Introduction to **Umple**

CSI5112– February 2018

Based on presentations from Lethbridge, Garzón and umple.org.

Outline

1. Introduction:
2. Overview of Model-Driven Development
   — Languages / Tools / Motivation for Umple
3. Class Modeling
   — Tools / Attributes / Methods / Associations / Exercises / Patterns
4. Modeling with State Machines
   — Basics / Concurrency / Case study and exercises
5. Separation of Concerns in Models
   — Mixins / Aspects / Traits
6. More Case Studies and Hands-on Exercises
   — Umple in itself / Real-Time / Data Oriented
7. Conclusion

Umple: Simple, Ample, UML Programming Language

1. Open source textual modelling tool set for 3 platforms
   - Command line compiler
   - Web-based tool (UmpleOnline) for demos and education
   - Eclipse plugin

2. Code generator for UML ++
   - Infinitely nested state machines, with concurrency
   - Proper referential integrity and multiplicity constraints on associations
   - Traits, mixins, aspects for modularity
   - Text generation templates, patterns, traits

3. Pre-processor to add UML, patterns and other features on top of Java, PhP, C++ and other languages

Websites

Entry-point: http://umple.org

UmpleOnline: http://try.umple.org

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

Wiki: http://code.google.com/p/umple/wiki/UmpleHome

Tutorials: http://code.google.com/p/umple/wiki/Tutorials

Publications:

https://code.google.com/p/umple/wiki/Publications

These slides are available

- http://www.site.uottawa.ca/~mgarz042/files/CSI5112-Umple.pdf

Motivation for developing Umple (1)

We want the ***best combination of features***:

- Textual editing and blending with other languages
- Ability to use in an agile process
  - Write tests, continuous integration, versioning
  - Combine the best of agility and modeling
- Excellent code generation
  - Complete generation of real systems (including itself)
- Multi-platform (command line, Eclipse, Web)
- Practical and easy to use for developers
  - Including great documentation
- Open source

Motivation for developing Umple (2)

Many existing tools:

- Lacked in usability
  - —Awkward to edit diagrams
  - —Many steps to do a task
  - —Lengthy learning process
- Lack in ongoing support
- Could be enhanced by us perhaps, but we would be tied to key decisions (e.g. Eclipse-only)

Some key Umple innovations

Model is code

- Traditional code is embedded in model

No need to edit generated code

- No ‘round-trip engineering’

Using Umple

We will mostly be using

- Umpleonline
  - In a web browser: http://try.umple.org
  - Or in Docker: http://docker.umple.org
- Umple on the command line: http://dl.umple.org
  - Needs Java 8 JDK on the command line: http://bit.ly/1IO1FSV
    - Java 9 works well too

Optional:

- Umple in Eclipse https://github.com/umple/umple/wiki/InstallEclipsePlugin
- cmake and gcc for compiling C++ code

Outline

1. Introduction:
2. Overview of Model-Driven Development
   — Languages / Tools / Motivation for Umple
3. Class Modeling
   — Tools / Attributes / Methods / Associations / Exercises / Patterns
4. Modeling with State Machines
   — Basics / Concurrency / Case study and exercises
5. Separation of Concerns in Models
   — Mixins / Aspects / Traits
6. More Case Studies and Hands-on Exercises
   — Umple in itself / Real-Time / Data Oriented
7. Conclusion

Umple class models – quick overview

Key elements:

- Classes
- Attributes
- Associations
- Generalizations
- Methods

We will look at all these using examples

Umple code/models are stored in files with suffix .ump

Exercise: Compiling and changing a model

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

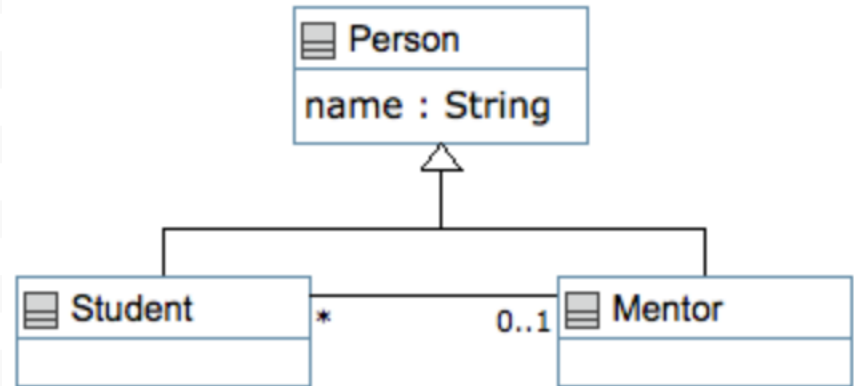
- Observe: attribute, association, class hierarchy, mixin

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 if you want

Hello World Example 2 in the User Manual

```
10. class Person {
11.   name; // Attribute, string by default
12.   String toString () {
13.     return(getName());
14.   }
15. }
16.
17. class Student {
18.   isA Person;
19. }
20.
21. class Mentor {
22.   isA Person;
23. }
24.
25. association {
26.   0..1 Mentor -- * Student;
27. }
28.
29. class Person {
30.   // Notice that we are defining more contents for Person
31.   // This uses Umple's mixin capability
32.
33.   public static void main(String [ ] args) {
34.     Mentor m = new Mentor("Nick The Mentor");
35.     Student s = new Student("Tom The Student");
36.     s.setMentor(m);
37.     System.out.println("The mentor of "  + s  + " is " +  s.getMentor());
38.     System.out.println("The students of " +  m  + " are " +  m.getStudents()
39.   }
40. }
```



Key tools:
UmpleOnline, command line, user manual

Hello World example 2 in UmpleOnline

Draw on the right, write (Umple) model code on the left, analyse and generate code from models.

