Verification and Validation (PA2405)

Lecture 2: Static Validation

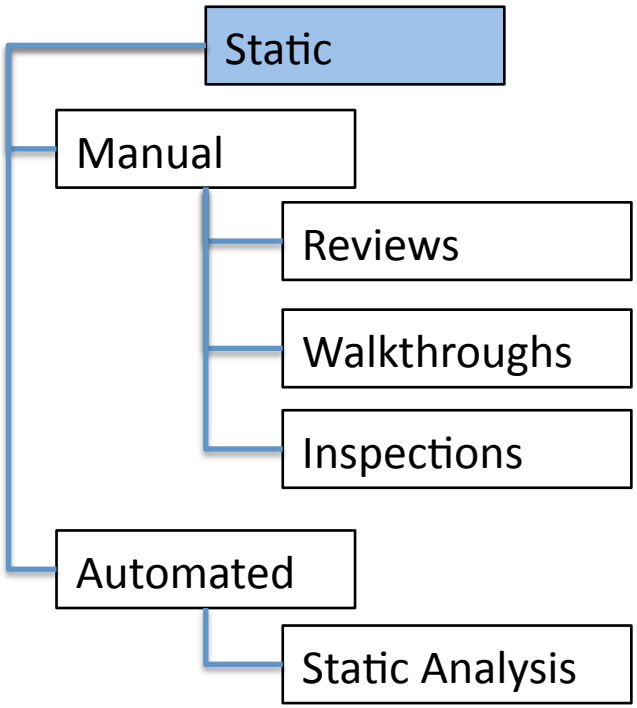
Static vs. Dynamic Validation

- Static validation
  - Examination of a software artifact (e.g. requirement, design, code, test cases, etc.) manually without execution to identify defects

- Dynamic validation
  - Execution of a software artifact (e.g. method, class, component, sub-system, system) to identify defects

- Should we do one or the other?

Overview of Static Approaches

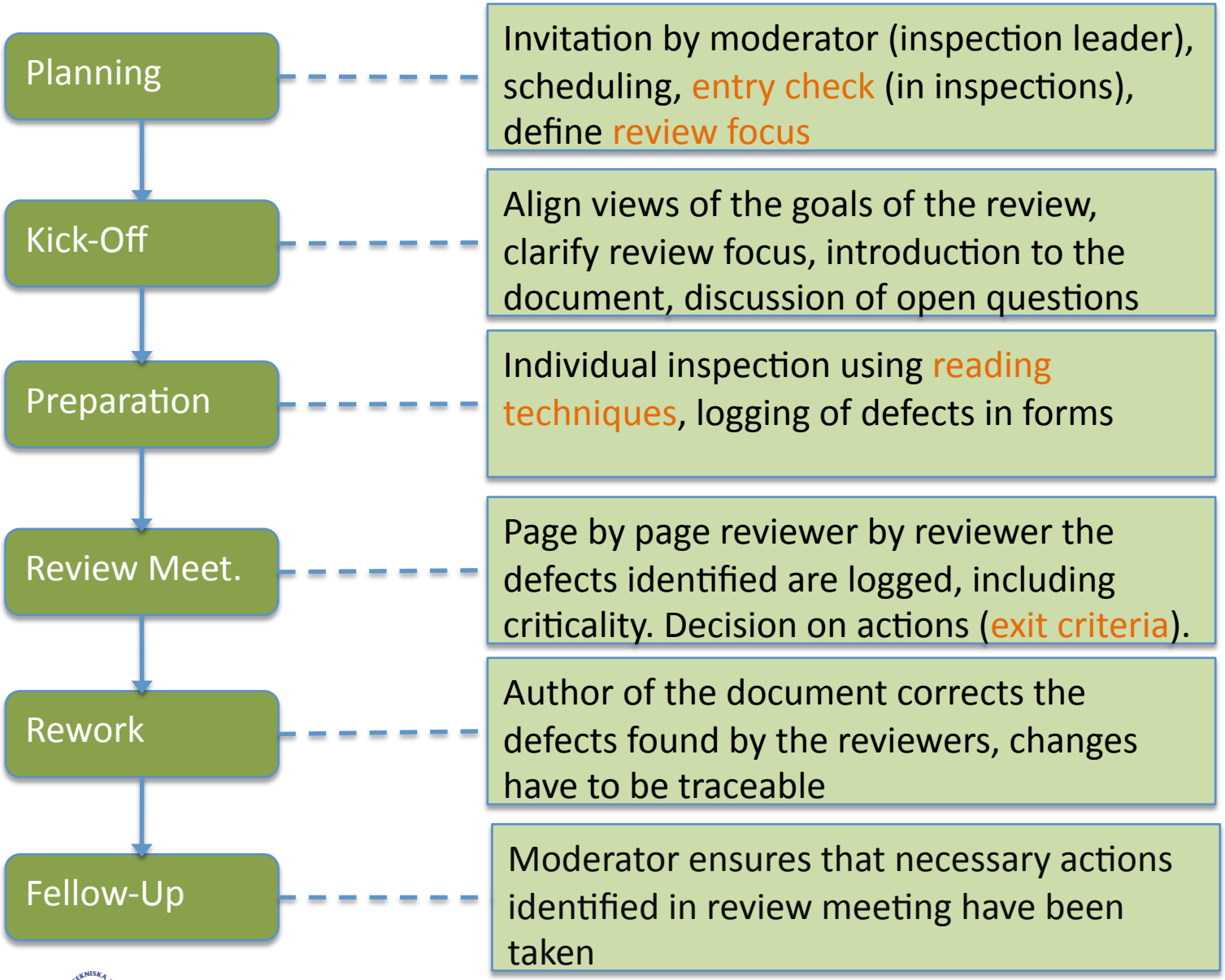


Manual Reviews come in different flavors (ranging from very informal to very formal)

- Informal Reviews: Most common in industry, not a documented process. Examples: Ask colleague to review a document, review a document with a pair of people.

