6 Implementation

- **Input:** software architecture, specification of system components
- **Goals:** programs, documentation, test documentation, verification documentation

Activities

- refinement
- development of algorithms and data structures
- documentation of implementation decisions
- coding
- testing

Transforming Models into Code

- Some models better suited than others:
  - + state charts (FSA), decision tables, class diagrams, Z, . . .
  - – sequence diagrams, Petri nets, . . .
- CASE tools support code generation from models (UML, SA, Z, . . .)
  - rudimentary
  - sometimes also:
    - round-trip engineering, reverse engineering
    - requires program analysis
  - open problems

Here:

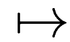
- Implementation of UML class diagrams
- Implementation of Z schemata

6.1 Code Generation for Class Diagrams

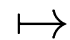
- assumption: class diagram refined to implementation/code perspective
- class diagrams cover static aspects
  - data model
  - inheritance
  - navigability
- dynamic aspects underspecified $\rightarrow$ stubs
- (directly) expressible in OO PL
- still grey areas: composition, aggregation, . . .

(example language: Java)

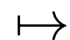
6.1.1 Code for Classes and Interfaces

Person 

```
public class Person {
  Person () {}
}
```

Window 

```
public abstract class Window {}
```

«interface»
Employee 

```
public interface Customer {}
```

Attributes

BankAccount
–status : int = 27 +balance : int

Minimal approach: map visibility and generate constructor

```
public class BankAccount {
  private int status = 27;
  public int balance;

  public BankAccount () {}
  public BankAccount (int balance) { this.balance = balance }
}
```

Encapsulated approach: hide all attributes, generate get and set methods

```
public class BankAccount {
  private int status = 27;
  private int balance;

  public BankAccount () {}
  public BankAccount (int balance) { this.balance = balance; }

  public int getBalance () { return this.balance; }
  public void setBalance (int balance) { this.balance = balance; }
}
```

Implementation decisions

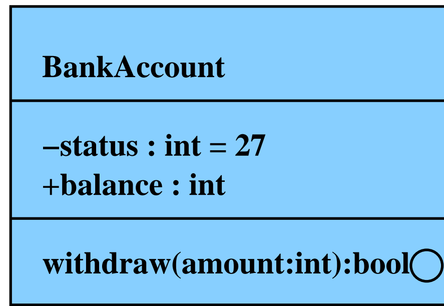
- signature of constructor
- access to attributes (naming convention)

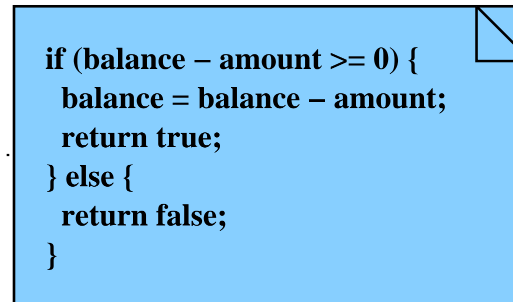
Operations

BankAccount
–status : int = 27 +balance : int
withdraw(amount:int):bool

Generate a code stub:

```
public boolean withdraw (int amount) { }
```

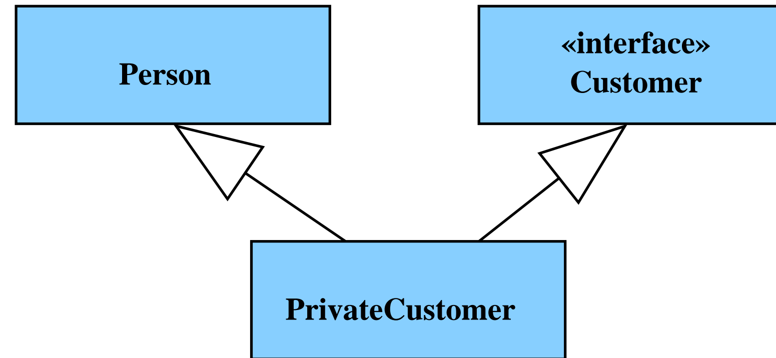




Generate code from template:

```
public boolean withdraw (int amount) {
  if (balance - amount >= 0 ) {
    balance = balance - amount;
    return true;
  } else {
    return false;
  }
}
```

Inheritance



corresponds to

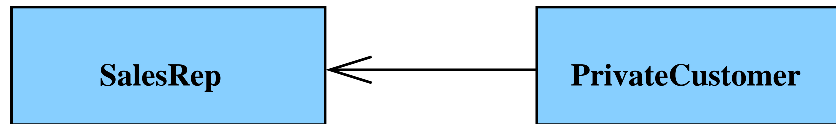
```
public class Person {...}
public interface Customer {...}

public class PrivateCustomer extends Person implements Customer
{
  public PrivateCustomer () { super{}; } // calls Person
}
```

- for Java multiple inheritance must be removed
- Are models independent of implementation language?

