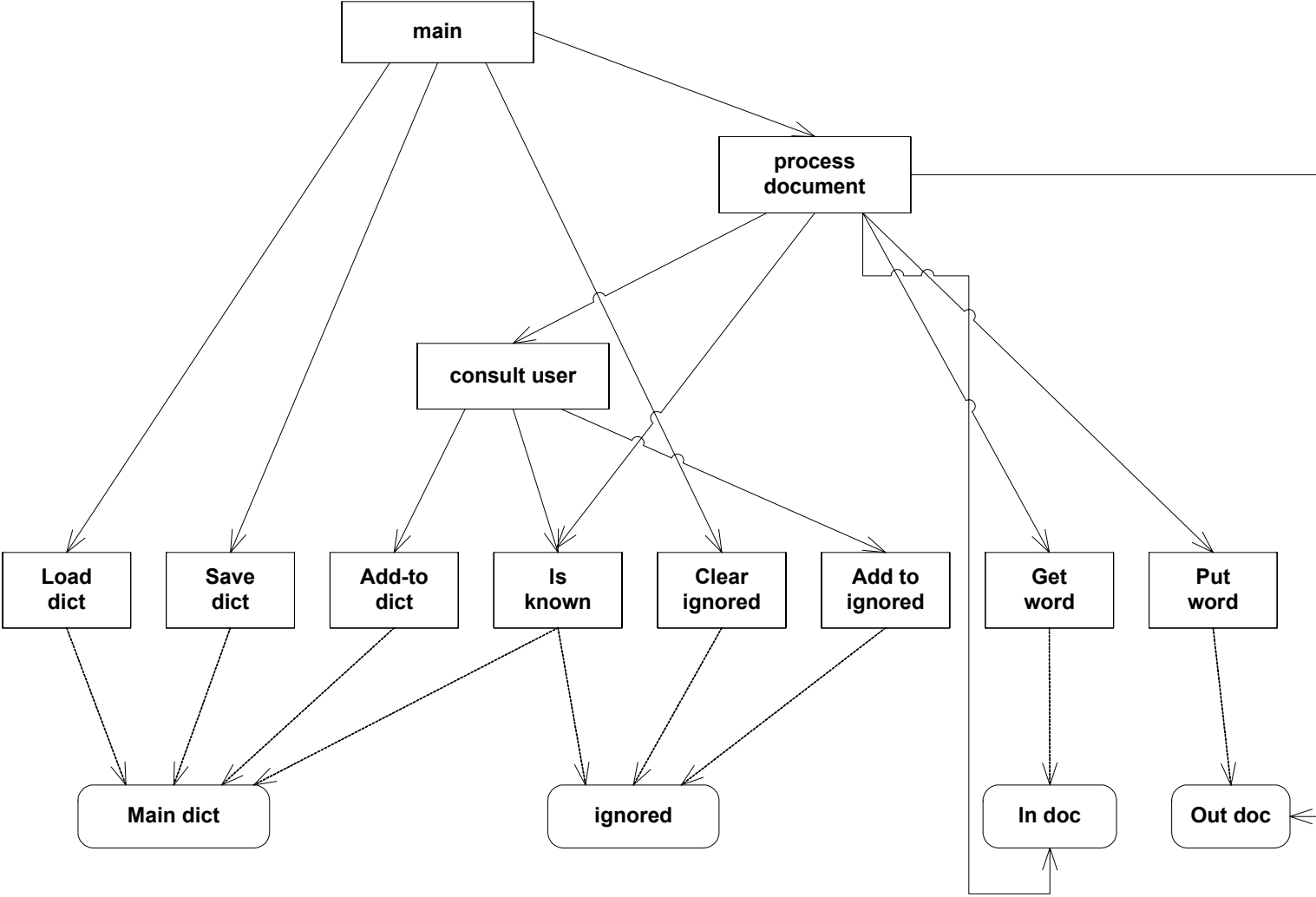


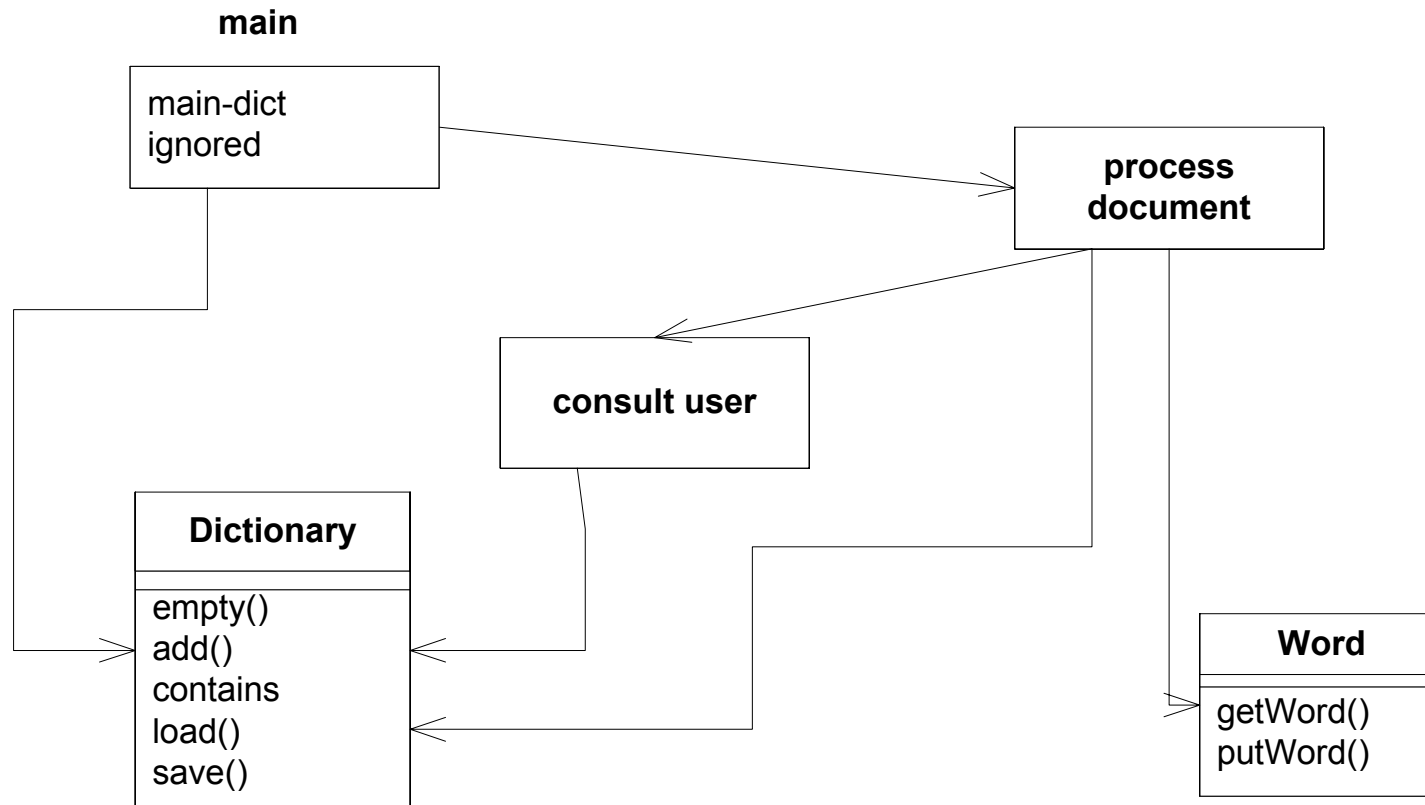
Paradigmes/perspectives

- Procedural/Imperative Programming
 - Sequences of operations on data
- Functional Programming
 - Mathematical functions
- Constraint-Oriented/Declarative (Logic) Programming
 - Equations
- Object-Oriented Programming
 - Physical modeling