- Formal Reviews: Documented review process with defined goals, process steps, and roles. Examples: Fagan inspections, technical review, walkthrough

Inspection Process (based on Fagan Inspection)



Roles

Role	Description
Moderator	Leads review process, performs entry check, control of quality of the review process, schedules meetings, makes sure data is collected properly
Author	Writer of the document under review, responsible for improving the quality based on the review feedback
Scribe	Records defects and improvements during the review meeting (Moderator/Author can be scribes)
Reviewers	Reviewers conduct individual checks and present the results during the review meeting. Reviewers can have different roles (tester, developer, manager) and hence provide different viewpoints
Manager	Manager supports review process by providing schedulable time for review, manager can also be a reviewer when having sufficient background

Overview of approaches

Attribute	Walkthrough	Inspection	Technical Review
Meeting of participants	Author of document presents, walks audience through the process	Moderator presents shortly, well prepared reviewers provide info about defects in meeting	Group reviews document together (less formal process in meeting)
Goals	Gain common understanding, knowledge transfer, discuss validity of proposed solutions	Improve document quality, remove defects efficiently and as early as possible, training and common understanding	Assess value of technical concepts, establish consistency in technical concepts, inform participants
Advantage	Large number of people can be present and be informed	More formal/systematic and hence repeatable (i.e. defect analysis and prediction becomes an option)	Less formal/systematic than inspections and hence very quick, easily accepted by practitioners
Disadvantage	Important information is left out, audience does not understand document on their own, not suited to find significant problems	Formal and high investment process (might not be suitable for every domain, but needed in some – e.g. safety critical)	Lack of repeatability due to lack of formality

Reading Techniques – Three Example Techniques

- Checklist-based reading
  - Every item of the list should be checked for the document under inspection

- Perspective-based reading
  - Different reviewers examine the document from their perspective (e.g. customer, designer, tester)

- Usage-based reading
  - Inspection based on whether the document under inspection fulfills usage scenarios

Reading Techniques (Checklist)

- Example items of a code inspection items:
  - Do the type attributes of actual and formal parameters match?
  - Are global variable definitions and usage consistent among modules?
  - Has each field been initialized before it is first used?
  - Are allocated resources used for an object freed up after usage?
  - Are modules independent of other modules?
  - etc.
- Examples for requirements inspection items:
  - Have the relevant users and stakeholders of the system been identified?
  - Was a rational stated for each requirement?
  - Has each actor (different types of users) been identified?
  - Is there at least one use case for each actor?
  - Does a prioritization of requirements exist?
  - Is the requirement unambiguously stated?
  - Is the level of detail of the requirement sufficient for deriving test cases and as input for implementation?
  - etc.

Java Code Inspection (Checklist)

- Checklists are often grouped into issues to be checked:
  - Adherence to specification/design
  - Initialization and Declaration
  - Method Calls
  - Operation and initialization of Arrays
  - Object Comparison
  - Output Format
  - Computation, Comparisons, and Assignments
  - Exceptions
  - Flow of Control
  - Files
- Examples for Java Inspection Checklists
  - http://users.csc.calpoly.edu/~jdalbey/205/Resources/InspectChecklist.html
  - www.dgp.toronto.edu/~ppacheco/course/444/java_checklist.pdf

Reading Techniques (Perspective-Based)

Idea: Different perspectives will find different defects

Each role receives different guides during the preparation/individual inspection.
Examples: Role-specific checklists, Scenario-based reading

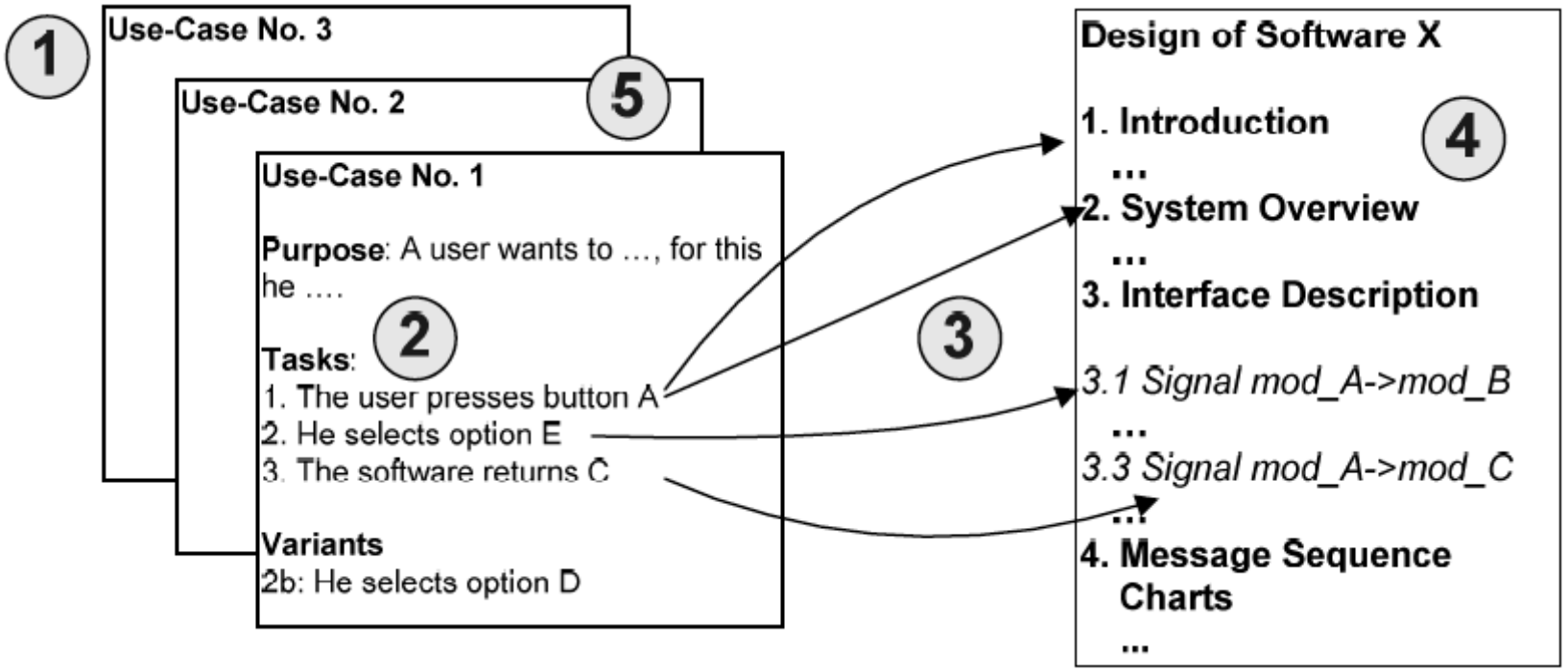
Reading Techniques (Evidence regarding Checklist-Based and Perspective-Based Reading)

