

Error Handling

Prof. Dr. Dirk Riehle

Friedrich-Alexander University Erlangen-Nürnberg

ADAP C06

Licensed under [CC BY 4.0 International](https://creativecommons.org/licenses/by/4.0/)

Focus of Lecture

- 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

Catching the Common Bug

- Best done during development (due to cost)
- Still, you can't avoid errors during runtime

Example of Poor Error Handling Code [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

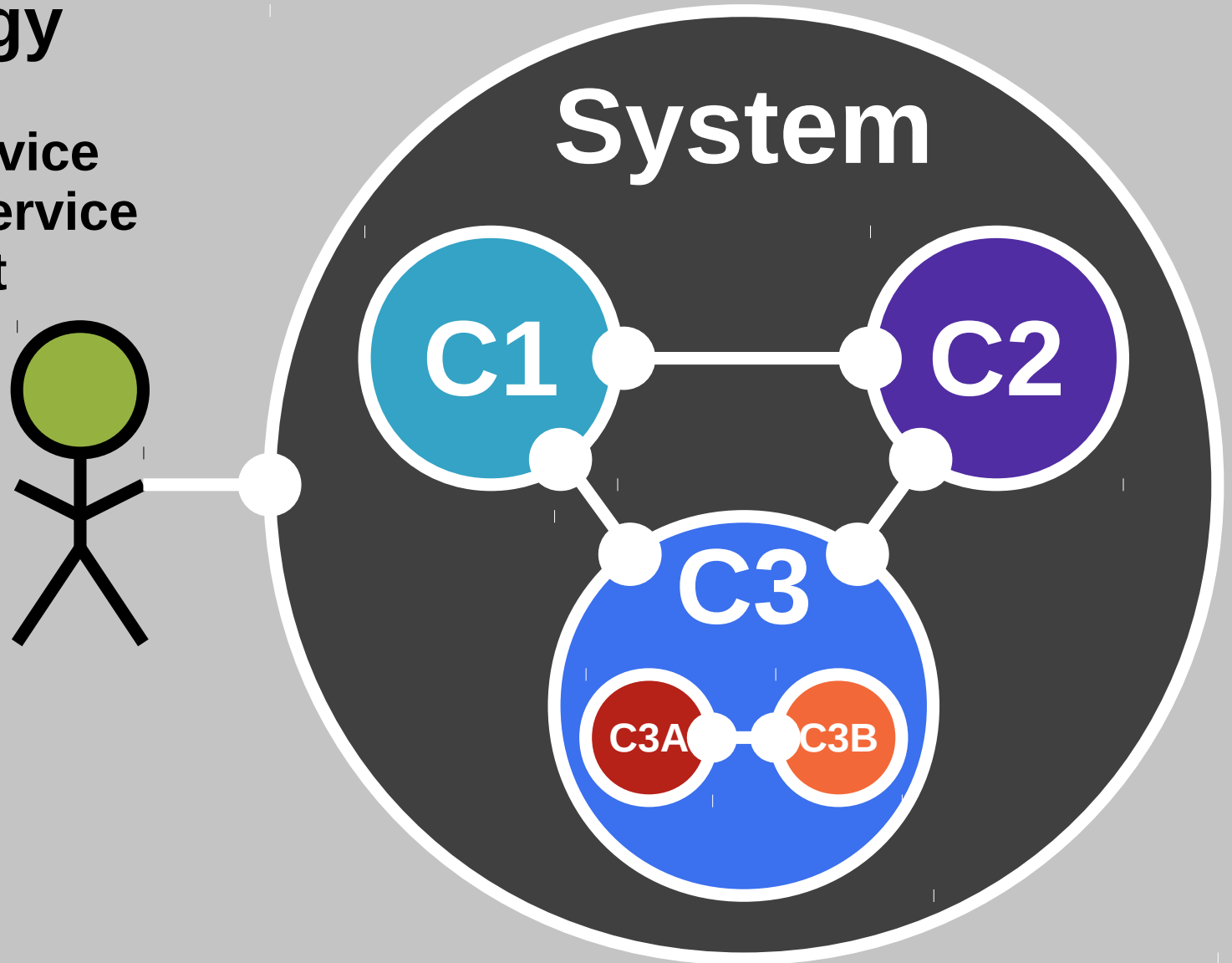
- General programming errors
 - Unclear preconditions; no assertions
 - No need for external buffer variable
 - No clean-up after resource use
- Specific bad practices of error handling
 - Overloading of purpose of return value
 - Mismatch between method signature and behavior
 - System exception swallowed without logging
 - Inconsistent use of error codes and exceptions
 - Unprofessional logging / error message useless

