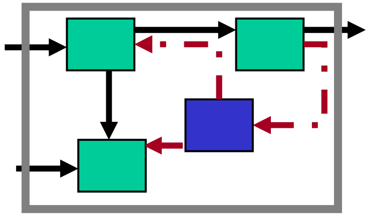
Advanced Topics in model-based Software Development

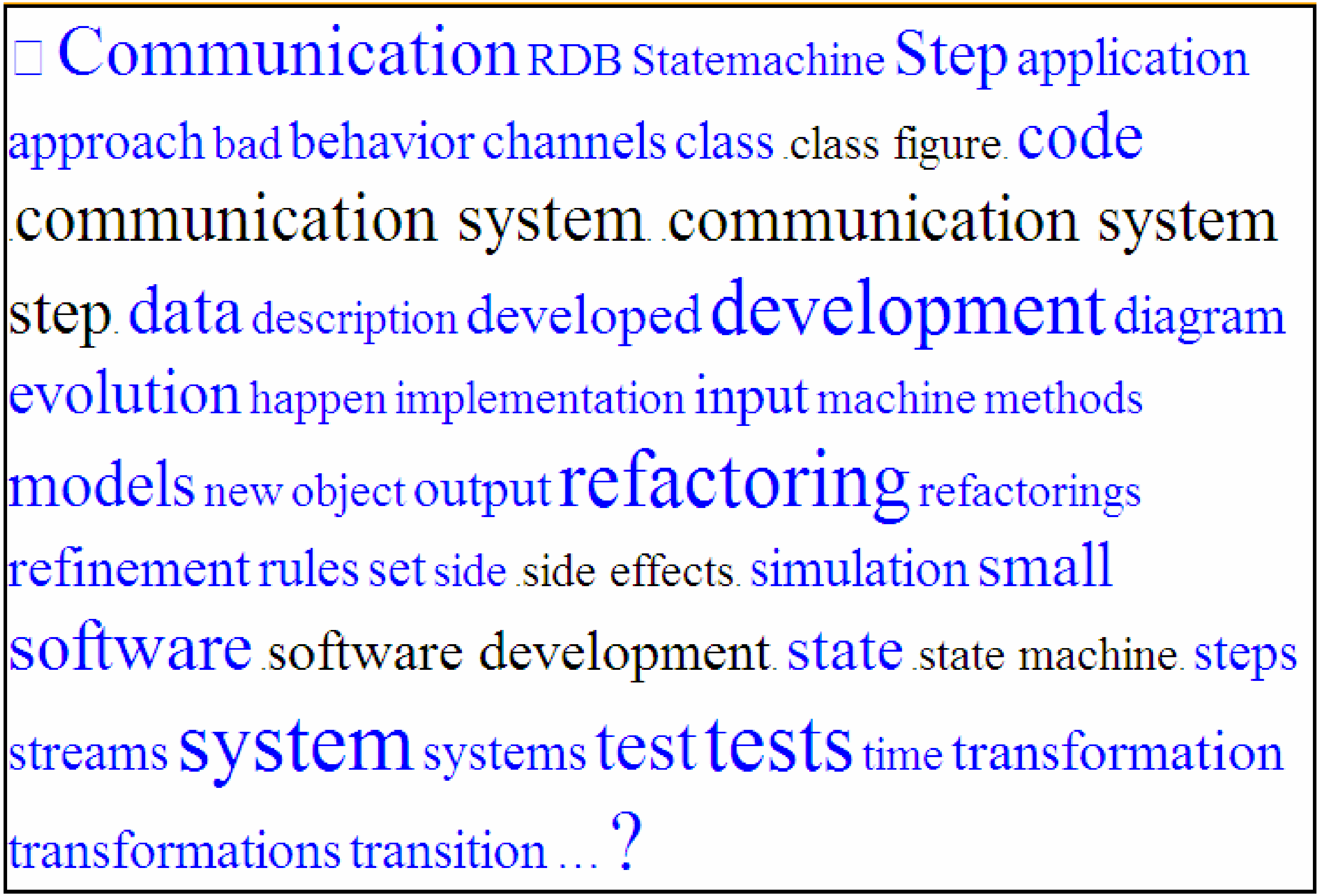


Prof. Dr. Bernhard Rumpe
ISIS - Institute for Software Integrated Systems
Vanderbilt University, Nashville

Software Systems Engineering
Technische Universität Braunschweig

http://www.sse-tubs.de/

Overview



□ Communication RDB Statemachine Step application approach bad behavior channels class .class figure. code .communication system. .communication system step. data description developed development diagram evolution happen implementation input machine methods models new object output refactoring refactorings refinement rules set side .side effects. simulation small software .software development. state .state machine. steps streams system systems test tests time transformation transformations transition … ?

Trends in software development

- Size and complexity of systems continually increase:
  - Isolated solutions → company-wide integration → E-Commerce → Systems-Of-Systems → World-Wide Cyber-Infrastructure
- New technologies:
  - EJB, XML, .Net, ...
- Diversification of application domains:
  - Embedded systems, business systems, telecommunication, mobility, ad-hoc changing infrastructures
- Growing methodological experience how to deal with these challenges
  - Agile Methods, e.g, address
    unstable requirements, time-to-market pressure,
    lean and effective development for small projects
  - Improved analytical techniques

⇨ Portfolio of software development processes / techniques etc.

Very short overview of Extreme Programming

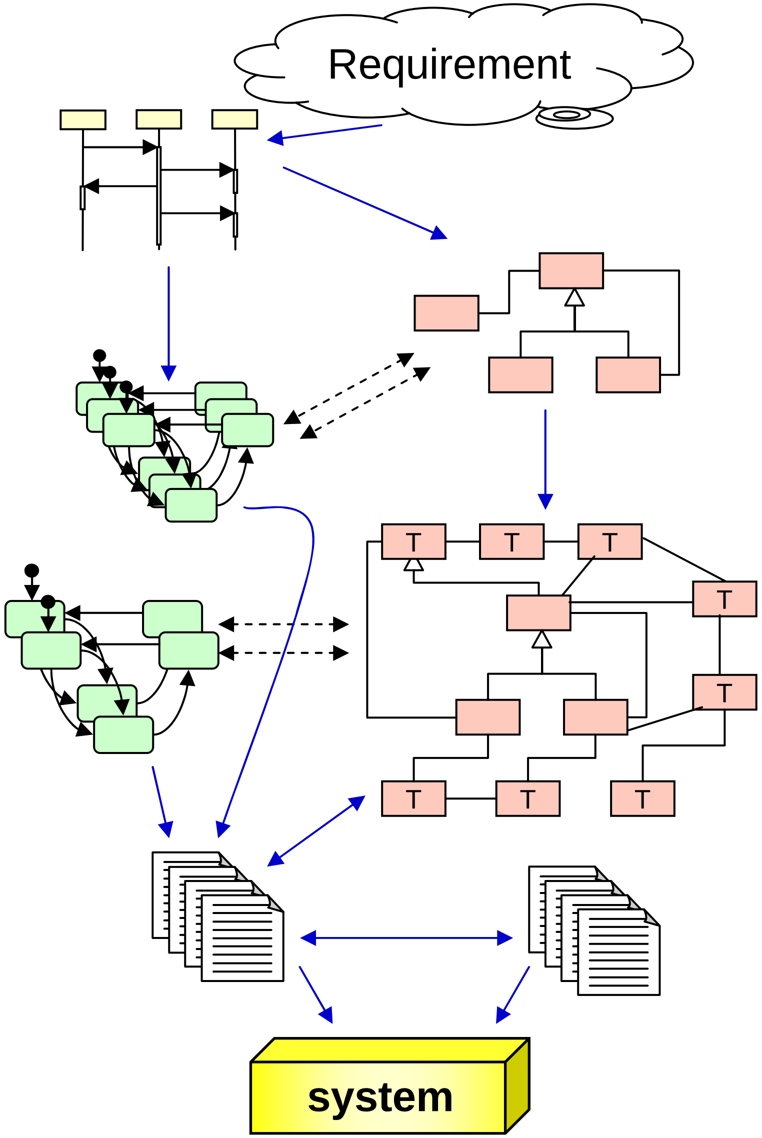
- „Best Practices“.
- Abandons many software development elements

- Activities (among others)

  Coding
  - Incremental
  - Coding standards
  - Runs of all tests
  - Refactoring

  Testing
  - Tests developed together with the code
  - Functional tests
  - Customers develop business logic tests

Idealized View on Model Driven Architecture



use cases and scenarios:
sequence diagram describes users viewpoint

application classes define data structures (PIM)

state machines describe
states and behavior

class diagram Nr. 2 („PSM“):
adaptation, extension, technical design

+ behavior for technical classes

code generation +
integration with manually written code

complete and running system

Core elements of an agile modelling method

- Incremental modelling
- Modelling tests
- Automatic analysis: Types, dataflow, control flow, ...
- Code generation for system and tests from compact models

