Experimental software risk assessment

Henrique Madeira
University of Coimbra, DEI-CISUC
Coimbra, Portugal

Universidade de Coimbra

Component-based software development

- **Vision:** development of systems using pre-fabricated components. Reuse custom components or buy software components available from software manufactures (Commercial-Off-The-Shelf: COTS).

- **Potential advantages:**
  - Reduce development effort since the components are already developed, tested, and matured by execution in different contexts
  - Improve system quality
  - Achieve of shorter time-to-market
  - Improve management of increased complexity of software

- **Trend** → **use general-purpose COTS components and develop domain specific components**.

Some potential problems

- COTS
  - In general, functionality descrition is not fully provided.
  - No guarantee of adequate testing.
  - COTS must be assessed in relation to their intended use.
  - The source code is normally not available (makes it impossible white box verification & validation of COTS).
- Reuse of custom components in a different context may expose components faults.

Using COTS (or reusing custom components) represent a risk!
How to assess (and reduce) that risk?

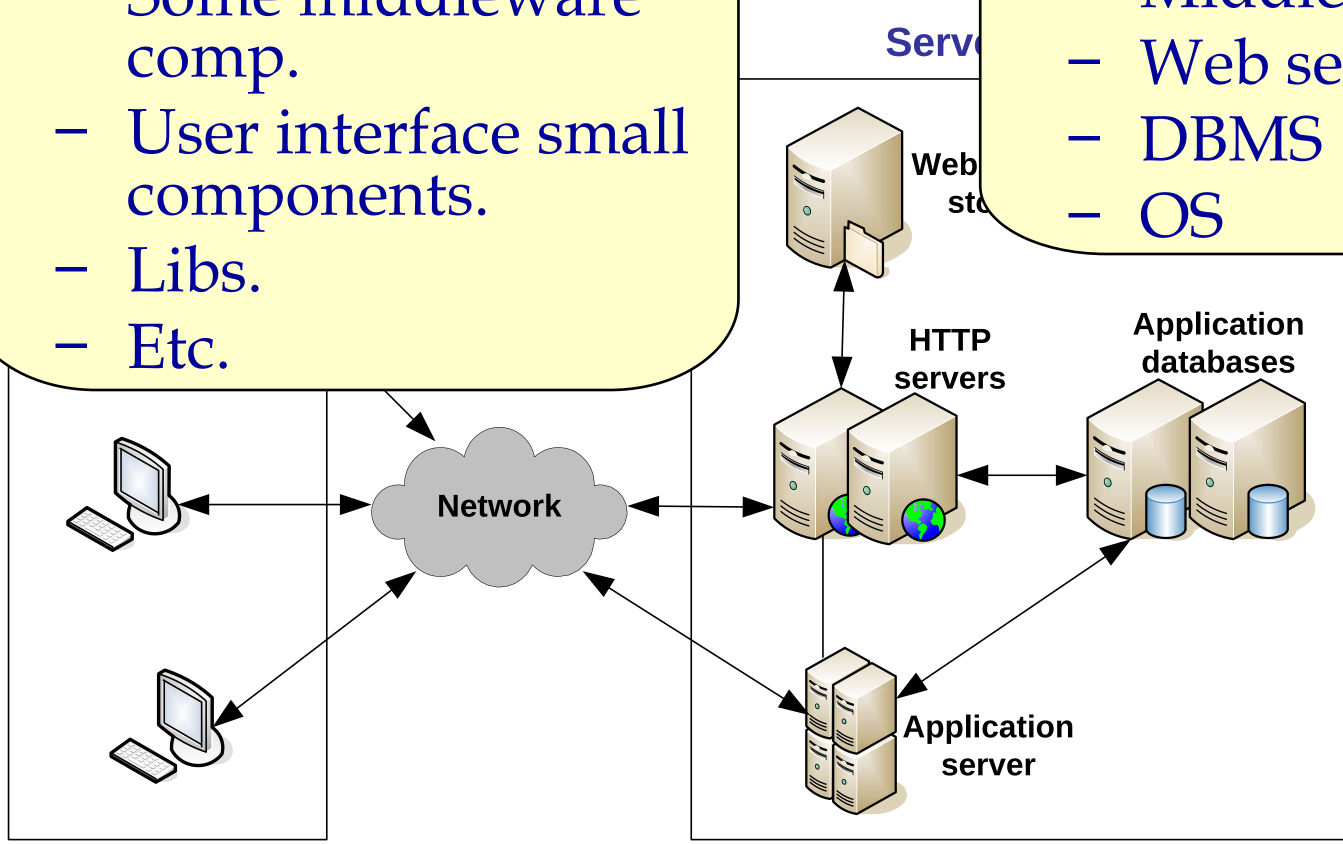
A real example:
COTS in very large scale systems

Fine grain COTS:

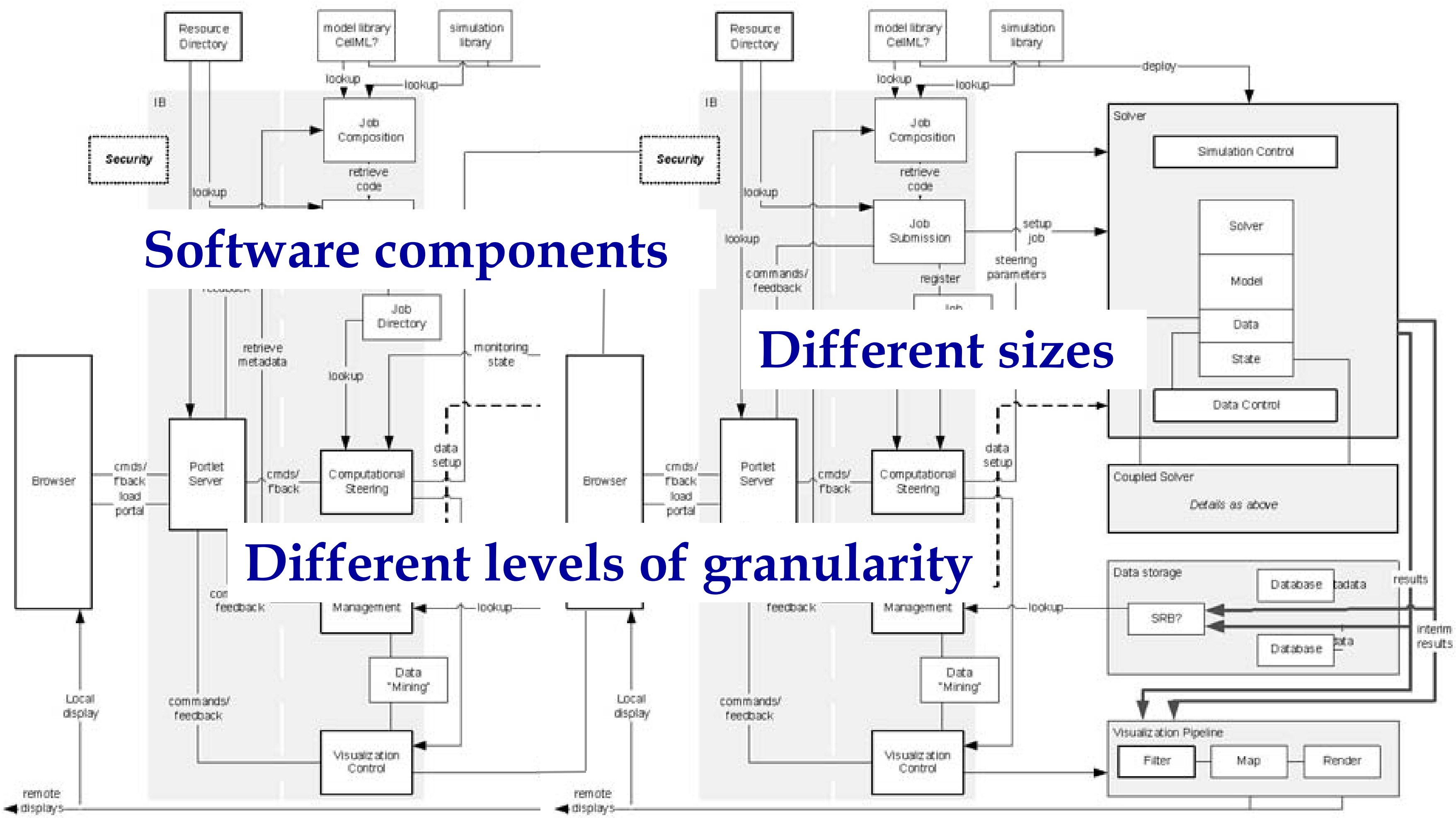
- Some middleware comp.
- User interface small components.
- Libs.
- Etc.

Coarse grain COTS:

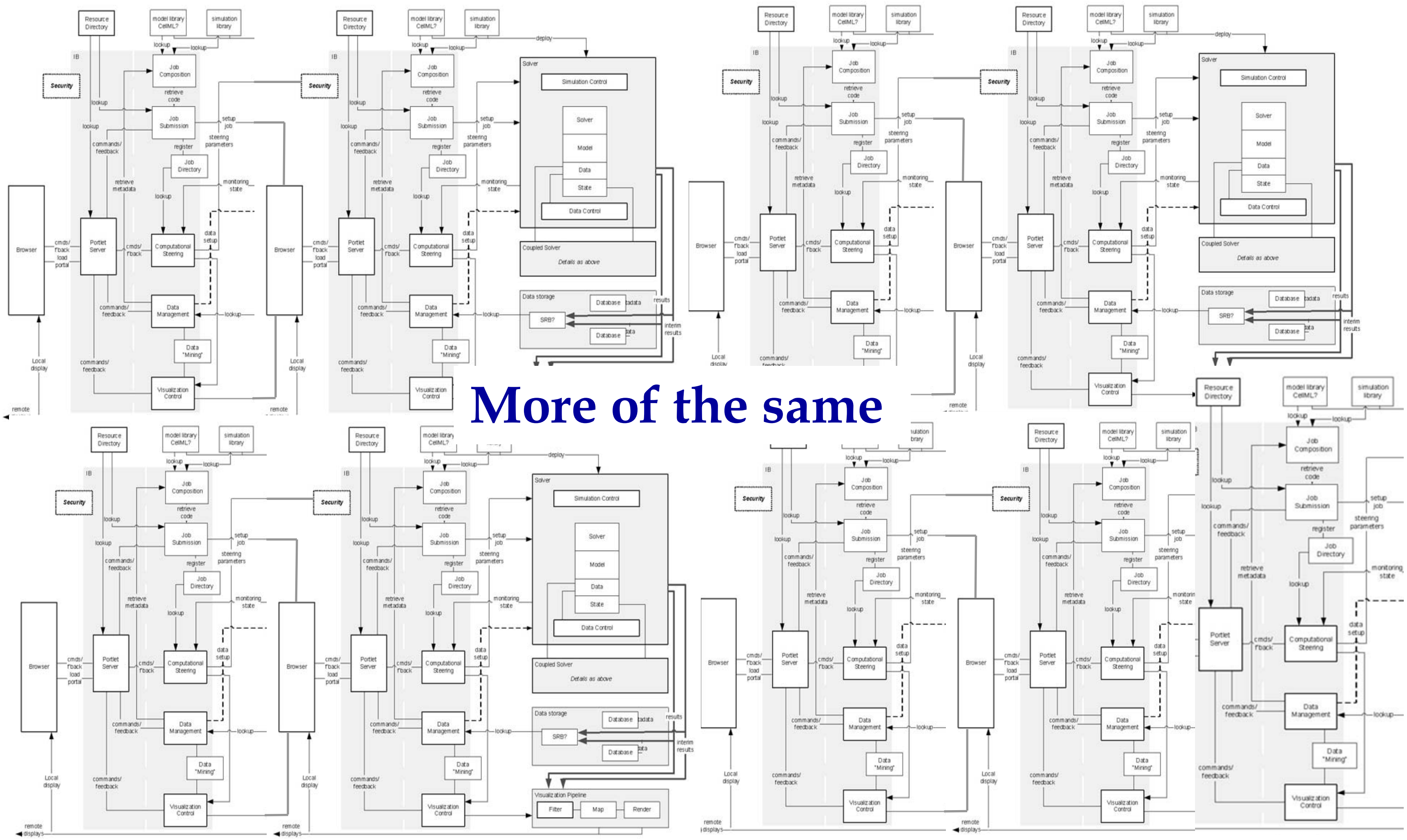
- Middleware comp.
- Web servers
- DBMS
- OS