Run in Docker for speed, or download For help: User manual Ask questions Report issue

Line= 1 E G S T D A M Generate Java Create Bookmarkable URL

```
1  /*
2   * Introductory example of Umple showing classes,
3   * attribute, association, generalization, methods
4   * and the mixin capability. Generate java and run
   this.
5   *
6   * The output will be:
7   * The mentor of Tom The Student is Nick The
   Mentor
8   * The students of Nick The Mentor are [Tom The
   Student]
9   */
10 class Person {
11   name; // Attribute, string by default
12   String toString () {
13     return(getName());
14   }
15 }
16
17 class Student {
18   isA Person;
19 }
20
21 class Mentor {
22   isA Person;
23 }
24
25 association {
26   0..1 Mentor -- * Student;
27 }
```

SAVE & LOAD

TOOLS

EXAMPLES

Class Diagrams

Select Example

DRAW

Class

Association

Generalization

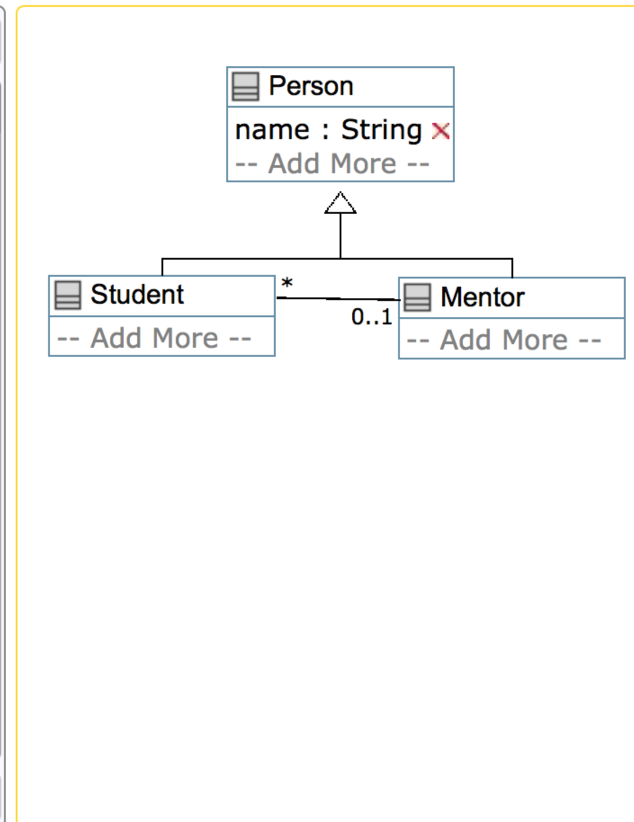
Delete

Undo

~~Redo~~

~~Sync Diagram~~

OPTIONS



Exploration of UmpleOnline

Explore class diagram examples

Options

- T or Control-t (hide and show text)
- D or Control-d (hide and show diagram)
- A, M to hide and show attributes, methods
- Default diagram types
  - —G/Control-g (Graphviz), S/Control-s (State Diagram)
  - —E/Control-e (Editable class diagram)

Generate code and look at the results

- In Umple you *never should modify generated code*
- It is designed to be readable for educational purposes

Use of the UmpleOnline Docker image

Umple's server can handle 80,000 transactions per hour

- Code generations, edits

But needs a good Internet connection

… and sometimes hundreds of students have assignments due!

To maximize speed of UmpleOnline run it in your local machine:

- Follow the instructions at http://docker.umple.org

Demo of compiling on the command line

To compile on the command line you will need Java 8

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

Basic compilation

- `java -jar  umple.jar model.ump`

- `java -jar  umple.jar --help`

To generate and compile the java to a final system

- `java -jar umple.jar model.ump -c -`

Quick walkthrough of the user manual

http://manual.umple.org

Note in particular

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

Attributes

Attributes

"Instance variables"

- Part of the state of an object
- Simple data that will always be present in each instance

Specified like a Java or C++ field or member variable

But, intended to be more abstract!

Example, with an initial value

```
a = "init value";
```

Code generation from attributes

Default code generation

- Generates a getName() and setName() method for name
  - public

- Creates an arguments in the class constructor by default

- An attribute is private to the class by default
  - Should only be accessed get, set methods

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

Attribute stereotypes (1)

Code generation can be controlled through *stereotypes*:

- lazy - don’t add a constructor argument

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

- Defaulted – can be reset

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

Attribute stereotypes (2)

- autounique – provide a unique value to each instance

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

- internal – don’t generate any methods

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

Immutability

Useful for objects where you want to guarantee no possible change once created

- e.g. a geometric point

Generate a constructor argument and get method but no set method

```
immutable String str;
```

No constructor argument, but allows setting just once.

```
lazy immutable z;
```

Lets explore attributes by example

Go to

http://attributes.umple.org

Derived attributes

These generate a get method that is calculated.

```
class Point
{
    // Cartesian coordinates
    Float x;
    Float y;

    // Polar coordinates
    Float rho =
      {Math.sqrt(Math.pow(getX(), 2) + Math.pow(getY(), 2))}
    Float theta =
      {Math.toDegrees(Math.atan2(getY(),getX()))}
}
```

Multi-valued attributes

Limit their use. Associations are generally better.

```
class Office {
   Integer number;
   Phone[] installedTelephones;
}

class Phone {
  String digits;
  String callerID;
}
```

Keys

Enable Umple to generate an equals() and a hashcode() method

```
class Student {
  Integer id;
  name;
  key { id }
}
```