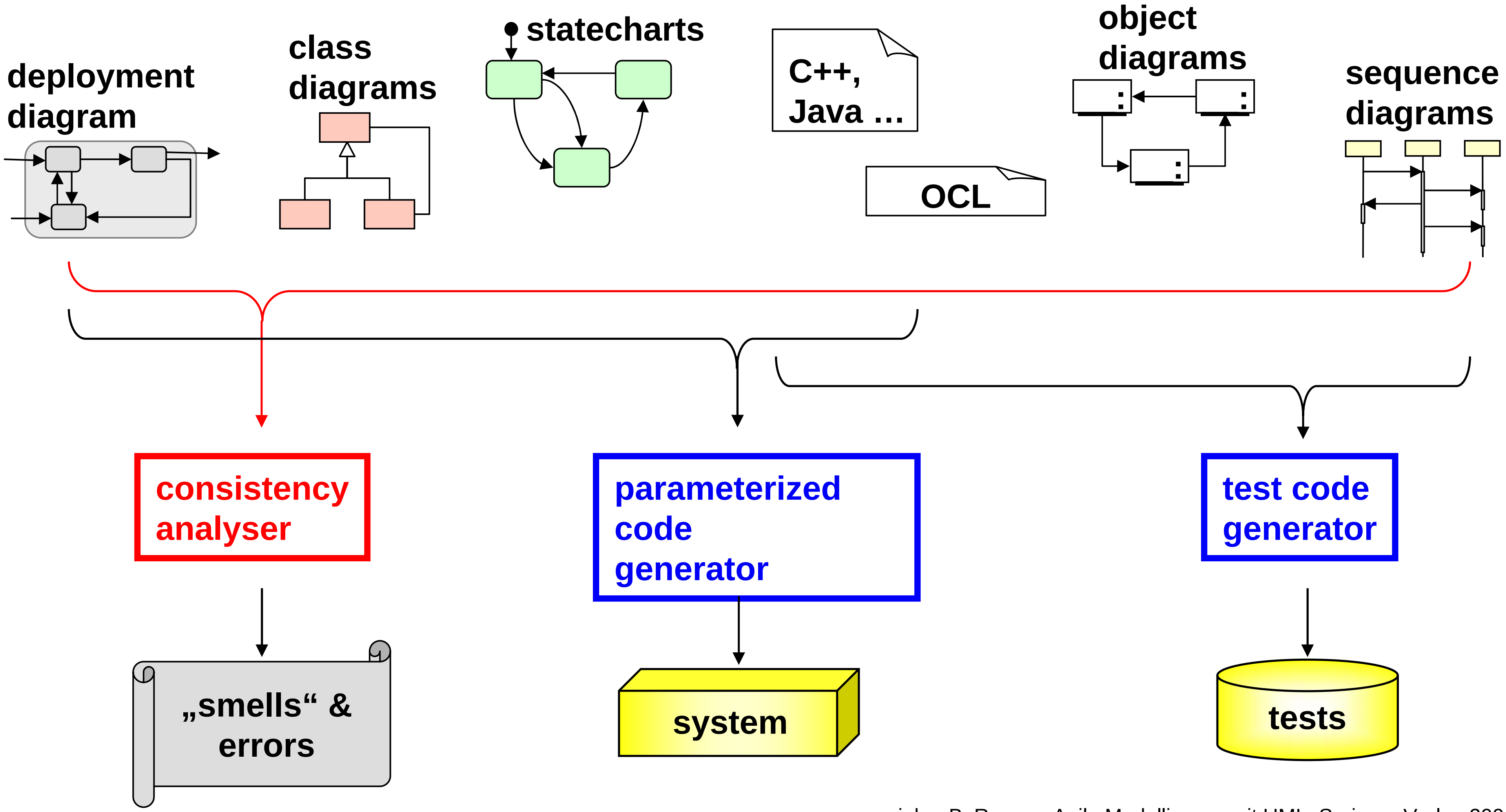
- Small increments
- Intensive simulation with customer participation for feedback

- Refactoring for incremental extension and optimisation
- Common ownership of models
- ...

⇨ This approach uses elements of agile methods based on the UML notation

Model-based “programming”

- Two kinds of models are used for the system and the executable tests



siehe: B. Rumpe: Agile Modellierung mit UML, Springer Verlag 2004

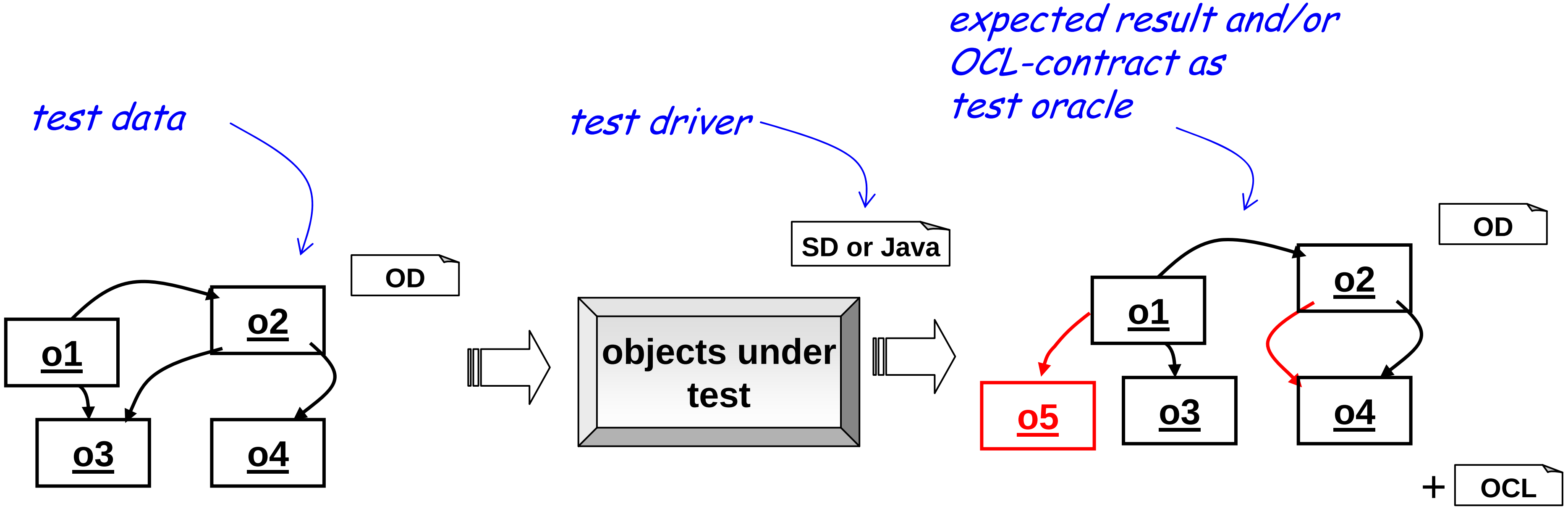
How the approach supports agile development

Core characteristics of agility:	Improvement through use of UML:
Efficiency of the developers	+ increased through advanced notation & tools
Reactivity: flexibility to deal with changes	+ incremental, small cycles + model-based refactoring
Customer focus	+ even more rapid feedback
Rely on individuals	+ less tedious work ? skilled people are necessary
Simplicity	+ refactoring increases extensibility
Quality is an emerging property	+ automated tests + better review-able designs + common ownership & pairwise development of models

Agile Model-Based Testing

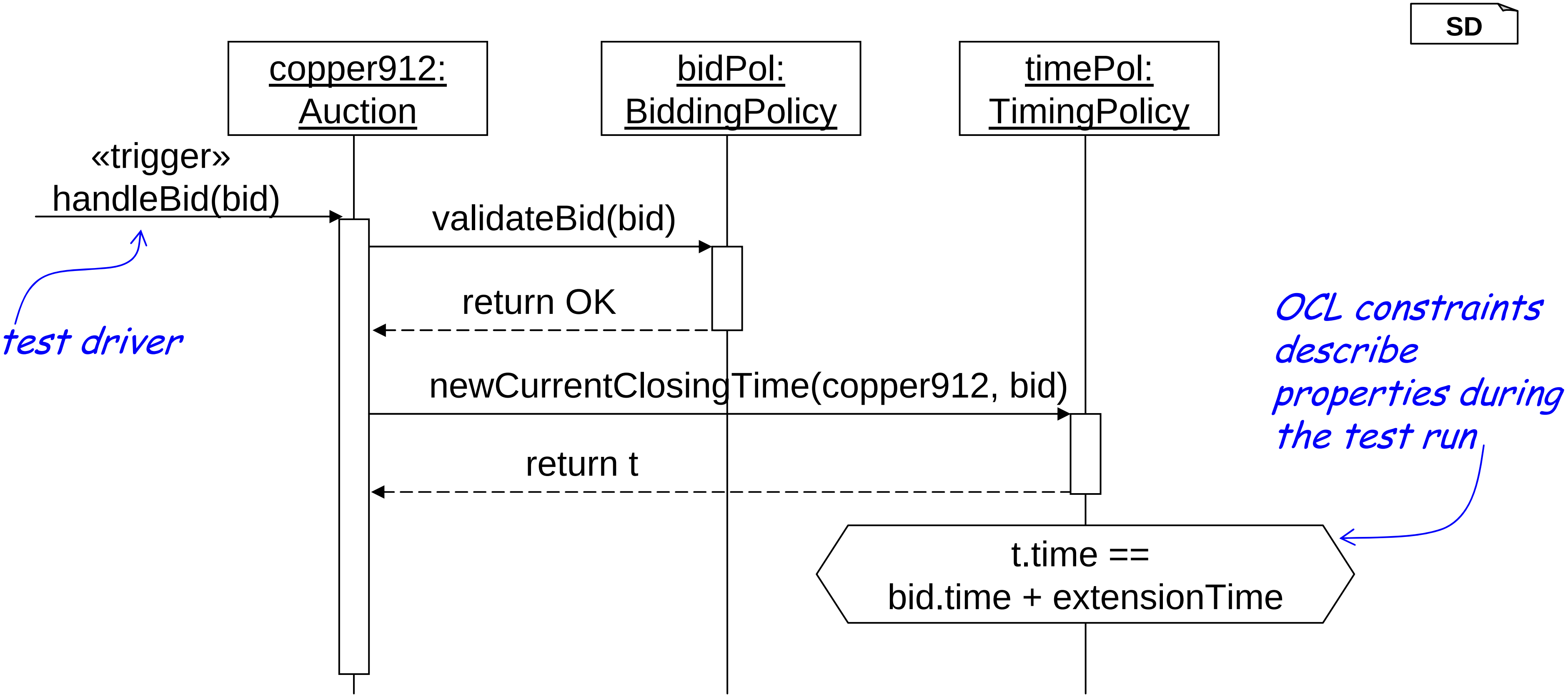
Typical infrastructure of an automated test

