universitätbonn ROOTS

Conditional Transformations

Fabian Noth

noth@cs.uni-bonn.de

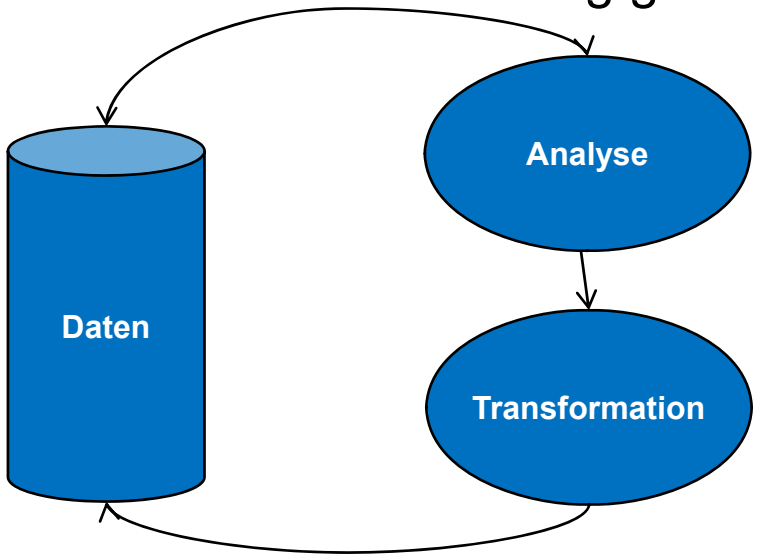
Motivation

Wofür brauchen wir Modelltransformationen?

Wieso brauchen wir dafür CTs?

Modelltransformationen

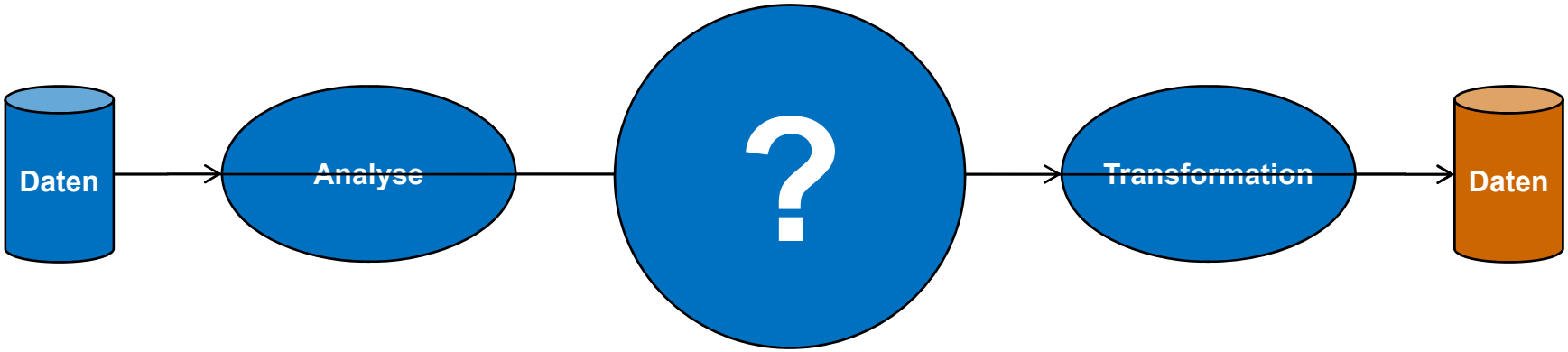
- Analyse und Transformation treten häufig gemeinsam auf



- Beispiele:
  - Compiler:
    - ⇨ Type-Checking (Analyse)
    - ⇨ Kompilieren (Transformation)
  - Refactorings
    - ⇨ Precondition-Checking (Analyse)
    - ⇨ Refactoring (Transformation)

Wie machen wir das?

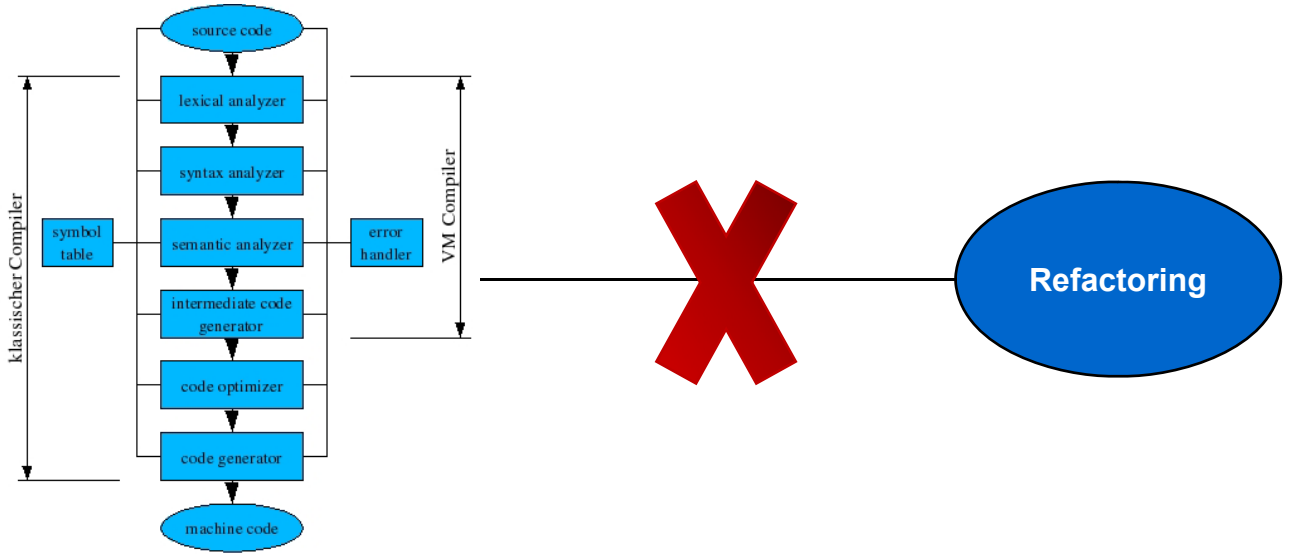
- ~~Analyse und Transformationen getrennt voneinander betrachten~~



- → Fazit: Beides in einem Gesamtsystem verbinden!

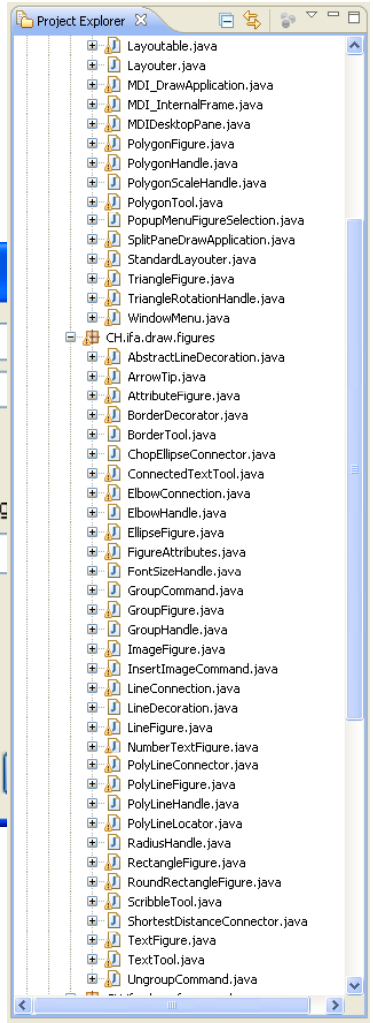
Wie machen wir das?

- Anwendungsspezifische Verbindung von Analyse und Transformation
  - ~~Compiler~~
  - Refactorings in Eclipse



Wie machen wir das?

- ~~Anwendungsspezifische Verbindung von Analyse und Transformation~~
  - ~~Compiler~~
  - ~~Refactorings in Eclipse~~
  - auf allen Klassen ?
  - nur auf Feldern vom Typ int?
  - auf allen Klassen die eine bestimmte Methode enthalten ?



- → Fazit: Generische Umgebung

Wie machen wir das?

- ~~Analyse und Transformationen getrennt voneinander betrachten~~
- ~~Anwendungsspezifische Verbindung von Analyse und Transformation~~

- Fazit: Generische Umgebung die Analyse und Transformation in einem System verbindet
- → Modelltransformation

- QVT
  - standarisierte Sprache für Modelltransformation
  - Bisher: Grundlagen
  - Jetzt:
    - ⇨ Kritik an QVT
    - ⇨ Aufzeigen einer besseren Alternative (Conditional Transformations)

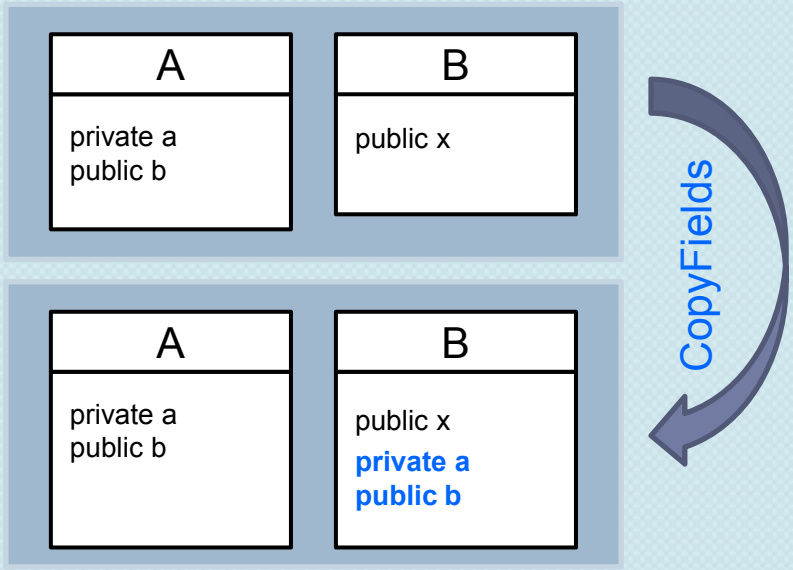
Erster Teil: Kritik an QVT

These: **QVT unterstützt keine historienabhängige Komposition**

Folien aus Diplomanden Vortrag von Marcel Becker
"Model Driven Architecture: Modell-Transformationen mit Conditional Transformations"

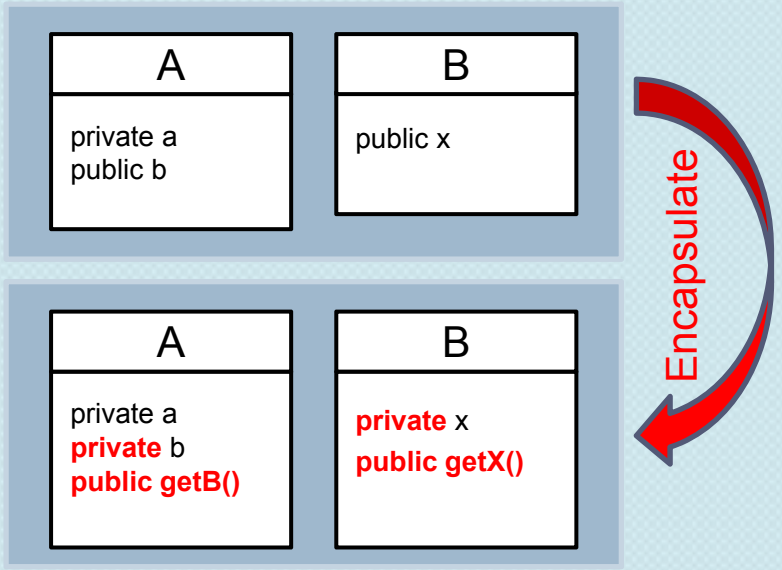
Komposition: Existierende Transformationen

