HUMBOLDT UNIVERSITY BERLIN
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Reengineering of a chaotic legacy software system

Dipl.-Inf. M. Hildebrandt

Overview

- the project behind: ATEO
  - Project
  - SAM & ATEO software system
- starting point: SAMs 2.0
  - history of development
  - problems
- reengineering
  - steps and their results
  - reengineered architecture
- comparision of variants
  - architectures
  - implementations

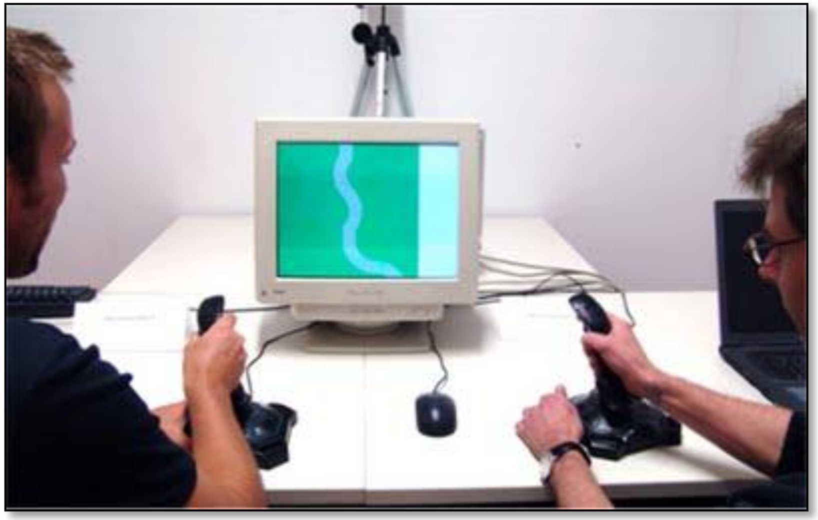
PROJECT ATEO

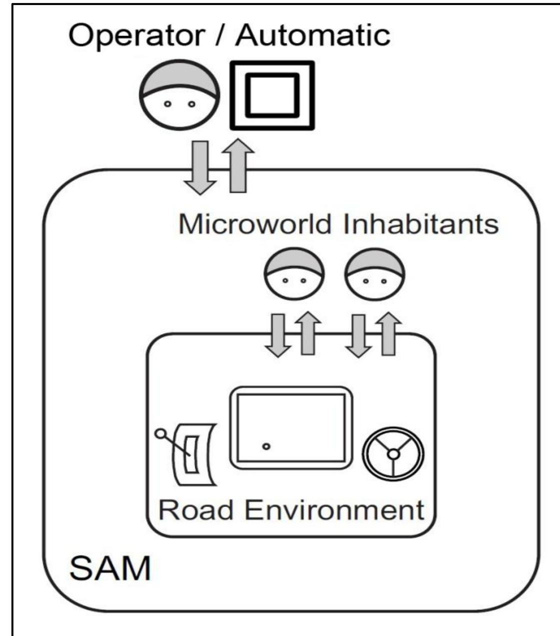
The Project ATEO

- part of the research training group **prometei**
  - cooperation of several universities and institutes
  - DFG funded

- **A**rbei**t**steilung **E**ntwickler-**O**perateur (ATEO)
  - engl.: Division of Labor between Developers and Operators
  - Researching the *optimal function allocation* between
    - humans (operator) and
    - machines (designed by developers)

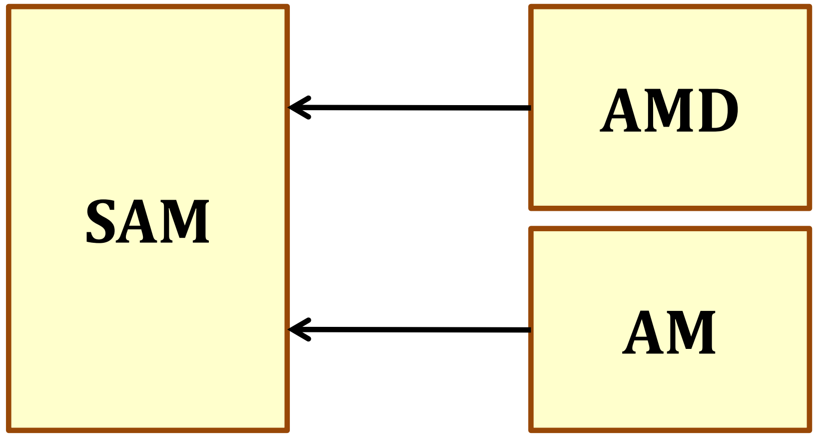
Socially Augmented Microworld (SAM)