The user manual has a sports team example showing keys on associations too

Note how this feature is not inherited from UML

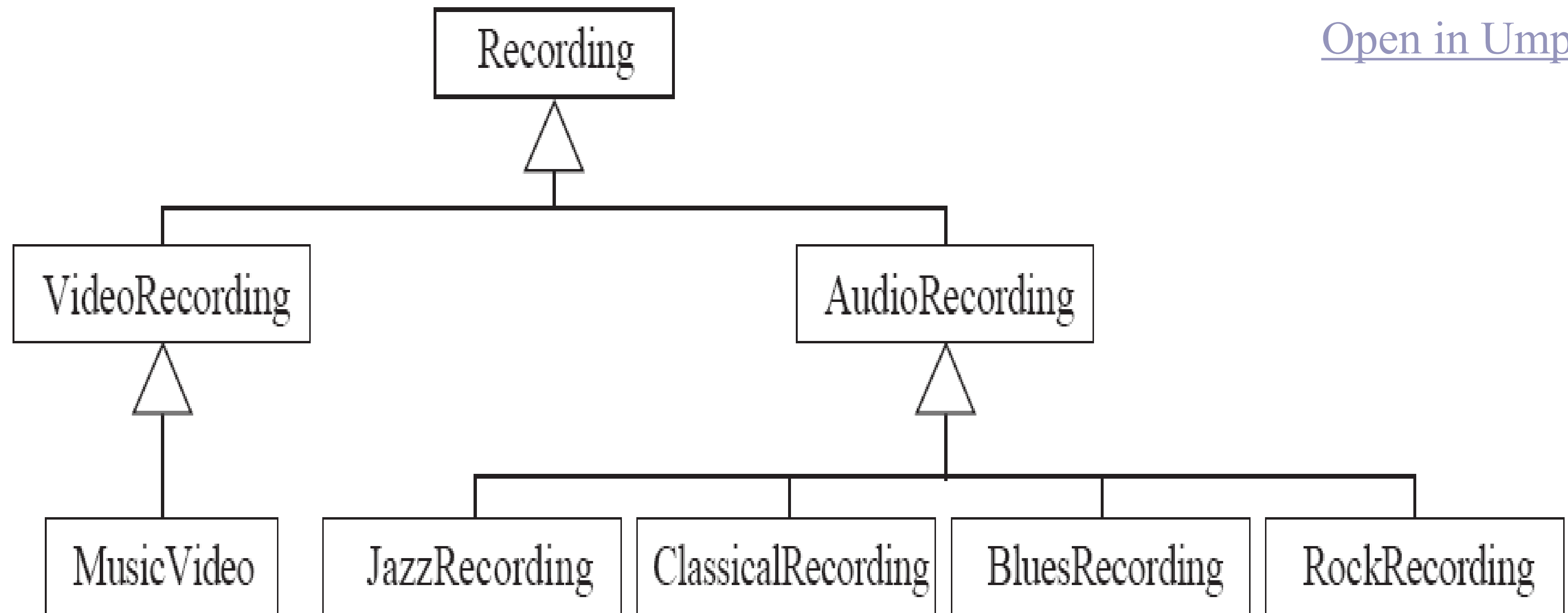
Generalization and interfaces

Generalization in Umple

Umple uses the isA keyword to indicate generalization

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

Avoiding unnecessary generalizations



Open in Umple

Inappropriate hierarchy of Classes

What should the model be?

Interfaces

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

Also declared using the keyword `isA`

Essentially the same concept as in Java

Let's explore examples in the user manual …

Methods

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 different bodies in different languages

We will look at examples in the user manual …

Associations

Associations

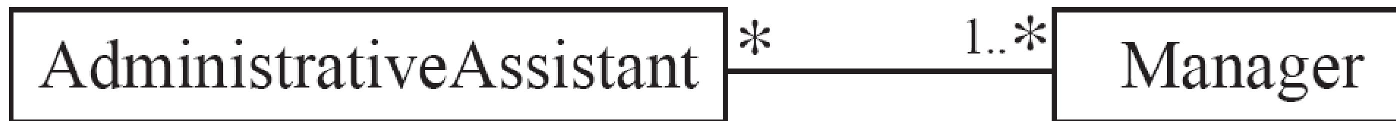
Describe how instances of classes are linked at runtime

- Bidirectional -- or unidirectional ->

Multiplicity: Bounds on the number of linked instances

* Or 0..*	0 or more
1..*	1 or more
1	Exactly 1
2	Exactly 2
1..3	Between 1 and 3
0..2	Up to 2

Basic UML associations



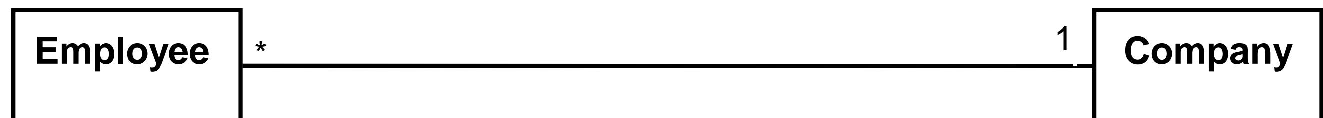
Many-to-one associations (1)

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

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

Many-to-one associations (2)

- A company has many employees,
- An employee can only work for one company.
  - This company will not store data about the moonlighting activities of employees!
- A company can have zero employees
  - E.g. a 'shell' company
- It is not possible to be an employee unless you work for a company
- Let's draw and write this in UmpleOnline:



Role names (optional, in most cases)

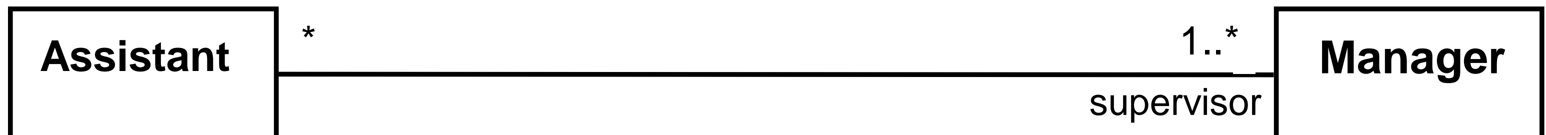
Allow you to better label either end of an association

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

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

Many-to-many associations

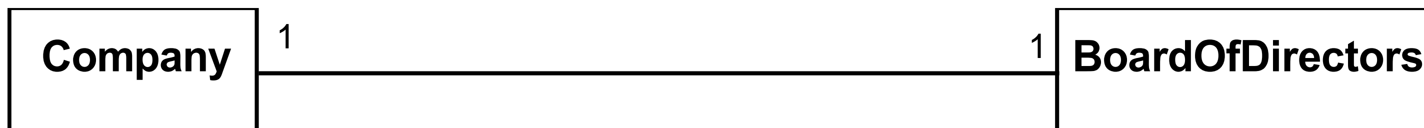
- An assistant can work for many managers
- A manager can have many assistants
- Assistants can work in pools working for several managers
- Managers can have a group of assistants
- Some managers might have zero assistants.
- Is it possible for an assistant to have, perhaps temporarily, zero managers?



Open in Umple

One-to-one associations (Use cautiously)

- For each company, there is exactly one board of directors
- A board is the board of only one company
- A company must always have a board
- A board must always be of some company

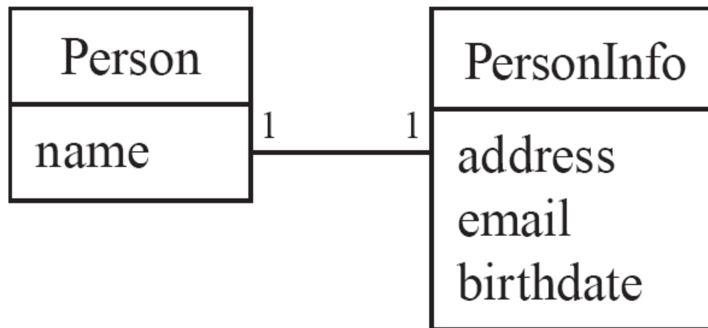


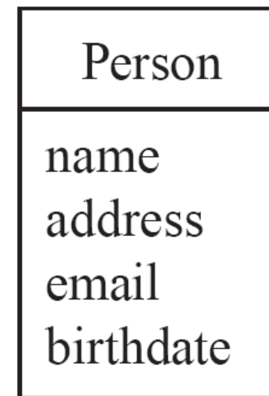
Open in Umple

Typical erroneous use of one-to-one

Avoid this

do this





Unidirectional associations

Associations are by default *bi-directional*

It is possible to limit the 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

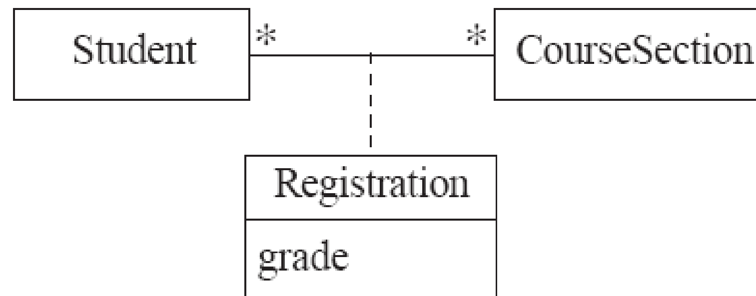
Association classes

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.)