Felder Kopieren



- Kopiert Felder aus einer Klasse in die andere

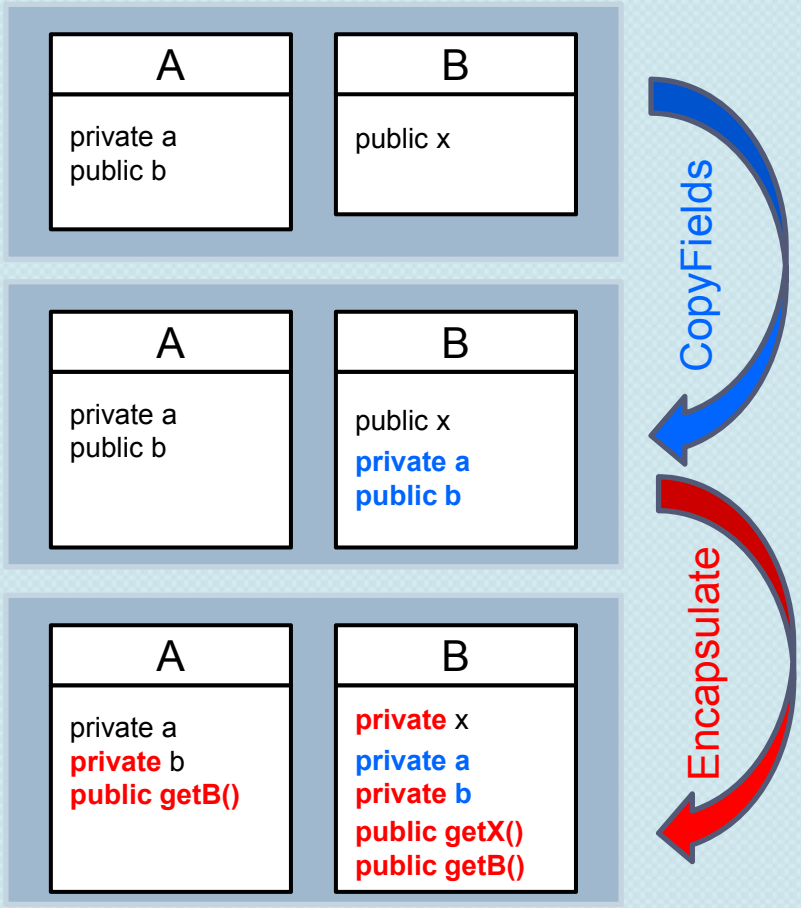
Felder Kapseln



- Macht öffentliche Felder privat
- Erzeugt dafür Zugriffsmethoden

Komposition:
Naive Sequenz

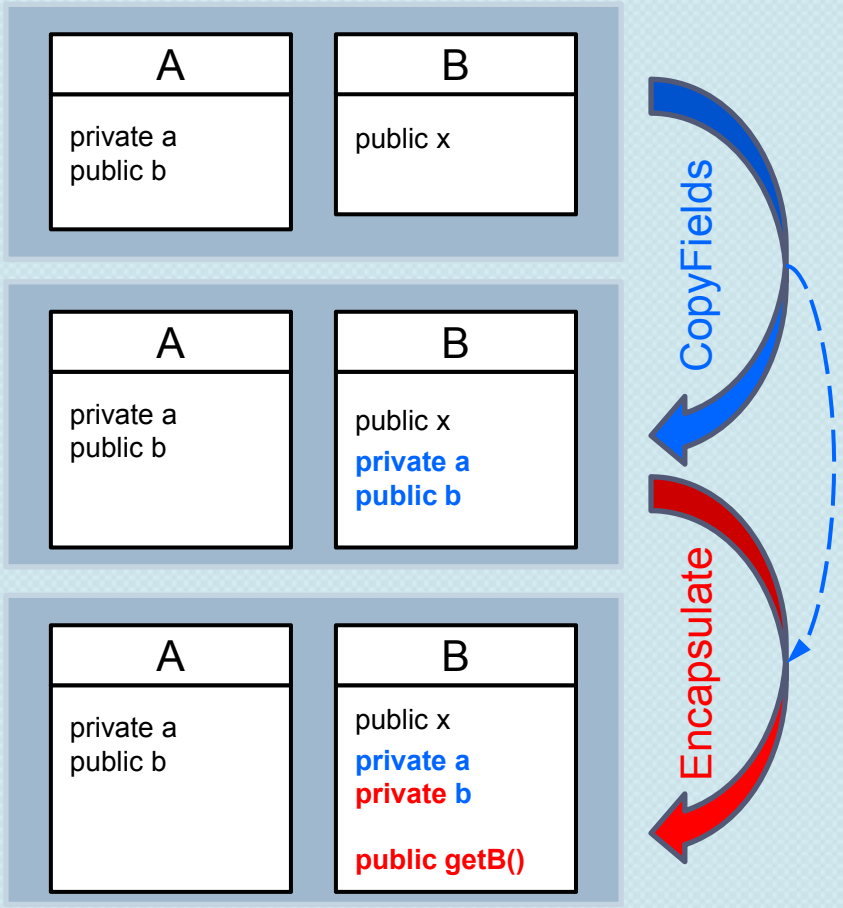
Felder kopieren und Felder einkapseln



- Auf dem Ergebnis von CopyFields wird Encapsulate ausgeführt
- … unabhängig davon, welche Elemente kopiert wurden

Komposition:
Historienabhängige Sequenz

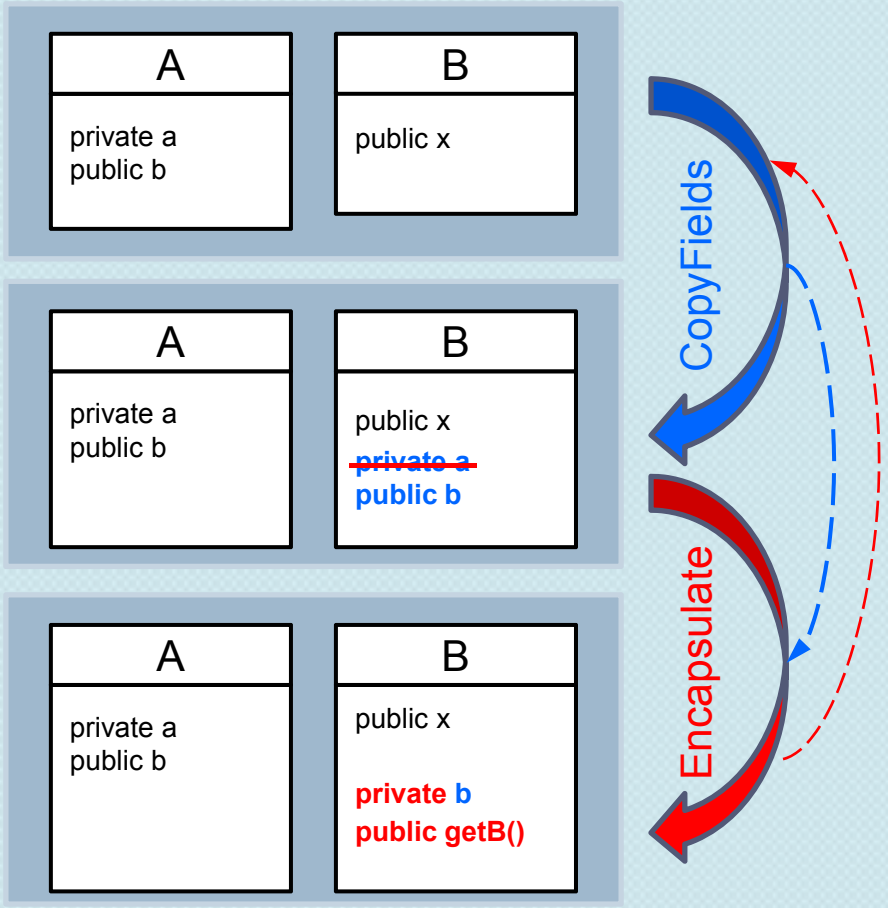
Felder kopieren und diese einkapseln



- Auf dem Ergebnis von CopyFields wird Encapsulate ausgeführt
- … aber nur auf den Elementen, die kopiert wurden!

Komposition:
Bidirektionale Historienabhängigkeit

... und nur kapselbare Felder kopieren!

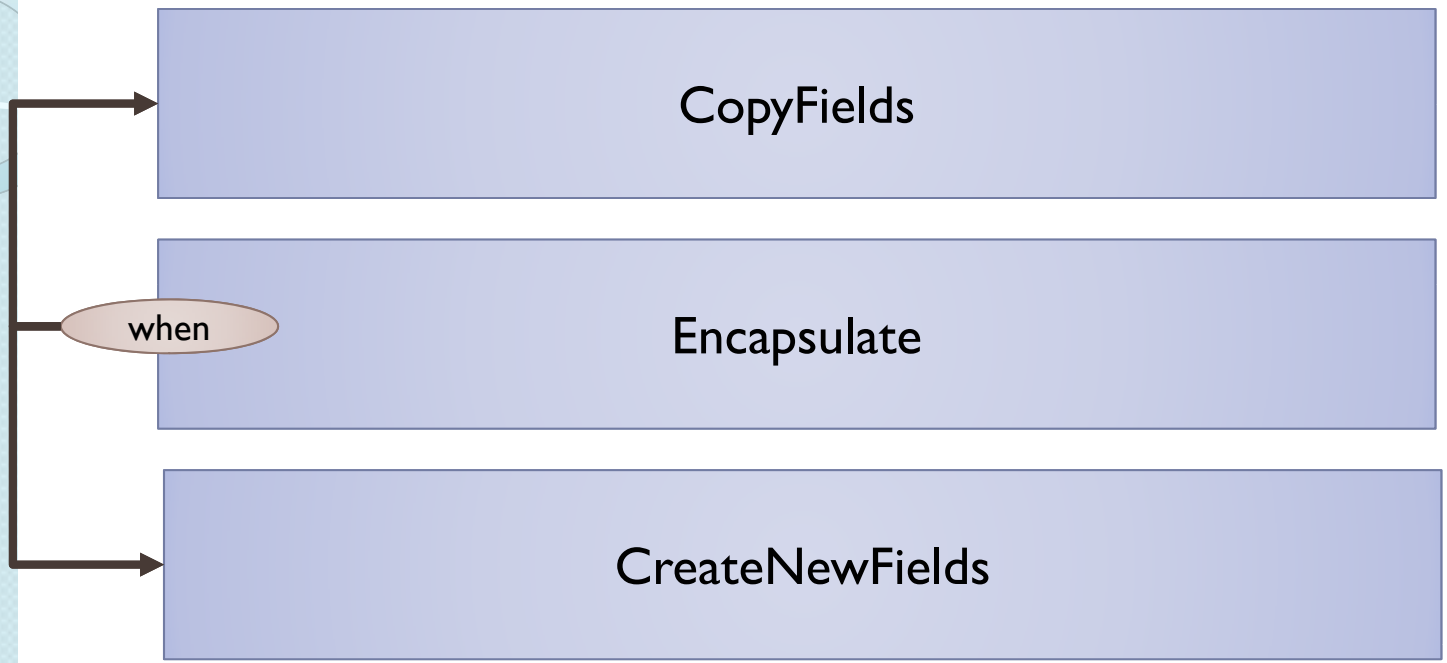


- CopyFields wird nur auf den Elementen ausgeführt, die gekapselt werden können
- dann wird Encapsulate ausgeführt
- … aber nur auf den Elementen, die kopiert wurden!

