

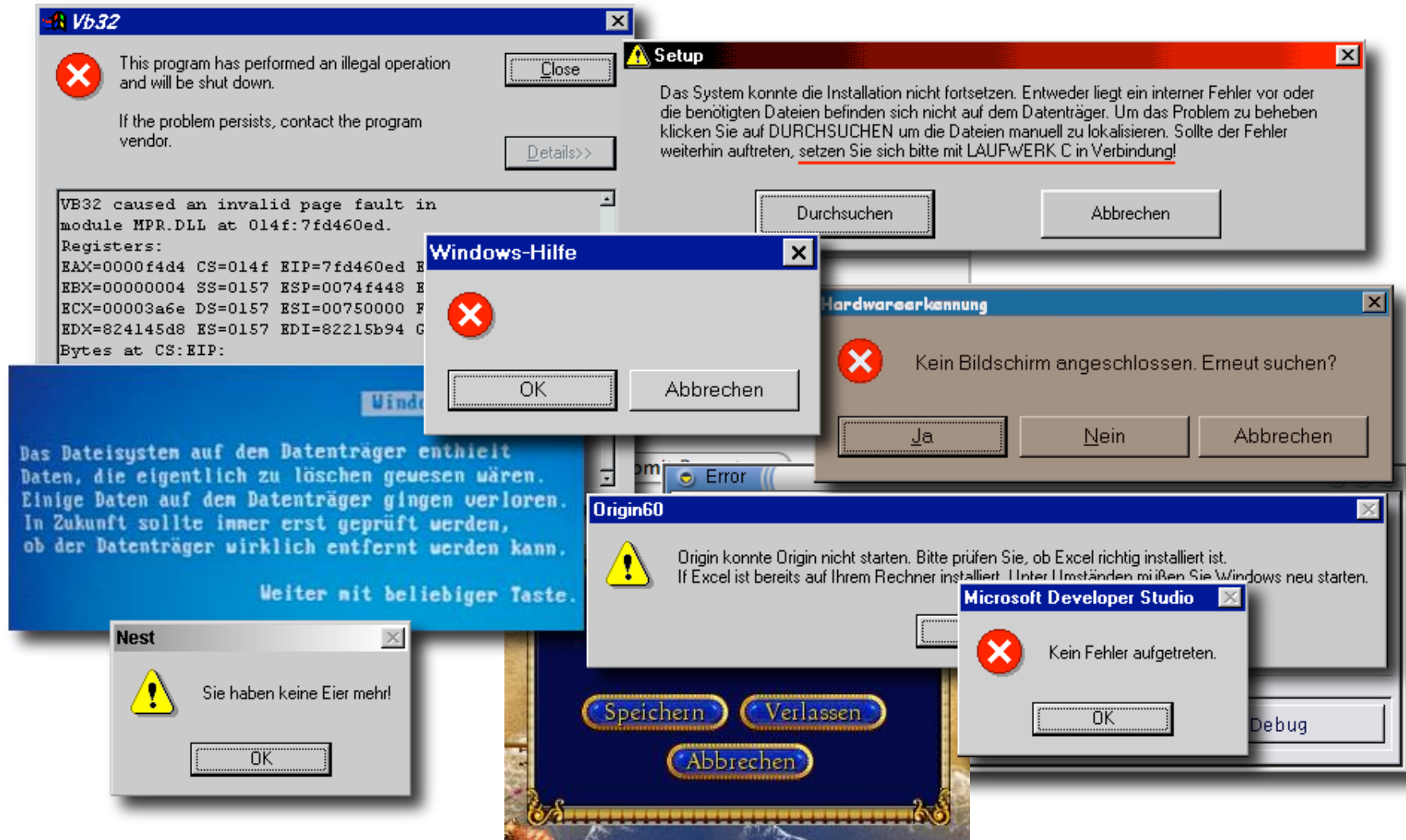
Debugging

Prof. Sven Apel

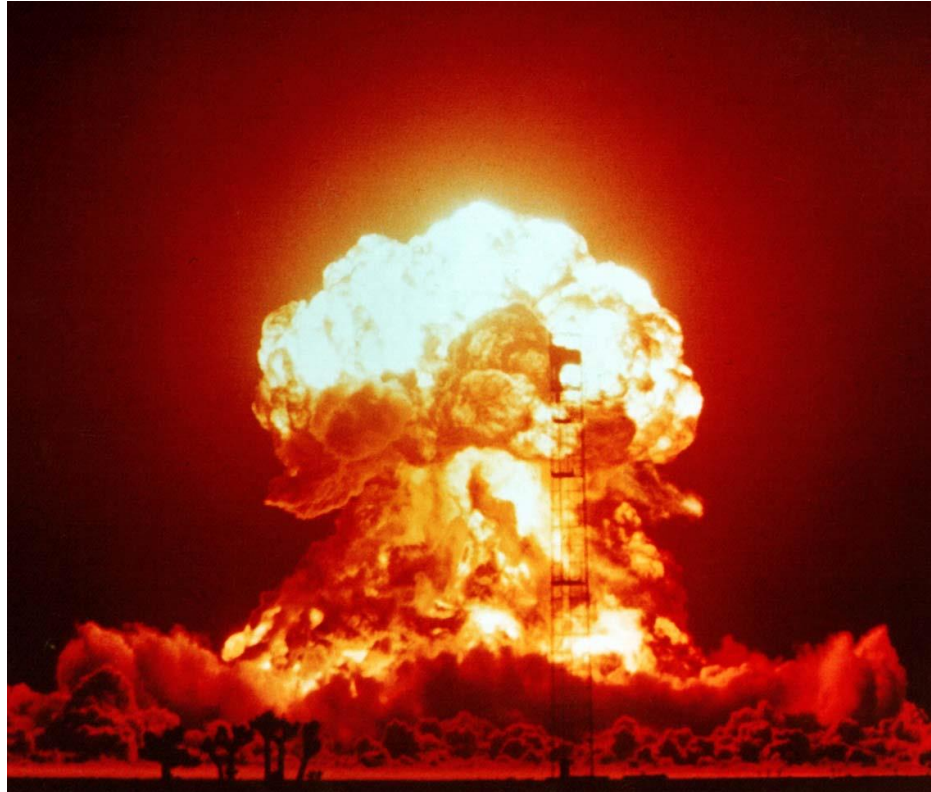
Universität des Saarlandes



Das Problem



Was nun?



Systematische Fehlersuche

Track the problem (Problem verfolgen)

Reproduce (Reproduzieren)

Automate (Automatisieren)

Find Origins (Ursprünge finden)

Focus (Fokussieren)

Isolate (Isolieren)

Correct (Korrigieren)

Problem verfolgen

TRAFFIC

The screenshot displays the GitLab interface for the 'mc-experience-paper' project. The main content area shows a list of 11 open issues, each with a title, number, author, and status labels. The sidebar on the left contains navigation links for Project, Repository, Issues (selected), Boards, Labels, Service Desk, Milestones, Merge Requests, CI/CD, Operations, Wiki, Snippets, and Settings. The top navigation bar includes links for Projects, Groups, Activity, Milestones, and Snippets, along with a search bar and user profile.

Issue Title	Issue Number	Author	Status Labels	Updated
