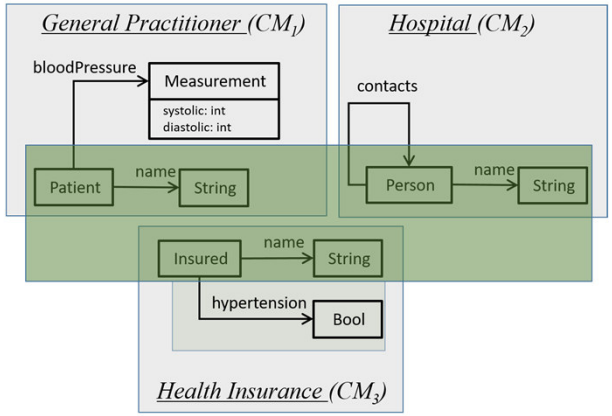
Informational symmetry:

- m_1, m_2 each have private parts not known to the other (=>Info Symmetry)
- For which pair is this also the case? m_2, m_3
- And where is information asymmetrically distributed? m_1, m_3, because in CM_1 the values systolic/diast. are private whereas the hypertension information can be calculated from the information in CM_1

28

Informational asymmetry is a concept to cognitively arrange the data at hand, see next Slide.

Informational (A-)Symmetry



Informational symmetry:

- $\mathcal{M}_1$, $\mathcal{M}_2$ each have private parts not known to the other (=>Info Symmetry)
- For which pair is this also the case? Answer: CM_2 and CM_3
- And where is information asymmetrically distributed? Answer: CM_1 and CM_3, because hypertension-value can be calculated from bloodPressure-values

29

Informational asymmetry is an imbalance of contained information, when one space contains more information than the other.

Informational symmetry means that both spaces each possess information that the other does not know.

Information not known to any other space is called *private* information.

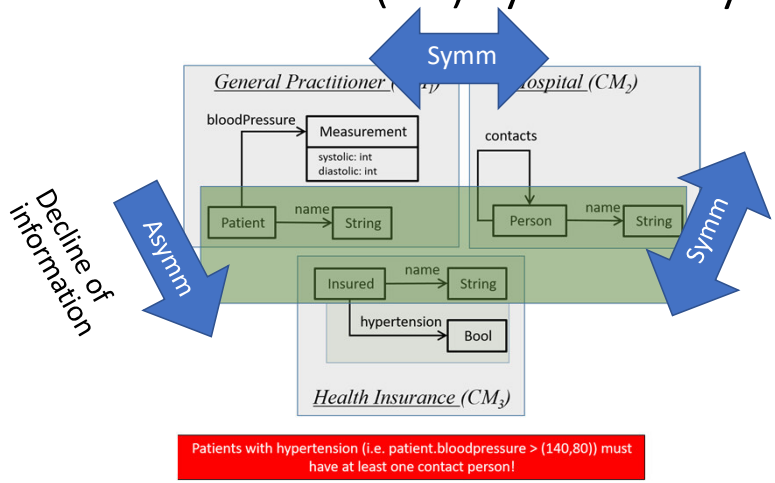
We have two cases of informational symmetry:

- Instances over $CM_{1/2}$ have their own private parts: A_1 in $\mathcal{M}_1$ contains blood pressure data not known in $\mathcal{M}_2$. A_2 in $\mathcal{M}_2$ contains contact information not known in $\mathcal{M}_1$.
- Instances over $CM_{2/3}$ also have their own private parts: A_2 in $\mathcal{M}_2$ knows nothing about hypertension, A_3 in $\mathcal{M}_3$ knows nothing about contacts.

And one case of informational asymmetry:

- Instances over CM_1 contain specific blood pressure values, from which the hypertension predicate in CM_3 can be derived. However, we cannot restore the concrete values in CM_1, if we only know the value of this predicate. That's why we call this situation *asymmetric.*

Informational (A-)Symmetry



Summary:

1. Info asymmetry means that changes can easily be propagated in one direction, but information is lost. In a propagation in the other direction, lost information cannot directly (without further knowledge) be re-established by a generator function.
2. Info Symmetry means that loss of information occurs in both directions and re-establishing is not directly possible in both directions

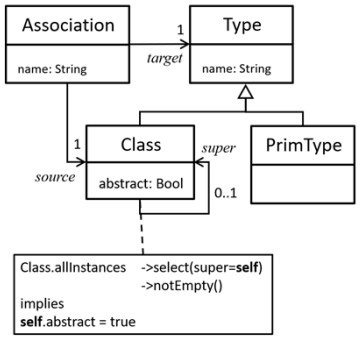
Private and common parts

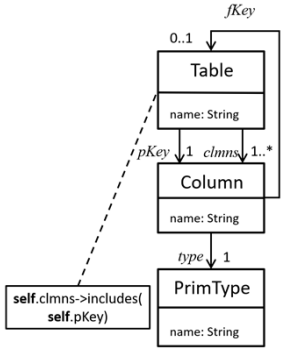
Exercise: Determine (informally) concepts that are

… *private*,

… *common* in the following two metamodels,

… and hence information of one model that is known / unknown to the other, when establishing an OR Mapping