Ziel

- Diese Arten der Kompositionen durchführen
  - Historienabhängig (von der Vergangenheit)
  - Bidirektional Historienabhängig
- ohne bestehende Transformationen zu verändern
- ohne genauen Inhalt der Transformationen kennen zu müssen

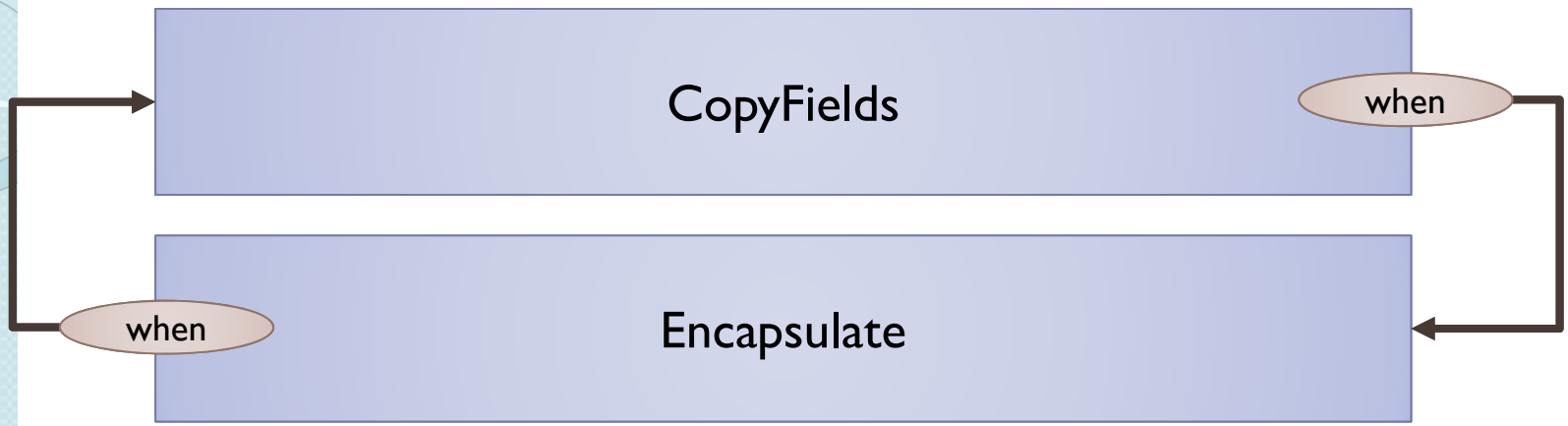
QVT: Historienabhängige Sequenz



Encapsulate muss wissen, was vorher ausgeführt wurde

Transformation muss, je nachdem in welcher Sequenz sie steht, verändert werden

QVT: Bidirektionale Abhängigkeit

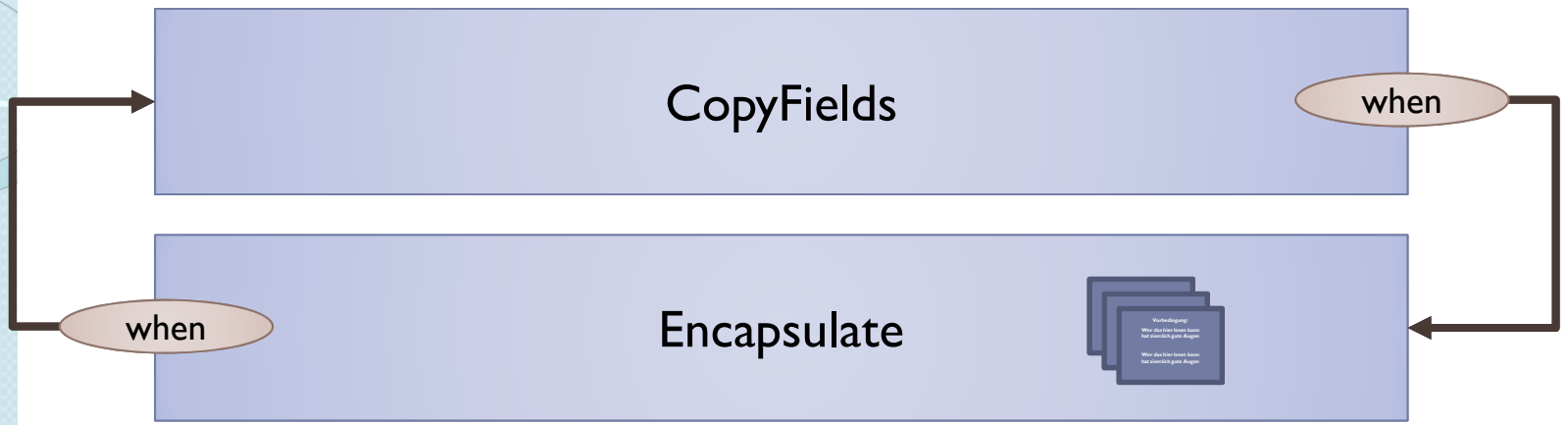


Encapsulate nur auf den Elementen ausführen, auf denen *CopyFields* bereits ausgeführt wurde

CopyFields nur auf den Elementen ausführen, auf denen *Encapsulate* bereits ausgeführt wurde

→ **das führt zu garnichts**

QVT: Bidirektionale Abhängigkeit



einzelne Vorbedingungen aus *Encapsulate* rausziehen und in *CopyFields* prüfen (redundant)

Einfacher: komplett neue Transformation schreiben

CopyAndEncapsulateFields

Fazit

- historienabhängige Komposition mit QVT nur schwer möglich
- bidirektionale historienabhängige Komposition nicht machbar
- Wir wollen:
  - System, welches historienabhängige Komposition unterstützt
    - Uni- und bidirektional
  - ohne bestehende Transformationen zu ändern
  - ohne wissen zu müssen, was die einzelne Transformation macht

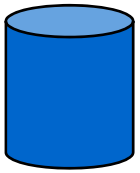
Teil 2: Bessere Alternative Conditional Transformations

Unterstützung für historienabhängige Komposition

Was sind Conditional Transformations?

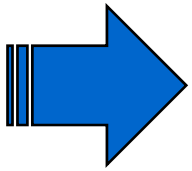
- Ziel: Analyse und Transformationen an Programmen / Modellen
- Einheitliche Grundlage
- Basierend auf Logik
  - Programm als logische Faktenbasis
- Unabhängigkeit von zugrunde liegendem Programm
  - Kein Hintergrundwissen notwendig
- Komplexe Sequenzen
  - durch Kombination von einfachen CTs

Überblick

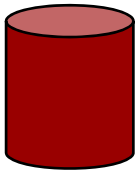


- Darstellung von Programmen
  - logische Faktenbasis

- Auswertung von Bedingungen (Conditions)
  - logische Ausdrücke auswerten



- Propagierung der Ergebnisse (Condition → Transformation)
  - Belegungen für Variablen (Substitutionen) propagieren

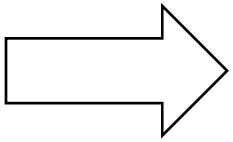


- Ausführen der Transformation
  - Faktenbasis ändern

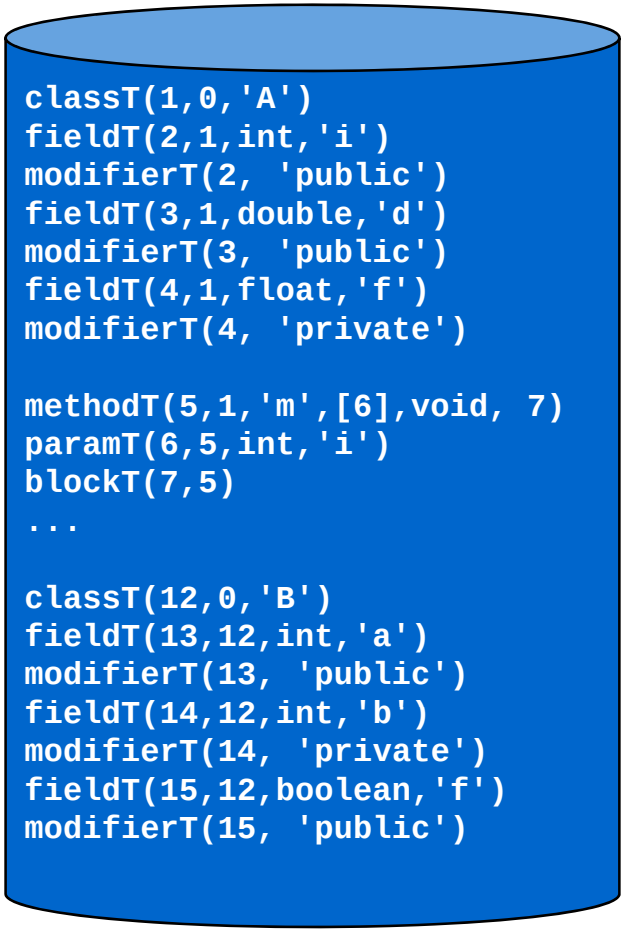
Darstellung von Programmen

Quelltext

```
class A {
   public int i;
   public double d;
   private float f;

   void m(int i)
   {
        m(i);
   }

}

class B {
   public int a;
   private int b;
   public boolean f;
}
```



