7CCSMSUF: Software Engineering and Underlying Technologies for Financial Systems

Kevin Lano

kevin.lano@kcl.ac.uk

Part 3. Specification using UML

Kevin Lano

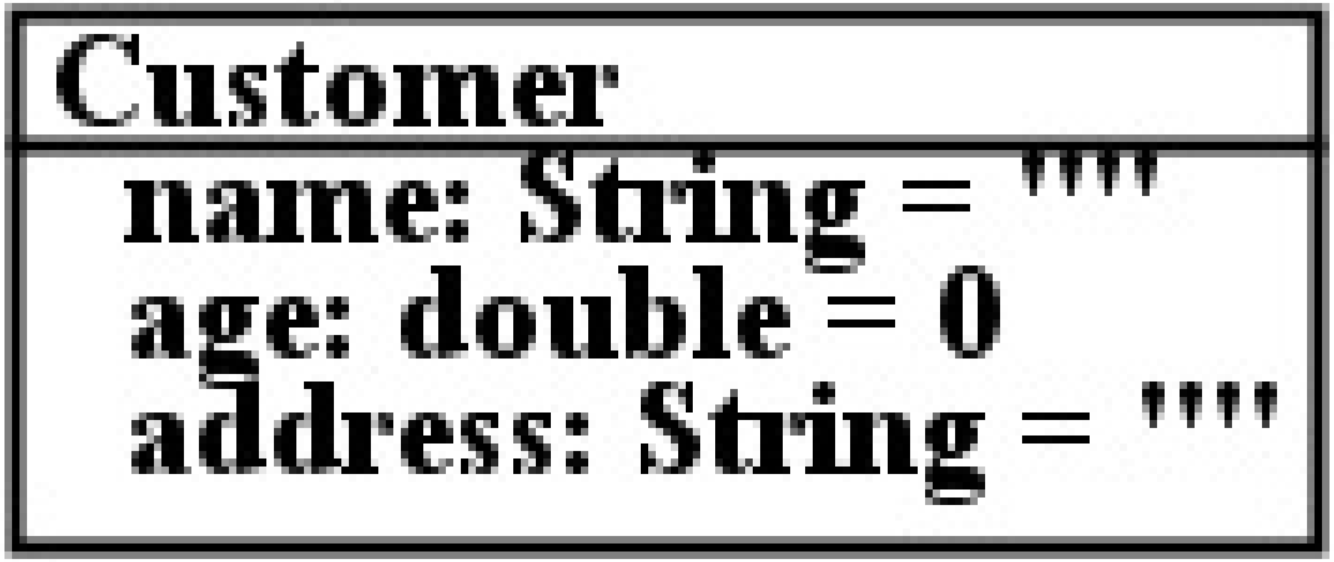
- Class diagrams: classes, attributes, associations, inheritance, operations
- Use cases
- OCL (Object Constraint Language)
- Specification revision; refactoring

In this part we will describe how to use UML for financial system specification.

Class diagrams

- Show the *entity types* of system: the data types (classes) which have instances (objects) with internal structure
- *Value types* include integers, reals, booleans, strings
- Classes have a name (usually singular, with initial capital)
- Classes have series of *attributes* of value type.

Formalise requirements such as “For each customer, their name, age and address are recorded”.



Example of class specification

Class diagrams

- Can be used for initial conceptual modelling of a system
- System specification, informal and formal
- As an executable specification in MBD
- Usually a UML class can be translated directly to a Java, C#, C++, Python, etc class, or to a C `struct`.

```
class Customer
{ String name = "";
  double age = 0;
  String address = "";

  ...
}
```

Attributes

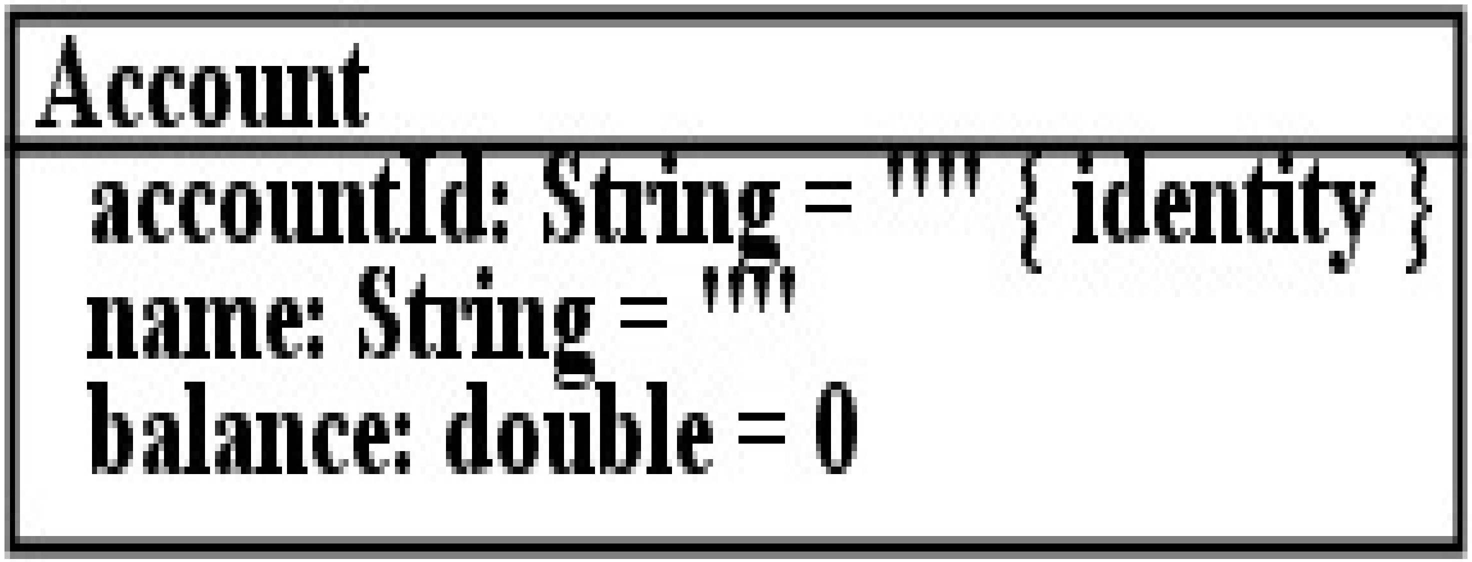
- Intrinsic properties of an object (class instance)
- Permanently attached to the object – although the attribute value can change
- Usually written with lowercase initial letter

If $att : T$ declared in class C, and obj is instance of C, then $obj.att$ is a value of type T.

Identity attributes

- Attributes $eId : String$ which uniquely identify objects of their class
- If $e1 : E$ and $e2 : E$ with $e1.eId = e2.eId$, then $e1 = e2$
- Same as concept of *primary key* for relational databases.

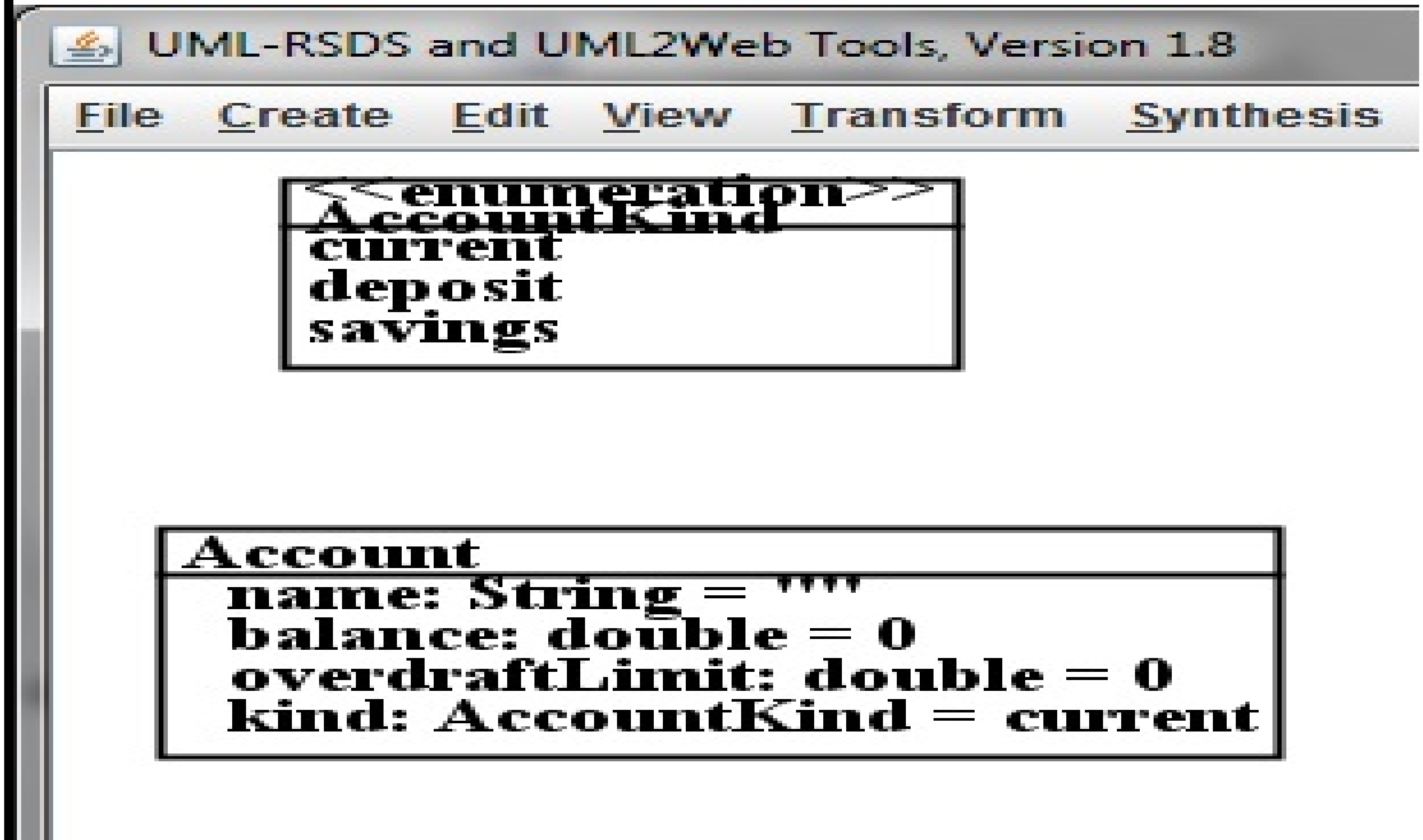
We use notation $E[eval]$ for the instance of E with eId value $eval$.



Identity attribute example

Enumerated types

- Can introduce new finite value types as enumerations
- Distinct named values are listed in an $\ll$ *enumeration* $\gg$ rectangle
- The enumerated type can be used as the type of attributes.

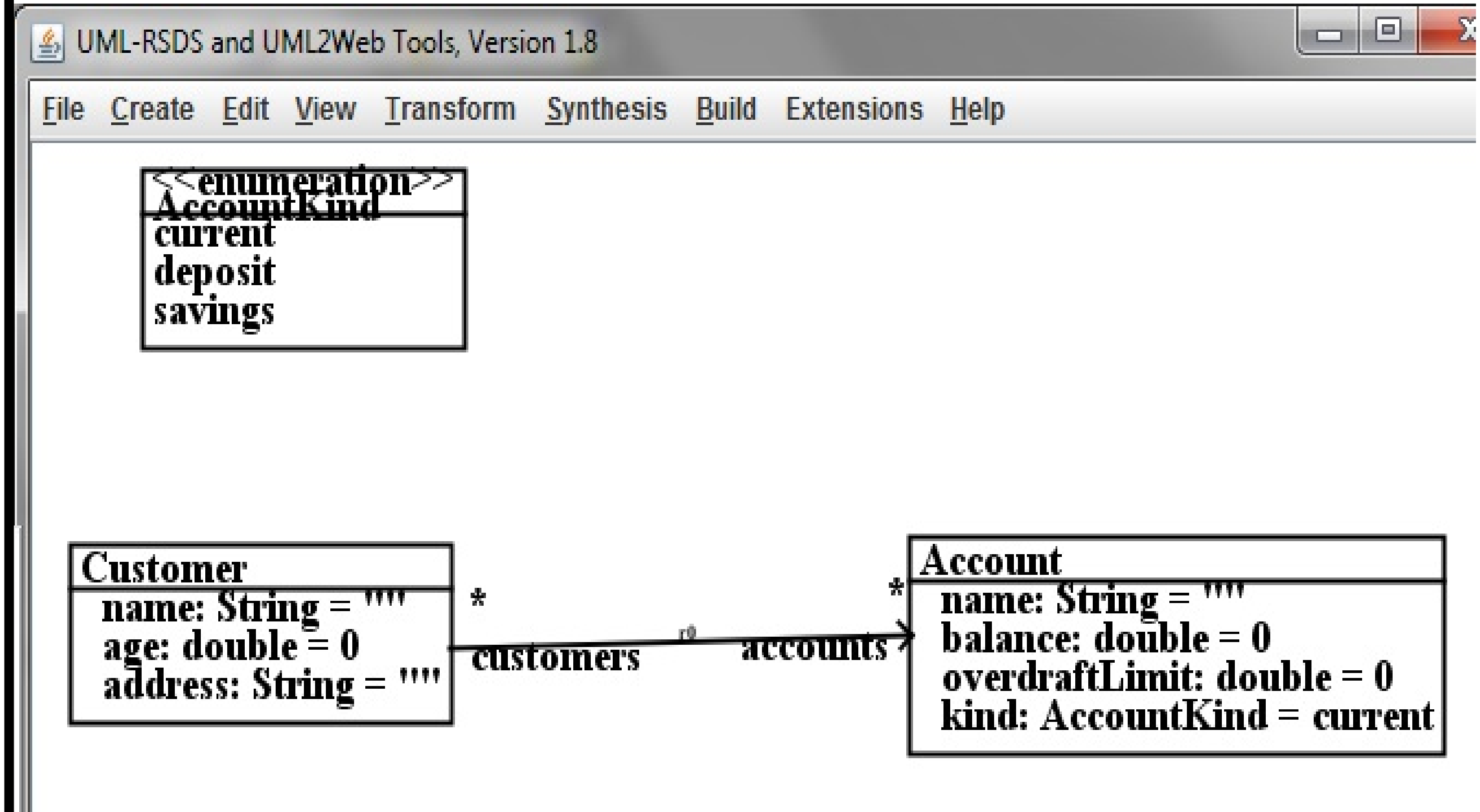


Example of enumeration

Associations

- Define relationships between entities
- Elements of an association are pairs (links) $obj1 \mapsto obj2$ of instances of the source and target entity types
- Rolenames and multiplicities at both ends of association line (role1 – at start of arrow – is optional)

Formalise requirements such as "Each customer has a set of accounts, and each account may belong to several customers".