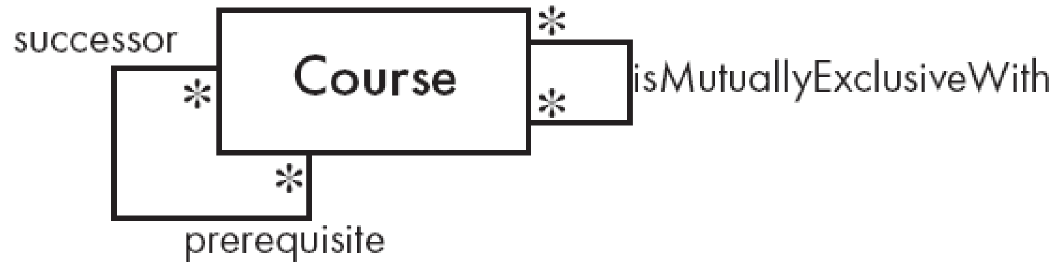
Umple code

```
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]

Inline vs. standalone associations

The following are equivalent to allow flexibility:

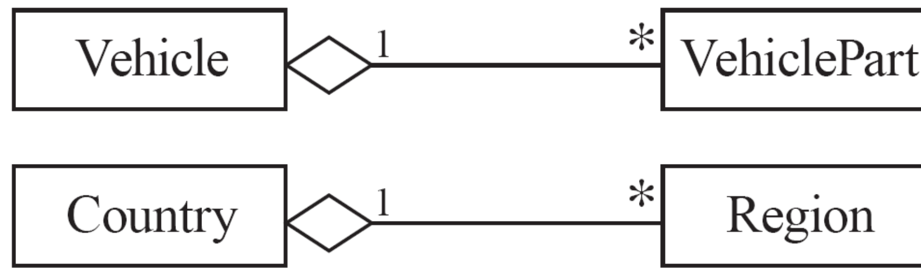
```
class X {}
class Y {
  1 -- * X;
}
```

```
class X {}
class Y {}
association {
  1 Y -- * X;
}
```

Aggregation

Aggregations are ordinary associations that represent part-whole relationships.

- The ‘whole’ side is often called the *assembly* or the *aggregate*
- This is a shorthand for association named `isPartOf`
- Umple has no special syntax currently




```
class Vehicle {
  1 whole -- * VehiclePart part;
}
class VehiclePart{
}
```

Composition

A *composition* is a strong kind of aggregation

- If the aggregate is destroyed, then the parts are destroyed as well

```
class Building {
  1 <@>- * Room;
}
class Room{
}
```

Sorted Associations

Order objects in the association according to a specific key

```
class Academy {
  1 -- * Student registrants sorted {id};
}

class Student {
  Integer id;
  name;
}
```

We will look at a more complete example in the User Manual

A final word on associations

More help and examples are in the user manual online at
http://associations.umple.org

Modeling exercises

Modeling Exercise

Build a class diagram for the following description. If you think there are key requirements missing, then add them.

1. A football (soccer) team has players. Each player plays a position. The team plays some games against other teams during each season. The system needs to record who scored goals, and the score of each game.

Simple patterns (if time)

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 later in state machines

Outline

1. Introduction:
2. Overview of Model-Driven Development
   — Languages / Tools / Motivation for Umple
3. Class Modeling
   — Tools / Attributes / Methods / Associations / Exercises / Patterns
4. Modeling with State Machines
   — Basics / Concurrency / Case study and exercises
5. Separation of Concerns in Models
   — Mixins / Aspects / Traits
6. More Case Studies and Hands-on Exercises
   — Umple in itself / Real-Time / Data Oriented
7. Conclusion

Basic state machines

Basics of state machines

- At any given point in time, the system is in one state.
- It will remain in this state until an event occurs that causes it to change state.
- A state is represented by a rounded rectangle containing the name of the state.
- Special states:
  - —A black circle represents the *start state*
  - —A circle with a ring around it represents an *end state*

Garage door state machine