- data gained from computer-based experiments
  - models a dynamic process as a **tracking task**
  - microworld inhabitants (probands) as **social factor**: enable an unpredictable but retrospectively explainable behaviour
  - operator (proband) / automatic as external factors
    - supervising and controlling the process

SAM within the ATEO system



- **SAM**
  - simulating tracking task
  - logging experimental data
- **ATEO Master Display** (AMD)
  - display and control panel of the operator
  - supervising and controlling of the tracking process
- **Automatics** (AM)
  - designed and implemented by developers
  - supervising and controlling of the tracking process

Starting point: SAMs

- implemented in Smalltalk/Squeak
  - integrated runtime and development environment (VM)
  - open source, freely available

- increased **quality requirements** concerning
  - **Stability**
    experiments must be conducted without interruptions
  - **Correctness**
    experiments must be conducted in the way they are designed
  - **Performance**
    soft real-time application, the simulation must be fluent
  - **Maintainability**
    requirements change often (according to new research data)

Starting point: SAMs (cont.)

- historically grown software (since 2004)
  - many changes
  - alternating developers (graduands, psychologists)
  - no software engineering
    - no requirements engineering
    - no architecture design
    - no quality management
    - no change management

- **so:** unknown architecture, i.e.
  - overall structure, dependencies unknown
  - quality properties only vaguely known
    - bad maintainability
    - bad performance

REENGINEERING

Approach: Overview

- **Reengineering** in 4 steps:

1. **Reverse Engineering**
   analysis and documentation of the existing architecure

2. **Restructuring**
   transformation of the existing architecture

3. **Forward Engineering**
   requirements, OOA, OOD

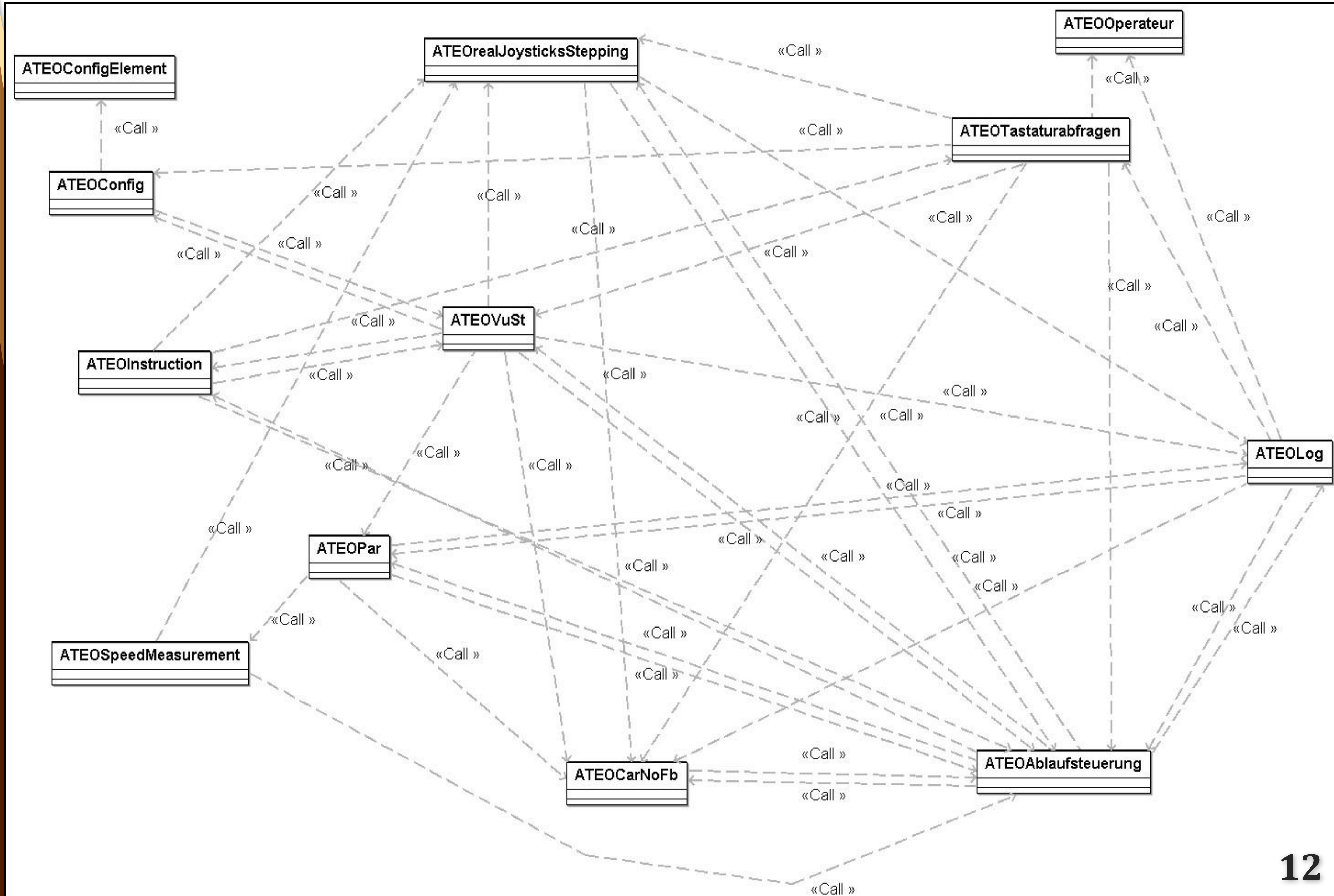
4. **Merging and Implementation**
   merging of the intermediate results