Example of association

Association multiplicities

*	Any finite number of objects at this end can be linked to one object at other end.
0..1	At most one object at this end can be linked to one object at other end.
1	Exactly one object at this end can be linked to one object at other end.

* means role at that end is a set or sequence of objects of end class, unbounded size. 0..1 means role is set/sequence of size ≤ 1. 1 means it is a specific (non-null) object of end class.

Associations

- If $A \ {}^{m1}\!\!\longrightarrow_{r}^{*} B$ (for some multiplicity $m1$) then for each instance obj of A, $obj.r$ is set (possibly empty) of B objects.
- Eg., $c.accounts$ for customer c
- r is a *feature* of A, as are attributes of A.

```
class Customer
{ String name = "";
  double age = 0;
  String address = "";
  Set<Account> accounts = new HashSet<Account>();
  ...
}
```

Associations

- If $A \;{}^{m1}\!\!-\!\!{}^{1}_{r}\; B$ for any multiplicity $m1$, then for each instance obj of A, $obj.r$ is a single B object. *many-one* association.
- If $A \;{}^{m1}\!\!-\!\!{}^{*}_{r}\;{}^{\{ordered\}}\; B$ then for each instance obj of A, $obj.r$ is sequence (possibly empty) of B objects.
- If $A \;{}^{m1}\!\!-\!\!{}^{0..1}_{r}\; B$ then for each instance obj of A, $obj.r$ is set of 1 or 0 (empty set) of B objects.

$A \;{}^{m1}_{r1}\!\!-\!\!{}^{m2}_{r2}\; B$ association with neither $m1$, $m2$ being 1, is termed a *many-many* association.

Bi-directional associations

- If $A\ {}_{r1}^{m1}\!\!—\!\!{}_{r2}^{m2}\ B$ then $r1$ is a feature of B, with multiplicity $m1$, and $r2$ is a feature of A, with multiplicity $m2$
- $r1$ and $r2$ depend on each other: if pair $a \mapsto b$ is in association, then b is a value of $a.r2$, and a is a value of $b.r1$.
- Maintaining this consistency is difficult using hand-written code
- Bi-directional associations create strong semantic links between classes; should only be used if essential to problem.

Eg., *Customer—Account* relationship.

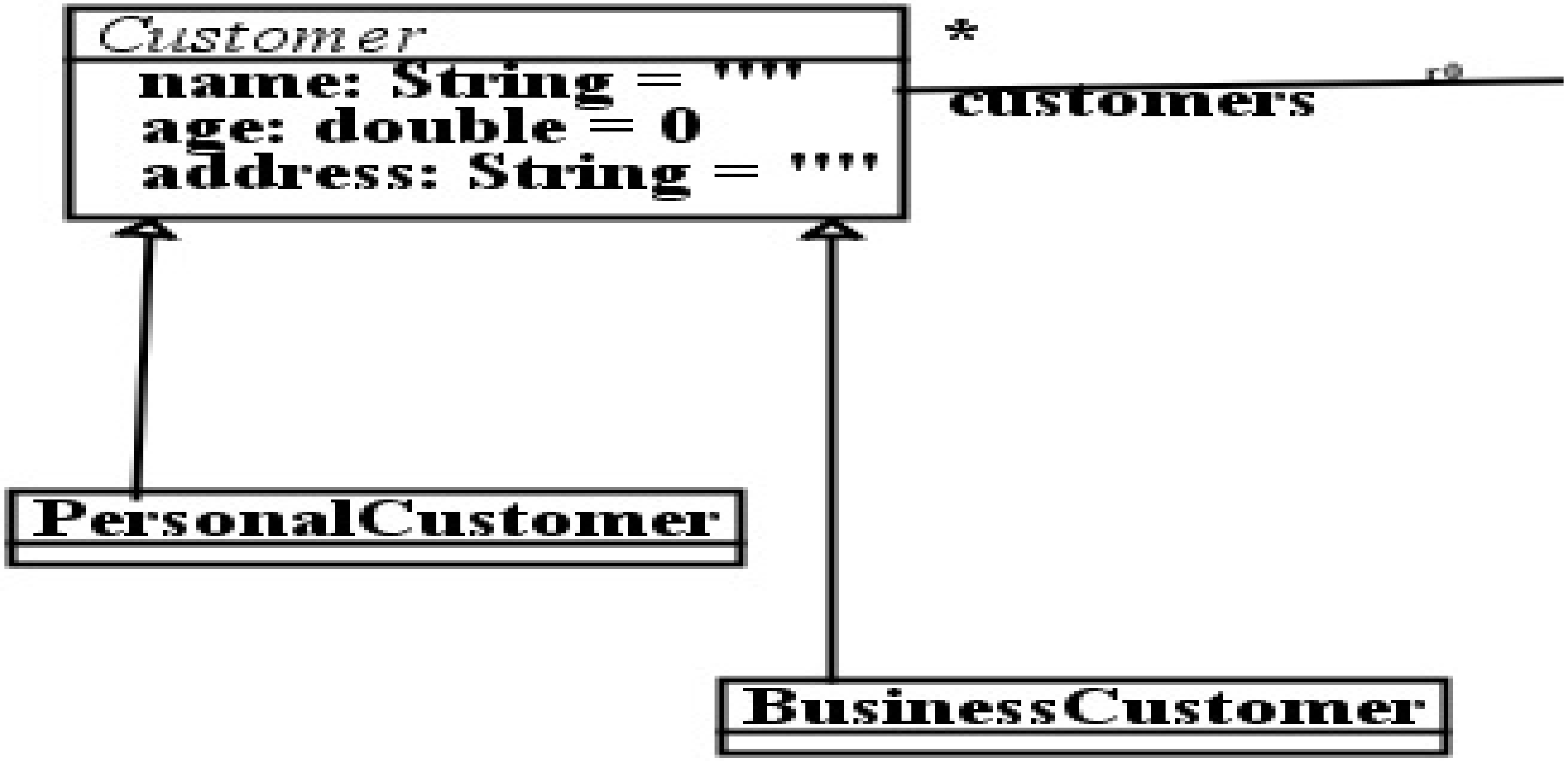
Aggregation (Composition)

- Models situation where one class has a whole-part relation to another (eg., car-wheels)
- Represented by black diamond at 'whole' end
- Semantic effect is that if a 'whole' object is deleted, so are all its linked parts (cascaded delete)
- Multiplicity must be 1 or 0..1 at 'whole' end.

Aggregation example

Inheritance

- Define specialisation/generalisation relationships between entities
- Inheritance arrow points from subclass (specialised entity type) to superclass (generalised entity)
- No rolenames/multiplicities on the line
- Superclass is usually *abstract*: instances cannot be created for it, only for subclasses. Name written in italic font.



Example of inheritance

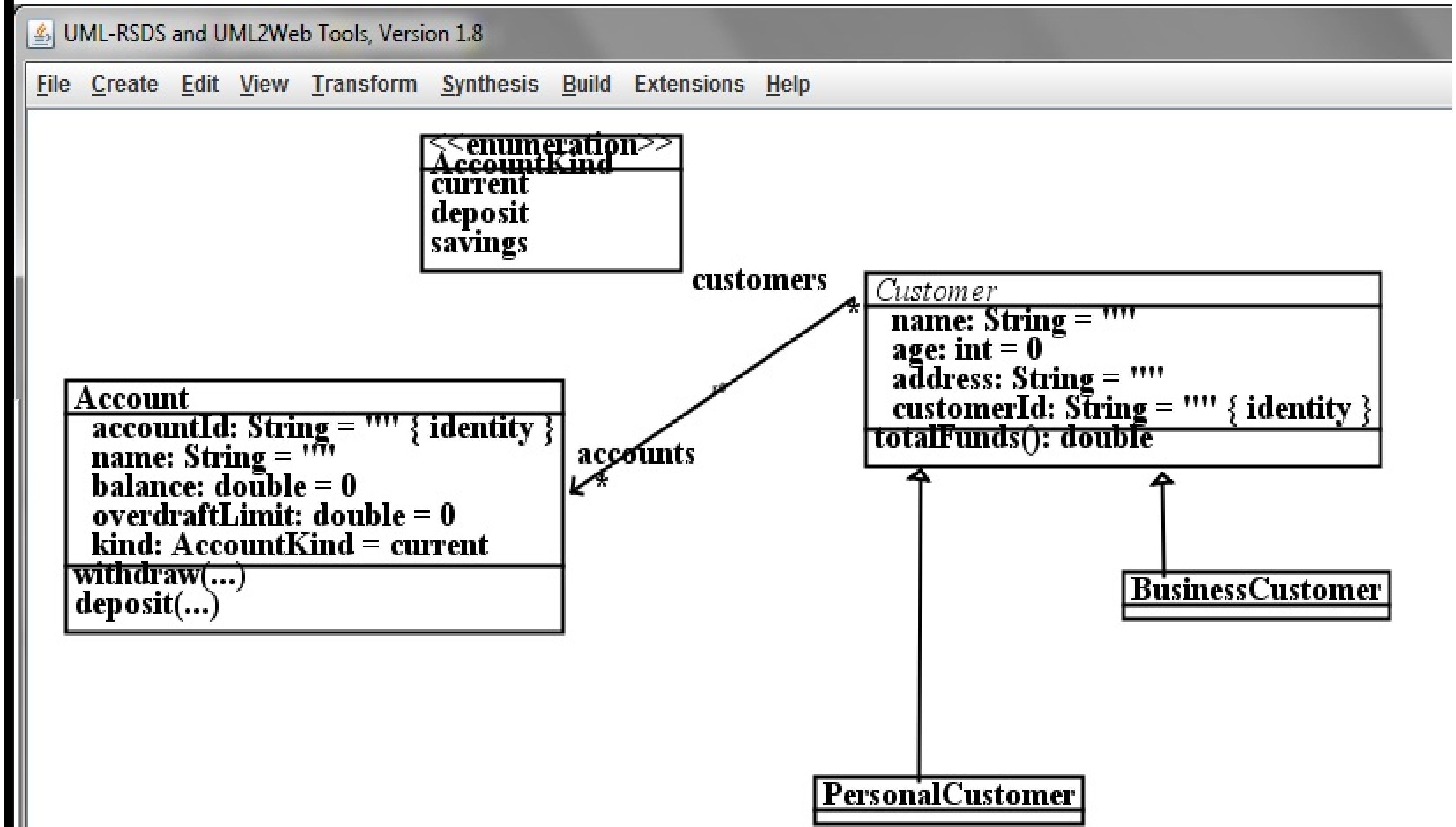
Inheritance

- Can have *multiple subclassing*: several specialisations of same superclass (eg., *PersonalCustomer* and *BusinessCustomer* as subclasses of *Customer*)
- More unusual to have *multiple inheritance*: several superclasses of one subclass (eg., *HouseBoat* as subclass of *Residence* and *Boat*)
- All features of all superclasses are inherited by a subclass
- Same concept as Java `extends` and C++ `public` inheritance.

Operations

- Operations of a class are either *query operations*: return a value and do not update the object state or *updaters*: modifying object state (and may return a value)
- Operations can be specified by *preconditions* and *postconditions*: expressions that define assumptions at start of execution, and define result state at end
- Alternatively, behaviour can be defined by statemachine or activity/pseudocode.

$totalFunds() : double$ is a query operation. $withdraw(amt : double)$ and $deposit(amt : double)$ are updaters.