Imperative/procedural spellchecker



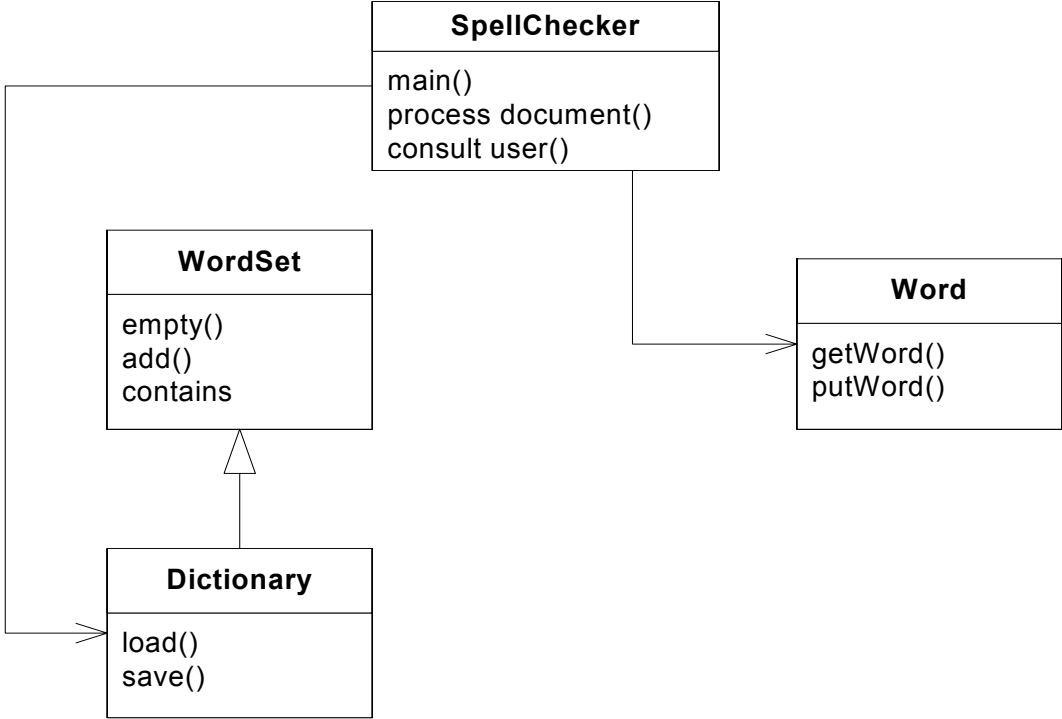
Procedural/functional-with-data-types spellchecker



Procedural
Functional

`add(Dictionary dict, Word w):`
`add: (word, dictionary) -> dictionary`

Object oriented spellchecker



```
aDictionary.add(Word w)
```

Prolog spellchecker

```
% add (D,W,D1) succeeds
% iff D1 is the dictionary obtained by adding W to D

add(empty,W,dict(empty,W,empty)).
add(dict(D1,W,Dr),W,dict(D1,W,DR)).
add(dict(D1,Wd,Dr),W,dict(D11,Wd,Dr)):-
    less(W,Wd),!,
    add(D1,W,D11).
add(dict(D1,W,Dr),W,dict(D1,Wd,Dr1)):-
    less(Wd,W),!,
    add(Dr,W,Dr1).
```

Object-oriented and/or functional programming

- War
- Use the right language for the right purpose (if possible), including combined use
- Learn from each other
 - Extend object oriented languages with funtions
 - Extend functional languages with object orientation

Software engineering/typing issue

Kim Bruce: Some Challenging Typing Issues
in Object-Oriented Languages

$$S \in \text{SExp} ::= n \mid - S$$
$$E \in \text{Exp} ::= n \mid - E \mid E + E$$

Functional solution I

$S \in \text{SExp} ::= n \mid - S$

```
datatype sexp = SConst of int | SNeg of sexp;
```

```
fun sinterp (SConst n) = n  
| sinterp (SNeg t) = ~(sinterp t);
```

Easy to add a new operation on expression:

```
fun sformatter (SConst n) = Int.toString(n)  
| sformatter (SNeg t) = "-" ^ (sformatter t);
```

Functional solution II

```
E ∈ Exp ::= n | - E | E + E
```

Not so easy to extend with + (i.e. new data structure):

```
datatype exp = Const of int | Neg of exp | Plus of exp*exp;
```

```
fun interp (Const n) = n  
  | interp (Neg t) = ~(interp t)  
  | interp (Plus t u) = (interp t) + (interp u);
```

```
fun formatter (Const n) = Int.toString(n)  
  | formatter (Neg t) = "-" ^ (formatter t)  
  | formatter (Plus t u) = (formatter t) ^ " + " ^  
    (formatter u);
```

Object-oriented solution I

$S \in \text{SExp} ::= n \mid - S$

```
public interface Form {int interp(); // Interpret formula}
```

```
public class ConstForm implements Form {  
    public int value; // value of constant  
    public ConstForm(int value) {this.value = value;}  
    public int interp() { return value; }  
}
```

```
class NegForm implements Form {  
    public Form base; // formula being negated  
    public NegForm(Form basep) {base = basep;}  
    public int interp() { return (- base.interp()); }  
}
```

Object-oriented solution II

$E \in \text{Exp} ::= n \mid - E \mid E + E$

Easy to extend with +:

```
public class PlusForm implements Form {
    public Form first, second;
    public PlusForm(Form firstp, Form secondp) {
        first = firstp;
        second = secondp;
    }
    public int interp() {
        return first.interp() + second.interp();
    }
}
```