**If bugs are inevitable,
how to handle them?**

System Model

Terminology

- System
- Correct service
- Incorrect service
- Component
- Boundary
- Interface
- Structure
- Behavior
- State
- User



Fault

- A fault
 - Is a condition that can cause an error
 - A fault is active, if it causes an error
 - A fault is dormant, if it has not yet caused an error
 - Can be classified by eight independent dimensions
- A software fault (the “common bug”)
 - Is always a development, internal, human-made fault
 - Is typically non-malicious, non-deliberate

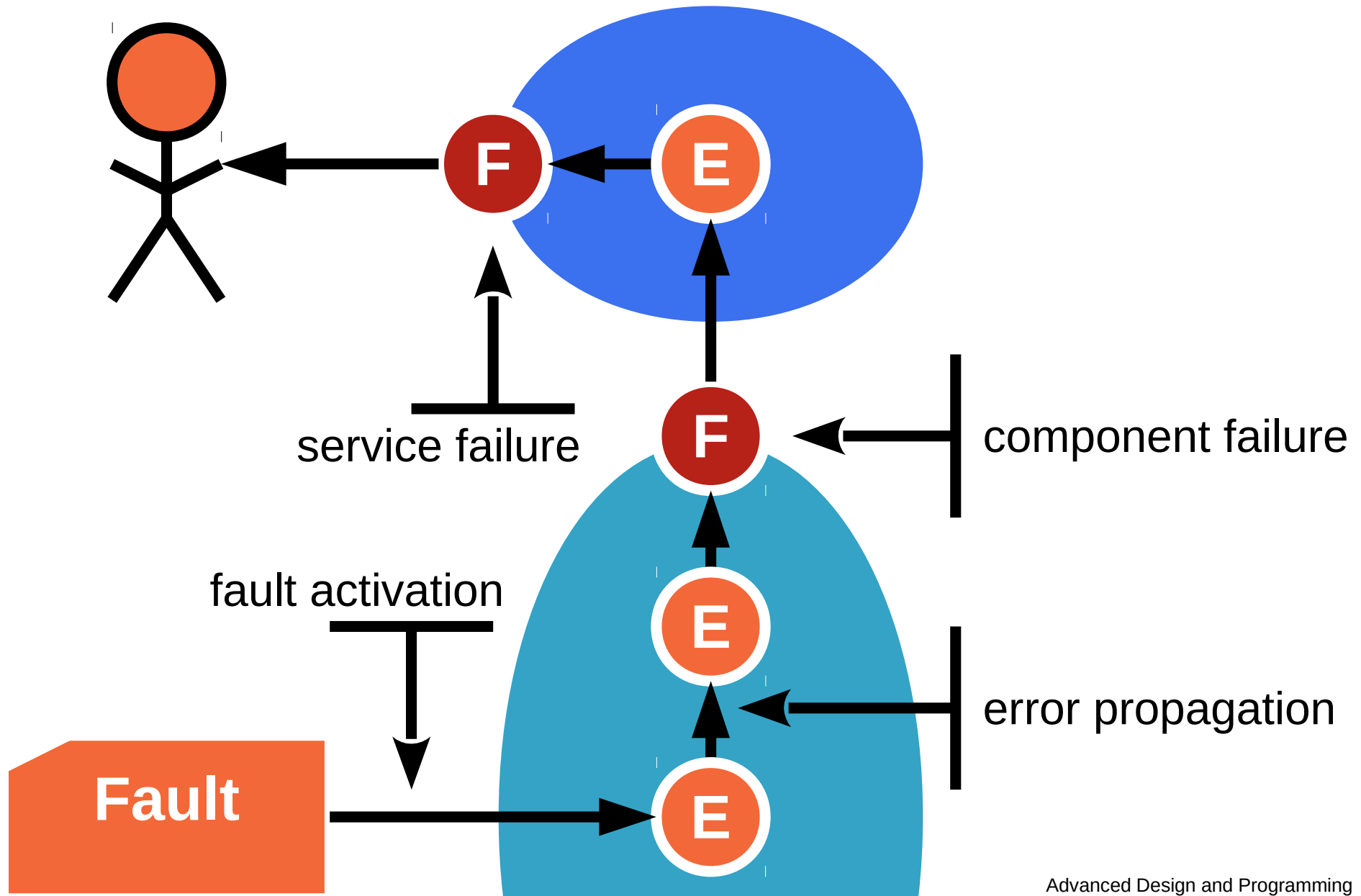
Error

- An error
 - Is a state of the system that may lead to a failure
 - Has been detected, if it has been indicated by error message or signal
 - An error is **latent**, if it has not been detected
 - Can be categorized by the failures it may cause

Failure

- A failure
 - Is an event that transitions the system from correct to incorrect service
 - 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)

Process Leading to Service Failure



- 1. Detection**
- 2. Signaling**
- 3. Handling**

Error Detection

- If an error is an incorrect state of a component, how to detect it?
 - You can't (in normal programming)
- But you can detect a failure to deliver the promised service
 - How? Using design by contract!
- Error detection = recognizing inability to provide service
 - Failing preconditions, class invariants, or post conditions
- Don't think in error states, think in failure to provide service
 - Remember: No need for defensive programming

Examples of Error Detection

```
public void insert(int i, String c) {  
    assertClassInvariants();  
  
    // assert preconditions  
    assertIsValidIndex(i, getNoComponents() + 1);  
    assertIsNonNullArgument(c);  
  
    // prepare assertion of postconditions  
    int oldNoComponents = getNoComponents();  
  
    doInsert(i, c);  
  
    // assert postconditions  
    assert (oldNoComponents + 1) == getNoComponents() : "...";  
  