Faktenbasis



Condition Beispiel

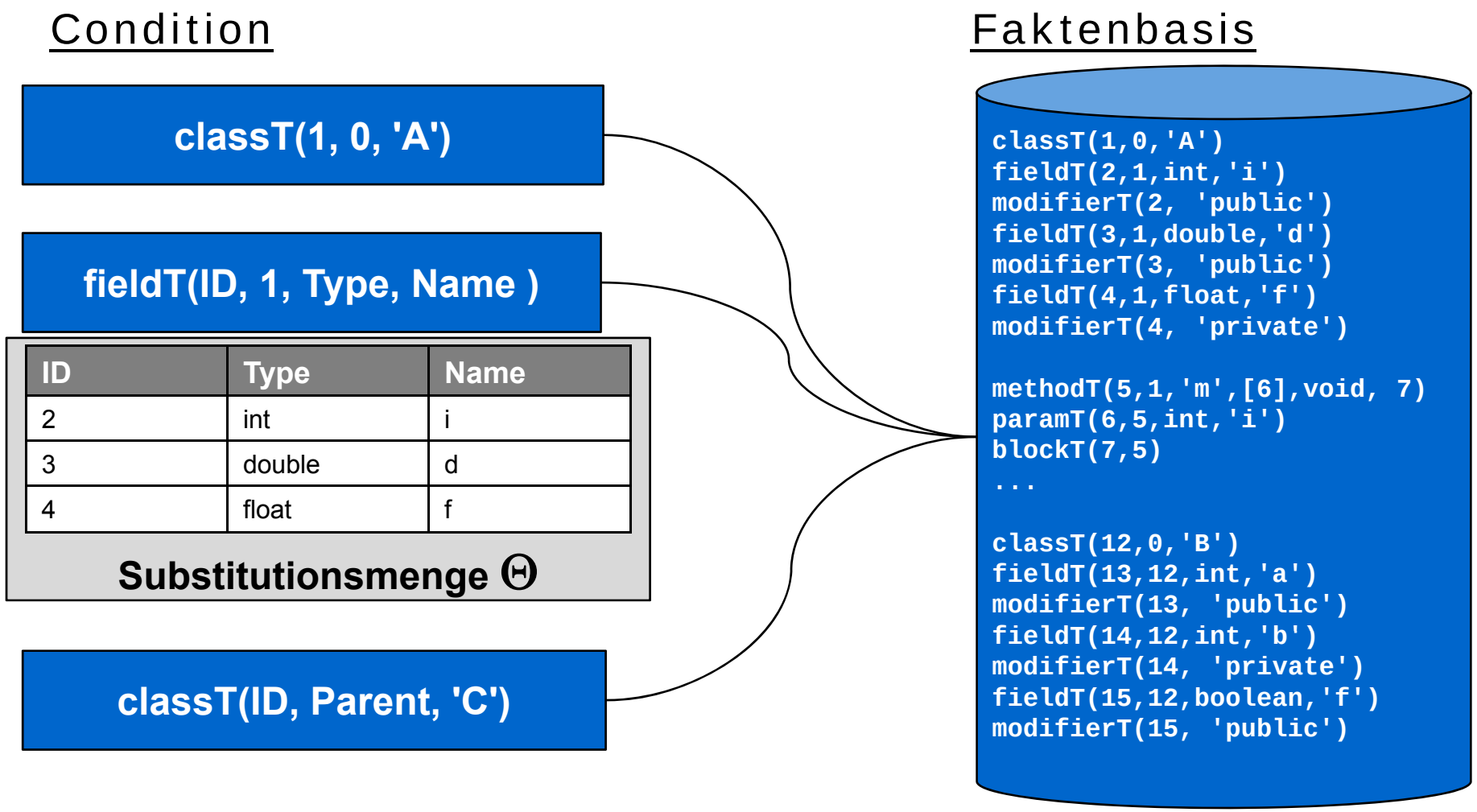
- Vorbedingung für Felder kopieren
  - Feld existiert in Quellklasse aber nicht in Zielklasse

```
fieldT( _, SrcClass,  _, FieldName ),
not( fieldT (_, TargetClass, _, FieldName ))
```

- Vorbedingung für Getter-Methode erstellen
  - Feld existiert und ist public
  - Methode getFeldname() existiert in dieser Klasse nicht

```
fieldT(FieldID, ClassID, Type, FieldName ),
modifierT(FieldID, 'public'),
concat('get', FieldName, MethodName),
not(methodT(_, ClassID, MethodName, [ ] , _, _ ))
```

Auswertung von Bedingungen



Auswertung von Bedingungen

Condition

classT(1, 0, 'A')

- Substitutionsmenge besteht nur aus leerer Substitution
  - Condition ist wahr, unabhängig von Variablen

fieldT(ID, 1, Type, Name)

ID	Type	Name
2	int	i
3	double	d
4	float	f

Substitutionsmenge Θ

- Substitutionsmenge beinhaltet Belegungen der Variablen

classT(ID, Parent, 'C')

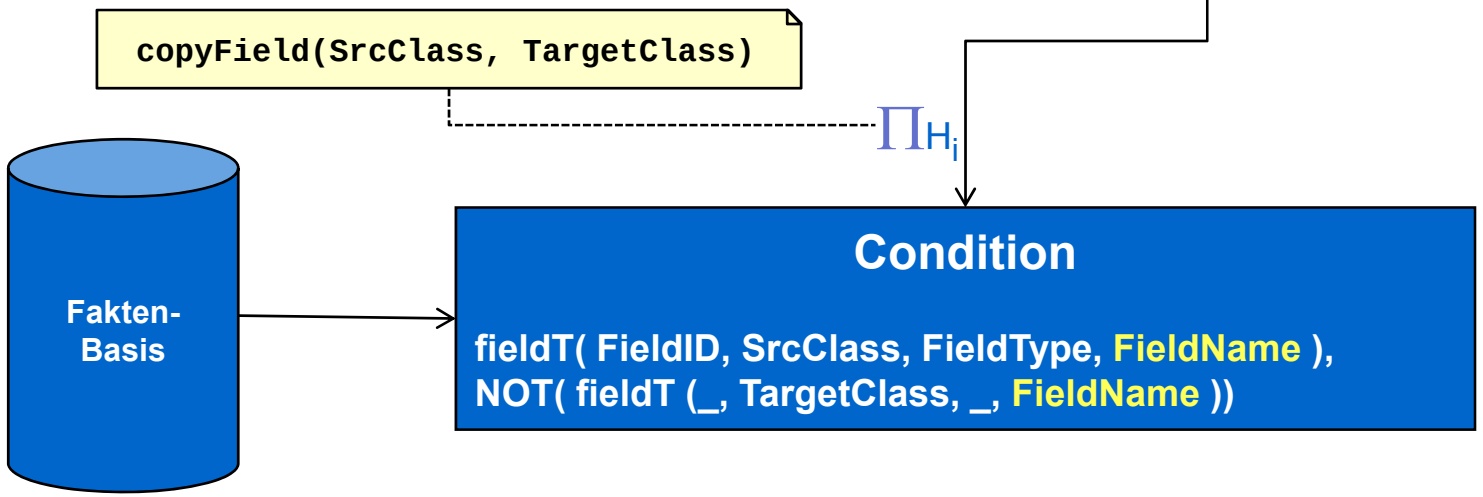
- leere Substitutionsmenge
  - es gibt keine Substitution, die die Condition wahr macht

Substitutionsmenge

Substitutionsmenge als Eingabe

- Ohne Substitution
  - Felder werden von jeder in jede Klasse kopiert
- Deswegen:
  - Condition auswerten aus:
    - ⇨ Faktenbasis
    - ⇨ bereits existierende Substitutionsmenge

SrcClass	TargetClass	Var	...
1	12	108	...



Transformation

- Transformation = Abbildung von Faktenbasis auf Faktenbasis

- Basis-Transformationen:
  - add (neues Fakt in Faktenbasis schreiben)
  - delete (Fakt aus Faktenbasis löschen)
  - replace (bestehendes Fakt ersetzen)

- Transformationen durch beliebige Kombination von Basis-Transformationen

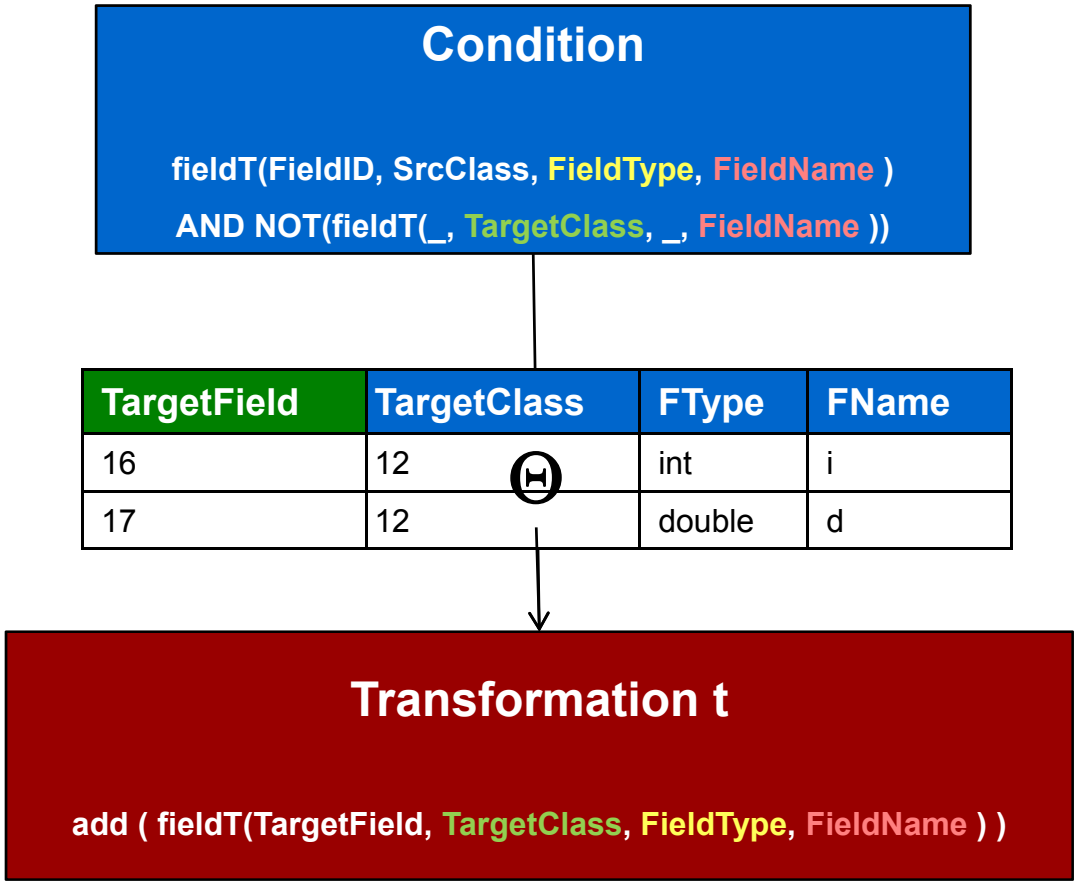
Transformation

- Beispiel: Feld hinzufügen

- Transformation t dient als Maske
- kann nur ausgeführt werden, wenn alle Variablen gebunden sind
- Substitutionsmenge als Eingabe
  - jede Substitution auf t anwenden
  - man erhält eine Menge von Transformationen

