Modelling small programs
Modelling large programs
Modelling application frameworks
Modelling design patterns

Specification language: desiderata (cont.)

- Combine theory & practice

It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing.

---

LePUS3 and Class-Z: desiderata

- Abstraction
  - Abstract ontology
  - Offer an answer: What are the conceptual building blocks of software design?
  - Scaling: capture the building blocks of very large programs
  - Detailed notation—structural diagrams
  - What NOT to model? What a specification should NOT represent?

- Rigour
  - Formal specification
  - Verification

- Decidability: Tool support in “round-trip engineering”
  - Automated verification is possible in principle
  - Tool support automates the verification process
  - Allows us to “close the loop” of round-trip engineering

- Visualization (Optional)
  - Offer a “picture” of a specification:
    - Existing program: a ‘roadmap’ to the millions of lines of code
    - A design motif: design pattern, architectural style
    - Makes software easier to understand and change

- Theory & practice
  - Be relevant to practical applications
  - Provide a sound foundation in a solid mathematical theory

What can be modelled in LePUS3/Class-Z?

Programs

Design patterns

Application frameworks

- Collections in java.util
- Collections in java.util
- The Iterator pattern
- Java Servlet
What cannot be modelled in LePUS3/Class-Z?

- Temporal relations
  - 'Method x should be called before method y'
  - No collaboration/interaction diagrams, flow charts, statecharts, ...
  - Statements about specific objects
- Strategic design
  - Architectural styles
  - Components
- Programs in procedural/functional/other programming paradigms
  - LePUS3 and Class-Z model object-oriented programs

LePUS3 vs. Class-Z

- Specifications can be expressed in one of two ways:
  - LePUS3: Visual, similar to UML Class Diagrams
  - Class-Z: symbolic, similar to Z
- Students are only required to learn one of the notations

The genealogy of LePUS3 and Class-Z

- Definition: As a subset of first-order predicate calculus (classical logic)
- Official reference manual:
- Historical roots:
  - Class-Z: Z [Spivey]
Modelling small programs in LePUS3 and Class-Z

Class and signature constants
Unary relation symbols
Binary relation symbols

Modelling individual classes
- **class constant**: represents any specific static type such as a class, Java interface, and primitive types

Modelling individual methods
- **signature constant**: represents a specific ‘signature’ of (one or more) methods

Modelling properties
- **Unary relation**: represents a property

Example:
```
linkedList, collection, int : CLASS
```

Example:
```
Method(newItr@linkedList)
Method(newItr@collection)
```
Modelling relations between individuals

- **Binary relation**: represents a relation between two individuals

  ![Diagram of binary relations between LinkedList, AbstractSet, ListIterator, and ListIterator]

  **Example**
  
  \[
  \text{LinkedList, AbstractSet, ListIterator, ListIterator : CLASS} \\
  \text{Inherit(LinkedList, AbstractSet)} \\
  \text{Inherit(ListIterator, ListIterator)}
  \]

Binary relations: *Member*

- **Example**
  
  \[
  \text{LinkedList, AbstractSet, ListIterator, ListIterator : CLASS} \\
  \text{Inherit(LinkedList, AbstractSet)} \\
  \text{Inherit(ListIterator, ListIterator)}
  \]

Binary relations: *Aggregate*

- **Example**
  
  \[
  \text{LinkedList, object : CLASS} \\
  \text{Aggregate(LinkedList, object)}
  \]

Binary relations: *Call*

- **Note that the Call relation abstracts away all information about the control-flow**

  ![Diagram of call relations between Test, PrintStream, and System]

  **Example**
  
  \[
  \text{Test, PrintStream : CLASS} \\
  \text{main, print : SIGNATURE} \\
  \text{Call(main, test, print) @ PrintStream}
  \]
**Binary relations: Forward**

```
public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest rq, HttpServletResponse rs) {
        super.doGet(rq, rs);
        // Method MyServlet.doGet forwards the call to HttpServlet.doGet
    }
}
```

**Example**

```
myServlet.httpServlet : CLASS
  doGet : SIGNATURE
  Forward(Forward(myServlet, doGet @ HttpServlet))
```

**Binary relations: Produce**

```
public class LinkedList {...
    public ListIterator listIterator(int index) {
        if (1 < 0)
            return new ListItr(index);
        return new ListItr(listItr(index));
    }
}
```

**Example**

```
linkedList.listItr : CLASS
  listIterator : SIGNATURE
  Produce(linkedList, listItr, listItr)
```

**Binary relations: Create**

```
public class MyClass {
    public void method() {
        ... for (int index = 0; ...)
        ...
    }
}
```