Private and common parts

Answers to the Exercise

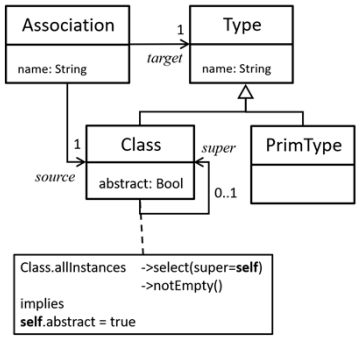
Private for instances typed over M_1

- Super-Relation
- Abstract
- The constraint

Private for instances typed over M_2

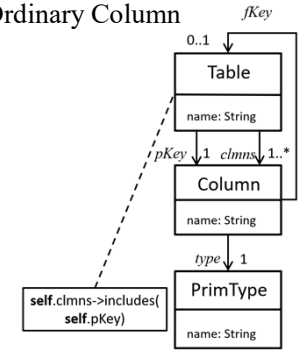
- Primary Keys

M_1



Commonalities

- Primitive Types
- Class and Table and their names
- Associations with target a class, Foreign Key
- Associations with target a PrimType, Ordinary Column

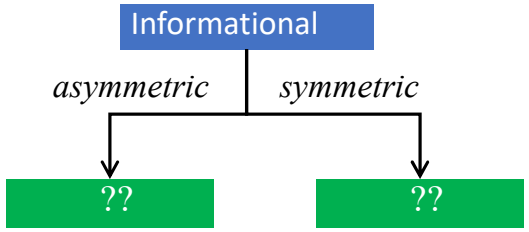


M_2

Goal

- *Definition:* Let $(A_1,A_2) \in C$ and $A_1 \rightarrow A_1'$ be an update. We *restore consistency of the (possibly inconsistent) model pair* $(A'_1, A_2) (\notin C)$, by specifying an update $A_2 \rightarrow A_2'$, such that $(A_1',A_2') \in C$.

- **Goal:** For model spaces $\mathcal{M}_1$ and $\mathcal{M}_2$, define a **conceptual base**, which enables consistency restoration of model pairs, avoids loss of information, and differentiates between informational symmetry and asymmetry



33

Bidirectional Model Synchronization

- Aka *Bidirectional Transformation (BX)*
- Arrange matters such that a single specification of the relationship between the model spaces may serve simultaneously for consistency restoration
- History: Origin is the view-update-problem in databases (1980)
  - Elements *v* of the view are computed from elements *s* of the source via a given function (usually called get)
  - Consistency of *(s,v)* is defined by *(s,v)* $\in C$ if and only if *v* = ***get****(s)*
  - Source-Updates *s* → *s'* can easily be handled*:* consistency is guaranteed by recalculating *v'*:= ***get****(s')*, ***but View-Updates are a problem****, because usually there is no inverse of* ***get*** .
  - It soon became clear, that not only (informationally) *asymmetric cases occur, but also symmetric ones* → e.g. in MDSE as we pointed out before
- Let's study the following concepts along a 3rd example (the famous composer example, see [1])

34

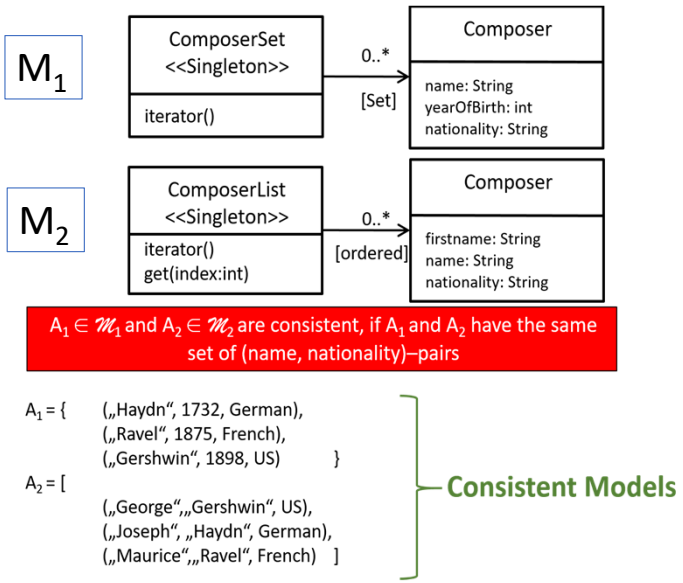
Turning from the transformational view to the relational one means to specify relationships (in both directions) and use it for synchronization issues. An attempt to make this possible is the QVT-R language (Query View Transformation Relational). Later, we will come back to implementations like this or similar ones.

The view-update problem deals with two spaces – the source and the view, elements of the latter being derivable from elements of the former. A simple example is the space of all pairs (x,y) of numbers on the one hand (the source) and the space of all numbers z (the view). Consistency between elements of the model space is defined by ((x,y),z) ∈ C, if and only if x + y = z. If someone updates the state (x,y) of the source to (x',y'), consistency can easily be restored by computing z' := x' + y'. The new consistent pair is then ((x',y'),z').

