Lecture notes in graduate module, CE839: Software Design and Architecture, Autumn term 2008/9
Dr Amnon H Eden, School of Computer Science and Electronic Engineering, University of Essex

Object-Oriented Modelling in LePUS3 and Class-Z

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

1

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—cluttered 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

2

Specification language: desiderata (cont.)

- Combine theory & practice

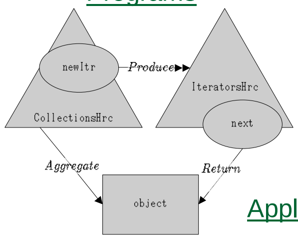
It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental 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.

-- Christopher Strachey

3

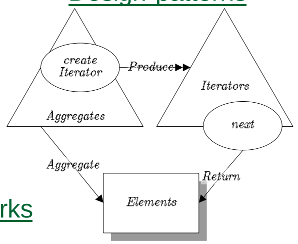
What can be modelled in LePUS3/Class-Z?

Programs



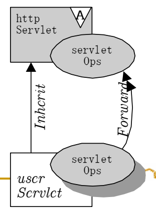
(Collections in java.util)

Design patterns



(The Iterator pattern)

Application frameworks



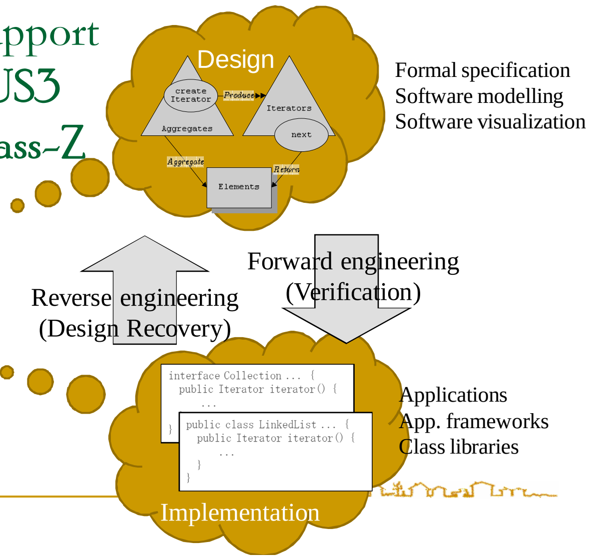
(Java Servlets)

4

What *cannot* be modelled in LePUS3/Class-Z?

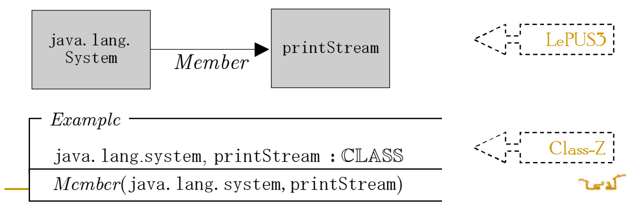
- Temporal relations
  - *'Method x should be called before method y'*
  - No collaboration/interaction diagrams, flow charts, statecharts, …
  - Statements about specific objects
- Strategic design
  - Architectural styles
  - Components
- Programs in procedural/functional/other programming paradigms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

Tool support in LePUS3 and Class-Z



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:
  - A.H. Eden, E. Gasparis, J. Nicholson. "LePUS3 and Class-Z Reference Manual". University of Essex, Tech. Rep. CSM-474, ISSN 1744-8050 (2007). http://lepus.org.uk/ref/refman/refman.xml
- Historical roots:
  - LePUS3: LePUS [Eden 2001]—LanguagE for Pattern Uniform Specification
  - Class-Z: Z [Spivey]

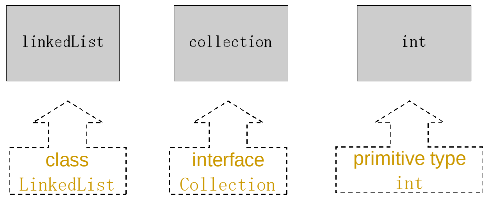
Modelling small programs in LePUS3 and Class-Z

Class and signature constants
Unary relation symbols
Binary relation symbols

9

Modelling individual classes

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



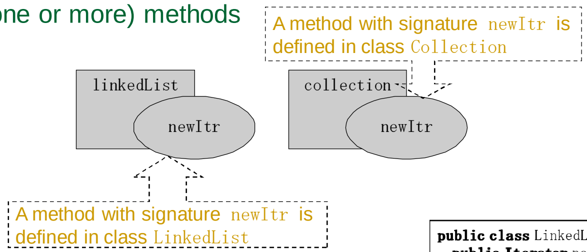
Example

$\mathtt{linkedList}, \mathtt{collection}, \mathtt{int} : \mathbb{CLASS}$

10

Modelling individual methods

- **signature constant**: represents a specific 'signature' of (one or more) methods




```
public class LinkedList ... {
  public Iterator newItr() {
    ...
  }
}
public class Collection ... {
  public Iterator newItr() {
    ...
  }
}
```

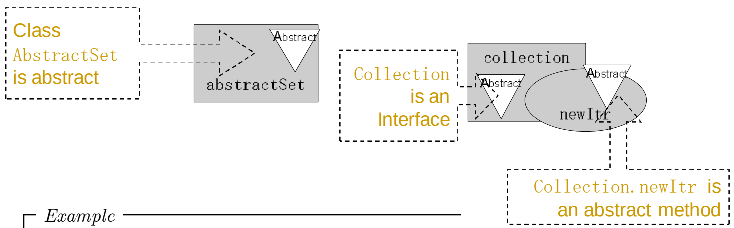
Example

$\mathtt{linkedList}, \mathtt{collection} : \mathbb{CLASS}$
$\mathtt{newItr} : \mathbb{SIGNATURE}$

$\mathit{Method}(\mathtt{newItr} \otimes \mathtt{linkedList})$
$\mathit{Method}(\mathtt{newItr} \otimes \mathtt{collection})$