    assertClassInvariants();  
}
```

Error Capture / Representation

- Information to be captured
 - Error ID
 - Error type
 - Source objects
 - Affected objects
 - Explanatory message
- Representation of information
 - Error codes
 - Error objects
 - Exception objects

Examples of Error Representation

```
protected void assertIsValidIndex(int i) throws IndexOutOfBounds... {  
    if ((i < 0) || (i >= getNoComponents())) {  
        throw new IndexOutOfBoundsException("invalid index = " + i);  
    }  
}
```

```
public class RegExpParseException extends ParseException {  
    protected String regExp = "";  
  
    public RegExpParseException(String msg, String exp, int offset) {  
        super(msg, offset);  
        regExp = exp;  
    }  
  
    public String getRegExp() {  
        return regExp;  
    }  
    ...  
}
```


Error Logging

- Possibly log the error information using system logger
 - May be helpful in case (poor) using code drops the error
 - Be slow to make assumptions about context
- Using the (system) logger
 - Write error object to appropriate logging level
 - Further functionality depends on the logger

Error Signaling

- A detected error needs to be (logged and) signaled
- Transitions the system from normal to abnormal program state

Normal vs. Abnormal Program State

- Normal program state (NPS)
 - Method performs its duties
 - Control flow returns to caller via return statement
- Abnormal program state (APS)
 - Method failed to provide service
 - Control flow returns to caller
 - via return error code
 - via thrown exception

Methods for Error Signaling

- Using normal control flow (via return)
 - Error information can be passed using
 - Return value
 - Method argument
 - Mailbox object
- Using abnormal control flow (via raising an exception)
 - Error information is passed using
 - Exception object as part of raised exception

Exercise for Error Detection and Signaling

- How to implement a basic buffer read method?

