

Advanced OO Design WS03/04

An Introduction to Smalltalk

There's More to OO than Java...

Dipl.-Ing. Michael Haupt

Alexanderstraße 10, Room 228

haupt@informatik.tu-darmstadt.de

www.st.informatik.tu-darmstadt.de

Agenda

- Why Smalltalk?
- History outline
- Introduction to the language
- Error handling, dependency

- ubiquitous demos
 - programming environment
 - tools: browser, inspector, debugger

Why Smalltalk?

- "When I invented the term 'object-oriented' I did not have C++ in mind." -- Alan Kay
- *Pure* object-oriented language
 - learn *concepts*, not syntax: very easy to learn
 - GC, no memory management, no pointers
- Look at and manipulate objects in real time
- High productivity by short round-trip times
- Written in itself, so better understandable

Smalltalk's History

- 1968: SIMULA
 - first "object-oriented" language

- 1971: Dynabook (Alan Kay)
 - easy-to-use computer for anyone
 - bitmapped display, pointing device, external storage
 - programming ~ creating simulations
 - Smalltalk-71

Smalltalk's History

- 1973: Xerox Alto computer
 - running Smalltalk (implemented in BASIC)
 - iconic characters, turtle graphics (LOGO)
 - classes (no hierarchies), instances, **self**
- 1974
 - performance improvements
 - first metaobjects
 - Alto: virtual memory (96 kB RAM, 2.4 MB hard disk)

Smalltalk's History

- 1976
 - Smalltalk bytecode interpreter in Alto microcode
 - everything is an object
 - class hierarchy, super
 - code browser, inspector, debugger

- 1978: NoteTaker
 - step towards Dynabook: portable computer
 - cancelled: "No one wants portability."

Smalltalk's History

- Smalltalk-80
 - 2 goals: hardware and community
 - language standard
 - massively adopts MVC
 - forerunner of today's GUIs

- Meta-circular implementation
 - compiler/interpreter implemented in Smalltalk
 - source code is always there
 - virtual machine vs. virtual image

Smalltalk Environment

- Environment

- virtual machine (usually with JIT)
- image = snapshot
- export applications as image, deliver with VM binary

- Benefits

- environment is IDE and run-time all the same
- powerful debugging facilities
- on-the-fly implementation modification

Smalltalk – the Language

- "Pure" object-oriented language
 - absolutely everything is an object
- Features
 - single inheritance
 - dynamically typed
 - powerful library and meta-level
 - strict class hierarchy (superclass: **Object**)

Smalltalk – the Language

- Lexical details

- comments: double quotes
- characters: `$<char>`
- strings: single quotes
- symbols: `#<sym>`
- arrays: `#(<elem> <elem> ...)`

- Operators

- assignment `:=`
- equality `=`
- identity `==`

```
"this is a comment"  
$x  
'this is a string'  
#thisIsASymbol  
#('this' 'is' 'an' 'array' 'with' 7 'elements')
```

Keywords and Messages

- Keywords are pervasive

C++ / Java:

```
Transformation t;
float a;
Vector v;
t->rotate(a,v); // for C++
t.rotate(a,v); // for Java
```

Smalltalk:

```
| t a v |
```

```
| aTransformation angle aVector |
```

~~t . rotate (a , v) ;~~

~~t rotate by: a around: v;~~

t rotateBy: a around: v

This is Smalltalk!

Keywords and Messages

- Rotation method source code

```
rotateBy: angle around: aVector  
| result |  
result := do some computations.  
^result
```

- Alternatives:

- **rotateAround: aVector by: angle**
- **rotate: angle and: aVector** (bad style)
- keywords tell the order of parameters

Keywords and Messages

- Methods (messages)

- general structure

```
keyword1: param1 keyword2: param2 ...  
| local variables |  
instructions
```

- instructions end with . (except for last)
- immediate return with **^<value>**
- message's selector is **keyword1:keyword2:...**

Keywords and Messages

- Unary messages

- no parameters
- for example: `someObject clone`
- definition:

```
messageName  
| local variables |  
instructions
```

- Binary operators

- there are no unary operators!
- keywords: special characters like `+` `-` `*` `/` `,`
- exceptions: e.g., `.` `;`

Keywords and Messages

- Message sending semantics
 - messages are sent to objects
 - first element in an expression is always an object
 - result of an expression is an object, too
 - evaluation: left to right
- No implicit sending to **self**
 - Java: `myMethod();`
`this.myMethod();`
 - Smalltalk: `self myMethod`

Keywords and Messages

- **Arithmetics**

- numbers are objects, even literals!
- expression `4 sqrt` results in 2.0
- expression `1 + 2 * 3` results in 9 (yes, indeed)
- expression `1 + (2 * 3)` results in 7