Modelling properties

- **Unary relation**: represents a property



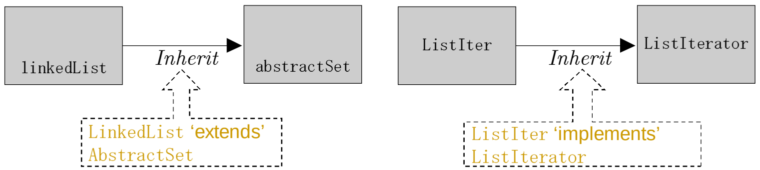
Example

$\mathtt{abstractSet}, \mathtt{collection} : \mathbb{CLASS}$
$\mathtt{newItr} : \mathbb{SIGNATURE}$

$\mathit{Abstract}(\mathtt{abstractSet})$
$\mathit{Abstract}(\mathtt{collection})$
$\mathit{Abstract}(\mathtt{newItr} \otimes \mathtt{collection})$

Modelling relations between individuals

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



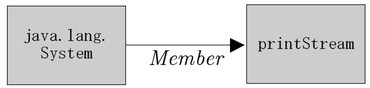
Example

linkedList, abstractSet, ListItr, ListIterator : CLASS

Inherit(linkedList,abstractSet)
Inherit(ListItr,ListIterator)

Binary relations: *Member*

```
public class System {
...
  public static PrintStream out;
...
}
```

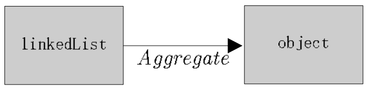


Example

java. lang.system, printStream : CLASS

Member(java. lang. system, printStream)

Binary relations: *Aggregate*



Example

linkedList, object : CLASS

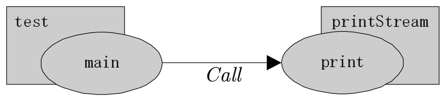
Aggregate(linkedList, object)

15

Binary relations: *Call*

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

```
public class Test {
  public static void main(String[] args) {
    if (args.length == 0)
      System.out.print("Insufficient arguments");
    ...
```



Example

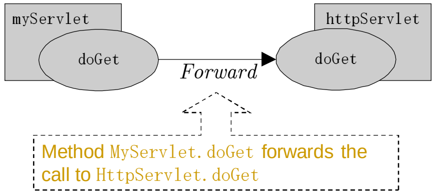
test, printStream : CLASS
main, print : SIGNATURE

Call(main⊗test, print⊗printStream)

Binary relations: *Forward*

```
public class MyServlet extends HttpServlet {
  public void doGet(HttpServletRequest rq, HttpServletResponse rs) {
    super.doGet(rq,rs);
  }
```

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



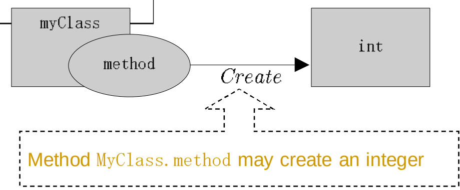
Example

mySerlvet, httpServlet : CLASS
doGet : SIGNATURE

Forward(doGet⊗mySerlvet, doGet⊗httpServlet)

Binary relations: *Create*

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



Example

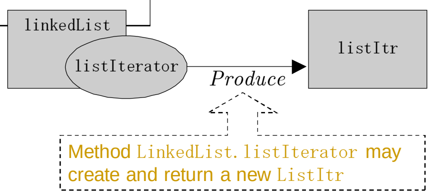
linkedList, listItr : CLASS
listIterator : SIGNATURE

Produce(listIterator⊗linkedList, listItr)

Binary relations: *Produce*

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

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



Example

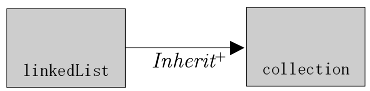
linkedList, listItr : CLASS
listIterator : SIGNATURE

Produce(listIterator⊗linkedList, listItr)

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  ...
```



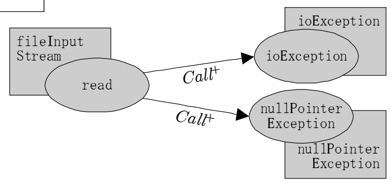
Example

linkedList, collection : CLASS

Inherit$^+$(linkedList, collection)

Transitive relations II

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



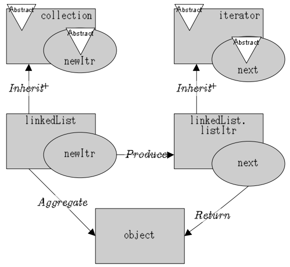
Example

fileInputStream, ioException, nullPointerException : ℂLASS
read, nullPointerException, ioException : SIGNATURE

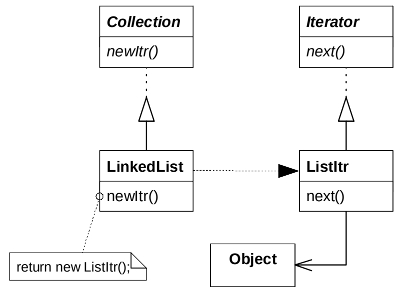
Call+(read⊗fileInputStream, ioException⊗ioException)
Call+(read⊗fileInputStream, nullPointerException⊗nullPointerException)

Case Study I: LinkedList and ListItr

LePUS3

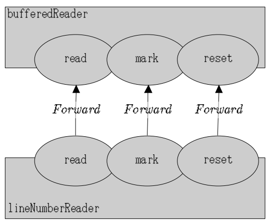


UML Class diagram



Case Study II: java.io.XXReader classes

```
public class LineNumberReader
  extends BufferedReader { ...
  public void read() { ...
    super.read(); ...
  } ...
  public void mark(int readAheadLimit) { ...
    super.mark(readAheadLimit); ...
  } ...
  public void reset() { ...
    super.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 `ListItr` 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

25

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?