Solution Using Error Codes

```
public class File {  
    public static final int ERROR_END_OF_FILE = 1;  
    public static final int ERROR_PARITY = 2;  
  
    public int readBytes(Buffer buf, int no) {  
        while (no-- >= 0) {  
            int err = readByte(buf);  
            if (err != 0) return err;  
        }  
        return 0;  
    }  
  
    protected int readByte(Buffer buf) {  
        if (handle.isEOF()) return ERROR_END_OF_FILE;  
        byte next = handle.getNextByte();  
        boolean parity = handle.getParity();  
        if (parity != calcParity(next) return ERROR_PARITY;  
        buf.add(next);  
        return 0;  
    }  
  
    ...  
}
```

Solution Using Exceptions

```
public class File {  
    public byte[] readBytes(int no) throws IOException {  
        byte[] buffer = new byte[no];  
        for(int i = 0; i < no; i++) {  
            byte next = readByte();  
            buffer[i] = next;  
        }  
        return buffer;  
    }  
  
    protected byte readByte() throws IOException {  
        if (handle.isEOF()) throw new EOFException(...);  
        byte next = handle.getNextByte();  
        boolean parity = handle.getParity();  
        if (parity != calcParity(next)) throw new IOException(...);  
        return next;  
    }  
  
    ...  
}
```

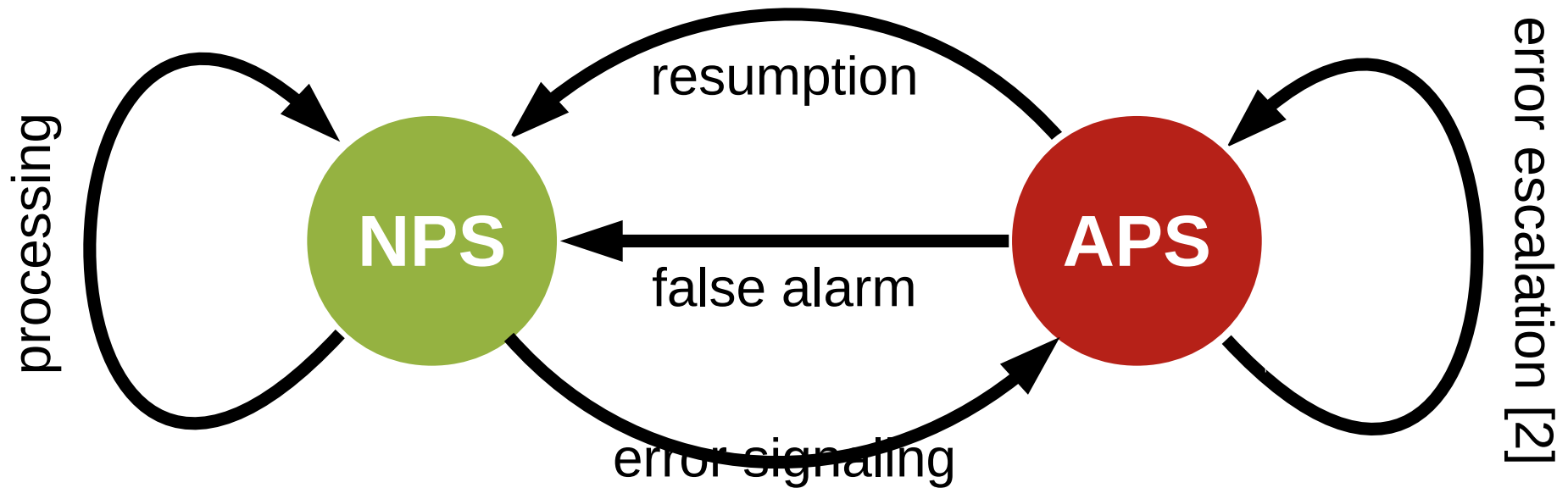
Error Code Conventions

- 0 typically indicates “no error”
- -1 typically indicates a generic error
- 1..onwards indicate specific errors

