Error and Exception Handling

Dirk Riehle, FAU Erlangen

ADAP B05

Agenda

1. The common bug
2. A system model
3. Error detection
4. Error signaling
5. Error handling
6. Service failure

Focus

In this lecture, we focus on a subset of [A+04], specifically

- Errors caused by software faults that are
  - Always development, internal, human-made faults
  - Typically non-malicious, non-deliberate
- Error detection by concurrent detection
- Error handling using any matching strategy

In other words, errors caused by the common bug

[A+04] Avižienis, A., Laprie, J. C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11-33.

1. The Common Bug

Catching the Common Bug

Best done during development (earlier is better)

But you can’t avoid errors at runtime

Deliberately Bad Java Example [1]

```
public int readInt(File f, Buffer b) throws ParseException {
    int result = 0;
    try {
        FileInputStream fis = new FileInputStream(f);
        fis.read(b);
        result = Integer.parse(b.toString());
    } catch (Exception ex) {
        // do nothing
    }
    if (result == 0) {
        // there should never be “0” in file
        System.out.println("something went wrong!");
        return -1;
    }
    return result;
 }
```

[1] Doesn’t compile; created for demonstration purposes only

Things Wrong With Example

General programming errors

- Unclear preconditions; no assertions
- No need for external buffer variable
- No clean-up after resource use

Specific bad practices of error handling

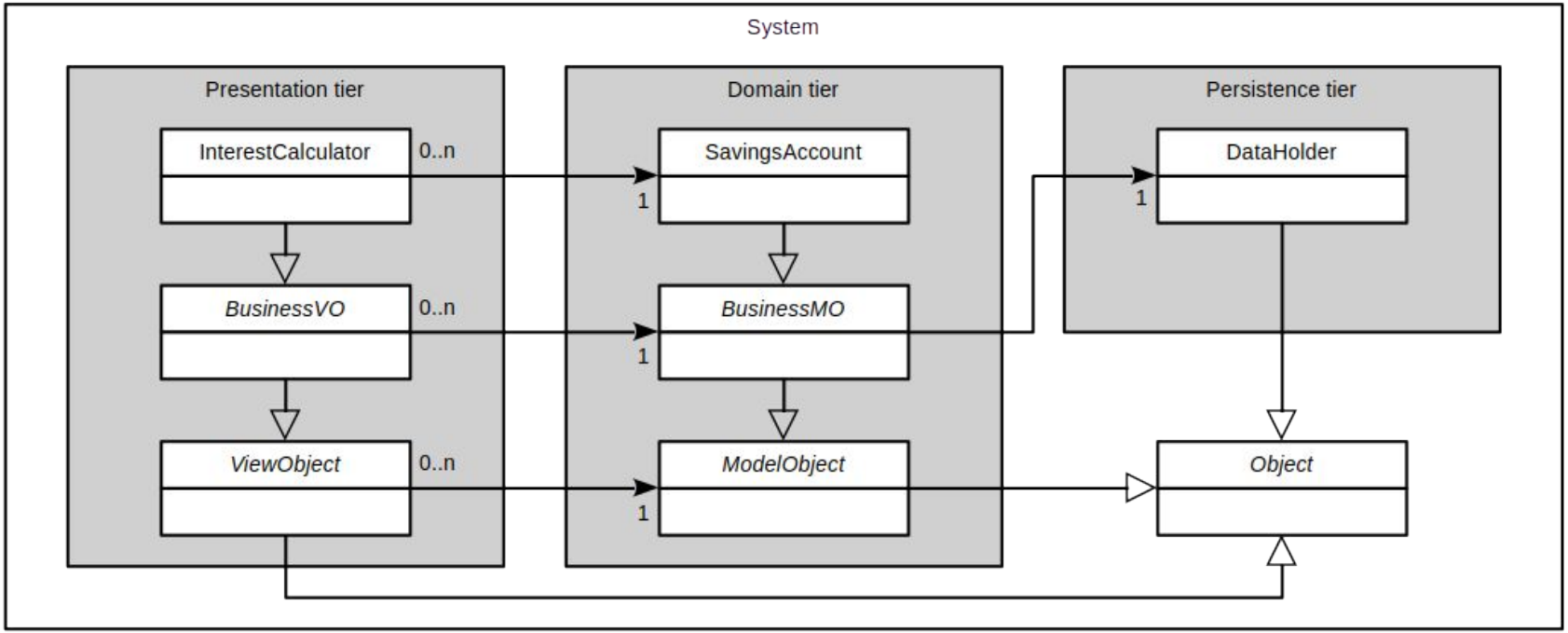
- Overloading of purpose of return value
- Mismatch between method signature and behavior
- System exception swallowed without logging
- Inconsistent use of error codes and exceptions
- Unprofessional logging / error message useless