26

Modelling large programs in LePUS3 and Class-Z

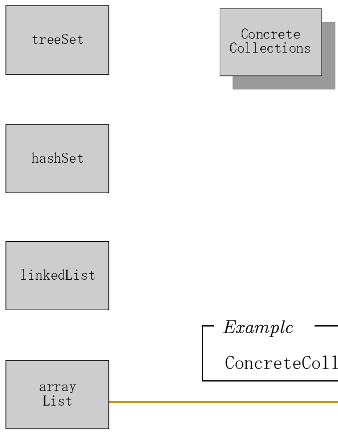
Hierarchy constant

1-dim class and signature constants

Total and Isomorphic predicates

27

Modelling sets of classes



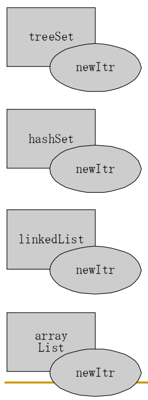
- **1-dimensional class constant**: Stands for a (non-empty) set of specific static types

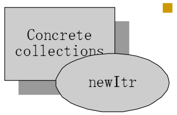
Example

ConcreteCollections : $\mathcal{P}$CLASS

28

Modelling sets of methods (one signature)





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

Example

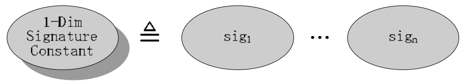
ConcreteCollections : $\mathcal{P}$CLASS
newItr : SIGNATURE

All(*Method*, newItr⊗ConcreteCollections)

29

Modelling sets of methods (many signatures)

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



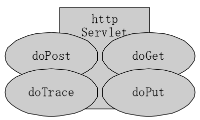
Example

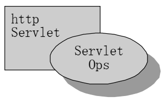
1-DimSignatureConstant : $\mathcal{P}$SIGNATURE

30

Modelling sets of methods (many signatures)

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





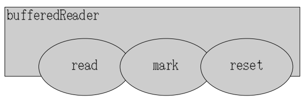
Example

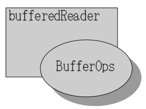
httpServlet : CLASS
ServletOps : $\mathcal{P}$SIGNATURE

All(*Method*, ServletOps⊗httpServlet)

31

Modelling sets of methods (many signatures): example





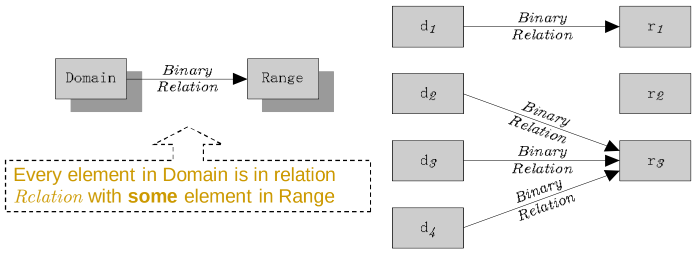
Example

BufferOps : $\mathcal{P}$SIGNATURE
bufferedReader : CLASS

All(*Method*, BufferOps⊗ bufferedReader)

32

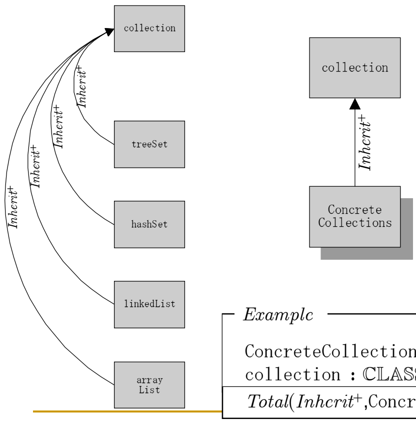
Modelling relations between sets: *Total*



Total(*BinaryRelation*,Domain,Range)

33

Total predicate: example



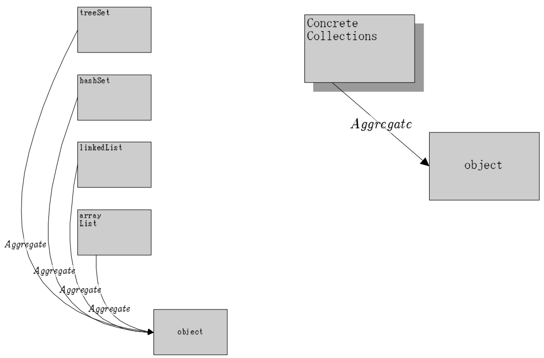
Example

ConcreteCollections : $\mathbb{P}$CLASS
collection : CLASS

Total(*Inherit*$^+$,ConcreteCollections,collection)

34

Total predicate: example II



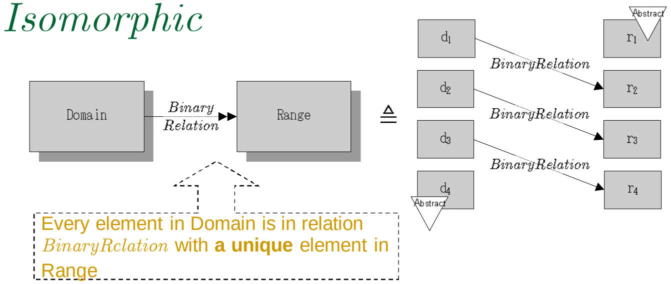
Example

ConcreteCollections : $\mathbb{P}$CLASS
object : CLASS

Total(*Aggregate*,ConcreteCollections,object)

35

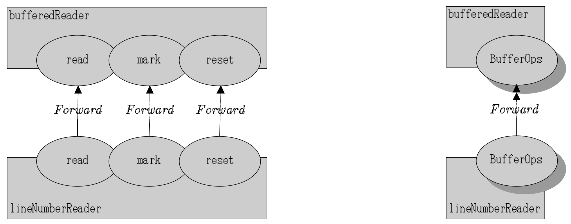
Modelling relations between sets: *Isomorphic*



Isomorphic(*BinaryRelation*,Domain,Range)

