Debugging - Introduction

Steven Costiou steven.costiou@inria.fr

RMoD / Inria Lille - Nord Europe

September 2022

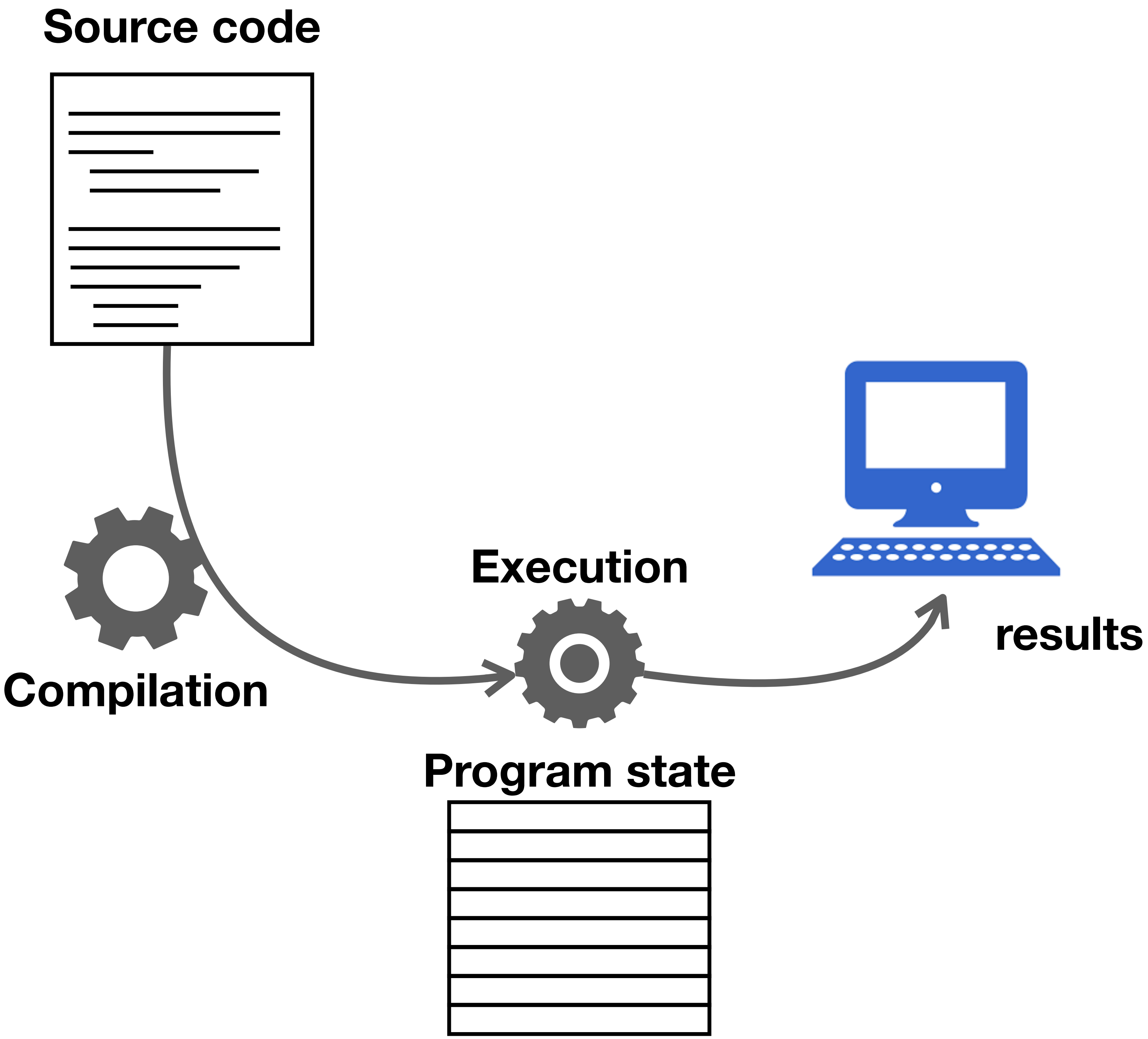
Summary

1. Bugs
2. Debugging
3. Debugging in the industry
4. Difficulties
5. Cost of debugging
6. Bibliography

Bugs

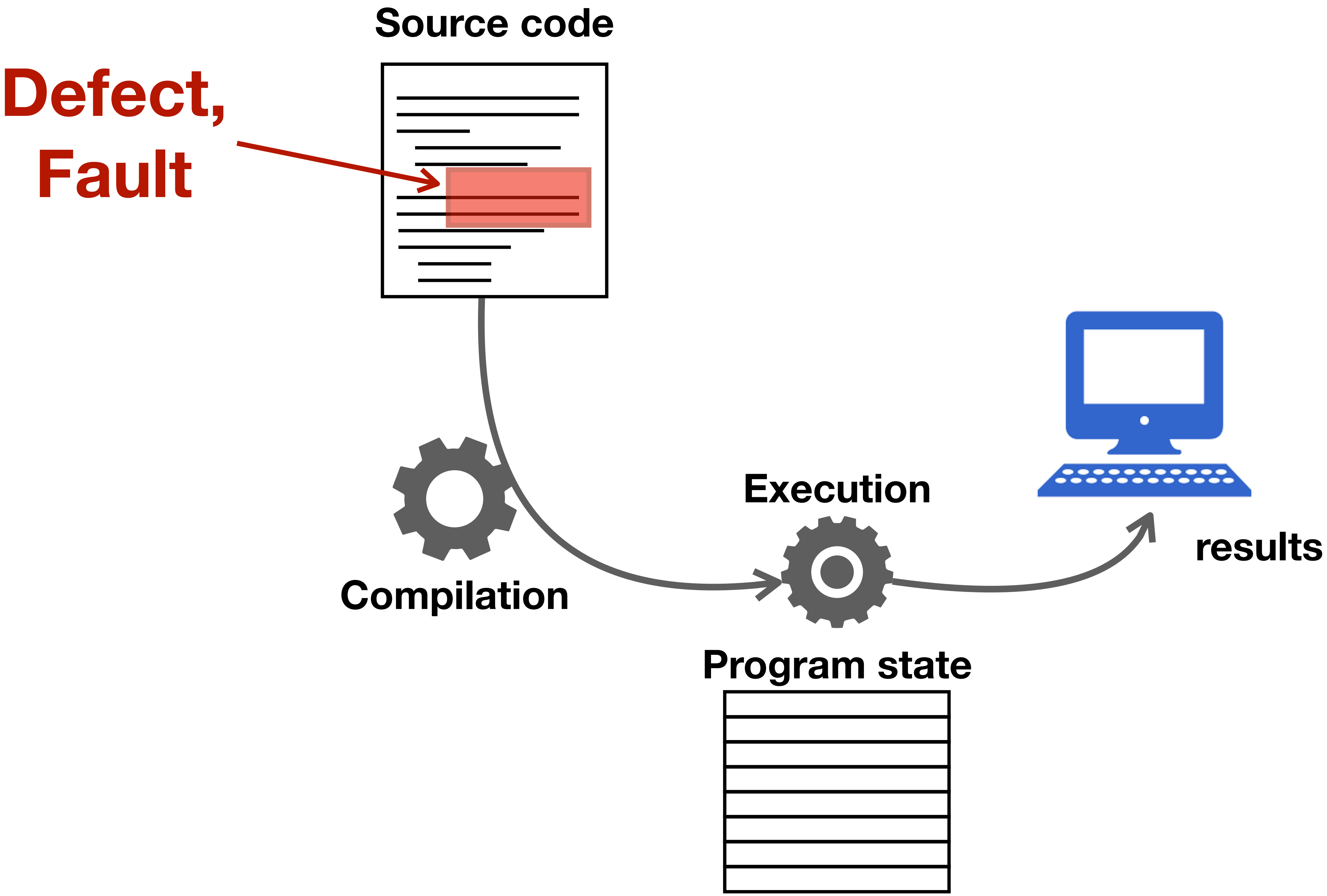
What’s a bug?

[Why programs Fail, Andreas Zeller, 2009]



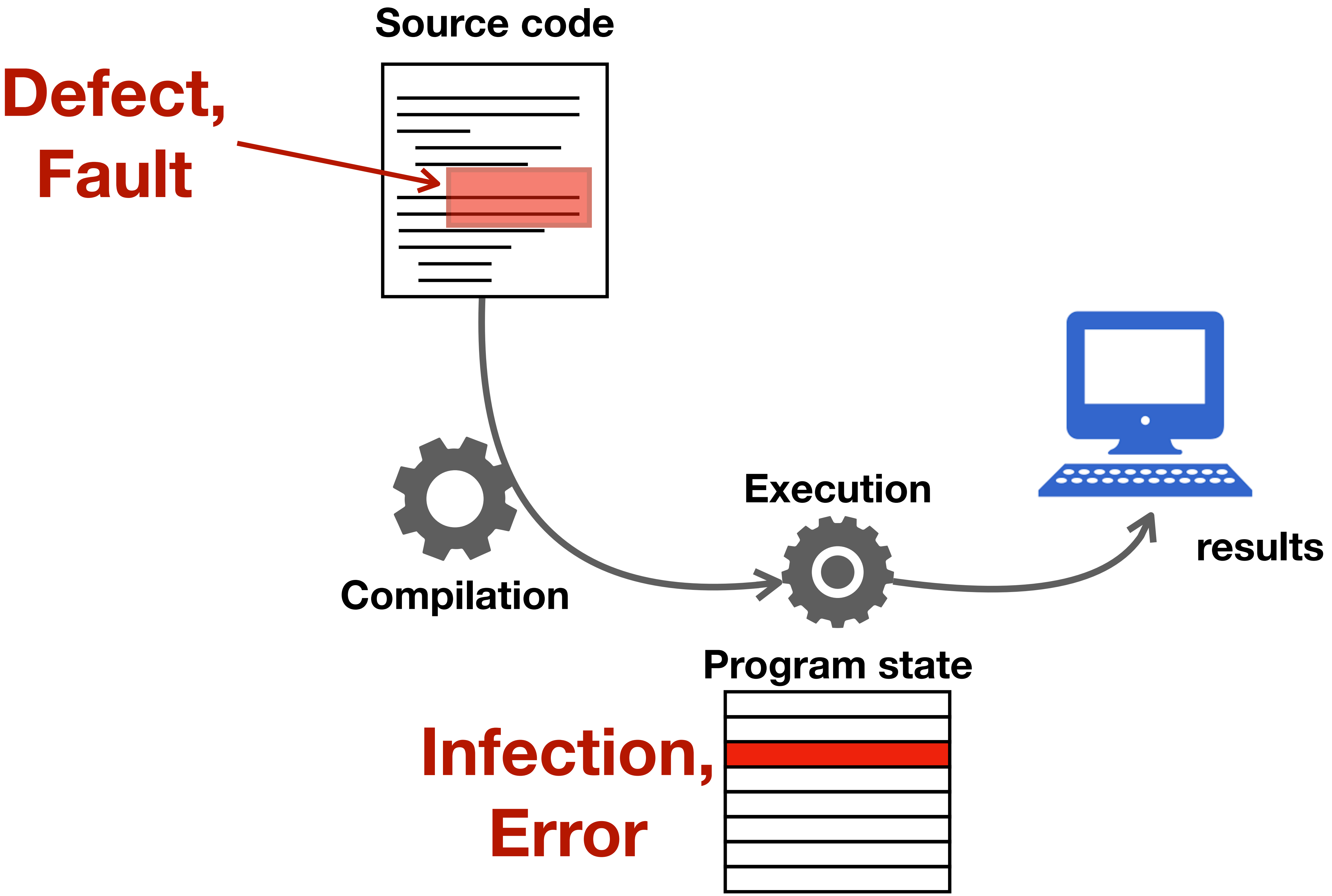
What’s a bug?

[Why programs Fail, Andreas Zeller, 2009]



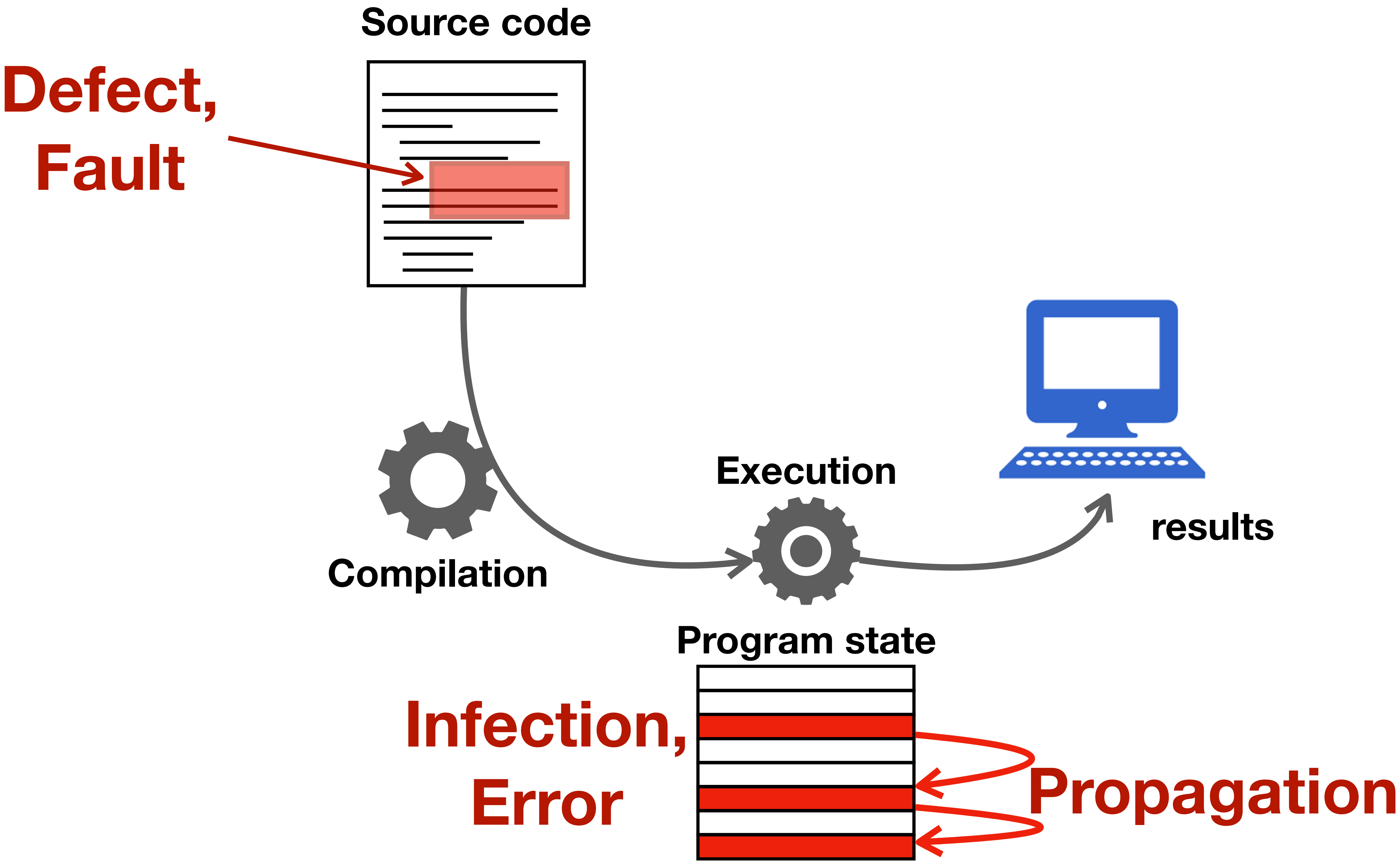
What’s a bug?