- Principle: use
  - relatively complete object diagram (OD) for test data
  - partial OD and OCL as oracle
  - sequence diagram (SD) or Java as test driver



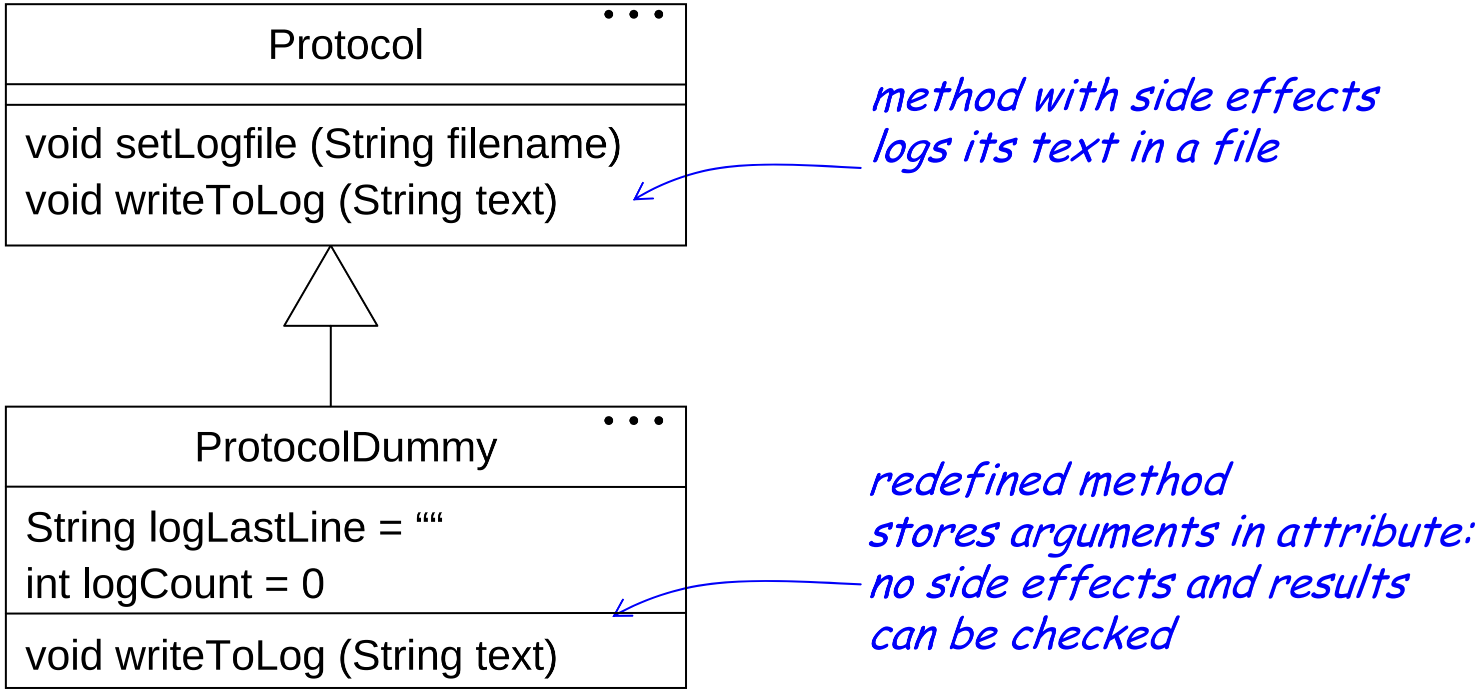
Sequence diagram: test driver and interaction description

- linear structure of an exemplaric system run
- + OCL for property description



Test pattern

- Systems need to be testable
- Example: Side effects like file protocol must be captured



method with side effects logs its text in a file

redefined method stores arguments in attribute: no side effects and results can be checked

Test pattern describe typical processes & structures for test definition

Test pattern for standard problems

- side effects (DB, GUI) → capsule with adapter & dummies
- static attributes → capsule with singleton object
- object creation → factory
- frameworks → separation of application and framework through adapter

- time → simulation through controllable clock
- concurrency → simulation through explicit scheduling
- distribution → simulation in one process space

Model-Based
Evolution / Refactoring

Software Evolution

- “Software evolution is the key problem in software development.”
  Oscar Nierstrasz

- Requirements change
- Platforms and system contexts evolve
- Bugs needs to be fixed
- Time and space optimisations are desired

- ⇒ Existing software needs to be evolved
- ⇒ Code as well as models need to be adapted to keep them consistent

Refactoring as a special form of transformation

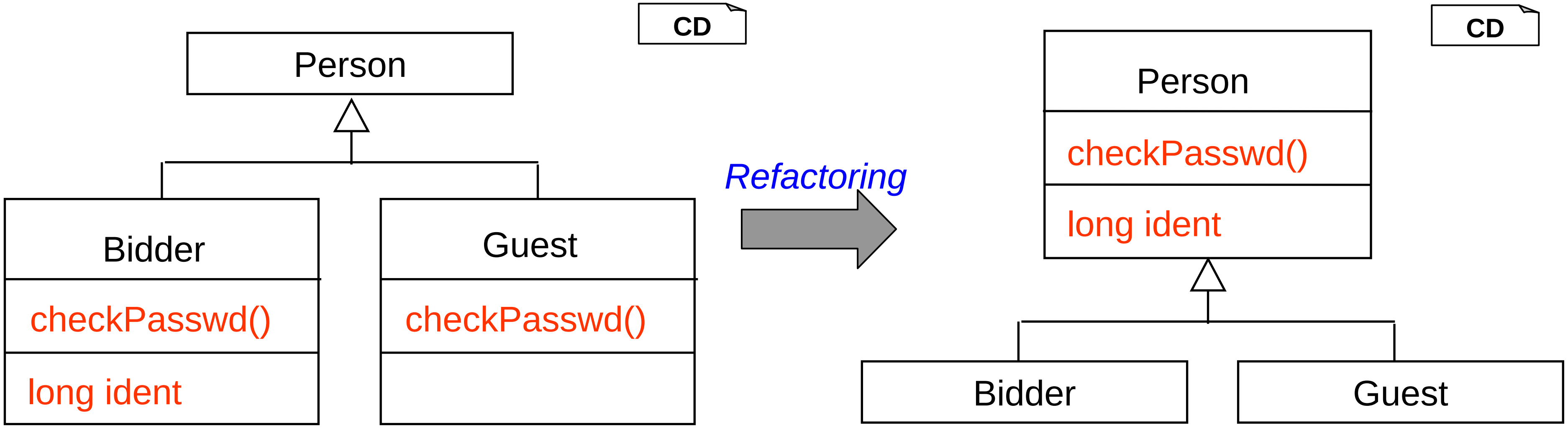
- Refactoring is a technique to
  - improve internal structure / architecture of a system, while
  - preserving observable behaviour

- Refactoring rules:
  - series of systematically applicable, goal directed steps

- Powerful through
  - simplicity of piecewise application and
  - flexibility of combination of systematic steps

- Roots:
  - Opdyke/Johnson 1992 had 23 refactorings on C++
  - Fowler’1999 has 72 refactoring rules for Java

