Umple Tutorial: Models 2020

Timothy C. Lethbridge, I.S.P, P.Eng.
University of Ottawa, Canada

Timothy.Lethbridge@ uottawa.ca

http://www.umple.org

Umple: Simple, Ample, UML Programming Language

Open source *textual modeling tool* and *code generator*

- Adds modeling to Java,. C++, PHP
- A sample of features
  - —Referential integrity on associations
  - —Code generation for patterns
  - —Blending of conventional code with models
  - —Infinitely nested state machines, with concurrency
  - —Separation of concerns for models: mixins, traits, mixsets, aspects

Tools

- Command line compiler
- Web-based tool (UmpleOnline) for demos and education
- Plugins for Eclipse and other tools

What Are we Going to Learn About in This Tutorial? What Will You Be Able To Do?

- Modeling using class diagrams
  - Attributes, Associations, Methods, Patterns, Constraints
- Modeling using state diagrams
  - States, Events, Transitions, Guards, Nesting, Actions, Activities
  - Concurrency
- Separation of Concerns in Models
  - Mixins, Traits, Aspects, Mixsets
- Practice with a examples focusing on state machines and product lines
- Building a complete system in Umple

What Technology Will You Need?

As a minimum: Any web browser.

For a richer command-line experience

- A computer (laptop) with Java 8-14 JDK
- Mac and Linux are the easiest platforms, but Windows also will work
- Download Umple Jar at http://dl.umple.org

You can also run Umple in Docker: http://docker.umple.org

Key Websites

Entry-point: https://www.umple.org

Github: https://github.com/umple/umple

Umple Online: https://try.umple.org

Tell Me About Yourselves:

Quick survey about

- What modeling tools do you use?
- Whether you are an academic, industrial practitioner or student
- Whether you generate code

https://www.surveymonkey.ca/r/J96KPBN

Some lists of tools:

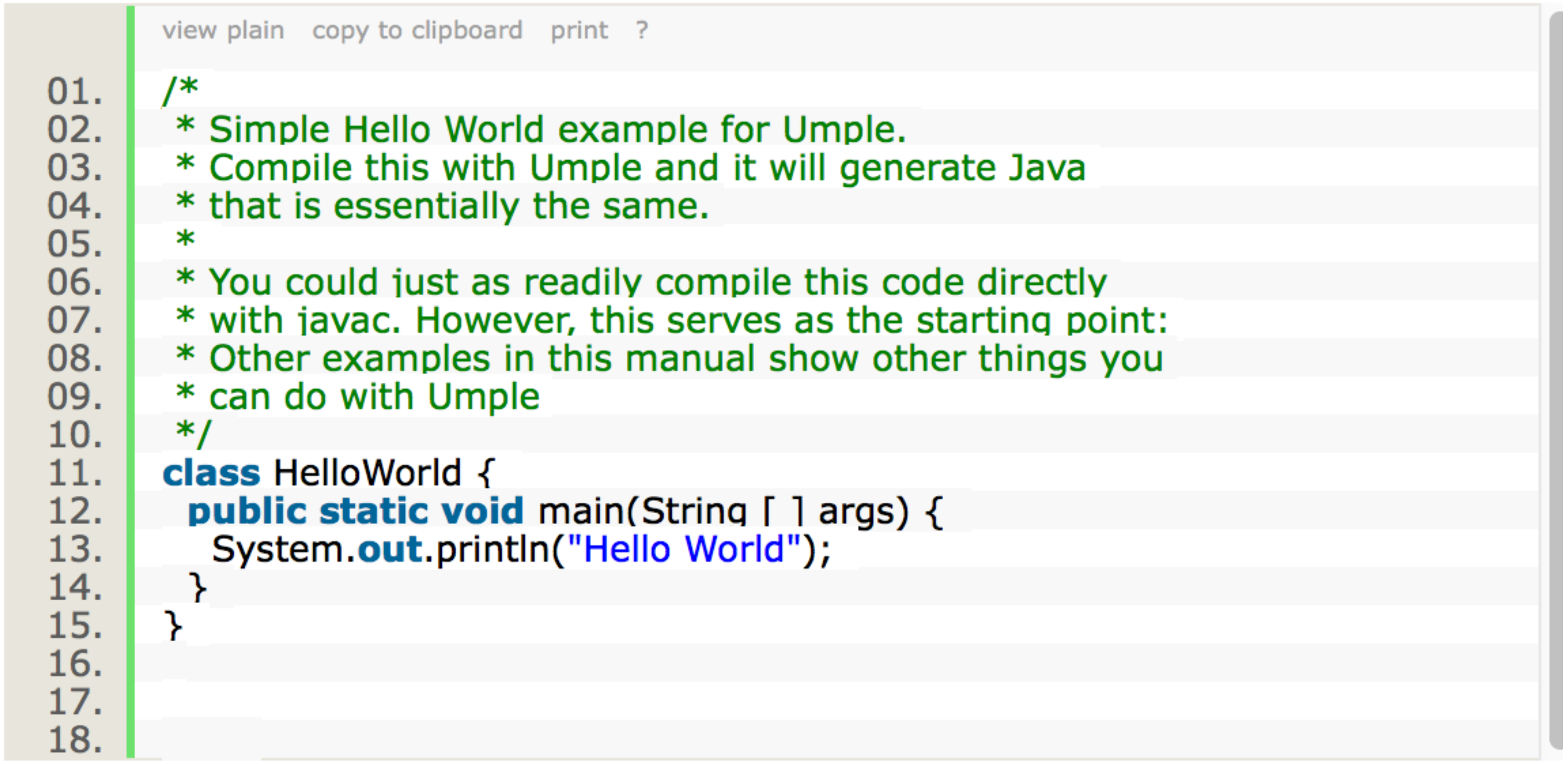
- https://www.guru99.com/best-uml-tools.html
- https://modeling-languages.com/uml-tools/

The User Manual and Hello World

Go to http://helloworld.umple.org

Look at the first example

- Observe: just a plain method



```
/*
 * Simple Hello World example for Umple.
 * Compile this with Umple and it will generate Java
 * that is essentially the same.
 *
 * You could just as readily compile this code directly
 * with javac. However, this serves as the starting point:
 * Other examples in this manual show other things you
 * can do with Umple
 */
class HelloWorld {
  public static void main(String [ ] args) {
    System.out.println("Hello World");
  }
}
```

Exercise: Compiling and Changing a Model

Look at the example at the bottom of
http://helloworld.umple.org (also on next slide)

- Observe: attribute, association, class hierarchy, mixin

Click on Load the above code into UmpleOnline

- Observe and modify the diagram
- Add an attribute
- Make a multiplicity error, then undo
- Generate code and take a look
- Download, compile and run (Follow as I show you how)