Case-study 1: *I-don’t-care-about software architecture diagram*



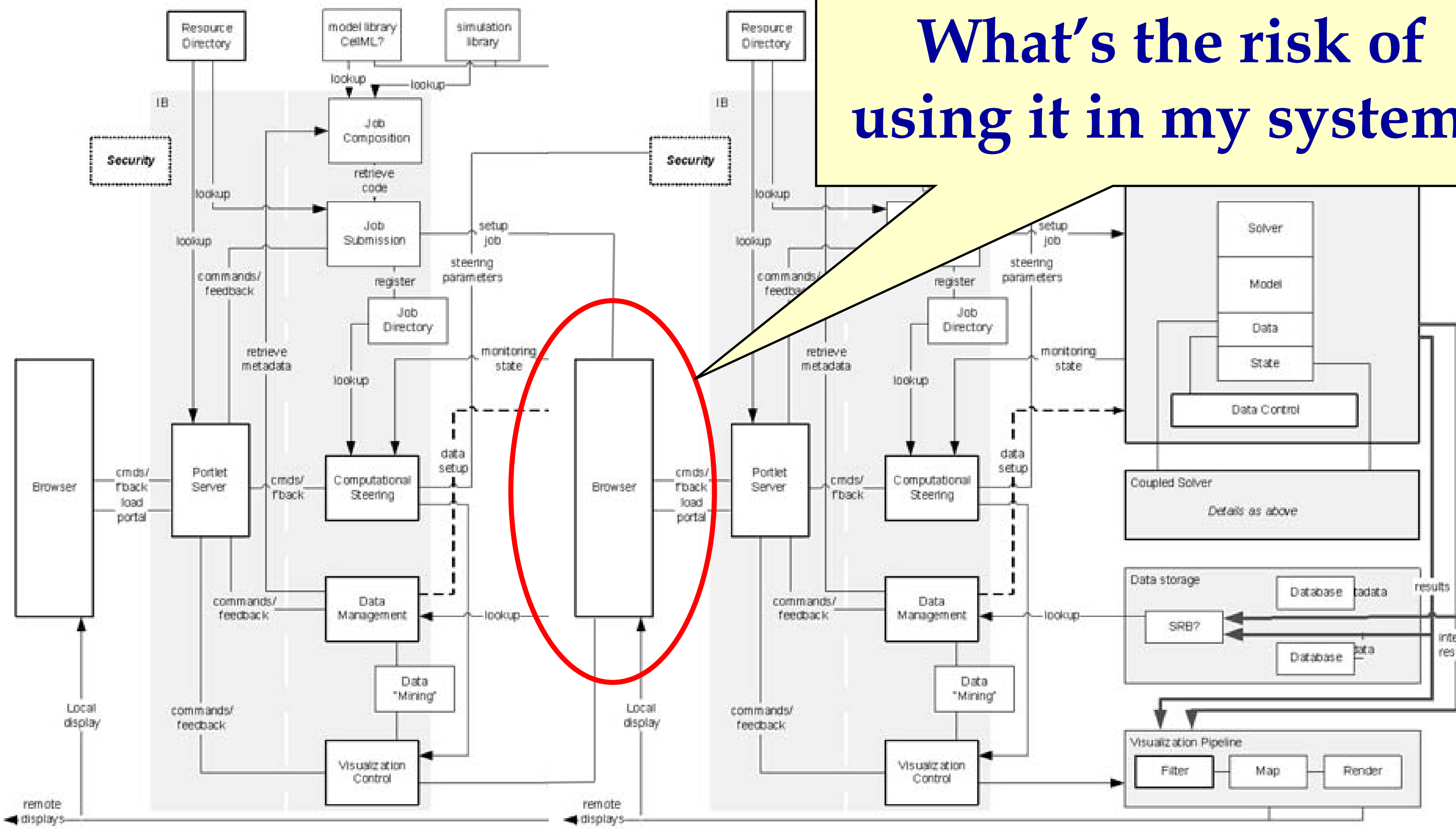
Case-study 2: *I-really-don’t-care-about software architecture diagram*