36

Isomorphic predicate: example I



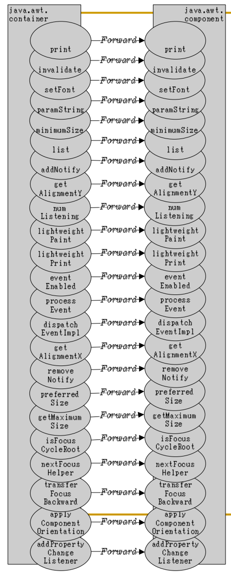
Example

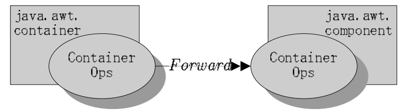
bufferedReader, lineNumberReader : $\mathbb{CLASS}$
BufferOps : $\mathcal{P}\mathbb{SIGNATURE}$

Inherit(lineNumberReader,bufferedReader)
Isomorphic(*Forward*, ServletOps⊗lineNumberReader,ServletOps⊗bufferedReader)

37

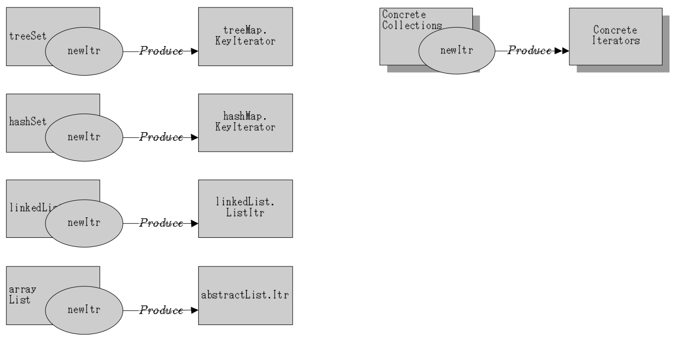
Isomorphic predicate: example II





38

Case study: collections & iterators in java.util



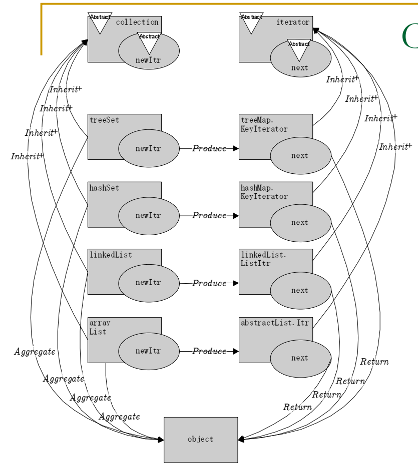
Example

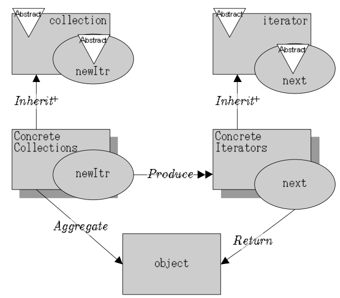
ConcreteCollections, ConcreteIterators : $\mathcal{P}\mathbb{CLASS}$

Isomorphic(*Produce*, newItr⊗ConcreteCollections, ConcreteIterators)

39

Collections & iterators in java.util (cont.)





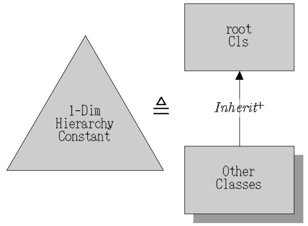
40

Exercise

- Translate the specification of collections and iterators to UML

Modelling class hierarchies

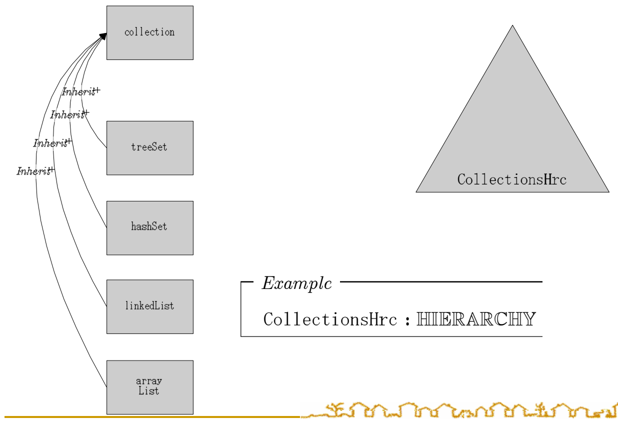
- **1-dimensional hierarchy constant**: a set of classes s.t. all inherit from one



Example

Hierarchy : HIERARCHY

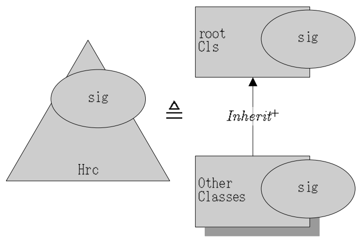
Hierarchy: example



Example

CollectionsHrc : HIERARCHY

Modelling sets of methods revisited

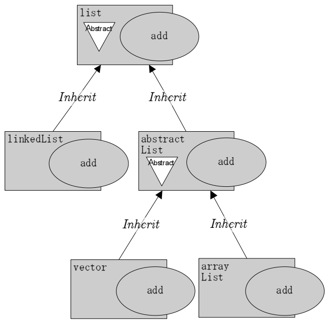


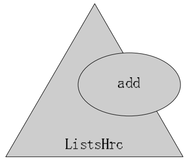
Example

Hrc : HIERARCHY
sig : SIGNATURE

$All(Method, sig \otimes Hrc)$

Example: the Lists hierarchy





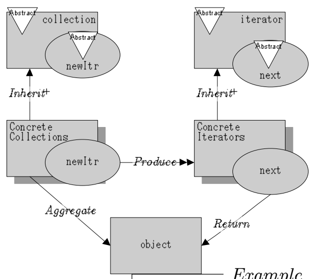
Example

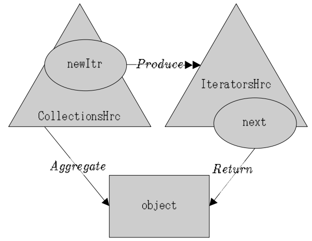
Lists : $\mathbb{HIERARCHY}$
add : $\mathbb{SIGNATURE}$

All(*Method*, add⊗Lists)