Demo of Compiling on the Command Line - 1

To compile on the command line you need Java 8 or higher

Download umple.jar from http://dl.umple.org

Suggestions in Mac/Linux

- put it in a directory tmp
- `alias umple='java -jar ~/tmp/umple.jar'`

Demo of Compiling on the Command Line - 2

Basic compilation

- `java -jar  umple.jar test.ump`
- `umple test.ump`

- `umple --help`

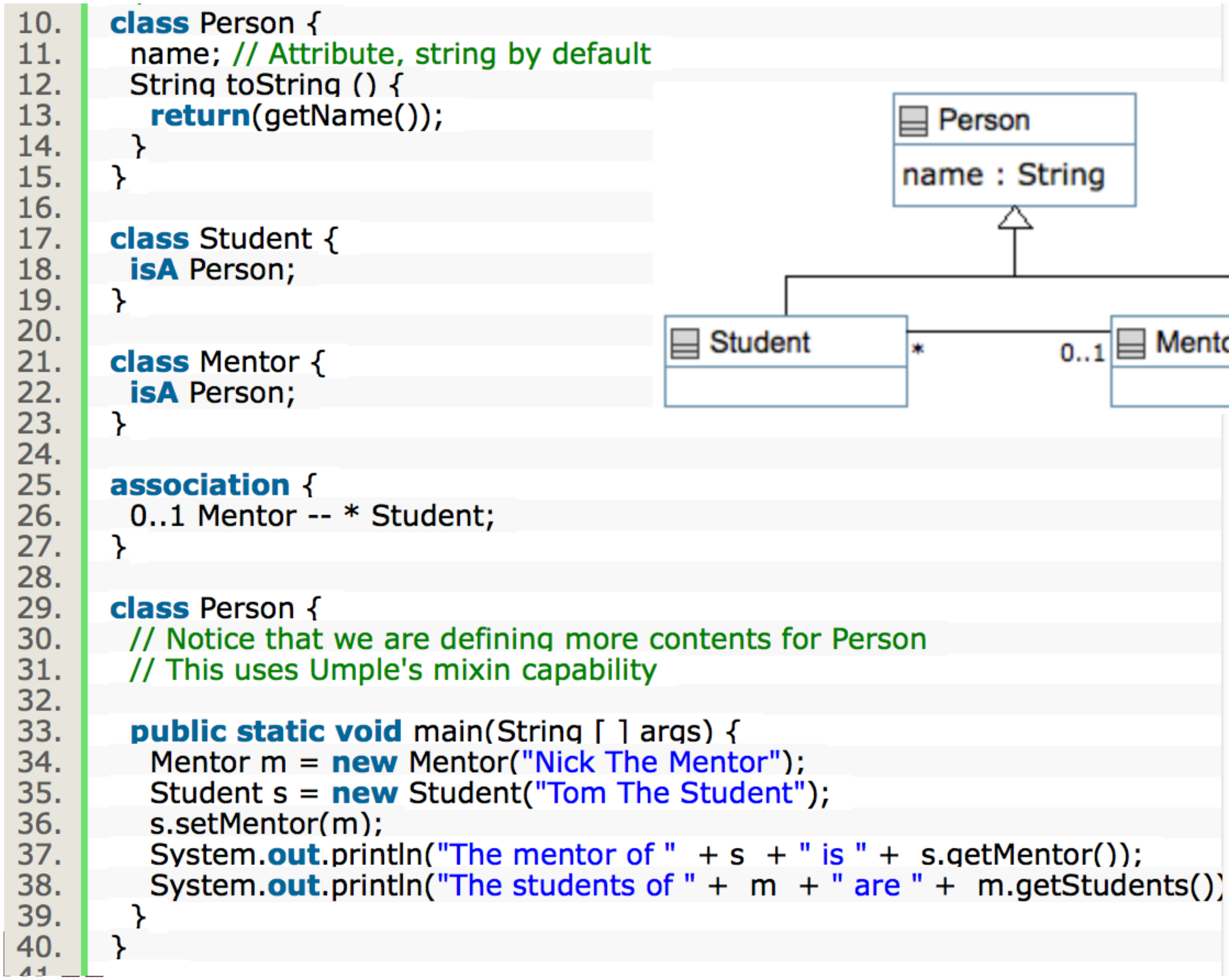
To generate and compile the java to a final system

- `umple model.ump -c -`

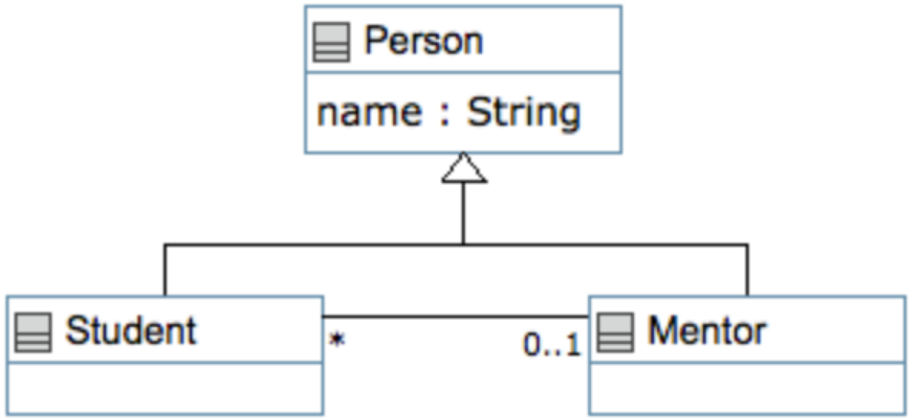
Use control-o and then paste into a terminal to copy any text from UmpleOnline and compile it to java on your machin