Keywords and Messages

■ Cascading

- send multiple messages to a single object

- instead of

```
| p |  
p := Client new.  
p name: 'Jack'.  
p age: 32.  
p address: 'Earth'
```

- use

```
| p |  
p := Client new.  
p name: 'Jack'; age: 32; address: 'Earth'
```

- or even

```
| p |  
p := Client new name: 'Jack'; age: '32'; address: 'Earth'
```

Conditions

- Conditions

- there are actually no control structures
- all "control structures" are messages

```
if(a < 0 || (b >= c && b <= d))  
  a = -c;
```

```
a negative | (b between: c and: d) ifTrue: [ a := c negated ]
```

- Messages that can be sent to **Boolean** objects
 - **ifTrue:**, **ifFalse:**
 - **ifTrue:ifFalse:**, **ifFalse:ifTrue:**
 - conditional functionality encapsulated in blocks

Conditions

- How are conditionals implemented?
 - class **Boolean** is abstract
 - classes **True** and **False** inherit from **Boolean**
 - each has one instance: **true** resp. **false**
 - implementation of **ifTrue:** in class **True**

```
ifTrue: alternativeBlock  
  ^alternativeBlock value
```

- implementation of **ifFalse:** in class **True**

```
ifFalse: alternativeBlock  
  ^nil
```

Blocks

■ Blocks

- nameless functions (closures, lambda expressions)

```
a = int f(x) { return x + 1; }
```

```
a := [ :x | x + 1 ]
```

- evaluate blocks by sending them **value** messages

```
b := a value: 41
```

b will contain 42 after that

- possible: **value**, **value:**, **value:value:**, ...
- return value is value of last expression in block

■ Scope and returning

- in a block, **self** is bound to the surrounding object
- **^** returns from the defining method

Blocks

- A closer look at returning
 - `^` returns from the defining method (context)
 - the following will not work

```
getBlock
  ^ [ ^42 ]
```

```
useBlock
  self getBlock value
```

- Does it make sense?
 - blocks are frequently passed to methods
 - returns should not terminate the target

```
divide: a by: b
  b = 0 ifTrue: [ ^'error' ] ifFalse: [ ^a/b ]
```

- not **ifXXX**: should be terminated, but **divide:by:**

Logical Operations

- Operators implemented in **Boolean**
 - **&**, **|** - eager evaluation (**Boolean** argument)
 - **and:**, **or:** - lazy evaluation (block argument)
 - **xor:**, **eqv:** - **Boolean** argument
 - **not**

```
| x |
x := nil.
x notNil & (x > 0) ifFalse: [ Smalltalk beep ]
```

This gets executed in any case but `nil` does not understand the `>` message!

```
| x |
x := nil.
(x notNil and: [ x > 0 ]) ifFalse: [ Smalltalk beep ]
```

Comparing

- Messages implemented in Object
 - `isNil`, `notNil`
 - `=`, `~=` - equal, not equal
 - `==`, `~~` - exactly equal, not exactly equal
- A word about exact equality (`==`, `~~`)
 - exact equality: same object pointer
 - *do not overload!*
- Other examples for comparing messages
 - `>`, `<`, `even`, `odd`, ...

Loops

- Counting loops

- do something 100 times

```
100 timesRepeat: [ Transcript show: 'Hello world.'; cr ]
```

- advanced "for" loops

```
1 to: 100 do: [ :i | Transcript show: i; cr ]
```

```
1 to: 100 by: 10 do: [ :i | Transcript show: i; cr ]
```

```
100 to: 1 by: -1 do: [ ... ]
```

```
0.5 to: 7.3 by: 1.1 do: [ ... ]
```

Loops

- While loops
 - blocks can return **Boolean** values
 - messages **whileTrue:** and **whileFalse:**

```
| a |  
a := 100 atRandom.  
[ a = 42 ] whileFalse: [ a := 100 atRandom ]
```

Classes and Objects

- Classes and objects
 - general superclass: **Object**
 - there are class and instance variables and methods
- Instantiation
 - "constructors" are class methods, e.g., **new**
 - expressive "constructor" names
 - possible instantiation method for a **Person** class

```
withName: name age: age address: address
^(self new name: name; age: age; address: address)
```
 - usage:

```
| p |
p := Person withName: 'John' age: 32 address: 'Earth'
```

Classes and Objects

- **Visibility**

- all instance variables are private
- all messages are public
- instances of the same class cannot access members

- **Instance variable access**

- convention: define two messages of the same name
- instance variable **age**
- getter message **age**
- setter message **age:**

```
age  
^age
```

```
age: newAge  
age := newAge
```