[Why programs Fail, Andreas Zeller, 2009]



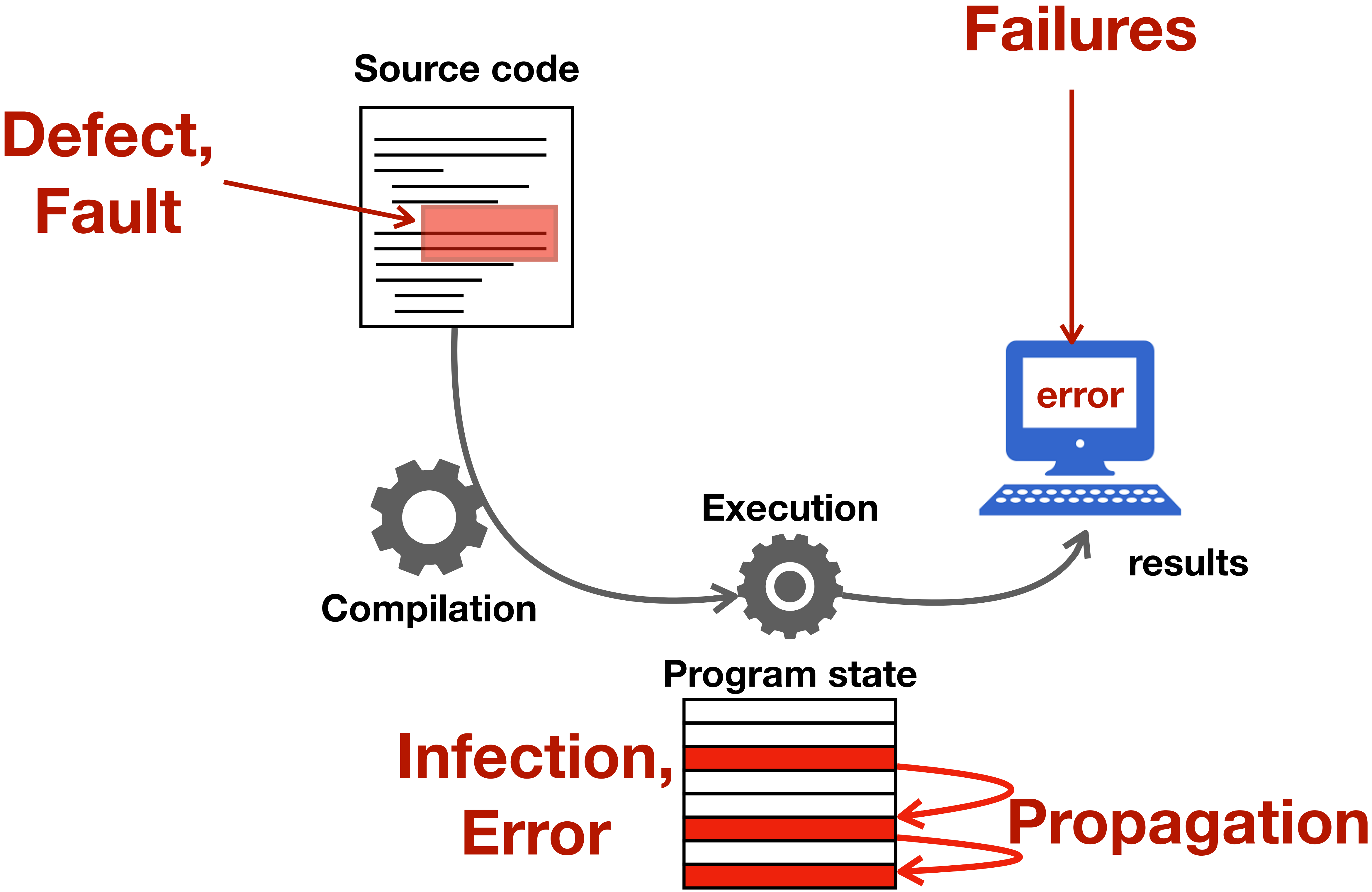
What’s a bug?

[Why programs Fail, Andreas Zeller, 2009]



What’s a bug?

[Why programs Fail, Andreas Zeller, 2009]



Terminology

[Why programs Fail, Andreas Zeller, 2009]

[Basic Concepts and Taxonomy of Dependable and Secure Computing, Avizienis et al., 2004]

- **Defect** or **Fault** : An incorrect program code
  - To have an effect, the defective code must be executed, sometimes under specific conditions
  - Can be at a single location or at many places (design or architectural **flaws**)
- **Infection** or **Error** : An incorrect program state
  - The state refers to all the system’s state: the program state + the execution state
  - An error may remain latent (no effect and undetected)
- **Propagation** : The infection spreads
  - The infected state is accessed by the program and its execution infrastructure
  - It infects more state as the execution progresses
  - Propagation might stop, be masked or fixed by other actions during the execution
- **Failure** : An observable incorrect program behaviour

Debugging

1 2 3 4 5 6

Definition

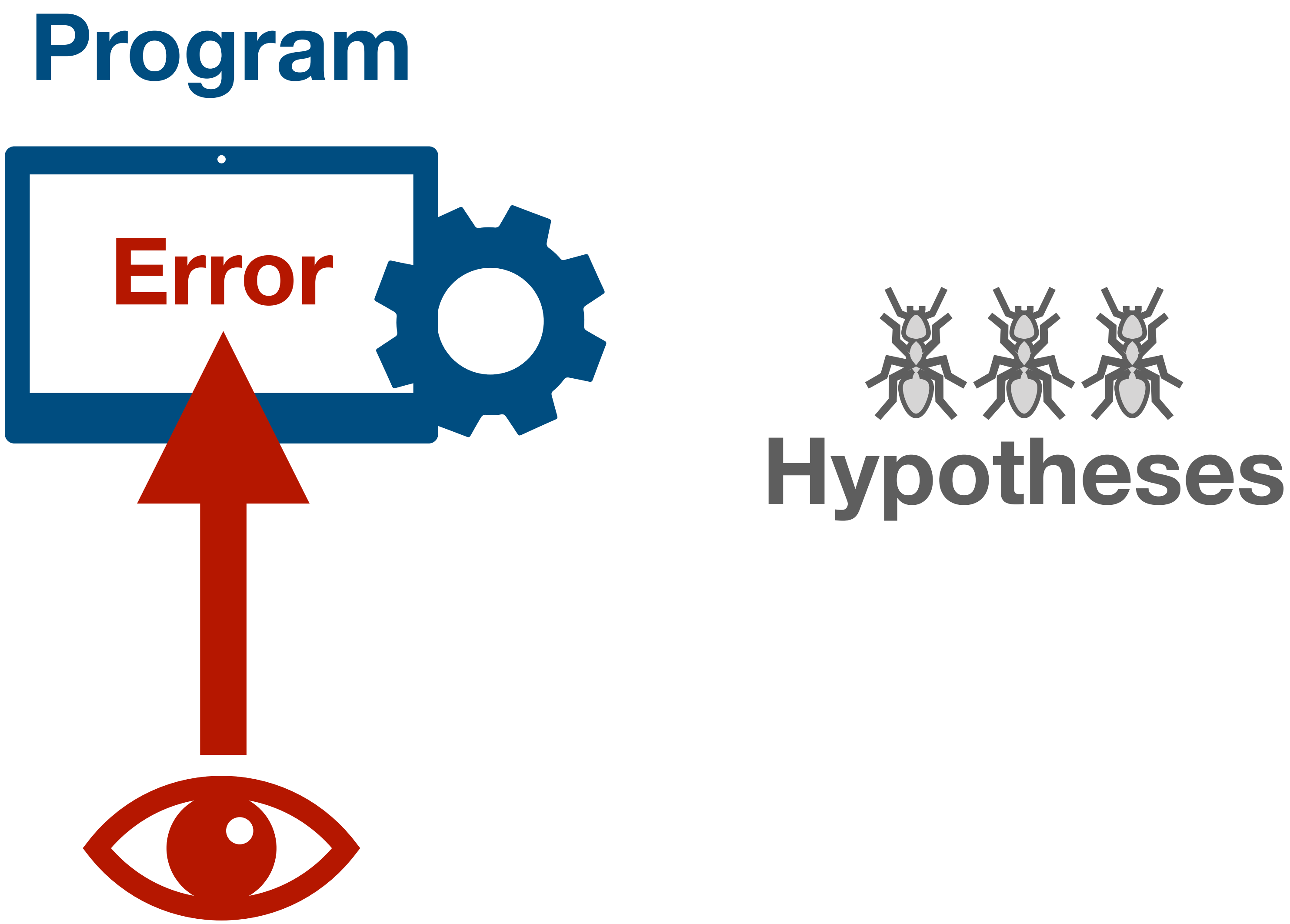
- **Debugging:**
  tracking and fixing defects in software systems.

- **Different steps**:
  - Observation of an error (a system failure)
  - Controlled reproduction of that error
  - Comprehension of the cause (defects/faults or flaws) of the error
  - Correction of the defect
  - Validation of the defect correction (tests)