- Example on next slide

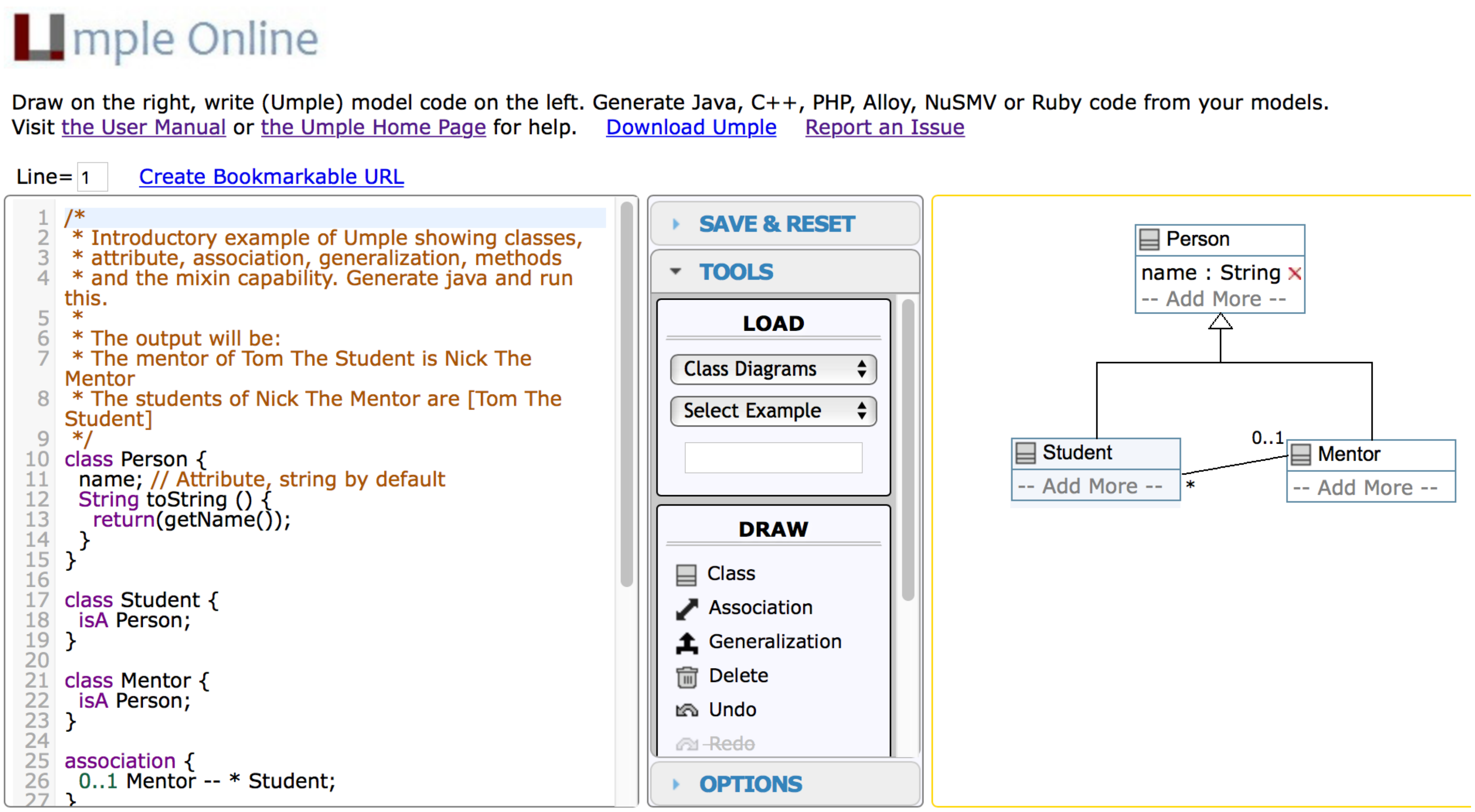
Hello World Example 2 in the User Manual



```
class Person {
  name; // Attribute, string by default
  String toString () {
    return(getName());
  }
}

class Student {
  isA Person;
}

class Mentor {
  isA Person;
}

association {
  0..1 Mentor -- * Student;
}

class Person {
  // Notice that we are defining more contents for Person
  // This uses Umple's mixin capability

  public static void main(String [ ] args) {
    Mentor m = new Mentor("Nick The Mentor");
    Student s = new Student("Tom The Student");
    s.setMentor(m);
    System.out.println("The mentor of "  + s  + " is " +  s.getMentor());
    System.out.println("The students of " +  m  + " are " +  m.getStudents()
  }
}
```



Hello World Example 2 in UmpleOnline



Umple Online

Draw on the right, write (Umple) model code on the left. Generate Java, C++, PHP, Alloy, NuSMV or Ruby code from your models.
Visit the User Manual or the Umple Home Page for help. Download Umple Report an Issue

Line= 1 Create Bookmarkable URL

```
/*
 * Introductory example of Umple showing classes,
 * attribute, association, generalization, methods
 * and the mixin capability. Generate java and run
this.
 *
 * The output will be:
 * The mentor of Tom The Student is Nick The
Mentor
 * The students of Nick The Mentor are [Tom The
Student]
 */
class Person {
  name; // Attribute, string by default
  String toString () {
    return(getName());
  }
}

class Student {
  isA Person;
}

class Mentor {
  isA Person;
}

association {
  0..1 Mentor -- * Student;
}
```

Exploration of UmpleOnline

Explore class diagram examples

Explore options

- View – hide and showing text, diagram
    - —Control-t (text), control-d (diagram) as shortcuts
- View – hide and showing methods, attributes
- Generate different default diagram types
    - —Control-g (Graphviz), control-s (state), control-e

Generate code and look at the results

- In Umple you *never should modify generated code*, but it is designed to be readable for education and certification

Walkthrough of parts of the User Manual

https://manual.umple.org

Note in particular

- Key sections: attributes, associations, state machines
- Grammar
- Generated API
- Errors and warnings
- Editing pages in Github

Attributes

As in UML, more abstract than instance variables

- Always <u>private</u> by default
- Should only be accessed get, set methods

- Can be *stereotyped* (upcoming slides) to affect code generation

- Can have *aspects* applied (discussed later)

- Can be *constrained* (discussed later)

Umple Builtin Datatypes

```
String // (default if none specified)
Integer
Float
Double
Boolean
Time
Date
```

The above will generate appropriate code in Java, C++ etc.

- e.g. Integer becomes int

Other (native) types can be used but without guaranteed correctness

Some Stereotypes Used on Attributes

By default, *each attribute adds an argument to the generated constructor*

To prevent this, use one of these

```
autounique x;  // sets attribute to 1, 2, 3 …

lazy b;  // sets it to null, 0, “” depending on type

a = "init value";

defaulted s = "def"; // resettable to the default

internal i; // doesn’t generate any get/set method
```

More more details: http://attributes.umple.org

Immutability

Useful for objects where you want to enforce no possible change of values after an object instance created

- e.g. a geometric point

Generate a constructor argument and get method but no set method

```
immutable String str;
```

The following ensures constructor argument, but allows setting just once.

```
lazy immutable z;
```

Generalization in Umple

Umple uses the isA keyword to indicate generalization

- Used to indicate superclass, used trait, implemented interface

```
class Shape {
  colour;
}
class Rectangle {
  isA Shape;
}
```

Interfaces

Declare signatures of a group of methods that must be implemented by various classes

Also declared using the keyword `isA`

Standard UML concept; the same concept as in Java

User-Written Methods in Umple

Methods can be added to any Umple code.