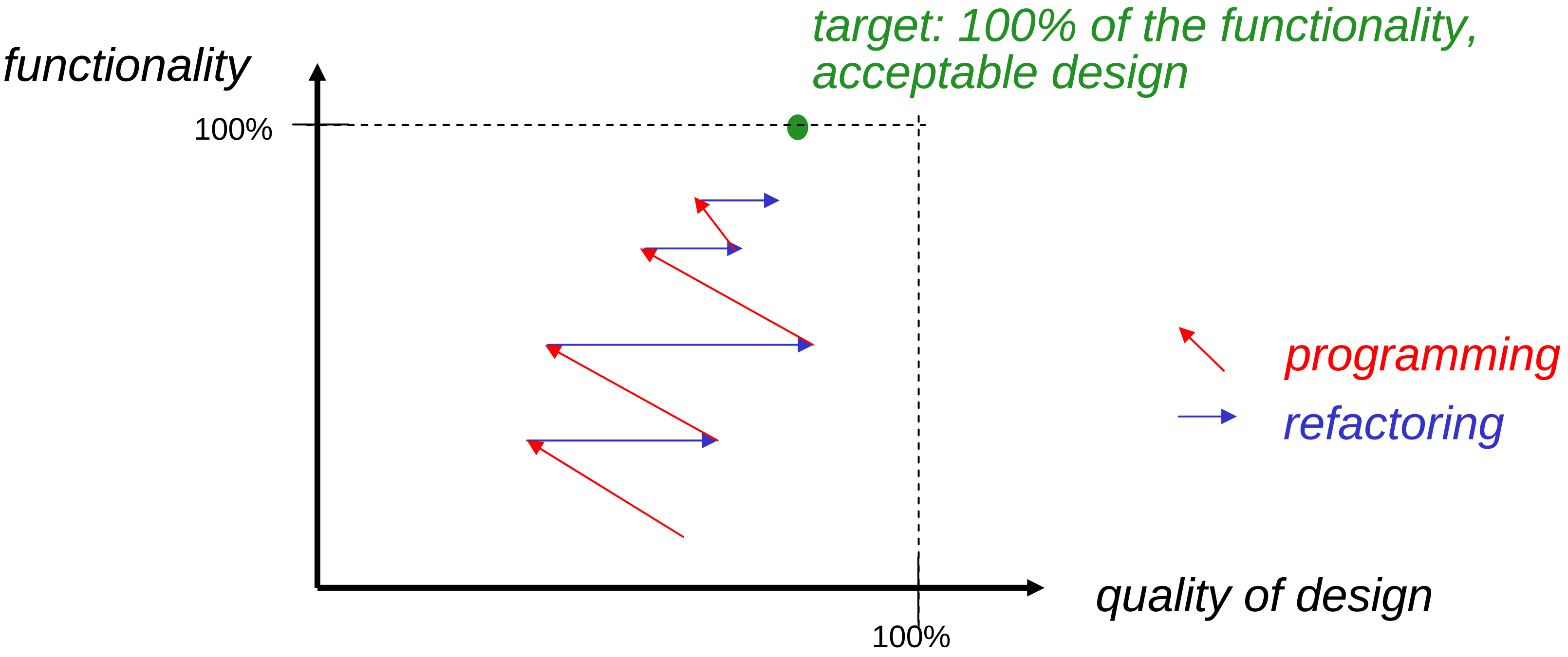
Refactoring example 1



- Pull Up Attribute “ident” into superclass: structural generalization
- Factor Method “checkPasswd()” and adapt it
- Preservation of observable behaviour?
    - depends on viewpoint: class, component, system

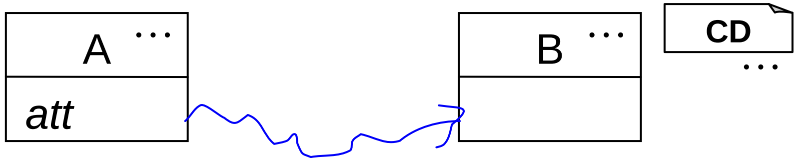
Principle of refactoring

- Refactoring is orthogonal to adding functionality
- An idealised diagram:



Example: moving an attribute

- Attribute „att“ shall be moved from class A to B

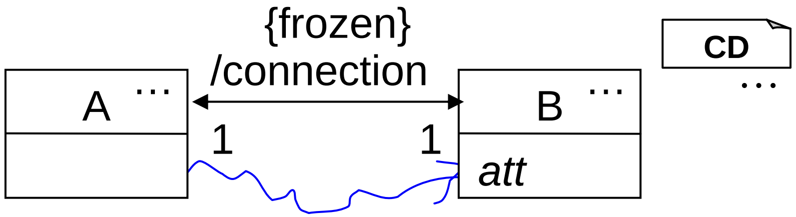


Code

```
// code
a.att
```

a.exp is the navigation path from A to B

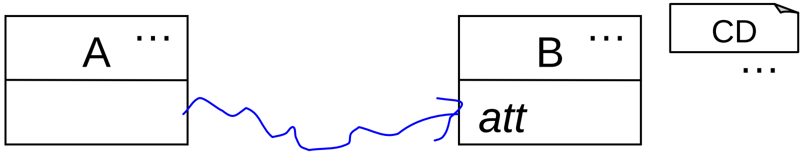
⇓



OCL

```
context A a inv:
 a.connection == a.exp;
context A a1, A a2 inv:
 a1 != a2 implies
   a1.connection != a2.connection
```

⇓



Code

```
// Code
a.exp.att
```

Refactoring example: changing data structures

A series of steps to apply:

1. Identify old data structure:
   here: long to be replaced by Money

Auction ...
long currentBidInCent

2. Add new datastructure + queries
   + compile

Auction ...
long currentBidInCent **Money bestBid**

3. Identify invariants to relate both

```
context Auction inv M:
  currentBidInCent ==
        bestBid.valueInCent()
```

4. Add code for new data structure & invariants wherever old data structure is changed
   + compile & run tests

```
currentBidInCent = ...
bestBid.setValue...
assert M
```

5. Modify places where old data structure was used
   + compile & run tests

= ... `currentBidInCent` ...

⇓

= ... `bestBid.valueInCent()` **...**

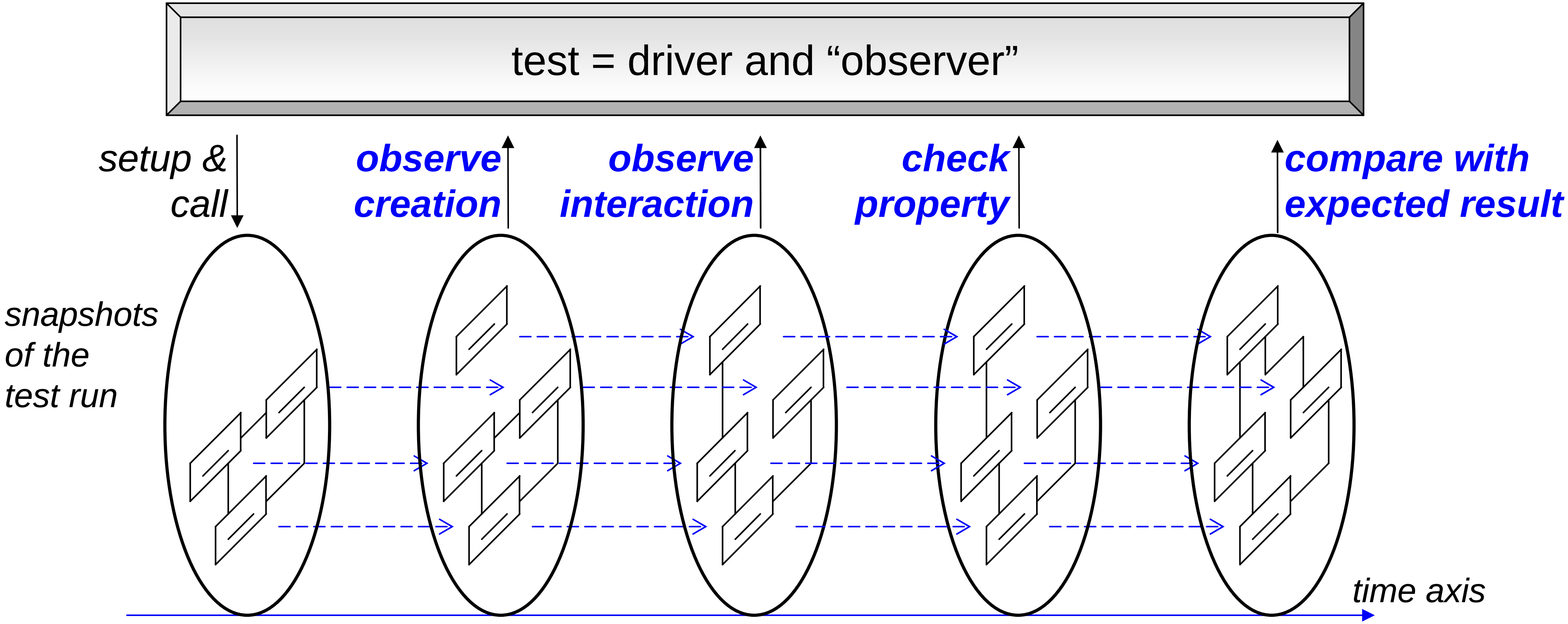
6. Simplify + compile & run tests

7. Remove old data structure + compile & run tests

Auction ...
Money bestBid

Test as observation for refactoring

- Both structure and behaviour are observed by tests



test = driver and "observer"