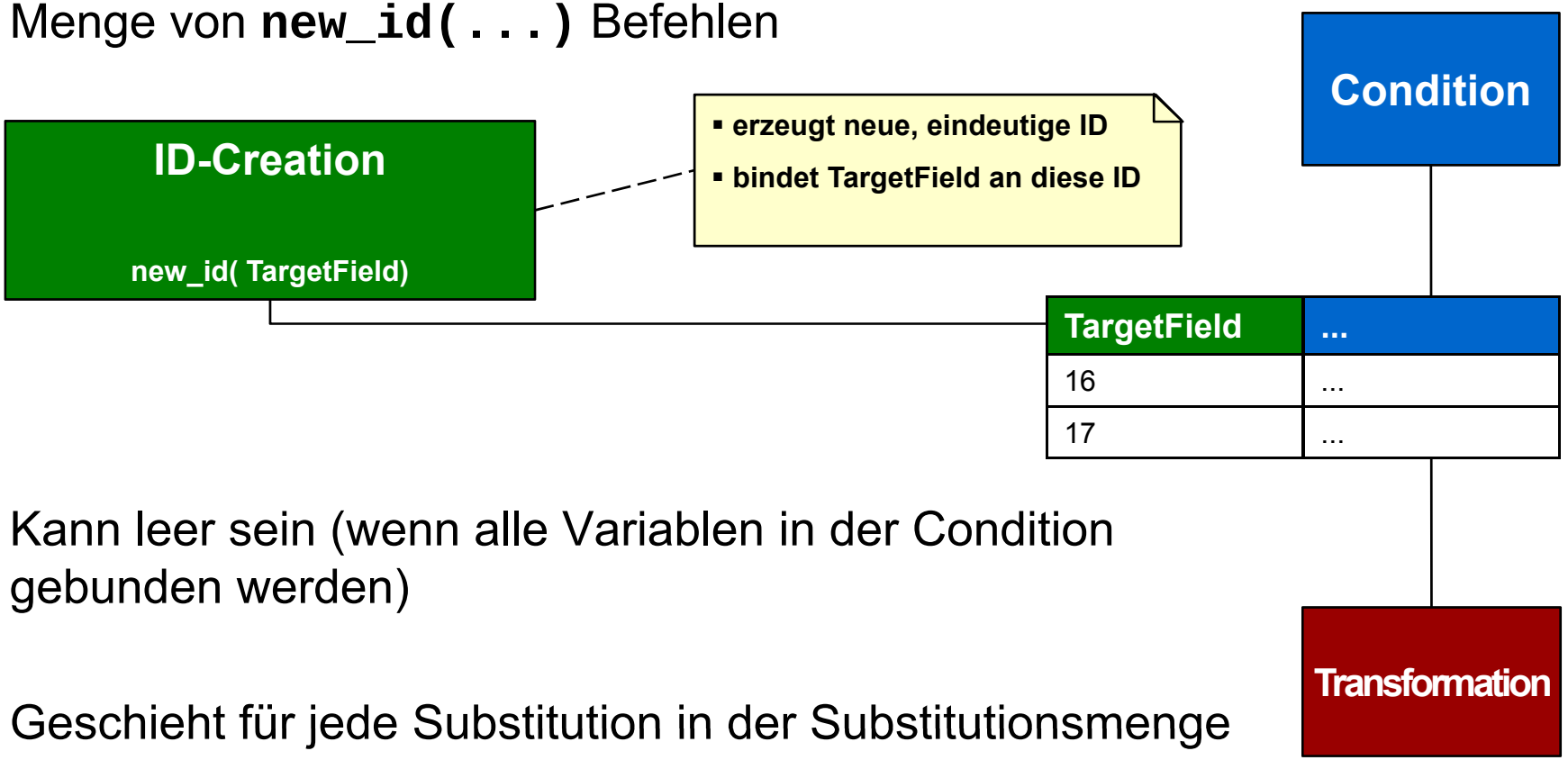
Substitution für Transformation

- Woher kommt Θ ?
- Condition
  - TargetClass
  - FieldType
  - FieldName
- Was ist mit TargetField ?
  - muss eindeutig sein
  - muss neu erzeugt werden
  - zusätzliche Zwischenphase: **ID-Creation**



ID-Creation

- Menge von **`new_id(...)`** Befehlen



- Kann leer sein (wenn alle Variablen in der Condition gebunden werden)
- Geschieht für jede Substitution in der Substitutionsmenge
- Nicht mehrere new_id(...) Befehle für die selbe Variable

Beispiel: copyField

```
ct( copyField(SrcClass,SrcField,FType,FName,TargetClass,TargetField),

    condition( (
      fieldT(SrcField,SrcClass,FType,FName),
      not(fieldT(_,TargetClass,_,FName))
    ) ),
    idcreation( (
      new_id(TargetField)
    ) ),
    transformation(
      add(fieldT(TargetField,TargetClass,FType,FName)
    )
  ).
```

Beispiel: copyField

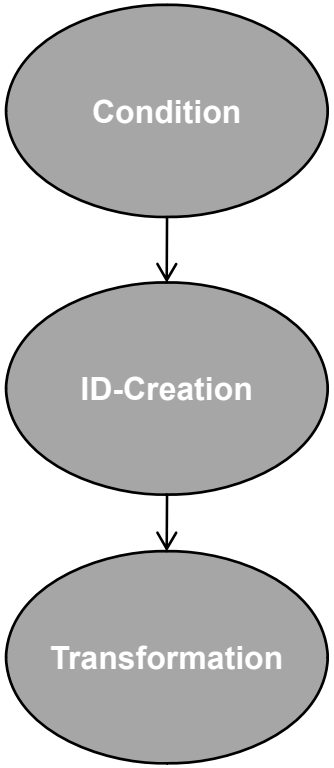
Aufruf: `copyField(1, SrcField, FType, FName, 12, TargetField)`

SrcClass	TargetClass
1	12

fieldT(SrcField, SrcClass ,FType, FName),
not(fieldT(_, TargetClass, _, FName))

Θ_{cond}

SrcClass	TargetClass	SrcField	FType	FName
1	12	2	int	i
1	12	3	double	d



Beispiel: copyField

Aufruf: `copyField(1, SrcField, FType, FName, 12, TargetField)`

Θ_{cond}

SrcClass	TargetClass	SrcField	FType	FName
1	12	2	int	i
1	12	3	double	d

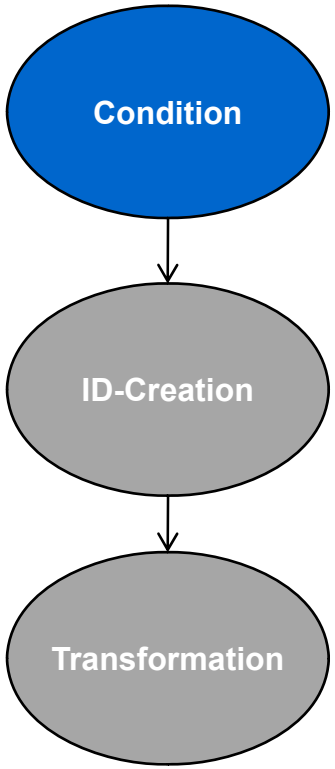
Π_T

Projektion auf Substitutionen die für Transformation benötigt werden

new_id(TargetField)

Θ_T

TargetClass	FType	FName	TargetField
12	int	i	16
12	double	d	17

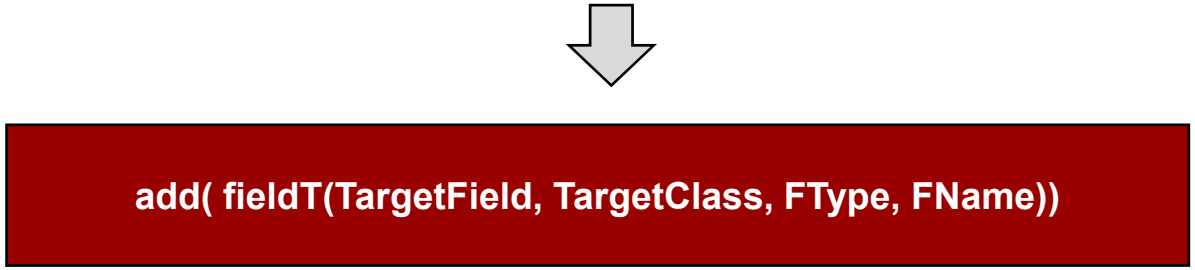


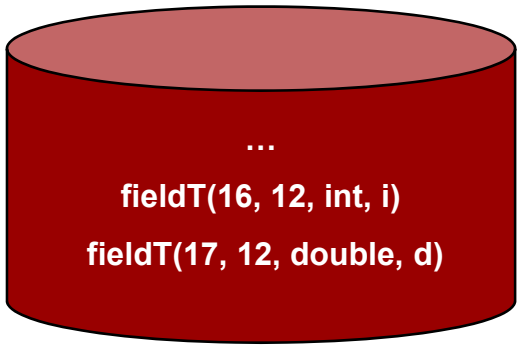
Beispiel: copyField

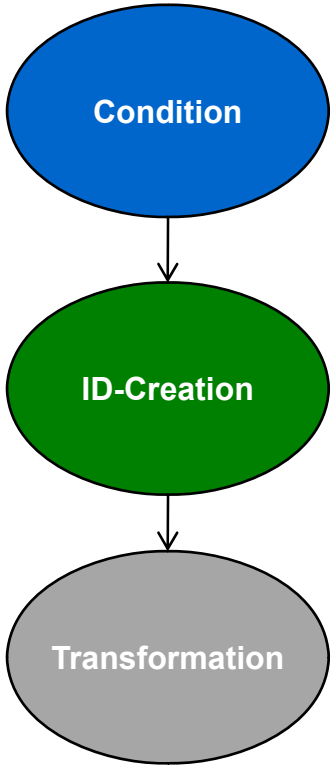
Aufruf: `copyField(1, SrcField, FType, FName, 12, TargetField)`

Θ_T

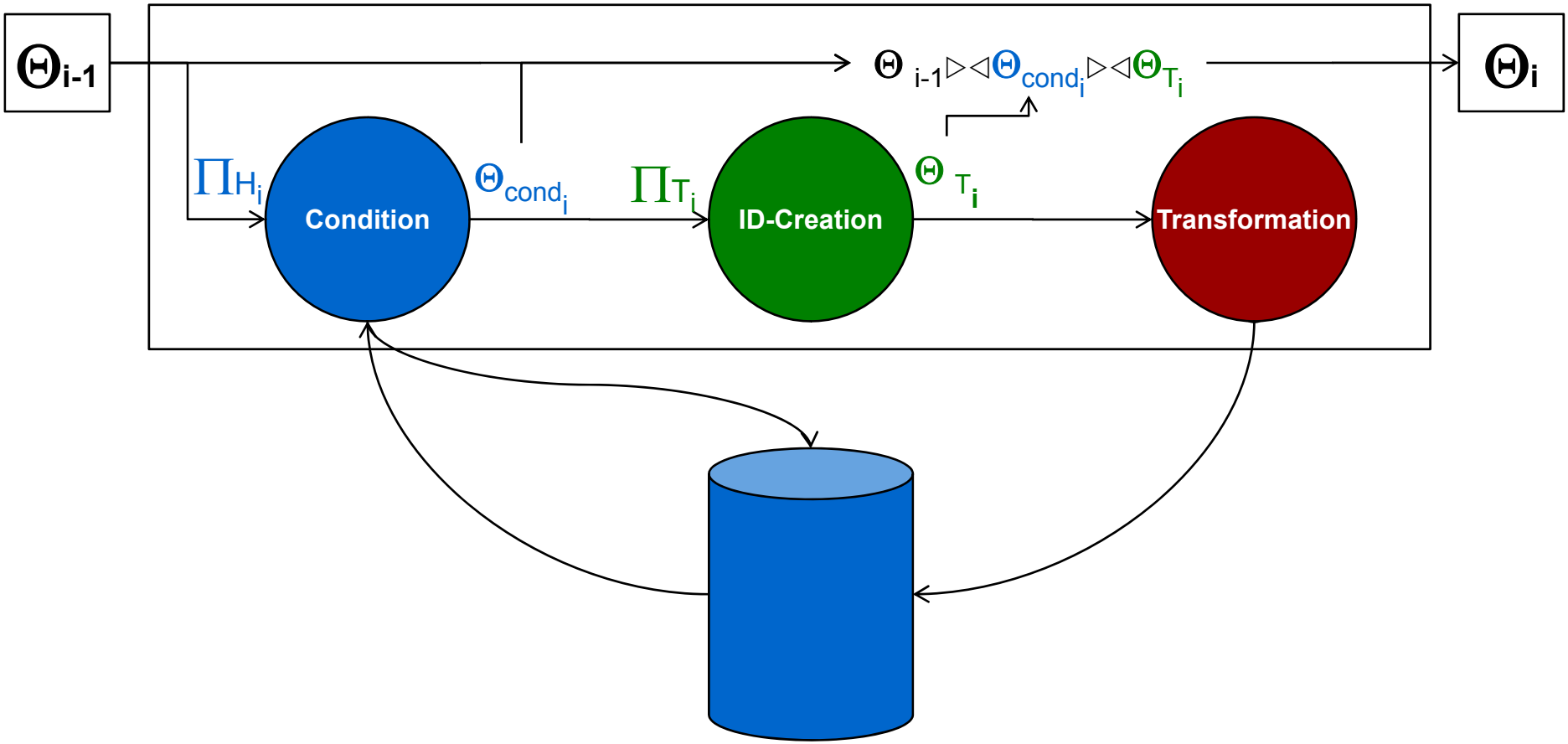
TargetClass	FType	FName	TargetField
12	int	i	16
12	double	d	17