- Individual studies are in conflict with each other (some say PBR is better than CBR, some say vice versa, some are inconclusive)
- Analyzing the effect size across studies the following conclusions can be drawn for team effectiveness:
  - Perspective-based reading is better than ad-hoc reading
  - Perspective-based reading is better than checklist based reading for design and code documents
  - Checklist-based reading is better than perspective-based reading for requirements documents

See: Marcus Ciolkowski, What do we know about perspective-based reading? An approach for quantitative aggregation in software engineering, ESEM 2010

Reading Techniques (Usage-Based Reading)

- Usage-based reading is a scenario-based reading technique
- Inspired by operational profile testing (i.e. test driven by usage profiles)



Reading Techniques (Evidence regarding Usage-Based and Checklist-Based Reading)

- Usage-based reading is better than checklist-based reading (Design) wrt. inspection effectiveness and efficiency
- Usage-based reading is not better than checklist based reading (Requirements) wrt. inspection efficiency

See: K. Petersen, K. Rönkkö, C. Wohlin, The impact of time controlled reading on software inspection effectiveness and efficiency: a controlled experiment, ESEM 2008, 139-148

Reading Techniques (Logging of Inspection Results)

Inspection record form

Inspection Record				
Document to inspect:	Software Top Level Design Document (STLDD)		Date:	
Vers.:		Name:		

Risk Assessment		
Risk	Interpretation	Interpretation when inspecting a design document
A	Crucial fault	The functions affected by these faults are *crucial* for the customer, i.e., the functions affected are important for the customer and are often used.
B	Important fault	The functions affected by these faults are *important* for the customer, i.e., the functions affected are either important and rarely used or not as important but often used.
C	Not important	An issue should be changed in the design, but is not an important or crucial fault.

Types of Fault		
Type	Interpretation	Interpretation when inspecting a design document
M	Missing	Some information is missing.
W	Wrong	The provided information is wrong.

Issues				
Risk	Total number of faults found:	Subjective estimation of the number of faults left after inspection: (do not include the ones that you have found)		
A:		Min:	Med:	Max:
A+B:		Min:	Med:	Max:

Reading Techniques (Data Collection)

Inspection record form (cont.)

Explanation of the Time Log	
Number:	Running number of faults you have found. Start with 1, then 2 and so on.
Session:	The Session taken from the Time Log.
Item Number:	inspectors fill in the UseCase Number that was utilized when the fault was found.
Clock Time	Minute when you found the fault.
Position:	Position(s) where you found the fault. For example "section 1.2.1", or "Table 1".
Risk:	"A", "B", or "C".
Type of Fault:	Write 'M' if information is missing or 'W' if the information in the design is wrong.

Time Log

Session		Start Clock Time hh:mm	End Clock Time hh:mm	Description
1				Reading through the requirements specification.
2				Reading through the design document.
3				Inspection
4				
5				
6				
7				
Total (minutes) ____ - breaktime ____ =				Net duration of inspection in minutes.

Inspection Issues

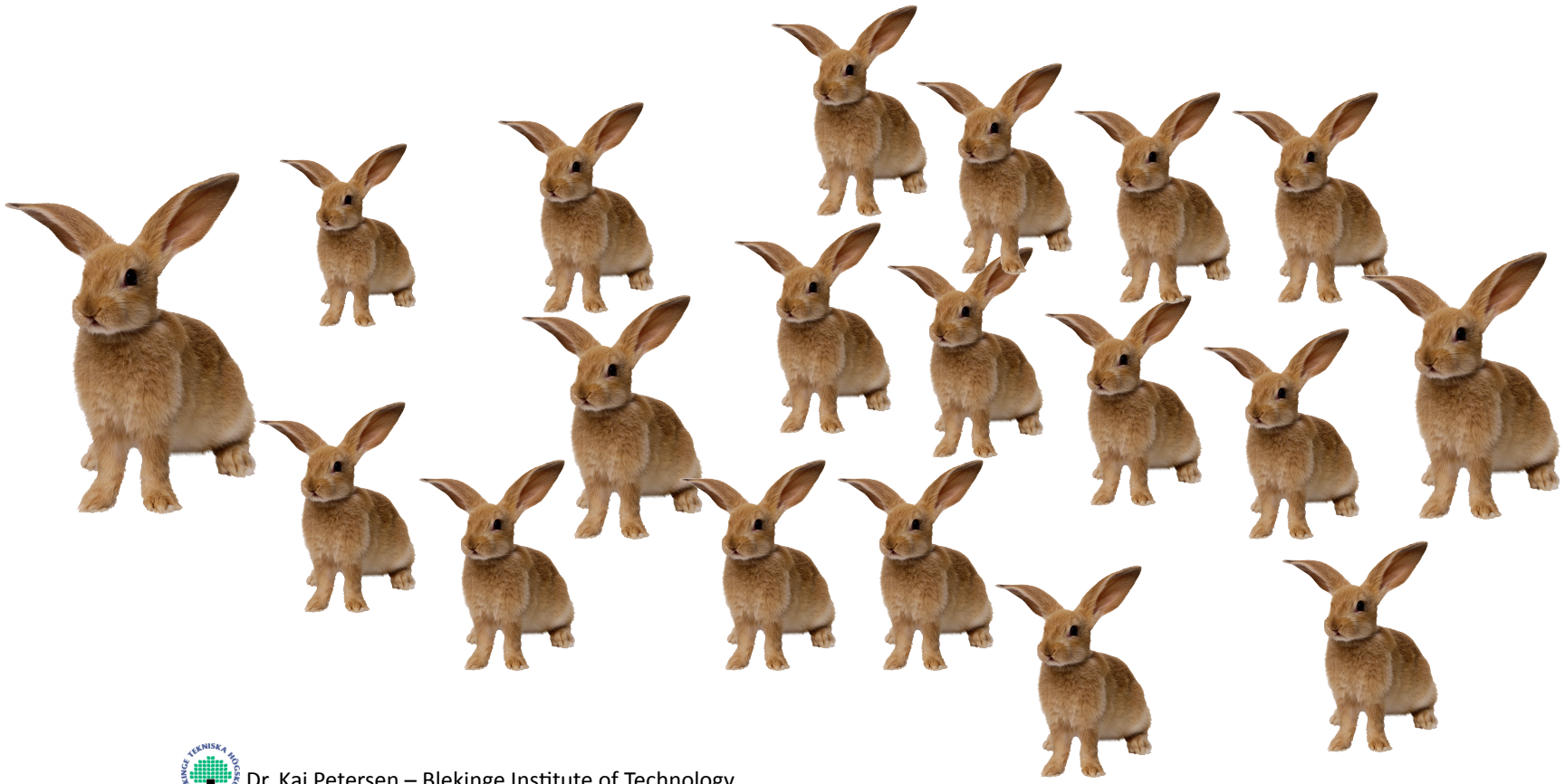
Number	Session	Item Number	Clock Time hh:mm	Position	Risk	Type of Fault	Descripion of Fault (Write fault description clear so anyone can read it and understand the fault)