6.1.2 Associations

Simple directed association



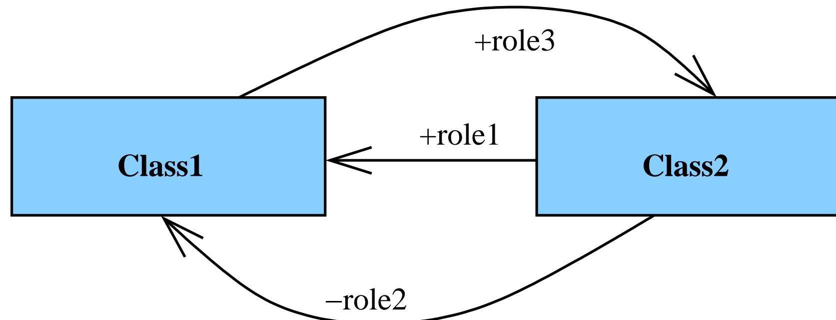
Meaning: objects of class `PrivateCustomer` can send messages to an object of class `SalesRep`

Implementation: instance variable, here with access functions

```
public class PrivateCustomer {
  private SalesRep salesRep;

  public SalesRep getSalesRep() { return salesRep; }
  public void setSalesRep (SalesRep salesRep) { this.salesRep = salesRep; }
}
```

Directed Association with Roles



- an instance variable per role
- visibility transferred

```
public class Class1 {
  public Class2 role3;
}
public class Class2 {
  public Class1 role1;
  private Class1 role2;
}
```

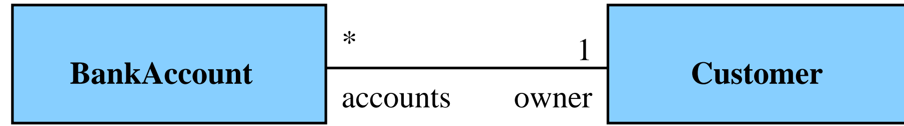
Association with Multiplicity

Simple approach (Rational Rose): arrays

```
public class BankAccount {
  public Customer owner;
}
public class Customer {
  public BankAccount[] accounts;
}
```

Alternatives: container classes (`Vector`), RDBMS

Refined approach:



≪BankAccount.java≫

```
public class BankAccount {
  private Customer owner;
  void setOwner (Customer newOwner) {
    if (newOwner == owner) { return; }
    if (owner != null) {
      owner.releaseAccount(this);
    }
    owner = newOwner;
    owner.addAccount(this);
  }
}
```

≪Customer.java≫

```
public class Customer {
  private Vector accounts = Vector();
  public boolean hasAccount(BankAccount account) {
    accounts.contains(account);
  }
  public Enumeration allAccounts () {
    accounts.elements();
  }
  public void releaseAccount(BankAccount account) {
    accounts.remove(account);
  }
  public void addAccount (BankAccount account) {
    accounts.add(account);
  }
}
```

6.2 Code Generation from Z

6.2.1 Refinement

- Goal: final refinement step results in executable code
- Undecidable in general
- Problem: refinement of
  - data structures
  - algorithms

Operation Refinement: $Increase \sqsubseteq IncreaseBy1$

Specification:	$x' > x$	abstract operation
Refinement:	$x' = x + 1$	concrete operation
Implementation:	`x = x + 1`	program statement

Data Refinement

State is set of elements of $[X]$

$$\textit{Abstract}$$
$$as : \mathbb{P}\, X$$

An operation that modifies the state

$$\textit{AStore}$$
$$\Delta \textit{Abstract}$$
$$x? : X$$
$$as' = as \cup \{x?\}$$

Task: implement this with arrays and lists!

1st step: **data refinement** from set to sequence

Concrete State and Operation

Concrete state is **sequence**

Concrete

$cs : \mathrm{seq}\ X$

Decision: store new element at end

CStore

$\Delta Concrete$
$x? : X$

$cs' = cs \frown \langle x? \rangle$

Question: in which sense is $CStore$ a refinement of $AStore$?