1. Reverse Engineering

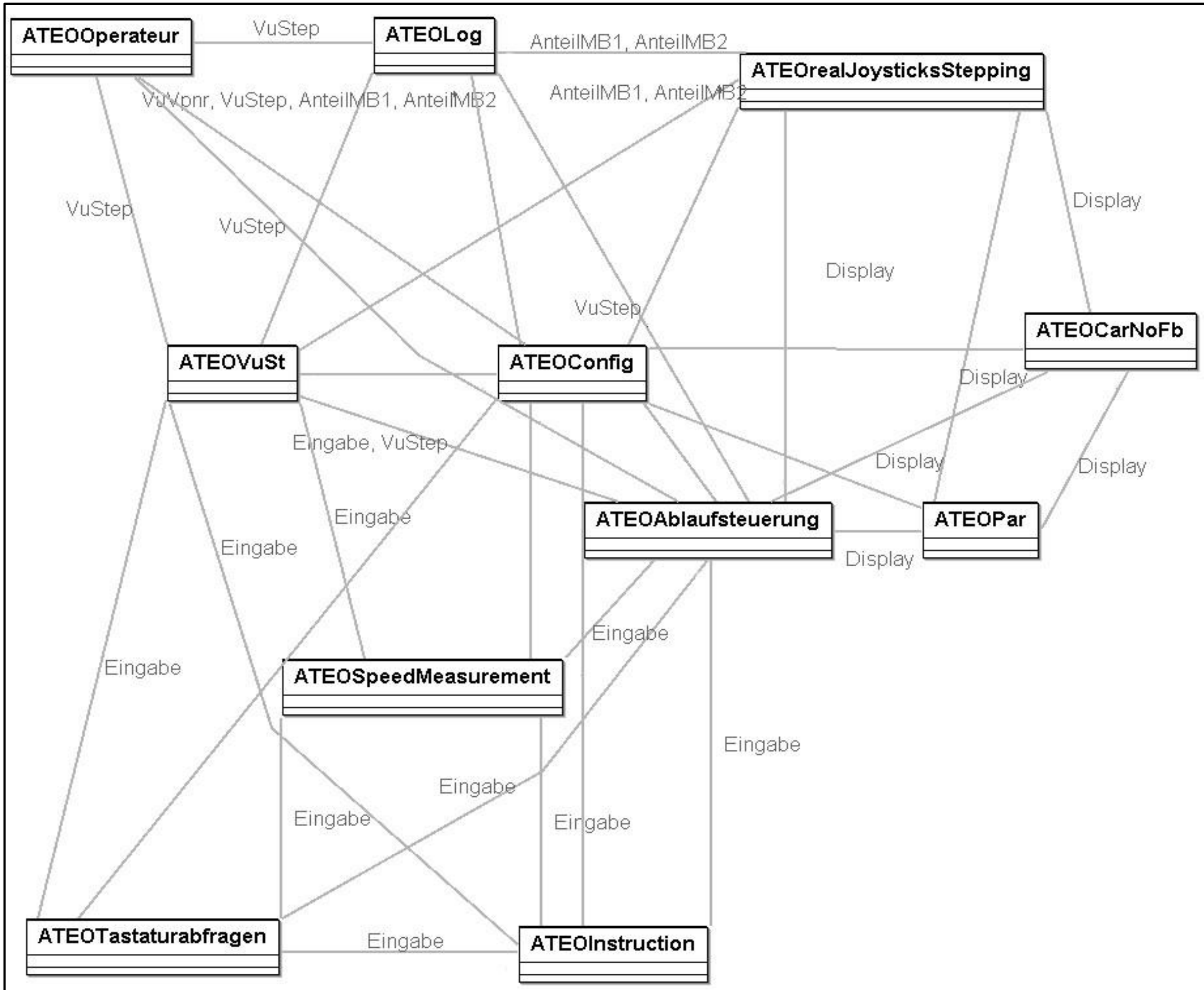
- **Reverse Engineering** of
  - **Requirements**:
    software specification (Use Cases etc)
  - **Design:**
    architecture (diagrams)
  - **Implementation:**
    code comments, class descriptions