Error Codes vs. Exceptions

- Error codes are a poor error signaling mechanism
 - Mix normal with abnormal program state code
 - Separate error signal code from error information object
 - **In Java, avoid using error codes if possible**
- Exceptions were designed for error signaling
 - They separate normal from abnormal program state
 - Specifically support passing error information in exception object
 - Corollary: Don't use exceptions to make a regular return

Error Handling State Model [M92] [1]



- [1] Adjusted terminology to [A+04]
- [2] Called “organized panic” in [M92]

- 1. False alarm**
- 2. Resumption**
- 3. Escalation**

Exercise of Error Handling

- How to handle error signaled by File component?

Solution Using Exception Handling

```
public class Document {  
    protected byte[] buffer;  
    public void loadFromFile(File file) throws DocumentException {  
        ...  
        int no = file.getLength();  
        buffer = new byte[no];  
        int tries = 0;  
        for(int i = 0; (i < no) && (tries < 3); i++) {  
            try {  
                buffer[i] = file.readByte();  
            } catch(EOFException eofex) {  
                throw new DocumentException(..., eofex);  
            } catch(IOException iex) {  
                tries++;  
            }  
        }  
        if (tries == 3) {  
            throw new DocumentException(...);  
        }  
    }  
    ...  
}
```

Error Escalation (“Organized Panic”)

- Error escalation
 - Is the process of cleaning-up and delegating error handling to caller
 - Basically, your code has exhausted its options and gives up

Steps in Error Escalation

- Clean-up
 - Always leave the current component in a viable state
 - Make sure you restore class and component invariants
 - Restore and/or release relevant resources
 - Use finally block in exception handling to ensure this
- Escalation
 - Enhance original error information with new insights
 - Do not hide your attempts to handle the error
 - Typically, chain exceptions
 - Attach prior error information (exception) to new one

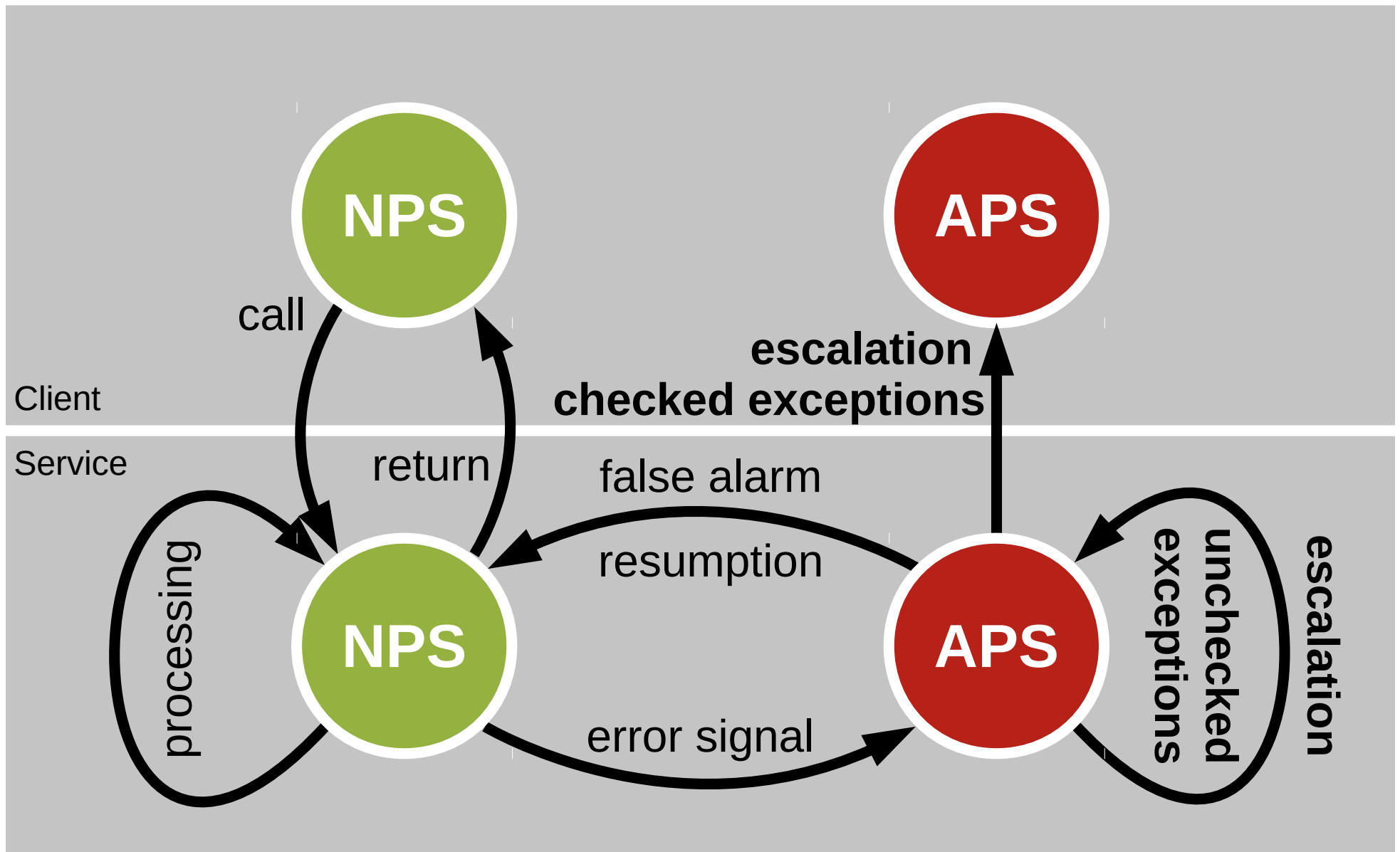
Checked vs. Unchecked Exceptions [DR]