Dependability and Fault Tolerance

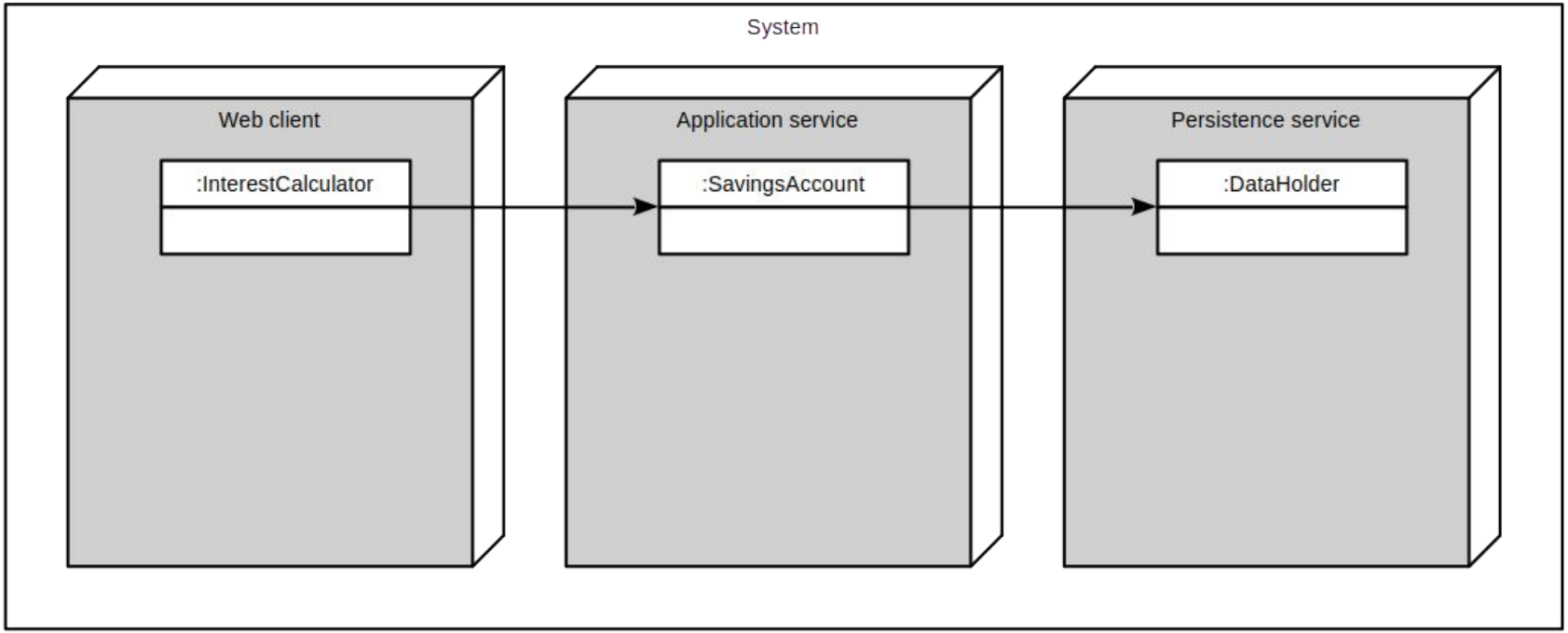
If bugs are inevitable, how to handle them?