Umple *parses the signature only; the rest is passed to the generated code*.

You can specify <u>different bodies in different languages</u>

We will look at examples in the user manual …

Associations in Umple

```
class Employee {
  id;
  firstName;
  lastName;
}

class Company {
  name;
  1 -- * Employee;
}
```

Referential Integrity

When an instance on one side of the association changes

- The linked instances on the other side know …
- And vice-versa

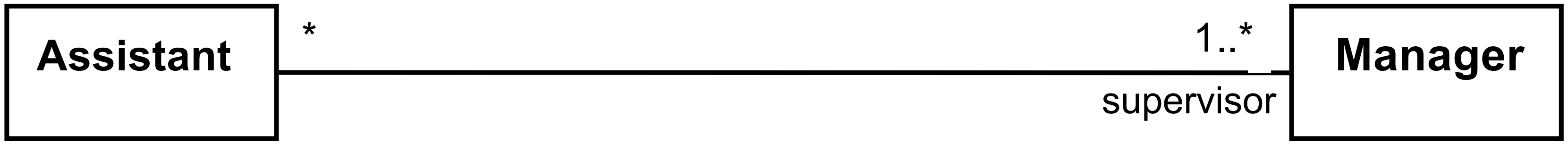
This is standard in Umple associations, which are bidirectional

Role Names (optional, in most cases)

Allow you to better label either end of an association

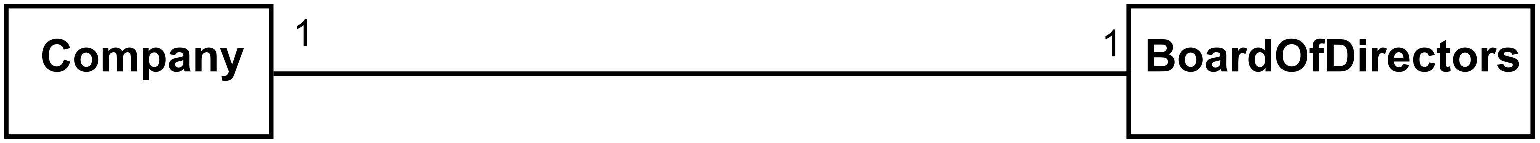
```
class Person{
  id;
  firstName;
  lastName;
}

class Company {
  name;
  1 employer -- * Person employee;
}
```

Many-to-Many Associations



Open in Umple

One-to-One Associations (Use Cautiously)



Open in Umple

Unidirectional Associations

Associations are by default *bi-directional*

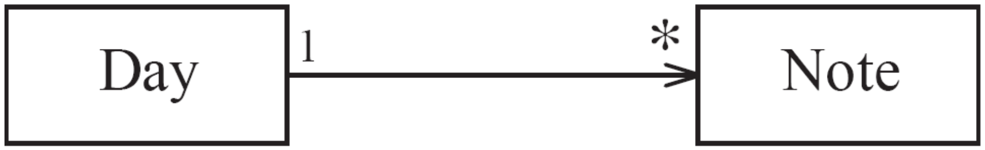
Limit the navigability direction of an association by adding an arrow at one end

In the following unidirectional association

— A Day knows about its notes, but a Note does not know which Day is belongs to

—Note remains ‘uncoupled’ and can be used in other contexts

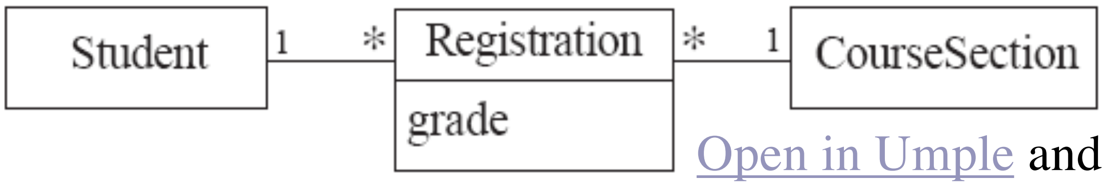
```
class Day {
  * -> 1 Note;
}
class Note {}
```



Open in Umple

Association Classes

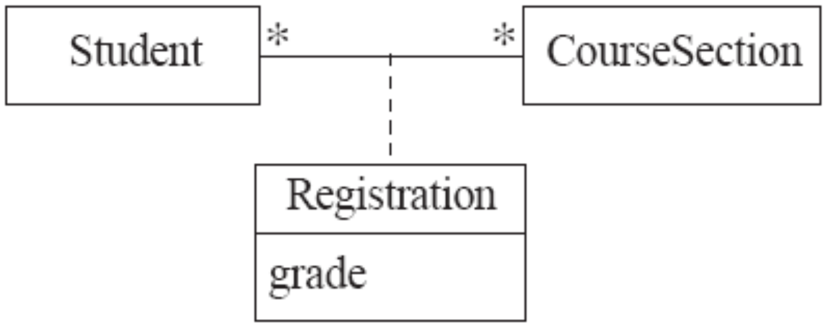
Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes



Open in Umple and extended example

The following are nearly equivalent

- The only difference:
  —in the association class there can be only a *single* registration of a given Student in a CourseSection



Association Classes (cont.)

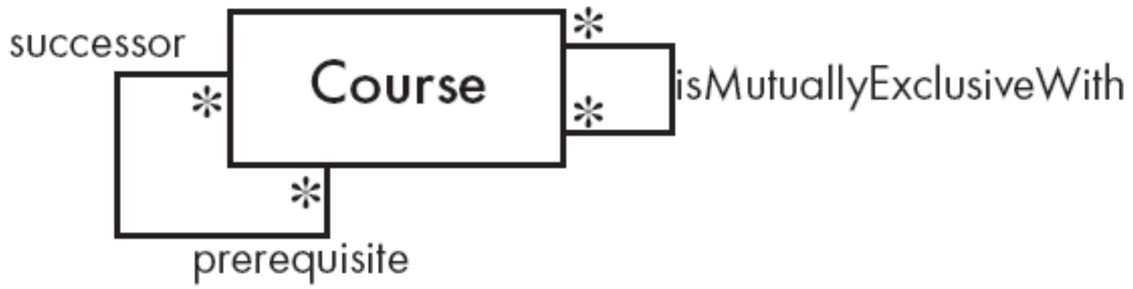
Umple code

```
class Student {}
class CourseSection {}
associationClass Registration {
  *  Student;
  *  CourseSection;
}
```

Open in UmpleOnline, and then generate code

Reflexive Associations

An association that connects a class to itself



```
class Course {
   * self isMutuallyExclusiveWith; // Symmetric
}

association {
     * Course successor -- * Course prerequisite;
}
```

Open in Umple

Singleton Pattern

Standard pattern to enable only a single instance of a class to be created.