Exit Criteria: Defect content estimation in inspection

- When inspecting we would like to know how many defects are left in the document (defect population) based on the defects found (defect sample)
- Possible approaches:
  - Capture-Recapture
  - Maximum Likelihood (see lecture “Reliability Prediction”)

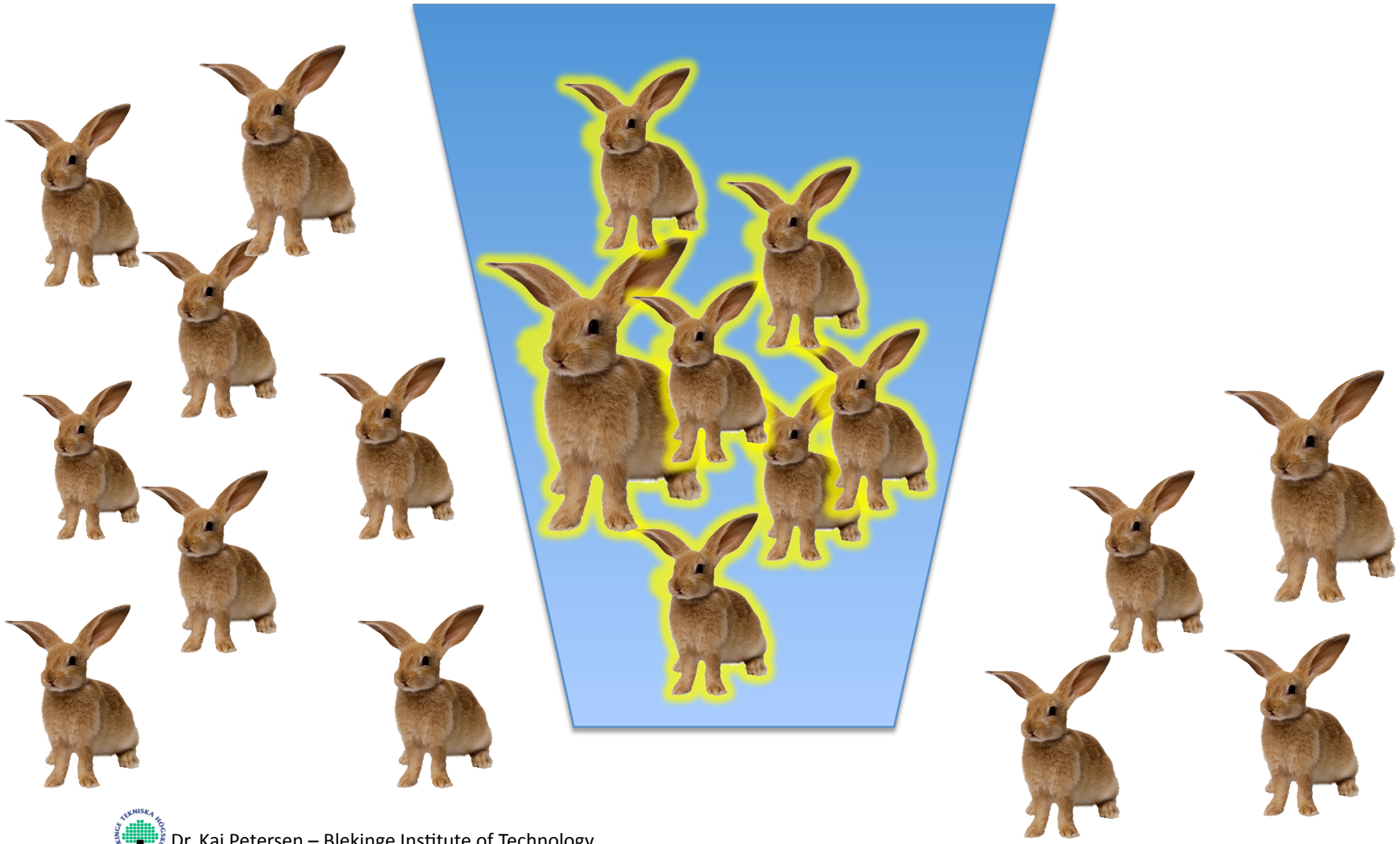
Exit Criteria: Capture Recapture

- Estimation of biological populations
- Example Lincoln-Petersen method



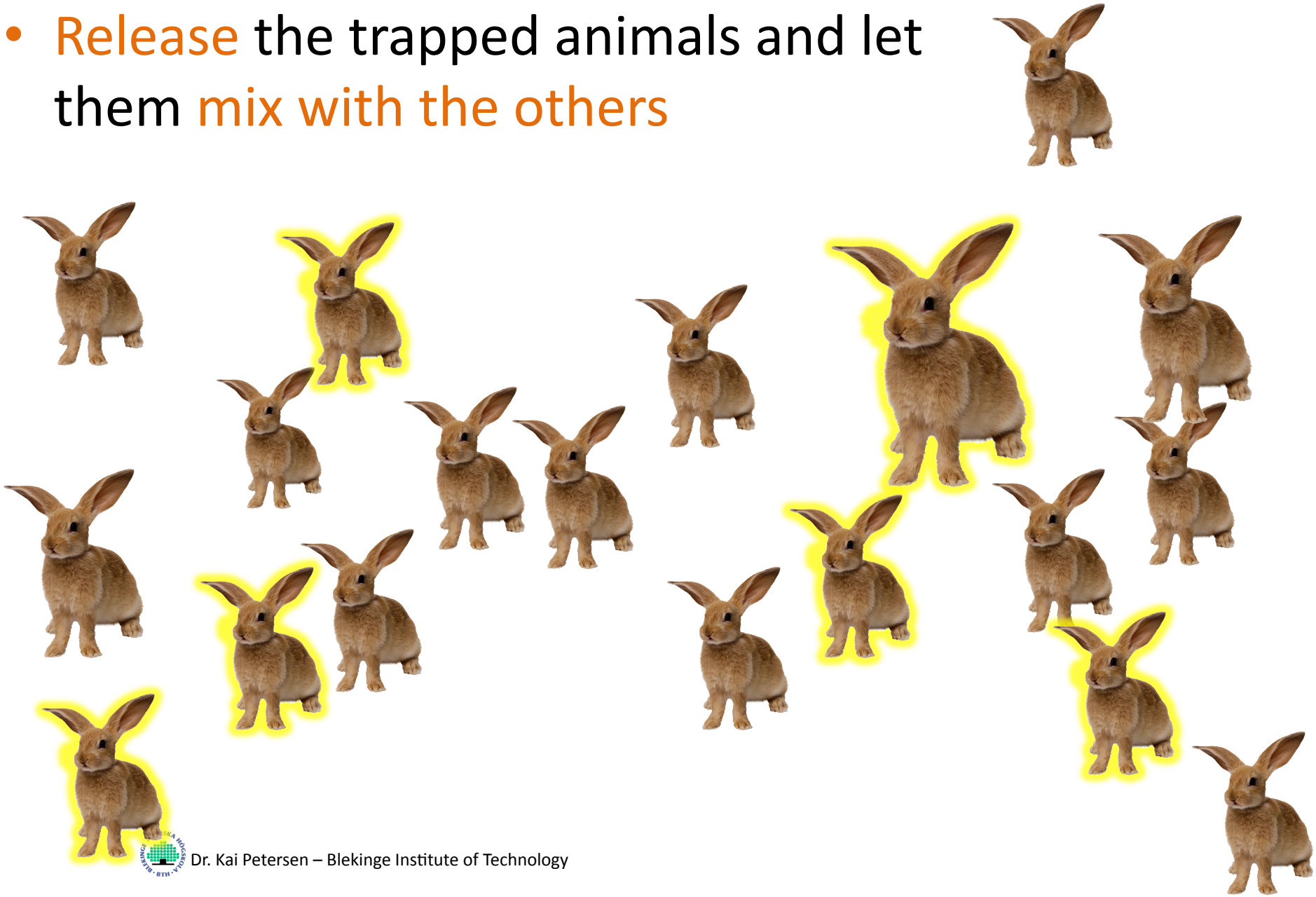
Exit Criteria: Capture Recapture

- Hunt and Tag



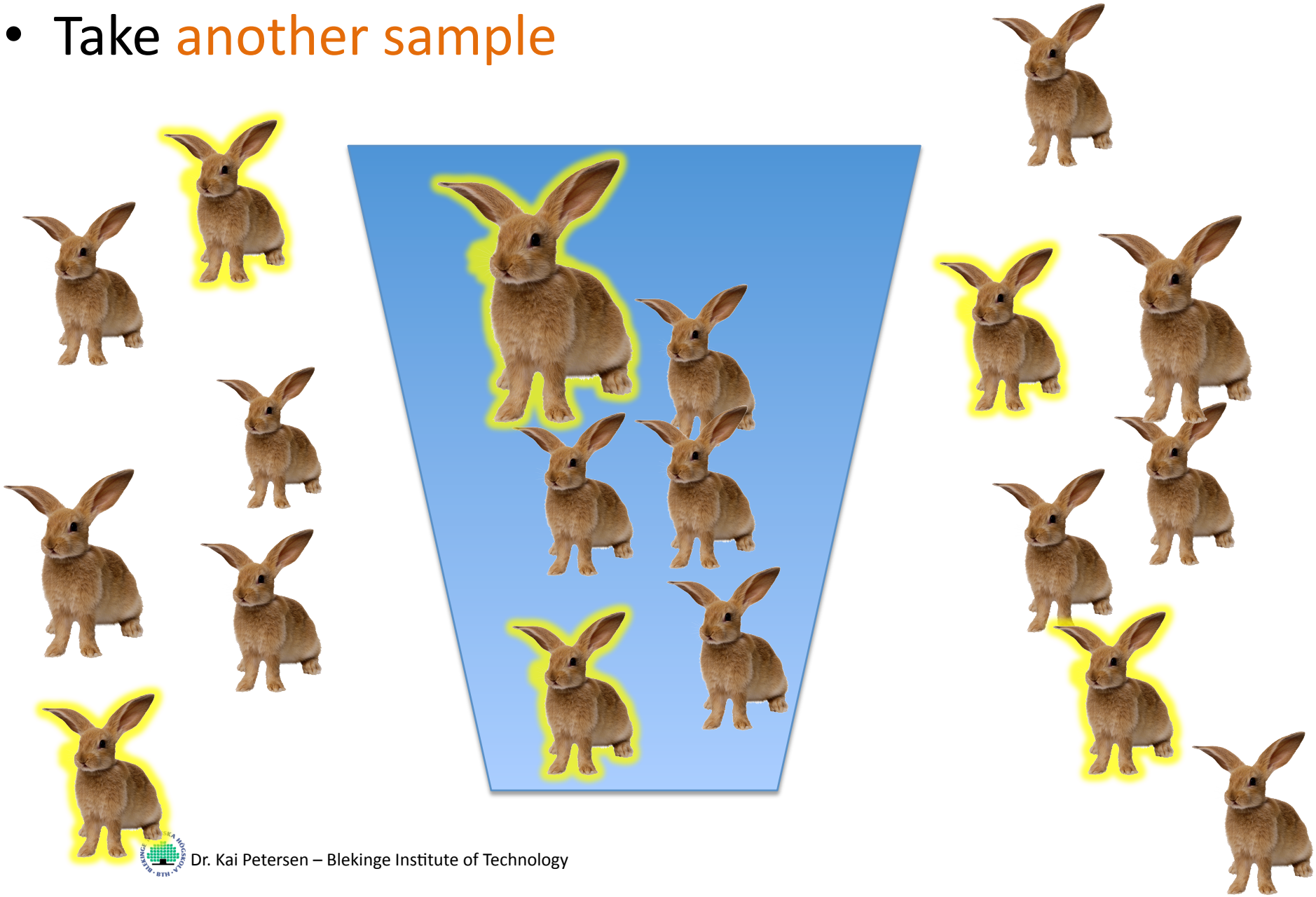
Exit Criteria: Capture Recapture

- Release the trapped animals and let them mix with the others



Exit Criteria: Capture Recapture

- Take another sample



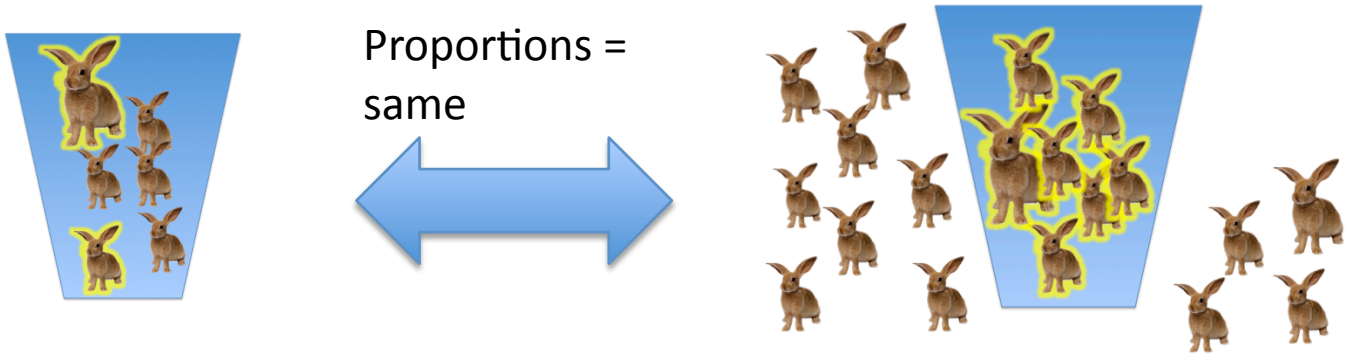
Exit Criteria: Capture-Recapture

- Lincoln-Petersen Assumptions
  - A1: Closed population (no birth, death, immigration, emigration)
  - A2: No marks fall off the animals