Missing bibliography entries and fields	#23	Stephan Lukasczyk	Paper, To Do	updated 1 year ago
Consistent use of Terms	#19	Andreas Stahlbauer	Paper	updated 1 year ago
Tracking of random variables with the BDD CPA	#14	Andreas Stahlbauer	Experiments	updated 1 year ago
Conclusion	#12	Andreas Stahlbauer	Paper	updated 1 year ago
Empirical Study: Setup	#11	Andreas Stahlbauer	Paper	updated 1 year ago
Empirical Study: Threads to Validity	#10	Andreas Stahlbauer	Paper	updated 1 year ago
Empirical Study: Discussion	#9	Andreas Stahlbauer	Paper	updated 1 year ago
Empirical Study: Results	#8	Andreas Stahlbauer	Paper, To Do	updated 1 year ago
Empirical Study: Operationalization	#7	Andreas Stahlbauer	Paper, Doing	updated 1 year ago
Research Questions	#4	Andreas Stahlbauer	Paper, Doing	updated 1 year ago
Problem Statement (Examples and Problems)				

Problem verfolgen

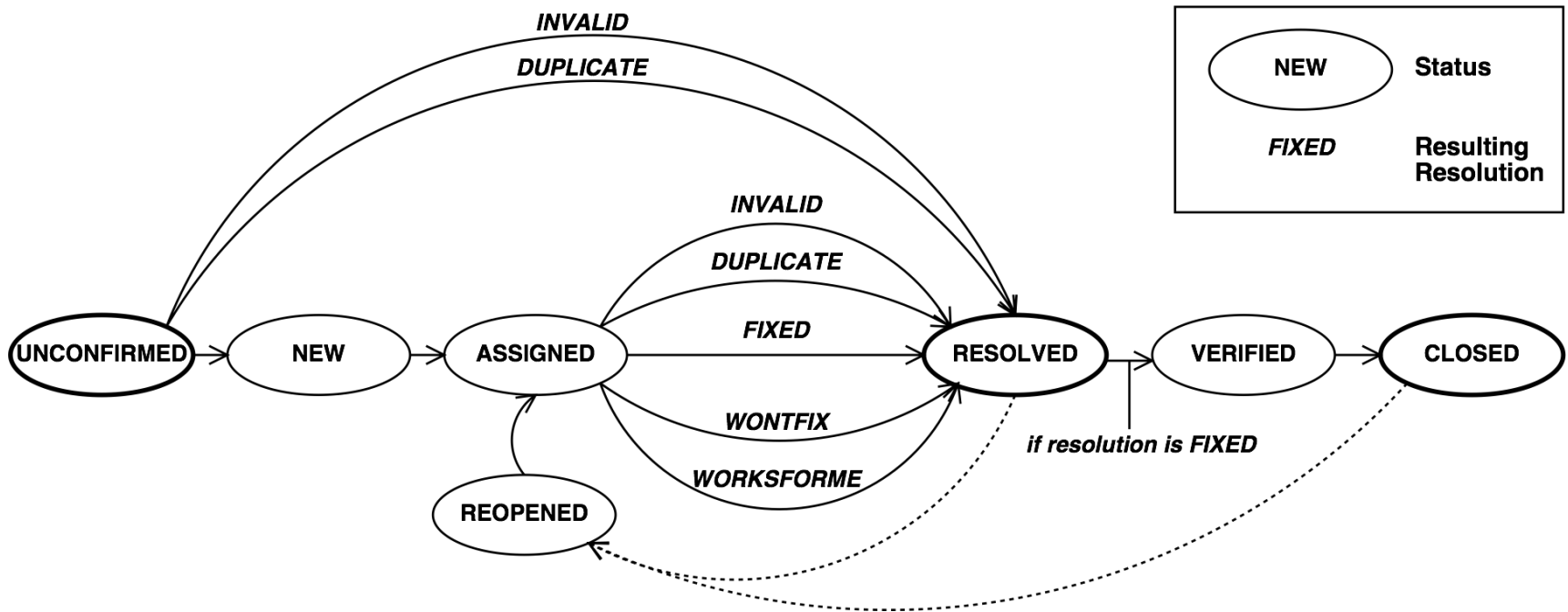
Jedes Problem wird in die Fehler-Datenbank eingetragen

