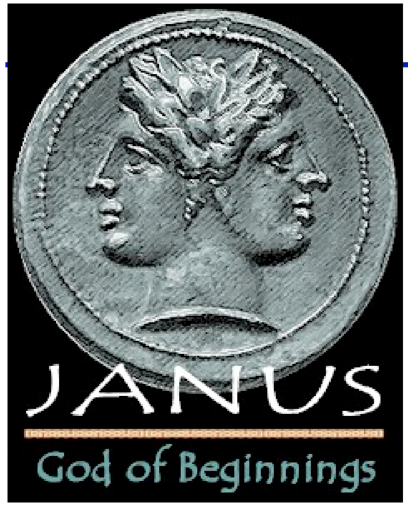


From Objects

to Components

to Services

Formal models for service-oriented interfaces and layered architectures

Manfred Broy

Technische Universität München
Institut für Informatik
D-80290 München, Germany

A brief look back on Marktoberdorf Summer Schools

- Structured programming (E.W. Dijkstra, C.A.R. Hoare, O.-J.Dahl)
- Data abstraction (C.A.R. Hoare)
- Weakest preconditions (E.W. Dijkstra)
- Program Transformation (F.L. Bauer)
- Abstract Data Types (J. Guttag)
- Denotational Semantics (D. Scott/J. Stoy)
- Logic Programming (J.A. Robinson)
- Temporal Logics (Manna/Pnueli)
- Concurrency (CCS: R. Milner/CSP: C.A.R. Hoare)
- Parallel Programming (J. Misra, M. Chandy)
- Process Algebras (J. Bergstra)
- Interactive Program Verification (J. Moore, L. Paulson)

What we face: characteristics of systems today

External view:

- Multi-functional, dialogue driven user interfaces
- High degree of interaction with the environment:
  - many user interfaces,
  - many peripheral devices,
  - many links to neighboured systems
- Communication and interaction
- Distributed on networks
- Concurrent
- Real time dependent

Internal view and structuring:

- Multi-layer architectures
- Hierarchical decomposition
- Flexible hardware/software partition

What we need: musts for software and systems engineering

System models that support

- Modularity
- Time
- Concurrency
- User interface specification
  - ◇ Property oriented
  - ◇ Functional (de)combination
- Component interface specification
  - ◇ Behavioural interface specifications
  - ◇ Nonfunctional properties (resources, QoS)
- Hierarchical composition/decomposition
- Abstraction layers and layered architectures
- Assumption/commitment (rely/guarantee) specs
- Export/import specifications

What does the "summer school" contribute

- So-called "Formal Methods"
  - Why "formal"
  - Why "methods"
- The significance of models
  - Understanding software & systems needs abstractions
  - Models are abstractions
- The evaluation of engineering theories
  - Clean (consistent) theory
  - Powerful enough and all the needed properties
  - Scales up
  - Tool support possible
  - Easy to comprehend
  - Applicable in an engineering context
  - Can be integrated into existing processes

Overview: Agenda for my presentation

- What we face: characteristics of systems today
- What we aim at: views and models
- What we need: musts for software and systems engineering
- What we miss: why objects are not enough
- What we suggest: Interfaces, components, and services
- Specifying services
- Combining services
- Service hierarchies
- Composition of services
- Service layers and layered architectures
- Specification of layers and layered architectures
- Summary and outlook

How to specify and verify systems

- Black Box (interface) specification
  - User interface
  - Component interface
- Architecture description
  - Systems as families of components (and their interface specs)
  - Putting together the components forming the architecture (composition)
  - Modularity: The interface abstraction supports the construction (composition) of the interface abstraction of a composed system from the
- Verification of an architecture
  - Prove the interface specification of the composed system from the interface specification of the components and the composition verification rules

Interfaces

- A connecting interface between two system describes
  - The information (messages, method calls) exchanged between the two systems
  - The logical rules and dependencies between the exchanged messages
- Ways to look at interfaces
  - Under which condition may a method be invoked or a message be send or received at an interface and what are the effects
  - Which sequences of methods or messages can be exchanged over an interface
- Interfaces of systems
  - How can we describe, in which connecting interfaces a system (component) fits

A first simple example: diaries and scheduling of dates

- As an example we consider a diary administrating the schedule for a person
- Operations:
  - ◇ Ask whether a date is possible/free
  - ◇ Enter a date that is possible
  - ◇ Delete a date
  - ◇ Ask for the set of dates that are in conflict with a given date in the diary

Data types for appointments

- We work with the following types

  Date

  Diary

  Person

  We do not give - for the moment - more specific explanations/specifications for these types

We assume a function

conflict : (Date, Date) Bool

A class CDiary - syntactic interface

```
Class CDiary =
{  y :  Diary
   p :  Person
```

This does not specify the Interface behaviour of the class

```
Method create_CDiary = (p : Person, r : Var CDiary)
Method isfree = (d : Date, b : Var Bool)
Method setdate = (d : Date)
Method deldate = (d : Date)
Method getdate = (d : Date, v : Var Set Date)

}
```

An algebraic specification of the basic data type

Specification of the theory of dates:

```
Spec DIARY =
{   imports Date
    Diary          : Type

    emptyd         : Diary
    isfreed        : (Diary, Date) Bool
    putd           : (Diary, Date) Diary
    getd           : (Diary, Date) Set Date
    deld           : (Diary, Date) Diary
```