The classical debugging cycle

Program
Error

The classical debugging cycle

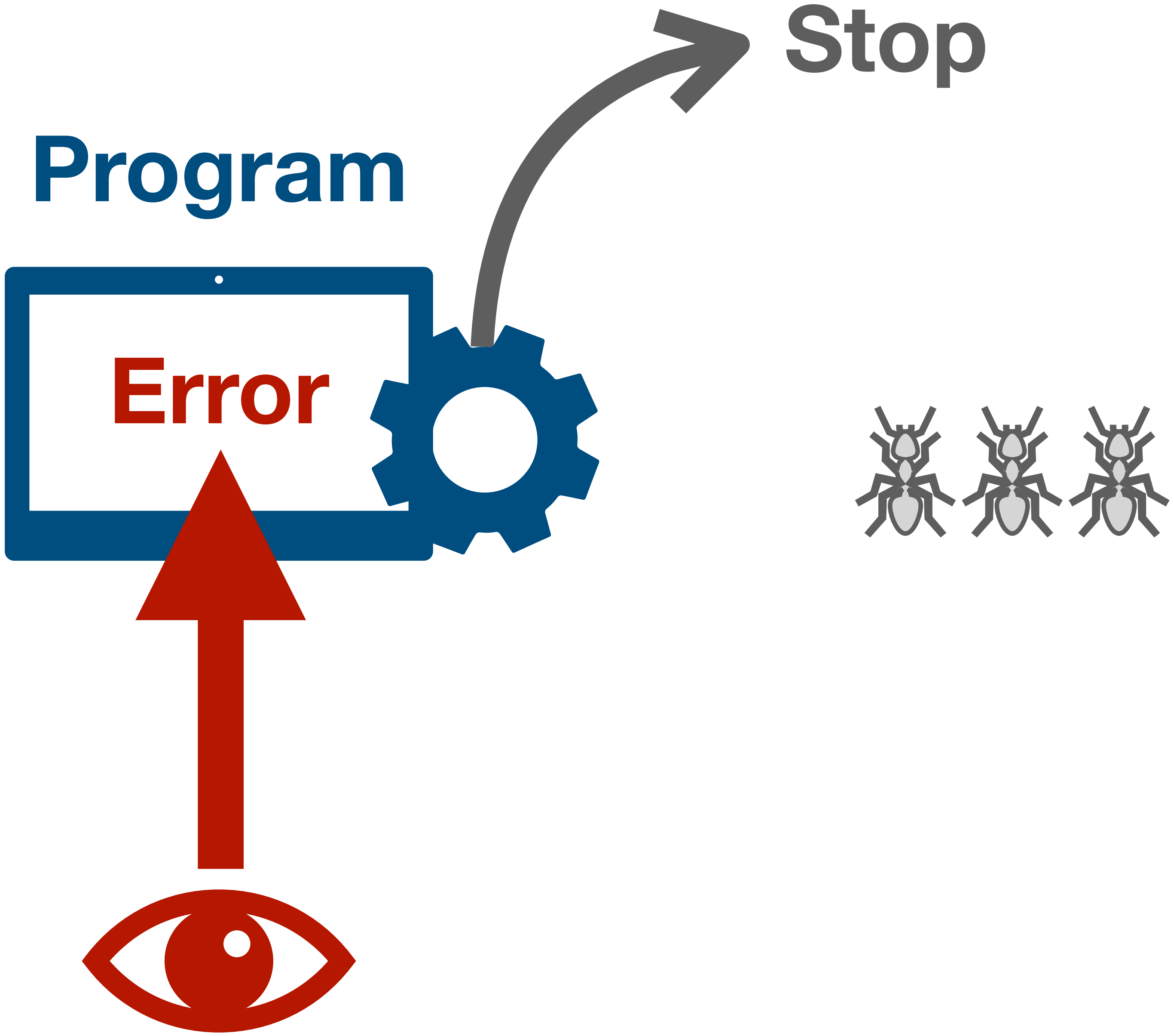


Program

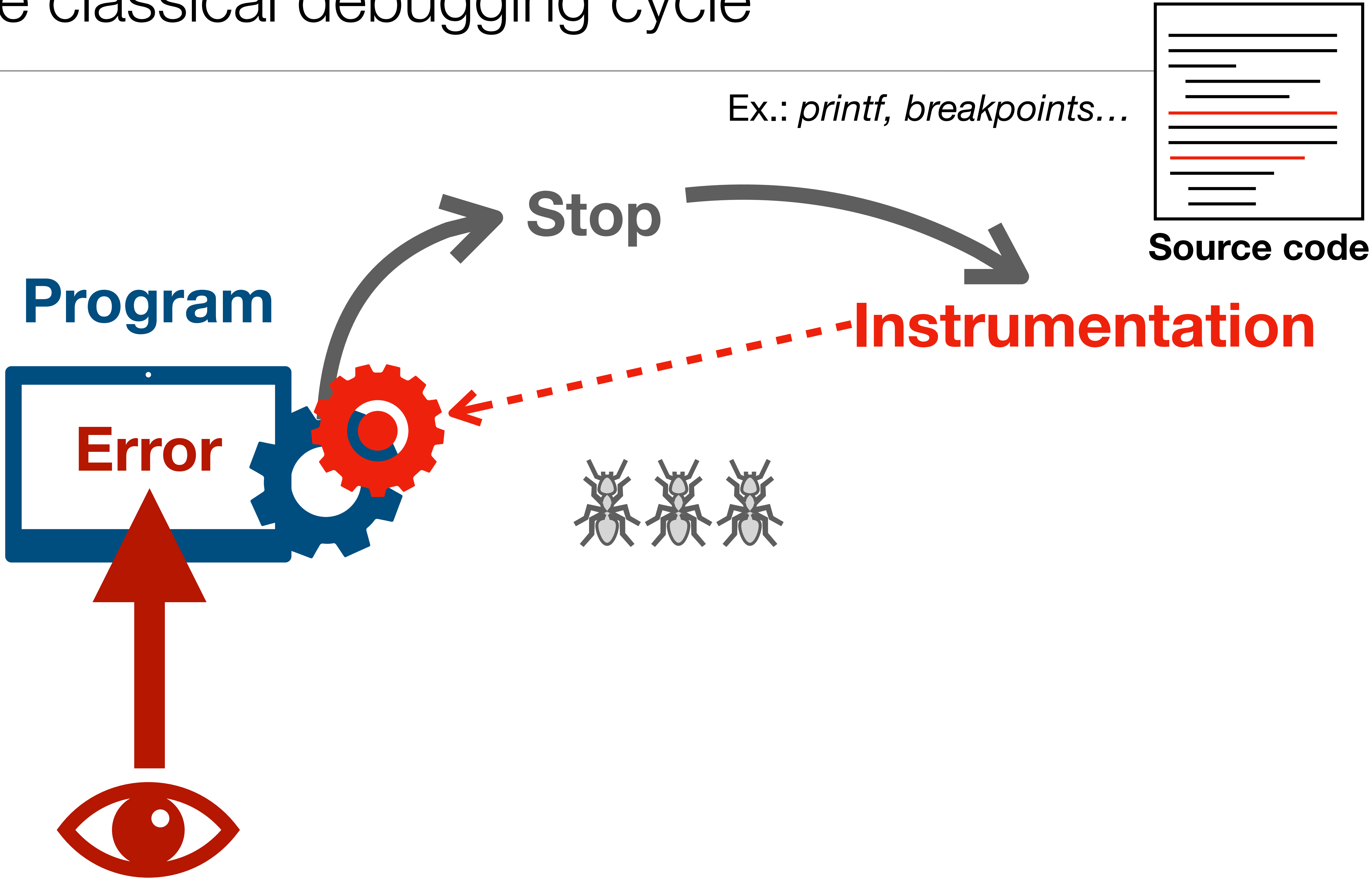
Error

Hypotheses

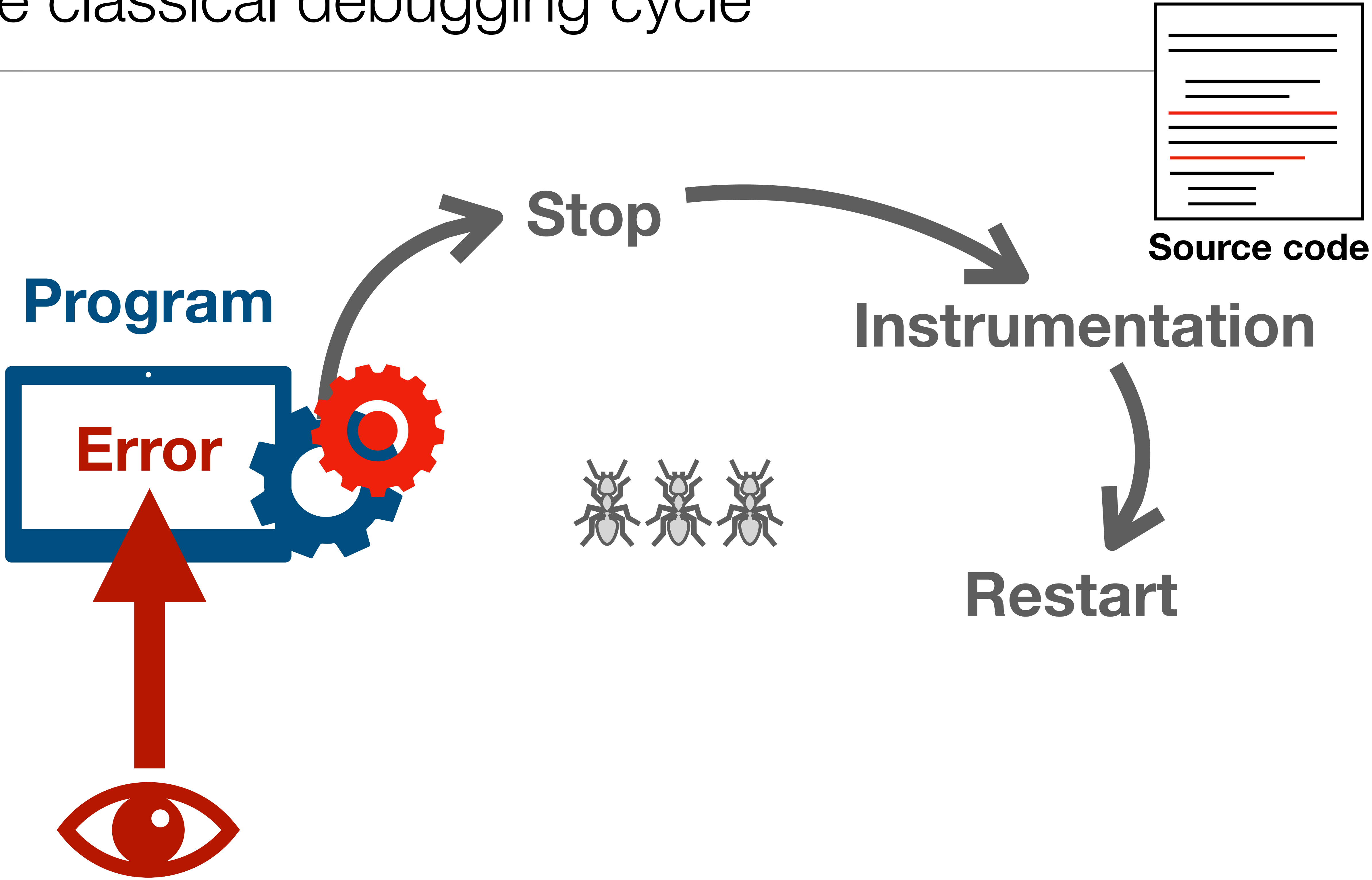
The classical debugging cycle



The classical debugging cycle



The classical debugging cycle



Source code

Stop

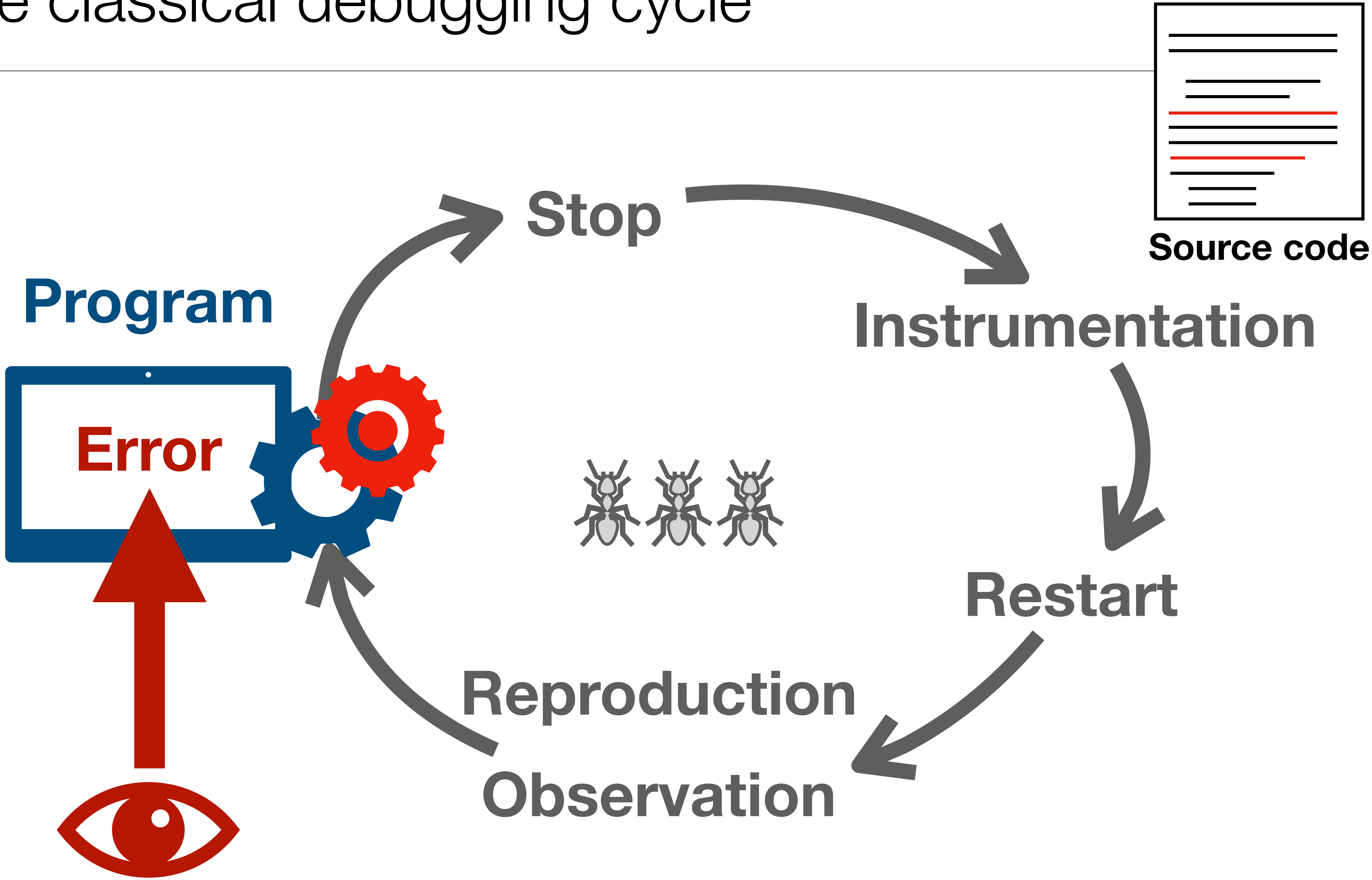
Program

Error

Instrumentation

Restart

The classical debugging cycle



Source code

Program

Error

Stop

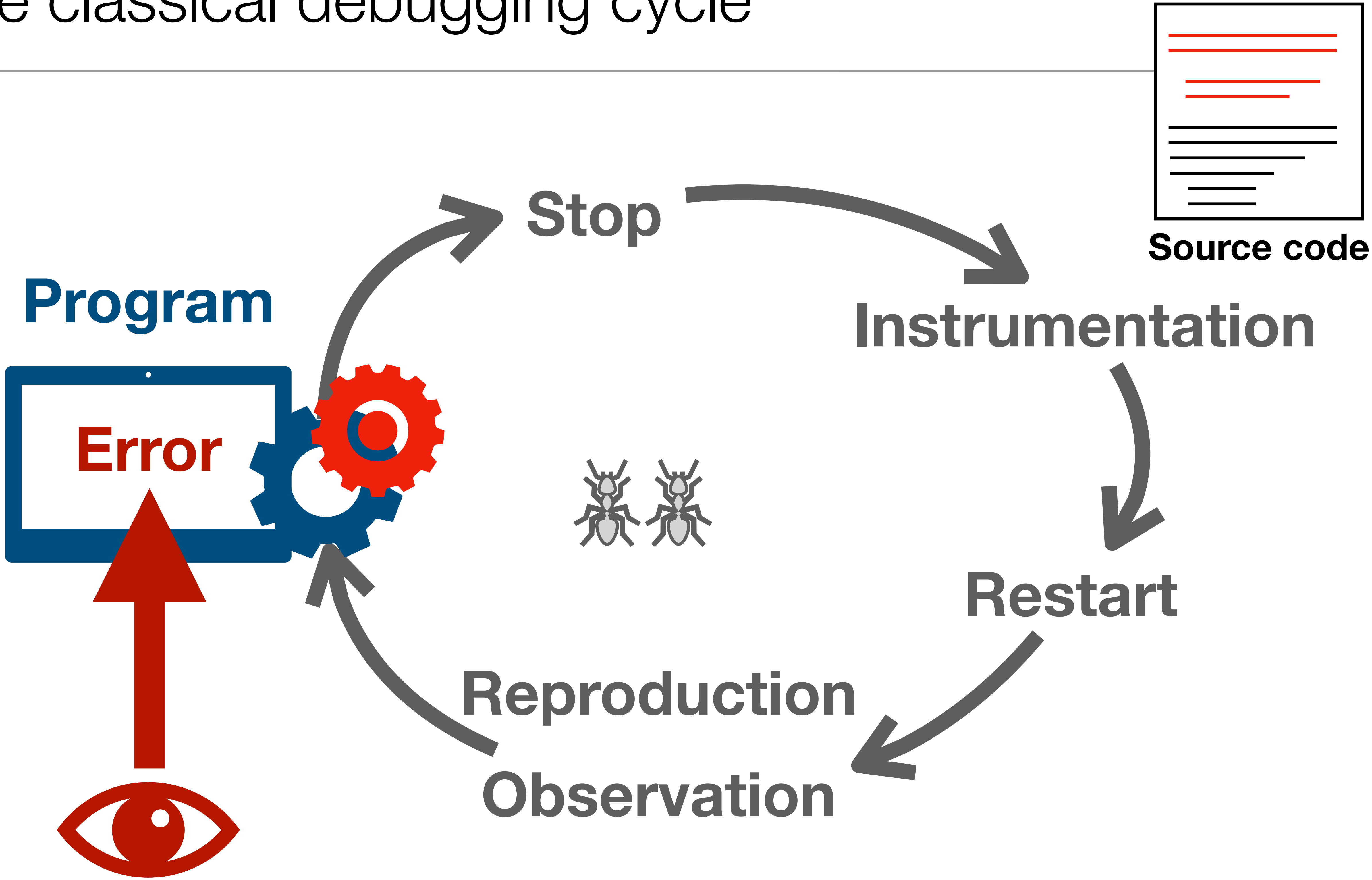
Instrumentation

Restart

Reproduction

Observation

The classical debugging cycle



Source code

Program

Error

Stop

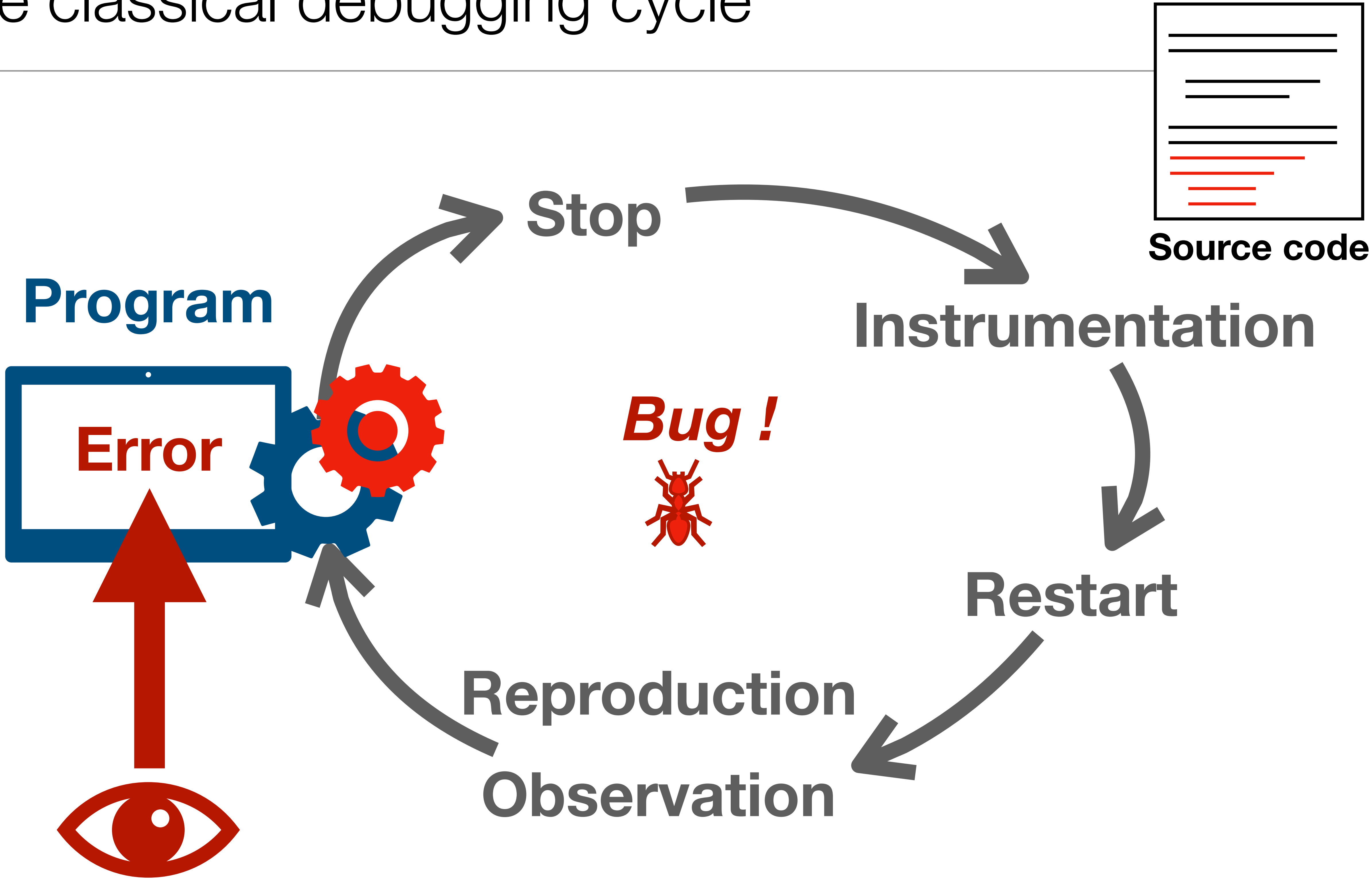
Instrumentation

Restart

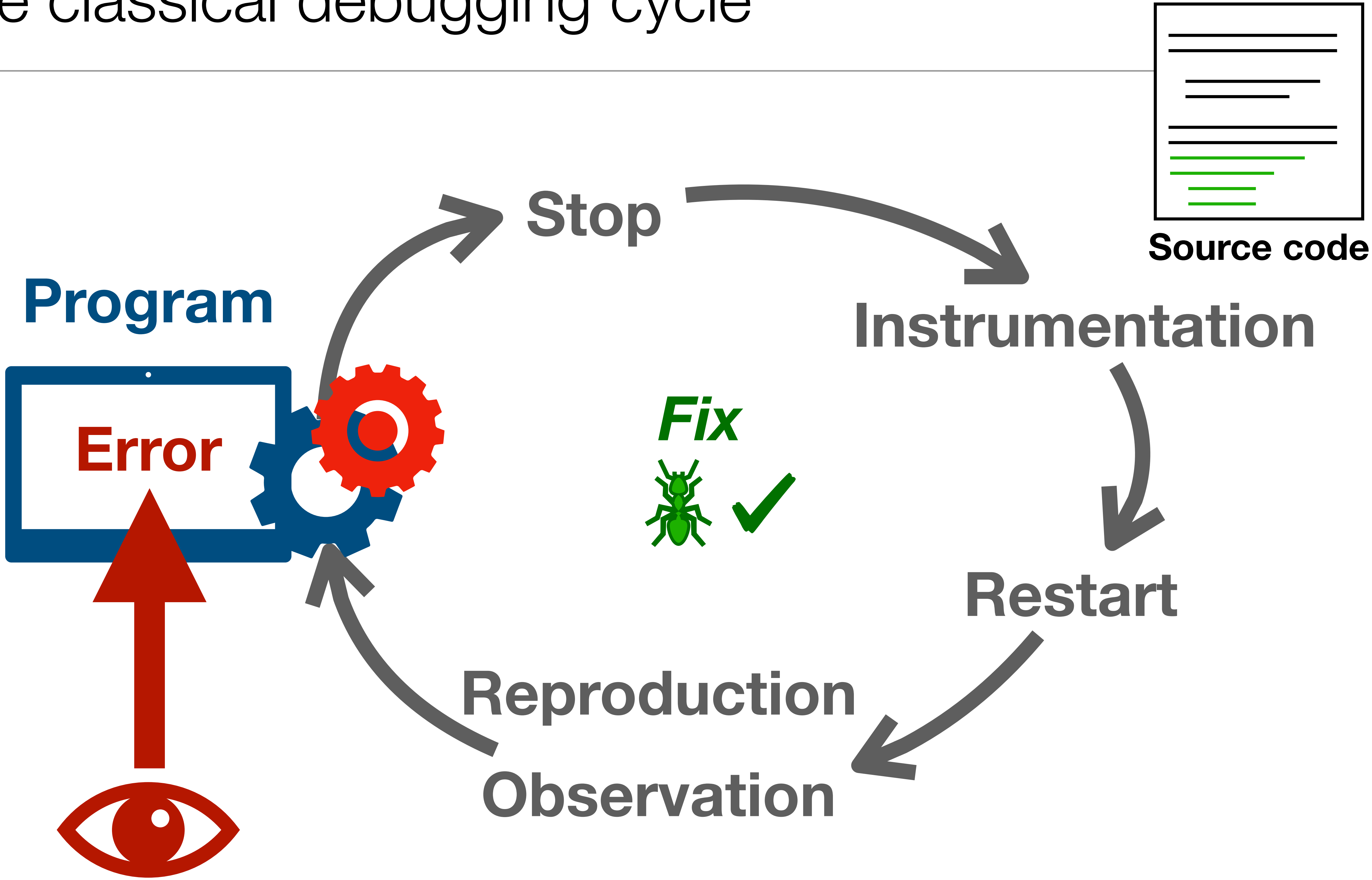
Reproduction

Observation

The classical debugging cycle



The classical debugging cycle



Source code

Program

Error

Stop

Instrumentation

Restart

Reproduction

Observation

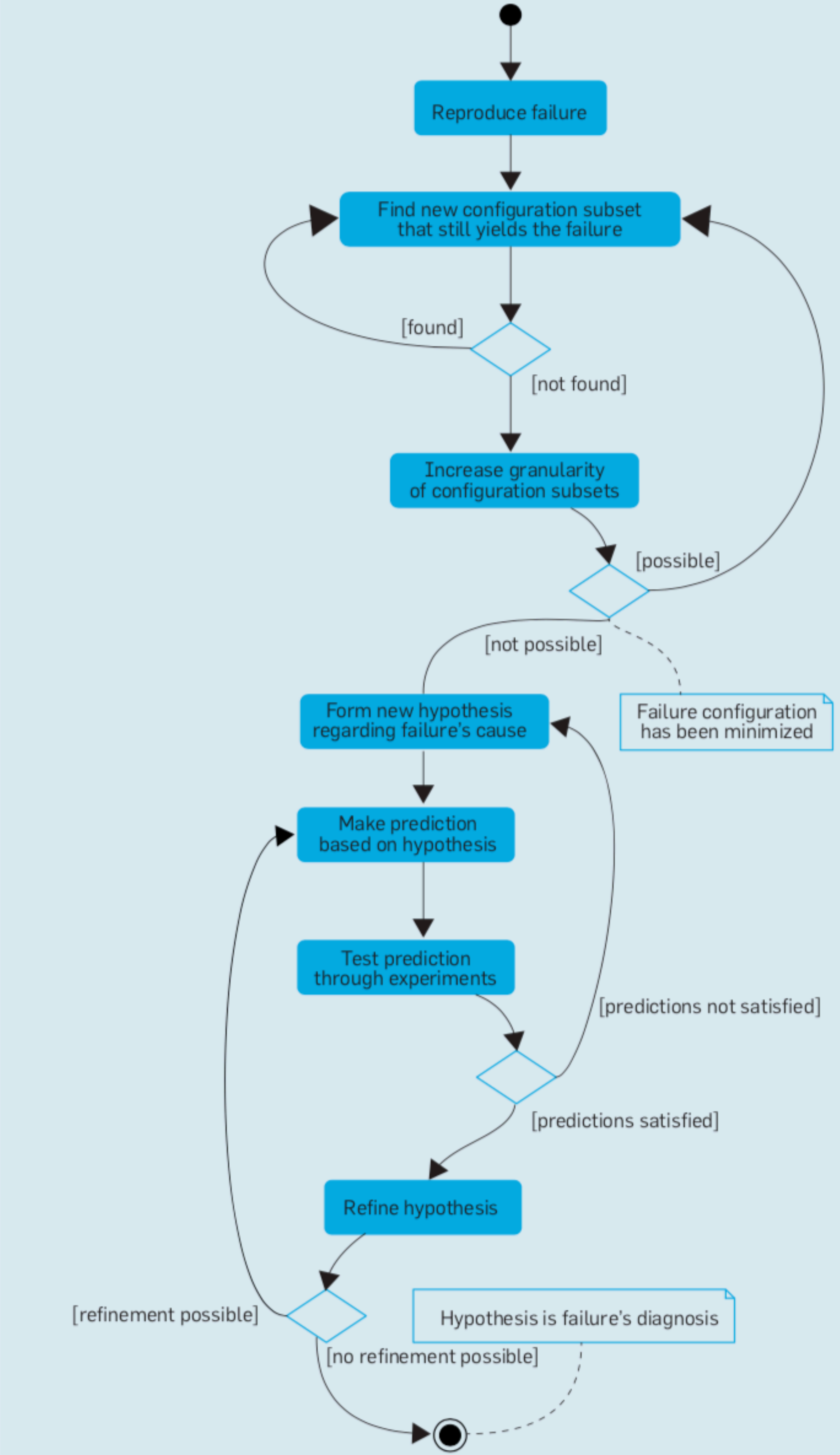
Fix

Systematic debugging

A process for systematic debugging.

Figure 1 from
[Modern Debugging: The Art of Finding a Needle in a Haystack, Diomidis Spinellis, 2018]