2. A System Model

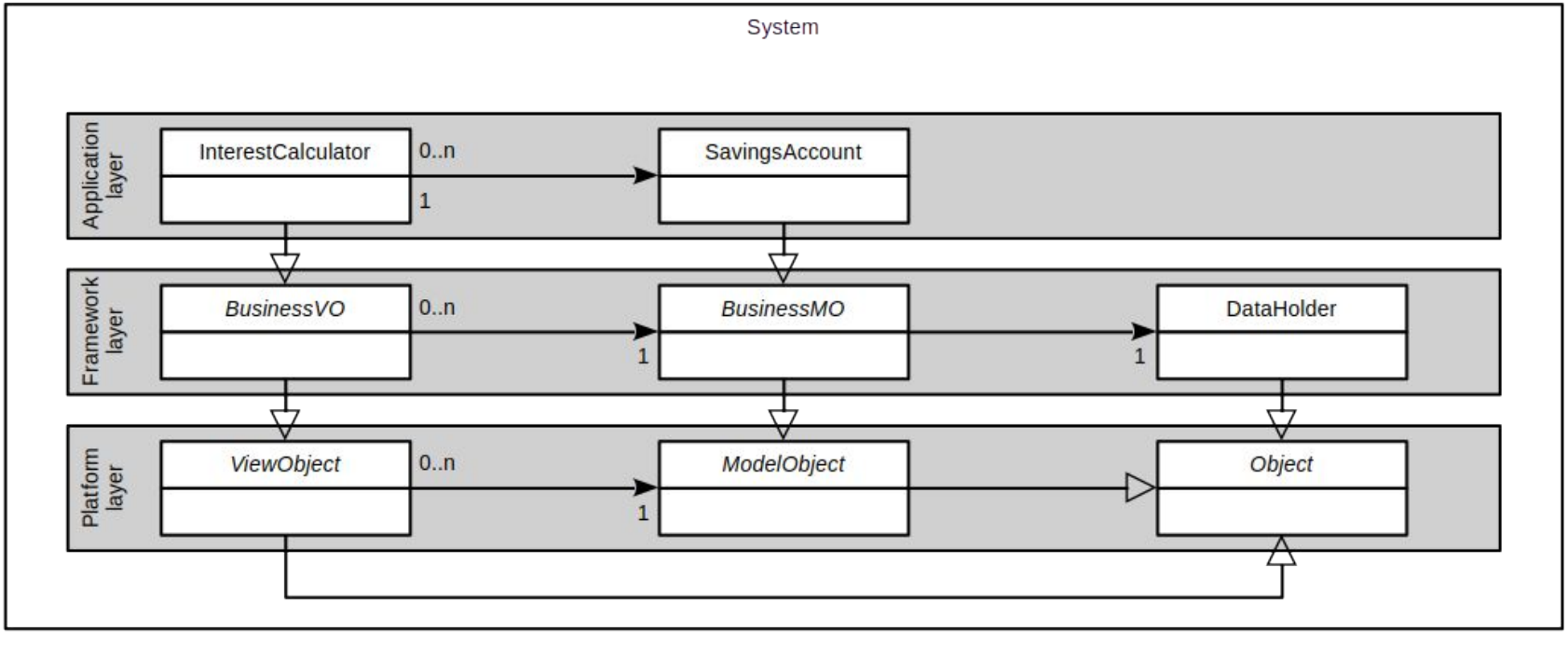
System Architecture (Runtime Tiers)