However, if someone changes the state of the view from z to z' and the state of the source is (x,y) (thus z = x + y), there is no unique way of restoring consistency. E.g. ((x,y),z') =

(keeping the first component), maybe (4,3) (keeping the second), maybe (3,4) (keeping the difference between components), etc

A typical symmetrical situation



- Let $\mathcal{M}_1$ = TypedGraphs$^!$(M_1) and $\mathcal{M}_2$ = TypedGraphs$^!$(M_2) be the model spaces of instances typed over the resp. models on the left
- Why is the situation informationally symmetric?
  - yearOfBirth in M_1, firstname in M_2

35

We have two spaces $\mathcal{M}_1$ and $\mathcal{M}_2$. Models of the first space are sets of composers (the class ComposerSet is nor really important, also iterator() only suggests that the elements of the set are accessible). Models of the second space are lists of composers (again the class ComposerList is nor really important, and get(index:int) only shows that the elements are ordered).
This is clearly a symmetric situation, because yearOfBirth is unknown in $\mathcal{M}_2$ and firstname is unknown in $\mathcal{M}_1$. The definition of the consistency relation has to be read carefully: It says A_1 and A_2 are consistent, if there is the same set of (name,nationality)-pairs in A_1 and A_2, i.e. if you

- … forget all years of birth in A_1 and …
- … forget firstnames in A_2 **and** convert the resulting (name,nationality)-pairlist into a set (merging all identical pairs in the list into one entry)

then these two sets must be equal.

German)} (consisting only of one composer) in $\mathcal{M}_1$ and the list [(Johann Sebastian, Bach, German), (Carl Phillip Emanuel, Bach, German)] in $\mathcal{M}_2$, then they are consistent, because the (name,nationality)-pairlist [Bach, German), (Bach, German)] consisting of two identical entries in $\mathcal{M}_2$ collapses to the set {(Bach, German)}, because duplicate entries in a set are prohibited. This is, however, the resulting set in $\mathcal{M}_1$, which proves consistency. Thus it is possible that two models are consistent although they do not contain the same number of composers.

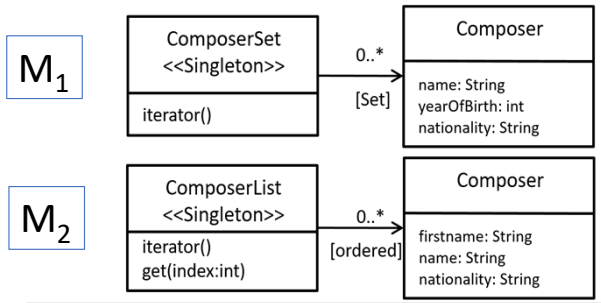
Exploring Consistency Restoration

- Let $A_1 \in \mathcal{M}_1$ and $A_2 \in \mathcal{M}_2$ be consistent
- Then a new composer is added in A_1, yielding an update $A_1 \rightarrow A'_1$
- What input data is needed for a restoration?

- If the data in A_2 would just be overwritten with the data in A_1, firstnames would be lost
- → Additional input should also be A_2, such that private information from A_2 is preserved
- → For *forward restoration* we need a function $\vec{R}$

$$\vec{R} : \mathcal{M}_1 \times \mathcal{M}_2 \rightarrow \mathcal{M}_2$$

Enables preservation of private information in A_2



$A_1 \in \mathcal{M}_1$ and $A_2 \in \mathcal{M}_2$ are consistent, if A_1 and A_2 have the same set of (name, nationality)–pairs

A_1 = { („Haydn", 1732, German), („Ravel", 1875, French), („Gershwin", 1898, US) }

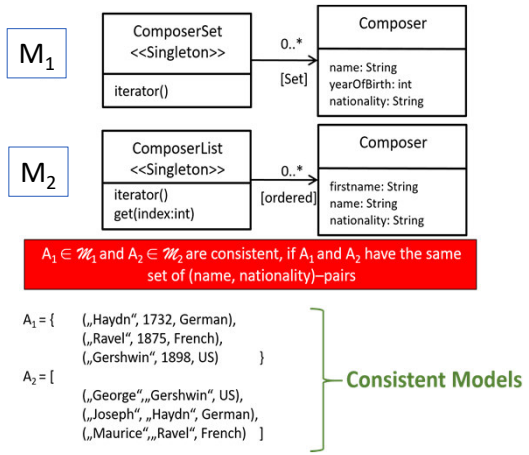
A_2 = [(„George",„Gershwin", US), („Joseph", „Haydn", German), („Maurice",„Ravel", French)]

Consistent Models

If there are private parts in $\mathcal{M}_2$, then a „forward restoration" must take into consideration the current contents (state) of A_2 in order not to destroy this private data. Thus the function which carries out forward restoration must have two input parameters, the new model A'_1 and the old model A_2 and outputs the updated model A'_2: I.e. $A'_2 = \vec{R}(A'1, A_2)$.

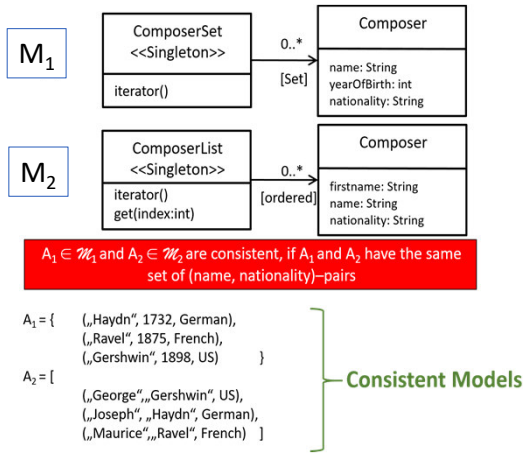
Symmetric Lens

- But informational symmetry also requires a reverse mechanism, thus we define:



Symmetric Lens

- But informational symmetry also requires a reverse mechanism, thus we define:



A *symmetric lens* consists of

1. A consistency relation C $\subseteq m_1 \times m_2$
2. A forward restoration $\vec{R}$: $m_1 \times m_2 \rightarrow m_2$
3. A backward restoration $\overleftarrow{R}$: $m_1 \times m_2 \rightarrow m_1$

38

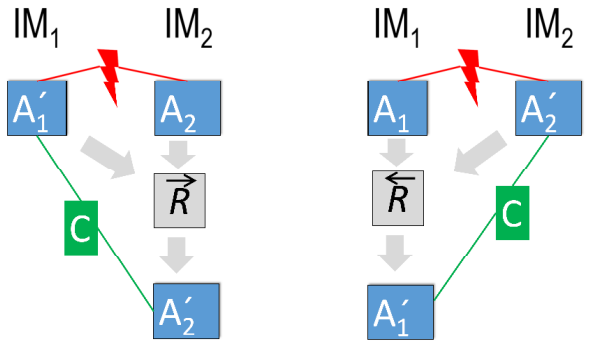
Our first important definition – in the Java Demo a symmetric lens is implemented to handle the second situation.

The forward restoration has input the current state of m_2 and the new state in m_1 and produces a new state in m_2
The backward restoration has input the current state of m_1 and the new state in m_2 and produces a new state in m_1

Symmetric Lens

A *symmetric lens* consists of

1. A consistency relation C ⊆ m_1x m_2
2. A forward restoration $\vec{R}$: m_1x m_2 → m_2
3. A backward restoration $\overleftarrow{R}$: m_1x m_2 → m_1



39

There are variants, in which (e.g. forward restoration) has as input the complete update A_1 → A'_1 and vice versa for backward restorations, because, sometimes it is important to know in detail, how the change from A_1 → A'_1 was carried out ("deltas do matter", Z. Diskin).

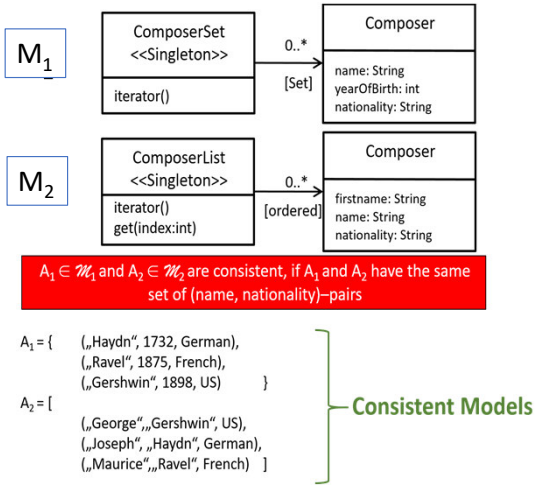
Example: We could (1) **delete** a composer **and later add** the same but with a different yearOfBirth in A_1 or (2) **change** the year of Birth. If we restore consistency immediately after each update action, we must likewise **delete and add** in A_2 in case (1), but we can leave all information in A_2 unchanged in case (2).
In case (1) this will destroy the firstname of the deleted composer, which can not be added later on without further policies.

Example

$\vec{R} : \mathcal{M}_1 \times \mathcal{M}_2 \rightarrow \mathcal{M}_2$ in the composer example can be defined by

- Input (A'_1, A_2)
- For each object *(name, yob, nat)* $\in A'_1$ do:
  - if there is no object *(fname, name, nat)* in A_2 add *(“”, name,nat)* at the end of the list in A_2 (*)
- For each object *(fname, name, nat)* $\in A_2$ do: if there is no object *(name, yob, nat)* $\in A'_1$
  delete (fname, name, nat) from A_2 (**)

In a similar way $\bar{R}: \mathcal{M}_1 \times \mathcal{M}_2 \rightarrow \mathcal{M}_1$ can be implemented



It can be shown that this algorithm yields a correct result, see the exercises.

By the way: We see here the effects in the scenario of the previous page. If we **delete** a composer **and later add** the same but with a different yearOfBirth in A_1, then (**) deletes information that can not be re-established in (*). Whereas all information is left unchanged in the case of a change of yearOfBirth.

Symmetric Lenses

- Of course, $(C, \vec{R}, \overleftarrow{R})$ could have been implemented differently
- Which implementation is correct? What is correctness?
- How can we express the correctness requirement for symmetric lenses in terms of the used functions $\vec{R}, \overleftarrow{R}$?
- And are there other requirements?

41

Of course other implementations are possible for the composer example. E.g. if there are two triples *(name, yob, nat), (name, yob', nat)* $\in A'_1$ and there is no object *(fname, name, nat)* in A_2, the solution adds *(“”, name,nat)* once at the end of the list in A_2 *(why?)*. However we could also have added *(“”, name,nat)* twice.