Classes with operations

Operations

```
query totalFunds() : double
pre: true
post:
  result = accounts->collect(balance)->sum()
```

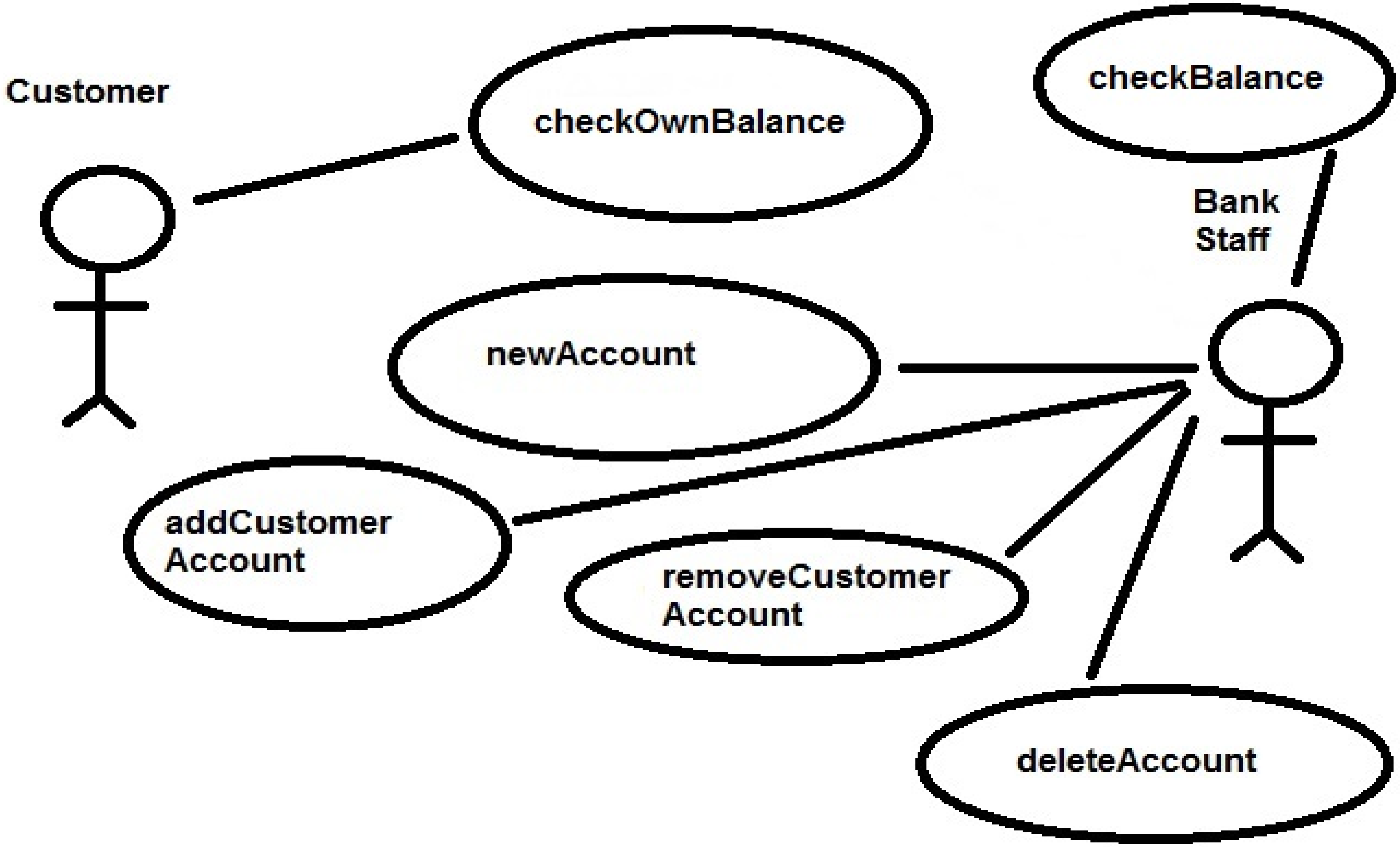
Precondition *true* means operation always available.

```
withdraw(amt : double)
pre: balance - amt  >=  -overdraftLimit
post:
  balance = balance@pre - amt
```

Operation should only be invoked if $balance - amt \geq -overdraftLimit$. The amt is then subtracted from $balance$.

Use Cases

- Define functionalities of system, the services it provides to users
- Each use case has name, written in oval, linked to agents/actors who interact with the case
- For banking system, should be use cases *checkBalance* (for staff), *checkOwnBalance* (for customer), *newAccount*, *addCustomerAccount*, *removeCustomerAccount* and *deleteAccount*.



Banking system use cases

Banking system use cases

- $checkBalance(aId)$ for staff – displays balance of the account
- $checkOwnBalance(cId, aId)$ for customers – only displays balance of one of customer's own accounts
- $newAccount(aId)$ creates account if it doesn't already exist
- $addCustomerAccount(cId, aId)$ links customer + account
- $removeCustomerAccount(cId, aId)$ unlinks them
- $deleteAccount(aId)$ deletes the account, if it has no customers.

Deriving use cases from user stories

- User stories are main form of behaviour requirement used in agile methods.
- Express unit of functionality which user of system expects from system.
- Expressed in format

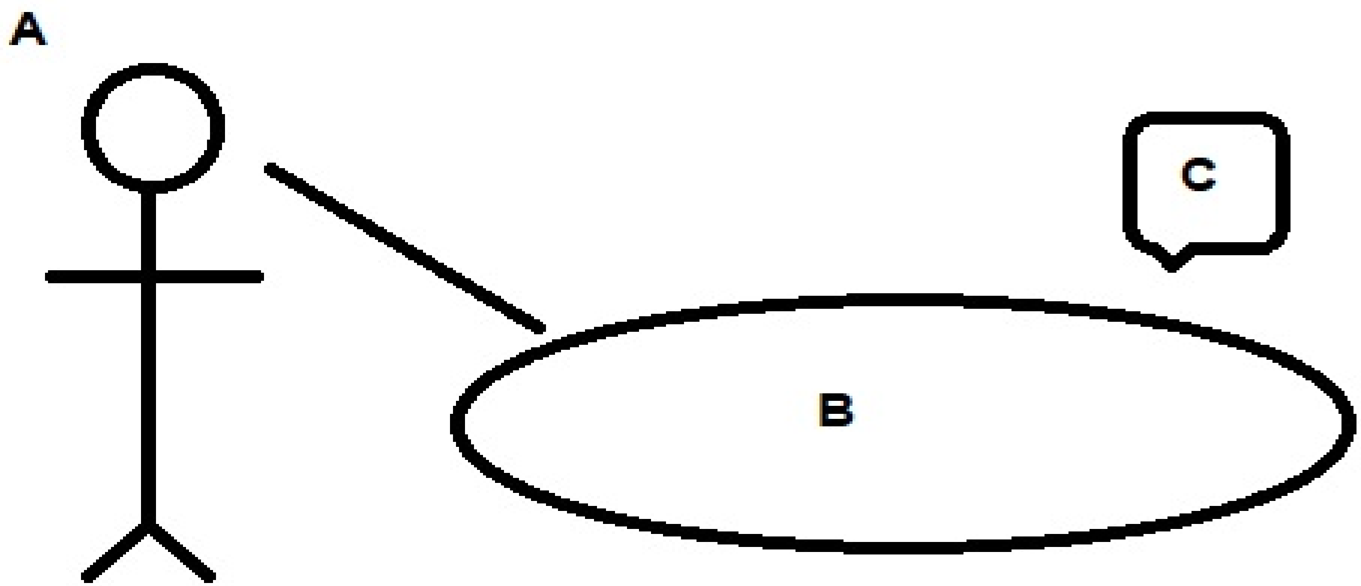
  $$[\textit{actor identification}]\ \textit{goal}\ [\textit{justification}]$$

  where actor identification and justification are optional.

For example:

"As an A, I wish to B, in order to C"

Derived use case has actor A, goal describes intended use case actions, justification explains use case purpose.



Use case derivation from user stories

Use Cases

- Use cases defined by sequence of steps, which perform operations on objects of system
- Use cases coordinate object behaviours to produce overall required functionality
- Eg., *checkBalance(aId : String)*:

```
Lookup account with accountId = aId;
Display balance of this account;
```

- *createAccount(cId : String, aId : String)*:

```
Lookup customer with customerId = cId;
Create a new account with accountId = aId;
Add this account to the customer accounts;
```

Use Cases

- Use cases can have *preconditions*, defining when they are valid to execute
- Eg., *createAccount(cId : String, aId : String)* has precondition that no account already exists with accountId = aId
- Use cases can have *postconditions*, defining effect and result by a series of expressions.

OCL (Object Constraint Language)

- Expression language used with UML
- Defines logical conditions, for preconditions, postconditions, invariants, etc
- Defines values of numeric, string, entity or collection types
- Precisely define operation and use case functionalities.

OCL (Object Constraint Language)

- Numeric value types *Integer*, *Real*. String value type *String*
- Usual numeric operators +, -, *, /, $<$, $>$, $\leq$, $\geq$, etc
- Numeric functions $r.sqrt()$, $r.cos()$, $r.pow(p)$, etc
- String functions $s.size()$, + (concatenation), $s.toLowerCase()$, etc.

OCL (Object Constraint Language)

Entity types:

- If E is a class diagram entity type, instances $e : E$ can be used in OCL expressions, and features $e.att$, $e.role$. Objects can be compared with $=$, $/=$
- A constraint with *context* E can refer directly to features of E, eg.:

  ```
  Account::
       balance >= -overdraftLimit
  ```

  as invariant of *Account*
- *self* object.

OCL (Object Constraint Language)

Collection types:

- $Set(T)$ is type of sets of T: a set $Set\{v_1, ..., v_n\}$ is an unordered collection of elements, with no duplicates (each element occurs only once)
- $Sequence(T)$ is type of sequences of T: a sequence $Sequence\{v_1, ..., v_n\}$ has elements in the listed order. Elements can occur multiple times.
- Eg., $Set\{1, 9, 9, 1\}$ only has 2 elements: $Set\{1, 9\}$, whilst $Sequence\{1, 9, 9, 1\}$ has 4 elements.

Matrix type: $Sequence(Sequence(double))$.

OCL (Object Constraint Language)

Collection operators use $\rightarrow$ symbol:

- $s \rightarrow size()$ is size of collection s
- $s \rightarrow sum()$ is sum of elements of collection s (of numbers/strings)
- $s \rightarrow prd()$ is product of elements of collection s (of numbers)
- $x : s$ is true if s contains element x, false otherwise. Also written as $s \rightarrow includes(x)$
- Eg., $9 : Set\{1, 9, 9, 1\}$
- Eg., $Sequence\{1, 9, 9, 1\} \rightarrow sum() = 20$

OCL (Object Constraint Language)

Collection operators:

- $s \rightarrow collect(e)$ is collection of elements $x.e$ for x in collection s
- $s \rightarrow select(x \mid P)$ is subcollection of collection s, containing the $x : s$ that satisfy P
- $s \rightarrow reject(x \mid P)$ is subcollection of collection s, containing the $x : s$ that do not satisfy P
- Eg., $accounts \rightarrow collect(balance)$ is sequence of *balance* values for a customer's accounts
- Eg., $accounts \rightarrow select(balance \geq 0)$ are the accounts that are not overdrawn.

Operators can be chained, eg: $accounts \rightarrow collect(balance) \rightarrow sum()$ is sum of balances of accounts of a customer.

OCL (Object Constraint Language)

Quantifiers:

- $s \rightarrow forAll(x \mid P)$ is true if every element x of collection s satisfies P, false otherwise
- $s \rightarrow exists(x \mid P)$ is true if some element x satisfies P, false otherwise
- Eg., $Set\{1, 9, 9, 1\} \rightarrow forAll(x \mid x \leq 10)$ is true.

Relation of OCL to mathematics and Python

Mathematical concept	*OCL notation*	*Python notation*
Set type $\mathbb{F}(T)$	$Set(T)$	`set`
Sequence type $seq(T)$	$Sequence(T)$	`list`
Select $\{x \in s \mid P\}$	$s \rightarrow select(x \mid P)$	`{ x for x in s if P }`
Collect $\{x \in s \bullet expr\}$	$s \rightarrow collect(x \mid expr)$	`{ expr for x in s }`
$\forall x \in s \bullet expr$	$s \rightarrow forAll(x \mid expr)$	`all([expr for x in s])`
$\exists x \in s \bullet expr$	$s \rightarrow exists(x \mid expr)$	`any([expr for x in s])`
Membership $x \in s$	$s \rightarrow includes(x)$	`x in s`
Union $s1 \cup s2$	$s1 \rightarrow union(s2)$	`s1.union(s2)`
Intersection $s1 \cap s2$	$s1 \rightarrow intersection(s2)$	`s1.intersection(s2)`
Size $\#s$	$s \rightarrow size()$	`len(s)`

Collection types and operators

OCL and data analysis

Data analytics may combine *filter*, *map* and *reduce* steps:

$$data \rightarrow select(x \mid P) \rightarrow collect(e) \rightarrow forAll(Q)$$

"For all the x in *data* that satisfy P, the $x.e$ value satisfies Q".

$$data \rightarrow select(x \mid P) \rightarrow collect(e) \rightarrow max()$$

"Find the maximum $x.e$ value from $x : data$ satisfying P"

For very large datasets, can be computed using Map/Reduce (Part 5).

OCL (Object Constraint Language)

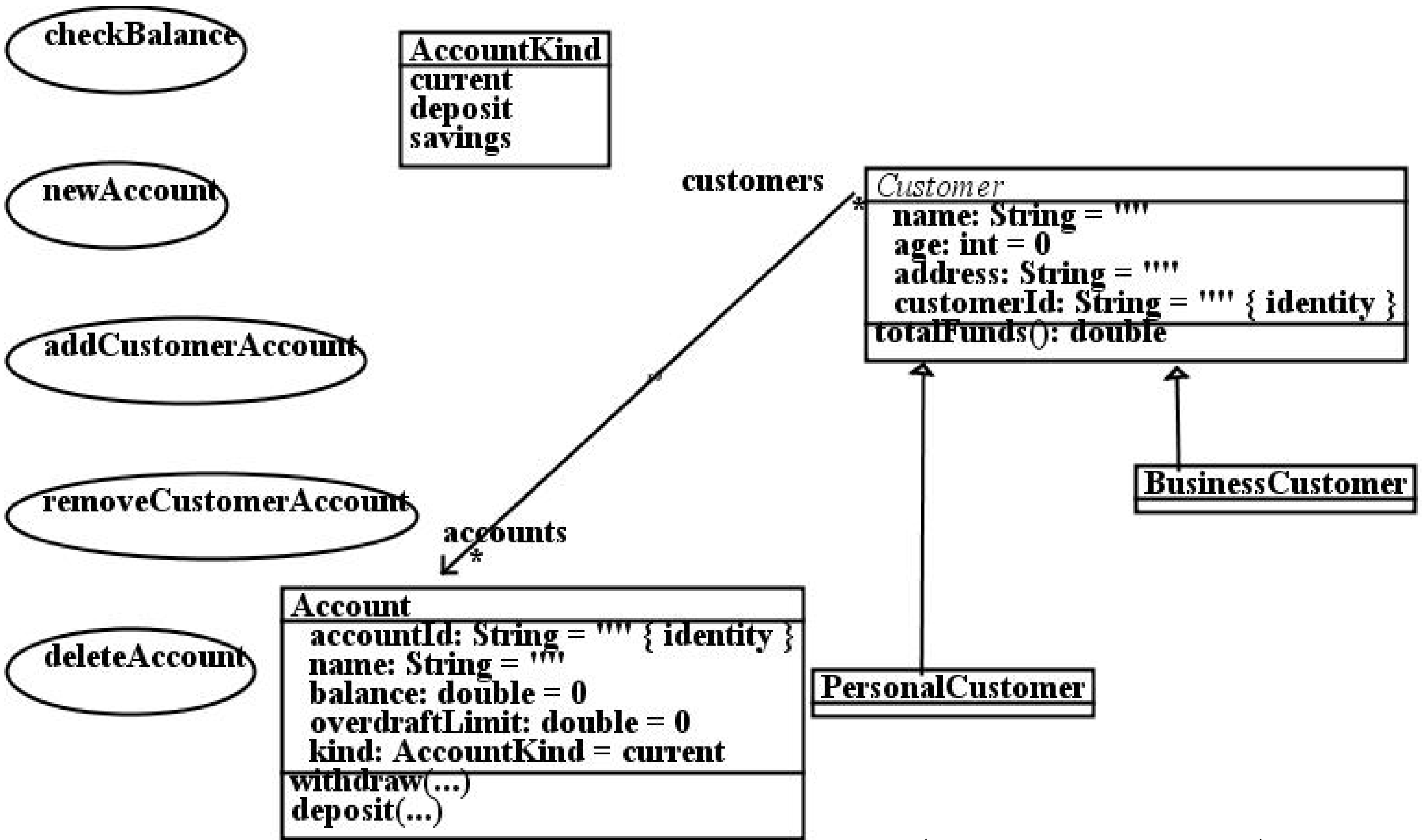
Expressions can define behaviours:

- $x = v$ can be interpreted as "Set value of x to v"
- $x : s$ can be interpreted as "Add x to s"
- $s{\rightarrow}forAll(x \mid P)$ as "Make P true for every element x of s"
- $E{\rightarrow}exists(x \mid P)$ for concrete entity type E as "Create an instance x of E and initialise it to satisfy P"

OCL can be used for financial specification, similar to Matlab or Excel.