setup & call

observe creation

observe interaction

check property

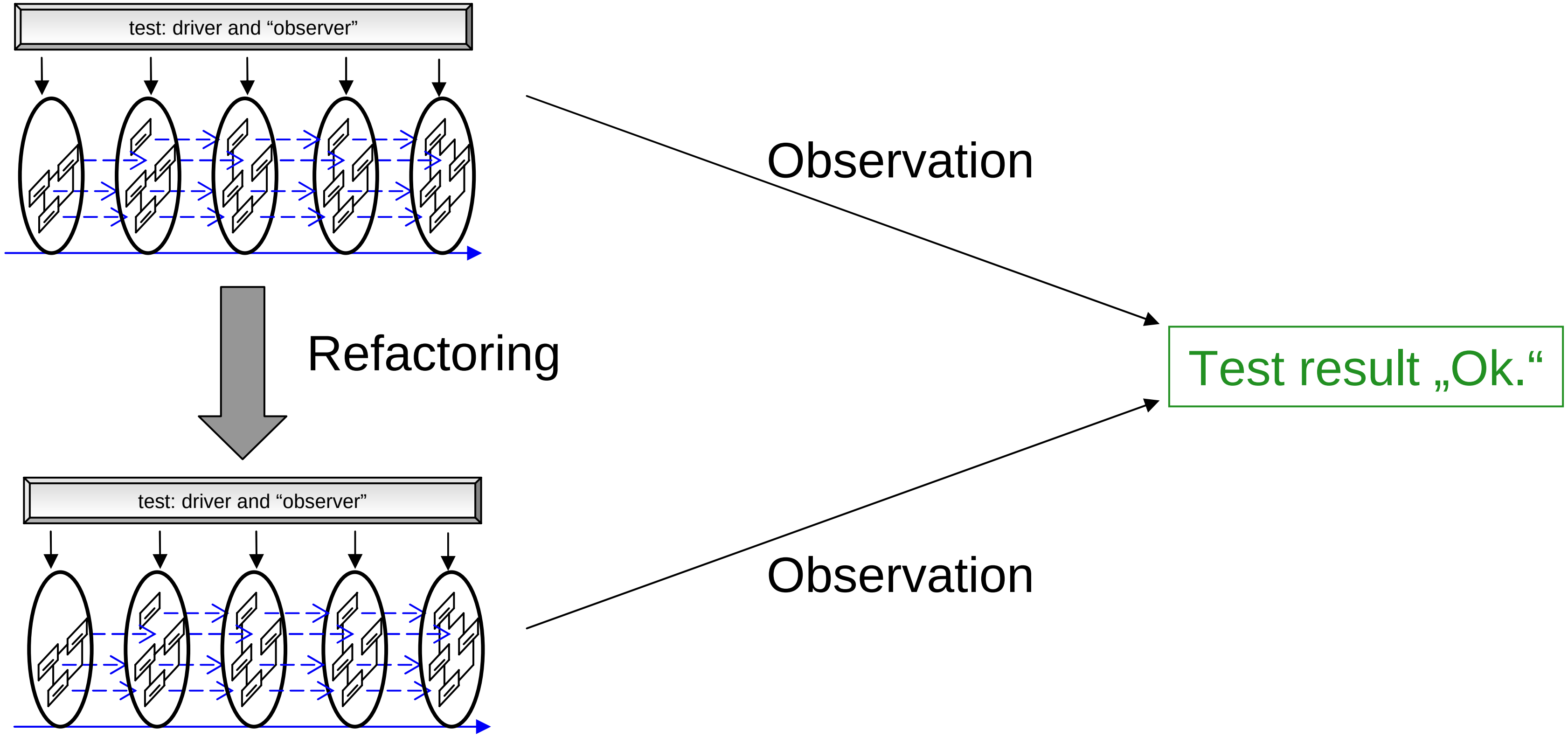
compare with expected result

snapshots of the test run

time axis

Validation of refactorings using tests

- Observation remains invariant under refactoring



Evolution as strategic refactoring

- Evolution in the small supports evolution in the large
- Evolution in the small:
  - Transformation rules
    = small, manageable and systematic steps

- General goals of transformations:
  - reasoning,
  - deriving implementation oriented artefacts,
  - building abstractions e.g. for reengineering,
  - evolutionary improvement

- Transformation calculi can serve as technical basis for an evolutionary approach to software development

Examples for Transformational Development

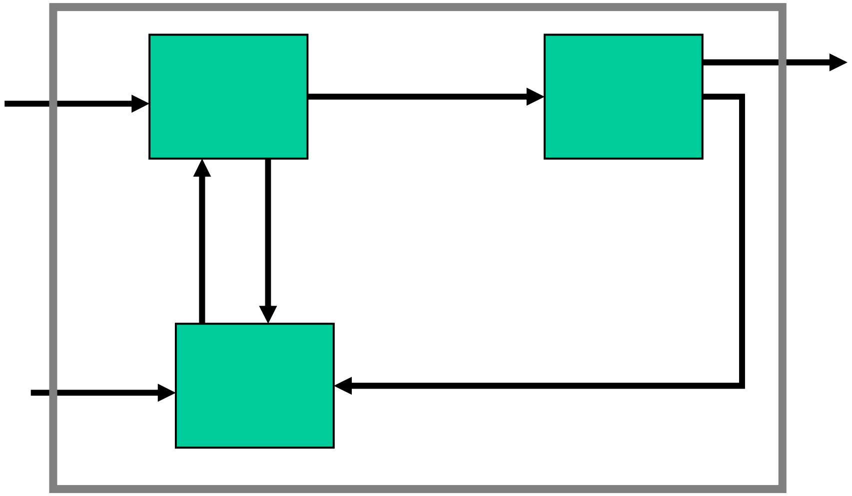
▪ Mathematical calculi	for reasoning
▪ State machine transformations	for error completion, determinism, …
▪ Stepwise refinement of programs (Bauer, Partsch)	for software development
▪ Hoare calculus	for reasoning over programs
▪ Refactoring (Opdyke, Fowler)	for evolution
▪ ….	

Streams & Behaviors

- Communication histories over channels are modeled by **streams**:
  - streams s = <1,2,a,3,b,b, ...>

- Channel valuations assign streams to channel names: $\overrightarrow{C} = C \to M^{\aleph}$
- An I/O behavior relates input and output channel valuations: $\beta : \overrightarrow{I} \to \mathbb{P}(\overrightarrow{O})$

Composition of behaviors can be modeled graphically:

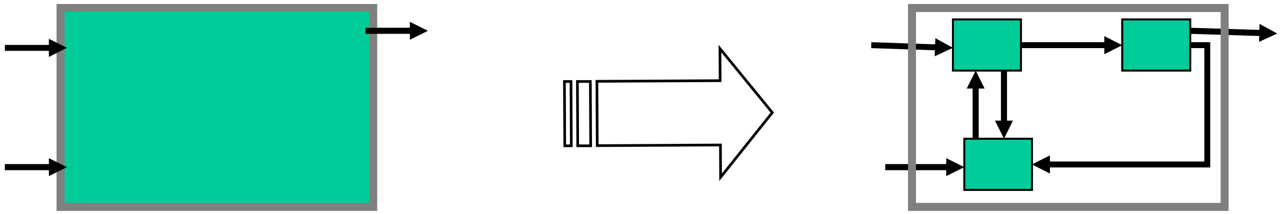


Kinds of Transformations

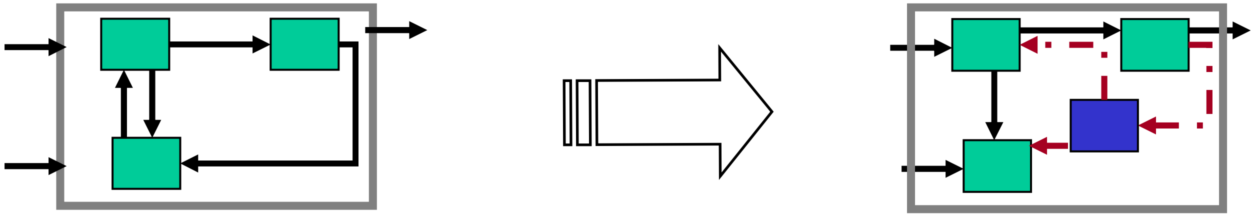
- Behavioral Refinement:
  - A behavior β**’** is a refinement of a behavior β

$$\forall x : \beta'(x) \subseteq \beta(x)$$

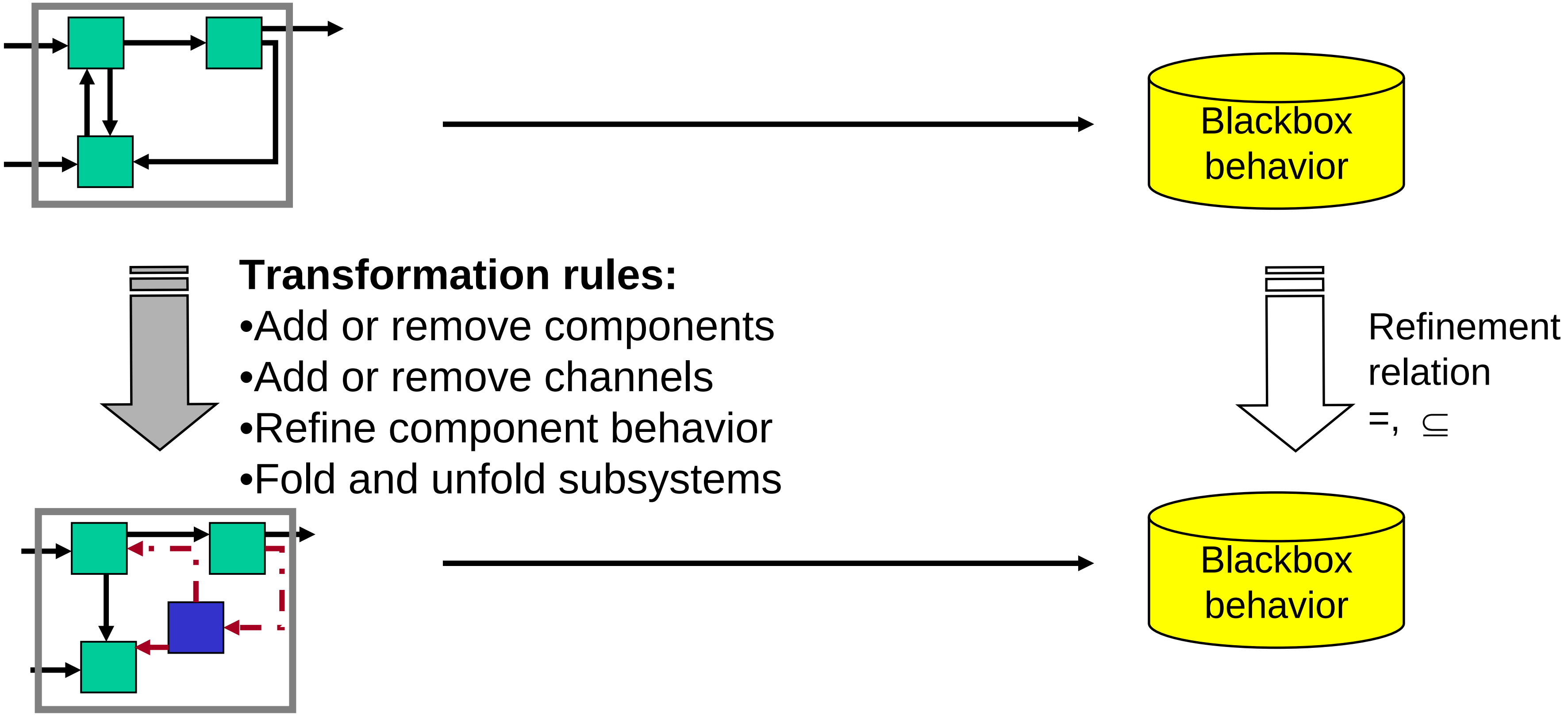
- Structural Refinement (Decomposition)