Workflow


Θi-1
Θ i-1▷◁Θcondi▷◁ΘTi
Θi
ΠHi
Condition
Θcondi
ΠTi
ID-Creation
Θ Ti
Transformation

CT als Blackbox

- Blackbox-System
  - Substitution und Faktenbasis als Ein- und Ausgabe
- Wird nicht verändert
- Implementierung muss nicht bekannt sein

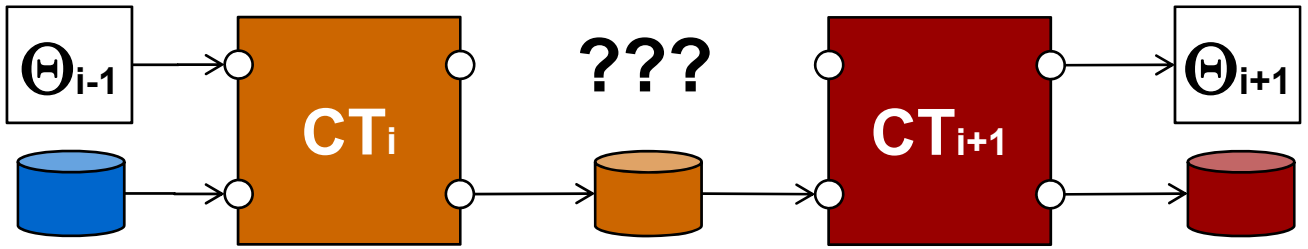
Sequenzen

Komposition einzelner CTs

Sequenzen

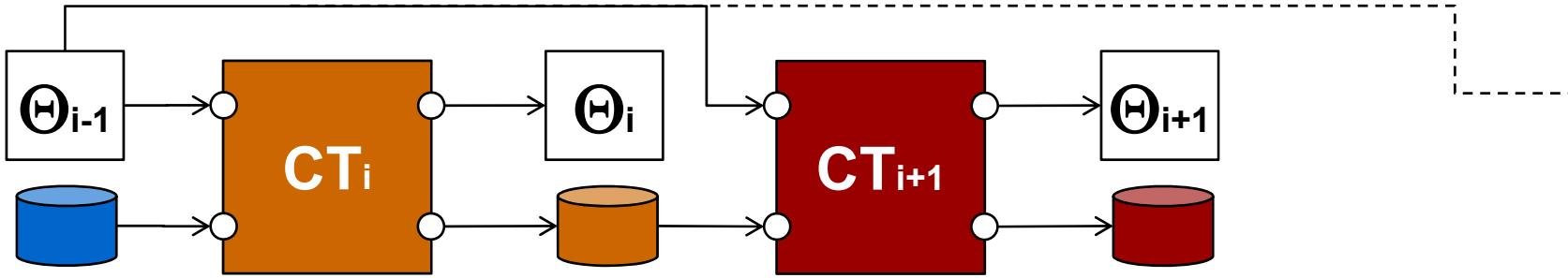
- Daten-Propagierung
  - Propagierung der Faktenbasen
  - Nachfolgende CT arbeitet auf der Faktenbasis die von der vorherigen CT verändert wurde

- Historienpropagierung
  - Propagierung der Substitutionen
  - verschiedene Möglichkeiten → verschiedene Sequenzen



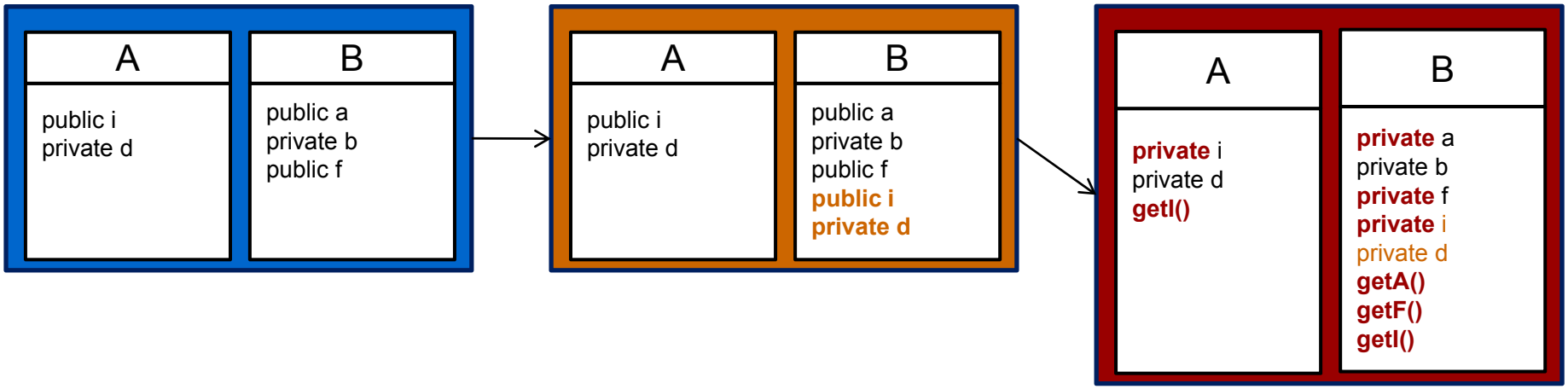
OR-Sequence: $CT_i \mid>> CT_{i+1}$

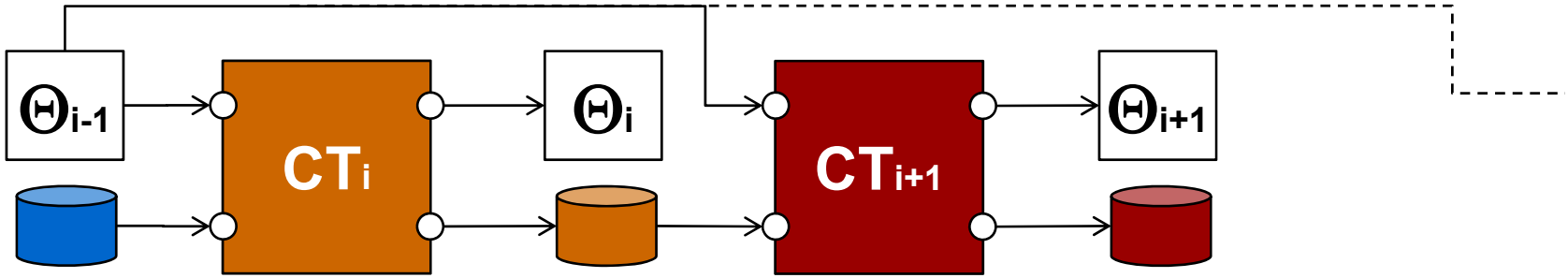
- Keine Historienpropagierung
- Einzelnes Ausführen der CTs



OR-Sequence: $CT_i \mid>> CT_{i+1}$

```
copyField(SrcClass, SrcField, FType, FName, TargetClass, TargetField)
|>> addGetter(TargetField)
```

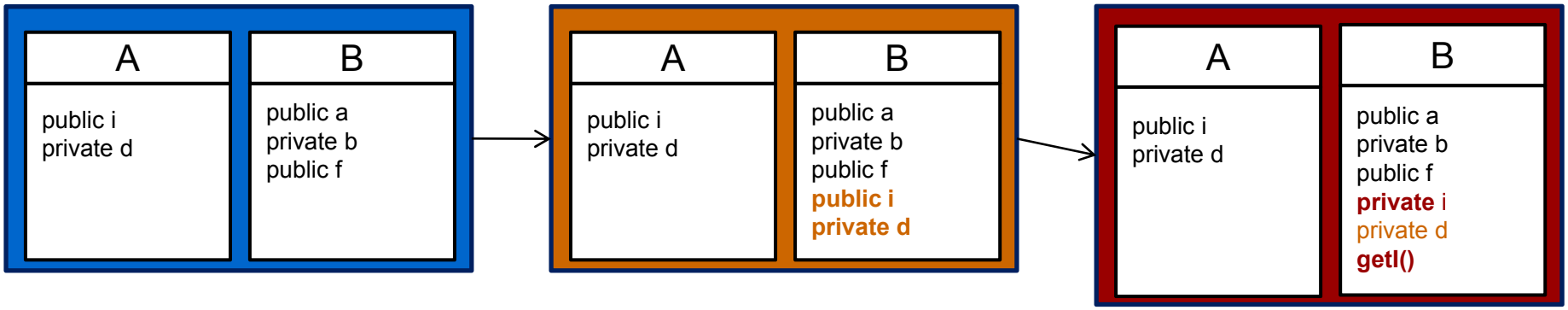


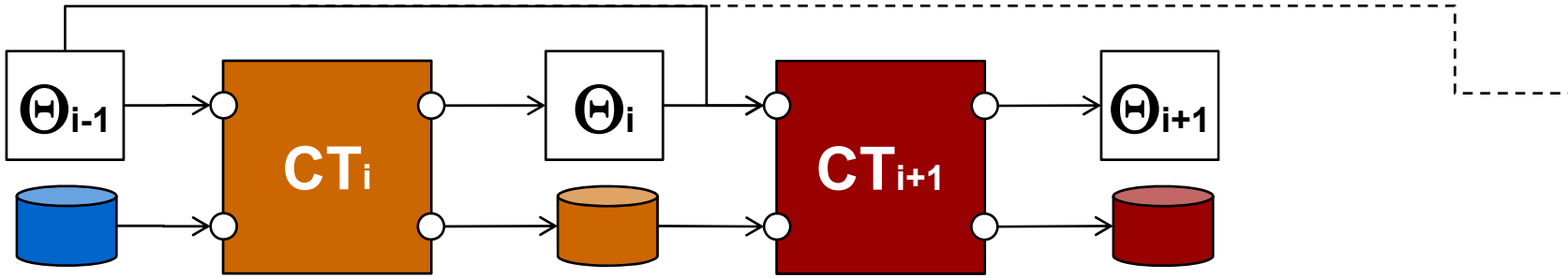


PROP-Sequence

CT_i +>> CT_{i+1}

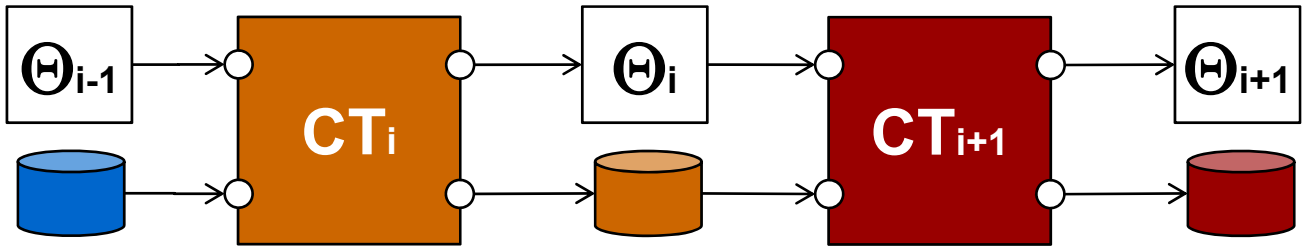
```
copyField(SrcClass, SrcField, FType, FName, TargetClass, TargetField)
+>> addGetter(TargetField)
```