Symmetric Lenses

- $(C, \vec{R}, \overleftarrow{R})$ could have been implemented differently
- Which implementation is correct? What is correctness?
- How can we express the correctness requirement for symmetric lenses in terms of the used functions $\vec{R}, \overleftarrow{R}$?
- And are there other requirements?
  - Restorations must restore, i.e. for all $A_1, A'_1 \in \mathcal{M}_1$ and $A_2, A'_2 \in \mathcal{M}_2$

$$(A'_1, \overrightarrow{R}(A'_1, A_2)) \in C$$
$$(\overleftarrow{R}(A_1, A'_2), A'_2) \in C$$

Correctness

  - Restorations are idle, if there is no cause for repair:

$$(A_1, A_2) \in C \Rightarrow \overrightarrow{R}(A_1, A_2) = A_2$$
$$(A_1, A_2) \in C \Rightarrow \overleftarrow{R}(A_1, A_2) = A_1$$

Hippocraticness*

*Hippocrates: „No causeless therapies“

42

Correctness is a must!
But Hippocraticness also, because we should not harm healthy situations - actually the oath of Hippocates includes the principle of **non-maleficence** (no maleficient actions w.r.t. the health of a patient) and swearing a modified form of the oath remains a rite of passage for medical graduates.

For Software Engineering this means that updates should be avoided, if there is already consistency, because they may lead to unforced errors!

Bidirectional Model Synchronization

- Composers were informationally symmetric: Forward restoration forgets year of birth, Backward restoration forgets firstname
- But there were examples with informational *asymmetry*!
- One of them is the famous „view update“ problem
  - $\mathcal{M}_1 = \mathbb{Z} \times \mathbb{Z}$, $\mathcal{M}_2 = \mathbb{Z}$
  - $C = \{((m,n), m+n) \mid m,n \in \mathbb{Z}\}$
  - Thus elements of $\mathcal{M}_2$ are a **view** calculated from **source** $\mathcal{M}_1$
  - Consistency Maintenance
  - Forward restoration: $A'_1 = (m',n')$, $A_2 = k$. Just need to replace k by $m'+n'$
    - $\vec{R}: \mathcal{M}_1 \rightarrow \mathcal{M}_2$ is enough
  - Backward restoration:
    - Given consistent pair $(A_1, A_2) = ((m,n),k)$, i.e. $k = m+n$
    - How to restore consistency, if k changes?
  - → *Restoration after view update is not uniquely determined*
  - Example: ((2,3),7) → Solutions for A'_1: (2,5), (4,3), (3,4), (-17,23), (6,0) …

43

In the asymmetric case, forward restoration must not consider the old state in $\mathcal{M}_2$. Thus the lens becomes simpler.
However, we need A_1 as input for backward restoration: $\overleftarrow{R}: \mathcal{M}_1 \times \mathcal{M}_2 \rightarrow \mathcal{M}_1$ remains.

Source and View Asymmetry

- Traditionally the involved spaces are no longer called $\mathcal{M}_1$ and $\mathcal{M}_2$, but (S)ource and (V)iew
- More examples
  - General database views
  - OR-Mapping (not regarding additional database information like indices or primary keys)
  - toString()-Method
- And – also traditionally – functions $\vec{R}, \overleftarrow{R}$ are renamed:
- View calculation: „get“ the view, i.e. function

  get: S → V
- This naturally determines the consistency relation:

  $C = \{(s, get(s)) \mid s \in \mathcal{S}\}$
- Backward propagation („put“ the view back to the source):

  put: S x V → S

44

A database view is usually a projection, i.e. given a table line as input, it outputs only some values of certain columns in the line. Thus it enjoys the same asymmetry as the previous example. The OR-Mapping becomes asymmetric if we do not considered private parts of the data model, e.g. primary keys. In this case, there is a unique mapping procedure, which converts a class model into database tables. Note that additional information must be provided if primary key columns shall be generated, e.g. type and name of the primary key. Vice versa, a reverse transformation is difficult, because there are several ways of arranging classes w.r.t. inheritance relations. We will discuss more on that a little bit later.
toString() also offers a certain view on an object. There are however many objects which have the same textual representation e.g. two persons with first names and last names and toString() = first names + last names may be (John Oscar, Petterson) and (John, Oscar Petterson) both being represented by “John Oscar Petterson”

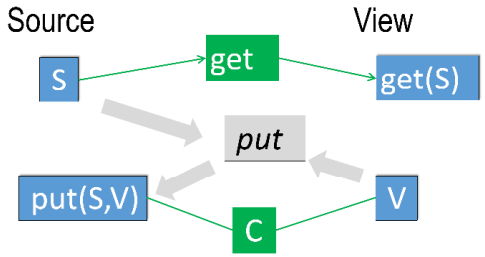
Asymmetric Lens

An *asymmetric lens* consists of

1. A forward restoration get: S → V (get the view)
2. A backward restoration put: S x V → S (put the view back to the source),

which naturally determines the consistency relation

$$C = \{(s, get(s)) \mid s \in S\}$$



45

Asymmetric Lens

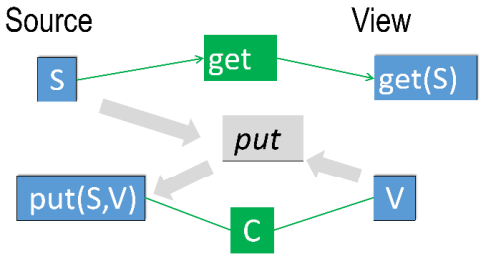
An *asymmetric lens* consists of

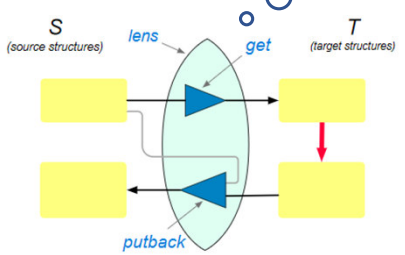
1. A forward restoration get: S → V (get the view)
2. A backward restoration put: S x V → S (put the view back to the source),

