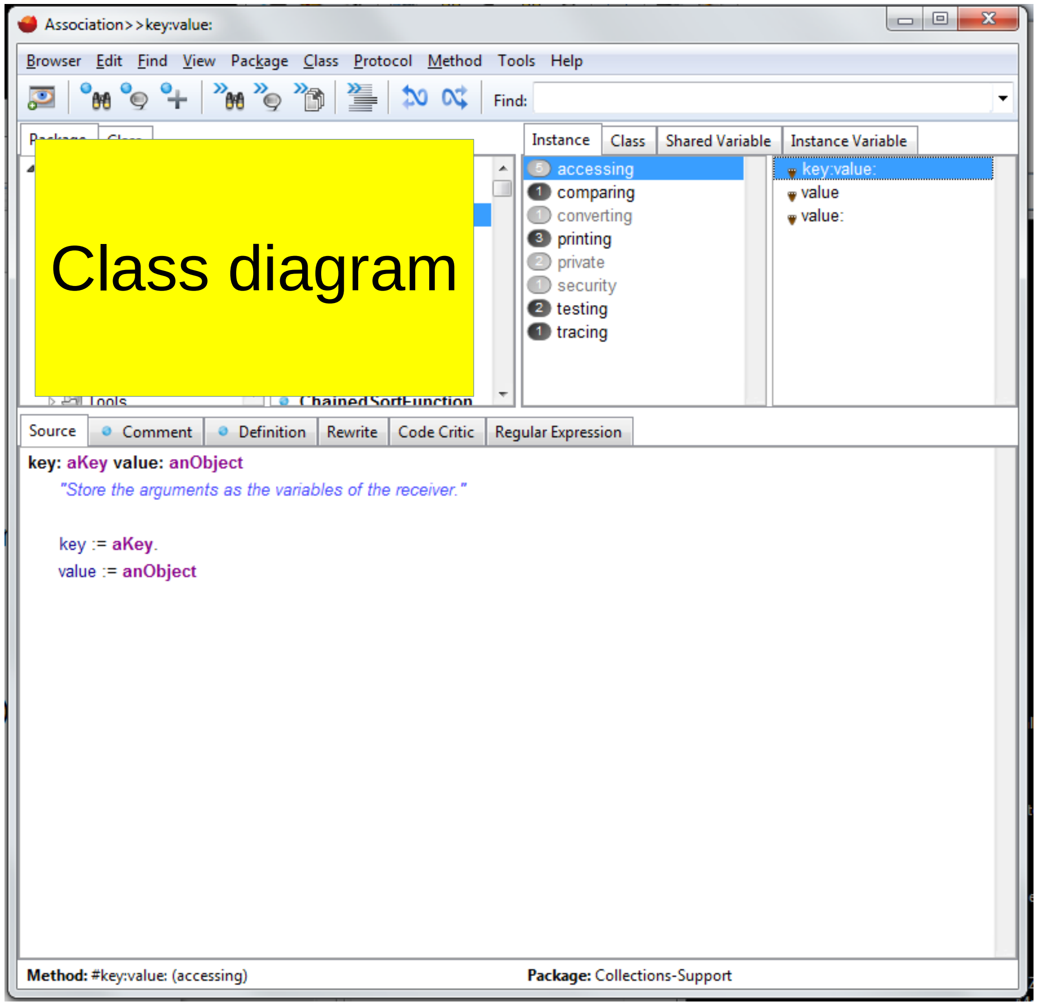
Rolemodeling as a graphic extension of the Smalltalk IDE

Peter Werner, peter_eh.werner@t-online.de

Introduction & Motivation

- Hobby, Experience
- Based on the ideas of Trygve Reenskaug
- 35 years dilemma!
  many 1000 lines of code = a big labyrinth
  when a developer wants to change anything (old/no doc)
- 35 years only strings for code!
  But some times a graphic can say more then 1000 words!

Example


Association>>key:value:
Browser Edit Find View Package Class Protocol Method Tools Help
Find:
Instance Class Shared Variable Instance Variable
Class diagram
accessing
comparing
converting
printing
private
security
testing
tracing
key:value:
value
value:
Tools
ChainedSortFunction
Source Comment Definition Rewrite Code Critic Regular Expression
Method: #key:value: (accessing)
Package: Collections-Support

```
key: aKey value: anObject
	"Store the arguments as the variables of the receiver."

	key := aKey.
	value := anObject
```

A class diagram in a system browser which represents always the current code –> that is possible!

Oh – it can be helpful !

But:

- it is to big – scroll, scroll …
  To many classes !!!
  To many details !!!