Object-oriented solution III

$E \in \text{Exp} ::= n \mid - E \mid E + E$

Not so easy to extend with new operation:

```
public interface FForm extends Form {
    String formatter();
}
public class FConstForm extends ConstForm implements FForm
{
    public FConstForm(int value) {super(value);}
    public String formatter() {
        return "" + value;
    }
}
```

Object-oriented solution IV

```
class FNegForm extends NegForm implements FForm { ... }

public class FPlusForm extends PlusForm implements FForm {
    public FPlusForm(FForm firstp, FForm secondp) {
        super(firstp, secondp);
    }
    public String formatter() {
        return "(" + ((FForm)first).formatter() + " + " +
            ((FForm)second).formatter() + ")";
    }
}
```

Object oriented languages with functions

<http://jakarta.apache.org/commons/sandbox/funcutor/>

- Functions as parameters, closures

Expression specialization I

```
class Item {  
    private String name;  
    private int price;  
  
    public int getPrice() {return price;}  
  
    public void setPrice(int inPrice){price = inPrice;}  
}
```

```
assert(item1.getPrice() >= item2.getPrice());
```

Expression specialization II

```
public class PriceComparator implements Comparator
{
    public int compare (Object o1, Object o2) {
        return (((Item)o1).getPrice() - ((Item)o2).getPrice());
    }
}
```

```
Comparator pc = new PriceComparator();
BinaryPredicate bp = new IsGreaterThanOrEqualTo(pc);
Item item1 = new Item(); item1.setPrice(100);
Item item2 = new Item(); item2.setPrice(99);
```

```
if (bp.test(item1, item2)) ... else ...;
```

```
Item item3 = new Item(); item3.setPrice(101);
```

```
if (bp.test(item1, item3)) ...else ...;
```

```
class IsGreaterThanOrEqualTo implements BinaryPredicate{
    public boolean test(Object left, Object right) {
        return comparator.compare(left,right) >= 0;
    }
}
```

```
public interface BinaryPredicate {
    boolean test(Object left, Object right);
}
```

Expression composition

```
class Multiply implements BinaryFunction
{
    public Object evaluate(Object left, Object right) {
        return new Double(((Double)left).doubleValue() *
            ((Double)right).doubleValue());    }
}

double discountRate = 0.1;    double taxRate=0.33;
Item item = new Item();    item.setPrice(100);
UnaryFunction calcDiscountedPrice =
    new RightBoundFunction(new Multiply(),
        new Double(1-discountRate));
UnaryFunction calcTax =
    new RightBoundFunction(new Multiply(),
        new Double(1+taxRate));
CompositeUnaryFunction calcNetPrice =
    new CompositeUnaryFunction(calcTax, calcDiscountedPrice);
Double netPrice = (Double)calcNetPrice.evaluate(
    new Double(item.getPrice()));
```

```
class RightBoundFunction implements UnaryFunction{
    public RightBoundFunction(
        BinaryFunction function, Object arg) {
        this.function = function;
        this.param = arg;
    }

    public Object evaluate(Object obj) {
        return function.evaluate(obj,param);
    }

    private BinaryFunction function = null;
    private Object param = null;
}
```

Another closure example

```
class Dictionary {  
    Word translate(Word w) {return new Word();}  
}  
class Translate{  
    Dictionary d;  
    Translate(Dictionary ad) {d = ad;};  
    Word translate(Word w) {return d.translate(w); }  
}
```

```
...  
Word w;  
Dictionary englishNorwegianDict = new Dictionary();  
Translate englishToNorwegian =  
    new Translate(englishNorwegianDict);  
...  
englishToNorwegian.translate(w);
```

Both classes and functions

- Beta pattern



```
<name>:<super> (#  
    enter <attributes>  
    do <local patterns>  
    exit <expr list>  
#)
```

record: Point: (# x, y : @ Integer #)

function: Sine: (# x, result: @ Real;
 enter x
 do { compute sine of x }
 exit result
#)

*class with
method:* Point: (# x, y: @ Integer;
 Move: (# dx, dy: @ Integer
 enter (dx, dy)
 do x + dx ® x;
 y + dy ® y
 #);
 ...
#)

*class with class, virtual pattern (i.e. virtual methods and
virtual classes),...,
pattern attributes (i.e. function and class variables)*

Functional languages with object orientation?