Axioms

Diary **generated_by** emptyd, putd

isfreed(emptyd, d) = true
isfreed(y, d) ⇒ isfreed(putd(y, d), e) = (¬conflict(d, e) ∧ isfreed(y, e))

isfreed(y, d) ⇒ getd(y, d) = { }
isfreed(y, d) ⇒ getd(putd(y, d), d) = {d}
conflict(d, e) ∧ isfreed(y, d) ⇒ getd(putd(y, d), e) = {d} ∪ getd(y, e)
¬conflict(d, e) ∧ isfreed(y, d) ⇒ getd(putd(y, d), e) = getd(y, e)

deld(putd(y, d), e) = **if** d = e **then** y **else** putd(deld(y, e), d) **fi**
}

<u>Theorem</u>: getd(y, d) = { } ⇔ isfreed(y, d)

A basic idea: design by contract

- The idea: describe for each method/message under which conditions (precondition) a method may be invoked/a message may be received/sent and which effects this has (postcondition).
  - This is expressed in terms of state assertions and predicates describing relation between states

Example:

Value of y before the method invocation

Method setdate = (d : Date)

Pre isfreed(y, d)

Post y’ = putd(y, d)

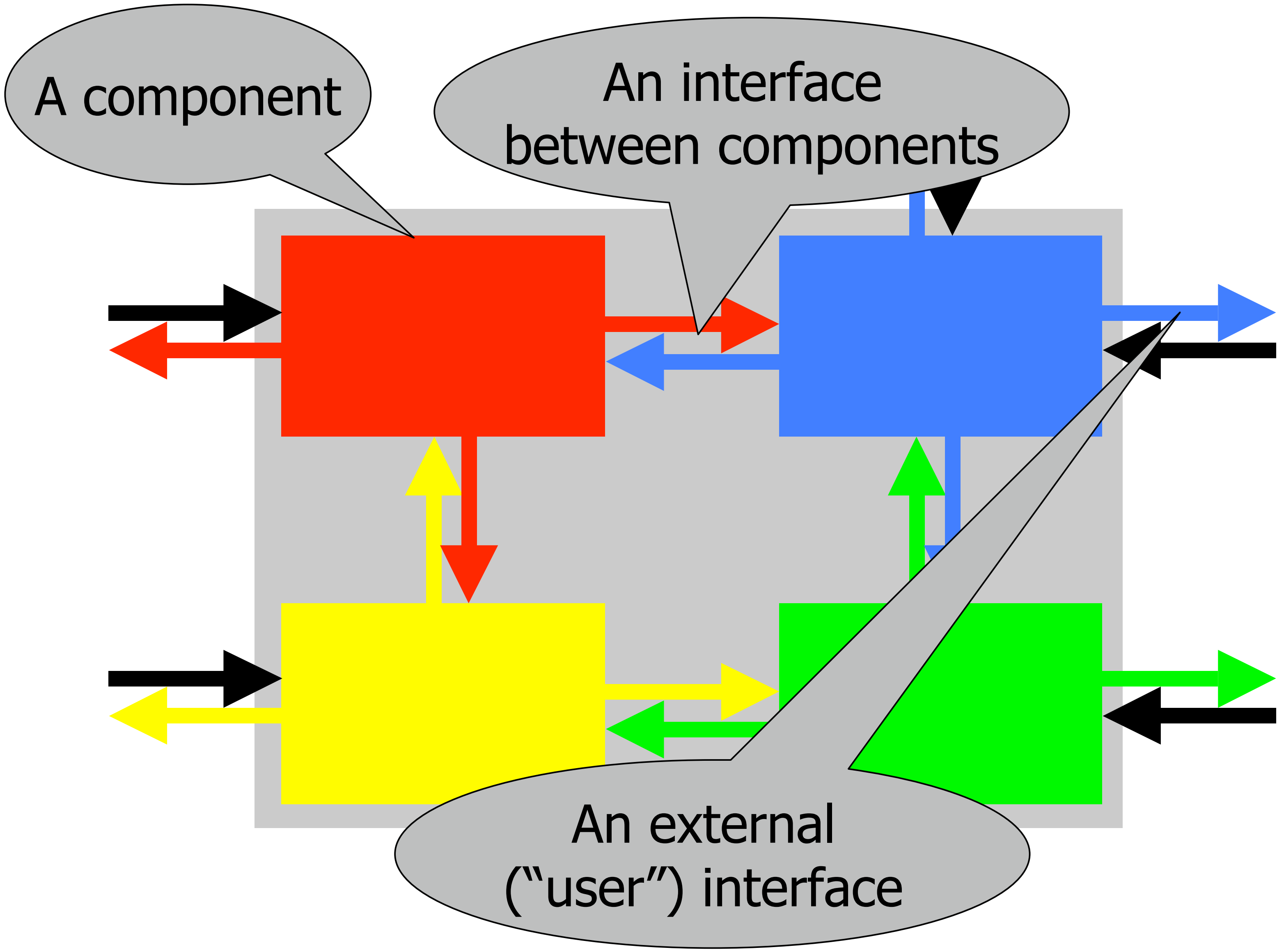
Value of y after the method invocation

A basic idea: design by contract

Observations

- Method invocations are considered as atomic state changes
- We have to refer to states of the systems/components that should be considered being hidden (following the principle of information hiding)
- We need a separated algebraic specification for the involved data types
- Methods/messages are treated in isolation and not in the context of comprehensive interactions

System structures and architectures


A component
An interface between components
An external ("user") interface

What is an observable/black box/interface behaviour

Component C1 is observable/behavioural compatible (has compatible interface behaviour) to component C2, if we can replace C2 in every (syntactically) correct system by C1 without violating the correctness.

If C1 is compatible to C2 and vice versa we call C1 and C2 observable/behavioural (have the same interface behaviours) equivalent.

Note: Classes with quite different state spaces may nevertheless be compatible/equivalent