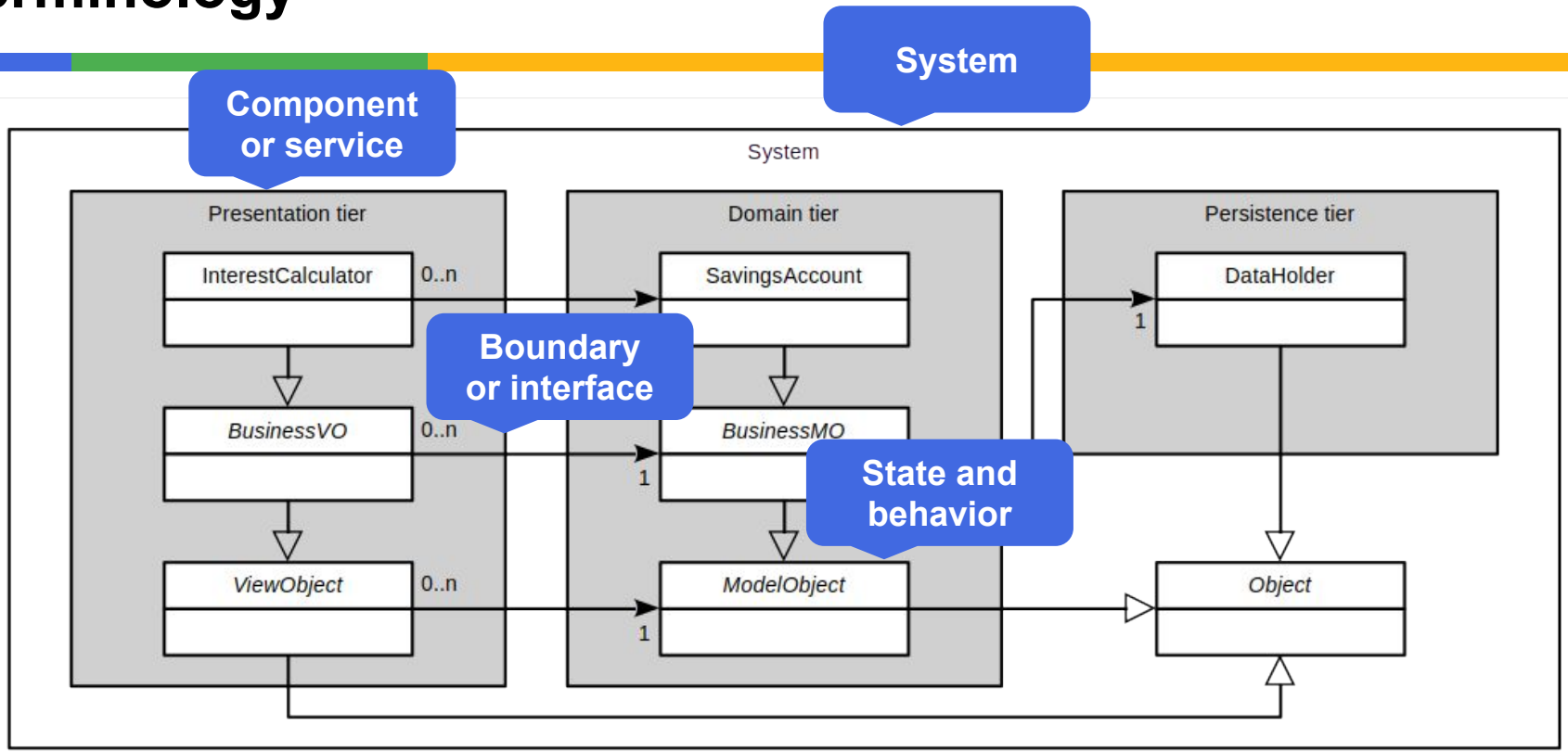
Runtime Objects



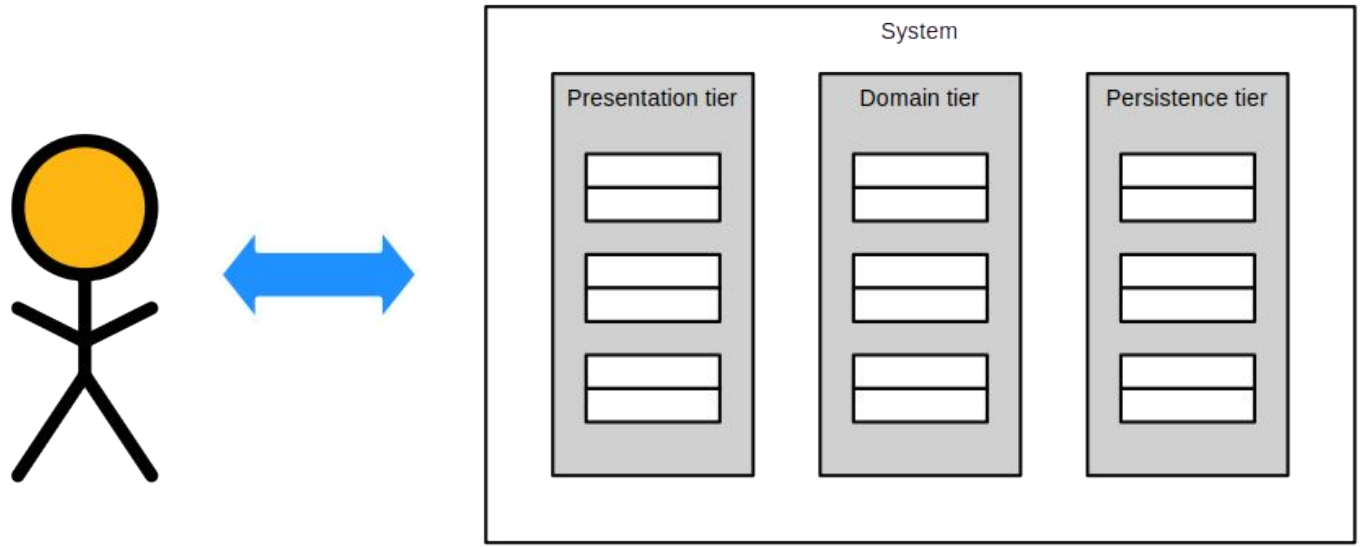
(Static) Code Layers



Terminology



Operator (User) Interaction [1]