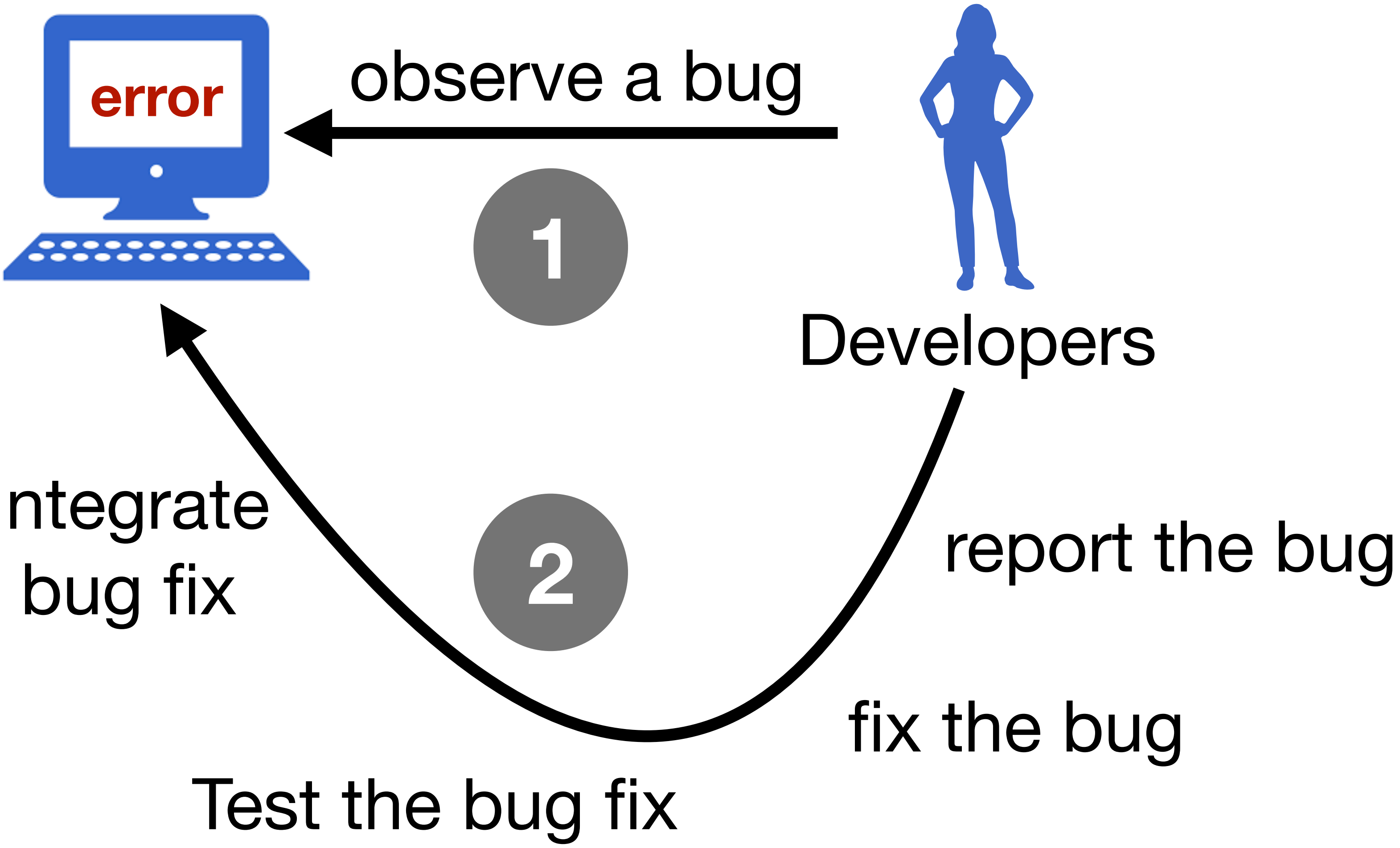
Also called
« **Simplified scientific method** »


Reproduce failure
Find new configuration subset that still yields the failure
[found]
[not found]
Increase granularity of configuration subsets
[possible]
[not possible]
Failure configuration has been minimized
Form new hypothesis regarding failure's cause
Make prediction based on hypothesis
Test prediction through experiments
[predictions not satisfied]
[predictions satisfied]
Refine hypothesis
[refinement possible]
Hypothesis is failure's diagnosis
[no refinement possible]

Debugging in the industry

1 2 3 4 5 6

Debugging in the industry: an illustration



observe a bug

1

Developers

Integrate
bug fix

2

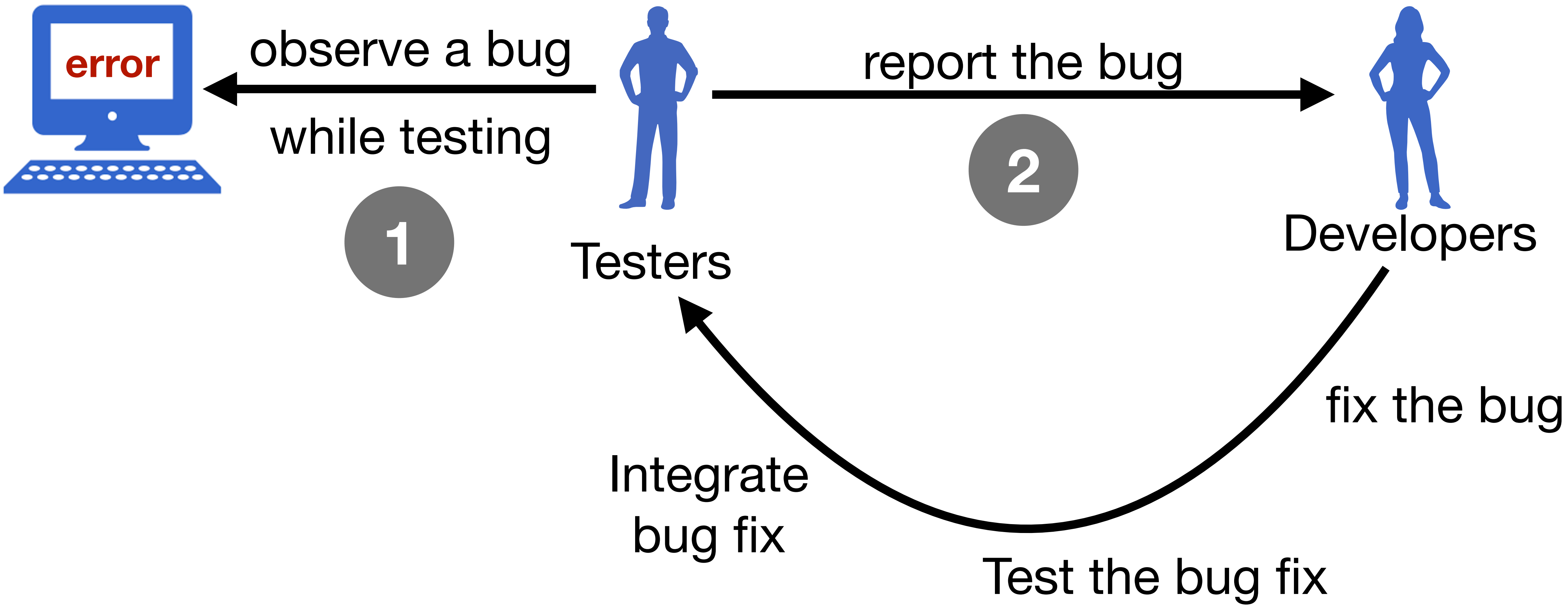
report the bug

fix the bug

Test the bug fix

During development $

Debugging in the industry: an illustration



observe a bug

while testing

Testers

report the bug

Developers

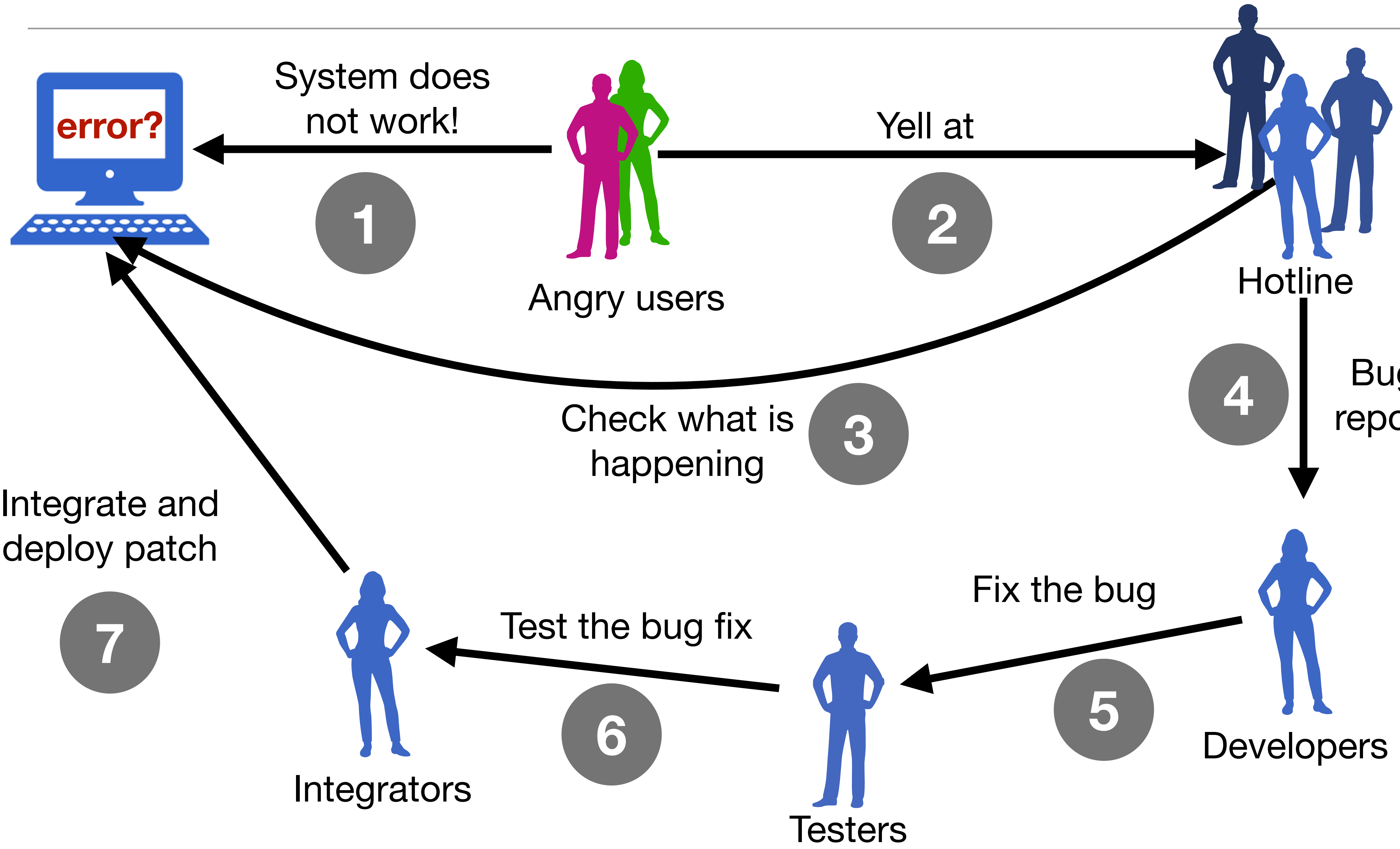
fix the bug

Test the bug fix

Integrate
bug fix

During Testing $$

Debugging in the industry: an illustration



System does not work!