- **further analysis** (tool based)
  - extraction of hidden dependencies between classes (via globals)
  - modeling call dependencies as a directed graph
  - depth-first cycle search
  - graph coloring (identifying SCCs)

SAMs architecture: call dependencies



SAMs architecture: hidden dependencies



Identified Central Issues

- **modularization / structure**
  - 1 layer, 1 package, 12 classes, 45 dependencies
  - no design patterns applied
  - no separation of Model, View and Control

- **cyclomatic dependencies**
  - 56 (simple) cyclomatic dependencies
  - 10 classes are on a strongly connected component (SCC)

- **global variables**
  - 25 commonly used variables
  - inducing hidden dependencies

- **outcome: very low maintainability, heavy impact on**
  - understandability
  - reusability
  - changeability
  - testability

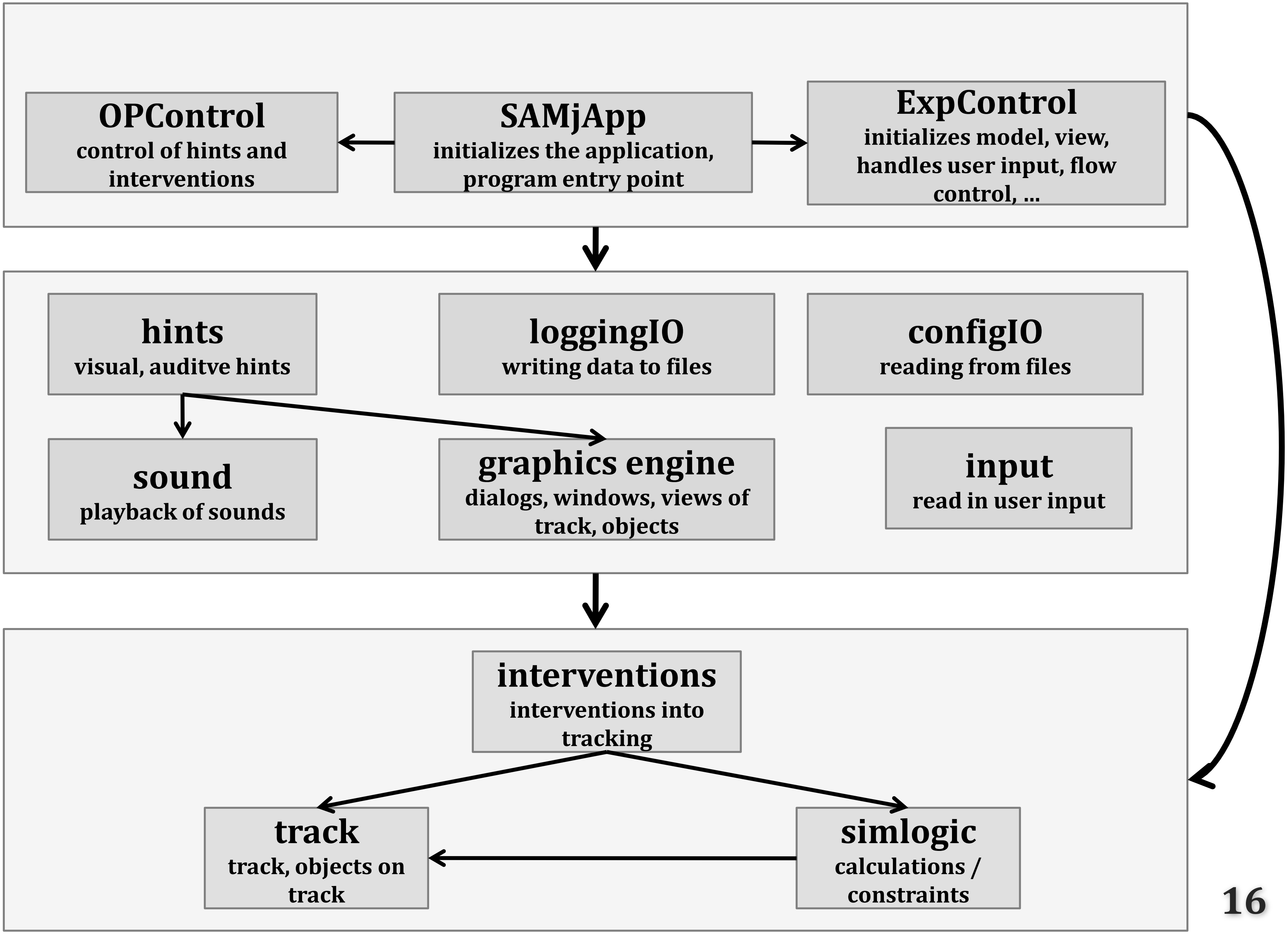
2. Restructuring

- **transformation** of the legacy architecture
  - into a layered architecture (while keeping functionality)
  - decomposition and arranging of the classes to the layers

- **based on**
  - results of Reverse Engineering: central issues
    - no cycles, no global variables, proper modularization
  - application of architecture principles / patterns
    - loose coupling, high coherence
    - separation of concerns / modularization
    - self-documentation
    - …