which naturally determines the consistency relation

$$C = \{(s, get(s)) \mid s \in S\}$$

The term "lens" stems from the original work by B. Pierce





46

Exercise

- Can an asymmetric lens AL *uniquely* be interpreted as a symmetric one?
  - $\vec{R}(s,v) := get(s)$, $\overleftarrow{R}(s,v) := put(s,v)$
  - This defines SL(AL)
- But not vice versa:
  - Because it is unclear, how to define *get(s)* from $\vec{R}$, since there are different choices for the 2nd argument: $get(s) := \vec{R}(s,?)$
- Let *AL* be an asymmetric lens, then denote with *SL(AL)* the resulting symmetric lens
- Now we can say that an AL is correct and hippocratic, if and only if *SL(AL)* is such:
- Let's spell that out:
  - First eq. for correctness for symm lens => $\{(s, get(s)) \mid s \in S\} \subseteq C$
  - 2nd eq. for correctness => $(put(s,v), v) \in C$ (*)
  - 1st eq. of hippo $(s,v) \in C \Rightarrow get(s) = v \Rightarrow \{(s, get(s)) \mid s \in S\} = C$

  => With (*): $get(put(s,v)) = v$ (i.e ***correctness***)
  - The corresponding axiom for Hippocraticness can similarly be derived

Symm Lens

$\vec{R} : m_1 \times m_2 \rightarrow m_2$
$\overleftarrow{R} : m_1 \times m_2 \rightarrow m_1$

$(A_1', \vec{R}(A_1', A_2)) \in C$
$(\overleftarrow{R}(A_1, A_2'), A_2') \in C$

$(A_1, A_2) \in C \Rightarrow \vec{R}(A_1, A_2) = A_2$
$(A_1, A_2) \in C \Rightarrow \overleftarrow{R}(A_1, A_2) = A_1$

Asymm Lens

get: S → V
put: S x V → S

put(s, get(s)) = s (Hippocr.)

get(put(s,v)) = v (Correctn.)

47

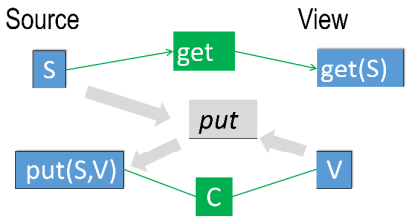
Correctness: The first equation of correctness states that {(s,get(s)) | s ∈ S} ⊆ C (*).
The 2nd equation becomes (put(s,v),v) ∈ C, hence **get(put(s,v)) = v**

Hippocraticness: The first equation of hippocraticness says that (s,v) ∈ C implies get(s) = v, i.e. C ⊆ {(s,get(s)) | s ∈ S} (**)
The second equation yields (s,v) ∈ C, (i.e. v = get(s) by (**)) implies put(s,v) = s, thus **put(s,get(s)) = s**

It is interesting to see that the consistency relation already postulated for ALs automatically arises from hippo and correctness of SLs by (*) and (**)!!

A well-known Asymmetric Lens*)

- java.util.Map<K,V>
  - Source S = All pairs *(m,k)* where *m* is a map object and *k:K* is a key
  - V as above (the type of the values of the map)
  - get: S → V :
    - get(m,k) := *m.get(k)* (⊥, if no mapping for the key exists)
  - put: S x V → S
    - put((m,k), v):= *m.put(k,v)* → results in a changed map object *m'* and also returns the key *k*, whose value changed, i.e. *(m',k) = m.put(k,v)*
  - Question: Is this lens definition correct/hippocratic?
    - Correctness:
    - Hippo:

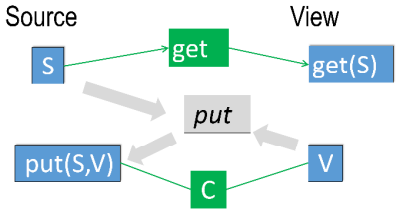


*) At least for Java Programmers

48

In order to consider Java Maps as Lenses, we must convert the object oriented view to the functional view: A message op(a_1,…,a_n) to object x, i.e. x.op(a_1,…,a_n) changing state of x to x' becomes the call of a function: op(x,a_1,…,a_n) with return value x'. Then there is the astonishing parallel of get/put in lenses with the get/put of maps, because m.get(k) becomes get(m,k) and m.put(k,v) becomes put((m,k),v) with return value the new state m' of the map. Formally one only has to add the fact that put also returns the key, whose value was updated. Moreover, V has formally to be enhanced with ⊥, if no value for key k exists.