**Example**

```
myClass.method : METHOD
    Create(int)
```

**Modelling indirect relations**

- **Transitive relation**: represents possibly indirect relation (the ‘transitive closure’ of a binary relation)

```
interface Collection {...
    class AbstractList implements Collection {...
    class AbstractSet extends AbstractList ...
    class LinkedList extends AbstractSet ...
}
```

**Example**

```
LinkedList.linkedList : CLASS
  Inherit* (linkedList, collection)
```

- **Produce**: A special kind of a Create relation in which the new object is returned (typical to ‘factory methods’)

```
Method LinkedList.listIterator may create and return a new ListItr
```

**Example**

```
linkedList.listItr : CLASS
  listIterator : SIGNATURE
  Produce(listIterator @ linkedList, listItr)
```

- **Forward**: A special kind of a Call relation in which the formal arguments are passed on to a method with same signature

```
Method MyServlet.doGet forwards the call to HttpServlet.doGet
```

**Example**

```
myServlet.httpServlet : CLASS
  doGet : SIGNATURE
  Forward(Forward(myServlet, doGet @ HttpServlet))
```
Transitive relations II

```
public class FileInputStream ... {
  public int read(byte[] b) throws IOException, NullPointerException {
    ... 

  }
}
```

Example

```
fileInputStream, IOException, nullPointerException : CLASS
read, nullPointerException, IOException : SIGNATURE

public FileInputStream(fileInputStream, IOException@IOException)
public FileInputStream(fileInputStream, nullPointerException@nullPointerException)
```

---

Case Study II: java.io.LineNumberReader classes

```
public class LineNumberReader
    extends BufferedReader {

  public void read() {
    ... 
  }

  public void mark(int readAheadLimit) {
    ... 

  super.mark(readAheadLimit); ... 

  public void reset() {
    ... 

  reset(); ... 

  }
}
```

```
ReaderExample
bufferedReader, lineNumberReader : CLASS
read, mark, reset : SIGNATURE

Forward(read@LineNumberReader, read@BufferedReader)
Forward(mark@LineNumberReader, mark@BufferedReader)
Forward(reset@LineNumberReader, reset@BufferedReader)
```

---

Exercise

- "Translate" the LePUS3 chart in the case study into a Class-Z specification
- Use either Class-Z or LePUS to model three DIFFERENT specifications of the classes Collection, Iterator, LinkedList, and ListIterator in the package java.util.
  - Each specification should include one or more of the methods defined in these classes. Which ones did you choose to model and why?
  - Note that each specification may contain a different set of relations
  - Note that each specification must be SATISFIED by the classes in java.util
Exercise (Cont.)

Which one is the most correct description of the purpose of a LePUS/Class-Z specification?
1. Each specification is model of a particular implementation
2. Each specification models an one or more possible implementations
3. Each specification models an aspect of one implementation abstracted for a particular purpose
4. Each specification models an aspect of one or more possible implementations abstracted for a particular purpose

Exercise (Cont.)

How many ways can you model a Java program in which class MyClass has a method therein with the signature method(String arg) which throws an exception of class MyException in LePUS3 or Class-Z?

Modelling large programs in LePUS3 and Class-Z

Hierarchy constant
1-dim class and signature constants
Total and Isomorphic predicates

1-dimensional class constant: Stands for a (non-empty) set of specific static types
Example
ConcreteCollections : PCLASS

ConcreteCollections : HashSet, TreeSet, LinkedList, ArrayList

Dr Amnon H. Eden, School of Computer Science and Electronic Engineering, University of Essex
Modelling sets of methods (one signature)

- **Clan**: A set of methods with same signature

  ![Diagram of Clan]

  Example:
  ```
  ConcreteCollections : PCLASS
  newItr : SIGNATURE
  All(Method, newItr@ConcreteCollections)
  ```

Modelling sets of methods (many signatures)

- **1-dimensional signature constant**: Stands for a (non-empty) set of specific method signatures

  ![Diagram of 1-dimensional signature constant]

  Example:
  ```
  1-DimSignatureConstant : PSIGNATURE
  ```

Modelling sets of methods (many signatures)

- **Tribe**: A set of methods in same class

  ![Diagram of Tribe]

  Example:
  ```
  httpServlet : CLASS
  ServletOps : PSIGNATURE
  All(Method, ServletOps@httpServlet)
  ```