1

Angry users

Yell at

2

Hotline

Check what is happening

3

4

Bug report

Developers

Fix the bug

5

Testers

Test the bug fix

6

Integrators

Integrate and deploy patch

7

After deployment $$$

Stakeholders

- Different stakeholders may observe failures in their system, for example:
  - **Users**
    - The system does not behave as they expect, does not do what they want : to them the system does not work
  - **Hotline technicians/engineers**
    - They are often in first line when users report their system does not work
    - They often have the responsibility to provide immediate help to users
    - They often decide if the users' problem is a system failure that must be reported
  - **Testers**
    - They test the system: if tests fail or produce errors, then there is a problem
  - **Developers**
    - They observe problems when developing the system, they investigate and fix bugs

Bug reports (1)

- After observing a failure, stakeholders report the failure (commonly called the "bug")

- **A bug report contains facts:**
  - A thorough description of the symptoms
  - A precise description of how to reproduce the bug (if possible)
  - A criticality: how important is this problem, and why?

- **A discussion:**
  - About all previous points if anything requires to debate or more information
  - The bug investigation itself once it started

Bug reports (2): **this is hard!**

- **Explaining with precision and concision is hard**
  - Sometimes stakeholders do not understand very well:
    - What they see, what is happening, what is wrong…
  - Sometimes, it is not their job to accurately report the bug (*e.g.* users)
  - Describing symptoms and steps to reproduce a bug is tedious and difficult

- **The person who report the bug is not always the developer who fixes the bug!**

Bug reports (3): examples

> « When we start the program it does not work »

Bug reports (3): examples

✗ « When we start the program it does not work »

« At startup, the program is frozen and cannot be used»

Bug reports (3): examples

✗ « When we start the program it does not work »

✗ « At startup, the program is frozen and cannot be used»

« At startup, instead of prompting the user input, the program freezes and cannot be used.

Reproduction steps: start the program and select interactive mode. The program does not show user input and is unusable. »

Bug reports (3): examples

✗ « When we start the program it does not work »

✗ « At startup, the program is frozen and cannot be used»

✓ **Better**

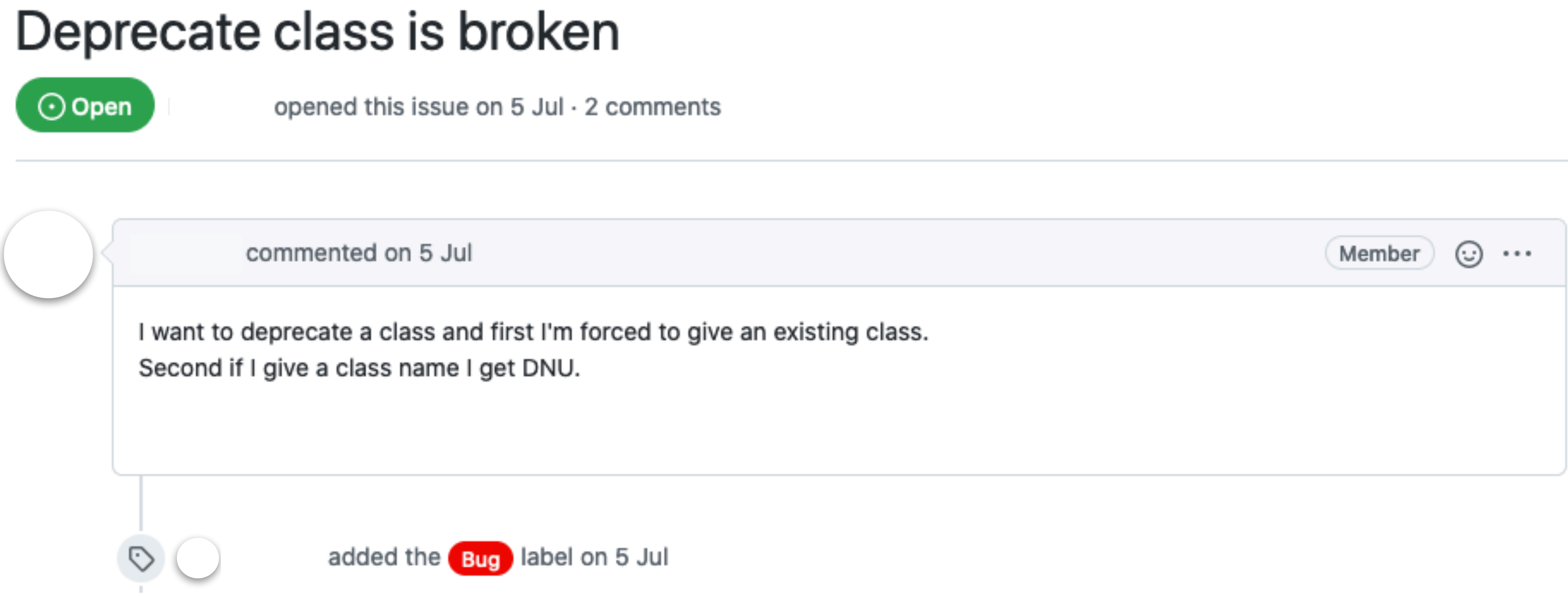
« At startup, instead of prompting the user input, the program freezes and cannot be used. **More details: precision**

Reproduction steps: start the program and select interactive mode. The program does not show user input and is unusable. »

Reproduction steps

Bug reports (4): real examples

https://github.com/pharo-project/pharo/issues



Deprecate class is broken

opened this issue on 5 Jul · 2 comments

commented on 5 Jul

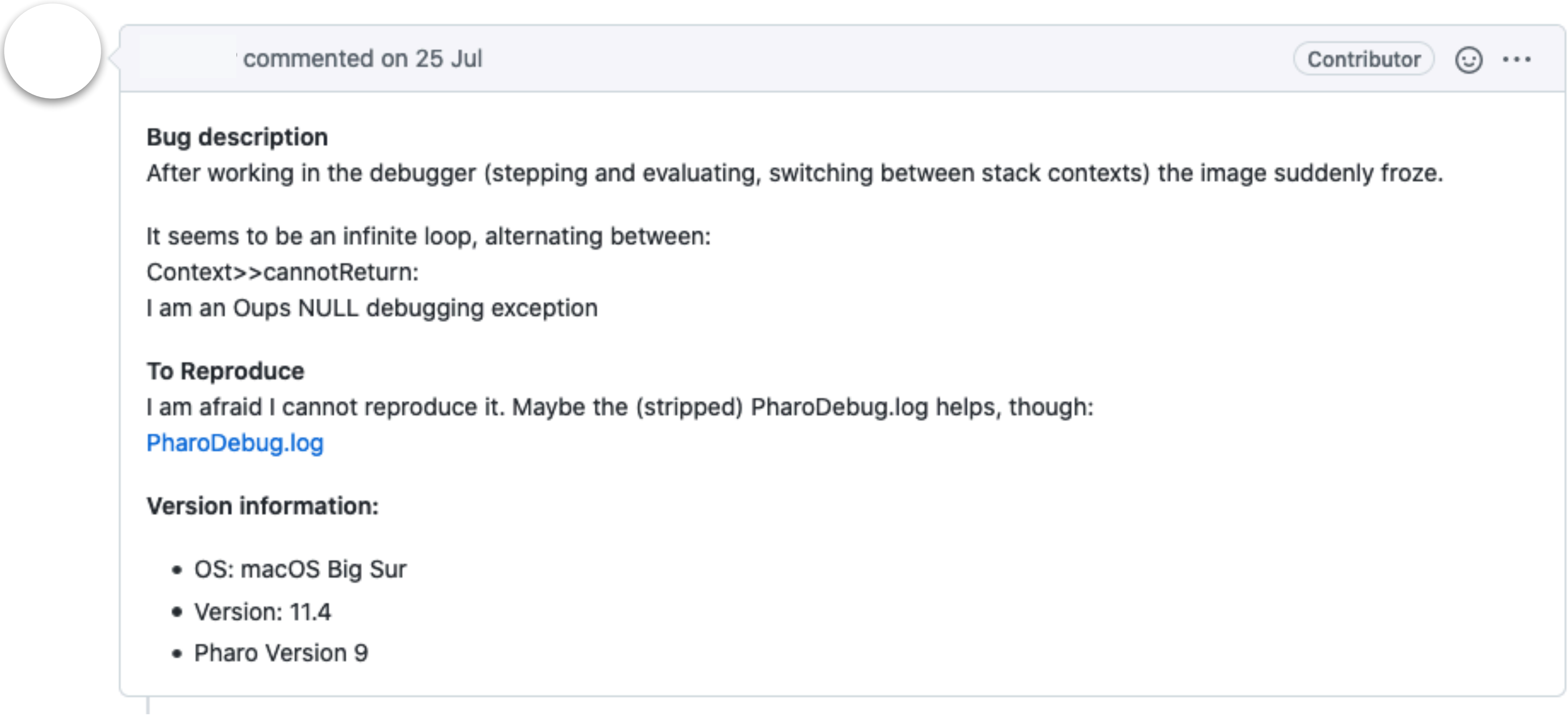
I want to deprecate a class and first I'm forced to give an existing class.
Second if I give a class name I get DNU.

added the Bug label on 5 Jul

✗

Bug reports (5): real examples

https://github.com/pharo-project/pharo/issues



commented on 25 Jul

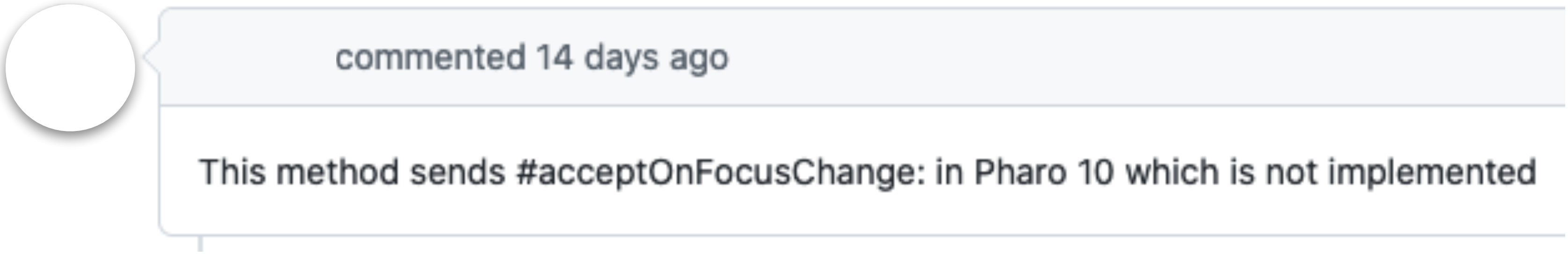
Bug description
After working in the debugger (stepping and evaluating, switching between stack contexts) the image suddenly froze.

It seems to be an infinite loop, alternating between:
Context>>cannotReturn:
I am an Oups NULL debugging exception

To Reproduce
I am afraid I cannot reproduce it. Maybe the (stripped) PharoDebug.log helps, though:
PharoDebug.log

Version information:

- OS: macOS Big Sur
- Version: 11.4
- Pharo Version 9



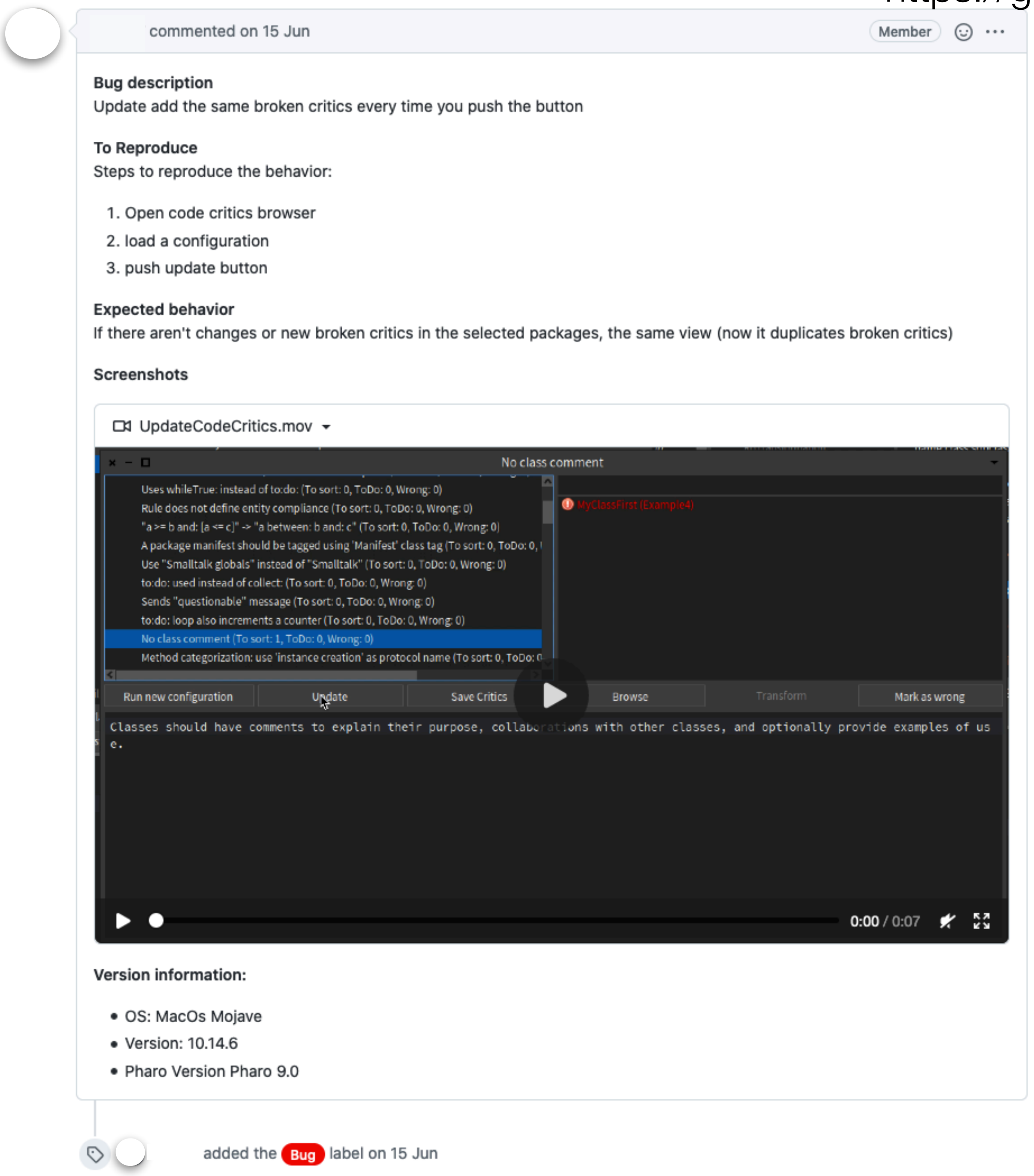
commented 14 days ago

This method sends #acceptOnFocusChange: in Pharo 10 which is not implemented

✓

Bug reports (6): real examples

https://github.com/pharo-project/pharo/issues



commented on 15 Jun

Bug description
Update add the same broken critics every time you push the button

To Reproduce
Steps to reproduce the behavior:

1. Open code critics browser
2. load a configuration
3. push update button

Expected behavior
If there aren't changes or new broken critics in the selected packages, the same view (now it duplicates broken critics)

Screenshots

UpdateCodeCritics.mov

No class comment

```
Uses whileTrue: instead of to:do: (To sort: 0, ToDo: 0, Wrong: 0)
Rule does not define entity compliance (To sort: 0, ToDo: 0, Wrong: 0)
"a >= b and: [a <= c]" -> "a between: b and: c" (To sort: 0, ToDo: 0, Wrong: 0)
A package manifest should be tagged using 'Manifest' class tag (To sort: 0, ToDo: 0, 
Use "Smalltalk globals" instead of "Smalltalk" (To sort: 0, ToDo: 0, Wrong: 0)
to:do: used instead of collect: (To sort: 0, ToDo: 0, Wrong: 0)
Sends "questionable" message (To sort: 0, ToDo: 0, Wrong: 0)
to:do: loop also increments a counter (To sort: 0, ToDo: 0, Wrong: 0)
No class comment (To sort: 1, ToDo: 0, Wrong: 0)
Method categorization: use 'instance creation' as protocol name (To sort: 0, ToDo: 0
```

MyClassFirst (Example4)

```
Classes should have comments to explain their purpose, collaborations with other classes, and optionally provide examples of us
e.
```