- **result**
  - first proposal for a layered architecture of SAMj 2.0

2. Restructuring: SAM 2.0

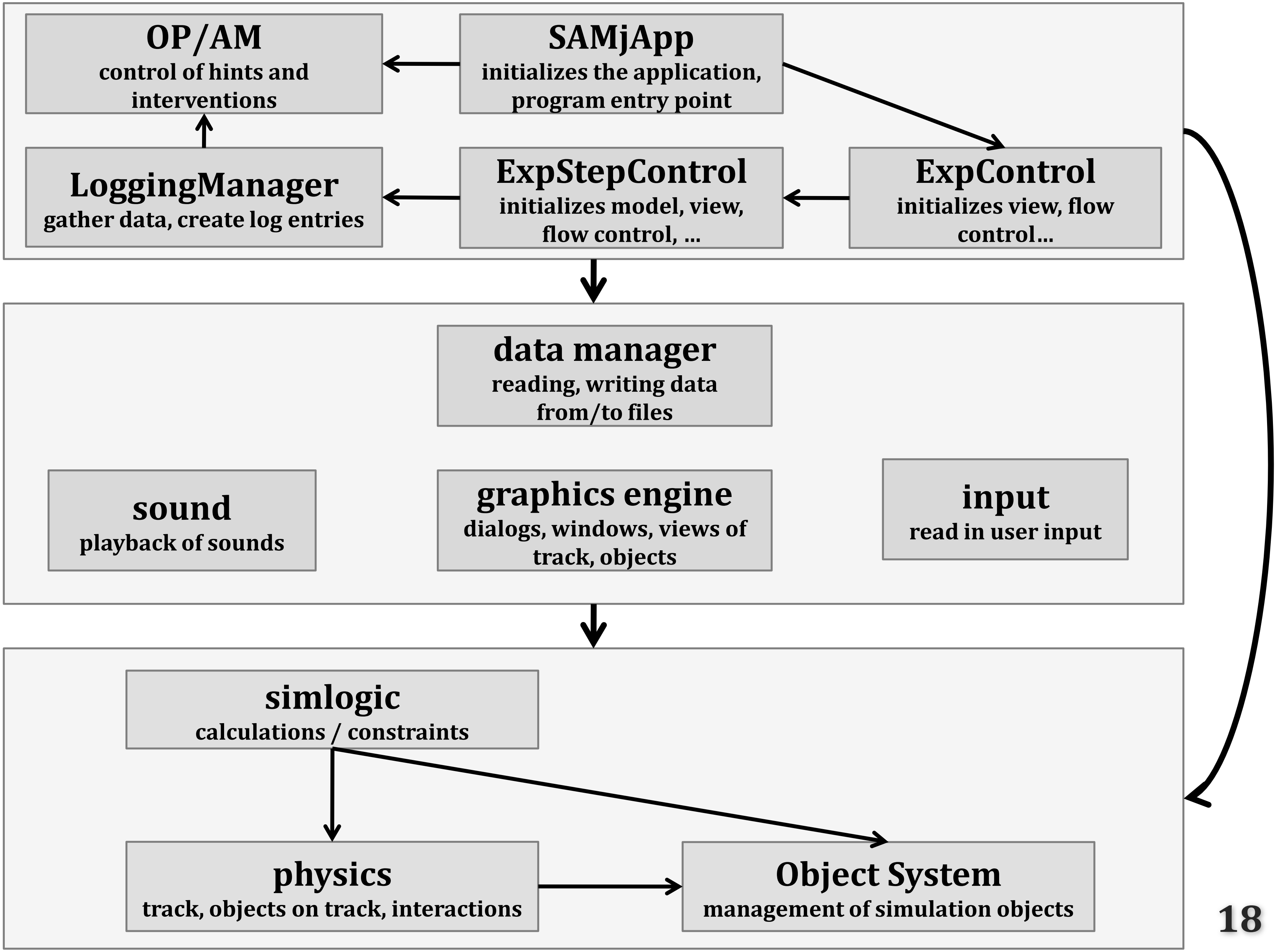


3. Forward Engineering

- building the **domain model**
  - from the reverse engineered requirements
  - performing OOA
    - deriving use cases, finding packages
    - identifying classes, methods, attributes, associations, …

- building the **architecture**
  - from domain model
  - performing OOD
    - designing view, control
    - redesigning model (if needed)
    - connecting layers
  - consideration of architecture patterns / principals