Modelling sets of methods (many signatures): example

![Diagram of example]

Example:
```
bufferedReader : CLASS
ServletOps : PSIGNATURE
bufferedReader : CLASS
All(Method, ServletOps@bufferedReader)
```
Modelling relations between sets: *Total*

Every element in Domain is in relation "Relation with some element in Range".

\[ Total(\text{BinaryRelation}, \text{Domain}, \text{Range}) \]

Total predicate: example

Example

ConcreteCollections : \text{CLASS}
collection : \text{CLASS}
Total(\text{Inherits}, \text{ConcreteCollections}, \text{collection})

Modelling relations between sets: *Isomorphic*

Every element in Domain is in relation "BinaryRelation with a unique element in Range".

\[ \text{Isomorphic}(\text{BinaryRelation}, \text{Domain}, \text{Range}) \]
Isomorphic predicate: example I

Example
bufferedReader, lineNumberReader : CLASS
BufferOps : PCLASS
Inherit(lineNumberReader,bufferedReader)
Isomorphic(Forward,ServletOps@lineNumberReader,ServletOps@bufferedReader)

Isomorphic predicate: example II

Collections & iterators in java.util (cont.)
Exercise

- Translate the specification of collections and iterators to UML

Modelling class hierarchies

- 1-dimensional hierarchy constant: a set of classes s.t. all inherit from one

Hierarchy: example

Example

Modelling sets of methods revisited

Example

All(Method.sig@Hrc)
Example: the Lists hierarchy

Example: lists, sets, and sorted sets

Case study: collections & iterators in java.util

Modelling application frameworks
Variables vs. constants

### Example
```
aClass, object : CLASS
```

Variables vs. constants II

### Example
```
linkedList, abstractSet, x, y : CLASS
Inherit(linkedList, abstractSet)
Inherit(x, y)
```

Assignments

- A specification with variables is only meaningful wrt a specific assignment
- **Example: Assignment-1**
  - x is assigned to LinkedList
  - y is assigned to AbstractSet
- **Example: Assignment-2**
  - y is assigned to LinkedList
  - x is assigned to AbstractSet
- Assignment-1 is satisfied by java.util
- Assignment-2 is satisfied by java.util

Example: Java Servlets

```
httpServlet, userServlet : CLASS
ServletOps : \( \text{SIGNATURE} \)
\[ \text{Isomorphism}(\text{Forward}, \text{ServletOps} @ \text{userServlet}, \text{ServletOps} @ \text{httpServlet}) \]
Inherit(userServlet, httpServlet)
```

Servlet class must extend class HTTPServlet
Servlet class can be any class that satisfies the relations
Servlet class must override a particular set of methods
Each method in your Servlet must forward call to respective method in superclass
Variables vs. constants: example I

A specific implementation

Any Java Servlet

```java
public class HelloText extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) {
        super.doGet(request, response);
        response.setContentType("text/plain");
        PrintWriter out = response.getWriter();
        out.println("Hello World");
    }
}
```