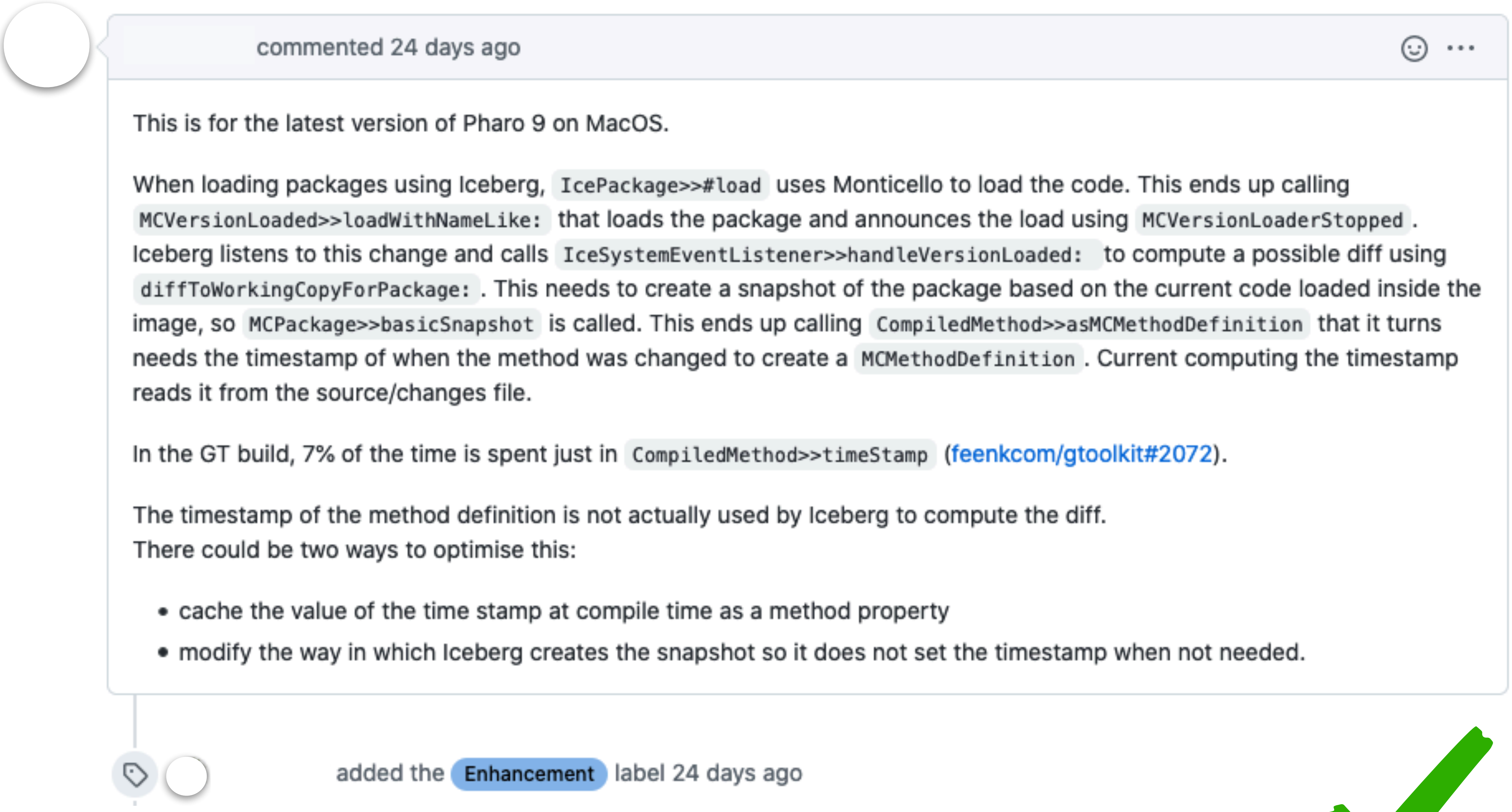
Version information:

- OS: MacOs Mojave
- Version: 10.14.6
- Pharo Version Pharo 9.0

added the Bug label on 15 Jun

Bug reports (7): real examples

https://github.com/pharo-project/pharo/issues



commented 24 days ago

This is for the latest version of Pharo 9 on MacOS.

When loading packages using Iceberg, `IcePackage>>#load` uses Monticello to load the code. This ends up calling `MCVersionLoaded>>loadWithNameLike:` that loads the package and announces the load using `MCVersionLoaderStopped`. Iceberg listens to this change and calls `IceSystemEventListener>>handleVersionLoaded:` to compute a possible diff using `diffToWorkingCopyForPackage:`. This needs to create a snapshot of the package based on the current code loaded inside the image, so `MCPackage>>basicSnapshot` is called. This ends up calling `CompiledMethod>>asMCMethodDefinition` that it turns needs the timestamp of when the method was changed to create a `MCMethodDefinition`. Current computing the timestamp reads it from the source/changes file.

In the GT build, 7% of the time is spent just in `CompiledMethod>>timeStamp` (feenkcom/gtoolkit#2072).

The timestamp of the method definition is not actually used by Iceberg to compute the diff.
There could be two ways to optimise this:

- cache the value of the time stamp at compile time as a method property
- modify the way in which Iceberg creates the snapshot so it does not set the timestamp when not needed.

added the Enhancement label 24 days ago

✓

Difficulties of debugging

1 2 3 4 5 6

General difficulties

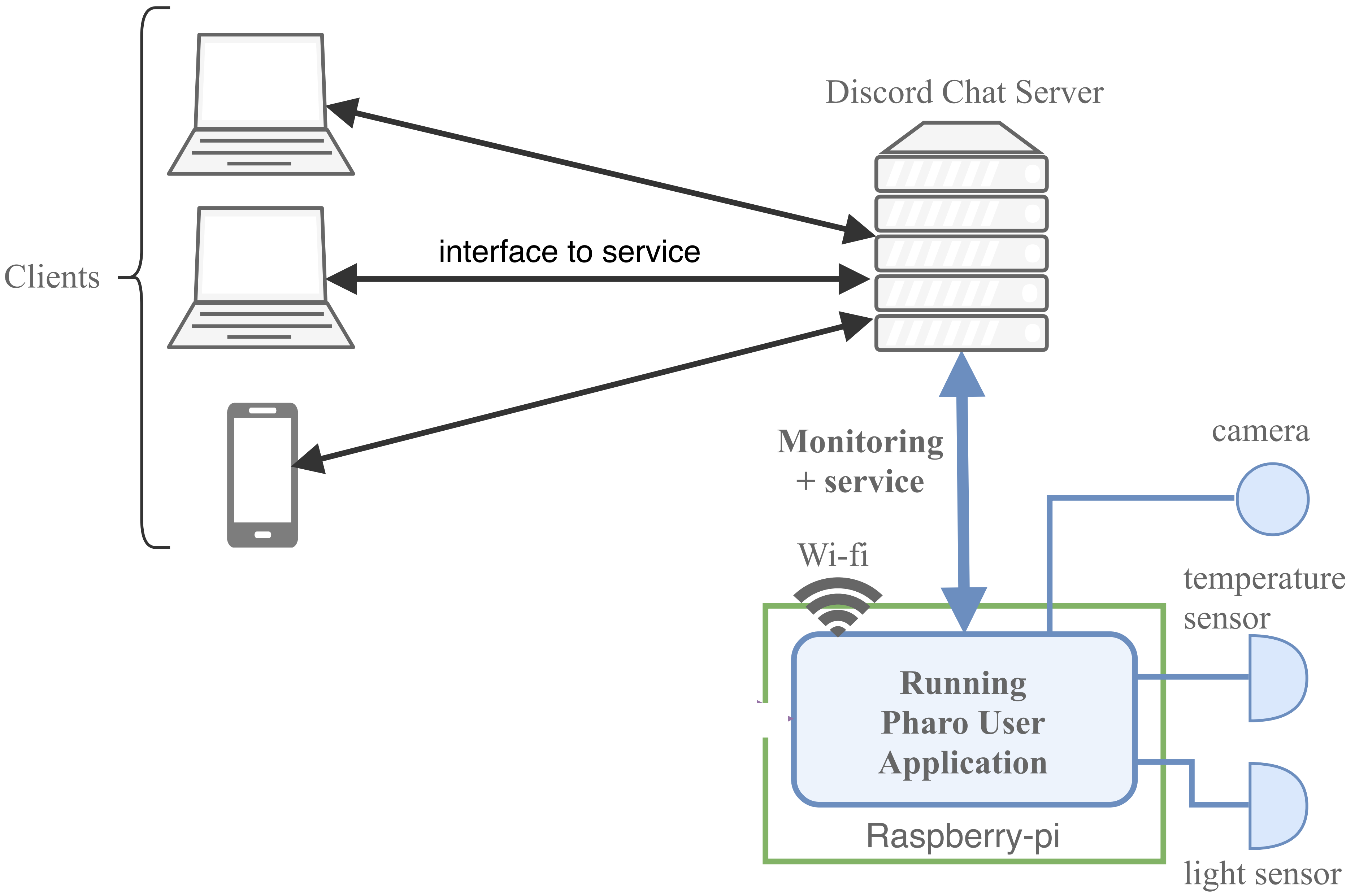
- Software systems are complex and heterogeneous
- Debugging tools are complex, hard to understand
- Describing failures is tedious:
  - Describing precisely their symptoms is hard
  - Describing how to reproduce them is annoying and painful, sometimes it is impossible

Distance source-symptom

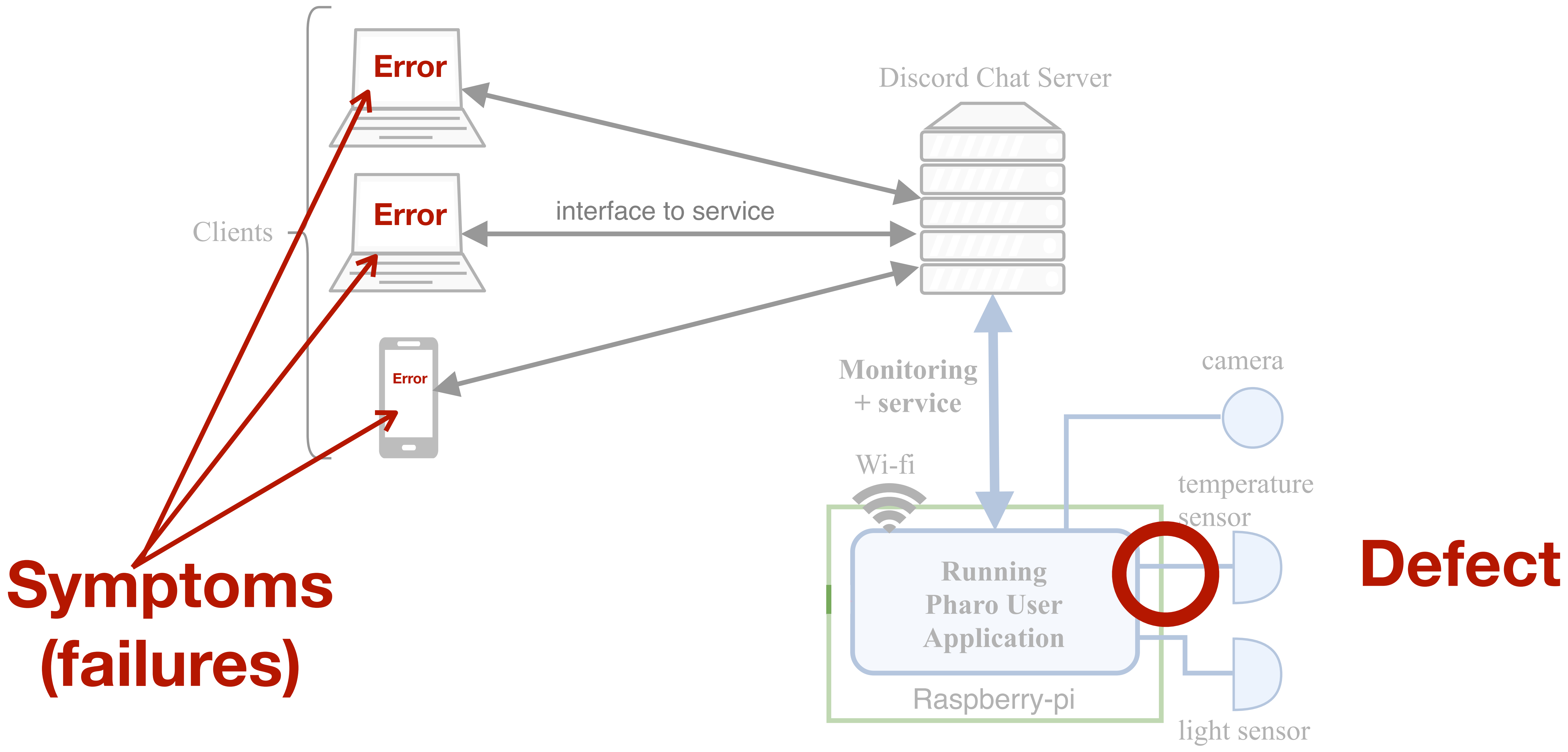
- **The symptom (the results of the error) that we observe is not the cause of the error: it is the defect that causes the error!**

- **The root cause of the defect can be distant from its observable effects**
    - The symptoms can occur in another source code location than its cause
    - The error can occur long before its symptoms are observed/observable

Distance source-symptom: example



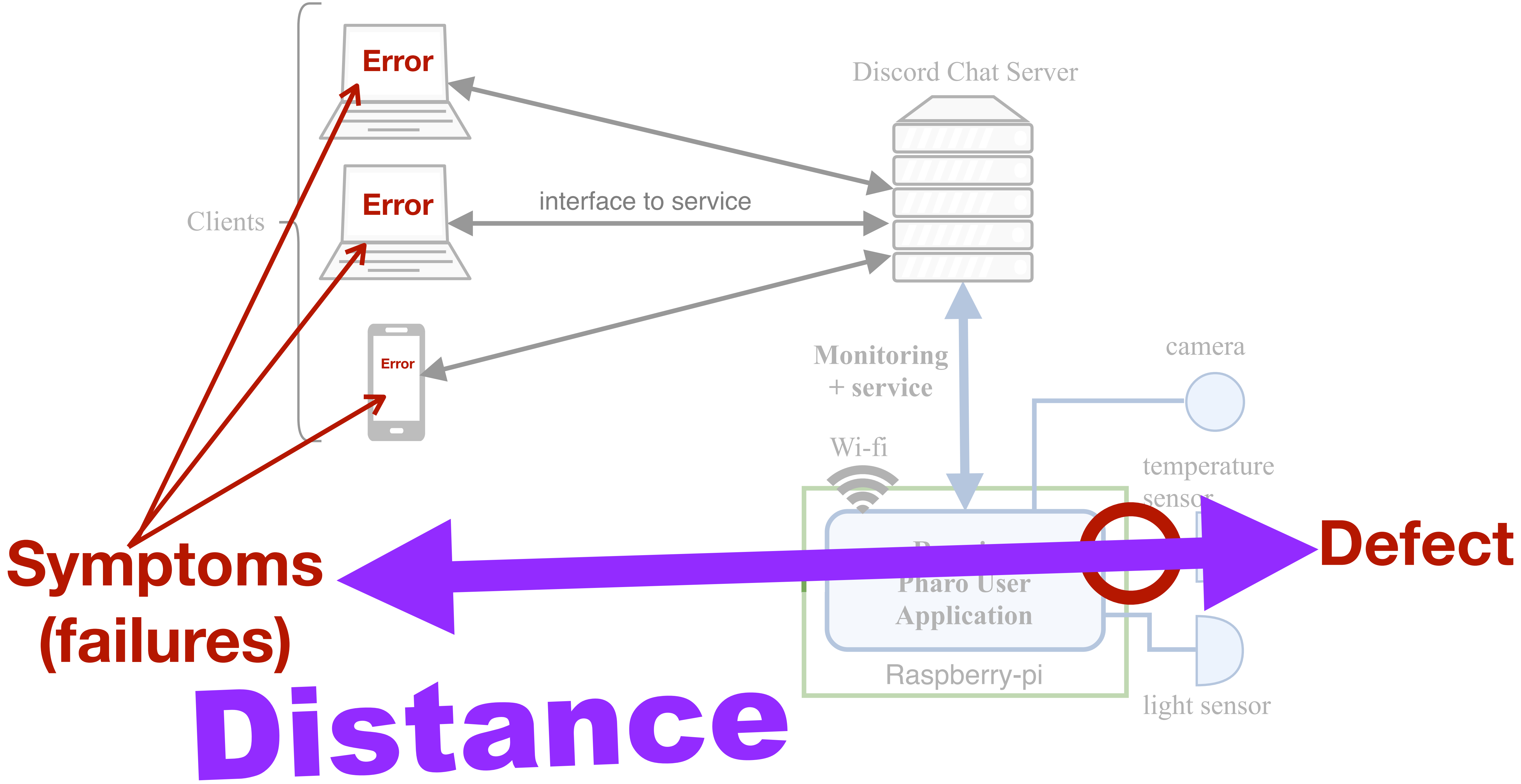
Distance source-symptom: example



Symptoms (failures)