3. Forward Engineering (cont.)

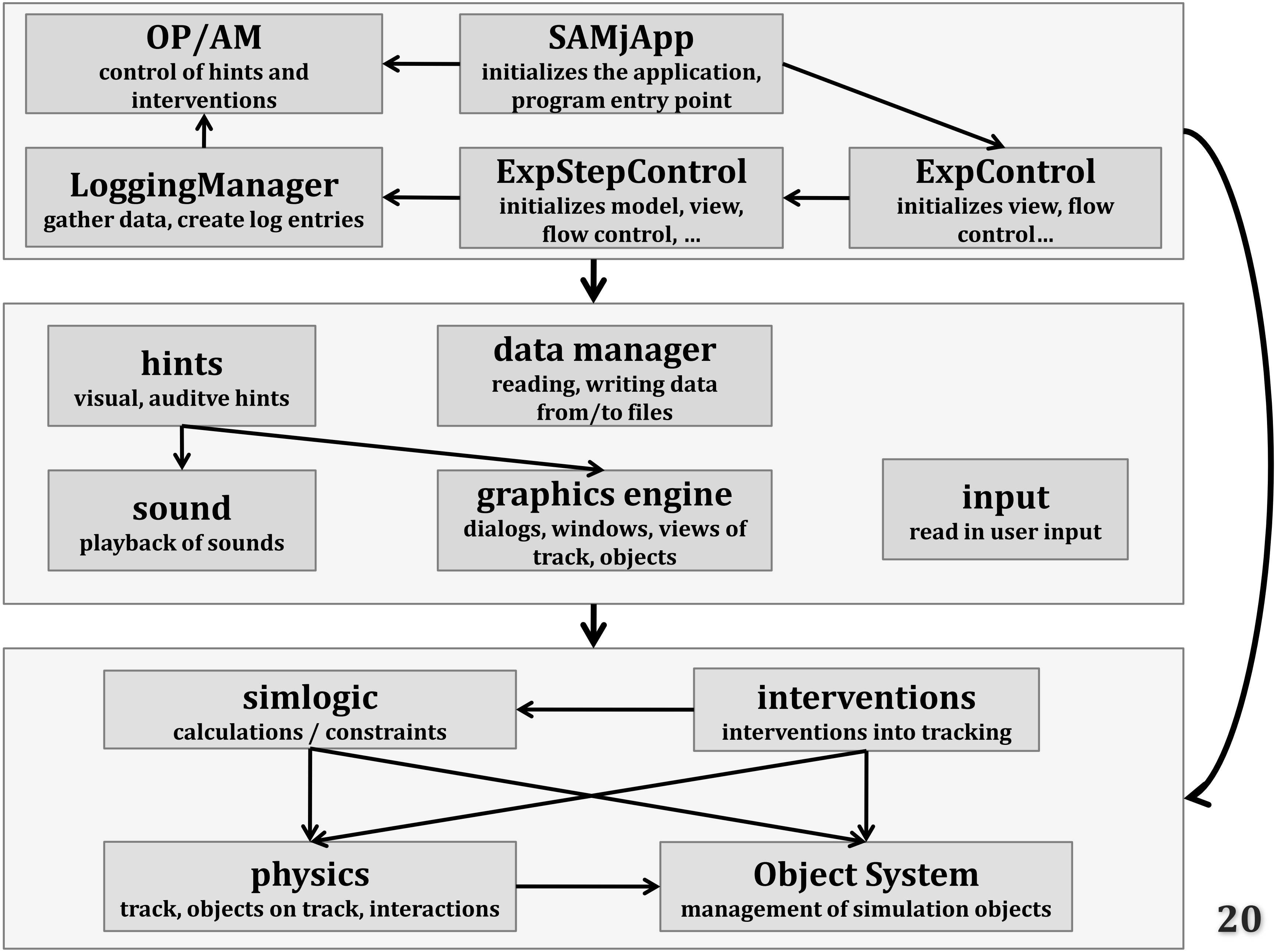


4. Merging and Implementation

- Architecture proposals are very similar
  - mainly in the formed classes
  - bigger components alsmost the same
  - Forward engineered architecture was more refined

- merged architecture
  - model and view layers were merged by combining the design ideas of both proposals