- Checked exceptions
 - Are exceptions that must be declared in a method signature
 - Are intended to force user to take notice of the exception
 - Works well if error handling code is close to where exception was raised
 - Are a pain to handle if code is far removed from origin of exception
 - **Use checked exceptions (or error codes) in component interface**
- Unchecked exceptions
 - Are exceptions that don't need to be declared
 - Are intended to pass through client code by default
 - May make you miss an error signal that you should have handled
 - Are the only way to not completely clutter your component code
 - **Use unchecked exceptions only within your component**

Component Failure

- Error signals
 - Are part of the component interface
 - Should be specific to the component
 - **Use only checked exceptions in interface**
- **Do not let an unchecked exception escape**

Failure Handling at Component Boundary

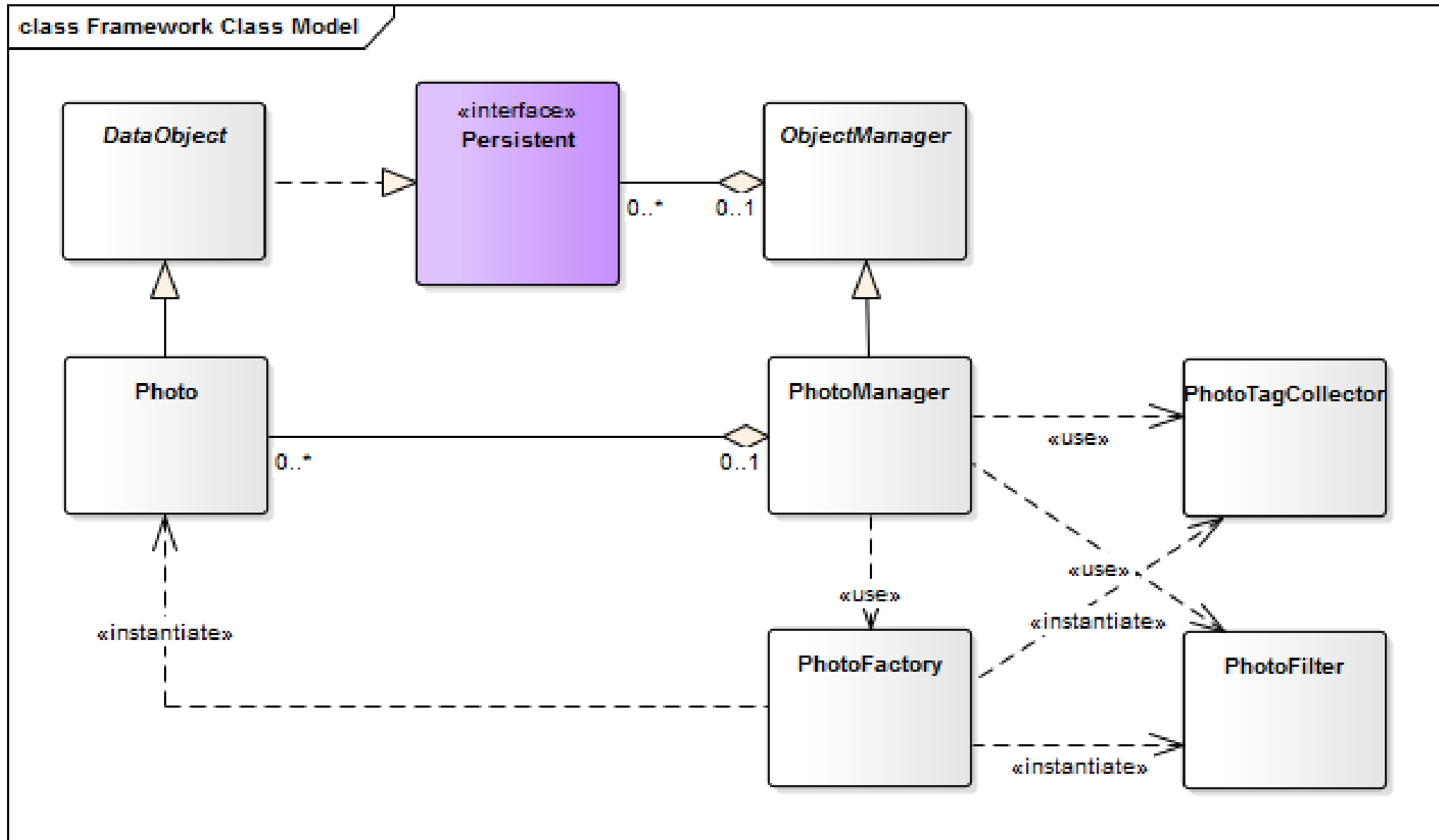


Methods for Failure Signaling

- Failure clean-up
 - Like clean-up for error signaling
 - Restore invariants
 - Release resources
- Failure escalation
 - Like error escalation but
 - Catch all unchecked exceptions
 - Escalate using checked exception
 - Provide exception chain with new one

Exercise for Component Failure

- How to handle an ObjectManager failure (to load an object)?



Solution to Component Failure Exercise

- Within component
 - Error detection
 - Catch error signal from storage layer (file or database exceptions)
 - Handle error to the extent possible; eventually, give up
 - Error signaling
 - Capture prior error signal; create new unchecked exception
 - Throw exception about inability to load object
 - Error escalation
 - If method can handle exception, do so
 - If not, let the exception pass through
- At component boundary
 - Capture internal error signal, wrap it in component-specific exception
 - Throw checked exception about component failure to environment

Service Failure (User Interface)

- A service failure
 - Is a component failure with the user as the client
 - User interface is the final system boundary
- Handling a service failure
 - Log the service failure (error)
 - Don't throw a checked exception
 - Convert the error into human-readable form and display it

Handling Faulty Components

- Well-behaved (but faulty) components
 - Follow error handling strategy as discussed
- Component of unclear quality
 - Wrap component in defensive code
 - Follow error handling strategy as discussed