45

Case study: collections & iterators in java.util





Example

CollectionsHrc, IteratorsHrc : $\mathbb{HIERARCHY}$
next, newItr : $\mathbb{SIGNATURE}$
object : $\mathbb{CLASS}$

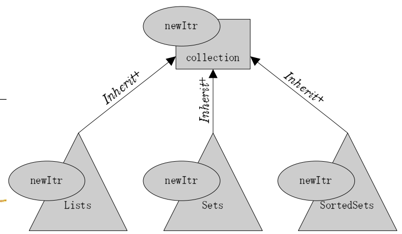
Isomorphic(*Produce*, newItr⊗CollectionsHrc, IteratorsHrc)
Total(*Return*, next⊗IteratorsHrc, object)
Total(*Aggregate*, CollectionsHrc, IteratorsHrc)

Example: lists, sets, and sorted sets

Example

Lists, Sets, SortedSets : $\mathbb{HIERARCHY}$
newItr : $\mathbb{SIGNATURE}$
collection : $\mathbb{CLASS}$

Total($Inherit^{+}$, Lists, collection)
Total($Inherit^{+}$, Sets, collection)
Total($Inherit^{+}$, SortedSets, collection)
Method(newItr⊗collection)
All(*Method*, newItr⊗Lists)
All(*Method*, newItr⊗Sets)
All(*Method*, newItr⊗SortedSets)

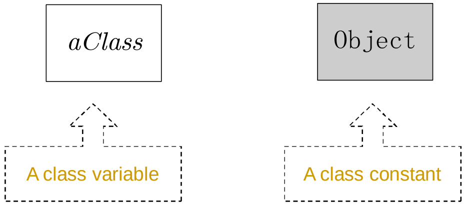


Modelling application frameworks

Variables

48

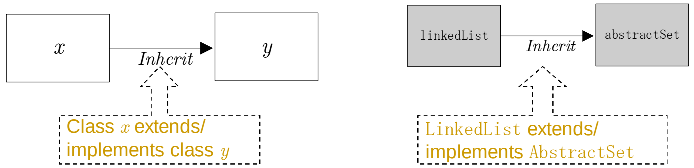
Variables vs. constants



Example

$aClass$, `Object` : CLASS

Variables vs. constants II



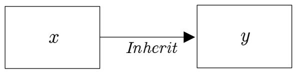
Example

`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.uti

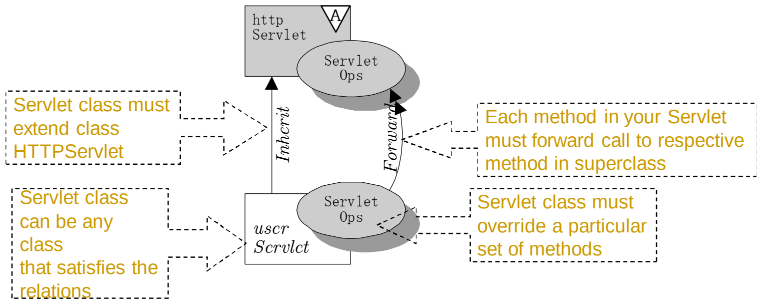


Example

x, y : CLASS

Inherit(x,y)

Example: Java Servlets



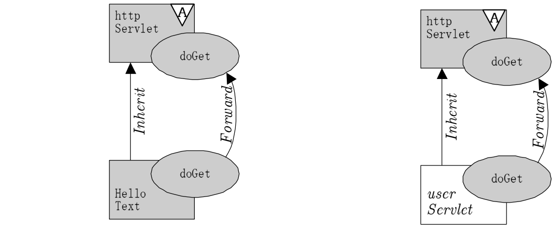
JavaServlet

`httpServlet`, *userServlet* : CLASS
`ServletOps` : $\mathbb{P}$(SIGNATURE)

Isomorphic(*Forward*, `ServletOps`⊗*userServlet*, `ServletOps`⊗`httpServlet`)
Inherit(*userServlet*, `httpServlet`)