  - A3: Equal probability of catching the animals
  - A4: Correct recording of the marks

Exit Criteria: Capture-Recapture

- Based on the assumption of equal probabilities of catching animals the estimation formula can be derived
- Given equal probabilities the proportion of re-captured animals to the total number of animals caught in second sample should be equal to the proportion of the marked animals in the first sample to the population.

$$\frac{N}{M} = \frac{C}{R}$$



$$N = \frac{M * C}{R}$$

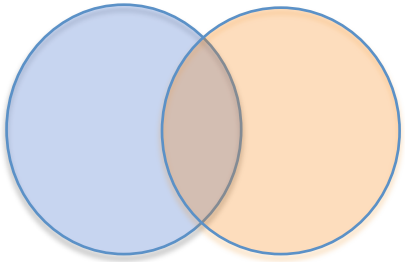
N = Estimate of total population size
M = Total number of animals captured on the first visit
C = Total number of animals captured in the second visit
R = Total number of animals captured on the second visit that were also captured on the first visit.

Exit Criteria: Capture-Recapture: Example calculation based on our example

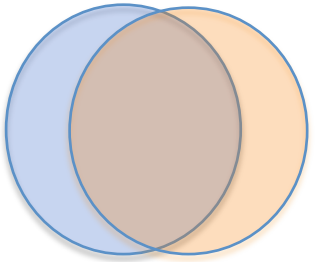
N = ?
M = 7
C = 6
R = 2

$$\frac{N}{7} = \frac{6}{2}$$