- Many people have tried to intruduce graphic elements into software development with a small success – why?

==> we need a smaller graphic?

The feeling of a Role

- E.g. take an actor (an **Object**) in a theater
- He can play 1 or more **Roles**
- He can be exchanged by another one without any influence of the played story
- The story is based on a set of **Roles** and **Relations** between them
- The story gives a statement (parable or **Pattern**) form the author

==> People and animals learn by playing **Roles** and they think in **Roles and Pattern** intuitive (by default).

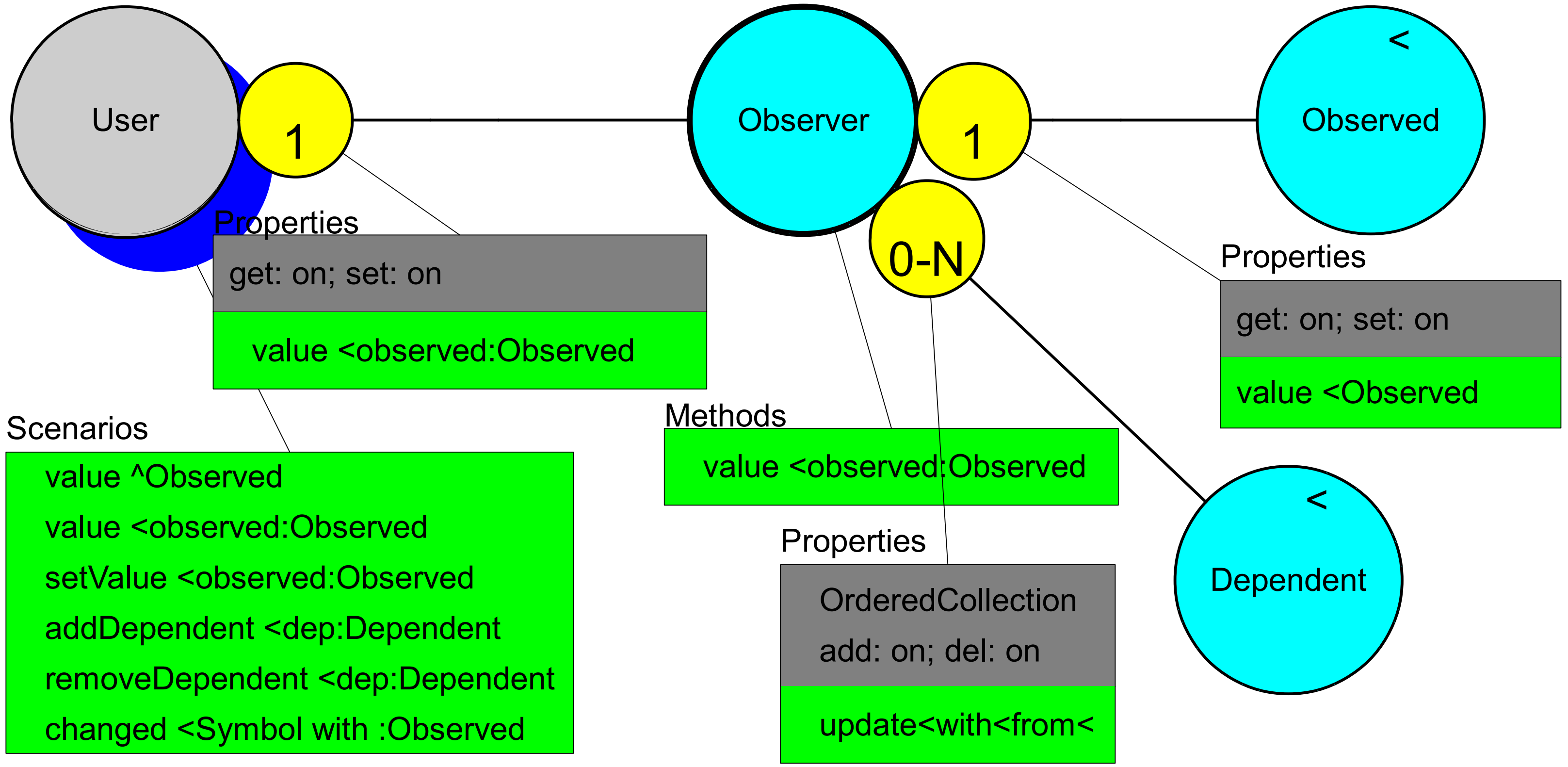
==> new point of view on Objects