  - control layer was took from the forward engineered proposal
    - subsumed the control layer of restructured proposal

Architecture of SAMj



COMPARISION

Comparision: architectures

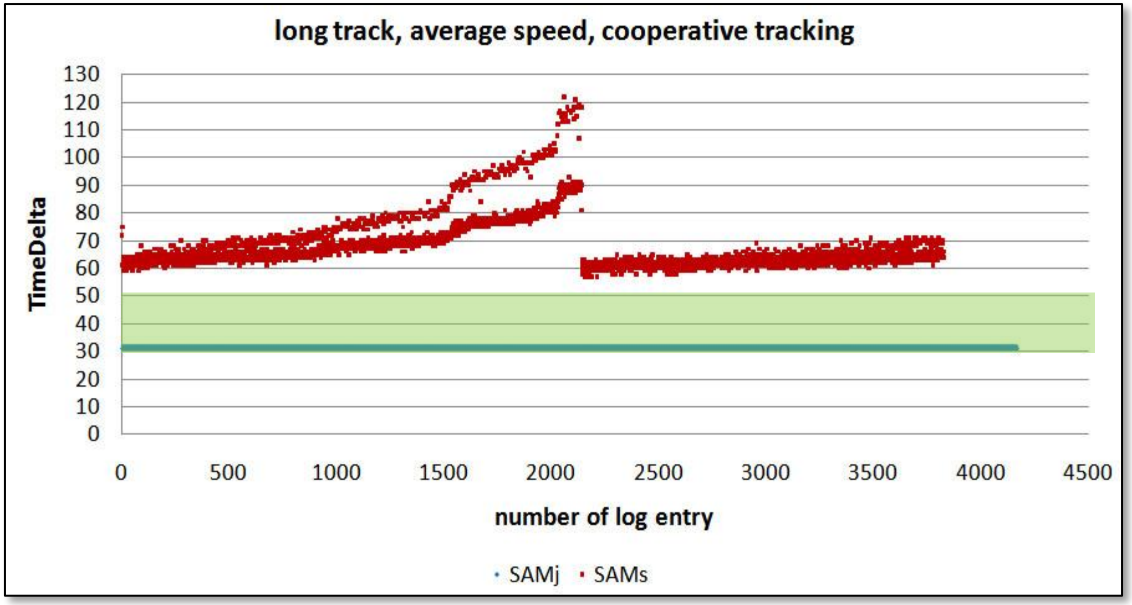
- **SAMs**
  - bad modularization (layers: 1, packages: 1)
  - no seperation of concerns
  - central issues
    - cycles: 56, global variables: 25, bad problem decomposition