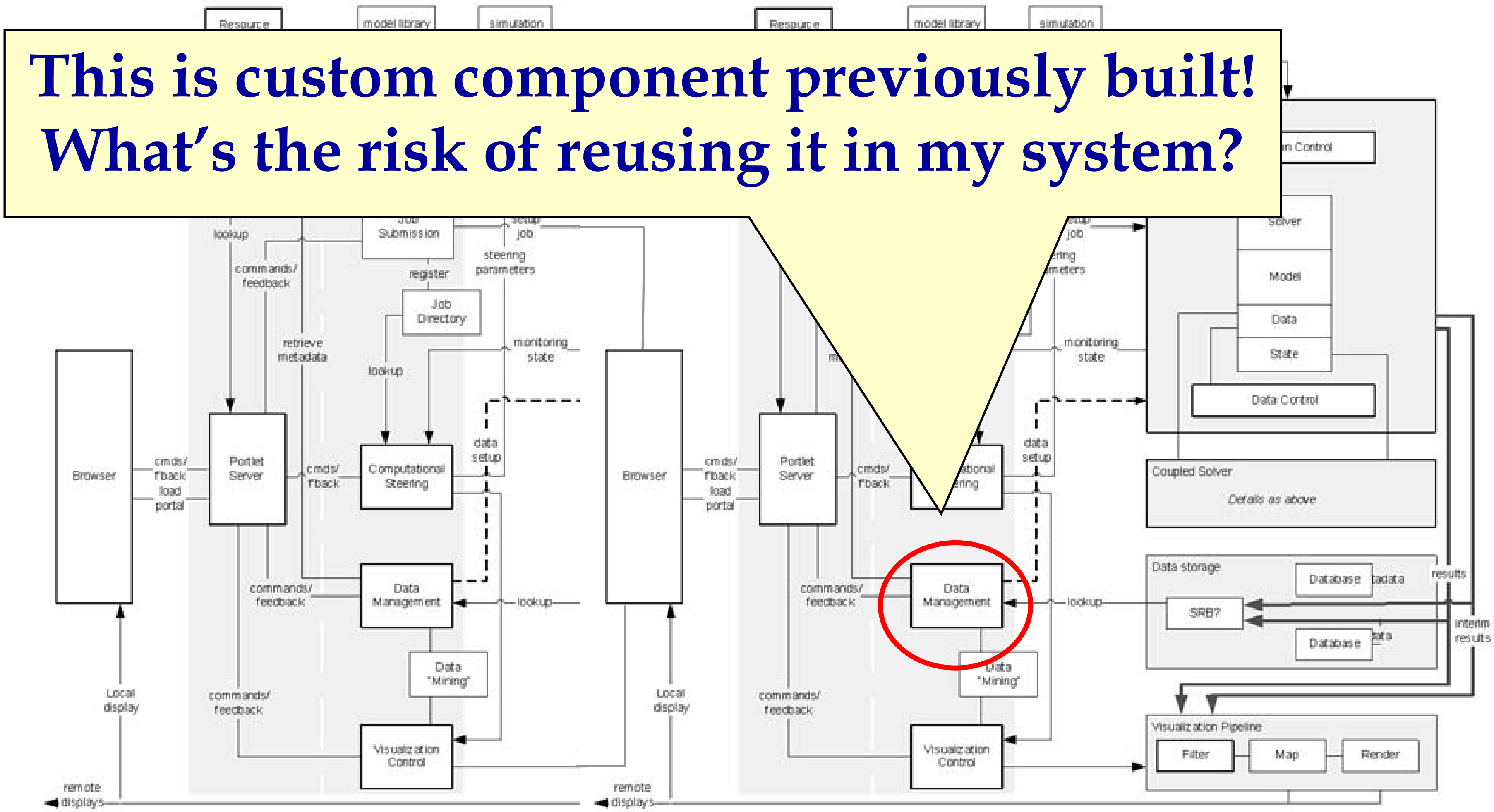
More of the same

Question 1

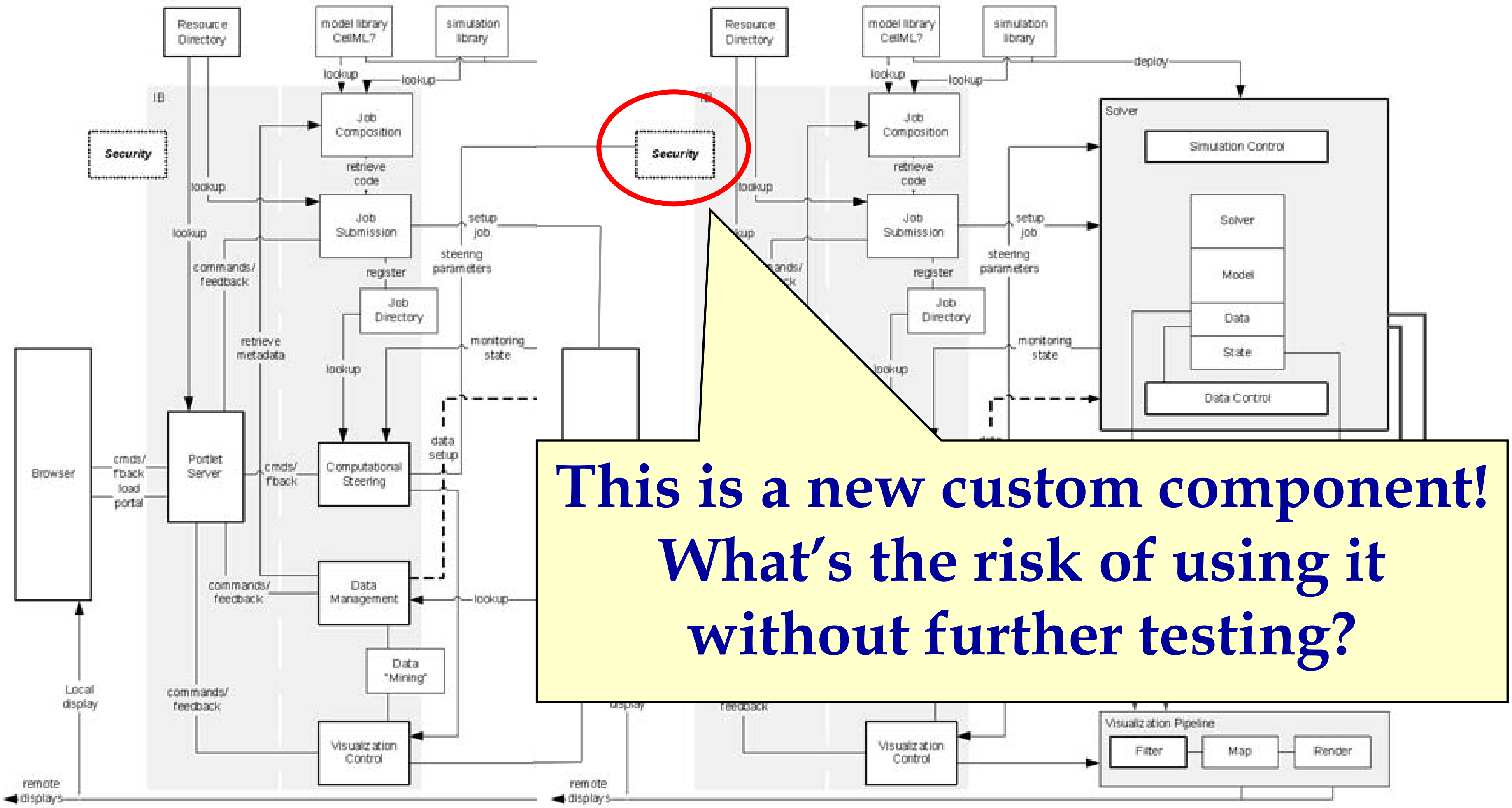
This is a COTS!
What's the risk of using it in my system?