Error Handling

- Dynamic typing caveat: unknown messages
 - in `Object`: message `doesNotUnderstand:`
 - invoked implicitly upon illegal message sends
 - override to deal with it

`doesNotUnderstand: aMessage`

Transcript show: 'Do not send me '; show: aMessage selector

- standard behaviour: invoke debugger

Error Handling

- Exceptions

```
[ x / y ] on: ZeroDivide do: [ :ex | ex resume: 1 ]
```

- upon a division by zero, return 1

- Signal exceptions

- superclass of all exceptions: **Exception**
- program errors: **Error**
- notifications that can be ignored: **Notification**
- warnings the user should be aware of: **Warning**
- signal e.g., like this

```
Error signal: 'Something really bad happened.'
```

Error Handling

- Return values

```
[ Error signal ] on: Error do: [ : ex | someOp. 42 ]
```

- this returns 42 if an error occurs
- return value of **on:do:**

- Resuming

- resumable exceptions (**Notification, Warning**)

```
[ Warning signal ] on: Warning do: [ :ex | ex resume: 42 ]
```

- returns 42, but where the **Warning** was signalled

Error Handling

- Exiting handler blocks
 - explicit exit: two semantically identical ways

```
[ Error signal ] on: Error do: [ :ex | 23 ]
```

```
[ Error signal ] on: Error do: [ :ex | ex exit: 23 ]
```

- Retrying
 - adjust erroneous values

```
[ ^ x / y ] on: ZeroDivide do: [ :ex | y := 0.000001. ex retry ]
```

- use a fallback

```
[ doSomethingUnsafely ] on: Error do:  
[ :ex | ex retryUsing: [ doItSafely ] ]
```

Error Handling

- Cleaning up

- place a lock on an external file
- locks must be released even if exceptions occur

```
doLocked: aBlock  
self lock.  
^aBlock ensure: [ self unlock ]
```

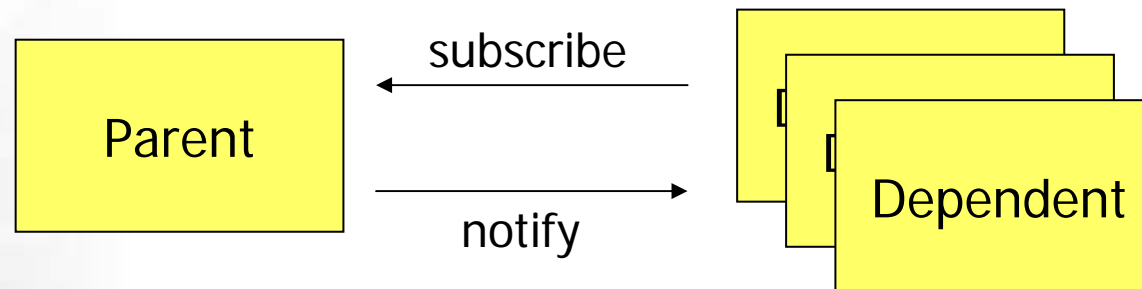
- Conditionally cleaning up

- perform cleaning up only if an exception occurred

```
someBlock ifCurtailed: [ cleanup ]
```

Dependency

- Observer design pattern
 - clients subscribe for notifications on subject updates
 - Smalltalk: clients = dependents, subject = parent



- Dependency support
 - standard, implemented in **Object**
 - subjects (i.e., all objects) implement notification

Dependency

- Notification messages
 - notify unspecified: **changed**
 - notify of specific update: **changed:**
- Notification for **age** instance variable
 - method **age:** in class **Person**

```
age: newAge  
| old |  
old := self age.  
age := newAge.  
self changed:  
  (Dictionary new  
   at: #parent put: self;  
   at: #member put: #age;  
   at: #oldValue put: old;  
   yourself)
```

Dependency

- Becoming a dependent

- send **addDependent:** to the parent

```
somePerson addDependent: self
```

- implement **update** or **update:**

```
update: param
```

```
Transcript show: 'Person was updated.'; show param; cr
```

- Revoking dependency

- send **removeDependent:** to the parent

```
somePerson removeDependent: self
```

Smalltalk Environment

- Smalltalk implementations (free)
 - Squeak (used here, open source)
<http://www.squeak.org/>
 - Smalltalk/X: free for non-commercial use
<http://www.exept.de/>
 - Cincom Smalltalk: free for non-commercial use
<http://www.parcplace.com/>
 - Strongtalk (typed Smalltalk)
[http://www.cs.ucsb.edu/projects/strongtalk/
pages/index.html](http://www.cs.ucsb.edu/projects/strongtalk/pages/index.html)