[1] Not using the term “user” because it is too broad

System Terminology

- A runtime component is
  - Some runtime data encapsulated (bounded) by an interface
  - Typically does not stand alone and cannot be deployed independently
- A service is a runtime component that
  - Has business value and maybe composed of other components
  - Typically does not stand alone (has clients), but can be deployed independently
- A system
  - Is a combination of services and operators (yeah, really)
  - Has business value and has stakeholders to who this value accrues
- Operators are stakeholders but not all stakeholders are operators

Fault

A fault is

- A condition that can cause an error
  - A fault is **dormant,** if it has not yet caused an error
  - A fault is **active,** if it causes an error

A fault

- Can be classified by eight independent dimensions [A+04]

Fault Classification

	Option 1	Option 2
Phase of creation / occurrence	Developmental	Operational
System boundaries	Internal	External
Phenomenological cause	Natural	Human-made
Dimension	Hardware	Software
Objective	Malicious	Non-malicious
Intent	Deliberate	Non-deliberate
Capability	Accidental	Incompetence
Persistence	Permanent	Transient

Error

An error is

- A state of the system that may lead to a failure
  - An error is **latent,** if it has not been detected
  - An error has been **detected,** if regular code identifies an erroneous state

An error

- Can be categorized by the failures it may cause

Failure

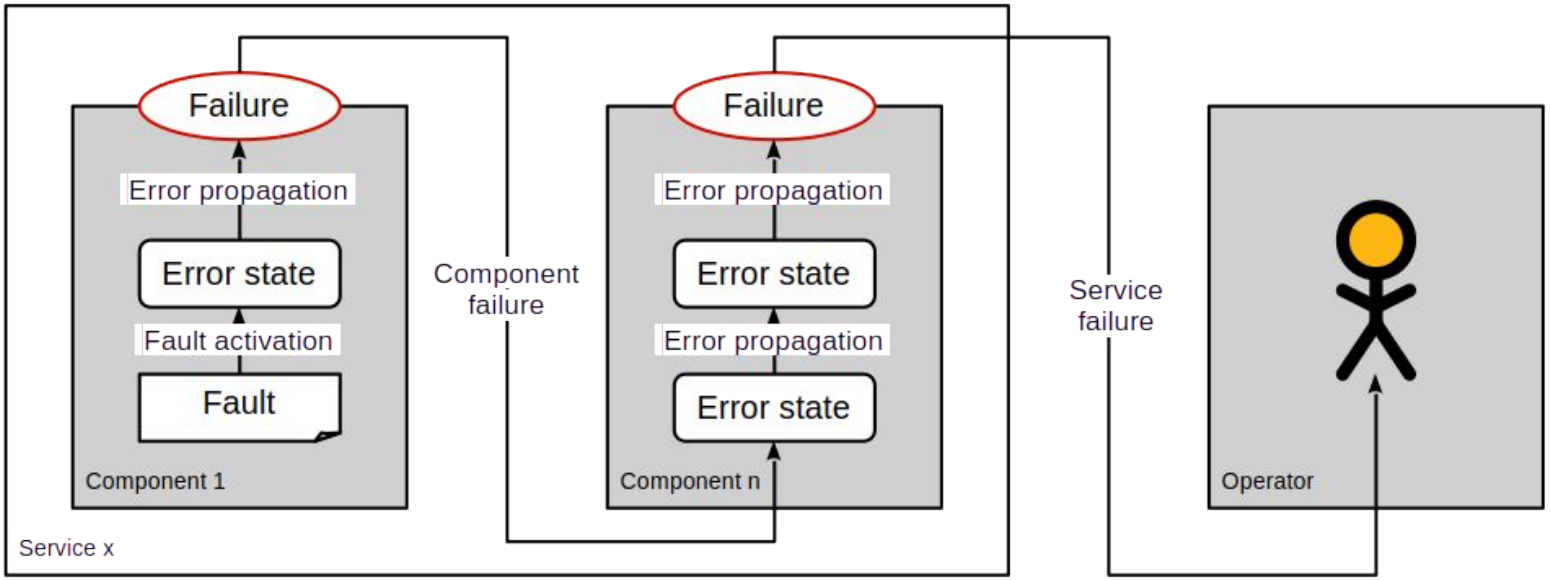
A failure is

- An event that transitions the system from correct to incorrect service

A failure

- Has a (failure) mode
- Can be categorized by four independent dimensions
  - Domain (content, early timing, late timing, halt, and erratic failures)
  - Detectability (signaled and unsignaled failures)
  - Consistency (consistent and inconsistent failures)
  - Consequences (minor to catastrophic failures)
- Can be ranked by severity (consequences)