- Evolution of architecture (Refactoring)



Semantics of Transformations



Transformation rules:

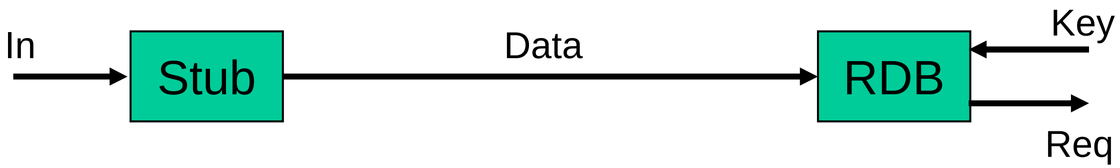
- Add or remove components
- Add or remove channels
- Refine component behavior
- Fold and unfold subsystems

Blackbox behavior

Refinement relation
=, ⊆

Blackbox behavior

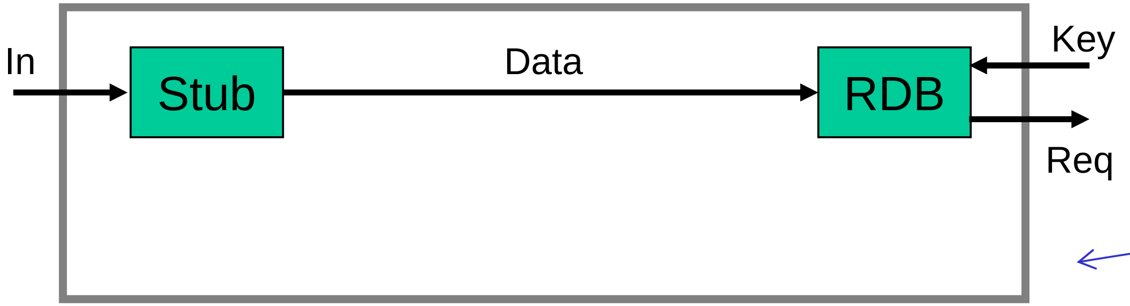
Example: Communication System



- Data (Consisting of key and value) is accepted via „In“
- and transmitted to the „Remote Data Base“ (RDB)
- Upon sending a key, the requested value is sent

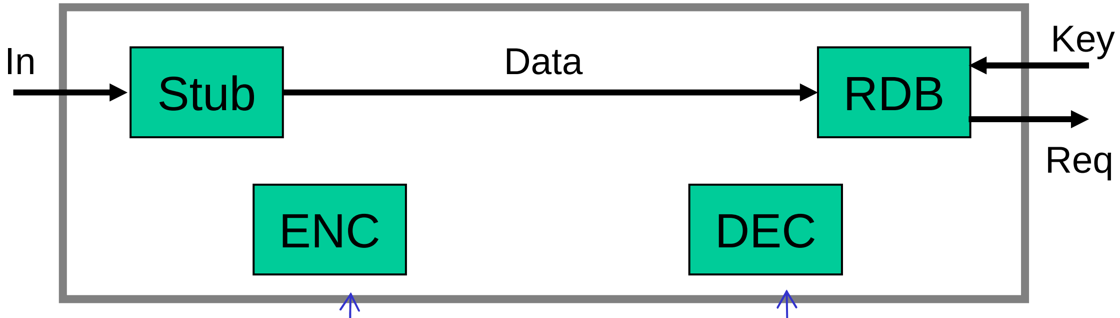
- Problem:
  - Transmission from Stub to RDB shall be encrypted
- Solution:
  - We evolve the part of the system, we are currently focusing on

Example: Communication System



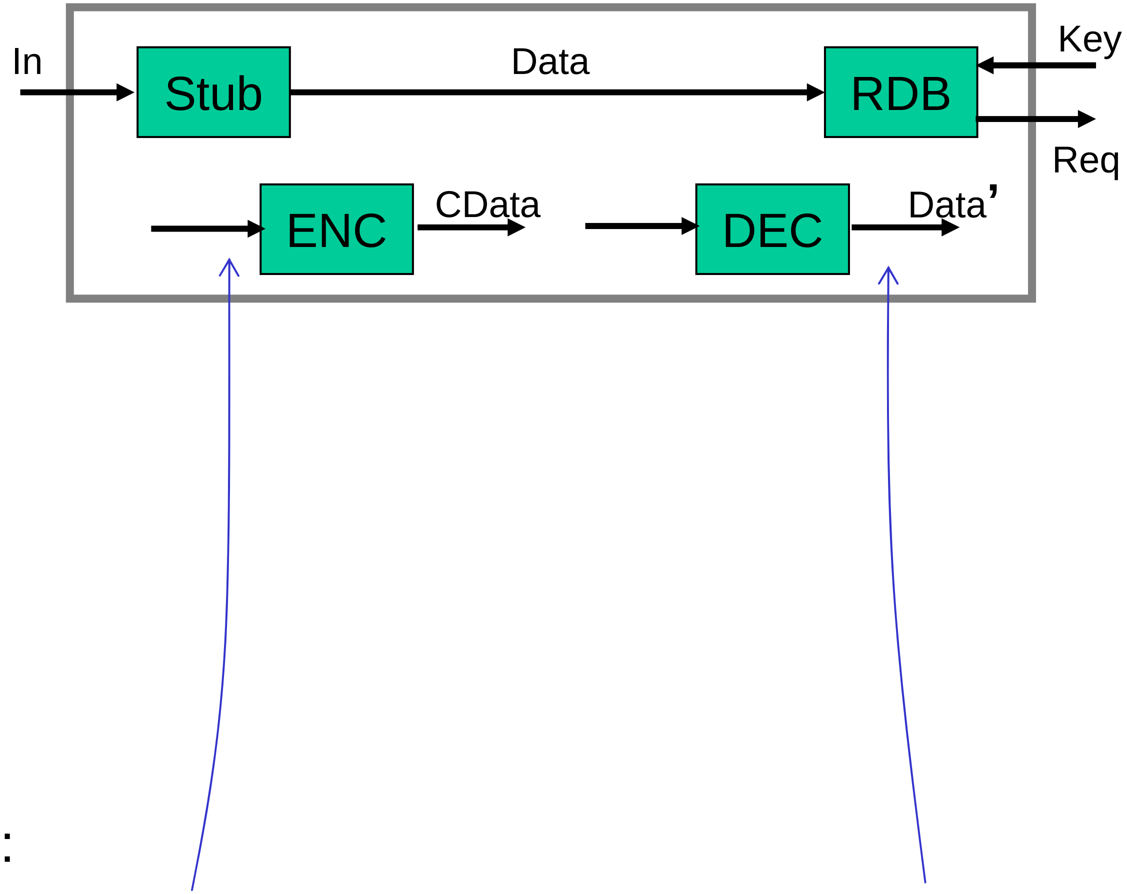
- Step 0:
  - Decide what the „observed behavior“ will be that shall not be changed.
  - Here, we group the observed channels into a component