Definition of Role and Rolemodel

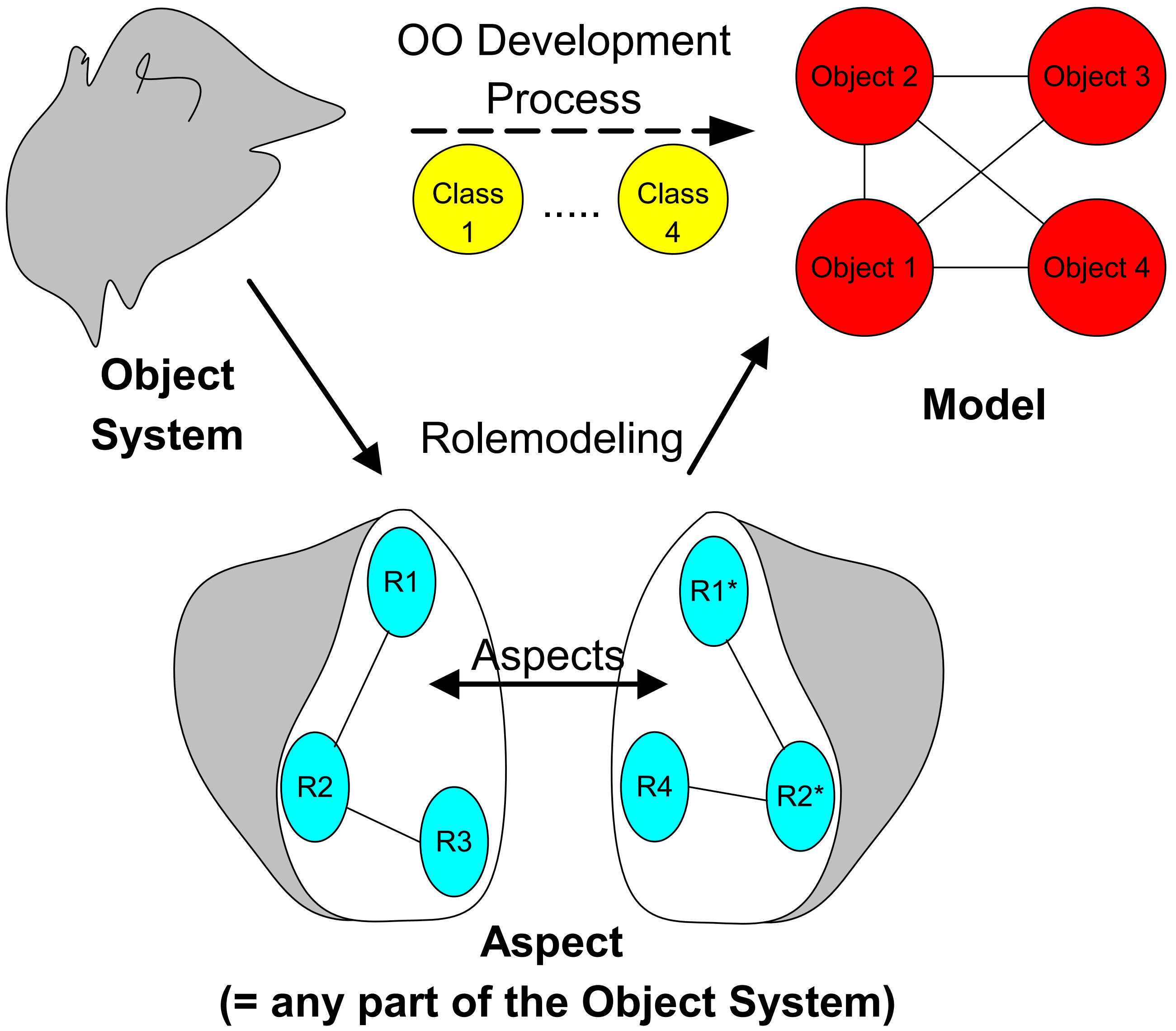
- A **Role** describes a part of an **Object** of the real world (**Object System**)
  - Similar to a class under multiple inheritance
- A **Rolemodel** describes a part (story/**Pattern**) of the real world from a given point of view, called **Aspect,** by:
  - A small fixed set of **Roles** and
  - **Relations** between the **Roles**
- **Roles** have properties e.g.:
  - **modeled** (which is to implement) or
  - **external**, to express how the the Model is to use (incoming, outgoing). The set of **external Roles** is called the Environment of the Model.