Defect

Distance source-symptom: example

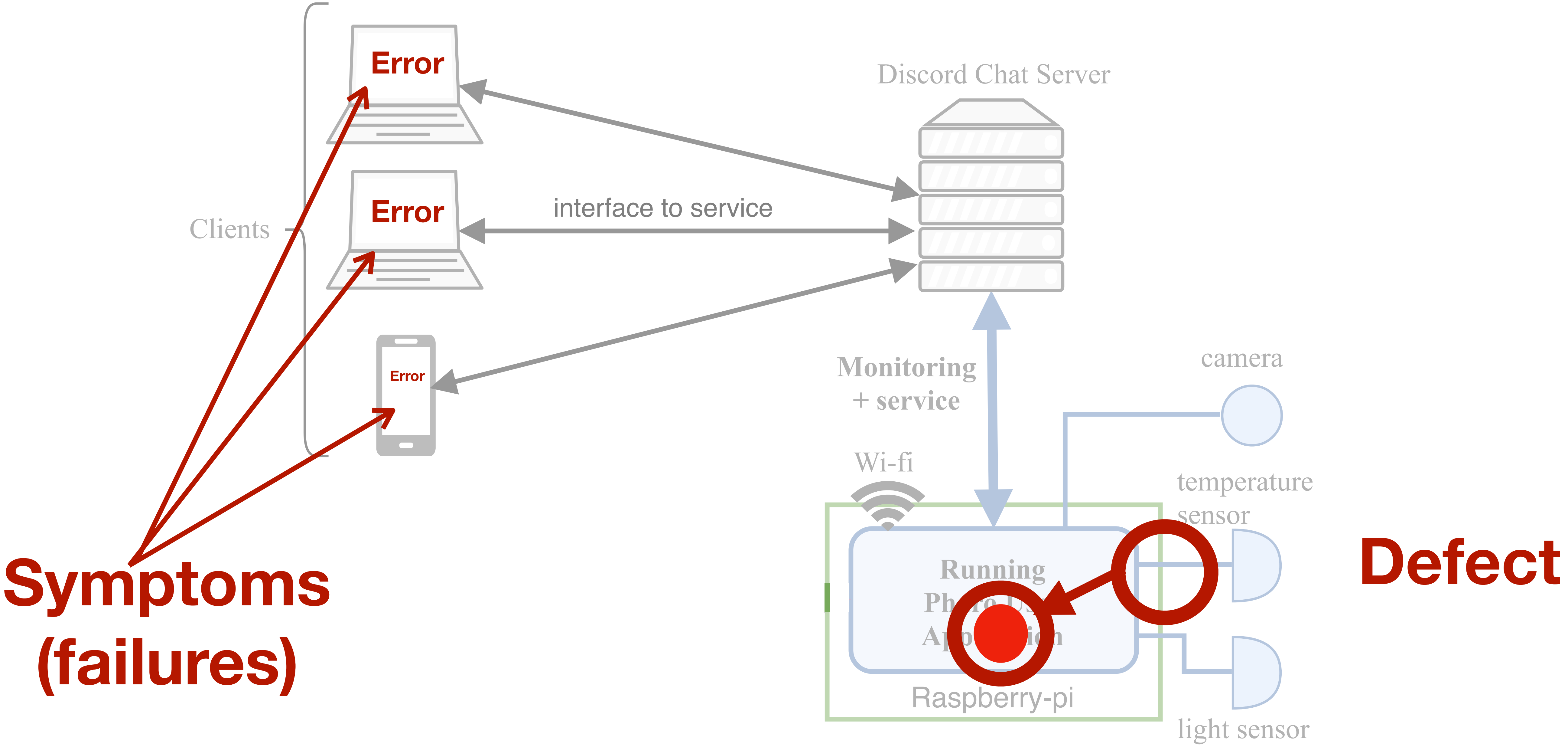


Symptoms (failures)

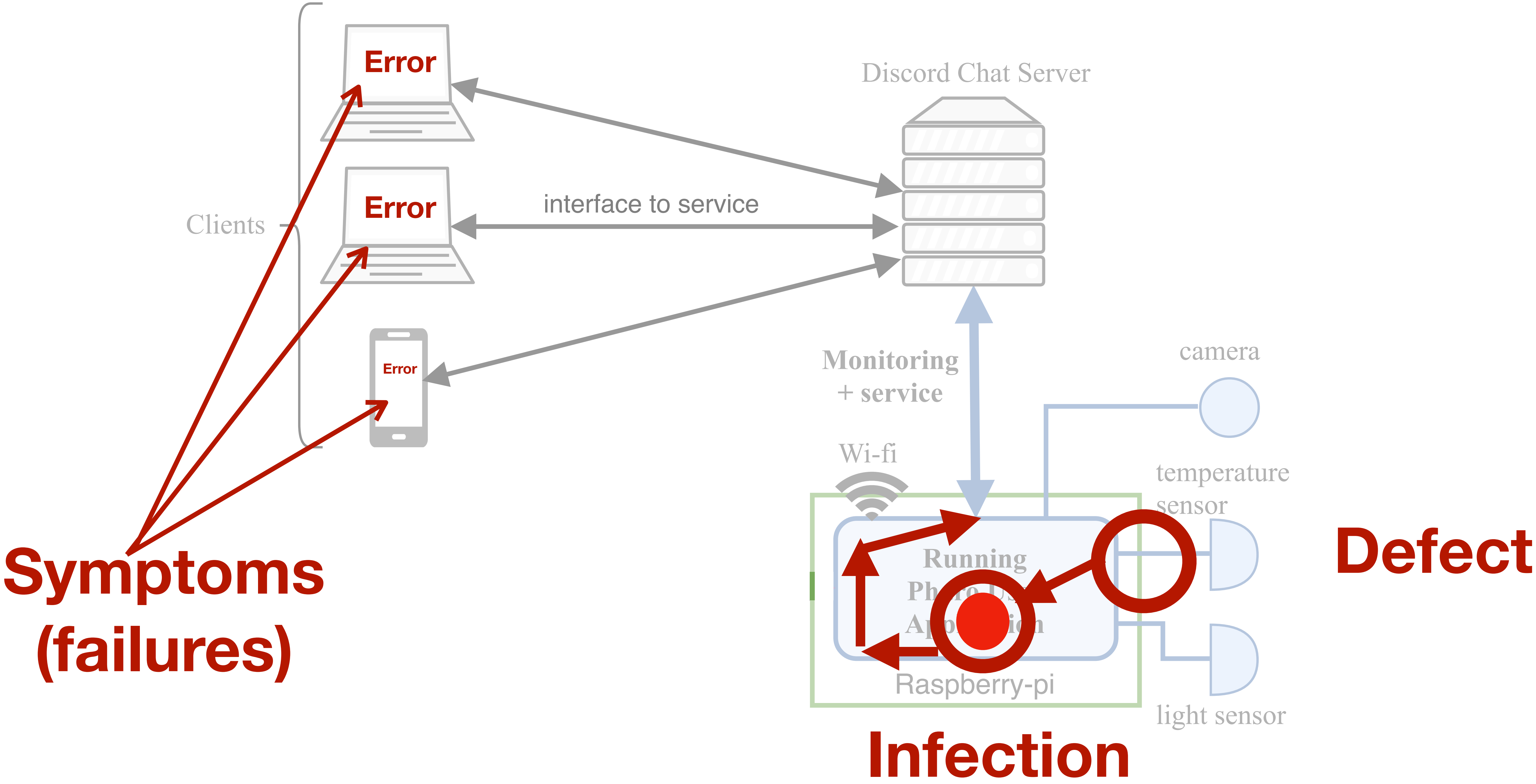
Defect

Distance

Distance source-symptom: example



Distance source-symptom: example

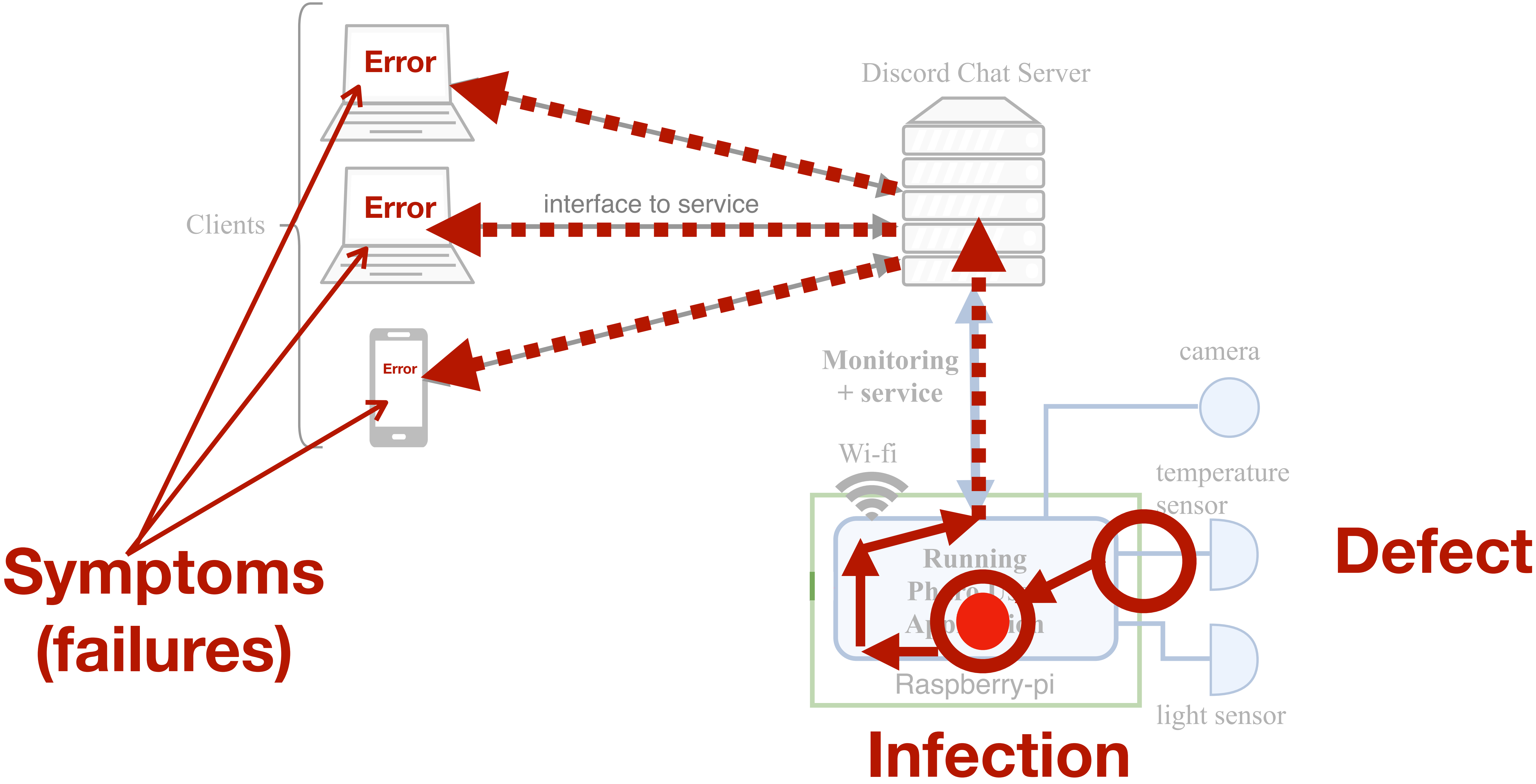


Symptoms (failures)

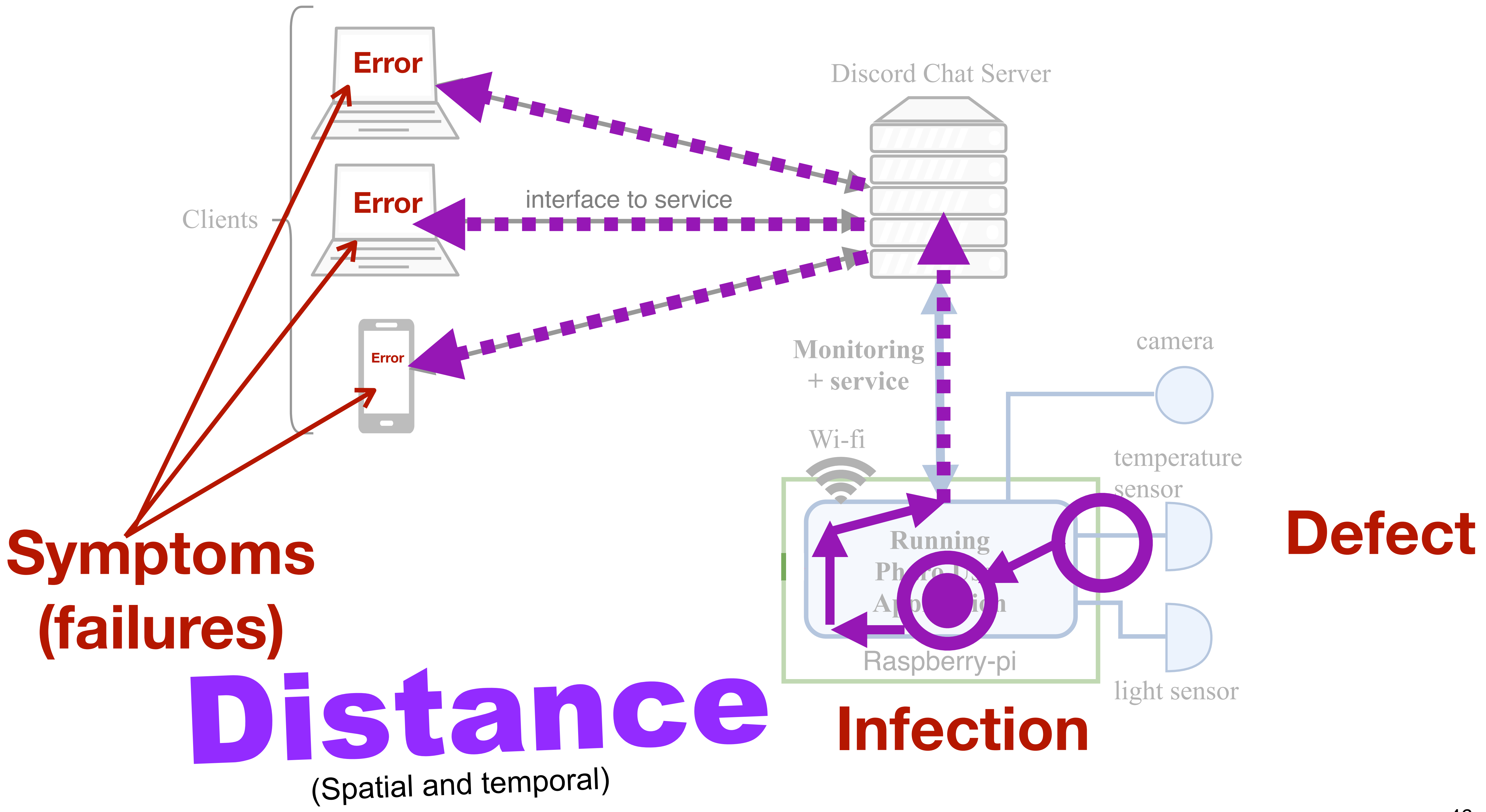
Defect

Infection

Distance source-symptom: example



Distance source-symptom: example



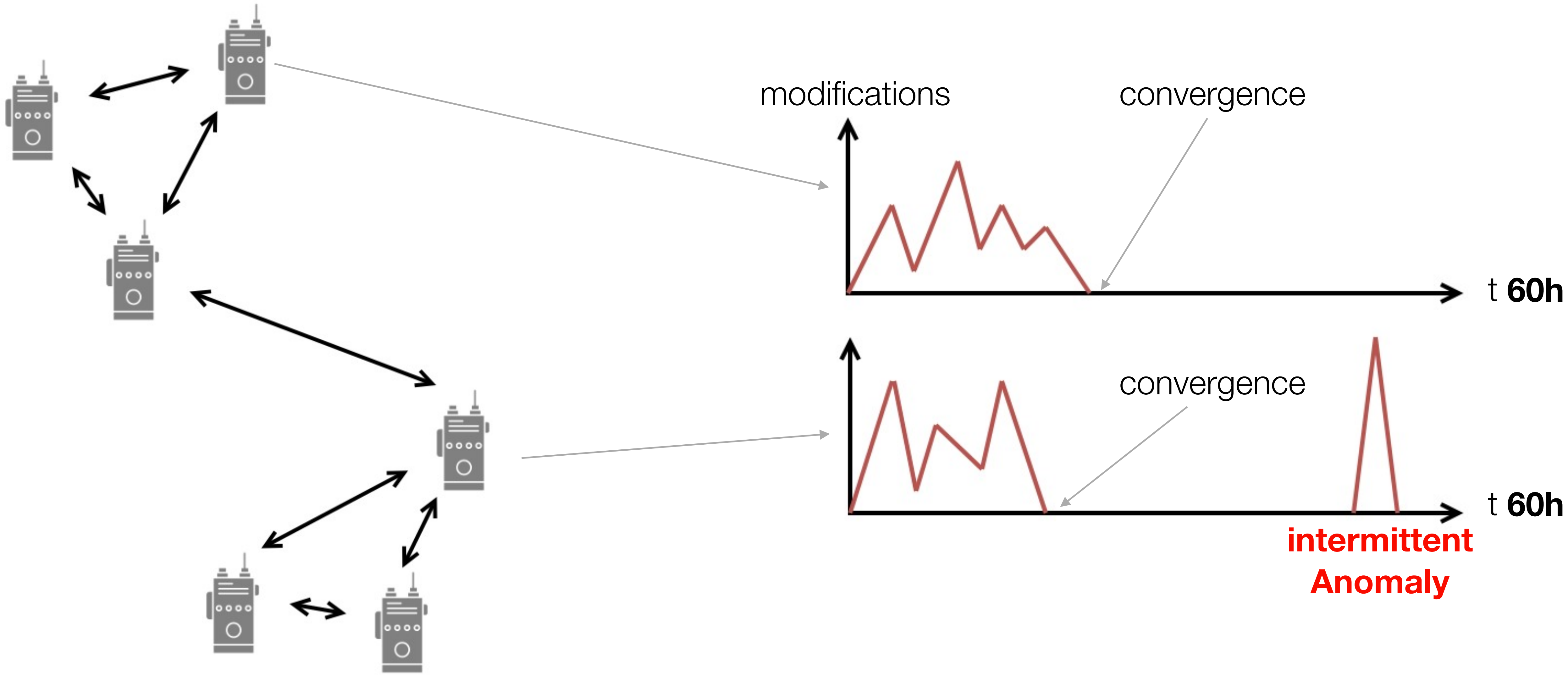
Errors due to parallelism and concurrency