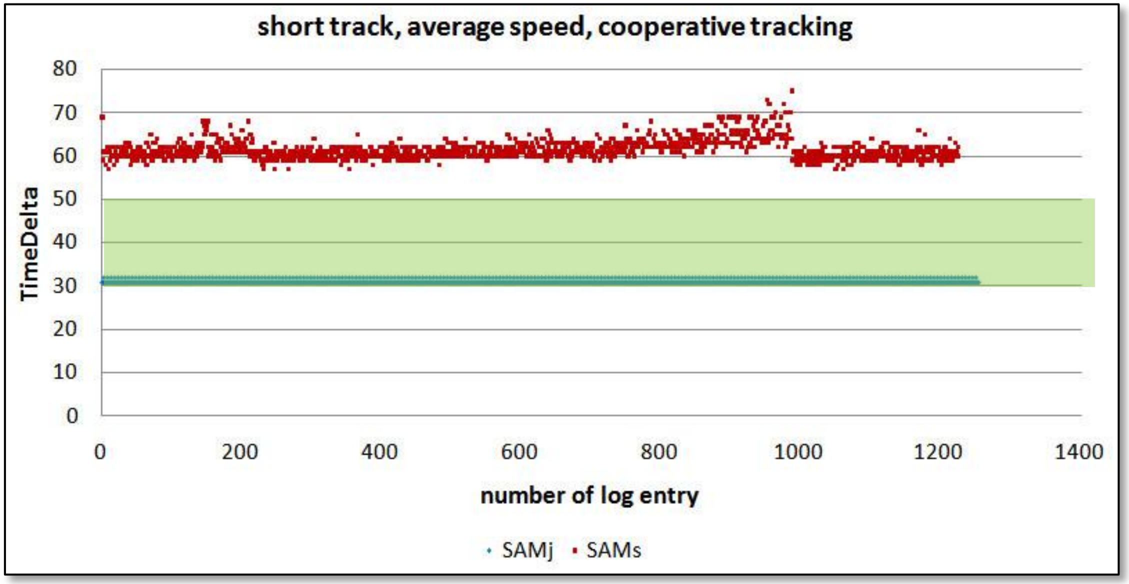
- **SAMj**
  - layered architecture (layers: 3, packages: 15)
  - designed according to architecture principals
  - solved central issues
    - cycles: 0,
    - global variables: 0,
    - better modularization / decomposition

- **result:** improvement of
  - understandabililty
  - reusability
  - changeability
  - testability

Comparision: implementations

- **Performance**
  - 8 test runs
    - length
    - mode of tracking
    - speed
  - **indicator**: *TimeDelta*
  - desired interval: [30,49) ms
  - **results**
    - SAMj: [31, 32], (31.2 ± 0.34) ms
    - SAMs: [57, 178], (66.5 ± 4.04) ms





Comparision: testing

- unit and integration tests
  - SAMs (in use)
    - 7 test classes (+ some stubs)
    - test cases only for SAMs 1.5
    - partially minor quality
    - no coverage measures known (lack of utilities)
  - SAMj (prototype)
    - 20 test classes
    - coverage measures
      - statement coverage: **96,07 %**
      - branch coverage: **89,95 %**
      - Simple condition coverage: **84,10 %**
      - Multiple condition coverage: **81,68 %**

Summary

- analysis and documentation of the SAMs architecture
- development of an improved architecture
  - hierarchical layer architecture
  - improved quality properties
- implementation of a prototype in java
  - improved performance
  - quality assurance: unit and integration tests
- comparision of variantes

THAT‘S IT!

Questions?

Hints?

Additions?