Combining use cases and class diagrams

- Use cases read + update data from class diagram, may call operations of classes
- Can show use cases on class diagram, without actors, to give complete system specification (data + functionality).

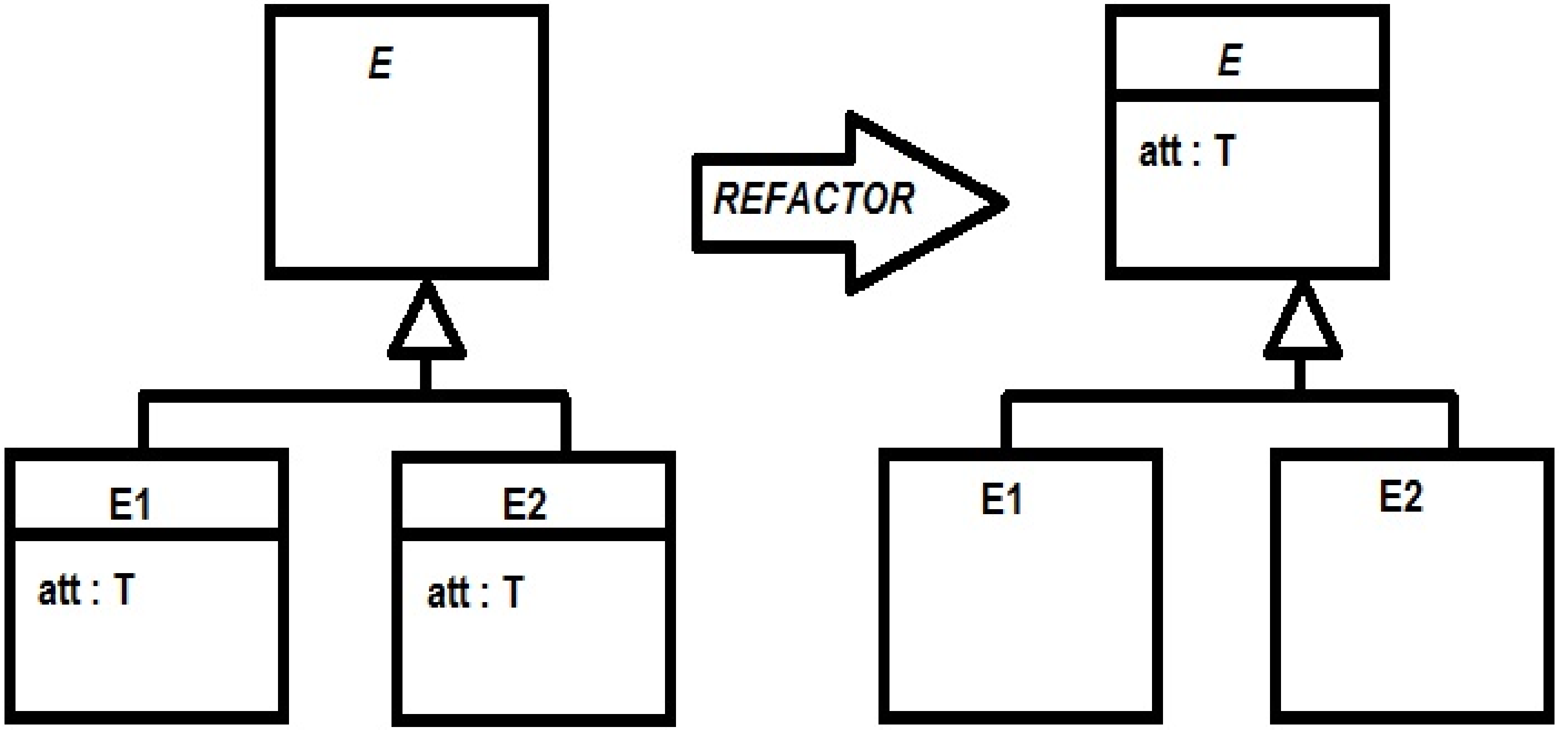


Completed banking system specification (staff operations)

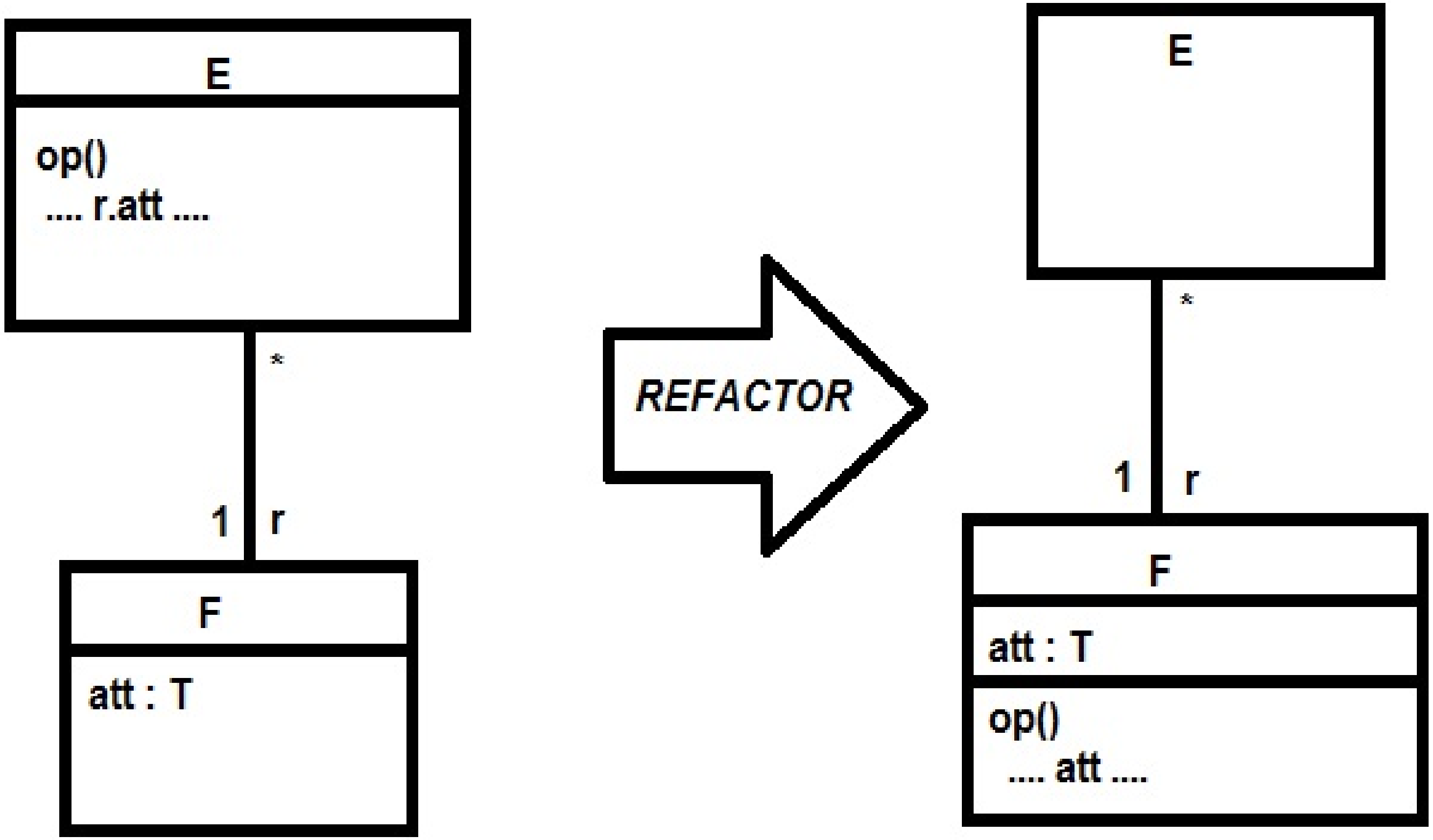
Specification revision/refactoring

Class diagrams can be refactored to improve structure, remove redundancies and improve correspondence to requirements:

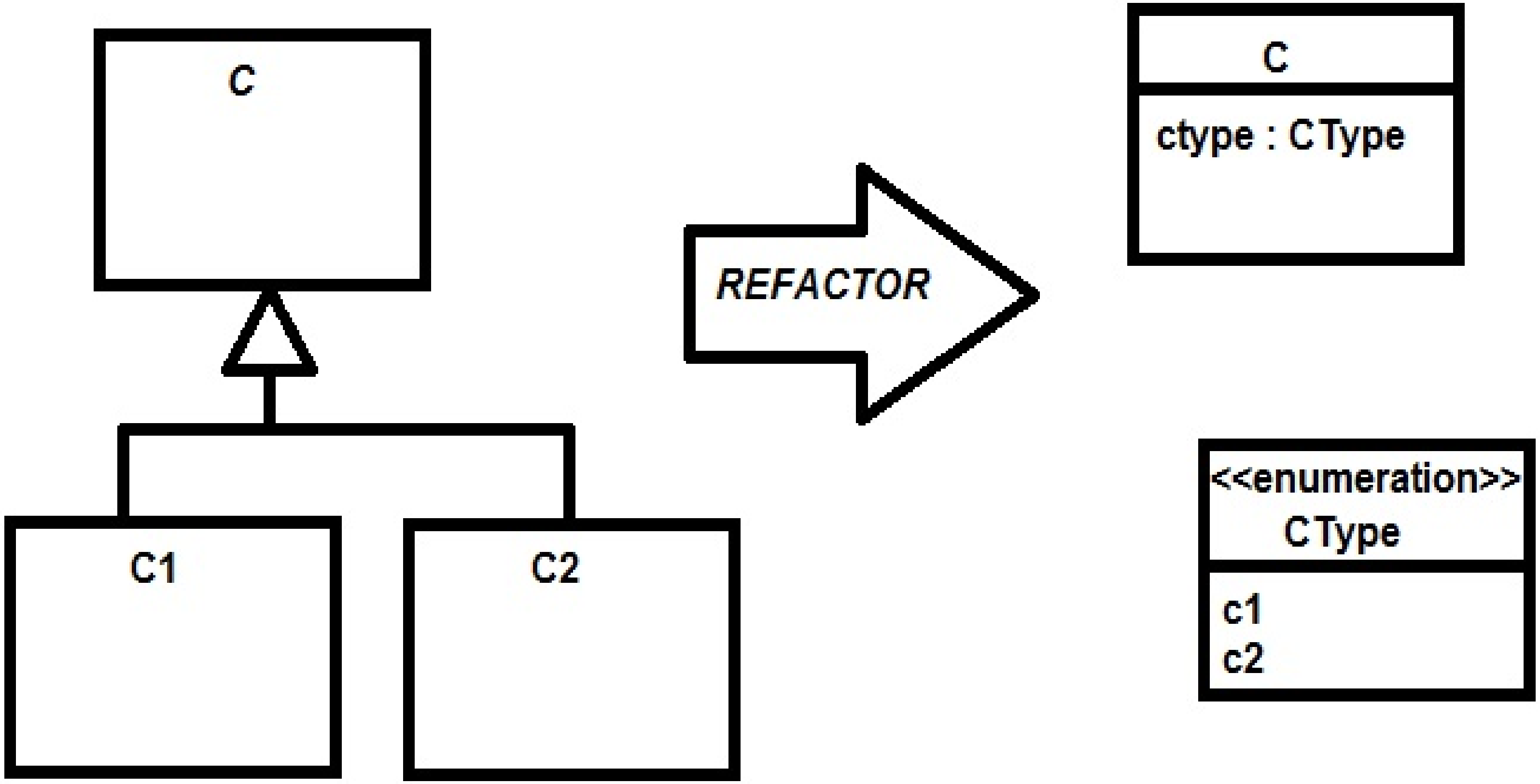
- "Pull up attribute" refactoring: if all (at least 2) direct subclasses of class E declare attribute $att : T$, replace by single definition in E
- "Move operation" refactoring: if operation op of E refers to attributes/roles of class F via association E—F, try moving op to F
- "Merge classes": if many subclasses of a class C, all empty, replace by flag attribute of C of enumerated type – eg., case of $Customer$.



Pull up attribute refactoring



Move operation refactoring



Merge classes refactoring

Summary of Part 3 (so far)

- Introduced essential UML class diagram notations
- Introduced use case concepts
- Introduced core OCL features and uses
- Considered class diagram refactoring.

In the remainder of Part 3, will consider larger specification examples.

Part 3 continued: Specification examples in UML

- Monte-Carlo share price simulator
- Bond pricing
- Yield curve fitting
- Derivative securities valuation.