Die Priorität bestimmt, welches Problem als nächstes bearbeitet wird

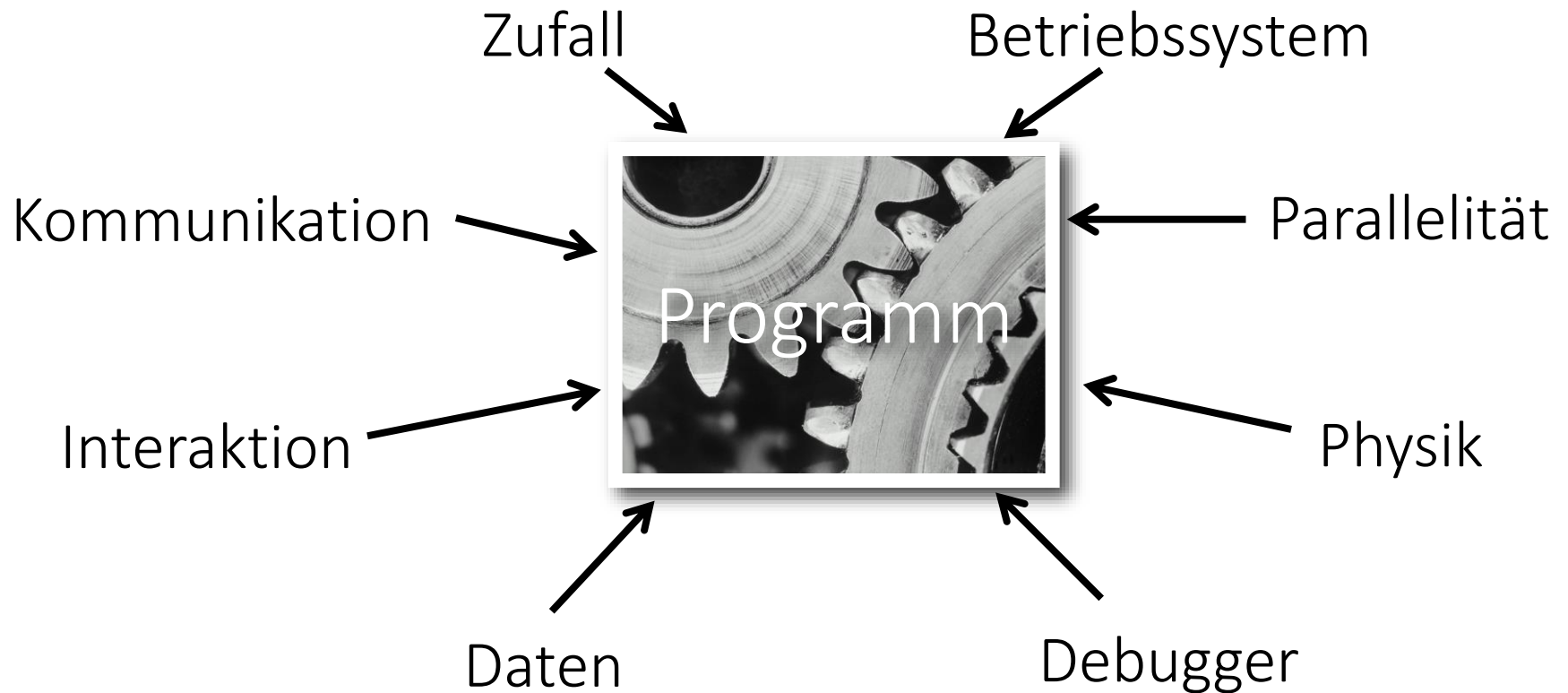
Sind alle Probleme behoben,
ist das Produkt fertig

Lebenszyklus eines Problems

TRAFFIC



Reproduzieren



Automatisieren

```
// Test for host
public void testHost() {
    int noPort = -1;
    assertEquals(askigor_url.getHost(), "www.askigor.org");
    assertEquals(askigor_url.getPort(), noPort);
}

// Test for path
public void testPath() {
    assertEquals(askigor_url.getPath(), "/status.php");
}

// Test for query part
public void testQuery() {
    assertEquals(askigor_url.getQuery(), "id=sample");
}
```