Variables vs. constants: example II

Collections in java.util

Iterator pattern

```java
Collection
- Aggregate
- Return

Iterator
- Produce
- Next
- Object

Collections in java.util
- CreateIterator
- Next
- Produce
- Object

Composite pattern

Composite in java.awt

Case study: Enterprise JavaBeans

From [Monson-Haefel 2001]:

1. “Every bean [class] obtains an EJBContext object, which is a reference to the container
2. “The home interface extends the ... javax.ejb.EJBHome interface
3. “A home [interface] may have many create() methods, ... of which must have corresponding ejbCreate() and ejbPostCreate() methods in the bean class. The number and datatype of the arguments of each create() are left up to the bean developer”
4. “When a create() method is invoked on the home interface, the container delegates the invocation to the corresponding ejbCreate() and ejbPostCreate() methods on the bean class.
5. [An implementation for the bean’s home interface is generated by the container.]”
Modelling Enterprise JavaBeans I

"Every bean [class] obtains an EJBContext object, which is a reference to the container"

"The home interface extends the ...javax.ejb.EJBHome interface"

Modelling Enterprise JavaBeans II

"A home [interface] may have many create() methods, ... each of which must have corresponding ejbCreate() and ejbPostCreate() methods in the bean class. The number and datatype of the arguments of each create() are left up to the bean developer"

"When a create() method is invoked on the home interface, the container delegates the invocation to the corresponding ejbCreate() and ejbPostCreate() methods on the bean class"

Summary: Enterprise JavaBeans

Modelling design patterns

The 'gang of four' patterns: Iterator, Proxy, Composite, Observer, Factory Method, Adapter (Object), Adapter (Class), Strategy
**Factory Method pattern**

- **Factory Method**
  - Produce
  - Factories
  - Products

**FactoryMethod**

Factories, Products : HIERARCHY
factoryMethod : SIGNATURE

Isomorphic(Product, factoryMethod; Factories, Products)

**Adapter (Class) pattern**

- **Adapter (Class)**
  - client
  - Operations
  - Requests
  - SpecificRequests
  - Total

**ClassAdapter**

client, target, adapter, adaptee : CLASS
Operations, Requests, SpecificRequests : P(SIGNATURE)

Abstract(target)
Total Call, Operations@client, Requests@target)
Total Call, Request@adapter, SpecificRequests@adaptee)
Inherit(adapter,target)
Inherit(adapter,adaptee)

**Adapter (Object) pattern**

- **Adapter (Object)**
  - client
  - target
  - adapter
  - SpecificRequests
  - Request

**ObjectAdapter**

client, target, adapter, adaptee : CLASS
Operations, Requests, SpecificRequests : P(SIGNATURE)

Abstract(target)
Total Call, Operations@client, Requests@target)
Total Call, Request@adapter, SpecificRequests@adaptee)
Inherit(adapter,target)
Inherit(adapter,adaptee)

**Strategy pattern**

- **Strategy**
  - client
  - Request
  - Call

**Strategy**

client, context : CLASS
request, algorithm : SIGNATURE
Interface : P(SIGNATURE)
Strategies : HIERARCHY

Member(context, Strategies)
Call(request@client, request@context)
Call(request@context, algorithm@Strategies)
Total Call, algorithm@Strategies, Interface@context
Tool support in LePUS3 and Class-Z

The Two-Tier Programming Toolkit

- Round-trip engineering
- Supports:
  - Specification & verification (forward engineering)
  - Visualization (reverse engineering)
- [http://ttp.essex.ac.uk](http://ttp.essex.ac.uk)

The TTP Toolkit (cont.): Specification in LePUS3
The TTP Toolkit (cont.): Specification in Class-Z

The TTP Toolkit (cont.): Verification

The TTP Toolkit (cont.): Visualization

Exercises

1. Summarize the differences between LePUS3/Class-Z and UML:
   1.1 Compare the LePUS3 chart with the UML Class diagram of LinkedList and the ListItr classes
   1.2 Use UML to model the Collection and the Iterator hierarchies in Java.util. How is this diagram different from the LePUS3/Class-Z specification?
   1.3 Use UML to model Java Servlets. How is this diagram different from the LePUS3/Class-Z specification?
   1.4 Use UML to model one of the design patterns. How is this diagram different from the LePUS3/Class-Z specification?
2. List four abstraction mechanisms in LePUS3/Class-Z
Exercises (cont.)

3. Compare the LePUS3/Class-Z Specification of each one of the design patterns to the description in Gamma et al. 1995
3.1 What is the book saying that the chart/schema does not say?
3.2 What is the chart/schema saying that the book does not say?
3.3 What are the advantages of the book over the chart?
3.4 What are the advantages of the chart over the book?

LePUS3 quick reference

- constants and variables

LePUS3 quick reference (cont.)

Methods

- Relations and predicates

Exercises (cont.)

4. Specify in LePUS3/Class-Z the following statements:
4.1 "There exists a method in class MyClass which creates instances of class String"

Answer:

```
        MyClass
     ________
    |        |
    | Create  |
    | ________|
    | string  |
```

LePUS3 quick reference (cont.)

- constants and variables

LePUS3 quick reference (cont.)

- relations and predicates

Methods

- relations and predicates

Exercises (cont.)

4. Specify in LePUS3/Class-Z the following statements:
4.1 "There exists a method in class MyClass which creates instances of class String"

Answer:

```
        MyClass
     ________
    |        |
    | Create  |
    | ________|
    | string  |
```
References

- Legend: Key to LePUS3 and Class-Z Symbols [pdf]
  <http://www.lepus.org.uk/ref/legend/legend.xml>
  Unpublished manuscript (ask me for a copy)
  <http://lepus.org.uk/ref/refman/refman.xml>
  Formal specification of design patterns in LePUS3 and Class-Z." University of
  <http://lepus.org.uk/ref/companion/index.xml>
  Conf. Multidisciplinary Design in Engineering CSME-MDE 2001 (21–22
  Nov. 2001), Montreal, Canada.