> *« …debugging parallel applications is especially difficult and may require specialized tools and methods not yet available. »*
>
> — Perscheid et al. 2017

[Studying the advancement in debugging practice of professional software developers, Perscheid et al., 2017]

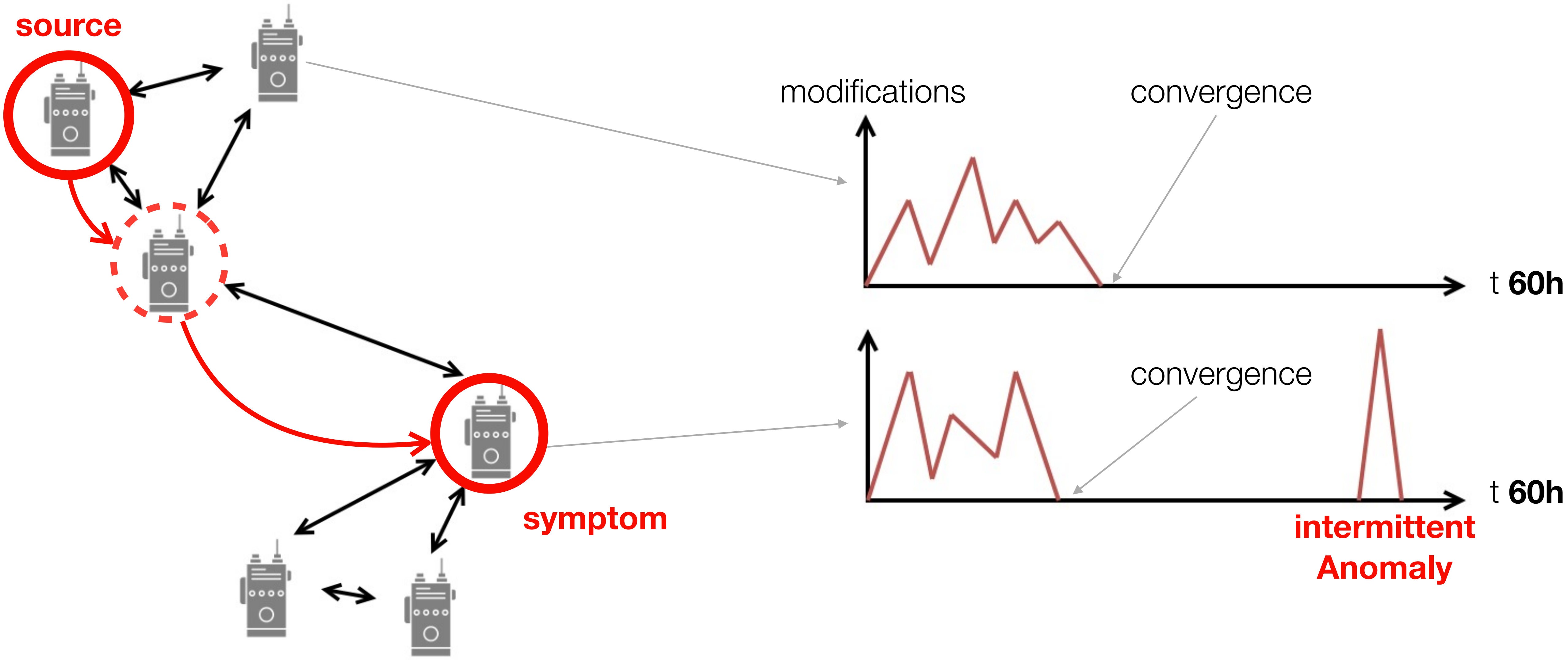
- **30% of hard bugs**
- Concurrent and parallel processes…
  - **Share state:** there can be conflicts while accessing state shared between different processes (race conditions)
  - **Interact with each other:** that interaction and its order can be non-deterministic

Errors due to parallelism and concurrency: example



Concurrent simulation of a software radio
Synchronisation between communications of radios

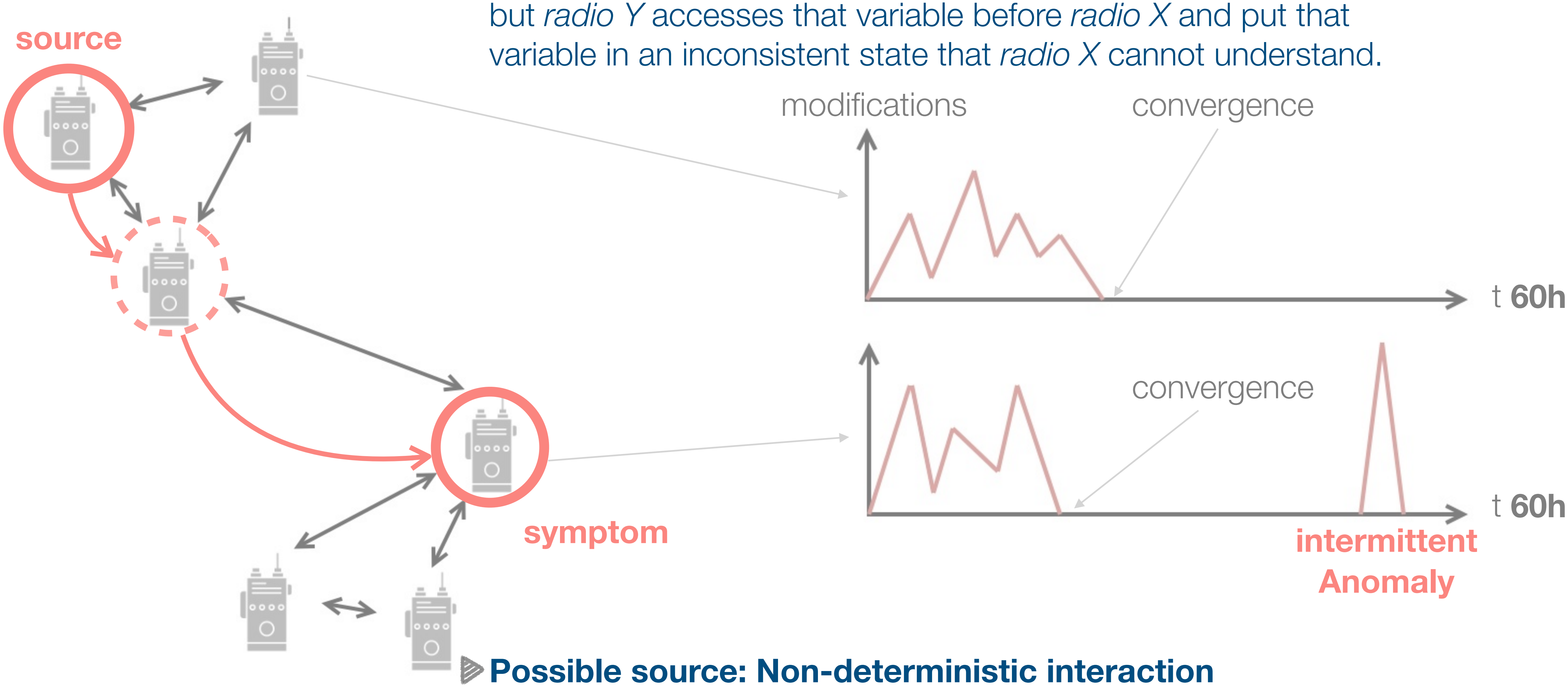
Errors due to parallelism and concurrency: example



Concurrent simulation of a software radio
Synchronisation between communications of radios

Errors due to parallelism and concurrency: example

Possible source: Race condition
Radio X must access a variable of the program before *radio Y*, but *radio Y* accesses that variable before *radio X* and put that variable in an inconsistent state that *radio X* cannot understand.

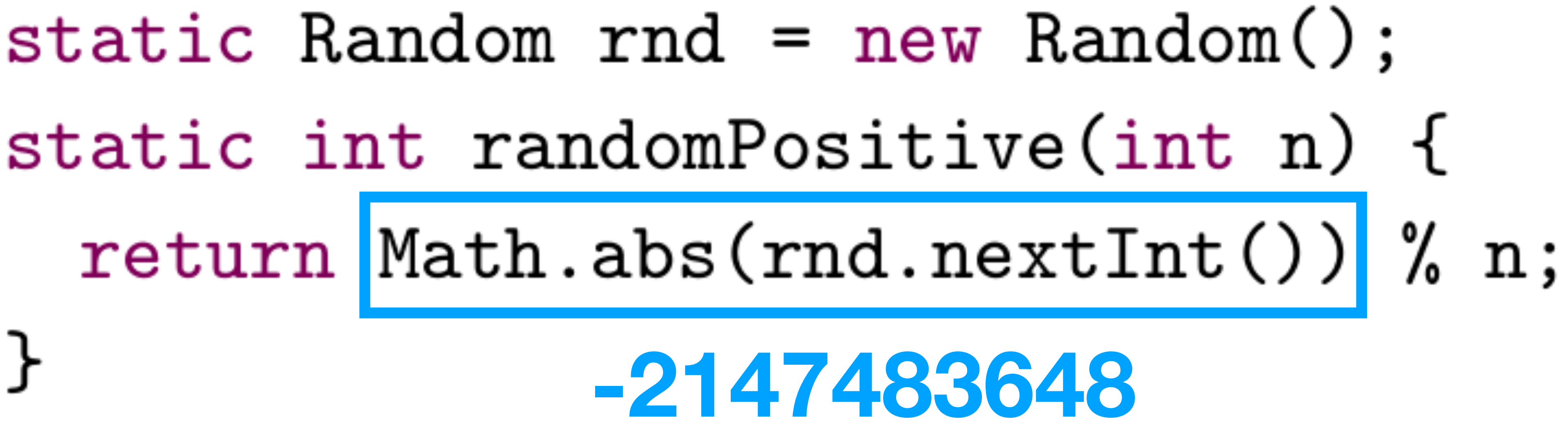


Possible source: Non-deterministic interaction
Under specific environment conditions, two radios interact while they should not. As those conditions are unpredictable, the problem only reproduces sporadically and are therefore hard to observe.

Non-deterministic errors

- These errors are due to unpredictable events, behavior or state of the system
  - They are hard to reproduce because we do not control the non-deterministic aspect of the problem
  - If we cannot reproduce errors, it is very difficult to understand them

Non-deterministic errors: an example



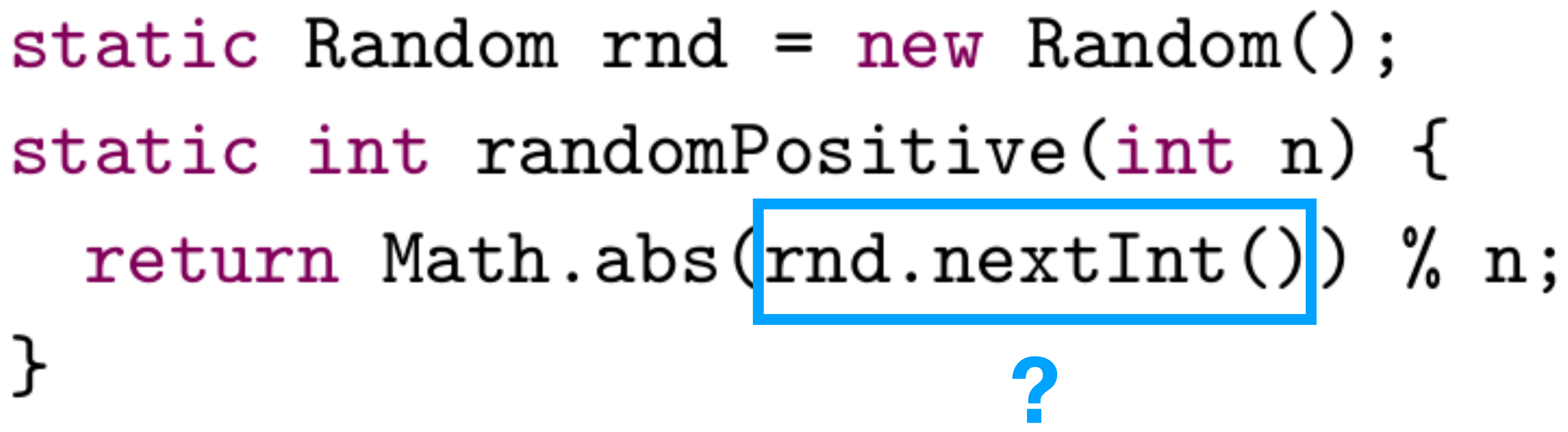
```
static Random rnd = new Random();
static int randomPositive(int n) {
  return Math.abs(rnd.nextInt()) % n;
}
```

-2147483648

An absolute function returns a negative number!

from: Redshell: Online Back-In-Time Debugging», Schulz and Bockish, 2017

Non-deterministic errors: an example



```
static Random rnd = new Random();
static int randomPositive(int n) {
  return Math.abs(rnd.nextInt()) % n;
}
```

We cannot reproduce it since it uses a random number as input

from: Redshell: Online Back-In-Time Debugging», Schulz and Bockish, 2017

The cost of debugging

At stake: maintenance and evolution of software

- **Tremendous cost for the software industry**
  - Up to 50% of the time spent on debugging and validation
  - Up to 75% of the development cost

At stake: maintenance and evolution of software

- **Tremendous cost for the software industry**
  - Up to 50% of the time spent on debugging and validation
  - Up to 75% of the development cost
- **Extremely difficult activity**
  - Some bugs are hard to understand, to solve and fix
  - Some bugs are fixed but never understood
  - Some bugs are understood but never fixed (because the cost is too high, or because it is impossible…)

At stake: maintenance and evolution of software

- **Tremendous cost for the software industry**
  - Up to 50% of the time spent on debugging and validation
  - Up to 75% of the development cost
- **Extremely difficult activity**
  - Some bugs are hard to understand, to solve and fix
  - Some bugs are fixed but never understood
  - Some bugs are understood but never fixed (because the cost is too high, or because it is impossible…)
- **Costs a lot: money, material, lives…**

Bibliography

References

1. **The new hacker’s dictionary**, E. S. Raymond and G. L. Steele, 1996

2. **How Debuggers Work**, Jonathan B. Rosenberg, 1996

3. **Debugging: The 9 indispensable rules for finding even the most elusive software and hardware problems**, David J. Agans, 2002

4. **Why Programs Fail**, Andreas Zeller, 2009

5. **Effective Debugging**, Diomidis Spinellis, 2018

6. **Redshell:** Online Back-In-Time Debugging», Schulz and Bockish, 2017

7. **My Hairiest Bug War Stories**, Mark Eisenstadt, 1997

8. **Studying the advancement in debugging practice of professional software developers**, Perscheid et. at. 2017

9. **Modern Debugging: The Art of Finding a Needle in a Haystack**, Diomidis Spinellis, 2018

10. **The Debugging Mindset Understanding the psychology of learning strategies leads to effective problem-solving skills**, Devon H. O’Dell, 2017

11. **Basic Concepts and Taxonomy of Dependable and Secure Computing**, Avizienis et al., 2004