- private constructor
- getInstance() method

Declaring in Umple

```
class University {
  singleton;
  name;
}
```

Delegation Pattern

A class calls a method in its ‘neighbour’

```
class RegularFlight {
  flightNumber;
}

Class SpecificFlight {
  * -- 1 RegularFlight;
  flightNumber = {getRegularFlight().getFullNumber()}
}
```

Full details of this example in the user manual

Basic Constraints

Shown in square brackets

- Code is added to the constructor and the set method

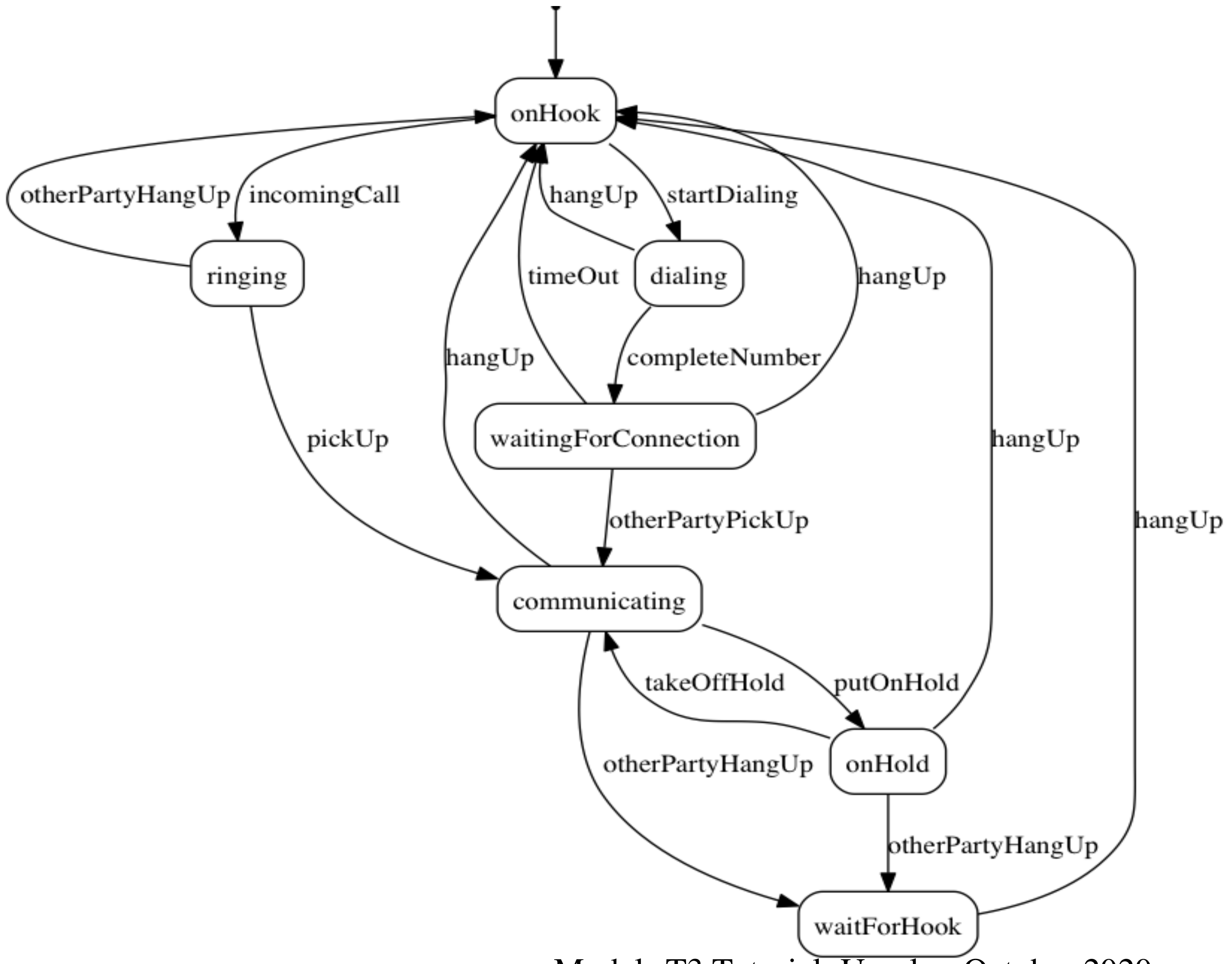
```
class X {
  Integer i;
  [! (i == 10)]
}
```

We will see constraints in state machines (as guards)

Basics of State Machines

- At any given point in time, the system is in one <u>state</u>.
- It will remain in this state until an <u>event</u> occurs that causes it to change state.
- Standard UML notation and semantics

State Machine with tracing: Phone and Lines example in UmpleOnline



Do Activities and Concurrency

A do activity executes

- In a separate thread
- Until
  - Its method terminates, or
  - The state needs to exit (killing the tread)

Example uses:

- Outputting a stream (e.g. playing music)
- Monitoring something
- Running a motor while in the state
- Achieving concurrency, using multiple do activities

Active Objects

These start in a separate thread as they are instantiated.

- Implemented as syntactic sugar for having a state machine with a single state with a do activity

Declared with the keyword

```
active
```

See the user manual for an example

State Tables and Simulations

Allow analysis of state machines statically without having to write code

We will explore these in UmpleOnline by looking at state machine examples and generating tables and simulations

Default Threading in State Machines

As discussed so far, basic code generated for state machines has the following behaviour:

- A <u>single</u> thread:
  - —Calls an event
  - —Executes the event (running any actions)
  - —Returns to the caller and continues

This has two problems:

1. If another thread calls the event at the same time they will '*interfere*'
2. There can be *deadlocks* if an action itself triggers an event

Queued State Machines

The ‘queued’ stereotype solves the threading problem:

- Callers can add events to a queue without blocking
- A separate thread takes items off the queue ‘as fast as it can’ and processes them

Umple syntax: **`queued`** before the state machine declaration

We will look at an examples in the manual

Pooled State Machines

Default Umple Behavior (including with queued):

- If an event is received but the system is not in a state that can handle it, then the event is ignored.

Alternative `pooled` stereotype:

- Uses a queue (see previous slide)
- Events that cannot be processed in the current state are *left at the head of the queue* until a relevant state reached
- The first relevant event nearest the head of the queue is processed
- Events may hence be processed out of order, but not ignored

Unspecified Pseudo-Event

Matches any event that is not listed

Can be in any state, e.g.