Variables vs. constants: example I

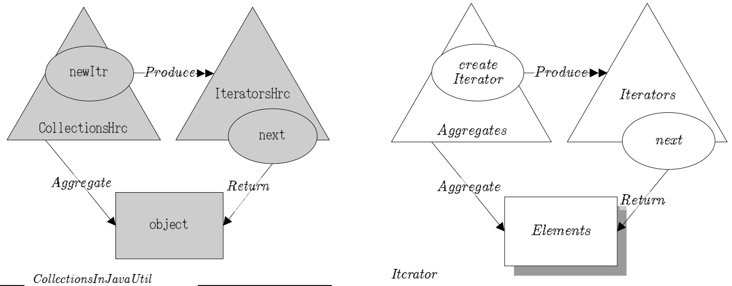
A specific implementation — Any Java Servlet




```
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



CollectionsInJavaUtil

Collections, Iterators : HIERARCHY
next, newItr : SIGNATURE
object : CLASS

Isomorphic(*Produce*, newItr⊗Collections, Iterators)
Return(next⊗Iterators, object)

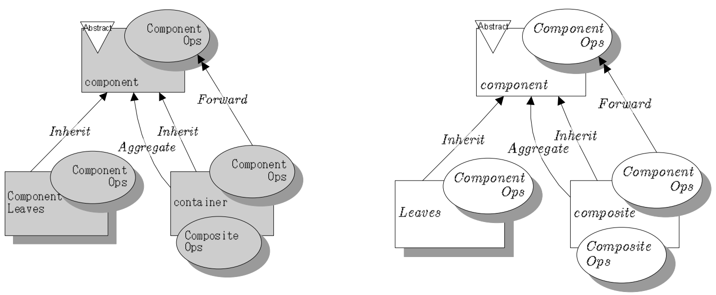
Iterator

Aggregates, Iterators : HIERARCHY
createIterator, next : SIGNATURE
object : CLASS

Isomorphic(*Produce, createIterator*⊗*Aggregates,Iterators*)
Return(*next*⊗*Iterators, object*)

Variables vs. constants: example III

Composite in java.awt — Composite pattern

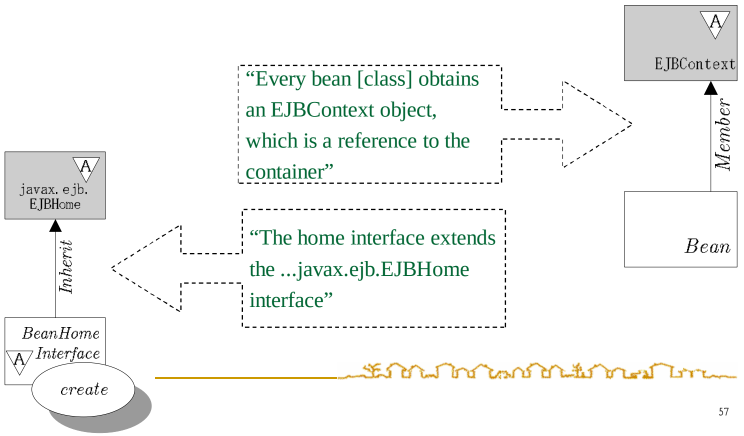


Case study: Enterprise JavaBeans

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, … , 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"
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.]"

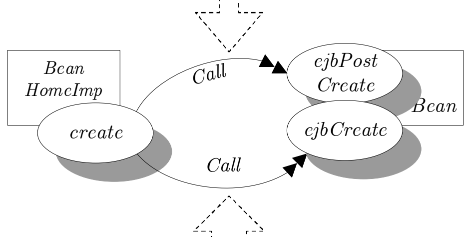
Modelling Enterprise JavaBeans I



57

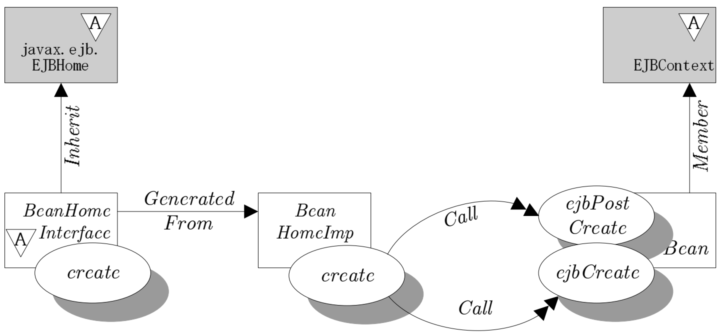
Modelling Enterprise JavaBeans II

“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



59

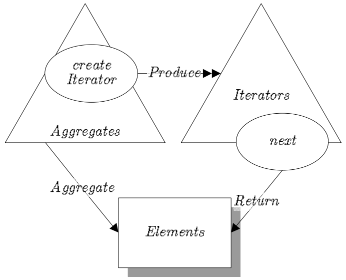
Modelling design patterns

The ‘gang of four’ patterns:

Iterator, Proxy, Composite, Observer, Factory Method, Adapter (Object), Adapter (Class), Strategy

60

Iterator pattern



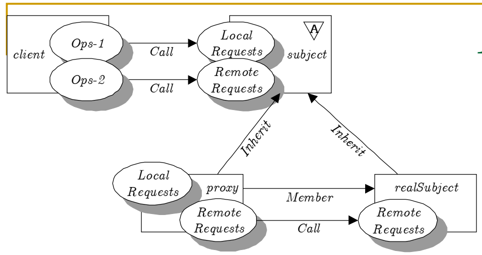
Iterator

Aggregates, Iterators : HIERARCHY
createIterator, next : SIGNATURE
Elements : ℘CLASS

Isomorphic(Produce, createIterator⊗Aggregates, Iterators)
Total(Return, next⊗Iterators, Elements)
Total(Aggregate, Aggregates, Elements)

61

Proxy pattern



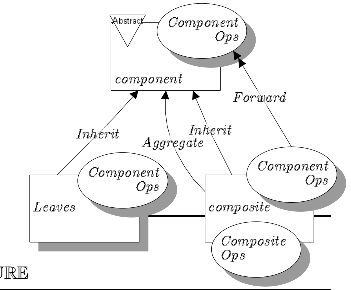
Proxy

client, subject, proxy, realSubject : CLASS
Ops-1, Ops-2, LocalRequests, RemoteRequests : ℘SIGNATURE

Abstract(subject)
Total(Call, Ops-1⊗client, LocalRequests⊗subject)
Total(Call, Ops-2⊗Client, RemoteRequests⊗subject)
Total(Call, RemoteRequests⊗proxy, RemoteRequests⊗realSubject)
Method(LocalRequests⊗proxy)
Member(proxy,realSubject)
Inherit(proxy,subject)
Inherit(realSubject,subject)

62

Composite pattern



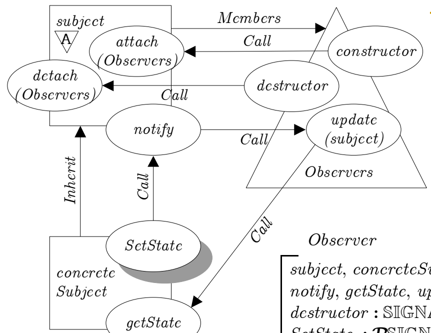
Composite

component, composite : CLASS
Leaves : ℘CLASS
CompositeOps, ComponentOps : ℘SIGNATURE