Example: Communication System



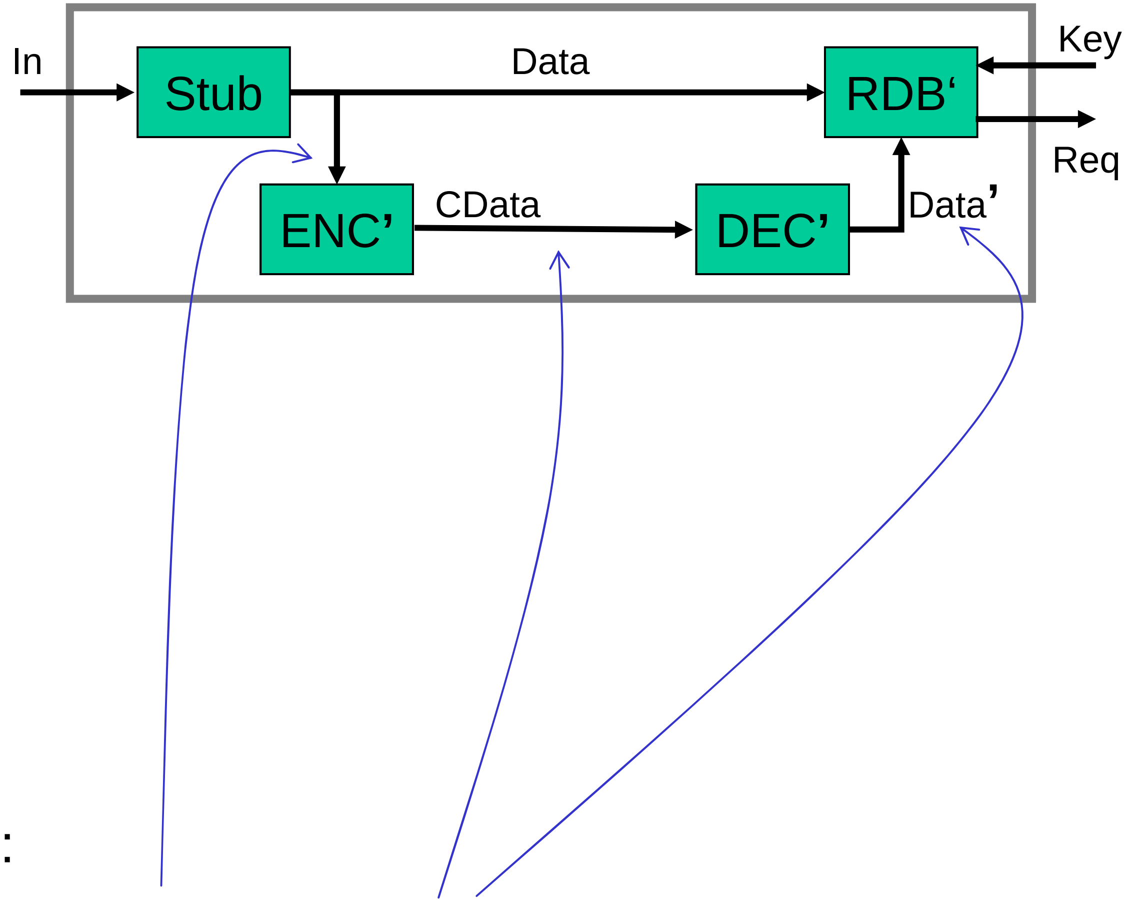
- Step 1:
  - Add encryption and decryption components
  - No connection to the rest of the system: Nothing bad can happen

Example: Communication System



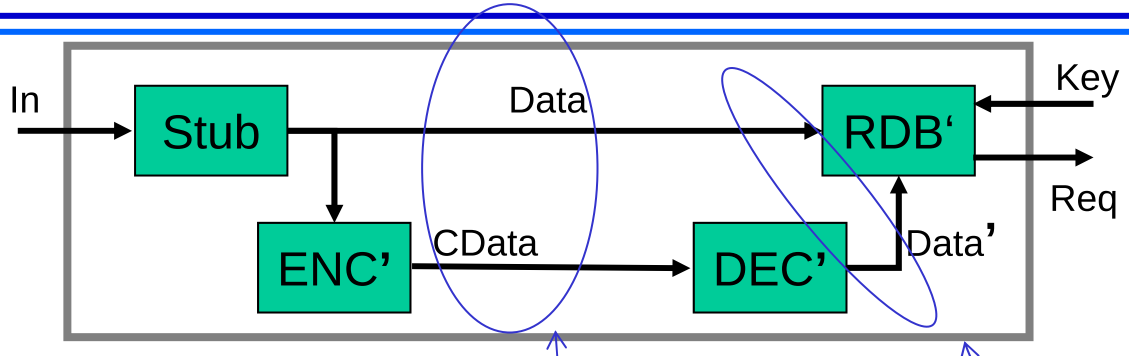
- Step 2:
    - Define signature and behavior of new components (may be we reuse of the shelf components?)
    - Still no connection to the rest of the system: Nothing bad can happen

Example: Communication System



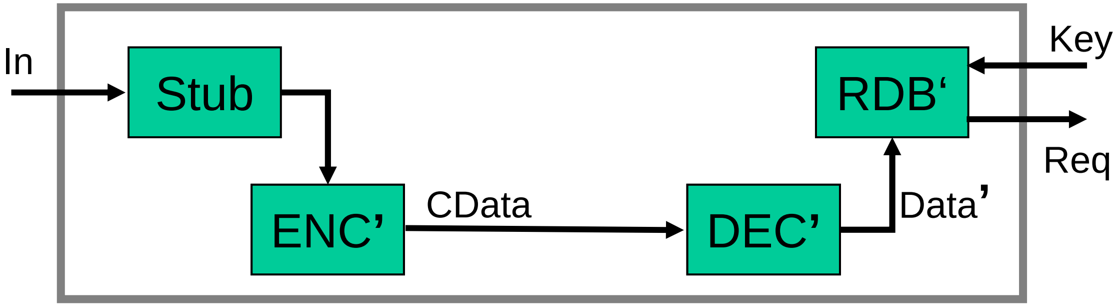
- Step 3:
  - connect Input and output channels
  - RDB now has an additional input channel, but doesn‘t use it yet
  - Still nothing bad can happen

Example: Communication System



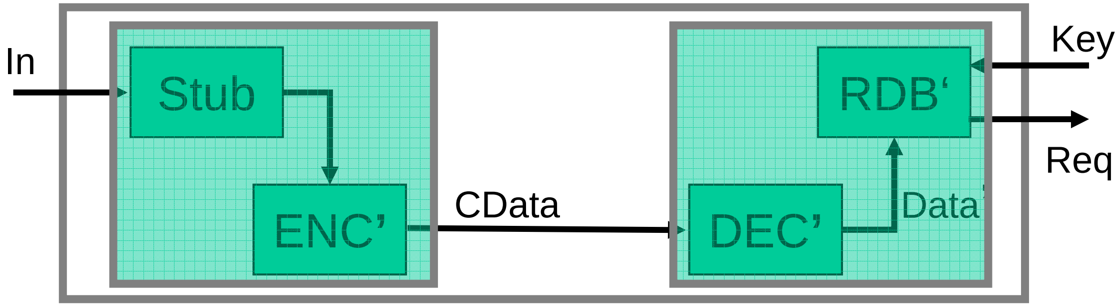
- Step 4:
  - establish invariant between channels:
    - CData = encrypt* (Data)
    - Data‘ = Data (modulo time)
  - RDB‘ now can use Data‘ instead of Data

Example: Communication System



- Step 5:
  - Remove unused channel Data

Example: Communication System



- Step 6:
  - Fold new parts into subcomponents

State machines

- A state machine is a tupel $A=(S,M,\delta,I)$ consisting of:
  - set of states S,
  - set of input and output messages M,
  - state transition relation δ: $(S\times M) \rightarrow 2^{(S\times M^*)}$ and
  - set $I \subseteq S\times M^*$ of initial states and outputs

- Nondeterminism = underspecification
- Partiality = total underspecification (chaos)

Conclusion

- Further diversification of SE techniques / tools / methods leads to a portfolio of SE techniques

- Intelligent use of models allows to improve development

- Methodical knowledge allows more efficient processes
  - correctness by construction
  - automated tests over documentation and reviews
  - evolutionary development (refactoring) over big-upfront-design phase

- “Model engineering”

State machines

- A state machine is a tupel $A=(S,M,\delta,I)$ consisting of:
  - set of states S,
  - set of input and output messages M,
  - state transition relation $\delta: (S \times M) \rightarrow 2^{(S \times M^*)}$ and
  - set $I \subseteq S \times M^*$ of initial states and outputs

- Nondeterminism = underspecification
- Partiality = total underspecification (chaos)

Semantics of a state machine

- One transition contains one input message and a sequence of output messages
- Semantics is a relationship between input and output streams

$$\boldsymbol{M} : (S,M,\delta,I) \rightarrow 2^{(M^* \times M^*)}$$

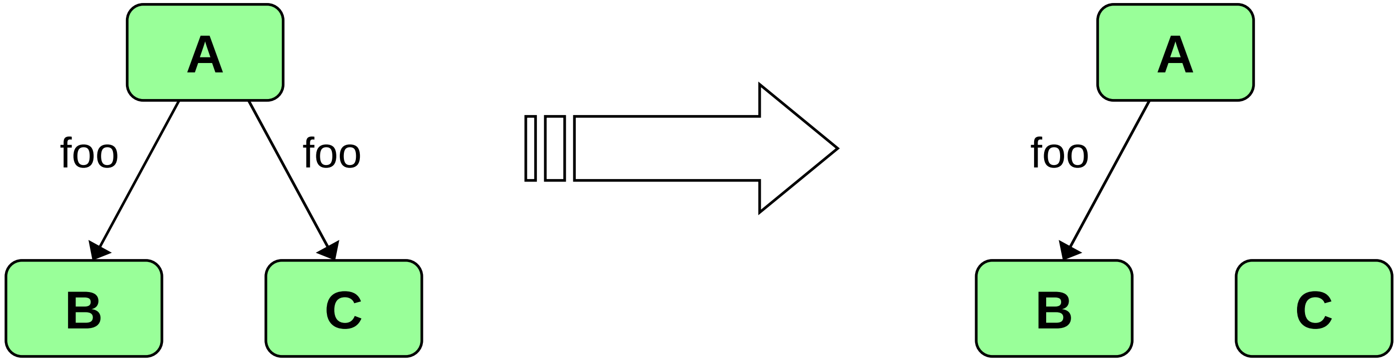
- Behavioral refinement between automata:

$$A_1 \Rightarrow A_2 \qquad \text{iff} \qquad \boldsymbol{M}[A_1] \supseteq \boldsymbol{M}[A_2]$$

- Refinement rules can be used to
  - constrain (detail) behavior description
  - inherit state machines
  - implementation of an interface