- `unspecified -> error;`

Example using unspecified

```
class AutomatedTellerMachine{
  queued sm  {
    idle {
      cardInserted -> active;      maintain -> maintenance;
      unspecified -> error1;
    }
    maintenance { isMaintained -> idle; }
    active {
      entry /{addLog("Card is read");}
      exit /{addLog("Card is ejected");}
      validating {
        validated -> selecting;
        unspecified -> error2;
      }
      selecting {select -> processing; }
      processing {
        selectAnotherTransiction -> selecting;
        finish -> printing;
      }
      printing {receiptPrinted -> idle;}
      cancel -> idle;
    }
    error1 {entry / {printError1();} ->idle;}
    error2 {entry / {printError2();} ->validating;}
  }
}
```

Model-Based Template Generation of Text

Allow output of complex text in any class

- Can generate XML, html, code, UI, etc.

Template for exactly the content

- `textToOutout <<!output this!>>`

Expression

- `<<=someCode();>>`

Internal logic within a template

- `<<# if(a==0){#>> … <<#}#>>`

Template Generation - Continued

We will look at examples in the in the User Manual

- Simple multiplication table
- Form letter

Mixins: Motivation

Product variants have long been important for

- —Product lines/families, whose members target different:
  - hardware, OS, feature sets, basic/pro versions

- —Feature-oriented development (separation of concerns

Separation of Concerns by Mixins in Umple

Mixins allow incremental addition to a class of attributes, associations, state machines, and any other feature

Example:

```
class X { a; }
class X { b; }
```

- The result would be a class with both a and b.

It doesn't matter whether the mixins are

- Both in the same file
- One in *one file*, that includes the other in another file
- In two separate files, with a third file invoking them

Typical Ways of Using Mixins

Separate groups of classes for

- model (classes, attributes, associations)
- Methods operating on the model

Allows a clearer view of the core model

Another possibility

- One feature per file

Advantages and Disadvantages of Mixins

Advantages:

- Smaller files that are easier to understand
- Possibility to define *variants* and *product lines*:
  - —Different versions of a class for different software versions (e.g. a professional version)

Disadvantage

- *Delocalization*:
  - —Bits of functionality of a class in different files
  - —The developer may not know that a mixin exists unless a tool helps show this

Aspects: Motivation

We often don’t quite like the code as generated

Or

We want to do a little more than what the generated code does

Or

We want to inject some feature (e.g. security checks) into many places of generated or custom code

Aspects: General concepts

Create a *pointcut* that specifies (advises) where to inject code at multiple points elsewhere in a system

- The pointcut uses a *pattern to match where to inject*
- Pieces of code that would otherwise be scattered are thus gathered into the aspect

But: There is potentially acute sensitivity to change

- If the code changes the aspect may need to change
- Yet without tool support, developers wouldn't know this

Drawback: Delocalization even stronger than for mixins

Aspect Orientation in Umple

It is common to limit a pointcuts a single class

- Inject code before, after, or around execution of custom or generated methods and constructors

```
class Person {
  name;
  before setName {
    if (aName != null && aName.length() > 20) { return false;
   }
  }
}
```

We have found these limited aspects nonetheless solve key problems

Traits: Motivation

We may want to inject similar elements into unrelated classes without complex multiple inheritance

- Elements can be
  - Methods
  - Attributes
  - Associations
  - States or state machines
  - .. Anything

Separation of Concerns by Traits

Allow modeling elements to be made available in multiple classes

```
trait Identifiable {
  firstName;
  lastName;
  address;
  phoneNumber;
  fullName = {firstName + " " + lastName}
  Boolean isLongName() {return lastName.length() > 1;}
}

class Person {
  isA Identifiable;
}
```

See more complete version of this in the user manual

Another Trait Example

```
trait T1{
   abstract void method1(); /* required method */
   abstract void method2();
   void method4(){/*implementation – provided method*/ }
}
trait T2{
    isA T1;
   void method3();
   void method1(){/*implementation*/ }
    void method2(){/*implementation*/ }
}
class C1{
  void method3(){/*implementation*/ }
}
class C2{ isA C1; isA T2;
  void method2(){/*implementation*/ }
}
```

Traits With Parameters

```
trait T1< TP isA I1 > {
   abstract TP method2(TP data);
   String method3(TP data){ /*implementation*/ }
}
interface I1{
  void method1();
}
class C1{ isA I1;
   isA T1<TP = C1>;
   void method1(){/*implementation*/}
   C1 method2(C1 data){ /*implementation*/ }
}
class C2{
   isA I1;
   isA T1< TP = C2 >;
   void method1(){/*implementation*/}
   C2 method2(C2 data){ /*implementation*/ }
}
```

Trait Parameters in Methods

```
trait T1 <TP>{
  String method1();
  String method2(){
    #TP# instance = new #TP#();
    return method1() +":"+instance.process();
  }
}
class C1{
  String process(){/*implementation*/}
}
class C2{
  isA T1< TP = C1 >;
  String method1(){/*implementation*/ }
}
```

Selecting Subsets of Items in Traits

```
trait T1{
  abstract method1();
  void method2(){/*implementation*/}
  void method3(){/*implementation*/}
  void method4(){/*implementation*/}
  void method5(){/*implementation*/}
}
class C1{
  isA T1<-method2() , -method3()>;
  void method1() {/*implementation related to C1*/}
}
class C2{
  isA T1<+method5()>;
  void method1() {
  /*implementation related to C2*/}
}
```

Renaming Elements when Using Traits

```
trait T1{
  abstract method1();
  void method2(){/*implementation*/}
  void method3(){/*implementation*/}
  void method4(){/*implementation*/}
  void method5(Integer data){/* implementation*/}
}
class C1{
  isA T1< method2() as function2 >;
  void method1() {/*implementation related to C1*/}
}
class C2{
  isA T1< method3() as private function3 >;
  void method1() {/*implementation related to C2*/}
}
class C3{
  isA T1< +method5(Integer) as function5 >;
  void method1() {/*implementation related to C3*/}
}
```

Associations in Traits: Observer Pattern

```
class Dashboard{
  void update (Sensor sensor){ /*implementation*/ }
}
class Sensor{
  isA Subject< Observer = Dashboard >;
}
trait Subject <Observer>{
  0..1 -> * Observer;
  void notifyObservers() { /*implementation*/ }
}
```

Using Traits to Reuse State Machines

```
trait T1 {
  sm1{
    s0 {e1-> s1;}
    s1 {e0-> s0;}
  }
}
trait T2 {
  isA T1;
  sm2{
    s0 {e1-> s1;}
    s1 {e0-> s0;}
  }
}
class C1 {
  isA T2;
}
```