Illustration of Process that Leads to Service Failure



Three Steps to Error Handling

1. Error detection
2. Error signaling
3. Error handling

3. Error Detection

How to Detect an Error (State)

If an error is an erroneous state of the system

- You need to explicitly check for that state

Assertions that fail identify an error (state)

- Assertions are regular code in normal processing state

Use design by contract

Example Name Instance (= Component) Failure

```
public insert(i: number, c: string): void {
    this.assertIsValidIndex(i);
    this.assertIsProperlyMasked(c);

    const oldNoComponents: number = this.getNoComponents();
    this.doInsert(i, c);

    this.assertClassInvariants();

    const condition: boolean = this.getNoComponents() == (oldNoComponents + 1);
    MethodFailedException.assertCondition(condition);
}
```

Typical Information to Capture / Gather

Meta information

- Error id (instance)
- Error type

Error-specific information

- Stacktrace
- Source objects
- Affected objects
- Explanatory message

How to Represent Error Information

- Error codes
- Error objects
- Exceptions

4. Error Signaling

How to Signal an Error?

A detected error state needs to be logged and signaled

Transition the system from normal to abnormal program / processing state

Normal vs. Abnormal Processing State

Normal processing state (NPS)

- Method performed its services
- Control flow returns via return statement

Abnormal processing state (APS)

- Method failed to perform service
- Control flow returns via raising an exception

Methods for Error Signaling

Using normal control flow (but in abnormal processing state)

- Use return statement but include error code in return value

Using abnormal control flow

- Raise an exception and include information in exception object

Error Code Conventions

0 = no error

-1 = generic error

1..n = specific error

Return Objects

A return object is

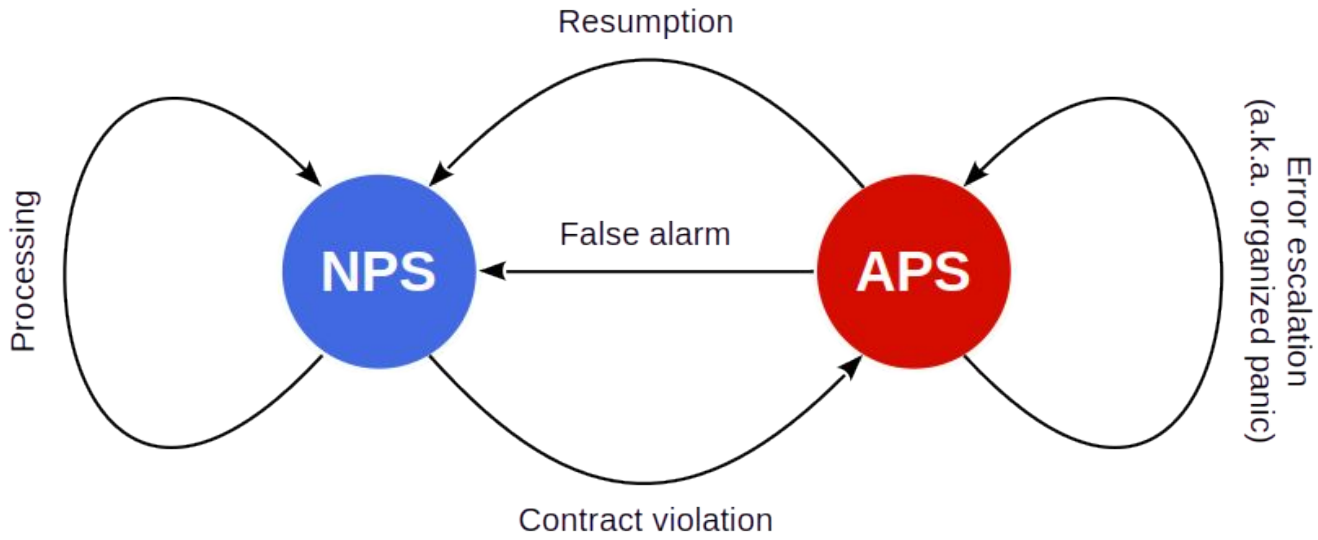
- An object that provides both
  - A possible error code, if any
  - The actual return results

Error Codes vs. Return Objects vs. Exceptions

	Pros	Cons
Error codes	Simple	Return values need to be passed back separately
Return objects	A solution if you have no good exception mechanism	Mix normal with abnormal processing state
Exceptions	Separate normal with abnormal processing state	–