2/6 of the animals in the first sample have been re-captured. That means, 1/3 of the total population was included in the first sample (i.e. M=7 is 1/3 of the total population).



More animals left



Fewer animals left

$$N = \frac{7 * 6}{2} = 21$$

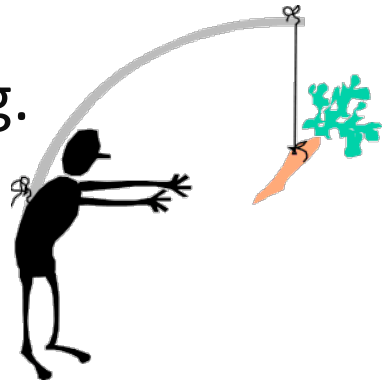
Note: If we would, e.g., would have re-captured 6 out of 7, the population would be estimated to be much smaller

Some last tips for inspections

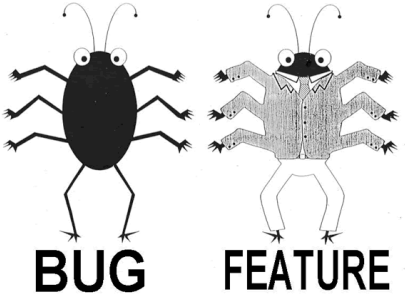
- Find a champion
- Pick things that really count
- Explicitly plan and track review activities
- Consider criticality of defects
- Train participants
- Manage people issues
- Follow the rules (repeatability) but keep it as simple as possible
- Continuously improve process and tools
- Report results
- Do it and stick to it

Automated Static Analysis (SAT)

- Motivation for automated static analysis
  - Programming languages are not safe to use (e.g. buffer overflows leading to security issues)