Satisfaction of Required Methods Through State Machines

```
trait T1{
  Boolean m1(String input);
  Boolean m2();
  sm1{
    s1{
      e1(String data) -> /{ m1(data); } s2; }
    s2{
      e2 -> /{ m2(); } s1; }
  }
}
class C1{
  isA T1;
  sm2{
    s1{ m1(String str) -> s2;}
    s2{ m2 -> s1;}
  }
}
```

Changing Name of a State Machine Region

```
trait T1{
  sm {
    s1{
      r1{ e1-> r11; }
      r11{}
        ||
      r2{ e2-> r21; }
      r21{}
    }
  }
}
class C1{
  isA T1<sm.s1.r1 as region1,sm.s1.r2 as region2>;
}
```

Changing the Name of an Event

```
trait T1 {
  sm1{
    s0 { e1(Integer index)-> s1;}
    s1 {e0-> s0;}
  }
  sm2{
    t0 {e1(Integer index)-> t1;}
    t1 {e0-> t0;}
  }
}
class C1 {
  isA T1<sm1.e1(Integer) as event1, *.e0() as event0>;
}
```

Mixins and Traits together

- Examples of mixins and traits combined in the user manual:
  - —Mixins with traits:
    - https://cruise.umple.org/umple/TraitsandUmpleMixins.html

Mixsets: Motivations

A feature or variant needs to inject or alter code in many places

- Historically tools like the C Preprocessor were used
- Now tools like “Pure: Variants”

There is also a need to

- Enable *model* variants in a very straightforward way
- Blend variants with code/models in core compilers
  - —With harmonious syntax + analysable semantics
  - —Without the need for tools external to the compiler

Mixsets: Top-Level Syntax

Mixsets are named sets of mixins

```
mixset Name {
  // Anything valid in Umple at top level
}
```

The following syntactic sugar works for top level elements (class, trait, interface, association, etc.)

```
mixset Name class Classname {
}
```

Use Statements

A use statement specifies inclusion of either

- A file, or
- A mixset

```
use Name;
```

A mixset is conceptually a *virtual file* that is composed of a set of model/code elements

The use statement for a mixset can appear

- *Before*, *after* or *among* the definition of the mixset parts
- In *another mixset*
- On the *command line* to generate a variant

Mixsets and Mixins: Synergies

- The blocks defined by a mixset are mixins
- Mixsets themselves can be composed using mixins
  - —e.g.

```
mixset Name1  {class X { a; } }
```

And somewhere else

```
mixset Name1  {class X { b; } }
use Name1;
```

Would be the same as:

```
class X { a; b;}
```

Mixset Definitions *Internal* to a Top-Level Element

```
class X {
  mixset Name2 {a;}
  b;
}
```

Is the same as,

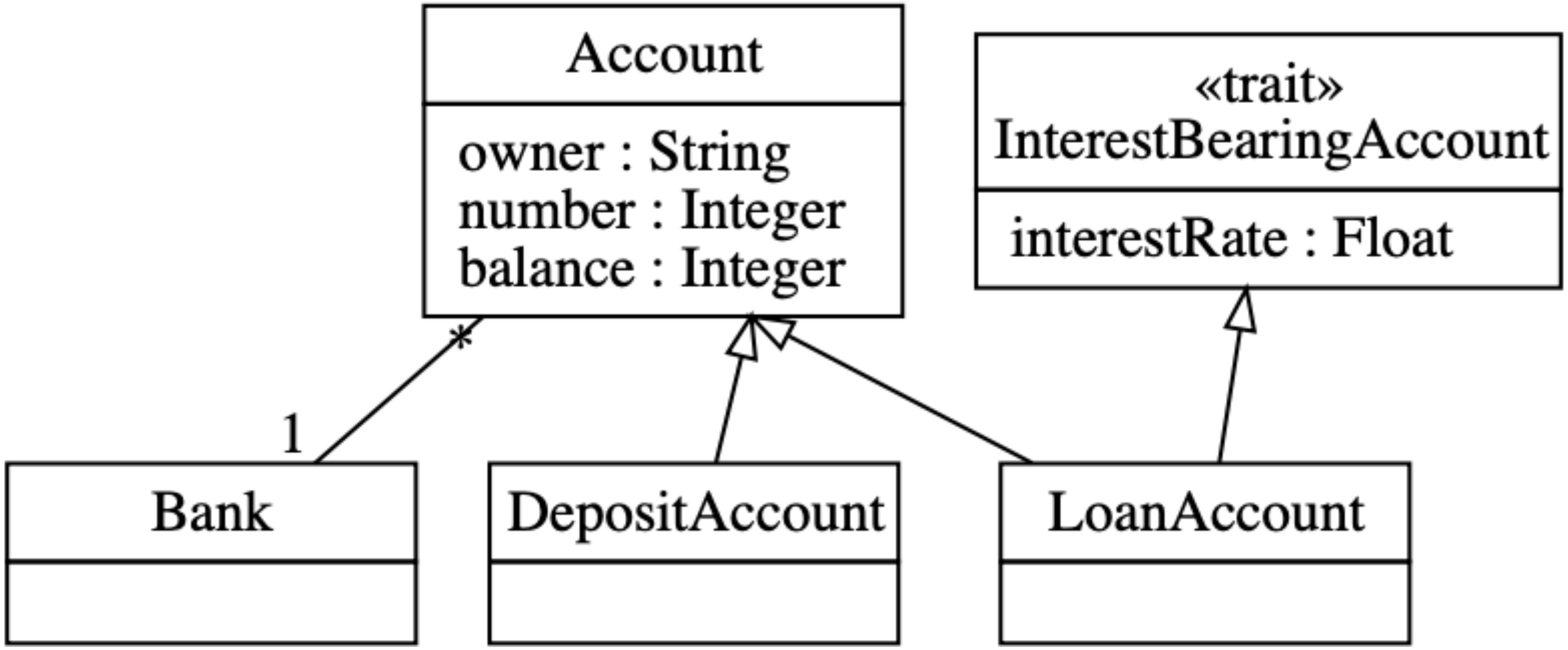
```
mixset Name2 class X {a;}
  class X {b;}
```

The above works for attributes, associations, state machines, states, etc.