Typically, for financial applications, UML class diagrams + OCL specifications express relevant financial theory, in computational terms.

OCL specifications can also express algorithms in platform-independent manner.

Example: Monte-Carlo simulation of share prices

- Model share price changes as *stochastic process* for change ΔS of price over period Δt:

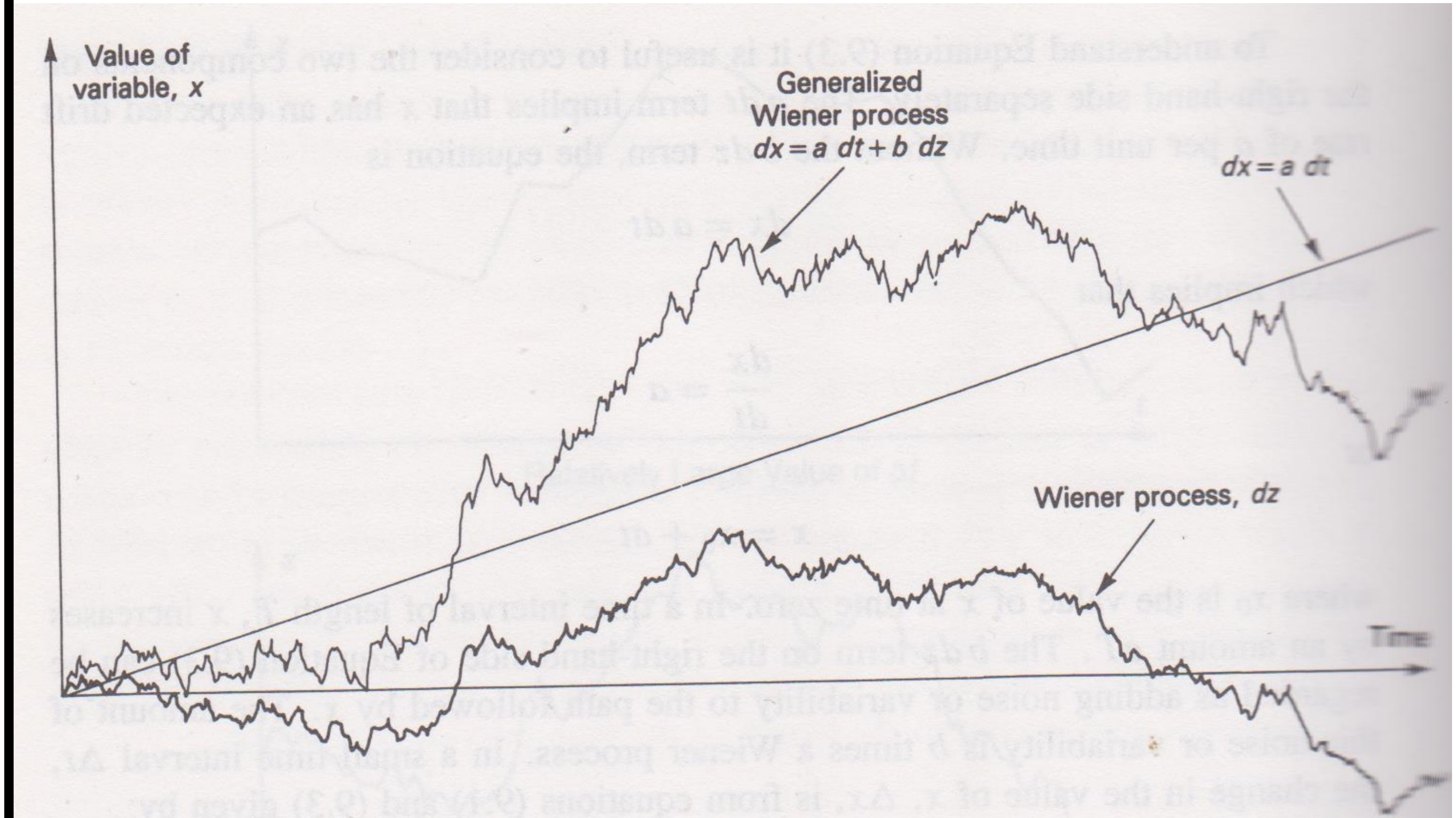
  $$\frac{\Delta S}{S} = \mu\,\Delta t + \sigma\eta\sqrt{\Delta t}$$

  μ is expected rate of return; σ price volatility; η a sample from normal distribution N(0,1)

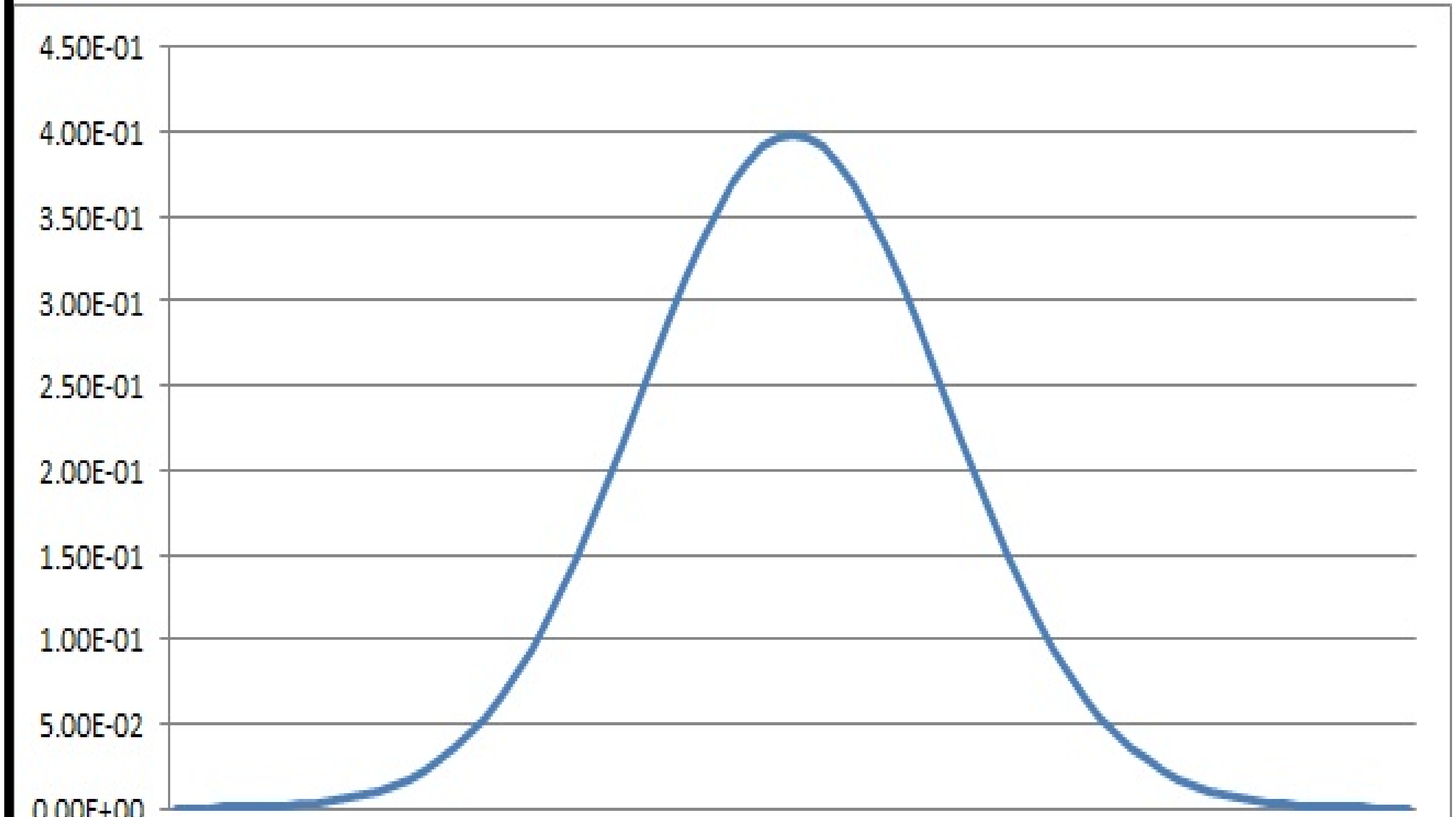
- Monte Carlo simulation generates possible price trajectories using equation

  $$S' = S + S\,\mu\,\Delta t + S\sigma\eta\sqrt{\Delta t}$$

- Take average of final price result of many possible runs.



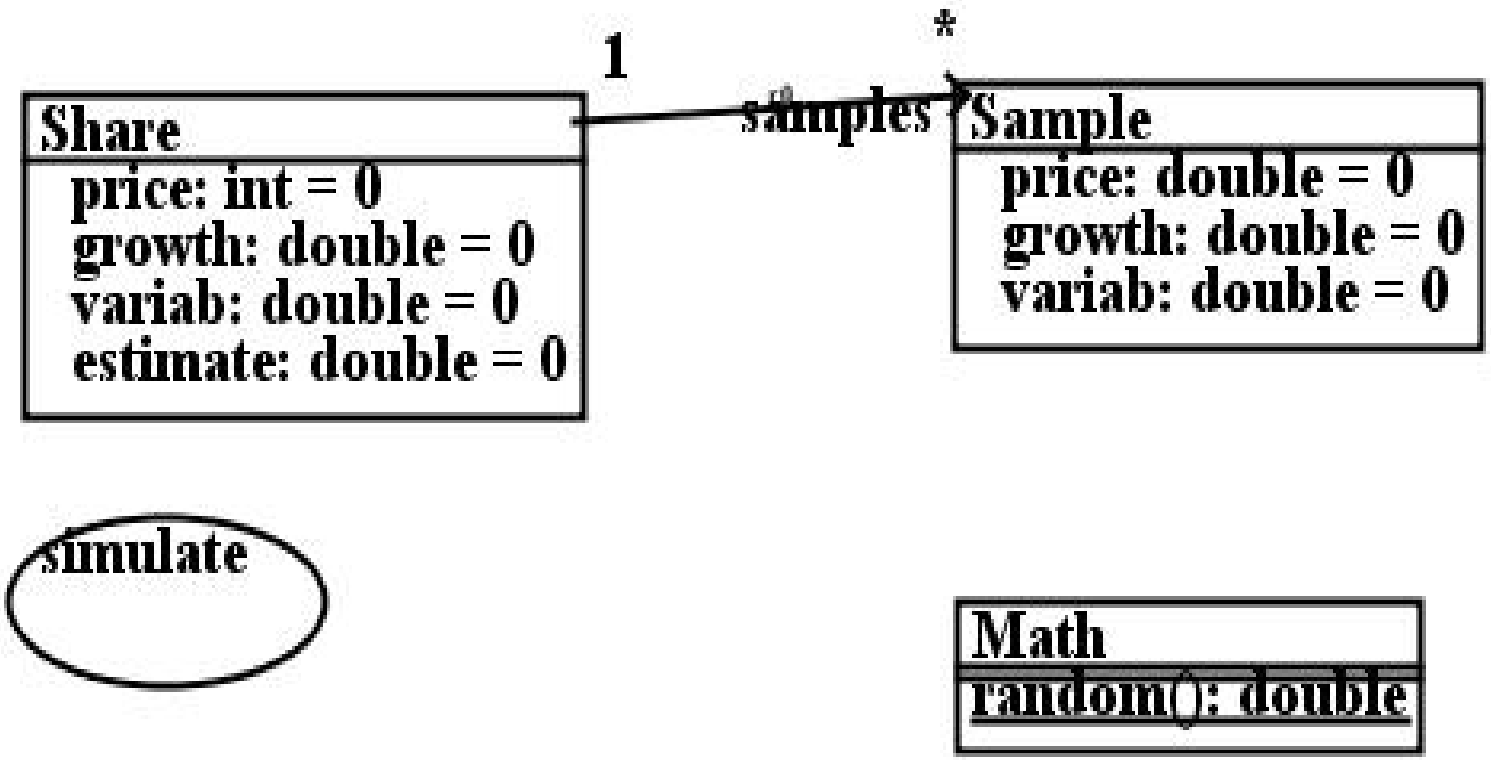
Stochastic model of share prices (cf. ‘Brownian motion’)



Normal distribution $N(0,1)$

Normal distribution

- Curve represents probability of a 'particle' movement as a sum of a large number of +ve and -ve impulses
- The most likely overall movement is 0
- Small movements are more likely than large
- Graph is symmetrical
- In terms of share prices the 'impulses' are share trades selling/buying the share – graph represents probability of different overall price movements
- Because share prices cannot be -ve, the alternative lognormal distribution is used to model the price itself.



Monte Carlo simulation system

Simulator specification

- Class *Sample* models *price* of share at a time point. *growth* represents μ (per unit time interval). *variab* represents σ.
- Class *Share* also has *estimate* for future price estimate, and a set *samples* of *Sample* objects.

Math is stereotyped as *external* class to indicate it is coded elsewhere (in Java library).

Simulator specification

- Functionality is expressed by use cases
- Each use case has a sequence of postconditions, expressing sequence of execution stages/steps
- Postconditions are written in OCL, *Object Constraint Language*, of UML
- Thus functionality is expressed in platform-independent manner.

Simulator specification

Use case *simulate* has parameters $price : double$, $growth : double$ and $varib : double$, and has three postconditions/stages.

The first produces 10 samples for each share instance:

```
Share::
  Integer.subrange(1,10)->forAll( j |
    Sample->exists( s | s.price = price &
                        s.growth = growth &
                        s.variab = variab & s : samples ) )
```

$Integer.subrange(a, b)$ is the range $a..b$ of integers. For $j = 1, 2, ..., 10$ a new sample s is created, initialised, and added to samples for the share.

Simulator specification

A second constraint performs simulation run for each sample, over 100 time points, with η approximated by Java's $(Math.random() * 2) - 1$:

```
Sample::
  Integer.subrange(1,100)->forAll( t |
     price = price@pre + price@pre * growth +
        price@pre * variab * ((Math.random()*2) - 1))
```

This computes each successive new price S' (*price*) from previous price S (*price*@pre).