- Features
  - Check coding standards
  - Code metrics
  - Structure
    - Control flow structure (e.g. identify dead code)
    - Data flow structure (e.g. referencing variable with undefined value)
    - Data structure (e.g. array, boundaries, complexity, matrices)
  - Parallelism (Multi-Core)



SAT: Coding Standards/Conventions

- Example: Java
  - File-Names
  - File-Organization
  - Indentation
  - Comments
  - Declarations
  - Statements
  - White Space
  - Naming Conventions
  - Programming Practices

http://java.sun.com/docs/codeconv/CodeConventions.pdf

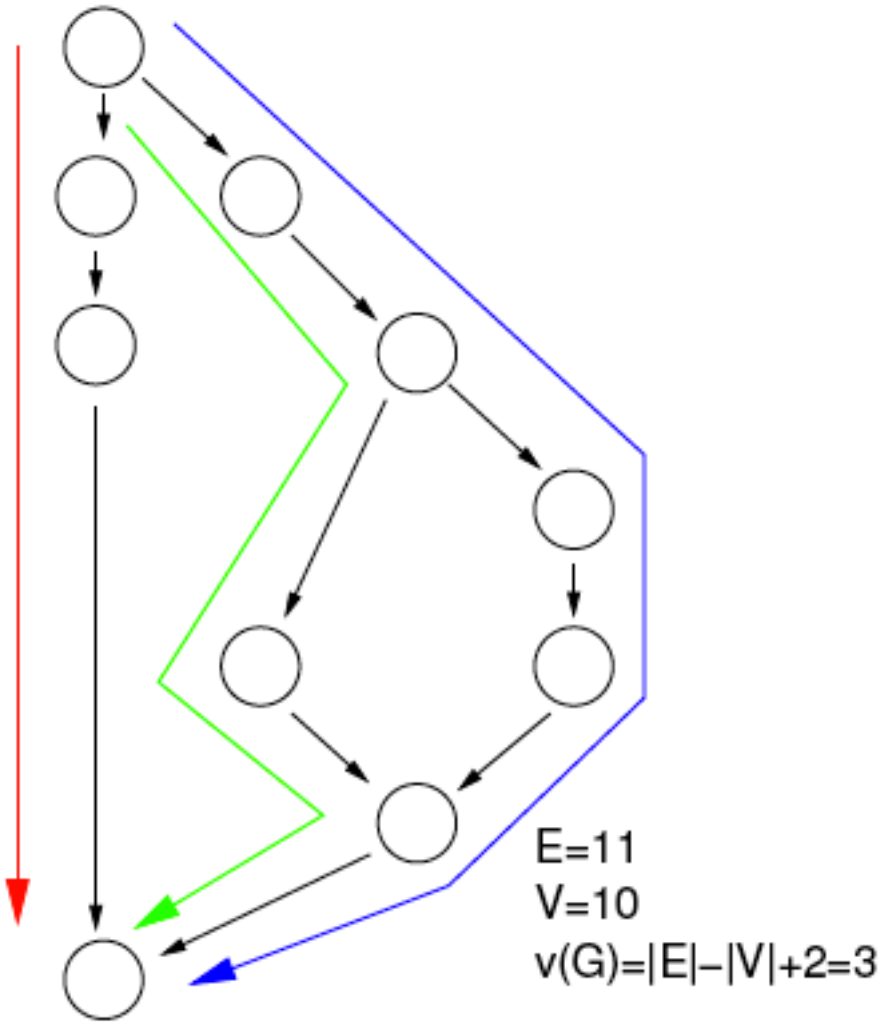
SAT: Code Metrics - Cyclomatic Complexity

Cyclomatic Complexity Definition:
M = E – N + 2

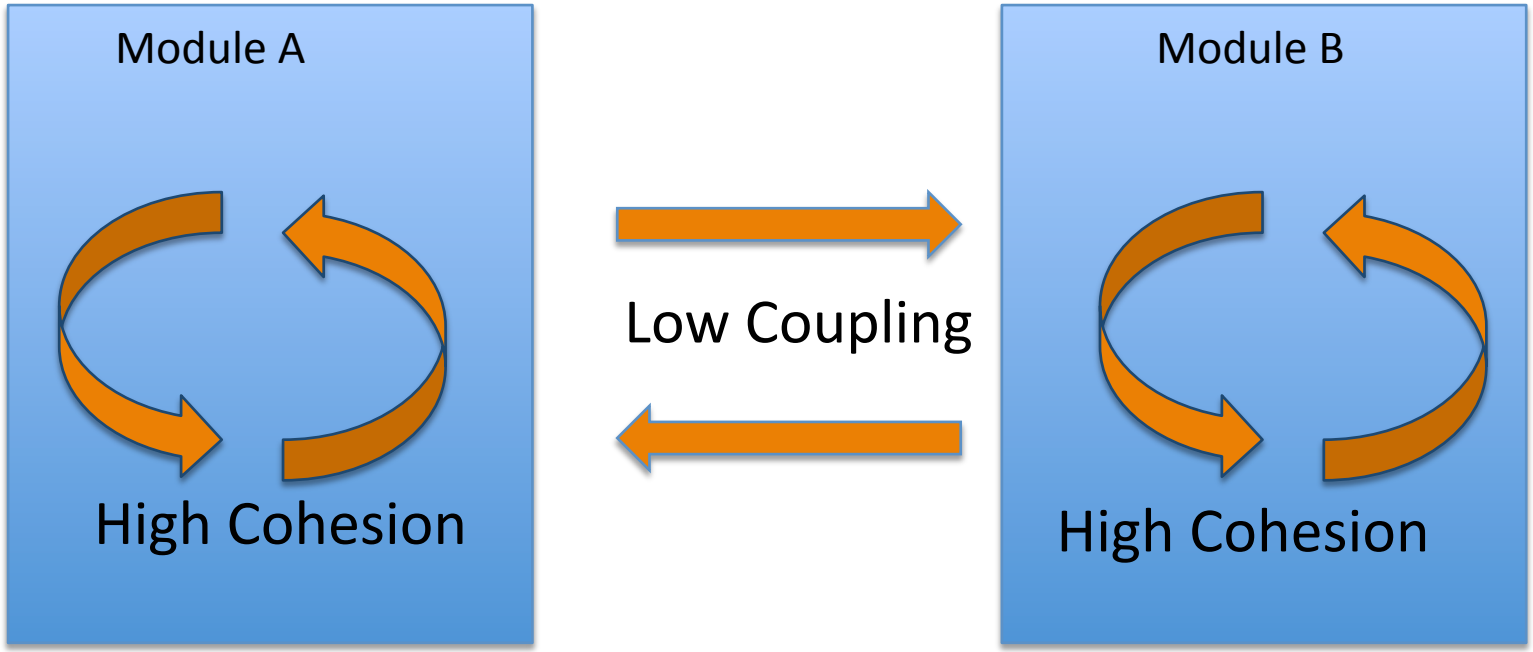
M = cyclomatic compexity
E = number of edges
N = number of nodes