Data Refinement — Correctness

Wanted: relation $ATo C$ between $Abstract$ and $Concrete$ which is preserved by the operation

- The abstraction relation

- For preservation have to show

$$ATo C \wedge AStore \wedge CStore \Rightarrow ATo C'$$

that is:

$$as = \mathsf{ran}\ cs \wedge as' = as \cup \{x?\} \wedge cs' = cs \frown \langle x? \rangle \Rightarrow as' = \mathsf{ran}\ cs'$$

Data Refinement — Proof of Preservation

Given that

$$as \stackrel{(1)}{=} \mathsf{ran}\ cs \land as' \stackrel{(2)}{=} as \cup \{x?\} \land cs' \stackrel{(3)}{=} cs \frown \langle x? \rangle$$

we show that

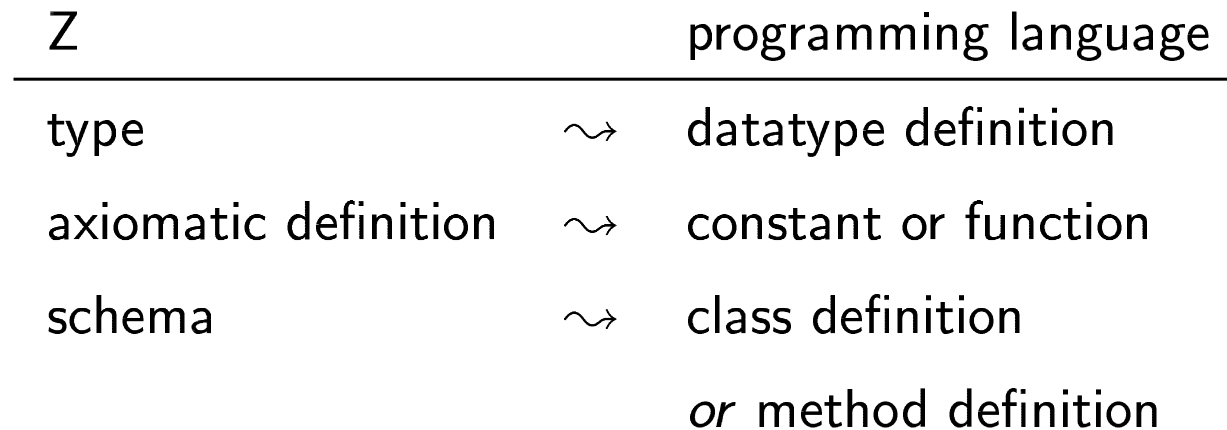
$$as' = \mathsf{ran}\ cs'$$

by calculcation:

$$\begin{aligned}
\mathsf{ran}\ cs' &\stackrel{(3)}{=} \mathsf{ran}\ cs \frown \langle x? \rangle \\
&\stackrel{(\frown)}{=} \mathsf{ran}\,(cs \cup \{\#cs + 1 \mapsto x?\}) \\
&\stackrel{(*)}{=} \mathsf{ran}\ cs \cup \mathsf{ran}\,\{\#cs + 1 \mapsto x?\} \\
&\stackrel{(\mathsf{ran})}{=} \mathsf{ran}\ cs \cup \{x?\} \\
&\stackrel{(1)}{=} as \cup \{x?\} \\
&\stackrel{(2)}{=} as'
\end{aligned}$$

6.2.2 Pragmatic Approach

Idea



Z		programming language
type	$\leadsto$	datatype definition
axiomatic definition	$\leadsto$	constant or function
schema	$\leadsto$	class definition
		or method definition

Free Types

Free types $\leadsto$ enumeration types

$$FAULT ::= overload \mid lineVoltage \mid overtemp \mid groundShort$$

$$faults', faults : \mathbb{P}\, FAULT$$

$$faults' = faults \cup \{overload\}$$

In Java

```
static final int overload = 1;
static final int lineVoltage = 2;
static final int overtemp = 4;
static final int groundShort = 8;

int faultsP, faults;

faultsP = faults | overload;
```

In Haskell (oder Pizza)

```
data FAULT = Overload
           | LineVoltage
           | Overtemp
           | GroundShort

faults', faults :: [FAULT]

faults' = Overload : faults
```

Relations

$[Name, PhoneNumber]$

$$\mid \; phonebook : Name \leftrightarrow PhoneNumber$$

Implementation like associations in class diagrams

- relational database
- (in Haskell) list of tuples:

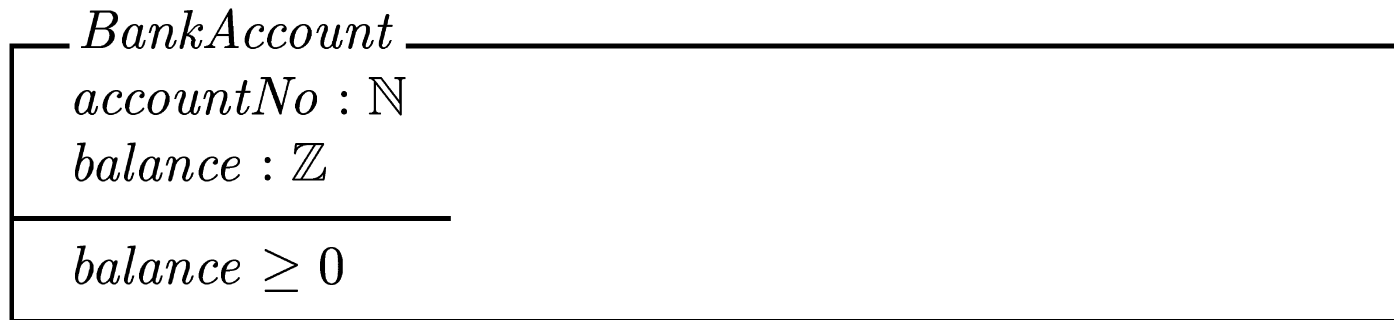
```
type Name = String
type PhoneNumber = String

phonebook :: [(Name, PhoneNumber)]
phonebook =  [("Georg", "8051") ,("Matthias", "8060")
             ,("Wolfram", "8026") ,("Wolfram", "8006")]
```

- (in Java) need to define a new class for tuples, then use *e.g.* `Vector`
- Functions: array, hash table, `AbstractMap` function (Haskell)

Schemata I

(data) schemata $\rightsquigarrow$ classes




```
class BankAccount {
  int accountNo;
  int balance;  // invariant: balance >= 0
}
```

From Predicates to Code

- expressions $\leadsto$ expressions

  $x \bmod 2 \neq 0 \quad \sqsubseteq \quad$ `x%2!=0`

- set membership

  $x \in s \quad \sqsubseteq \quad$ Search for x in data structure s

- function application

  $f(x) \quad \sqsubseteq \quad$ `f.get(x)` if `AbstractMap f`

Assignment

- no change, no code

  $x' = x \quad \sqsubseteq \quad$ `// nothing`

- one change, one assignment

  $x' = e \wedge y' = y \wedge z' = z \quad \sqsubseteq \quad$ `x = e`

- multiple changes more difficult, in general:

  $x' = y \wedge y' = x \quad \not\sqsubseteq \quad$ `x = y; y = x`

  $x' = y \wedge y' = x \quad \sqsubseteq \quad$ `t = x; x = y; y = t`

- direct assignment may be impossible (data structure)

  $s' = s \cup \{x\} \quad \sqsubseteq \quad$ put x in data structure s

Logical Operations

- conjunction $P \wedge Q$: if P precondition and Q state change, then sometimes refinable to conditional

  Example:

  $$x = e_1 \wedge x' = e_2 \qquad \sqsubseteq \qquad \texttt{if(x = e}_1\texttt{) \{x = e}_2\texttt{\}}$$

- disjunction $P \vee Q$: can also refine to conditional

  Example:

  $$(P \wedge S) \vee (Q \wedge T) \qquad \sqsubseteq \qquad \texttt{if(P) \{S\} else if(Q) \{T\}}$$

- incomplete!

  $$d > 0 \wedge n = q' * d + r' \wedge 0 \leq r' < d \qquad \sqsubseteq \qquad ???$$

Quantifiers

Quantifier over finite set $\leadsto$ loop

$$\forall\, x : S \bullet P(x) \qquad \sqsubseteq \qquad \texttt{b} = \texttt{true};\ \texttt{for}(\texttt{x} \in \texttt{S})\ \{\ \texttt{if}(!\texttt{P}(\texttt{x}))\ \{\texttt{b} = \texttt{false};\ \}\}$$

Further refinement possible if implementation of S known

```
b = true;
Iterator iter = S.iterator();
while (iter.hasNext()) {
  if (P (iter.next())) {
    b = false;
  }
}
```

Question: refinement of $\exists\, x : S \bullet P(x)$?

Schemata II

(operation) schemata $\leadsto$ methods (example)

Transaction

$\Delta BankAccount$

$amount? : \mathbb{Z}$

$output! : Message$

$(accountNo' = accountNo$

$balance + amount? \geq 0$

$balance' = balance + amount?$

$output! = TransactionComplete)$

$\vee\ (accountNo' = accountNo$

$balance + amount? < 0$

$balance' = balance$

$output! = AccountOverdrawn)$

```
class BankAccount {
  ...
  public Message
  transaction ( int amount) {
    // precondition TransactionOk
    if (balance + amount >= 0) {
      balance = balance + amount;
      return TransactionComplete;
    }
    // precondition TransactionError
    if (balance + amount < 0) {
      return AccountOverdrawn;
    }
  }
}
```

Summary

- code generation needs manual interaction
- refinement guarantees adherence to specification
- OO variants of Z (Object-Z, OOZE, Z++) can lead to better results
- interesting open problems