Finally, overall estimate for price at time 100 is calculated as average of sample prices:

```
Share::
  estimate = (samples->collect(price)->sum())/10.0
```

Simulator specification

$samples \rightarrow collect(price)$ is sequence of all price values $s.price$ for s in $samples$ – duplicate values are preserved.

From specification, an implementation in Java can be generated using UML-RSDS tools: Generate Design option (Synthesis menu) and Java4 (Build menu).

E.g., with starting price of 100, growth per month as 0.1, variability as 0.05, 10 samples produce range of price estimates from 105 to 114, and average of 111.

Simulator specification

A postcondition constraint

```
E::
  Assumption => Conclusion
```

defines behaviour

```
for (self : E)
do
  if Assumption then Conclusion
```

This tries to establish $Assumption \Rightarrow Conclusion$ for each existing instance of E.

A *Conclusion* of the form

```
  s->forAll( x | P )
```

defines behaviour

```
for (x : s)
do P
```

This tries to establish P for each element x of s.

Compare: std::for_each(s.begin(), s.end(), fP) in C++ STL.

A *Conclusion* of the form

```
  E->exists( x | P )
```

for concrete class E defines behaviour

```
E x := new E();
P
```

In general, a conclusion A & B defines sequencing of A followed by B.

Simulation specification: review

The UML specification is both a formalisation of requirements, *and* a definition of a design/implementation. Should have properties:

- Specification should express intended functionality concisely & clearly, in language-independent manner.
- Specification should be easy to evolve and reuse.
- Specification should define an efficient solution.

Simulation specification: review

- Specification is concise and clear, but depends on Java Math.random(). Also incorrect, since a normal distribution is needed, not uniform distribution.
- Number of samples + iterations is hard-coded – these should be parameters.
- Developer should generate code + check efficiency for large execution situations.

Simulation specification: revision

Make use case reusable by defining 5 parameters:

- *currentPrice* : *double*
- *timeDays* : *int* – length of period considered for the prediction
- *growthRate* : *double* per day
- *variation* : *double* per day
- *runs* : *int* – number of samples to be generated

A *result* parameter gives estimated future price as a double. Name of use case is changed to more meaningful *estimateFuturePrice*, and constraints modified to be:

```
::
  Share->exists( s | s.price = currentPrice &
                 s.growth = growthRate & s.variab = variation )
```

```
Share::
  Integer.subrange(1,runs)->forAll( j |
        Sample->exists( s | s.price = price & s.growth = growth &
                              s.variab = variab & s : samples ) )

Sample::
  Integer.subrange(1,timeDays)->forAll( t |
         price = price@pre + price@pre * growth +
                    price@pre * variab * NormalDist.sample() )

Share::
  estimate = (samples->collect(price)->sum()) / runs

::
  result = Share.estimate->any()
```

Final constraint copies estimated price of the single *Share* instance to *result* parameter of the use case.

NormalDist is a language-independent library defined in OCL.

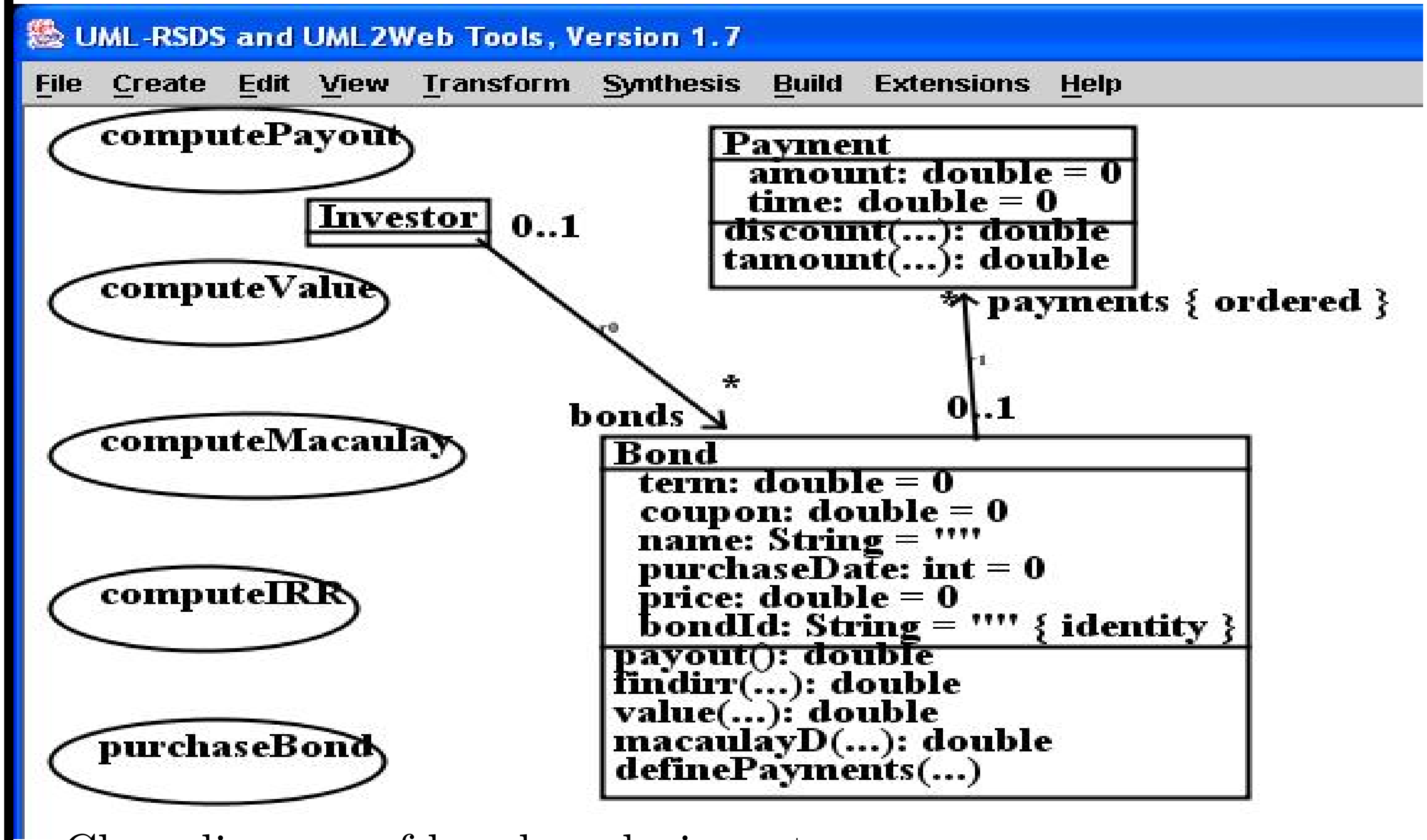
Another alternative could be to use Apache math library class *NormalDistribution* and operation *sample* to obtain samples from N(0,1).

Following table shows time complexity of this version, using $initialPrice = 100$, $timeDays = 50$, $growth = 0.1$, $variability = 0.09$, and varying number of samples. Conclude that use case is of adequate efficiency to use in practice.

Number of samples	*Java 4*	*Java 7*	*C#*	*C++*
100	40ms	31ms	22ms	31ms
1000	160ms	219ms	100ms	297ms
10,000	5,140ms	7,176ms	4,546ms	2,012ms

Bond pricing

- Fixed-rate bonds are either *coupon bonds*, paying regular percentage of investment amount (usually annually or twice per year) during term, or *zero-coupon bonds*, paying accumulated coupon amounts and capital at end of term.
- For coupon bond, its *Macaulay duration* gives the term of equivalent zero-coupon bond
- Bond IRR gives the *yield* of the bond – measure of efficiency of returning value to investor
- Use market data yields and durations to derive a yield curve – fitting yield curve to the market data
- Finally, use interest rates from the yield curve to price any bond starting from current date.



Class diagram of bond analysis system

Bond properties

Assume investment amount is 100, coupon is specified as annual percentage of this.

A *coupon bond* consists of series of cash flows: an initial investment (-ve cash flow), followed by coupon payments/dividends (+ve cash flows to investor), + repayment of capital at end of term (maturity).

Eg.: a £100 bond is purchased for price of £105, with term of 10 years, pays 8% annual interest bi-annually (20 payments of £4), then £100 capital repayment (redemption) at term.

In contrast, a *zero-coupon* bond only pays back accumulated gains + capital at term.

Represent +ve cash flows as *Payment* objects.

Bond properties

The $value(r : double)$ of a bond is sum of positive cash flows, discounted by interest rate r:

$$\Sigma_{i=1}^{payments.size} payments[i].amount/(1 + r)^{payments[i].timePoint}$$

The present value of the payments received at future time points, using discrete compounding of interest.

In OCL this is:

```
value(r: double): double
pre: true
post: result = payments->collect( p | p.discount(r) )->sum()
```

where:

```
discount(r: double): double
pre: true
post: result = amount / ( ( 1 + r )->pow(time) )
```

Internal rate of return

IRR is rate r such that:

$$price = value(r)$$

r can be estimated by numerical approximation, eg., secant method or bisection.

An approach using bisection is:

```
query findirr(r: double,rl: double,ru: double): double
pre: true
post: v = value(r) &
 ( ( v > price + 0.0010 =>
      result = findirr(( ru + r ) / 2,r,ru) ) &
   ( v < price - 0.0010 =>
      result = findirr(( r + rl ) / 2,rl,r) ) &
   ( true => result = r ) )
```

Bond pricing application

This searches for r in range $[rl, ru]$. If $value(r)$ is too high, it tries instead with the midpoint of range $[r, ru]$. If too low, with midpoint of range $[rl, r]$.

In addition, also compute *Macaulay duration* of bond (time to maturity of equivalent zero-coupon bond):

$$
\begin{aligned}
&duration = \\
&\quad (\Sigma_{i=1}^{payments.size} payments[i].timePoint * payments[i].amount / \\
&\qquad (1 + yield)^{payments[i].timePoint}) / \\
&\quad value(yield)
\end{aligned}
$$

Computation uses previously-computed IRR (yield) value of each *Bond* instance.

Test data

Example test case of 8 coupon bonds ranging from 1 year to 12 year terms:

```
BondId, Settlement, Maturity, Price, Coupon, Frequency
"1" , 1999 , 2000 , 103.78 , 6.5 , 2
"2" , 1999 , 2001 , 106.72 , 8.0 , 2
"3" , 1999 , 2002 , 112.58 , 10.0 , 2
"4" , 1999 , 2003 , 98.53 , 5.5 , 2
"5" , 1999 , 2004 , 107.68 , 8.0 , 2
"6" , 1999 , 2006 , 108.46 , 8.0 , 2
"7" , 1999 , 2009 , 101.07 , 7.0 , 2
"8" , 1999 , 2011 , 93.11 , 6.0 , 2
```

The computed yields and durations are then:

```
BondId, Yield, Duration
"1" , 0.0448200657634527 , 0.9844067709167799
"2" , 0.06058085141979801 , 1.8896640585964184
"3" , 0.07333535911340823 , 2.6770012951695747
"4" , 0.05734245546660278 , 3.6423011566865124
"5" , 0.06911033292597388 , 4.235571418580719
"6" , 0.07030851391741318 , 5.531443157037469
"7" , 0.06899785164137454 , 7.352752247451705
"8" , 0.06578688900172859 , 8.611712068377704
```

Yield-curve fitting

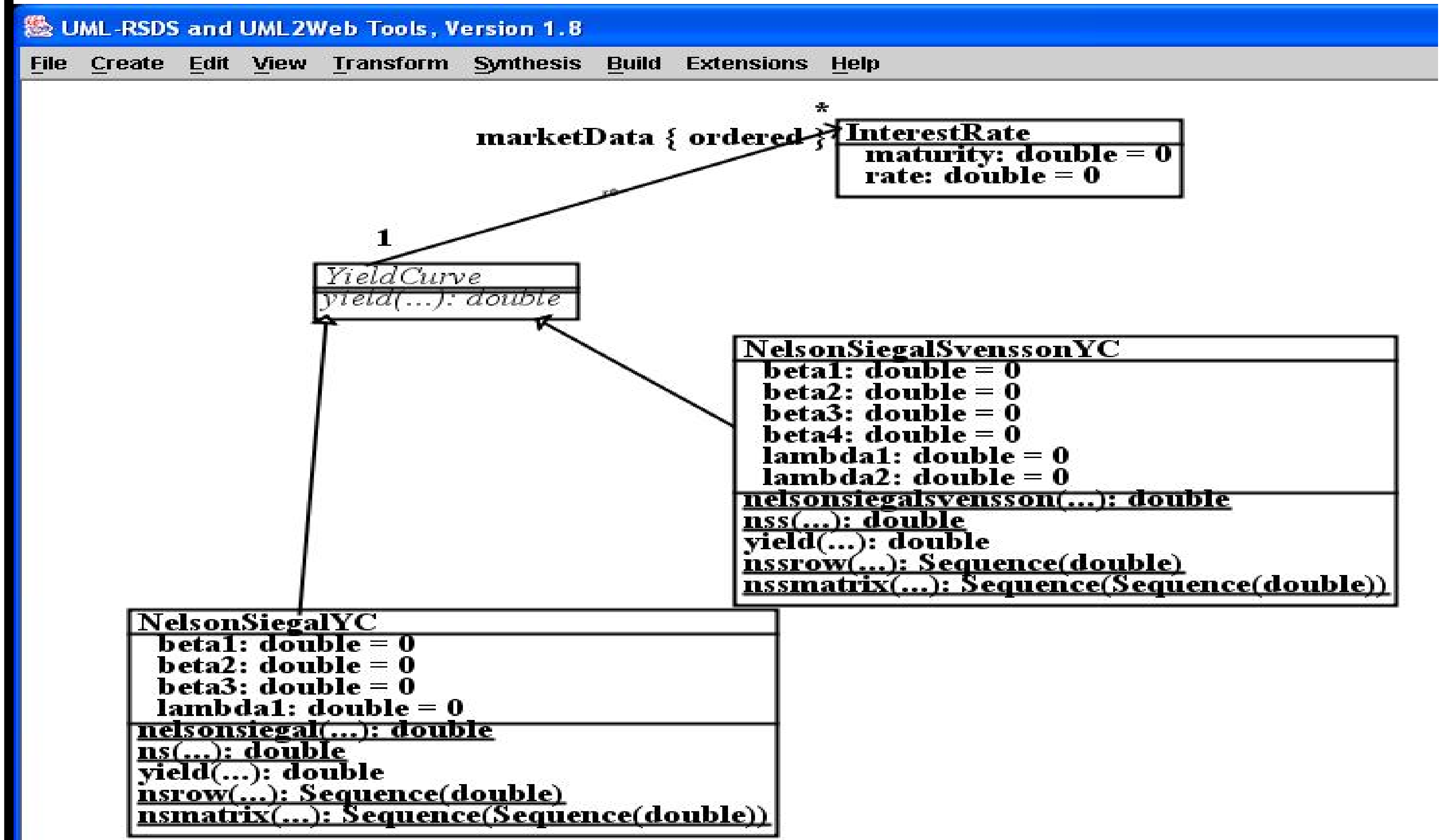
The yields and durations are then used as input datapoints for fitting a yield curve, according to yield curve model such as Nelson-Siegel or Nelson-Siegel-Svensson (NSS) models. We use adapted (NSS) model defined by equation:

$$\begin{aligned} y(t) = \beta_1 + \beta_2 * (1 - exp(-t/\lambda_1))/(t/\lambda_1) + \\ \beta_3 * ((1 - exp(-t/\lambda_2))/(t/\lambda_2) - exp(-t/\lambda_2)) \end{aligned}$$

This models how yield $y(t)$ of a bond varies depending on its duration t.