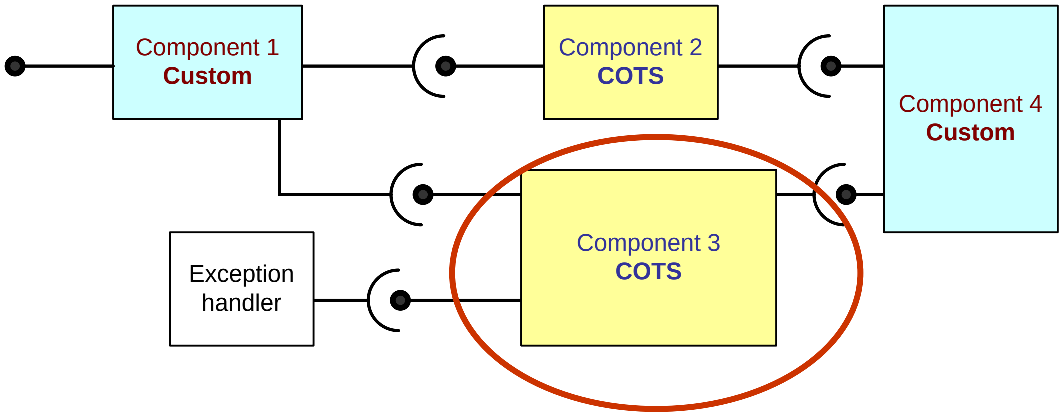
Question 2


This is custom component previously built!
What's the risk of reusing it in my system?
Browser
cmds/ f'back load portal
Portlet Server
cmds/ f'back
Computational Steering
Data Management
Data "Mining"
Visualization Control
Job Submission
Job Directory
lookup
commands/ feedback
retrieve metadata
register
steering parameters
monitoring state
data setup
lookup
Local display
remote displays
Data Control
Coupled Solver
Details as above
Data storage
Database
SRB?
results
interim results
Visualization Pipeline
Filter
Map
Render
Solver
Model
Data
State

Question 3


Resource Directory
model library CellML?
simulation library
lookup
Job Composition
Security
retrieve code
Job Submission
setup job
steering parameters
register
Job Directory
monitoring state
Browser
Portlet Server
Computational Steering
data setup
Data Management
Data "Mining"
Visualization Control
Local display
remote displays
deploy
Solver
Simulation Control
Model
Data
State
Data Control
Visualization Pipeline
Filter
Map
Render
This is a new custom component!
What's the risk of using it without further testing?

Experimental risk assessment



Example of question:

What's the risk of using Component 3 in my system?

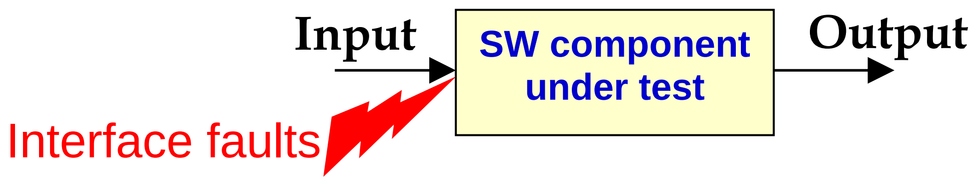
Risk = prob. of bug * prob. of bug activation * impact of bug activation