A well-known asymmetric Lens*)



- java.util.Map<K,V>
  - Source S = All pairs *(m,k)* where *m* is a map object and *k:K* is a key
  - V as above (the type of the values of the map)
  - get: S → V :
    - get(m,k) := *m.get(k)* (⊥, if no mapping for the key exists)
  - put: S x V → S
    - put((m,k), v):= *m.put(k,v)* → results in a changed map object *m'* and also returns the key *k*, whose value changed, i.e. *(m',k) = m.put(k,v)*
  - Question: Is this lens defintion correct/hippocratic?
    - Correctness: *get(put(s,v)) = get(put((m,k),v)) = get(m.put(k,v)) = m'.get(k) =(!) v*
    - Hippo: Let *s = (m,k)* then *put(s,get(s)) = put((m,k),get(m,k)) = put((m,k), m.get(k)) = (m,k)* (no change!) = *s*
    - (The case *m.get(k)* = ⊥ requires ignoring map-entries (k, ⊥))

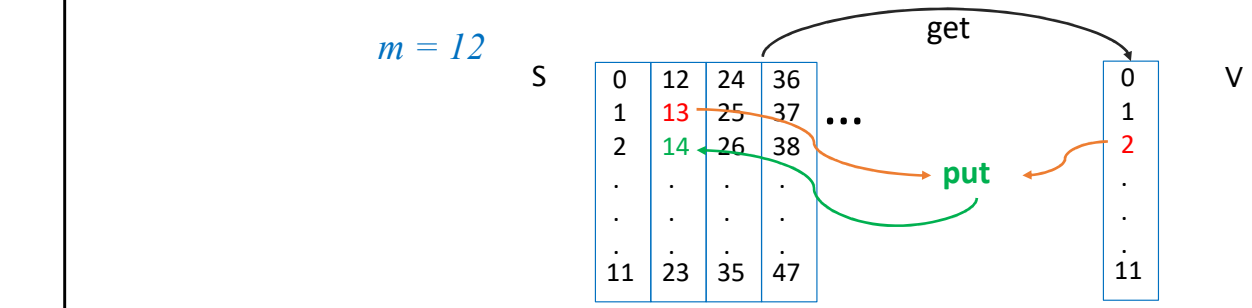
*) At least for Java Programmers

49

There are programming languages into which lenses are built-in (e.g. Rust)

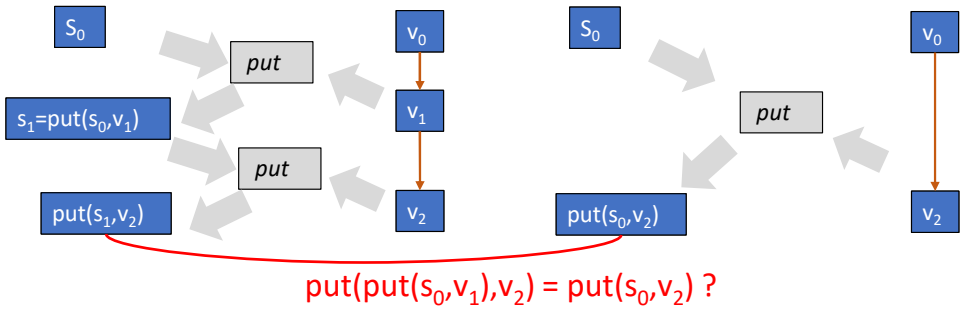
Another Example

- Modulo Arithmetic
  - Let $m \in \mathbb{N}$ (the module)
  - S = (Positive)Integer, V = {0,1,…, m-1}
  - get: S → V is defined by *get(n) := n mod m*
  - put: S x V → S is defined by $put(n,j) = (n/m)*m + j$ for any $0 \leq j < m$ (where *n/m* is Integer division)
  - Correctness: $get(put(n,j)) = get((n/m)*m + j) = j$
  - Hippocraticness: $put(n,get(n)) = (n/m)*m + n \bmod m = n$



More requirements

- *History ignorance*
  - An update sequence in the source with intermediate synchronisation …
    - $(s_0 \rightarrow s_1)$, sync: v_1=get(s_1), $(s_1 \rightarrow s_2)$, sync: v_2= get(s_2)
  - … should yield the same effect than synchronising once in the end …
    - $(s_0 \rightarrow s_1)$, $(s_1 \rightarrow s_2)$, sync: v'_2= get(s_2)
  - … i.e $v_2 = v'_2$!
  - This is, of course, always true, because *get(s_2)* is unique
- But what about update sequences *of the view?* :

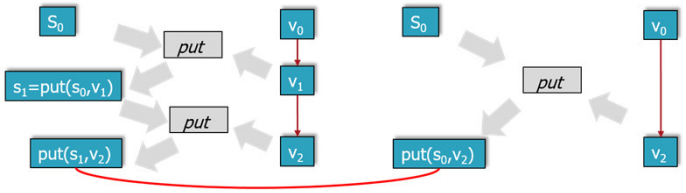


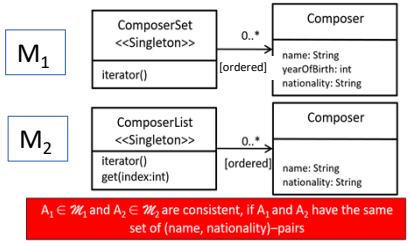
51

History ignorance is an important property, because in a distributed system there may be many subsequent updates in one of the systems. In certain situations it is desirable to restore consistency only once in the end (variant on the right side) instead of continuously restoring the source after each update (variant on the left side) which would cause heavy network traffic. But this is only possible, if the results in the two variants coincide.