Yield curve has long-term rate component (β_1), a short-term rate ($\beta_1 + \beta_2$), and a 'hump' (3rd factor).

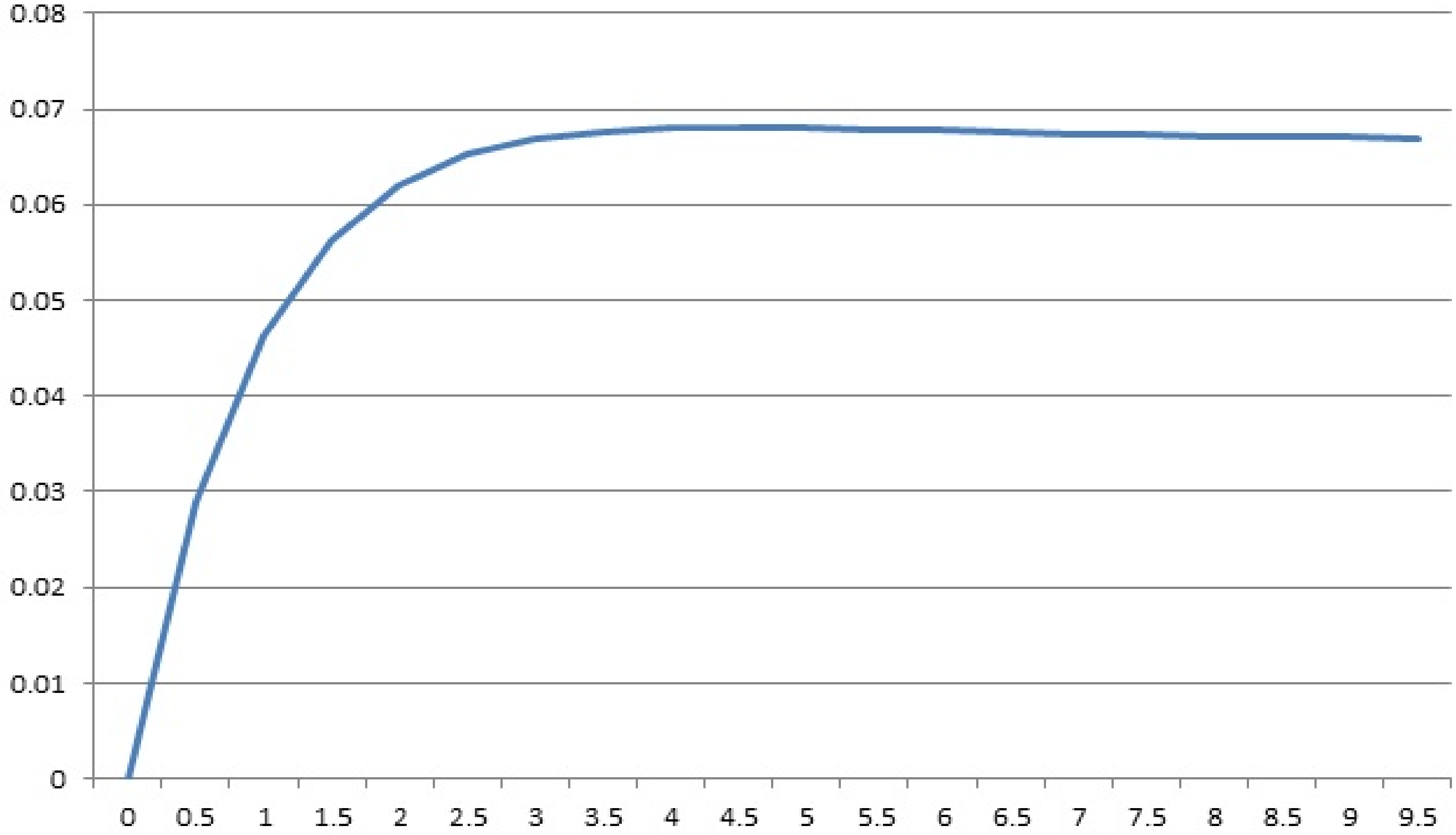
Problem is to estimate the β_i and λ_j, given market data – 'fitting the curve' to this data.



Yield curve models

Yield-curve fitting

- Estimation procedures include genetic algorithms (GA) and Matlab's `fminsearch` using a simplex algorithm.
- Having derived the NSS parameters, model can be used to calculate $yield = y(t)$ of bond of any duration $t > 0$, and hence fair price of bond as $value(yield)$.
- The market data + bond being priced should be from same issuer.
- Eg: Bank of England nominal yield curves (https://www.bis.org/publ/bppdf/bispap25m.pdf).

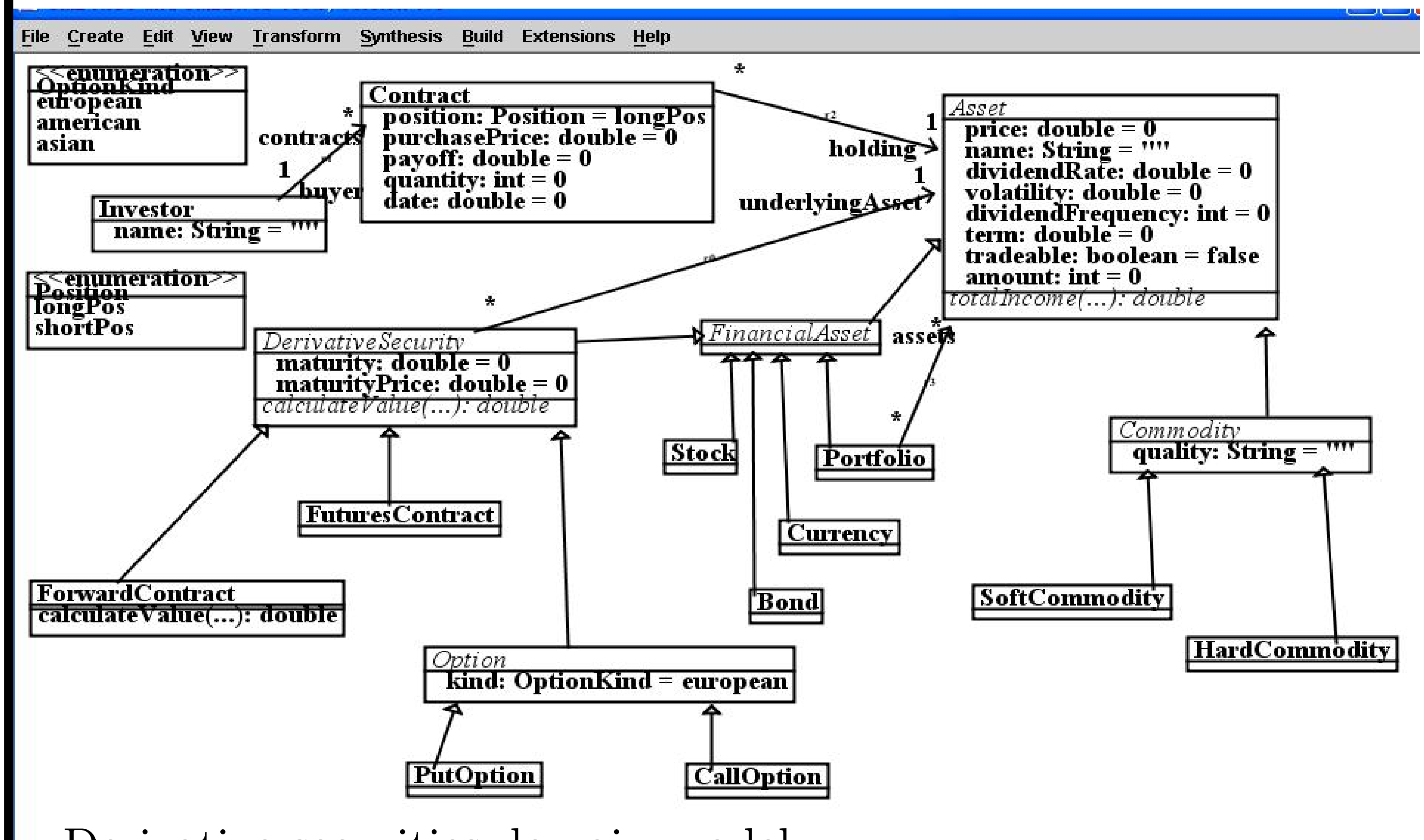


Estimated NSS yield curve

Derivative securities

Domain of derivative securities can be formally specified as a class diagram model.

- Entities include *Derivative security*, with specialisations *Option*, *Forward contract*, *Futures contract*, etc.
- Other entities could include: *Trader*, *Contract*, *Investor*, *Exchange*, *Asset*, *Commodity*, *Stock*, *Margin account*, etc.
- Associations and attributes could include: *underlyingAsset*, *repo rate*, *value*, *futures price*, *expiration date*, *delivery price*, *strike price*, *spot price*, etc.
- Operations can be defined to compute value and maturity price of each category of derivative security.



Derivative securities domain model

Derivative securities domain model

- *Contract* class expresses properties intrinsic to a contract (eg, *long* position for buyer, *short* position for seller) and properties that may be open to negotiation between investor and the supplier of a financial product (asset).
- *Asset* and its subclasses contain properties intrinsic to the product and cannot be negotiated, such as volatility.
- Attribute *maturityPrice* of *DerivativeSecurity* expresses common concept of delivery, futures and strike prices found in specific kinds of derivative security.

Instance models

An instance model shows data of specific contracts and assets:

```
a : Stock
a.name = "IBM"
a.amount = 100
a.tradeable = true
a.price = 145.16
x : CallOption
x.amount = 1
x.maturity = 2021.0
x.maturityPrice = 125
x.underlyingAsset = a
```

Instance model example: hedging

E.g.: an investor in shares wants to hedge risk of price of equity asset a

```
c : Contract
a : Stock
c.holding = a
```

being below a level P at future date $now + t$, they can also enter into contract cf : $Contract$ to hold American put options f with $f.maturity = now + t$, $cf.quantity = c.quantity$, $f.underlyingAsset = a$, and $f.maturityPrice$ equal to P.

At $now + t$, if price of a is $Q > P$, then investor will not exercise the options, and instead can sell $c.quantity$ of a at Q per unit. Profit only reduced by cost V of the options.

If $Q < P$, they exercise options and sell $cf.quantity$ of a at P (because of contract cf), making profit $(P - Q) * c.quantity - V$.

Valuation of derivative securities

- Value of each security can be calculated using a $calculateValue(now, r)$ operation, where now represents the current date, and r is relevant risk-free interest rate.
- Long forward contracts satisfy general equation (using continuous compounding):

$$value = (S - I) * e^{-q*dt} - K * e^{-r*dt}$$

where S is current price of underlying asset, q continuous dividend yield rate of the asset, T the maturity date (years), t the current date, $dt = T - t$, and I is present value of income from fixed payments from asset over period dt (eg., coupon payments of a coupon bond), K is maturity price, r is relevant risk-free rate of interest.

Value definitions

Therefore, can specify the operation definition:

```
ForwardContract::
query calculateValue(now : double, r : double) : double
post:
  dt = maturity - now &
  maturityValue = maturityPrice*((-r*dt)->exp()) &
  income = underlyingAsset.totalIncome(now, maturity) &
  dividendYield = underlyingAsset.dividendRate &
  result = (underlyingAsset.price - income) *
                ((-dividendYield*dt)->exp()) - maturityValue
```

The definition also applies for *FuturesContract*. For short forward/future contracts the value is $-calculateValue(now, r)$.

Value definitions

Maturity price K which makes the present value 0 is given by:

$$K = (S - I) * e^{(r-q)*dt}$$

This can be expressed as operation

```
ForwardContract::
query calculateMaturityPrice(now : double, r : double) : double
post:
  income = underlyingAsset.totalIncome(now, maturity) &
  dividendYield = underlyingAsset.dividendRate &
  result = (underlyingAsset.price - income) *
                ((r-dividendYield)*(maturity-now))->exp()
```

This is valid for both long and short positions in the forward/future contract.