Example: Role Relation Diagram

(a view on a Rolemodel; is garphic oriented code)



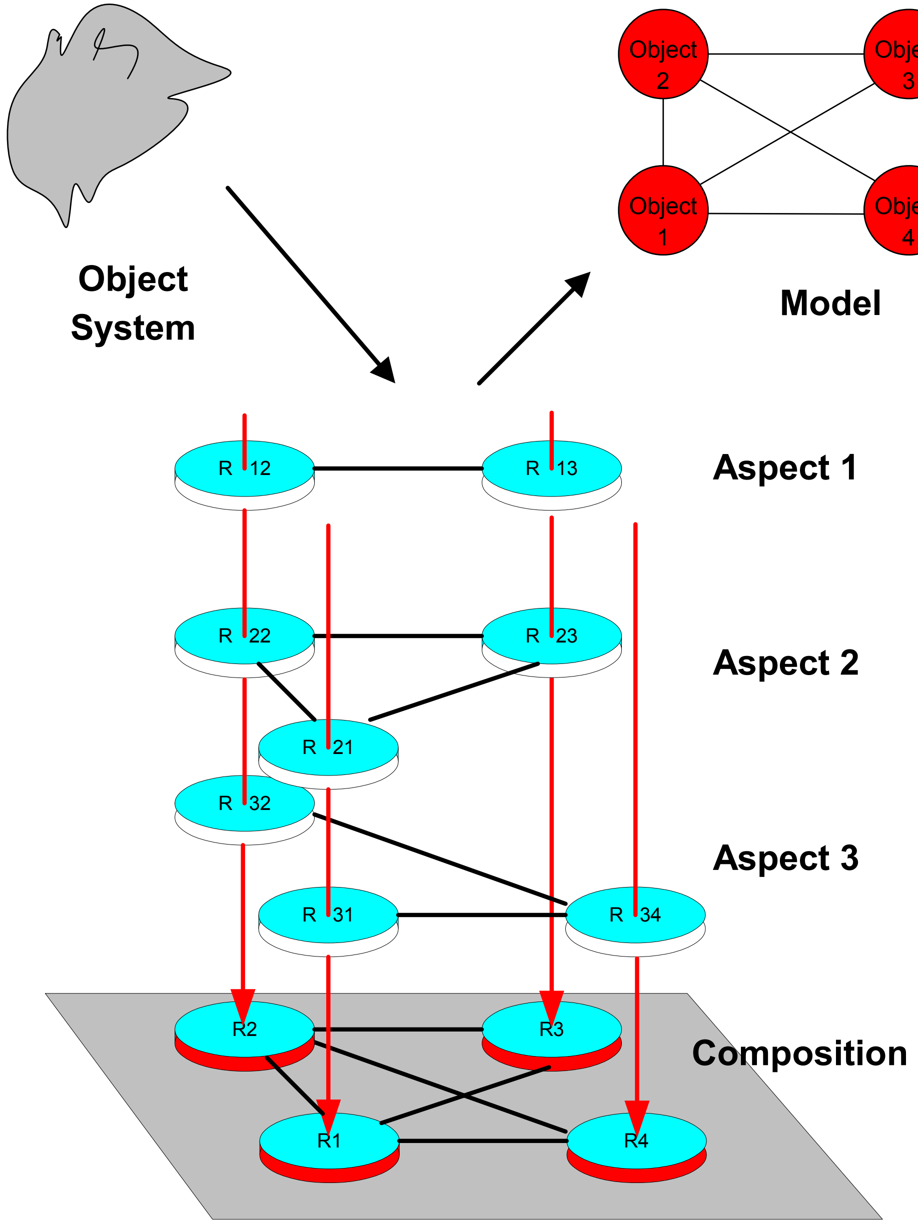
The development process



- isolated class and method descriptions
- comments/doc ?
- collecting variables and methods per class

- detect **Aspects**
- documents every **Aspect** explicit via a **Rolemodel**,
- so that a **Rolmodel** has small number of collaborating **Roles** (blue)
- reuse Rolemodels!

Composition



- define a simple **Rolemodel** RM or
- compose it by others:
  Composition(RMi) =$_{\text{def.}}$
  modify&generalize(
  unify $_{i=1...n}$(
  derive&specialize(
  RMi)))
- simplify it for high reuse, hide/unhide details and already completed parts
- plugged = derive + specialize + plugg it
- add **Role Changes** = message call

Role Work Shop (RWS)