- Should ML be object-Oriented?
- Encapsulation
 - No, already part of ML
- Inheritance
 - Class and inheritance should not be added, because of problems with method redefinition and open recursion (methods calling other methods)
- Subtype polymorphism
 - Yes, but only record subtyping
- Dynamic dispatch
 - No, already part of ML
- OO has no basis (or many)

Physical Modeling

- Modeling approach
 - is based upon the conception of the application domain in terms of phenomena and concepts.
- A physical model
 - consists of elements (physical material) that represent phenomena and concepts from the application domain.

phenomena	modelled by	objects
concepts	modelled by	classes

Material

- Paper, plastic , wood, Lego bricks, etc. are examples of (physical) material used to construct (physical) models
- Objects are the computerized material used to construct computer based physical models

Aspects of phenomena

- Substance
 - Characterized by identity, volume and a unique location in time and space
- Measurable properties of substance
 - Properties of substance can be measured
 - The results of measurements can be compared and described by values of types
- Transformations on substance
 - Partial ordered sequence of actions changing the (measurable properties of) some substance

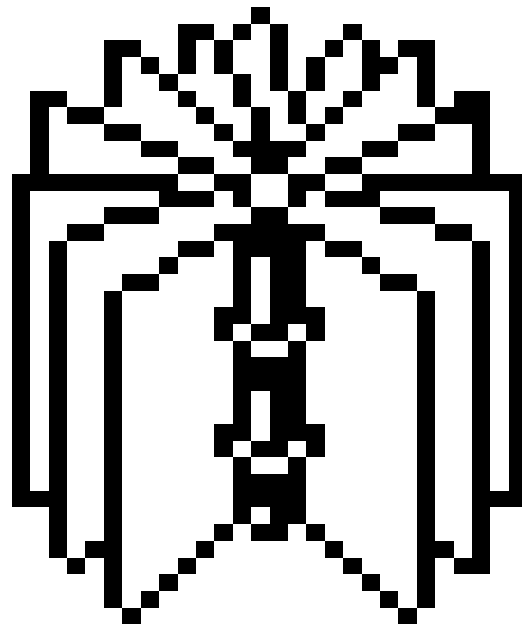
Aspects of concepts

- Designation
 - The names by which the concept is known
- Intension
 - Properties that characterize the phenomena in the extension of the concept
- Extension
 - The phenomena covered by the concept

Object Orientation as Physical Modelling

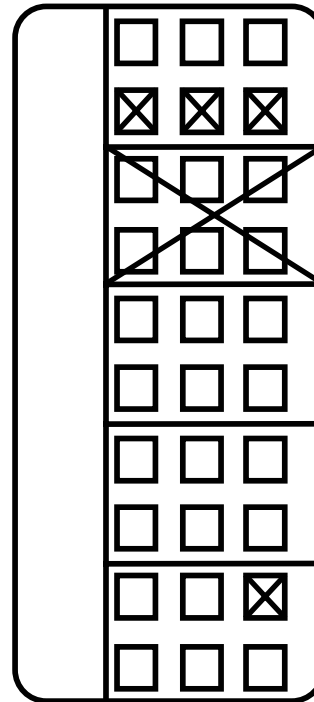
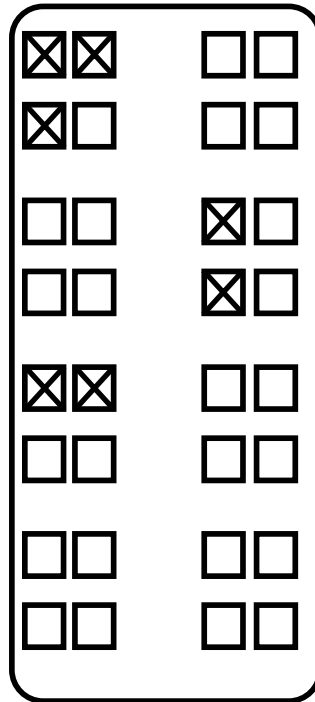
- Phenomena represented by **objects**.
- Measurable properties represented by **states** of objects.
- Transformations of state represented by **action sequences** performed by objects.
- Concepts represented by **classes** that define the common properties of categories of objects, and classes are organized in **concept hierarchies**.

Norwegian Train Example



Carriage no. 16

Carriage no. 17



OOPSLA

- Microsoft: Software Factories
 - UML not suited, too general – not handling .NET properties
 - Domain Specific Languages
- Challenges
 - Language maintenance
 - What about more than one domain?
 - What about domain languages sharing semantics?