Software complexity metrics

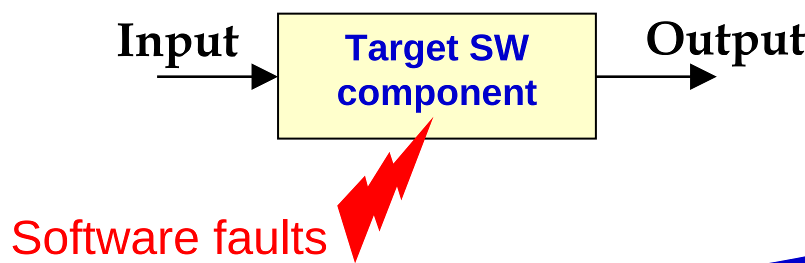
Injection of software faults

Two possible injection points

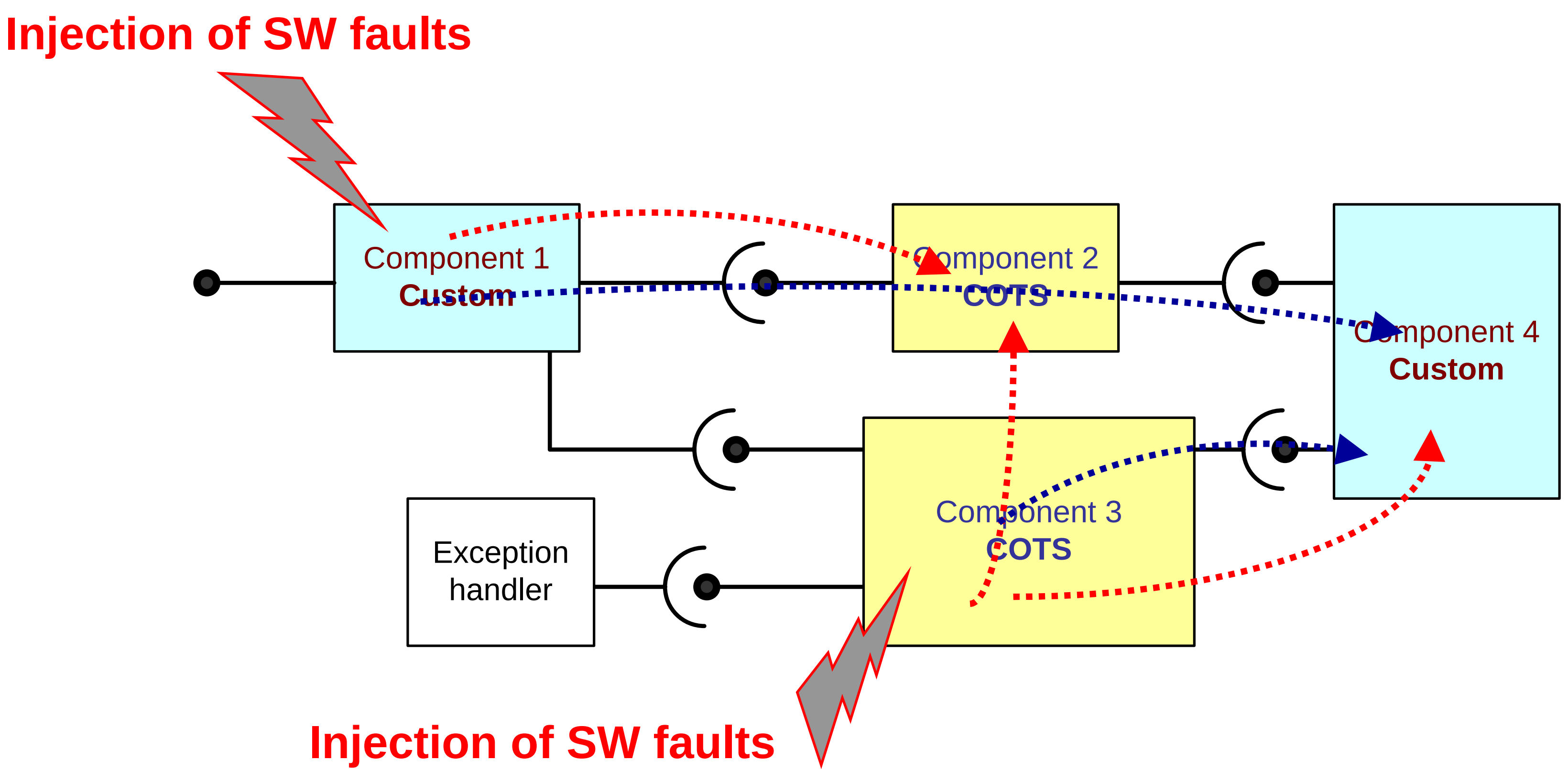
1. Injection of interface faults in software components (classical robustness testing: Ballista, Mafalda, …)



2. Injection of **realistic** software faults inside software components (new approach)



Why injection or real software faults?