Automatisieren

Jedes Problem sollte *automatisch reproduzierbar* sein

Dies geschieht über geeignete JUnit-Testfälle

Nach *jeder Änderung* werden die Testfälle ausgeführt

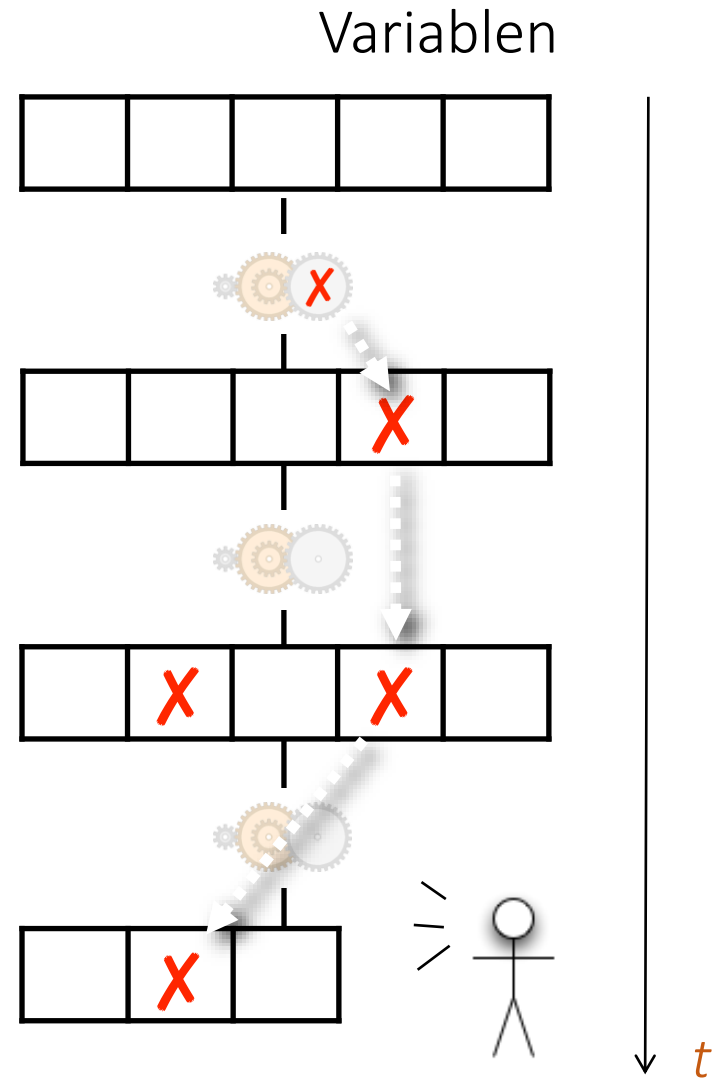
Ursprung finden

Der Programmierer erzeugt einen *Defekt* – einen Fehler im Code

Der ausgeführte Defekt erzeugt eine *Infektion* – einen Fehler im Zustand

Die Infektion breitet sich aus...

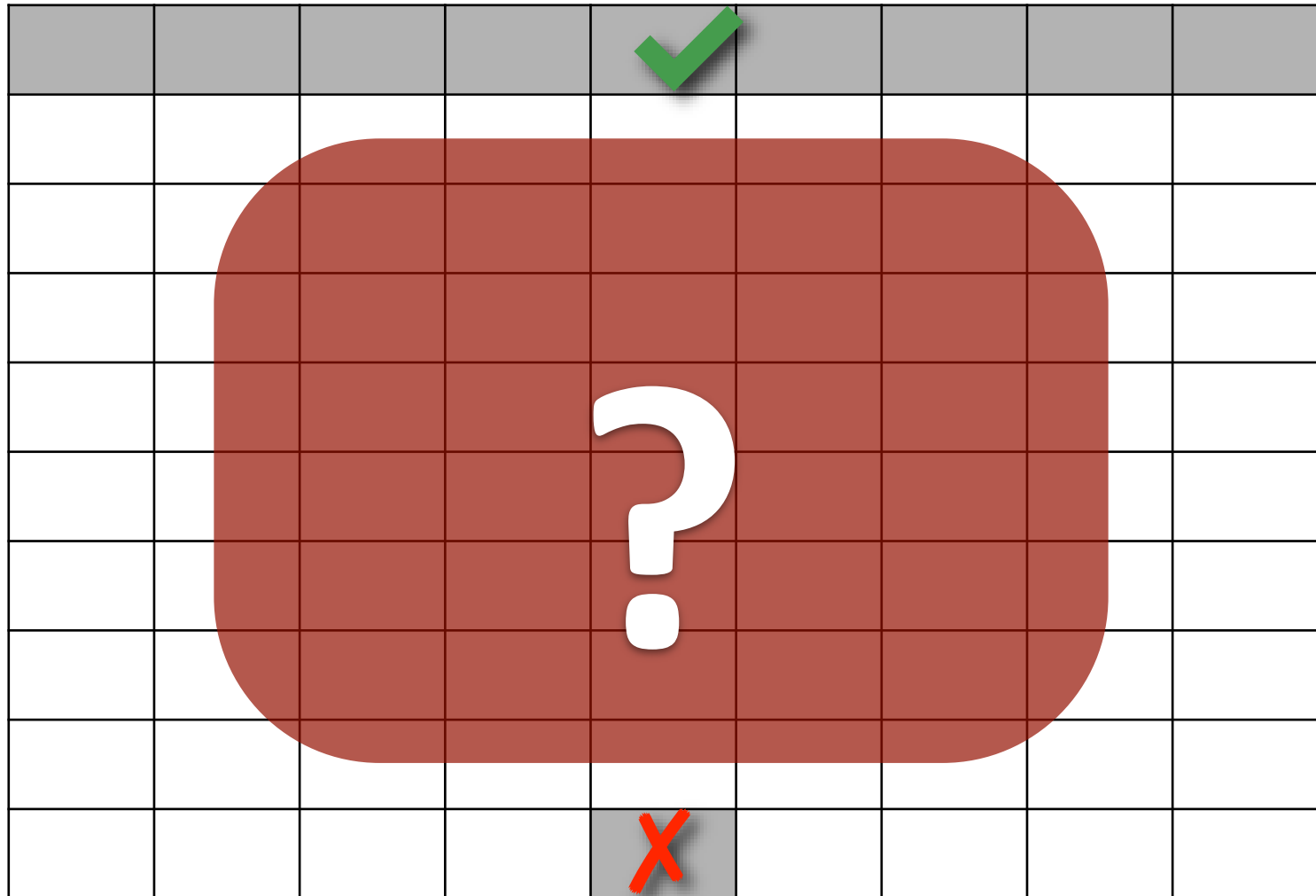
...und wird als *Fehlverhalten* sichtbar.