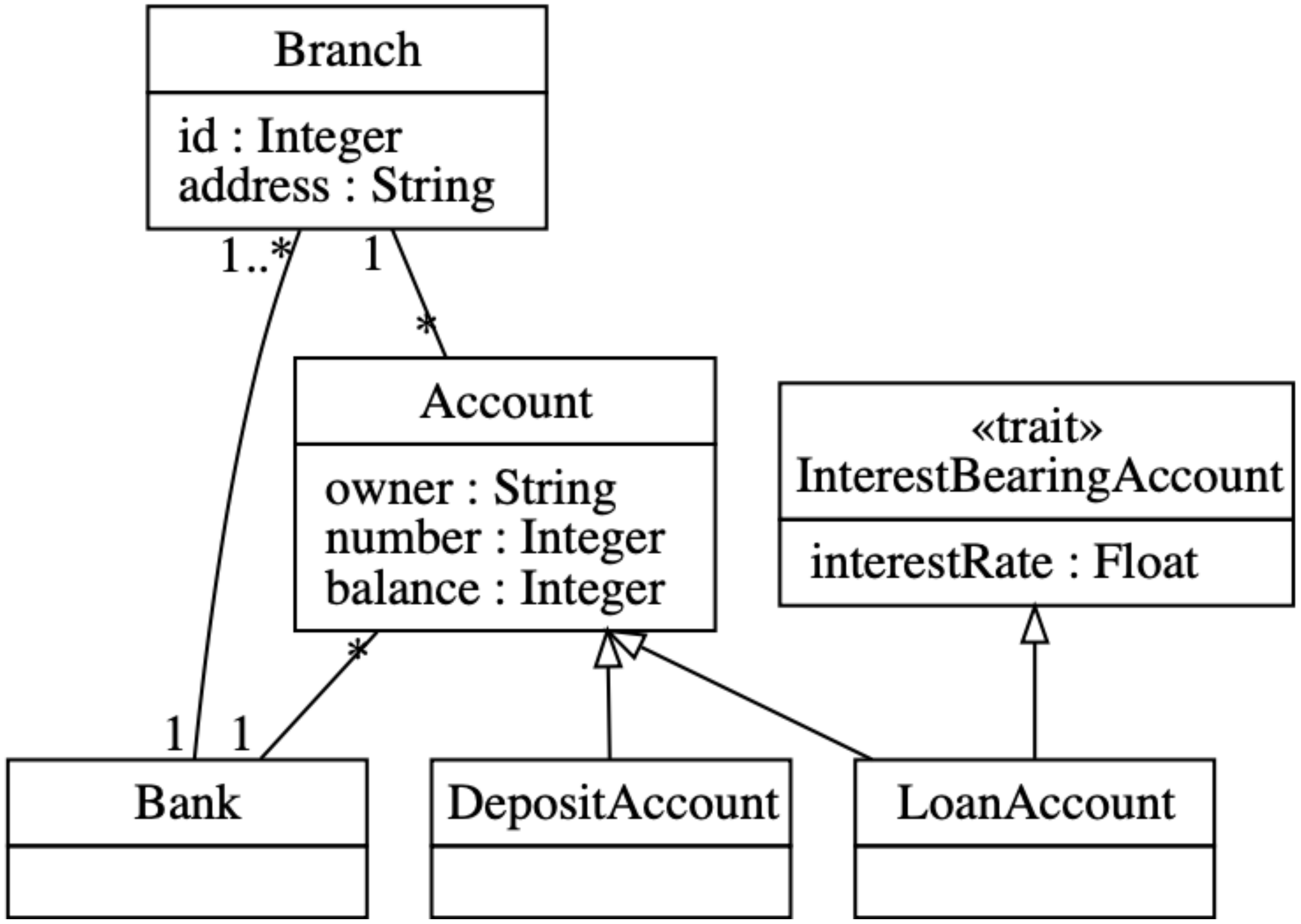
Motivating Example: Umple Model/Code for Basic Bank

```
class Bank {
  1 -- * Account;
}

class Account {
  owner; Integer number; Integer balance;
}

trait InterestBearingAccount {
  Float interestRate;
}

class DepositAccount {
  isA Account;
}

class LoanAccount {
  isA Account, InterestBearingAccount;
}
```

Class Diagram of Basic Bank Example: Generated from Umple



Adding Optional Multi-branch Feature



Example: Multi-branch Umple Model/Code

```
class Bank {
  1 -- * Account;
  mixset Multibranch 1 -- 1..* Branch;
}

mixset Multibranch class Branch {
  Integer id; String address;
}

class Account {
  owner; Integer number; Integer balance;
  mixset Multibranch * -- 1 Branch;
}

trait InterestBearingAccount {
  Float interestRate;
}

class DepositAccount {
  isA Account;
  mixset OverdraftsAllowed {
    Integer overdraftLimit;
    isA InterestBearingAccount;
  }
}

class LoanAccount {
  isA Account, InterestBearingAccount;
}
```

Alternative Approach (same system)

```
class Bank {
  1 -- * Account;
}

class Account {
  owner; Integer number; Integer balance;
}

trait InterestBearingAccount {
  Float interestRate;
}

class DepositAccount {
  isA Account;
  mixset OverdraftsAllowed {
    Integer overdraftLimit;
    isA InterestBearingAccount;
  }
}

class LoanAccount {
  isA Account, InterestBearingAccount;
}

mixset Multibranch {
  class Bank {1 -- 1..* Branch}
  class Branch {Integer id; String address;}
  class Account {* -- 1 Branch}
}
```

Constraints on Mixsets

```
require [Mixset1 or Mixset2];
```

Allowed operators

- `and, or, xor`
- `not`
- `n..m of {…}`

Parentheses allowed

opt X (means 0..1 of {X})

Case Study and Exercise 1: Modifying the banking example

I will give you the text of the banking example and set up a task for you to:

- Add the ability to have one or more account holders
- Add the ability to have one or more co-signers

Case Study and Exercise 2: Dishwasher example

We will start with the Dishwasher example in UmpleOnline

We will use UmpleOnline's Task capability to ask you to split the Dishwasher example into **two versions**

- A cheap version that only does normal wash and not fast wash
- A full version that does everything

Hint: Pull out the relevant state and transition for fast wash and **wrap it in a mixset**

Case Study 3: Umple itself, written in Umple

We will look at:

- Code in Github
- Generated Architecture diagrams
- Generated Javadoc
- Sample master code
- Sample test output
- Sample code for generators (that replaced Jet)
- UmpleParser (that replaced Antlr

Wrapup:
Umple Philosophy 1-4

P1. Modeling is programming and vice versa

P2. An Umple programmer should never need to edit generated code to accomplish any task.

P3. The Umple compiler can accept and generate code that uses nothing but UML abstractions.

- The above is the inverse of the following

P4. A program without Umple features can be compiled by an Umple compiler.

- e.g. input Java results in the same as output

Wrapup:
Umple Philosophy 5-8

P5. A programmer can incrementally add Umple features to an existing program

- Umplification

P6. Umple extends the base language in a minimally invasive and safe way.

P7. Umple features can be created and viewed diagrammatically or textually

P8. Umple goes beyond UML