```
class GarageDoor{
  status {
    Open {
     buttonOrObstacle -> Closing;
    }
    Closing {
      buttonOrObstacle -> Opening;
      reachBottom -> Closed;
    }
    Closed {
      buttonOrObstacle -> Opening;
    }
    Opening {
      buttonOrObstacle -> HalfOpen;
      reachTop -> Open;
     }
     HalfOpen {
       buttonOrObstacle -> Opening;
     }
   }
}
```

Events

An occurrence that *may trigger a change of state*

- Modeled in Umple as generated methods that can be called

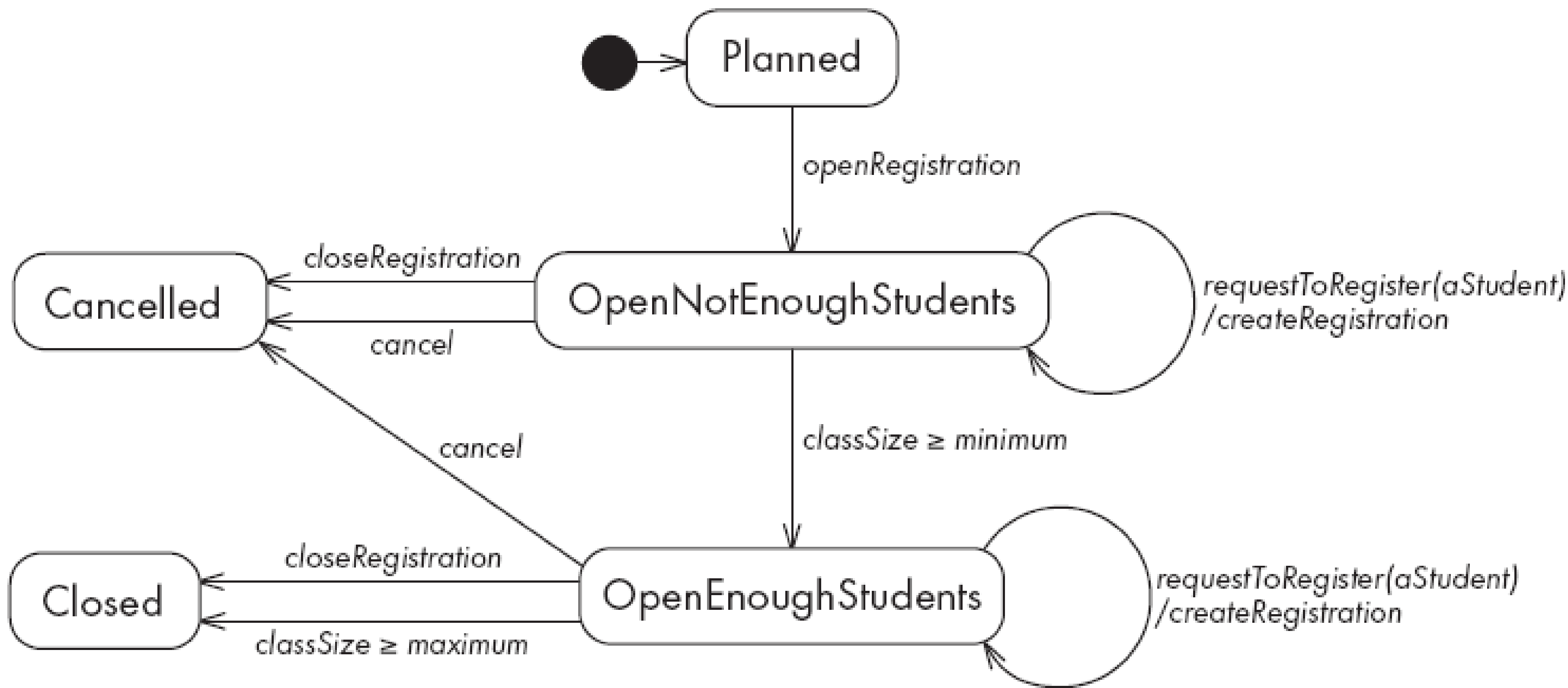
Several states may be able to respond to the same event

Transitions

- A change of state in response to an event.
  - It is considered to occur instantaneously.

- The label on each transition is the event that causes the change of state.

State diagrams – an example with conditional transitions



Actions in state diagrams

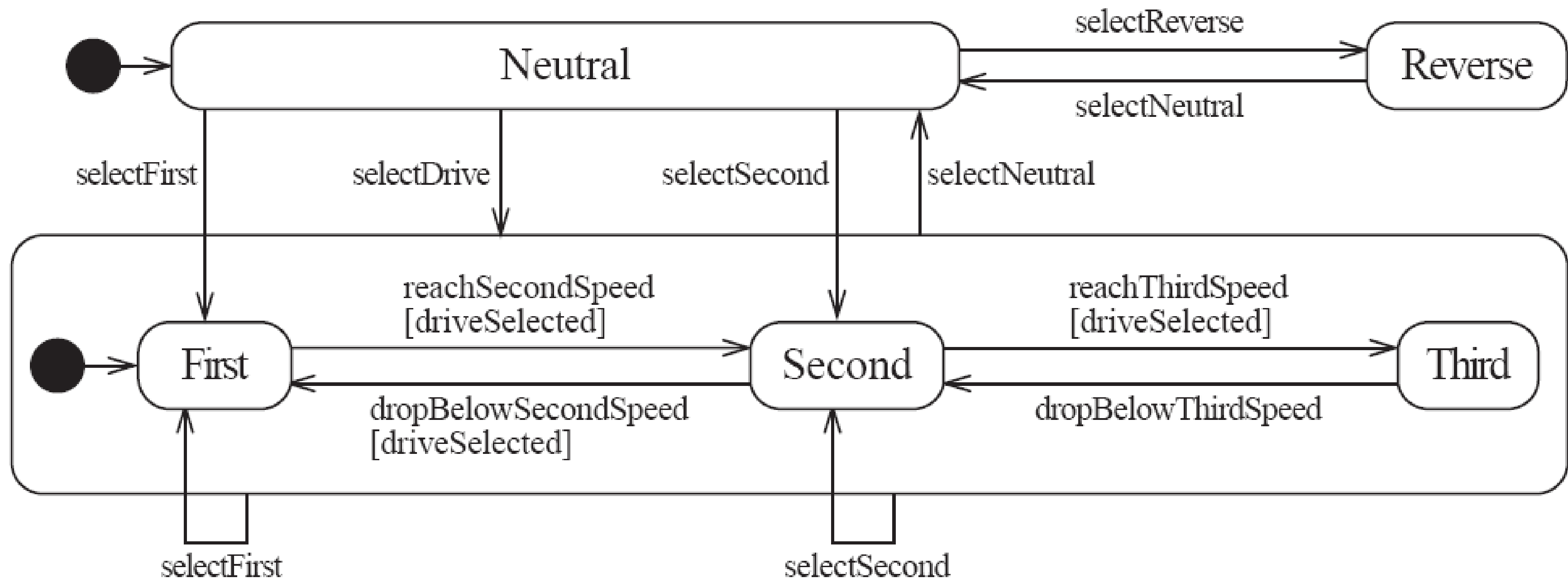
- An *action* is a block of code that must be executed effectively *instantaneously*
  - —When a particular transition is taken,
  - —Upon entry into a particular state, or
  - —Upon exit from a particular state

- An action should consume no noticeable amount of time

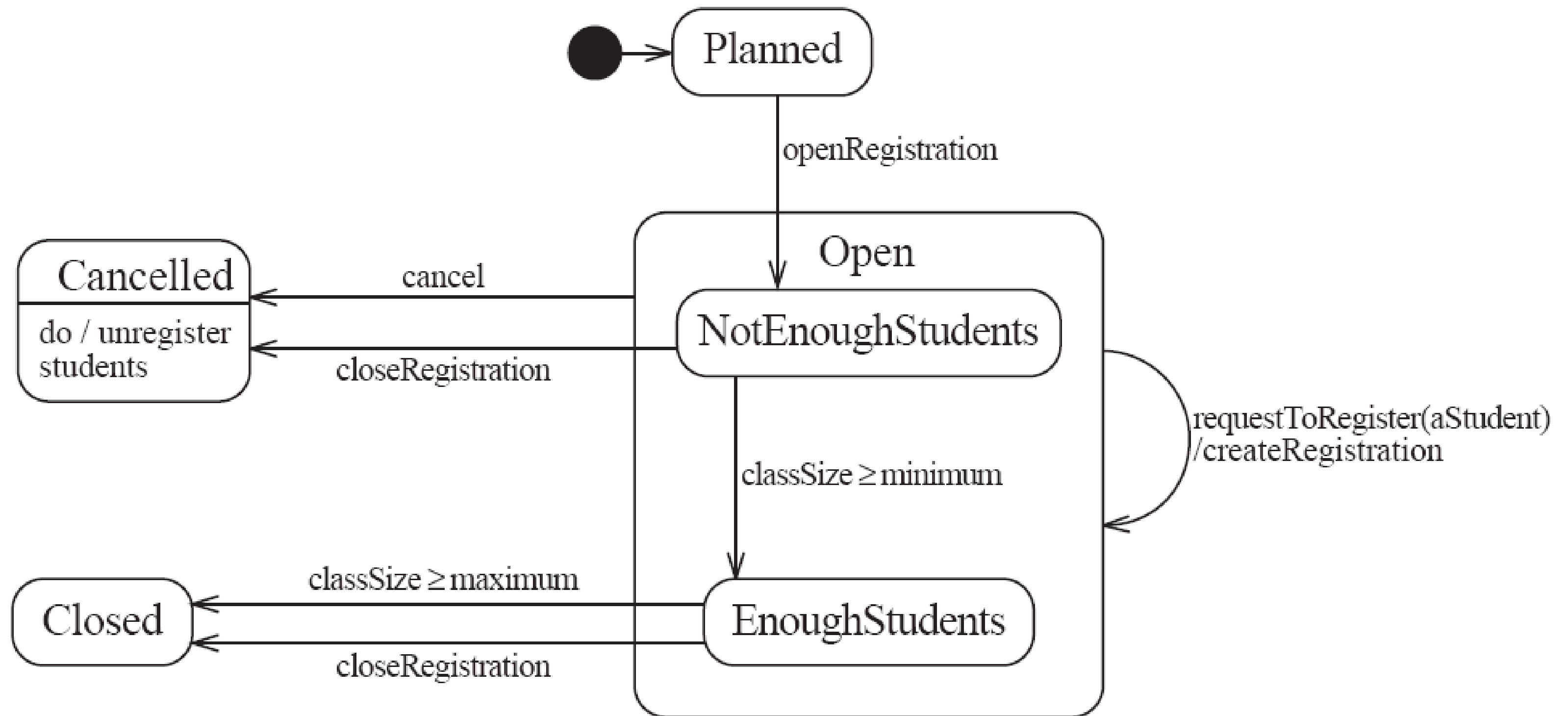
Nested substates and guard conditions

A state diagram can be nested inside a state.

- The states of the inner diagram are called *substates*.



Nested state diagram – Another example



Auto-transitions

A transition taken immediately upon entry into a state

- Unless guarded

We will look at an example in the user manual

Events with parameters

Parameters can be referenced in guards and actions.

We will look at an example in the user manual.

Analysing models

Models can be analysed in several ways

Visually

Automatically generated errors and warnings

State tables (next slide)\

Metrics

Formal methods (nuXMV)

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

Concurrency

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.

Declared with the keyword

`active`

Default threading in state machines

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

- A single 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

Solve 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 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*

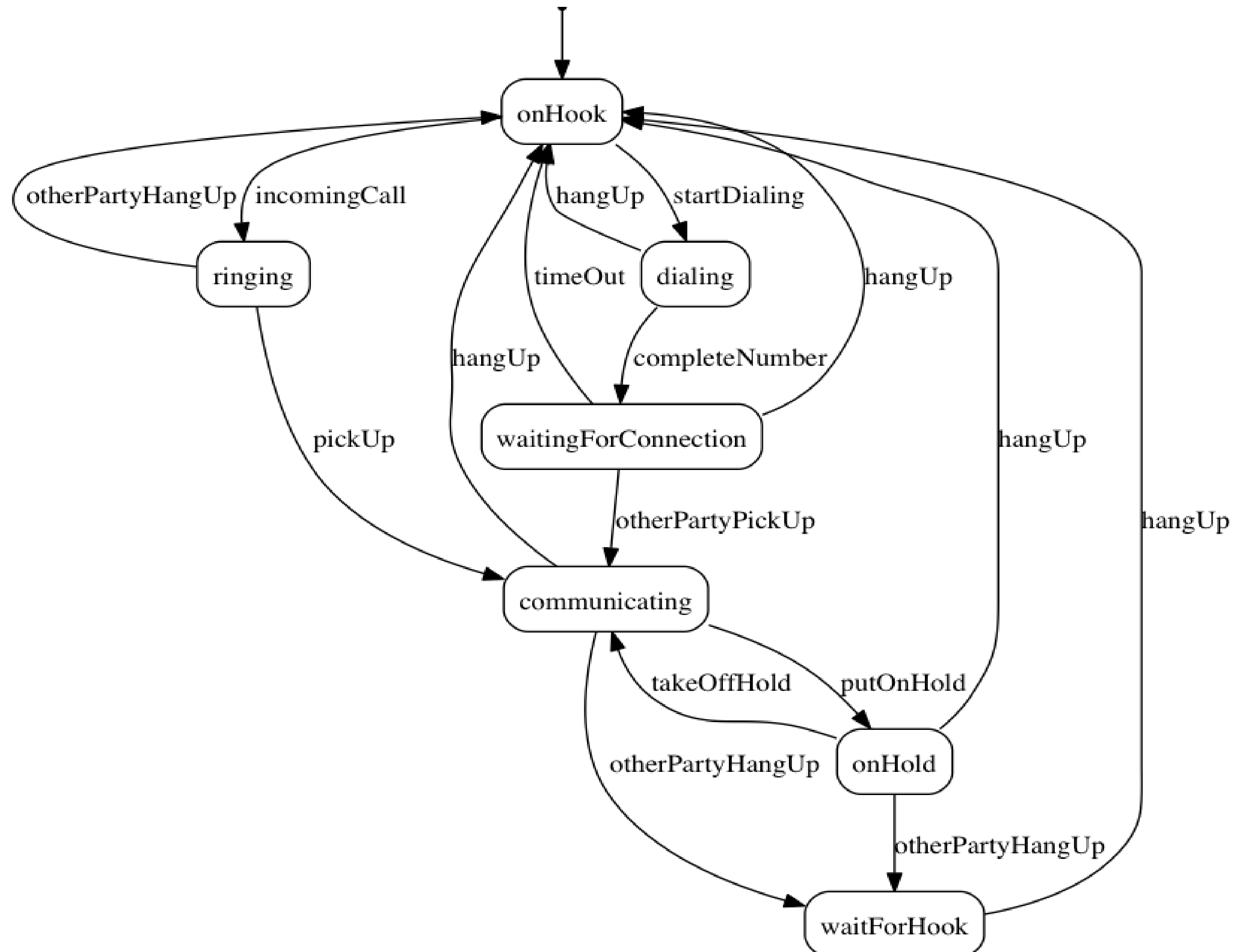
```
class AutomatedTellerMachine{
  queued sm  {
    idle {
      cardInserted -> active;      maintain -> maintenance;
      unspecified -> error1;
    }
    maintenance { isMaintained -> idle; }
    active {
      entry /{addLog("Card is read");}
      exit /{addLog("Card is ejected");}
      validating {
        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;}
  }
}
```

State machines in the user manual

http://statemachines.umple.org

State machine case study

State machine for a phone line



Umple for the phone line example