Diese Infektionskette müssen wir *brechen*!

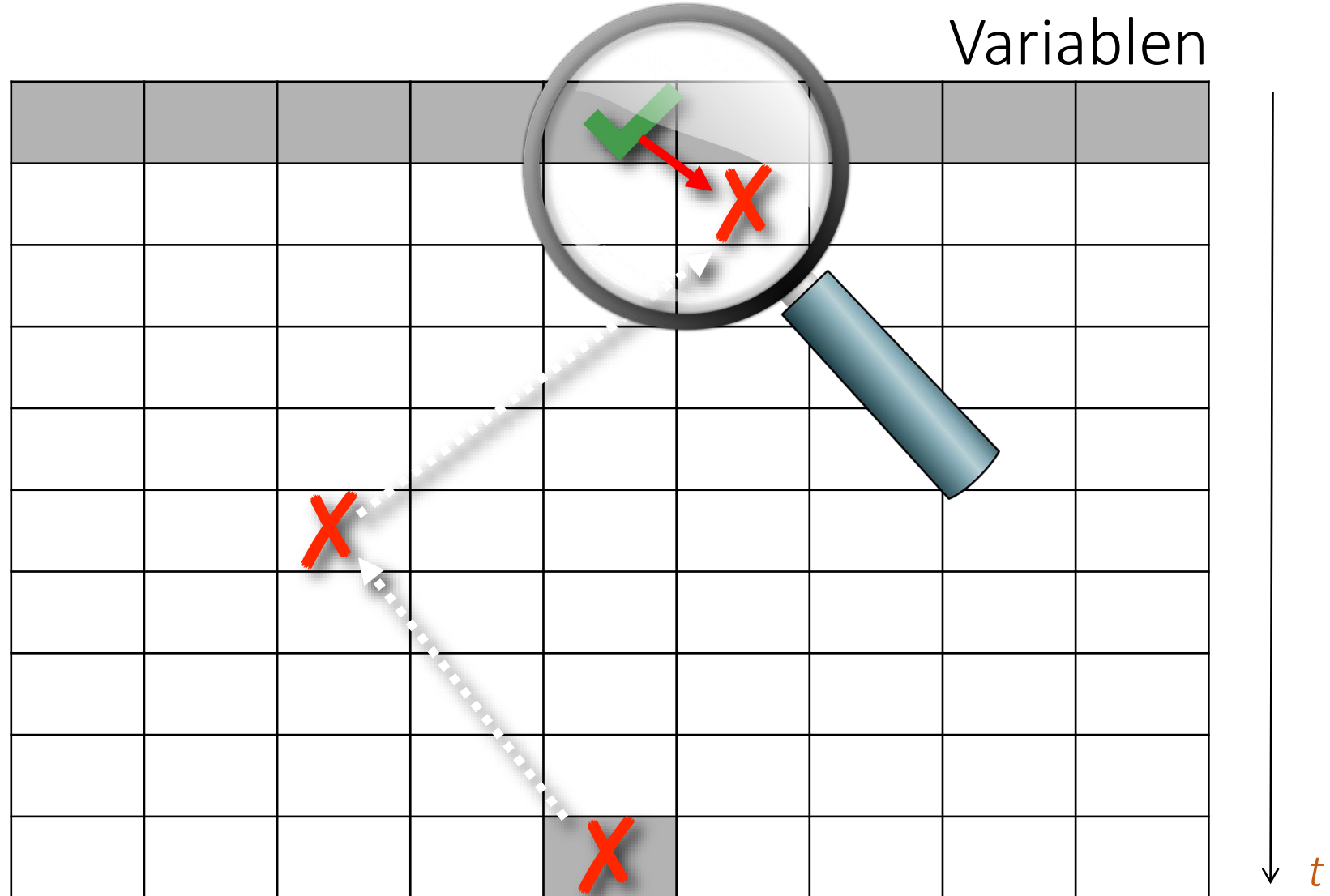
Ursprung finden


Variablen





Ursprung finden


TRAFFIC





 EXPLORER


 OPEN EDITORS


 SPRING-PETCLINIC


 2


 12












































▸ .mvn

▸ .vscode

▸ src

▸ main

▸ java

▸ org

▸ springframework

▸ samples

▸ petclinic

▸ model

BaseEntity.java

NamedEntity.java

package-info.java

Person.java

▸ owner

▸ system

▸ vet

▸ visit

Visit.java