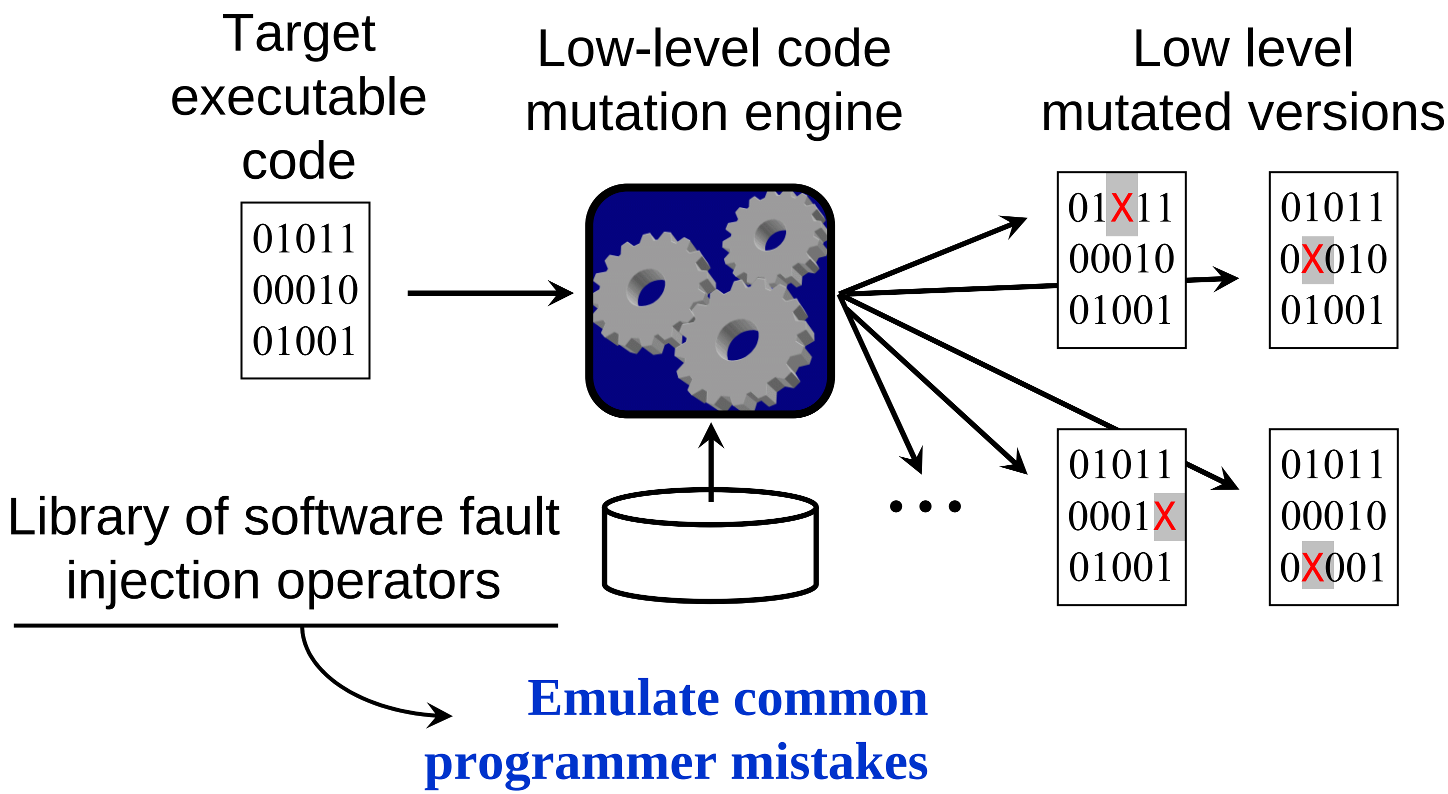


- Error propagation through non conventional channels is a reality.
- Faults injected inside components are more representative.

How to inject software faults?

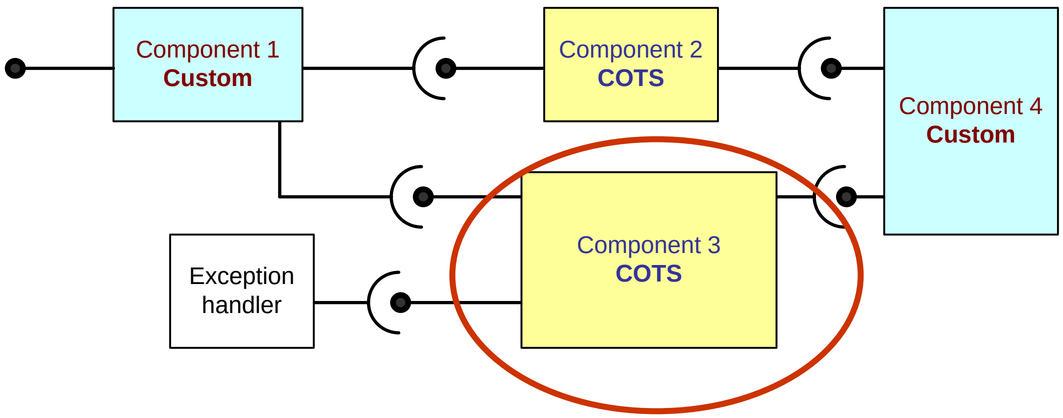
- **Use G-SWFIT (ISSRE 2002, DSN 2003, DSN 2004)**
  - Injects the **top N** most common software faults.
  - This top N is based on field data (our study + ODC data from IBM) and corresponds to ~65% of the bugs found in field data.
  - Injects faults in executable code.
  - Largely independent on the programming language, compiler, etc that have generated the executable code.
- **G-SWFIT is now a reasonably mature technique.**

G-SWFIT
Generic software fault injection technique



The technique can be applied to binary files prior to execution or to in-memory running processes

Experimental risk assessment (again)



Example of question:

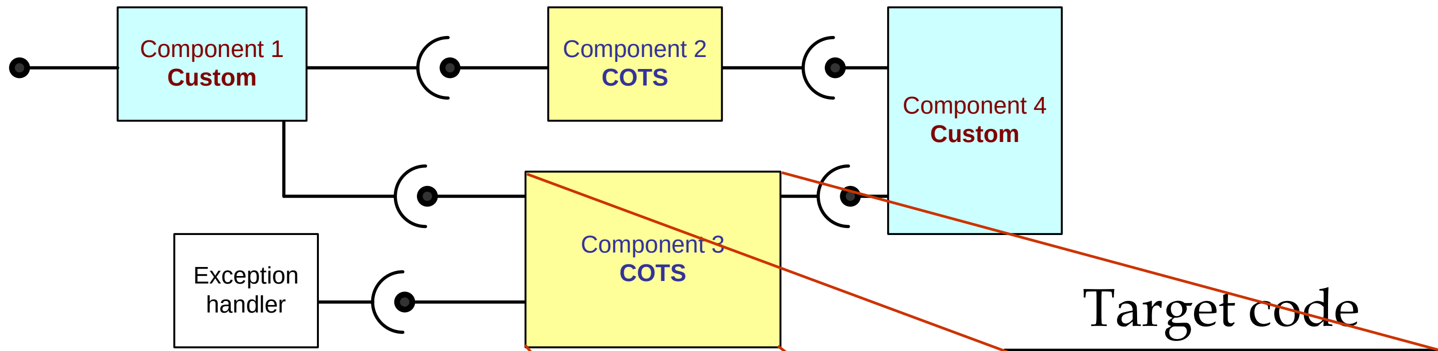
What's the risk of using Component 3 in my system?