Abstract(component)
All(Method, ComponentOps⊗Leaves)
All(Method, CompositeOps⊗composite)
Inherit(composite,component)
Total(Inherit, Leaves, component)
Aggregate(composite,component)
Isomorphic(Forward, ComponentOps⊗composite, ComponentOps⊗component)

3

Observer pattern

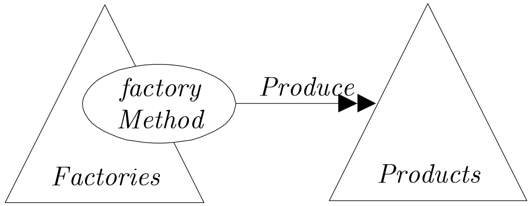


Observer

subject, concreteSubject : CLASS
notify, getState, update, attach, detach, constructor,
destructor : SIGNATURE
SetState : ℘SIGNATURE
Observers : HIERARCHY

Abstract(subject)
Inherit(concreteSubject,subject)
Total(Call, SetState⊗concreteSubject, notify⊗subject)
Total(Call, notify⊗subject, update⊗Observers)
Total(Call, update⊗Observers, getState⊗concreteSubject)
Total(Call, constructor⊗Observers, attach⊗subject)
Total(Call, destructor⊗Observers, detach⊗subject)
Members(subject,Observers)

Factory Method pattern



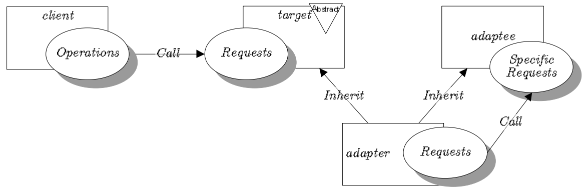
FactoryMethod

Factories, *Products* : HIERARCHY
factoryMethod : SIGNATURE

Isomorphic(*Produce*, *factoryMethod* $\otimes$ *Factories*, *Products*)

65

Adapter (Class) pattern



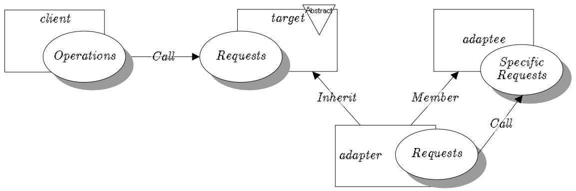
ClassAdapter

client, *target*, *adapter*, *adaptee* : CLASS
Operations, *Requests*, *SpecificRequests* : $\mathcal{P}$SIGNATURE

Abstract(*target*)
Total(*Call*, *Operations* $\otimes$ *client*, *Requests* $\otimes$ *target*)
Total(*Call*, *Requests* $\otimes$ *adapter*, *SpecificRequests* $\otimes$ *adaptee*)
Inherit(*adapter*,*target*)
Inherit(*adapter*,*adaptee*)

66

Adapter (Object) pattern



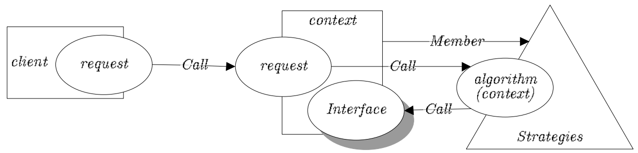
ObjectAdapter

client, *target*, *adapter*, *adaptee* : CLASS
Operations, *Requests*, *SpecificRequests* : $\mathcal{P}$(SIGNATURE)

Abstract(*target*)
Total(*Call*, *Operations* $\otimes$ *client*, *Requests* $\otimes$ *target*)
Total(*Call*, *Requests* $\otimes$ *adapter*, *SpecificRequests* $\otimes$ *adaptee*)
Inherit(*adapter*,*target*)
Member(*adapter*,*adaptee*)

67

Strategy pattern



Strategy

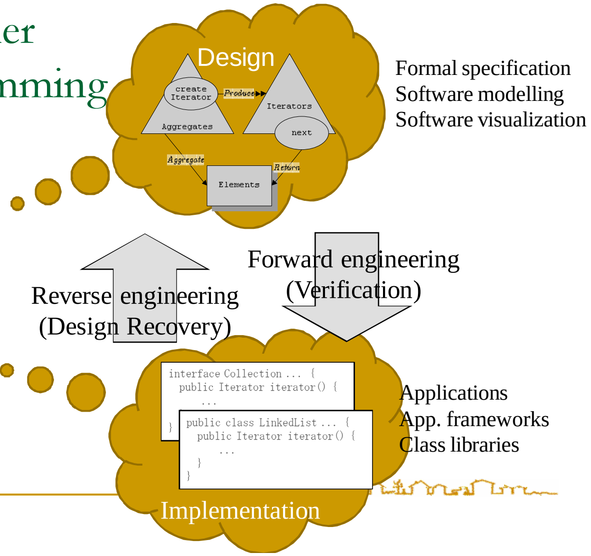
client, *context* : CLASS
request, *algorithm* : SIGNATURE
Interface : $\mathcal{P}$(SIGNATURE)
Strategies : HIERARCHY

Member(*context*, *Strategies*)
Call(*request* $\otimes$ *client*, *request* $\otimes$ *context*)
Call(*request* $\otimes$ *context*, *algorithm* $\otimes$ *Strategies*)