VisitRepository.java

PetClinicApplication.java

▸ less

1

▸ MAVEN PROJECTS






PROBLEMS 13

OUTPUT

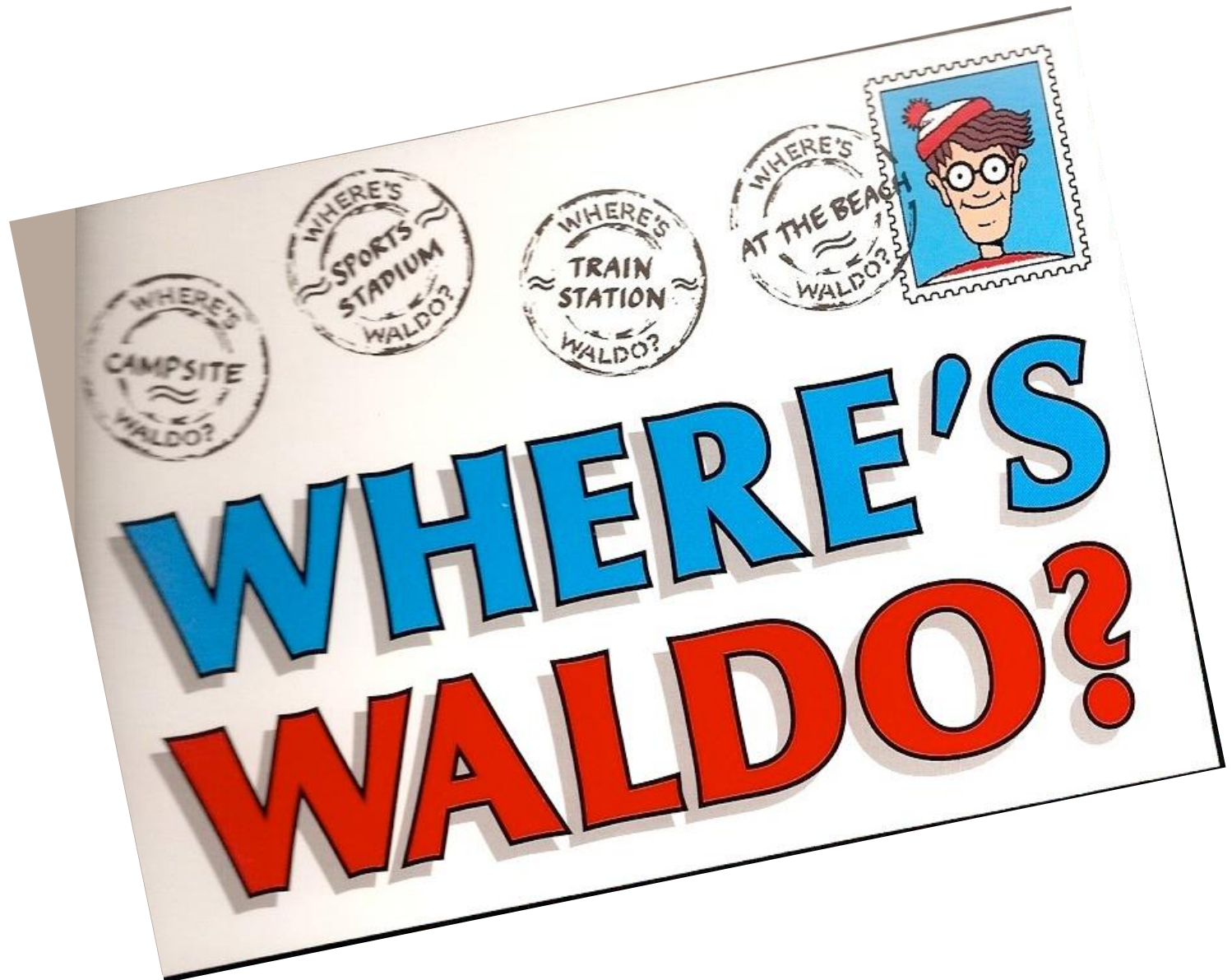
DEBUG CONSOLE

TERMINAL



Suche

TRAFFIC





Bei der Suche nach Infektionen konzentrieren uns auf Stellen im Zustand, die

wahrscheinlich falsch sind (z.B. weil hier früher Fehler aufgetreten sind)

explizit falsch sind (z.B. weil sie eine Zusicherung verletzen)

Zusicherungen sind das effektivste Mittel, Infektionen zu finden.