Cyclomatic Complexity Definition:
E = 11
N = 10
M = 11 – 10 + 2 = 3

Rule of thumb:
M<10, but debated



SAT: Code Metrics - Cohesion/Coupling



Measuring Cohesion: Example
LCOM 1 = P - Q, if P > Q, LCOM = 0 otherwise.
LCOM = 0 => cohesive class.

Rule for setting P and Q:
For each pair of methods in a class, check if they access the same or different instance variances. If they share at least one variable, increase Q by one. If they access different sets of instance variables, increase P by one.

LCOM 1: Example

Two instance variables V1 and V2,
four Methods M1, M2, M3, and M4.

M1 accesses V1
M2 accesses V2
M3 accesses V1 and V2
M4 accesses V2

	M1	**M2**	**M3**	**M4**
M1	-	Diff (P+1)	V1 (Q+1)	Diff (P+1)
M2		-	V2 (Q+1)	V2 (Q+1)
M3			-	V2 (Q+1)
M4				-

P = 2
Q = 4

If (P > Q) then LCOM = P-Q (not the case)
else LCOM = 0 (is the case) => hence this is a cohesive class with respect to the methods, which is visible from that the majority of methods share access to the same variable

LCOM 1: Problem with the Metric

LCOM1 suffers from a number of problems:

1. Consistency: For very different classes the value of 0 can be obtained.
2. The definition is based on method interaction with instance variables. However, this is not necessarily a good definition of cohesion (oversimplification), cohesion means much more that could be looked at:

Coincidental: The elements of a method have nothing in common besides being within the same method.
Logical: Elements with similar functionality such as input/output handling are collected in one method.
Temporal: The elements of a method have logical cohesion and are performed at the same time.
Procedural: The elements of a method are connected by some control flow.
Communicational: The elements of a method are connected by some control flow and operate on the same set of data.
Sequential: The elements of a method have communicational cohesion and are connected by a sequential control flow.
Functional: The elements of a method have sequential cohesion, and all elements contribute to a single task in the problem domain. Functional cohesion fully supports the principle of locality and thus minimizes maintenance efforts.

SAT: Structure (Example 1)

Simple "off-by-one" warning (not a security risk for code injection, but should be fixed – simple)

```
Event overrun-local: Overrun of static array
"buff" of size 32 at position 32 with index
variable "32"
30:         buff[32] = '\0';
```

SAT: Structure (Example 2)

If condition 1, 2, and 3 are met the program would return with an error.

Security problem:
Producing the error does not lead to de-allocated memory. Hence, re-producing the three conditions many times would lead to filled up memory, which allows for a Denial of Service (DoS) attack.

```
Event alloc_fn: Called allocation function
"operator new (unsigned int)"
7859: daData = new CcnDdrSet(daTagNo);

At conditional (1):
"StringTokenizer::hasMoreTokens() != 0" taking
true path

7863: if (st.hasMoreTokens())
7864: {

At conditional (2):
"DicosString::compare(const char *) const !=
0" taking true path

7867:    if (daAmountString.compare("") != 0)
7868:    {

At conditional (3): "daAmount > 1000000"
taking true path

7872:       if (daAmount > MAX_DA_AMOUNT)
7873:       {

Event leaked_storage: Returned without freeing
storage "daData"
7875:             return false;
```

SAT: Parallelism/Concurrecy

- Parallelism leads to a number of problems
  - Bugs/defects
  - Bug patterns (certain styles of programming that frequently lead to failures) -> provide checking rules
- Examples for bugs/bug patterns

Bug-patterns
Deadlocks
Livelocks
Race conditions
Unpredictable results
Efficiency/Performance

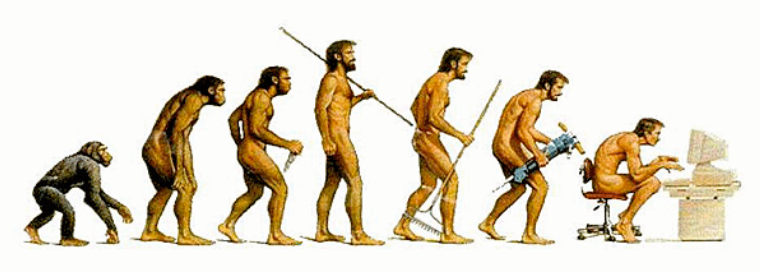
Deatlock
- Blocking-Critical-Section
- Hold and wait

Livelock
- Orphaned Thread

Data Race/Race conditions
- Wrong lock/no lock
- Non-atomic
- Lazy Initialization
- Condition for wait
- etc

SAT - Experiences

- … one thing first: The **job is not done** after running the static code analysis tool (SAT)!
- Tool provides a number of warnings, which have to be reviewed and evaluated
- You have to decide which action to take for each warning
- There are false positives! Correcting them might introduce new defects.
- Some SATs are better than others with respect to false positives (important criterion for goodness of SATs)



- Recommended reading (available through our database systems in our library)
  - Aybüke Aurum, Håkan Petersson, Claes Wohlin: State-of-the-art: software inspections after 25 years. Softw. Test., Verif. Reliab. 12(3): 133-154 (2002)
  - Marcus Ciolkowski: What do we know about perspective-based reading? An approach for quantitative aggregation in software engineering. ESEM 2009: 133-144
  - Dejan Baca, Bengt Carlsson, Kai Petersen, Lars Lundberg: Improving software security with static automated code analysis in an industry setting. Softw., Pract. Exper. 43(3): 259-279 (2013)