The BX freaks call this property the (famous) **put-put law**

Example





- Assume in the composer example S = $\mathcal{M}_1$ maintaining composers also as lists and V = $\mathcal{M}_2$ not containing firstnames. Let get and put are defined elementwise

get(name, yob, nat) = (name, nat)

and

put((name,yob,nat), (name',nat')) = (name',yob,nat')

put defines yob = 0000, if composer is added in a model in $\mathcal{M}_2$

Is it history ignorant?

No it's not, because

→ you may have (Bach, 1685, German) in $\mathcal{M}_1$ and (Bach, German) in $\mathcal{M}_2$. A sequence of deletion, restoration, insertion of (Bach, German) in $\mathcal{M}_2$ leads to (Bach, 0000, German) in $\mathcal{M}_1$

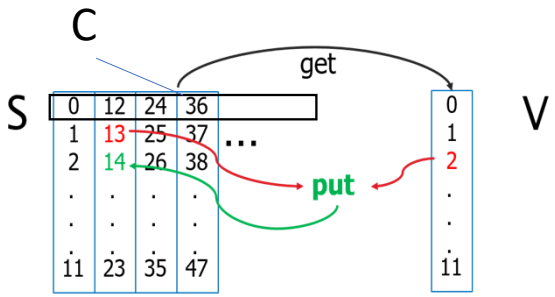
→ But deletion, insertion of (Bach, German), restoration in $\mathcal{M}_2$ will leave the model in $\mathcal{M}_1$ unchanged due to hippocraticness

52

We see that already this simple example lacks history ignorance – so it is not a general requirement. We will see later that – unfortunately - this requirement is fulfilled only in very rare cases.

Constant Complements

- Suppose $S = V \times C$ for some set C (1)
  - *get(v,c) = v*
  - *put((v,c),v') = (v',c)*
  - Correctness:
    - get(put((v,c),v')) = get(v',c) = v'
  - Hippocraticness
    - put((v,c),get(v,c)) = put((v,c),v) = (v,c)
- Remember modulo arithmetic
  - Here $S = V \times C$ (more precisely $S \cong V \times C$, i.e. both sets are in 1:1 correspondence) with $C = \{0,12,24,36, \ldots\} \cong \mathbb{N}$, i.e. $S \cong V \times \mathbb{N}$
- Given an asymmetric lens as in (1). It is history ignorant!!!
  - $put(put((v_0,c_0),v_1),v_2) = put((v_1,c_0),v_2) = (v_2,c_0)$
  - $put((v_0,c_0),v_2) = (v_2,c_0)$
  - → $put(put((v_0,c_0),v_1),v_2) = put((v_0,c_0),v_2)$!
- The set C is called a complement (of V in S). Because of the invariance of c_0 in these computations, the property is called *constant complement property*



53

The modulo arithmetic example is a typical example for the constant complement property: E.g. for m = 12, each element *n* of the natural numbers can be decomposed into the two values *n mod m* and *(n/m)*m* (the biggest multiple of *m* still smaller or equal to n). E.g. 35 can be encoded as the pair (11,24). Thus $\mathbb{N} \cong V \times \{0,12,24, \ldots\}$ and in this encoding *get(v,c) = v* and *put((v,c),v') = (v',c).*

This slide shows that the constant complement property yields pleasing situations, because of the easy implementation for *get* and *put*. Moreover the implication

Constant complement property ⇒ History Ignorance

infers another desired property.

Programming Lenses: Current Approaches

- QVT-R(elational)
- Rust
- JTL (Janus Transformation Language)

```
relation ClassToTable{
        checkonly domain uml c:Class { namespace=p:Package {},
                                       kind='Persistent', name=cn }
        enforce domain rdbms t:Table {schema=s:Schema {}, name=cn}
        when { PackageToSchema(p, s); }
        where { AttributeToColumn(c, t); }
}
```

```
impl Lens<TwoStrings, String> for Concat {…}

fn build_root_widget() -> impl Widget<TwoStrings> {
Flex::column().with_child(named_text_box("first:
").lens(TwoStrings::first))
        .with_child(named_text_box("second:
").lens(TwoStrings::second))
        .with_child(named_text_box("both:
").lens(Concat))
}
```

```
transformation hsm2nhsm(source : HSM, target : NHSM) {

top relation StateMachine2StateMachine {

   enforce domain source sSM : HSM::StateMachine;
   enforce domain target tSM : NHSM::StateMachine;

}

top relation State2State {

   enforce domain source sourceState : HSM::State;
   enforce domain target targetState : NHSM::State;

   when {
     sourceState.owningCompositeState.oclIsUndefined();
   }

}
```

54

QVT-R language (Query View Transformation Relational) is a declarative language. It **enforce**s the consistency relation within the transformation procedure. Enforcement in both directions is possible, but the specification is unclear about the recursive nature of this back and forth transformation. Rust is a new promising programming language with many intelligent features such as security, i.e. built-in concurrency handling. It also uses lenses (a trait) that can be implemented and easily integrated e.g. in a user interface. JTL is an extension of QVT-R and is able to keep pairs of artefacts in sync more intelligently.

Summary

- For model spaces $\mathcal{M}_1$ and $\mathcal{M}_2$, we defined a **conceptual base**, which enables consistency restoration of model pairs, avoids loss of information whenever possible, and differentiates between organizational symmetry and asymmetry