Infektionen finden

```
class Time {  
public:  
    int hour();      // 0..23  
    int minutes();   // 0..59  
    int seconds();    // 0..60 (incl. leap seconds)  
  
    void set_hour(int h);  
    ...  
}
```

Jede Zeit von 00:00:00 bis 23:59:60 ist gültig

Ursprung finden

```
bool Time::repOK()  
{  
    return (0 <= hour() && hour() <= 23) &&  
           (0 <= minutes() && minutes() <= 59) &&  
           (0 <= seconds() && seconds() <= 60);  
}
```

```
void Time::set_hour(int h)  
{  
    assert (repOK()); // Vorbedingung  
  
    ... // Code der eigentlichen Methode  
  
    assert (repOK()); // Nachbedingung  
}
```

Ursprung finden

```
bool Time::repOK()  
{  
    return (0 <= hour() && hour() <= 23) &&  
           (0 <= minutes() && minutes() <= 59) &&  
           (0 <= seconds() && seconds() <= 60);  
}
```

`repOK()` ist die *Invariante* eines Time-Objekts:
gilt *vor* jeder öffentlichen Methode
gilt *nach* jeder öffentlichen Methode

Ursprung finden

Vorbedingung schlägt fehl = Infektion *vor* Methode

Nachbedingung schlägt fehl = Infektion *nach* Methode

Alle Zusicherungen ok = keine Infektion

```
void Time::set_hour(int h)
{
    assert (repOK()); // Vorbedingung

    ... // Code der eigentlichen Methode

    assert (repOK()); // Nachbedingung
}
```

Komplexe Invarianten

```
class RedBlackTree {  
    ...  
    boolean repOK() {  
        assert (rootHasNoParent());  
        assert (rootIsBlack());  
        assert (redNodesHaveOnlyBlackChildren());  
        assert (equalNumberOfBlackNodesOnSubtrees());  
        assert (treeIsAcyclic());  
        assert (parentsAreConsistent());  
  
        return true;  
    }  
}
```

Fokussieren

Alle möglichen Einflüsse müssen geprüft werden

Konzentration auf wahrscheinlichste Kandidaten

Zusicherungen helfen schnell, Infektionen zu finden

Fehlerursachen sollen *systematisch* eingeeengt werden – mit Beobachtungen und Experimenten.

Wissenschaftliche Methode

Beobachte einen Teil des Universums

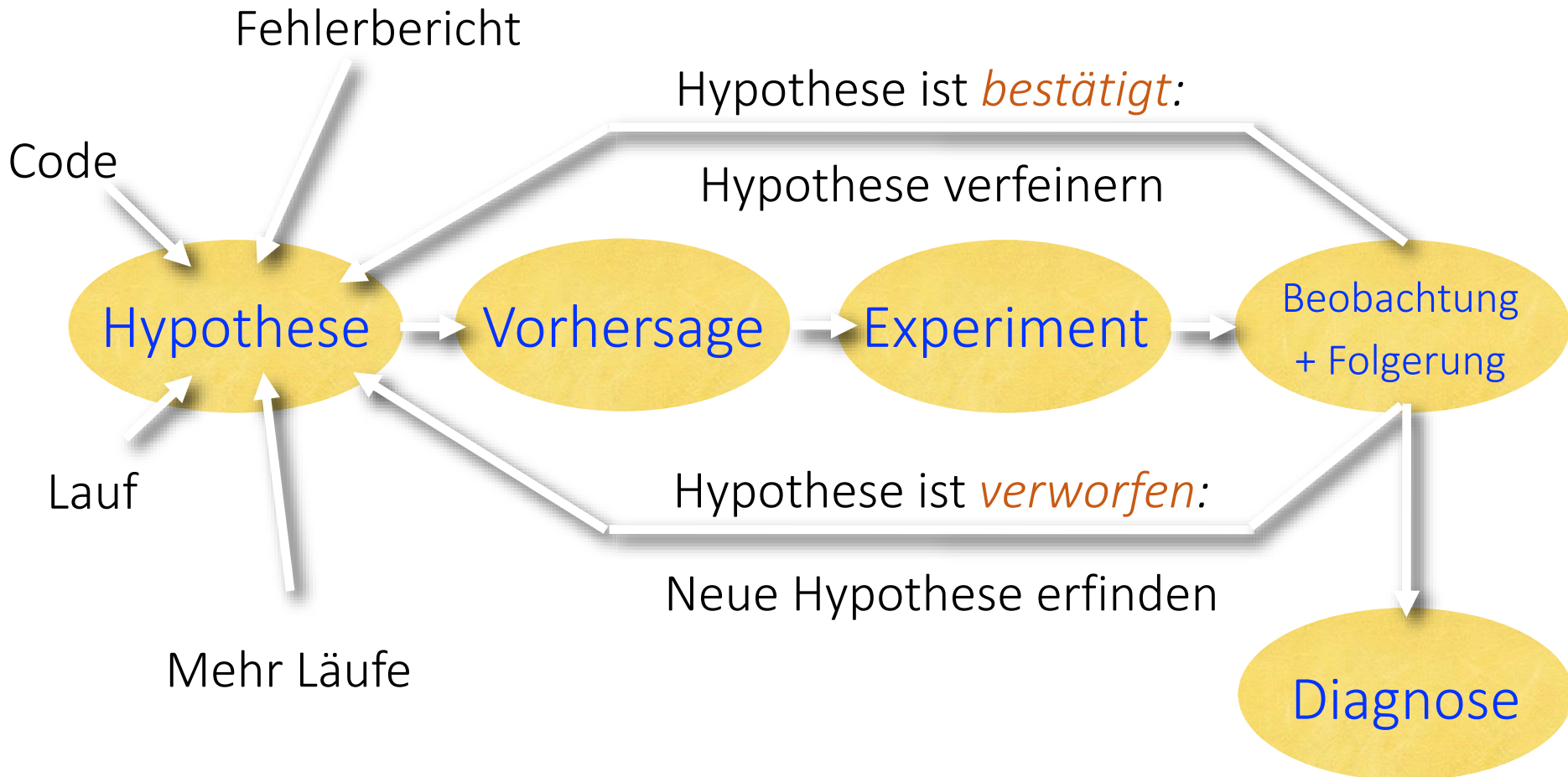
Erfinde eine *Hypothese*, die mit der Beobachtung übereinstimmt