5. Error Handling

Error Handling State Model



Error Handling

A false alarm is

- An erroneously identified error state

Resumption is

- When you try again, possibly after fixing something

Escalation (a.k.a. organized panic) is

- When you give up and signal the error to your caller

How to Handle Component Failures?

```
public read(noBytes: number): Int8Array {
    let result: Int8Array = new Int8Array(noBytes);
    // do something

    let tries: number = 0;
    for (let i: number = 0; i < noBytes; i++) {
        try {
            result[i] = this.readNextByte();
        } catch(exception) {
            tries++;
            // Oh no! What @todo?!
        }
    }

    return result;
}
```

1

```
if (tries == 3) {
    // ignore
}
```

2

```
if (tries == 3) {
    throw exception;
}
```

3

```
if (tries == 3) {
    result = {
        errorCode: -1,
        errorMessage: "couldn't finish"
    }
}
```

4

```
MethodFailedException.assertCondition(tries < 3);
```

How to Escalate an Error

First, clean up

- Leave the component in a viable state
  - Restore component invariants
- Release any unneeded resources
  - Use finally block to ensure this

Then, escalate

- Enhance original error information
  - Keep chaining exceptions

Checked vs. Unchecked Exceptions

Use unchecked exceptions within a component

- You also can use checked exceptions, but human psychology suggests that you won’t be able to follow through

Use checked exceptions at a service’s boundary

- Typescript does not have checked exceptions, so you may have to find alternative ways of dealing with service boundaries

6. Service Failure

Service Failure

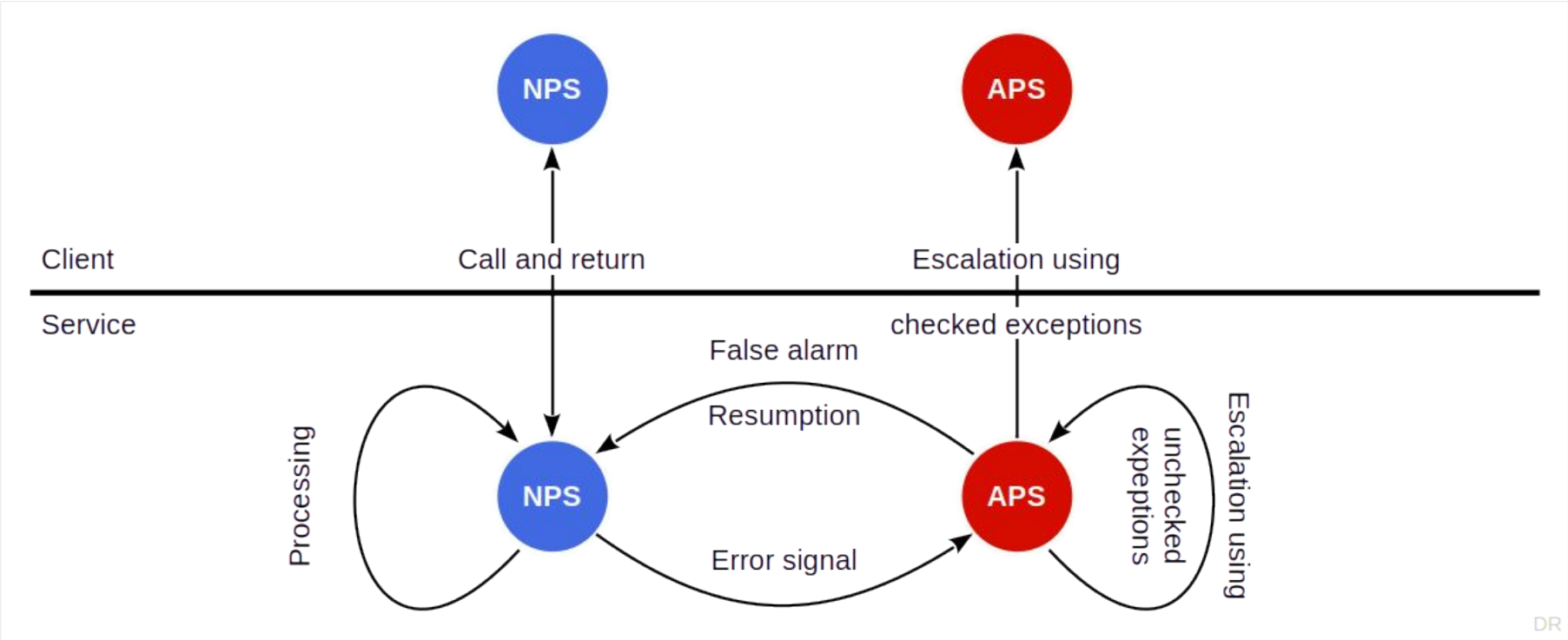
A service failure is a component failure