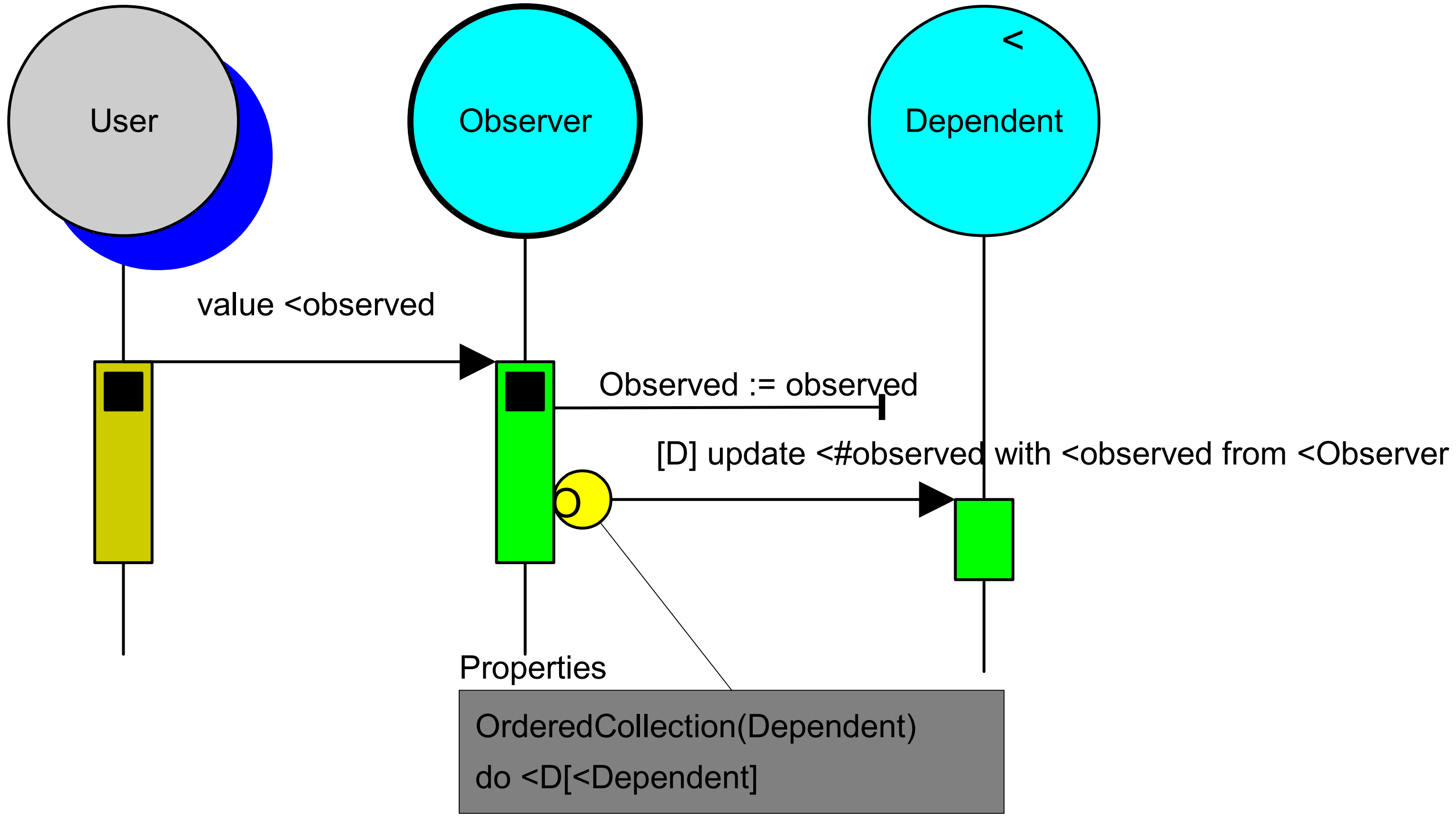
Brings a ‘*common language*’ for all participants of the development process (in analysis, design, partly implementation) by:

- extending the Smalltalk IDE via some browsers
- which allow to model the top level things of an Application by 4 different **Diagrams** (= real graphic statements):
  - **Role Relation** (bit similar to a class diagram)
  - **Scenario** (bit similar to message flow diagram)
  - **Model Composition** (new)
  - **Design Class** (new, complete the Model, can ‘*generate*’ complete **Control Code**, strongly separated from *)
- Diagrams have slots for text code snippets
- **Detail Code*** is always written as text (outside the **Rolemodels)**

Example: Scenario
(a view on a Rolmodel)

<	input parameter (readOnly)
:	input parameter like Smallatlk
^	output parameter

SC: value <observed:Observed


User
Observer
<
Dependent
value <observed
Observed := observed
[D] update <#observed with <observed from <Observer
Properties
OrderedCollection(Dependent)
do <D[<Dependent]