```
class phone {
  state {
    onHook {
      startDialing -> dialling;
      incomingCall -> ringing;
    }
    ringing {
      pickUp -> communicating;
      otherPartyHangUp -> onHook;
    }
    communicating {
      hangUp -> onHook;
      otherPartyHangUp -> waitForHook;
      putOnHold -> onHold;
    }
    onHold {
      hangUp -> onHook;
      otherPartyHangUp -> waitForHook;
      takeOffHold -> communicating;
    }
    dialing {
      completeNumber ->
waitingForConnection;
      hangUp -> onHook;
    }
    waitingForConnection {
      otherPartyPickUp -> communicating;
      hangUp -> onHook;
      timeOut -> onHook;
    }
    waitForHook {
      hangUp -> onHook;
    }
  }
}
```

In-class modeling exercise for state machines

Microwave oven system state machine

- Events include
  - pressing of buttons
  - door opening
  - door closing
  - timer ending
  - etc.

Outline

1. Introduction:
2. Overview of Model-Driven Development
   - Languages / Tools / Motivation for Umple
3. Class Modeling
   - Tools / Attributes / Methods / Associations / Exercises / Patterns
4. Modeling with State Machines
   - Basics / Concurrency / Case study and exercises
5. Separation of Concerns in Models
   - Mixins / Aspects / Traits
6. More Case Studies and Hands-on Exercises
   - Umple in itself / Real-Time / Data Oriented
7. Conclusion

Mixins

Separation of concerns by mixins in Umple

Mixins allow including attributes, associations, state machines, groups of states, stereotypes, etc

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 an other file
- In two separate files, with a third file invoking them

Typical ways of using mixins

Separate model files (classes, attributes associations)

… from files for the same class containing methods

- Allows a clearer view of the core model

Separate system features, each into a separate file

Advantages and disadvantages of mixins

Advantages:

- Smaller files that are easier to understand
- Different versions of a class for different software versions (e.g. a professional version) can be built by using different mixins

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

Aspect orientation

Aspect orientation

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

- The pointcut uses a *pattern*
- 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

Delocalization even stronger than for mixins

Aspect orientation in Umple

Pointcuts are currently limited to a single class

- Just inject code before and after execution of methods and constructors

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

We have found these limited abilities nonetheless solve key problems

Traits

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*/ }
}
```