- At the interface to the calling client

Use checked exceptions to signal service failure

- Only use exceptions specific to the failure

Client / Service Error Handling State Model



Service Failure Exceptions to Use [1]

On the service side, for service failures, use

- ServiceFailureException

To be interpreted as a “to check” (checked) exception

[1] JavaScript does not support checked exceptions, hence we have to simulate it

Error Escalation of Failure Signal

On the client side, if you need to escalate a service failure

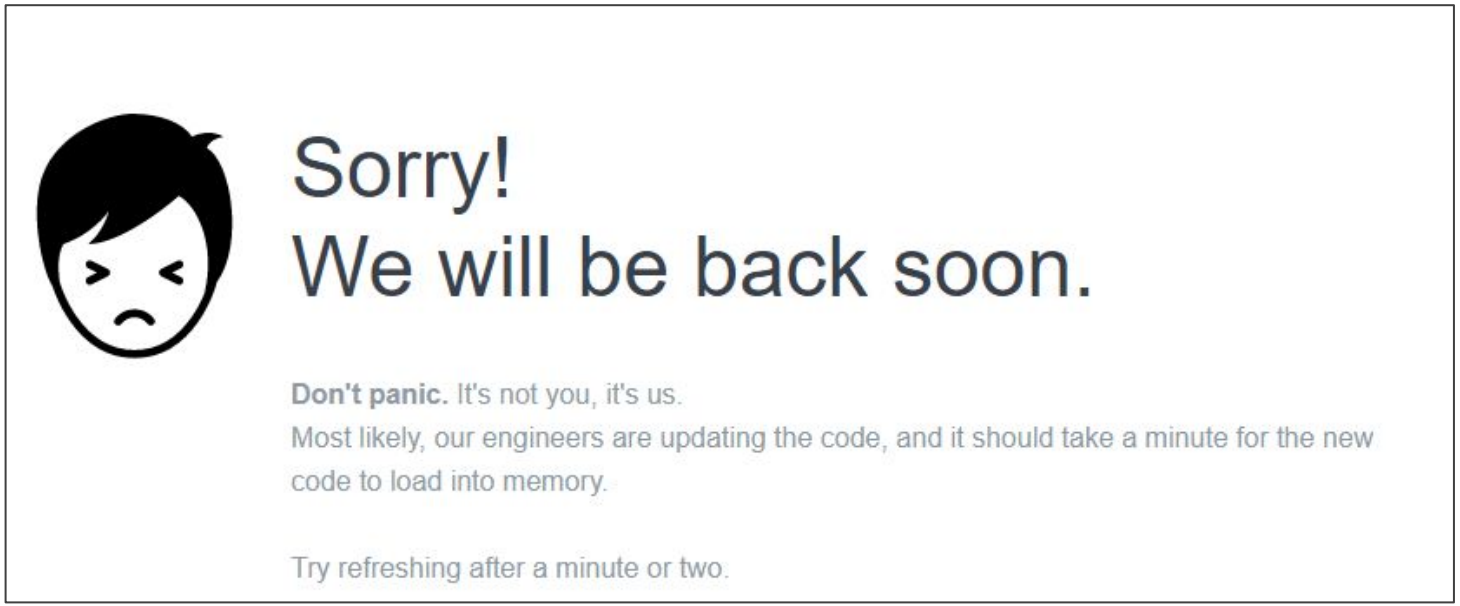
- Wrap the checked exception in an unchecked exception

The game may repeat itself at the next service boundary

Service Failure at the User (Operator) Interface

Rather than display in text, a checked exception

- Convert the error into human-readable form and display it



The Final Word on Exceptions

```
let up: Exception = new Exception("something is wrong");
throw up; // haha
```

Summary

1. The common bug
2. A system model
3. Error detection
4. Error signaling
5. Error handling
6. Service failure

Thank you! Any questions?

dirk.riehle@fau.de – https://oss.cs.fau.de

dirk@riehle.org – https://dirkriehle.com – @dirkriehle

Legal Notices

License

- Licensed under the CC BY 4.0 International license

Copyright

- © 2012, 2018, 2024 Dirk Riehle, some rights reserved