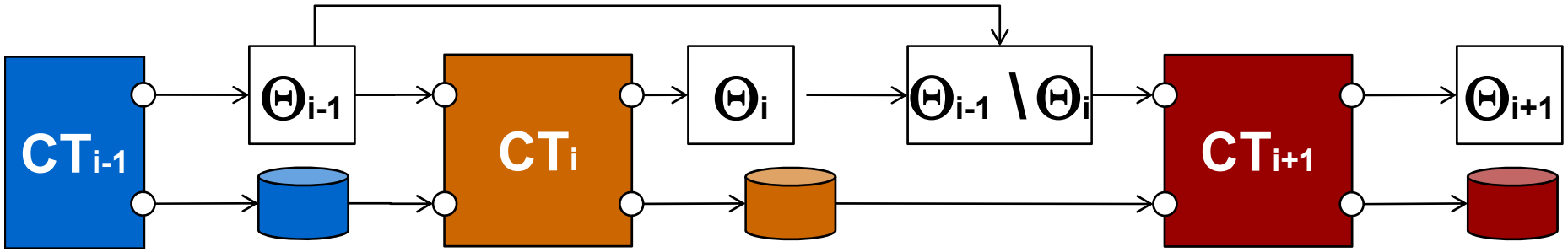
PROP-Sequence
CT_i + > > CT_{i+1}

- Historienpropagierung erfolgreicher Substitutionen
- von links nach rechts (Historienabhängige Sequenz)
- Nachfolge-CT wird mit Ergebnis-Substitutionsmenge der vorherigen CT ausgeführt



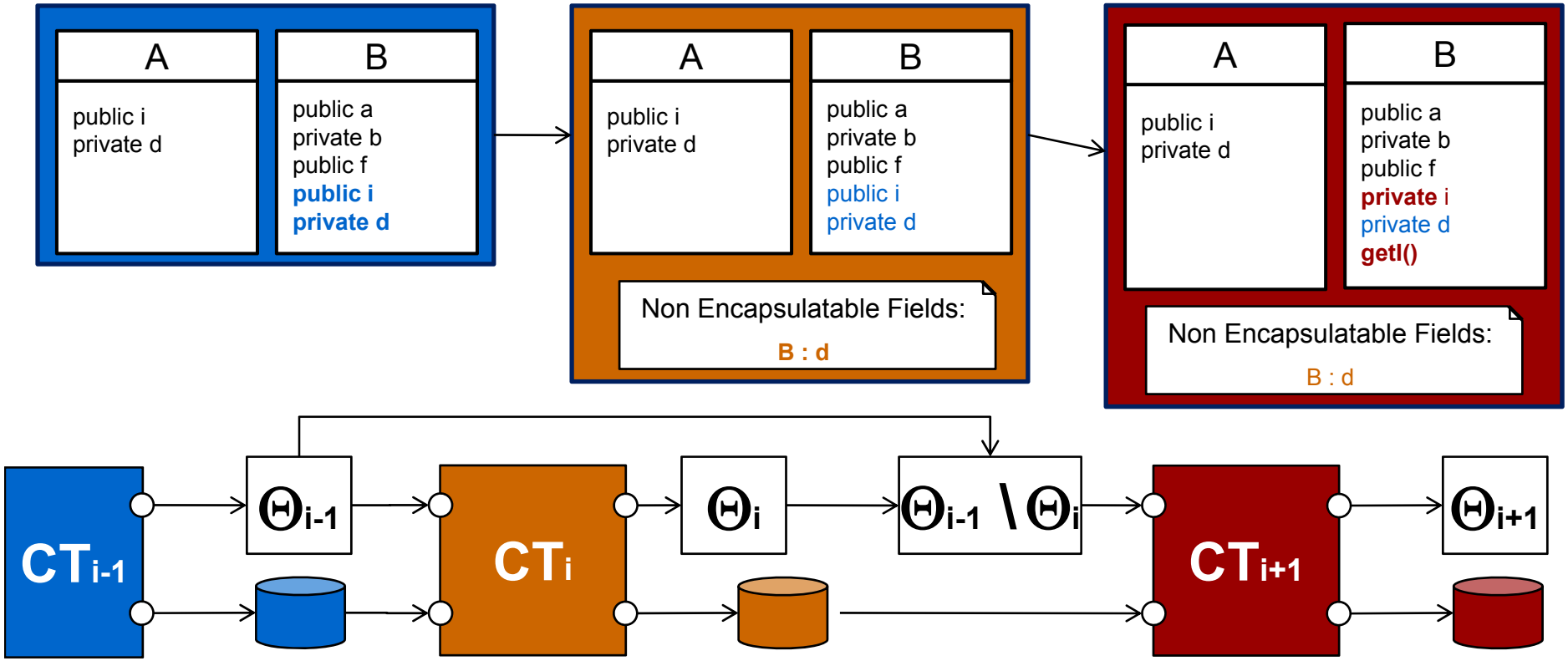
NEGPROP-Sequence
CT_i ->> CT_{i+1}

- Historienpropagierung fehlgeschlagener Substitutionen
- von links nach rechts
- Teilmenge der Substitutionsmenge wird gelöscht



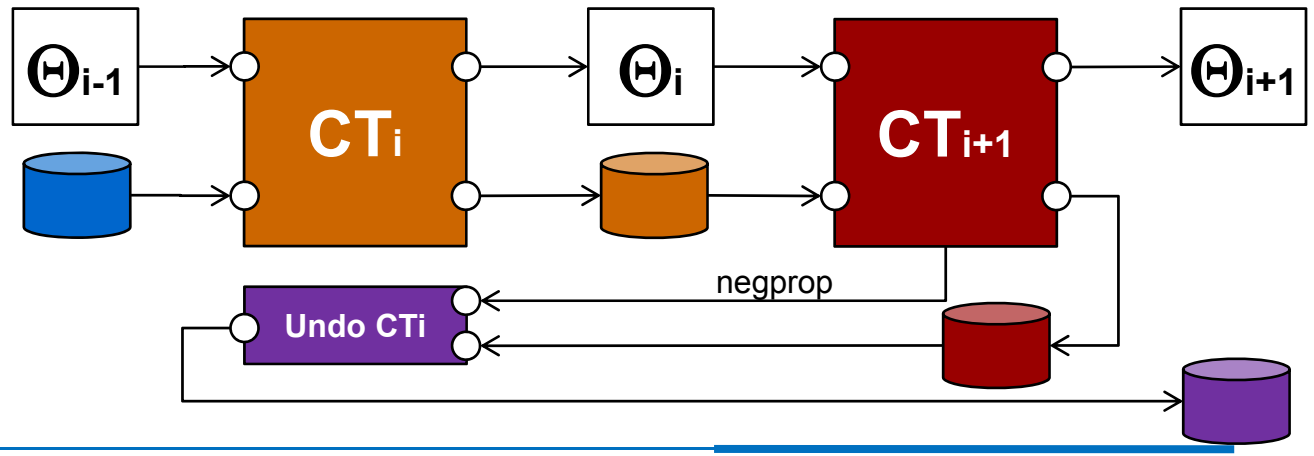
NEGPROP-Sequence
CT_i ->> CT_{i+1}

```
copyField( SrcClass, SrcField, FType, FName, TargetClass, TargetField)
+>> reportNonEncapsulatableField(TargetField)
->> addGetter(TargetField)
```



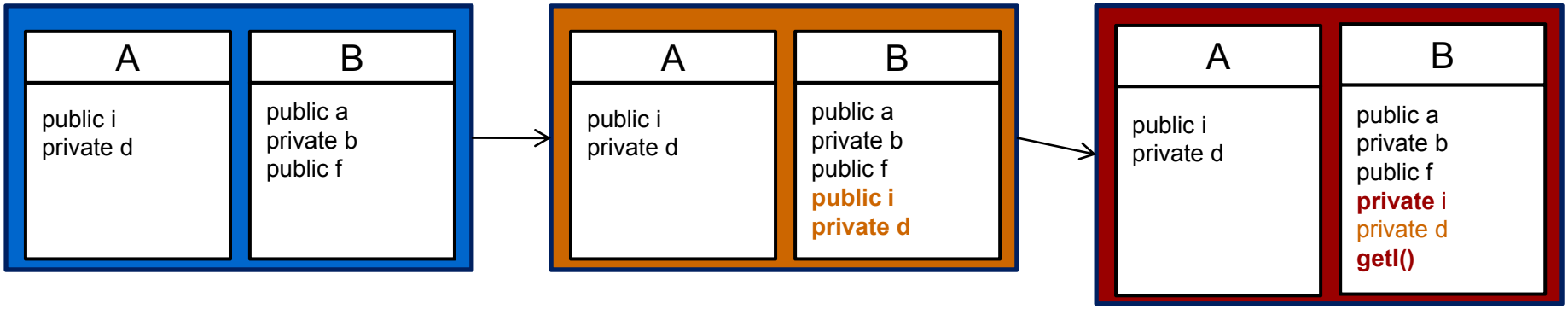
AND-Sequence
CT_i &>> CT_{i+1}

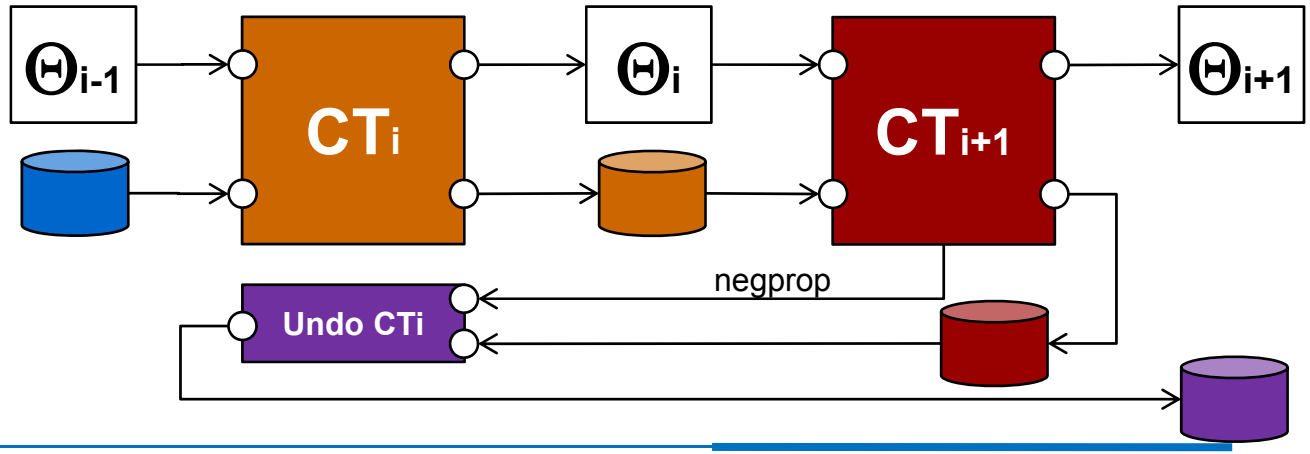
- Propagierung erfolgreicher Substitutionen
- bidirektionale Historienabhängigkeit