Total(*Call*, *algorithm* $\otimes$ *Strategies*, *Interface* $\otimes$ *context*)

Tool support in LePUS3 and Class-Z

69

Two-Tier Programming

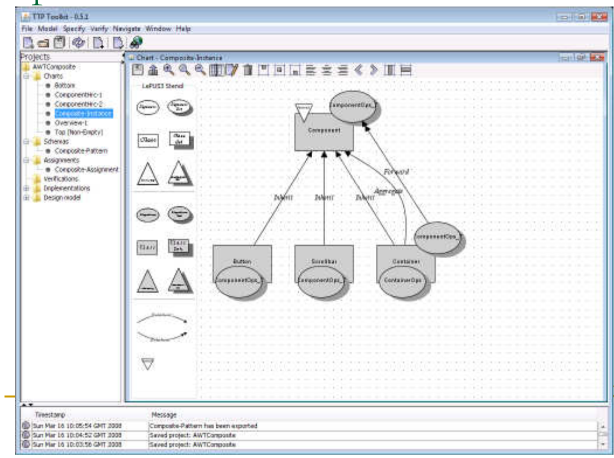


The Two-Tier Programming Toolkit

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

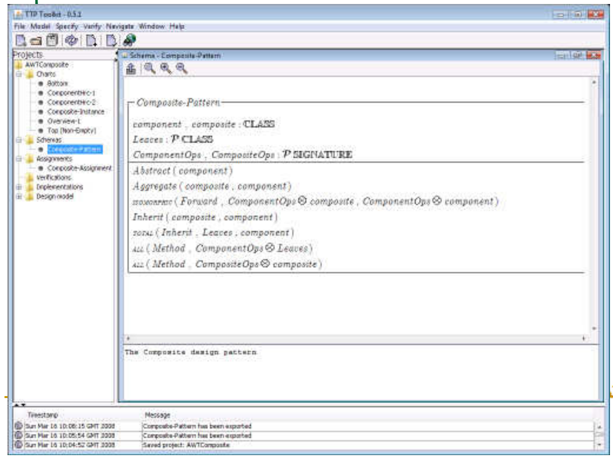
71

The TTP Toolkit (cont.): Specification in LePUS3



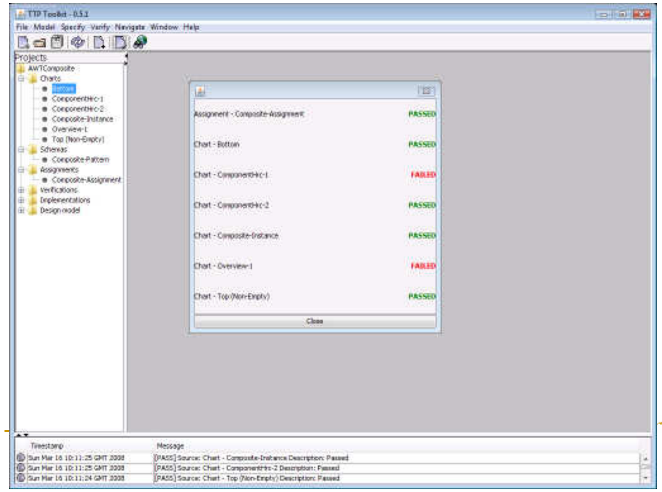
72

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



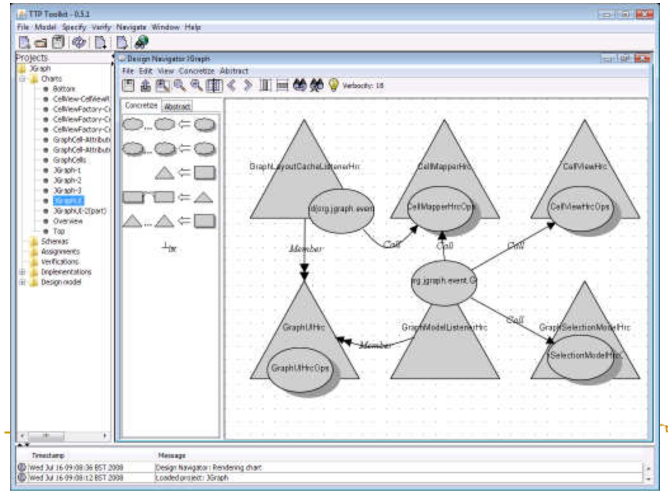
73

The TTP Toolkit (cont.): Verification



74

The TTP Toolkit (cont.): Visualization



75

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

76

Exercises (cont.)

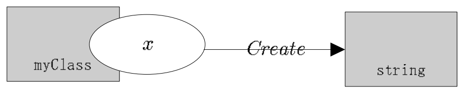
3. Compare the LePUS3/Class-Z Specification of each one of the design patterns to the description in Gamma et al. 1995
   - 3.1 What is the book saying that the chart/schema does not say?
   - 3.2 What is the chart/schema saying that the book does not say?
   - 3.3 What are the advantages of the book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

77

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`"

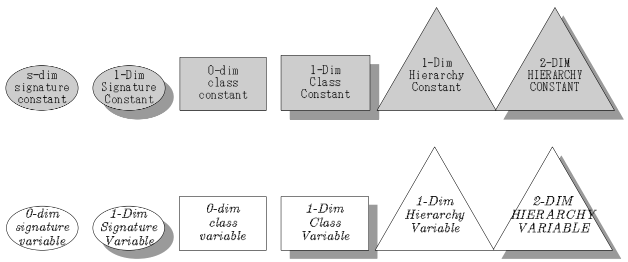
Answer:



78

LePUS3 quick reference

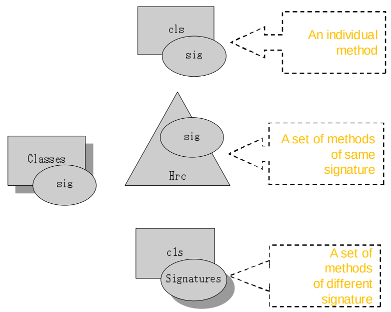
- constants and variables



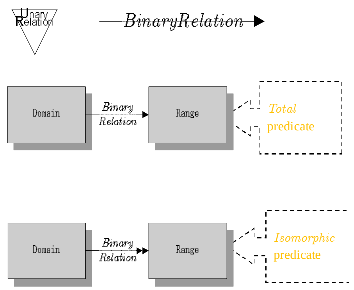
79

LePUS3 quick reference (cont.)

Methods



Relations and predicates



80

References

- Legend: Key to LePUS3 and Class-Z Symbols [.pdf] <http://www.lepus.org.uk/ref/legend/legend.xml>
- A.H. Eden, J. Nicholson. *Object-oriented modelling: Theory and Practice*. Unpublished manuscript (ask me for a copy)
- A.H. Eden, E. Gasparis, J. Nicholson. "LePUS3 and Class-Z Reference Manual". University of Essex, Tech. Rep. CSM-474, ISSN 1744-8050 (2007). <http://lepus.org.uk/ref/refman/refman.xml>
- A.H. Eden, E. Gasparis, J. Nicholson. "The 'Gang of Four' Companion: Formal specification of design patterns in LePUS3 and Class-Z." University of Essex, Tech. Rep. CSM-472, ISSN 1744-8050 (2007). <http://lepus.org.uk/ref/companion/index.xml>
- A.H. Eden. "Formal Specification of Object-Oriented Design." *Proc. Int'l Conf. Multidisciplinary Design in Engineering CSME-MDE 2001* (21–22 Nov. 2001), Montreal, Canada.

81