Outline

1. Introduction: Who am I and who are you?
2. Overview of Agility
3. Overview of Model-Driven Development
   - Languages / Tools / Motivation for Umple
4. Agile Class Modeling
   - Tools / Attributes / Methods / Associations / Exercises / Patterns
5. Agile Modeling with State Machines
   - Basics / Concurrency / Case study and exercises
6. Separation of Concerns in Models
   - Mixins / Aspects / Traits
7. **More Case Studies and Hands-on Exercises**
   - **Umple in itself / Real-Time / Data Oriented**
8. Conclusion

Unit Testing with Umple

To see how to integrate Unit Testing with Umple, see the sample project at

- https://github.com/umple/umple/tree/master/sandbox

And the build script at

- https://github.com/umple/umple/blob/master/build/build.sandbox.xml

Command line from build directory

```
ant -f build.xml sandbox
```

A Look at How Umple is Written in Itself

Source:

https://github.com/umple/umple/tree/master/cruise.umple/src

Umple's own class diagram generated by itself from itself:

- http://metamodel.umple.org
- Colours represent key subsystems
- Click on classes to see Javadoc, and then Umple Code

Testing:
TDD with100% pass always required

Multiple levels: https://cruise.eecs.uottawa.ca/qa/index.php

- Parsing tests: basic constructs

- Metamodel tests: ensure it is populated properly
  - E.g.
  - https://github.com/umple/umple/blob/master/cruise.umple/test/cruise/umple/compiler/AssociationTest.java

- Implementation template tests: to ensure constructs generate code that looks as expected

- Testbed semantic tests: Generate code and make sure it behaves the way it should

Umple issues list

Tagged by

Priority

Perceived difficulty

Scale (bug, project, research project)

Milestone (slow release)