Example transformation rule

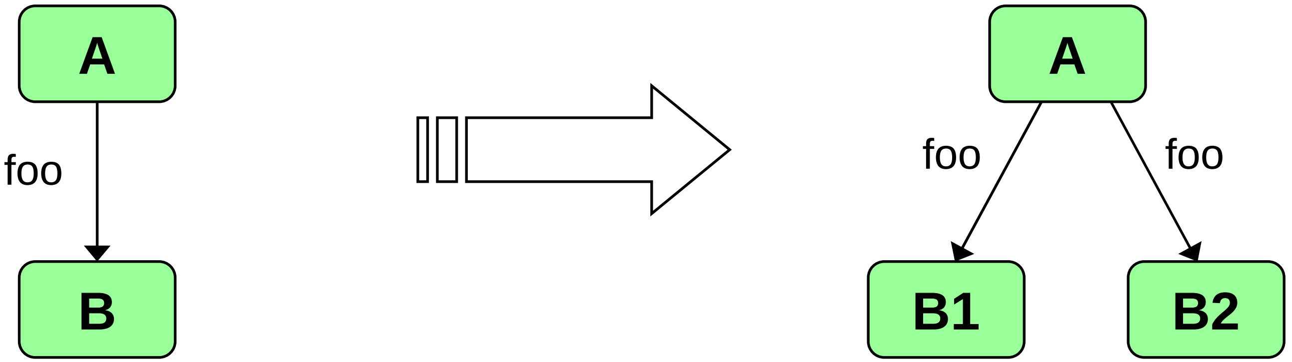
- Remove a transition:
  - if there is an alternative



- If preconditions are present, the remaining transition must overlap the removed transition. This must be proven.

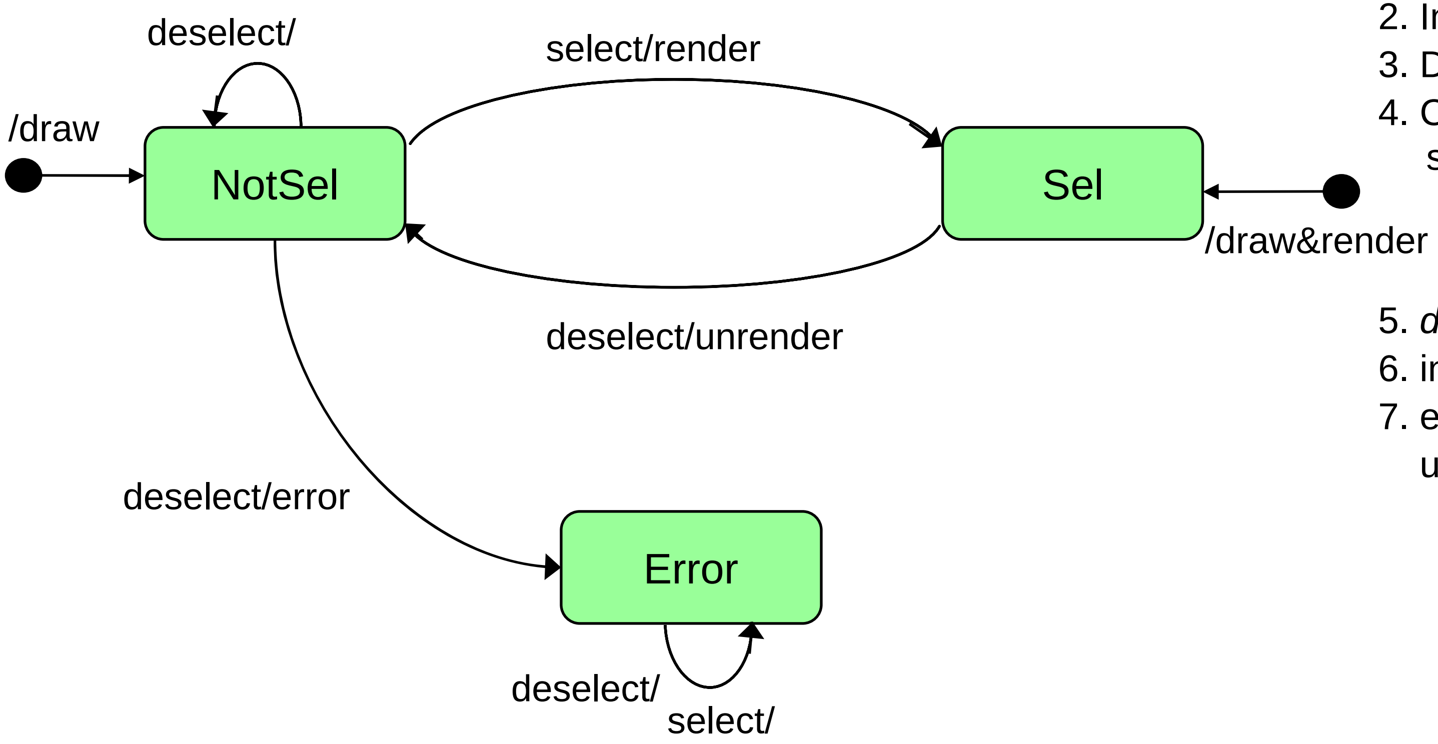
Example transformation rule 2

- Split a state



- Multiplies transitions
  - useful to remove unwanted transitions

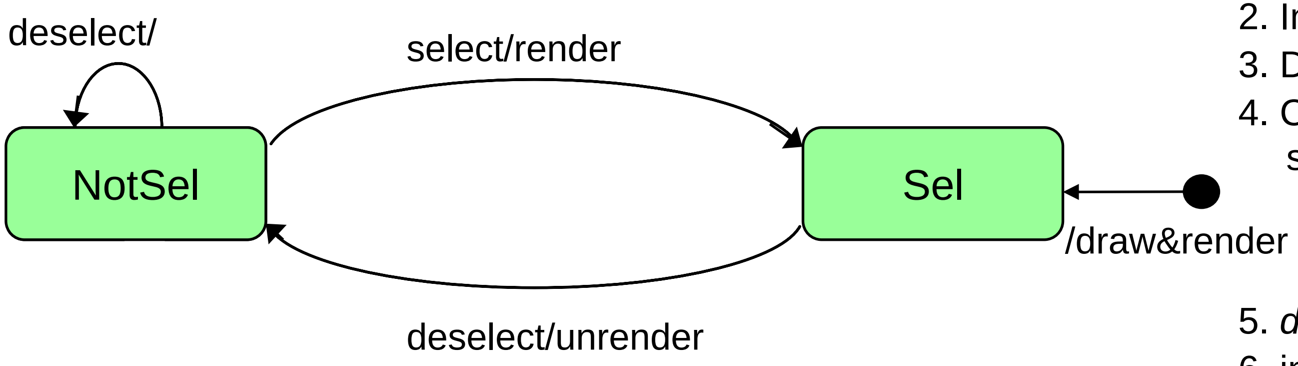
Example: Statemachine for class Figure



1. Intro. state *Sel*(ected)
2. Intro. state *NotSel*(ected)
3. Define init states
4. Constrain method *select* in
   state *NotSel*

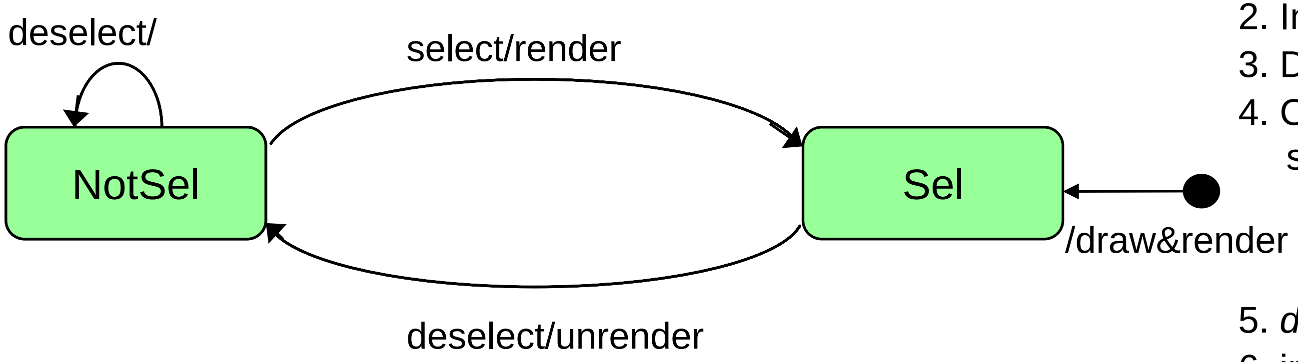
5. *deselect* in state *Sel*
6. introduce error state
7. error completion
   using underspecification

Example: Statemachine for class Figure



1. Intro. state *Sel*(ected)
2. Intro. state *NotSel*(ected)
3. Define init states
4. Constrain method *select* in state *NotSel*
5. *deselect* in state *Sel*
6. introduce error state
7. error completion using underspecification
8. specialize de*select*: remove transition
9. remove error state
10. specialize initial states

Example: Statemachine for class Figure



1. Intro. state *Sel*(ected)
2. Intro. state *NotSel*(ected)
3. Define init states
4. Constrain method *select* in state *NotSel*
5. *deselect* in state *Sel*
6. introduce error state
7. error completion using underspecification
8. specialize de*select*: remove transition
9. remove error state
10. specialize initial states

Each step is a refinement of the observable behavior of that class