Value definitions

For options, can use results of Black-Scholes equation. For European call option with term $dt = T - t$, on non-divided paying share asset with price S, current value is

$$c = S * N(d_1) - X * e^{-r*dt} * N(d_2)$$

where N(x) is cumulative probability distribution function for normal distribution N(0,1): area under curve from $-\infty$ to x.

d_1 and d_2 are defined in terms of S, X (maturity price), $dt = T - t$, r and σ, the volatility of the underlying asset. Also value of an American call option, assuming not optimal to exercise such an option prior to maturity.

$$d_1(S, X, dt, r, \sigma) = \frac{(S/X) \rightarrow log() \ + \ (r + \sigma.sqr/2) * dt}{\sigma * dt.sqrt}$$

and

$$d_2(S, X, dt, r, \sigma) = d_1(S, X, dt, r, \sigma) - \sigma * dt.sqrt$$

Value definitions

A European put option on such an asset has value

$$p = X * e^{-r*dt} * N(-d_2) - S * N(-d_1)$$

There is no precise formula for an American put option, but various numerical approximation techniques can be used to compute it.

If asset pays an known income over its lifetime (e.g., a share that pays known cash dividends at certain time points), then present value I of income is subtracted from S:

$$c = (S - I) * N(d_1') - X * e^{-r*dt} * N(d_2')$$

where d_1' and d_2' are computed as $d_1(S - I, X, dt, r, \sigma)$ and $d_2(S - I, X, dt, r, \sigma)$.

Corresponding put valuation is

$$p = X * e^{-r*dt} * N(-d_2') - (S - I) * N(-d_1')$$

Value definitions

If instead of a specific income, asset pays a continuous dividend ($dividendRate$) equal to q, we have valuations:

$$c = S * e^{-q*dt} * N(d_1'') - X * e^{-r*dt} * N(d_2'')$$

and

$$p = X * e^{-r*dt} * N(-d_2'') - S * e^{-q*dt} * N(-d_1'')$$

where d_1'' and d_2'' are computed as $d_1(S, X, dt, r - q, \sigma)$ and $d_2(S, X, dt, r - q, \sigma)$.

Value definitions

euPutOptionPrice can therefore be defined as:

```
static query euPutOptionPrice(s : double, x : double,
    r : double, q : double, dt : double,
    sigma : double, income : double) : double
pre: sigma > 0 & dt > 0
post:
  adjustedS = (s - income)*((-q*dt)->exp()) &
  d1 = FinLib.bsd1(s-income,x,dt,r-q,sigma) &
  d2 = FinLib.bsd2(s-income,x,dt,r-q,sigma) &
  result =
    x*((-r*dt)->exp())*NormalDist.cumulative(-d2) -
                adjustedS*NormalDist.cumulative(-d1)
```

Where:

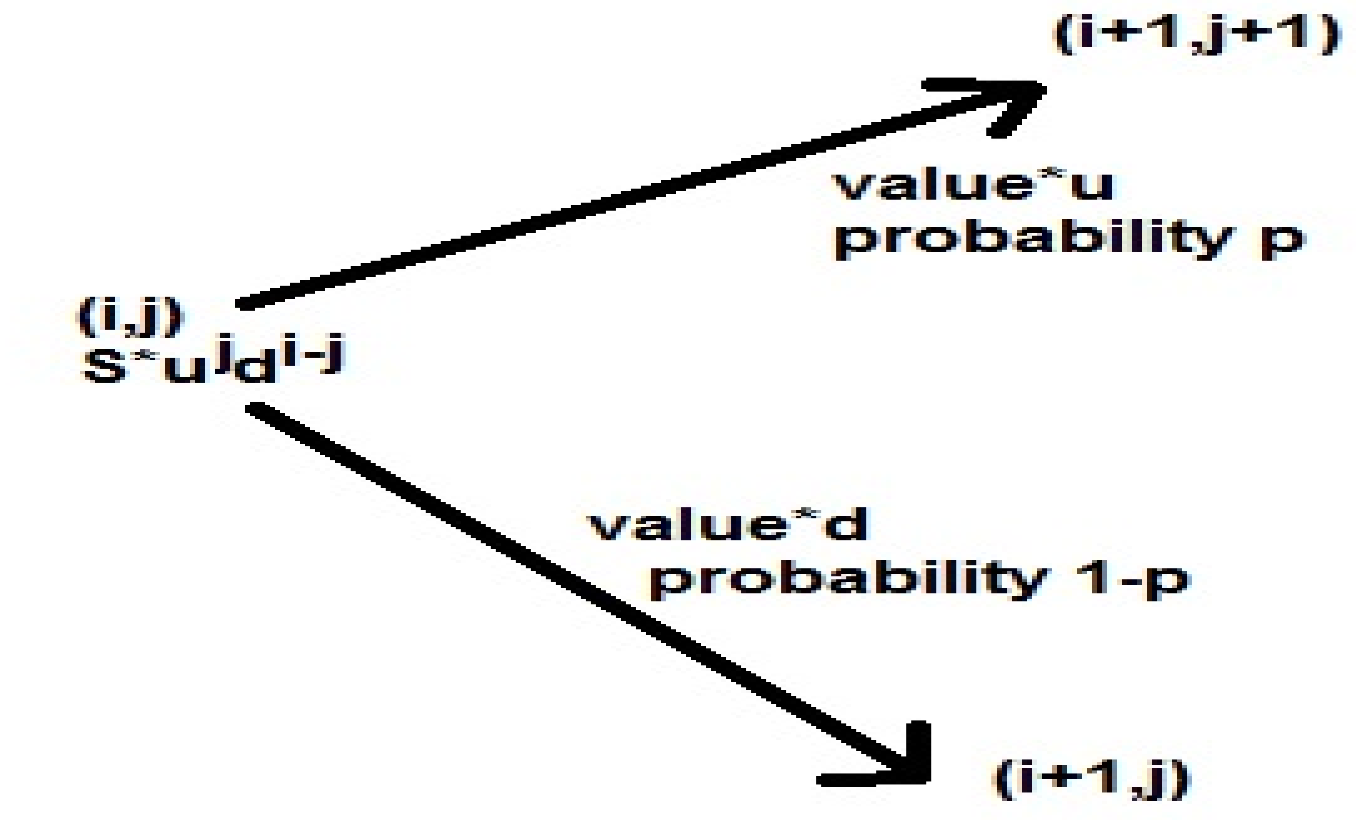
```
static query bsd1(s : double, x : double, dt : double,
      r : double, sigma : double) : double
pre: sigma > 0 & dt > 0
post:
  result = ((s/x)->log() +
            (r + sigma.sqr/2.0)*dt)/(sigma*dt.sqrt)
```

and

```
static query bsd2(s : double, x : double, dt : double,
      r : double, sigma : double) : double
pre: sigma > 0 & dt > 0
post:
  result = FinLib.bsd1(s,x,dt,r,sigma) - sigma*dt.sqrt
```

Numerical valuation of American put options

- Numerical procedure for approximate valuing of American put options based on shares can be defined, using binomial trees.
- Similar to Monte-Carlo simulation: multiple trajectories of possible share price changes are simulated over time period Δt, to obtain overall expected value of option on the share at start of Δt, based on option values at endpoints of trajectories, at maturity of the option.
- Binomial model assumes that share changes price over small time interval δt from S up to $S * u$ with probability p, or down to $S * d$ with probability $1 - p$, where $u > 1$ and $d < 1$.



Binomial tree model of share price movements

Numerical valuation of American put options

- Tree of nodes describing all possible share prices, starting at current time T_0 and terminating at time T, the *maturity* of American put option under consideration.
- Divide interval $T - T_0$ into N steps of duration $\delta t = (T - T_0)/N$ and index nodes in tree by their position (i, j)
- i is number of time steps forward from T_0, j is number of upward price movements (so $i - j$ is number of downward movements).
- Share price starts at S at T_0. By time $T_0 + i * \delta t$, at node (i, j), the price is $S * u^j * d^{i-j}$.
- Probability of each trajectory (branch of tree) is product of probabilities of its steps: p for upward steps and $1 - p$ for downward steps.

Numerical valuation of American put options

Values of u, d, p depend upon share price volatility σ, risk-free interest rate r, and size of δt. Appropriate choices are

$$\delta t = (T - T_0)/50$$
$$u = (\sigma * \sqrt{}(\delta t)) \to exp()$$
$$d = \frac{1}{u}$$
$$p = \frac{(r * \delta t) \to exp() - d}{u - d}$$

At end nodes (N, j) of tree, at time T, value $v(N, j)$ of the option is

$$Set\{0,\ X - S * u^j * d^{N-j}\} \to max()$$

since owner of put only gains from exercising it if X (strike price) is higher than actual share price at T.

Numerical valuation of American put options

At interior nodes (i, j) with $i < N$, expected value $v(i, j)$ of option is either

1. $(p * v(i+1, j+1) + (1-p) * v(i+1, j)) * (-r * \delta t) \rightarrow exp()$ if option is not exercised at this node, or
2. $X - S * u^j * d^{i-j}$ if option is exercised because it is in-the-money at this node (X is greater than share price).

Maximum of these values taken as $v(i, j)$, since exercise would not be optimal if (1) is greater than (2).

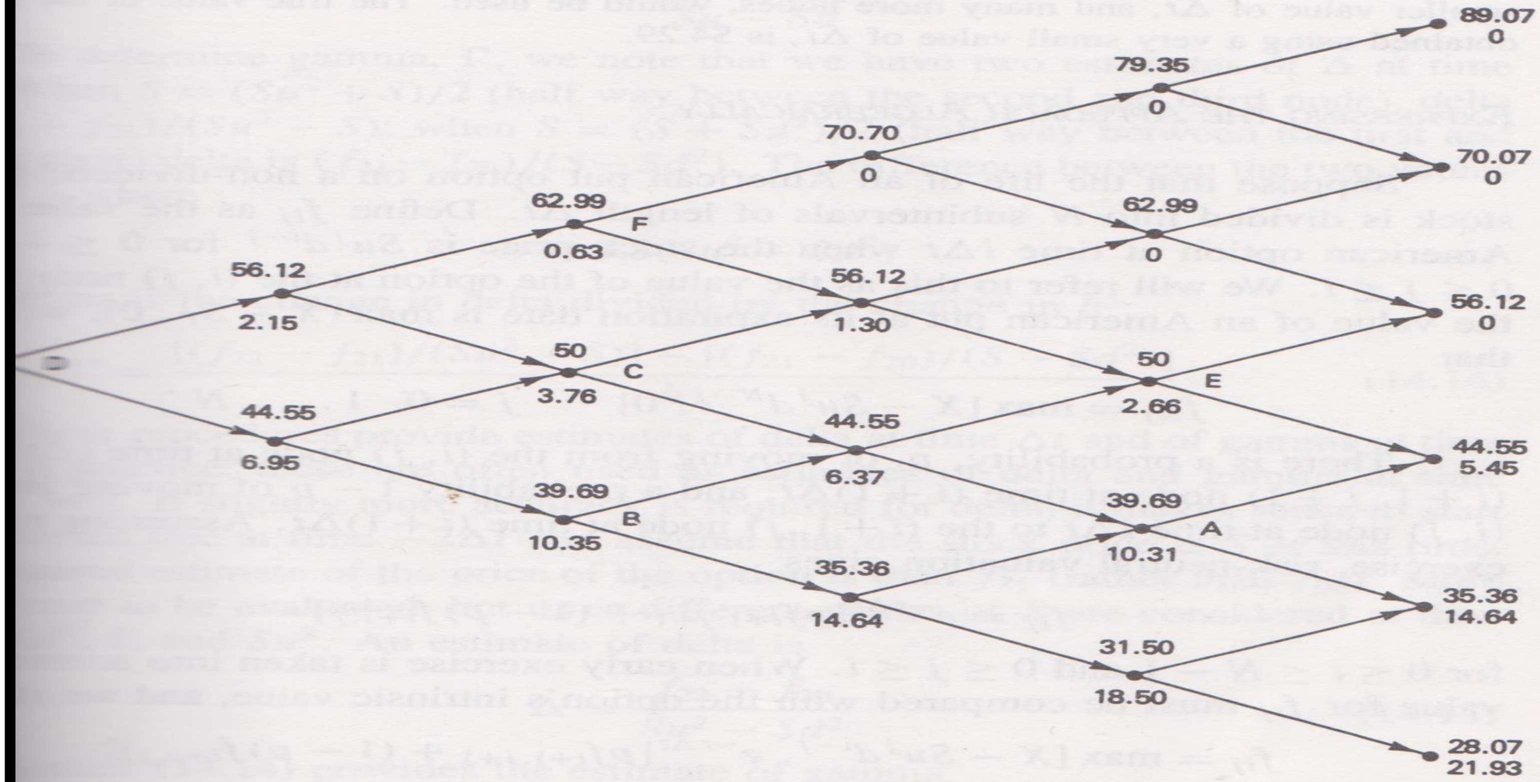


Figure 14.3 Binomial Tree for American Put on Non-Dividend-Paying Stock (Example 14.1)

Example of binomial tree, N = 5

Numerical valuation of American put options

Thus recurrence can be derived for v, as recursive definition:

```
query v(i: int, j: int, N: int, u: double, p: double,
        r : double, dt : double) : double
pre: 0 <= j & j <= i & i <= N & 0 < u & 0 < p
post:
  (i = N =>
     result = Set{0,
       maturityPrice-underlyingAsset.price*u->pow(2*j-N)}->max()) &
  (i < N =>
     result = Set{ (p*v(i+1,j+1,N,u,p,r,dt) +
         (1-p)*v(i+1,j,N,u,p,r,dt))*(-r*dt)->exp(),
         maturityPrice - underlyingAssetPrice *
            u->pow(2*j - i)}->max())
```

$v(0, 0, N, u, p, r, dt)$ is estimated put option value at time T_0.

Numerical valuation of American put options

- In practice, an iterative procedure is used to compute v over the tree, starting from leaf nodes and working backwards
- values of nodes can be stored in a matrix $values : Sequence(Sequence(double))$ where rows represent successive levels of binomial tree from root to the leaves.

Numerical valuation techniques

Binomial trees appropriate when there are holder decisions.

Monte Carlo simulation – when history of prices significant.

Option kind	*Valuation method*
European put/call	Analytic
American call	Analytic
American put	Binomial trees
Parisian	Binomial trees; Monte-Carlo simulation
Lookback	Analytic; Monte-Carlo simulation
Asian	Monte-Carlo simulation

Summary of Part 3

- Described specification techniques
- Introduced refactoring
- Illustrated specification of finance applications.