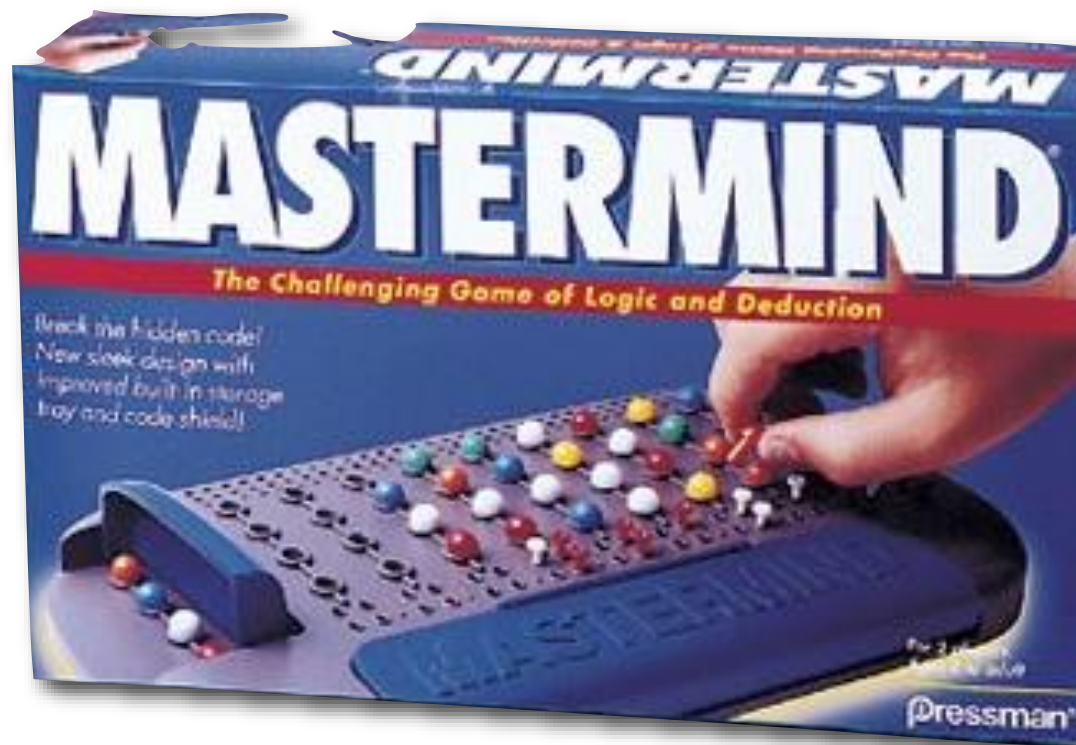
Nutze die Hypothese, um *Vorhersagen* zu machen.

Teste die Vorhersagen durch Experimente oder Beobachtungen und passe die Hypothese an.

Wiederhole 3 and 4, bis die Hypothese zur *Theorie* wird.

Wissenschaftliche Methode





Explizite Hypothesen

Hypothesis The execution causes $a[0] = 0$

Prediction At Line 37, $a[0] = 0$ should hold.

Experiment Observe $a[0]$ at Line 37.

Observation $a[0] = 0$ holds as predicted.

Conclusion Hypothesis is confirmed.

Explizite Hypothesen

Hypothesis The execution causes $a[0] = 0$

Prediction ... should hold.

Experiment ... 7.

Observation ... holds as predicted.

Conclusion Hypothesis is confirmed.

Wer alles im Kopf
behält, spielt
Mastermind blind!

Isolieren

Wir wiederholen die Suche nach Infektions-Ursprüngen, bis wir den Defekt gefunden haben.

Wir gehen *systematisch* vor – im Sinne der wissenschaftlichen Methode

Durch *explizite* Schritte leiten wir die Suche und können sie jederzeit nachvollziehen

Korrektur

Vor der Korrektur müssen wir prüfen,
ob der Defekt

tatsächlich ein *Fehler ist* und
das *Fehlverhalten verursacht*

Erst wenn beides verstanden ist, dürfen wir den
Fehler korrigieren.

Hausaufgaben

Tritt das Fehlverhalten nicht mehr auf?
(Falls doch, sollte dies eine große Überraschung sein)

Könnte die Korrektur neue Fehler einführen?

Wurde derselbe Fehler woanders gemacht?

Ist meine Korrektur ins Versionsmanagement
und Problem-Tracking eingespielt?

Systematische Fehlersuche

Track the problem (Problem verfolgen)

Reproduce (Reproduzieren)

Automate (Automatisieren)

Find Origins (Ursprünge finden)

Focus (Fokussieren)

Isolate (Isolieren)

Correct (Korrigieren)