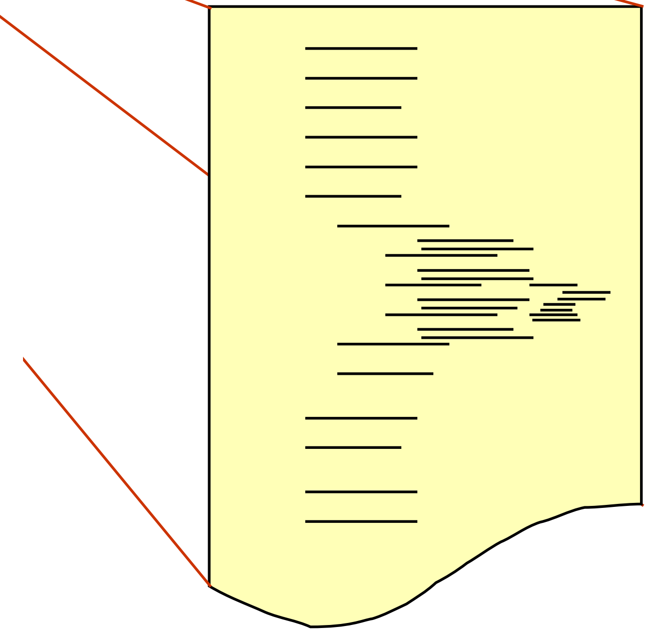
Risk = prob. of bug * prob. of bug activation * impact of bug activation

Software complexity metrics

Injection of software faults

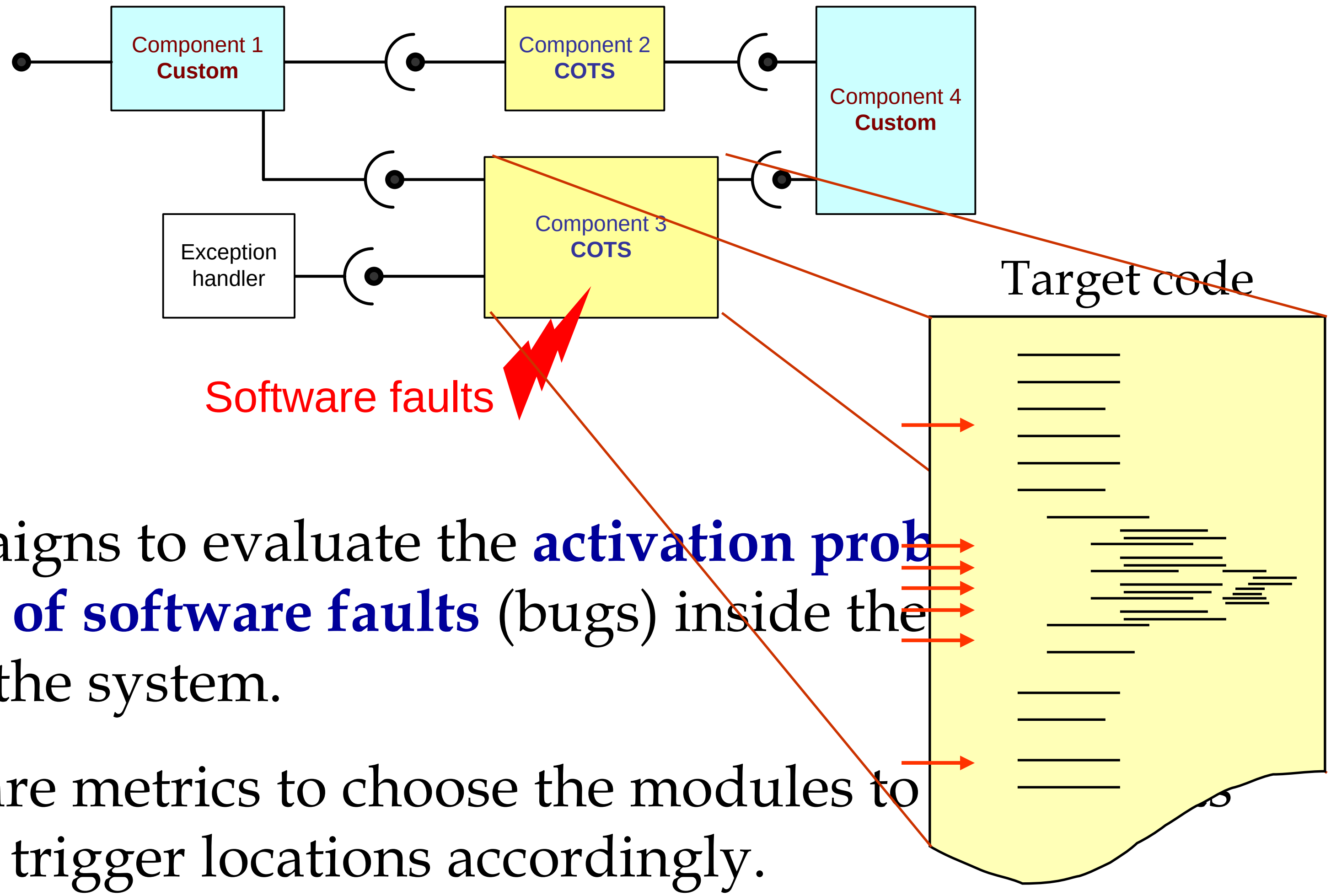
Estimation of the probability of residual bugs





- Many studies indicate that fault probability correlates with the software module complexity
- Metrics of software complexity base on:
  - Static feature of the code;
  - Dynamic features;
  - Possible information on the development process (type of tests, etc);
  - ...

Estimation of bug activation probability and bug impact



- Test campaigns to evaluate the **activation prob** the **impact of software faults** (bugs) inside the the rest of the system.
- Use software metrics to choose the modules to s and define trigger locations accordingly.

Conclusions and current work on experimental risk assessment

- Experimental software risk assessment seems to be viable.
- Risk is a multi-dimensional measure. Many software risks can be assessed, depending on the property I'm interested in.
- Current work:
  - Improve the G-SWFIT technique:
    - Improving current tool.
    - Expansion of the mutation operator library
    - Construction of a field-usable tool for software fault emulation in Java environments
  - Study of software metrics and available tools.
  - Define a methodology for experimental software risk assessment.
  - Real case-studies to demonstrate the methodology.