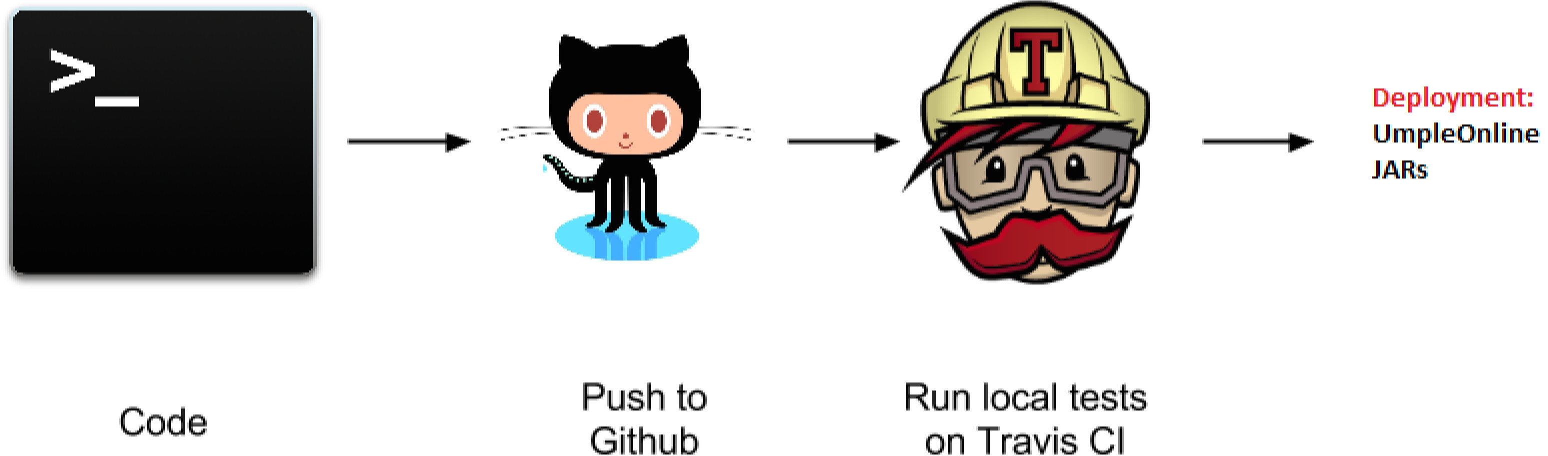
http://bugs.umple.org

Using Umple with Builds and Continuous Integration

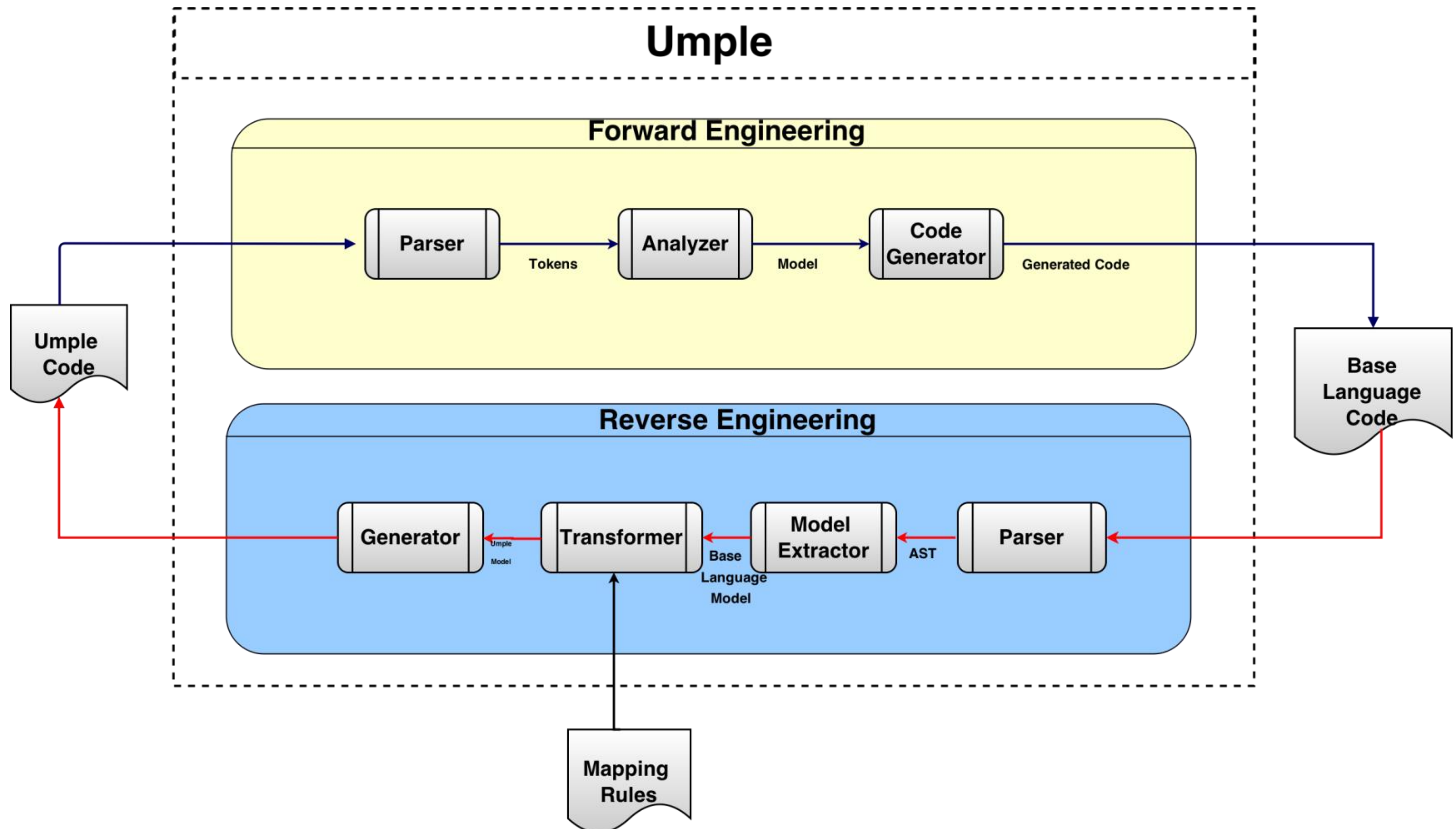
Example build scripts

Example travis.yml

Umple's own Travis page



Umple's Architecture



Umplification - Example

Person.java

```
1 package university;
2 public class Person {
3  public String getName() {return this.name;}
4  public void setName(String name){
5   this.name= name;
6  }
7 }
```

Listing 3.2: Student.java

```
20 package university;
21
22 public class Student extends
      Person{
23
24  public static final int
      MAX_PER_GROUP = 10;
25  private int id;
26  private String name;
27  public Mentor mentor;
28
29  public Student(int id,String
      name){
30   id = id; name = name;
31  }
32  public String getName(){
33   String aName = name;
34    if (name == null) {
35     throw new RuntimeException("
      Error");
36    }
37    return aName;
38  }
39  public Integer getId() {
40   return id;
41  }
42  public void setId(Integer id) {
43   this.id = id;
44  }
45  public boolean getIsActive() {
46   return isActive;
47  }
48  public void setIsActive(boolean
      aIsActive) {
49   isActive = aIsActive;}
50  }
51  public Mentor getMentor() {
52   return mentor;
53  }
54  public void setMentor(Mentor
      mentor) {
55   this.mentor = mentor;
56  }
57 }
```

Listing 3.3: Mentor.java

```
1 package university;
2 import java.util.Set;
3
4 public class Mentor extends
     Person{
5
6  Mentor() {}
7  public Set<Student> students;
8  public Set<Student> getStudents
     () {
9   return students;
10 }
11 public void setStudents (Set<
     Student>students) {
12  this.students = students;
13 }
14 public void addStudent( Student
     aStudent){
15  students.add(aStudent);
16 }
17 public void removeStudent(
     Student aStudent) {
18  students.remove(aStudent);
19 }
20 public String toString() {
21      return(
22         (name==null ? " " : name
     ) + " " +
23         students.size()+ "
     students"
24      );
25 }
26 }
```

UMPLE

Conclusion

Umple

- Is simple but powerful modeling tool
- Generates state-of-the-art code
- Enables agility + model-driven development

- We call the overall approach **model-based programming**

Thank-you!