AND-Sequence
CT_i &>> CT_{i+1}

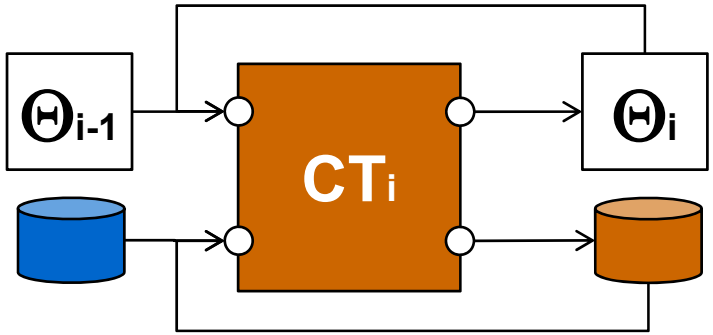
```
copyField( SrcClass, SrcField, FType, FName, TargetClass, TargetField)
&>> addGetter(TargetField)
```





LOOP-Sequence

- Fixpunktiteration
- einzelne CT oder ganze Sequenzen wiederholt ausführen...
- ... bis sich die Faktenbasis nichtmehr ändert



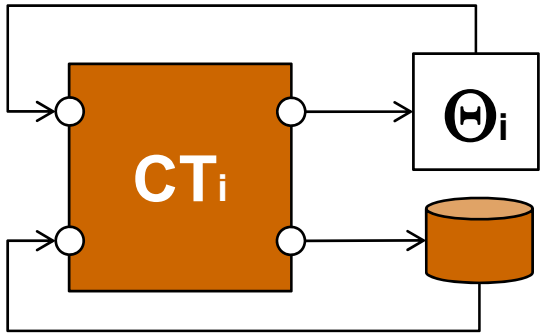
… bis Fixpunkt erreicht ist

LOOP-Sequence Beispiel: Normalisierung

```
public void m()
 {
   i = a.getB().getC().getInt();
 }
```

```
public void m()
 {
   B temp1 = a.getB();
   i = temp1.getC().getInt();
 }
```

```
public void m()
 {
   B temp1 = a.getB();
   C temp2 = temp1.getC();
   i = temp2.getInt();
 }
```



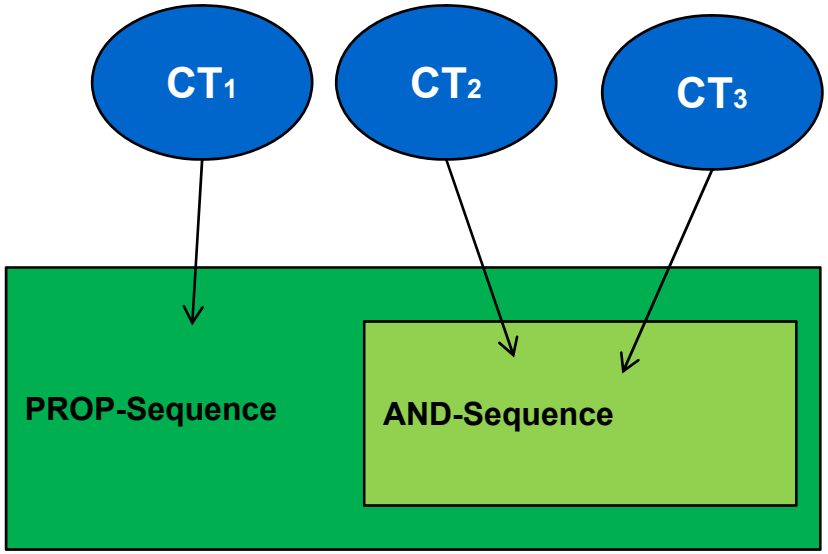
… bis Fixpunkt erreicht ist

Sequenzen Zusammenfassung

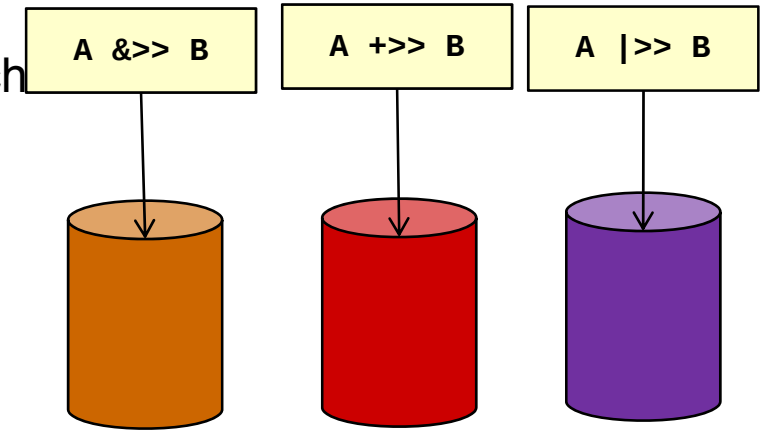
Sequenz	Daten-propagierung	Historien-propagierung	Bemerkungen
OR	+	-	
PROP	+	$\rightarrow$	
NEGPROP	+	$\rightarrow$	fehlgeschlagene Substitutionen
AND	+	$\leftrightarrow$	
LOOP	+	$\rightarrow$	Fixpunktiteration

Fazit: Sequenzen

- Einfache CTs
- Kombination mittels Sequenzen
- Schachtelungen möglich



- Sehr einfach
  - keine Anpassung der CTs erforderlich
- Sehr effektiv
  - sehr unterschiedliche Ergebnisse



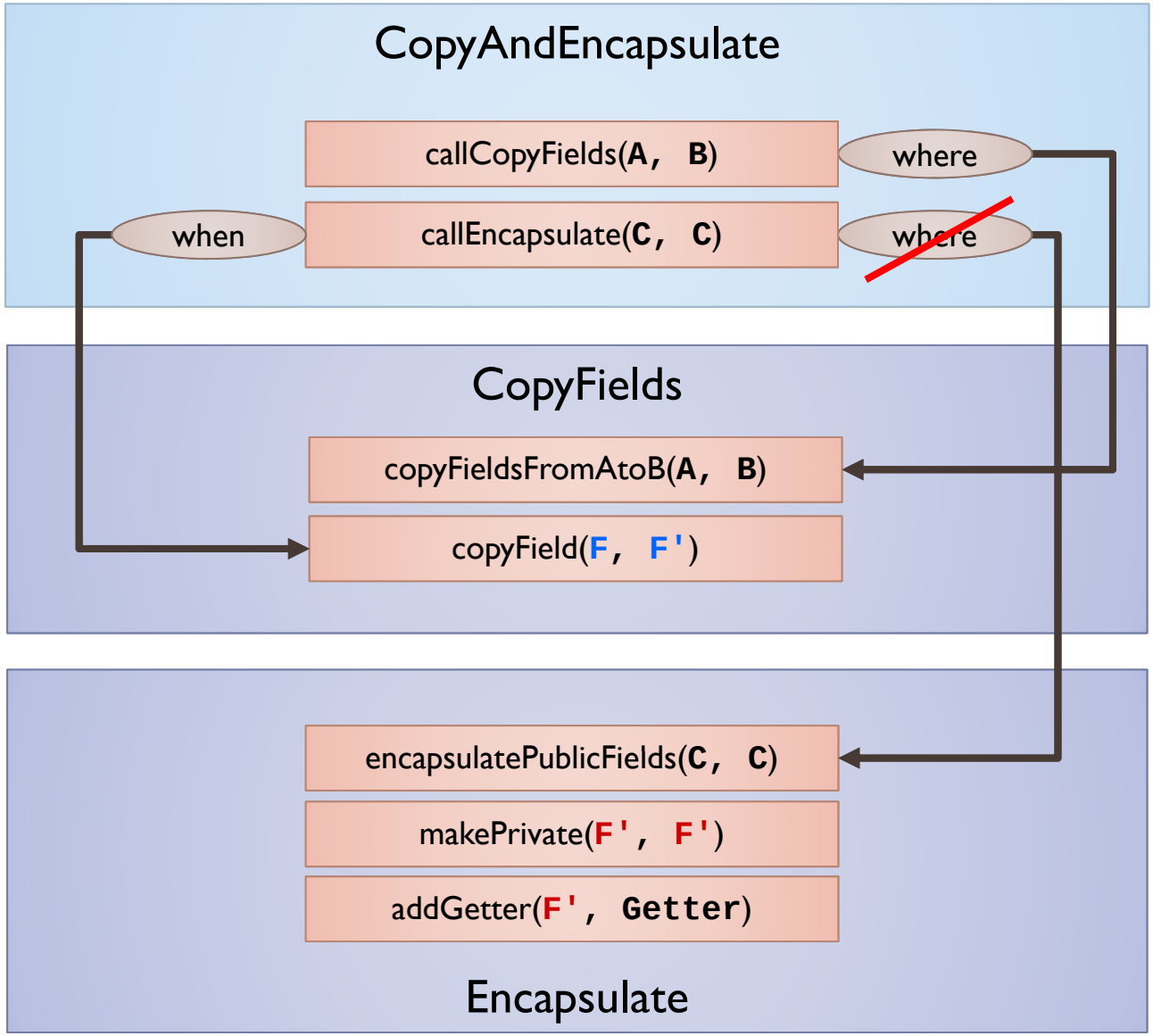
Fazit: QVT vs. CTs

	QVT	CTs
Historienabhängige Sequenz	kompliziert	PROP-Sequence
Bidirektional abhängige Sequenz	unmöglich	AND-Sequence

- Ohne Änderungen an den einzelnen CTs!
- komplexe Transformationen durch zusammengesetzte, einfache CTs
- ➔ CTs unterstützen historienabhängige Komposition sehr viel besser als QVT

Vielen Dank für die Aufmerksamkeit!

QVT: CopyAndEncapsulate



QVT: CopyAndEncapsulate