Final Example of Raising an Exception [S11]

```
Exception up = new Exception("Something is wrong.");  
throw up; // ha ha
```


Review / Summary of Session

- Error handling
 - System structure: Class, component, service
 - Terminology: Fault, error, exception
- Error handling processes
 - Detection, signaling, handling
 - Life-cycle model of processing state

Thanks! Questions?

dirk.riehle@fau.de – <http://osr.cs.fau.de>

dirk@riehle.org – <http://dirkriehle.com> – [@dirkriehle](#)

Credits and License

- Original version
 - © 2012-2018 [Dirk Riehle](#), some rights reserved
 - Licensed under [Creative Commons Attribution 4.0 International License](#)
- Contributions
 - ...

Error Handling

Prof. Dr. Dirk Riehle
Friedrich-Alexander University Erlangen-Nürnberg

ADAP C06

Licensed under [CC BY 4.0 International](#)

It is Friedrich-Alexander University Erlangen-Nürnberg – FAU, in short.
Corporate identity wants us to say “Friedrich-Alexander University”.

Focus of Lecture

- 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

Catching the Common Bug

- Best done during development (due to cost)
- Still, you can't avoid errors during runtime

Example of Poor Error Handling Code [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

Advanced Design and Programming
© 2018 Dirk Riehle - Some Rights Reserved

Things Wrong with Example

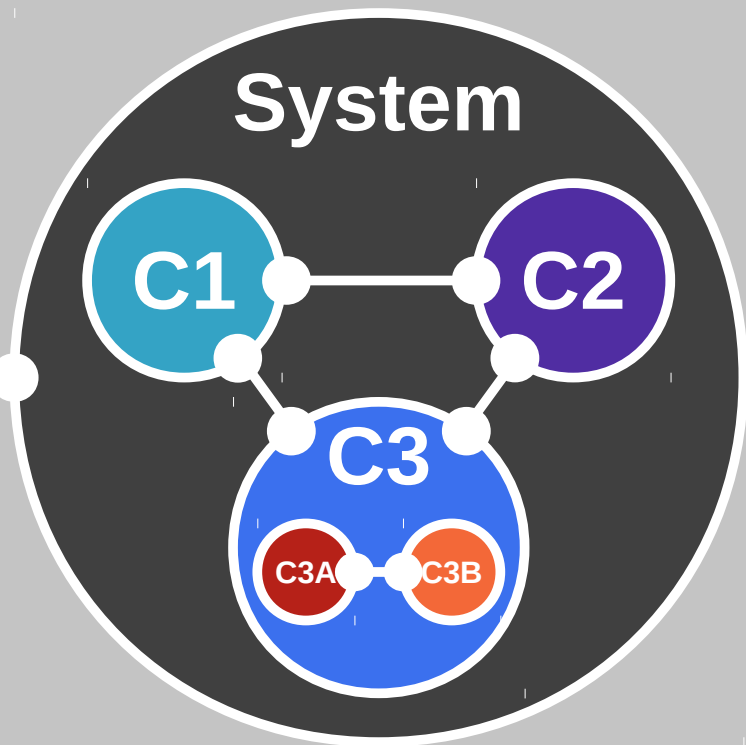
- General programming errors
 - Unclear preconditions; no assertions
 - No need for external buffer variable
 - No clean-up after resource use
- Specific bad practices of error handling
 - Overloading of purpose of return value
 - Mismatch between method signature and behavior
 - System exception swallowed without logging
 - Inconsistent use of error codes and exceptions
 - Unprofessional logging / error message useless

**If bugs are inevitable,
how to handle them?**

System Model

Terminology

- System
- Correct service
- Incorrect service
- Component
- Boundary
- Interface
- Structure
- Behavior
- State
- User



- A fault
 - Is a condition that can cause an error
 - A fault is active, if it causes an error
 - A fault is dormant, if it has not yet caused an error
 - Can be classified by eight independent dimensions
- A software fault (the “common bug”)
 - Is always a development, internal, human-made fault
 - Is typically non-malicious, non-deliberate

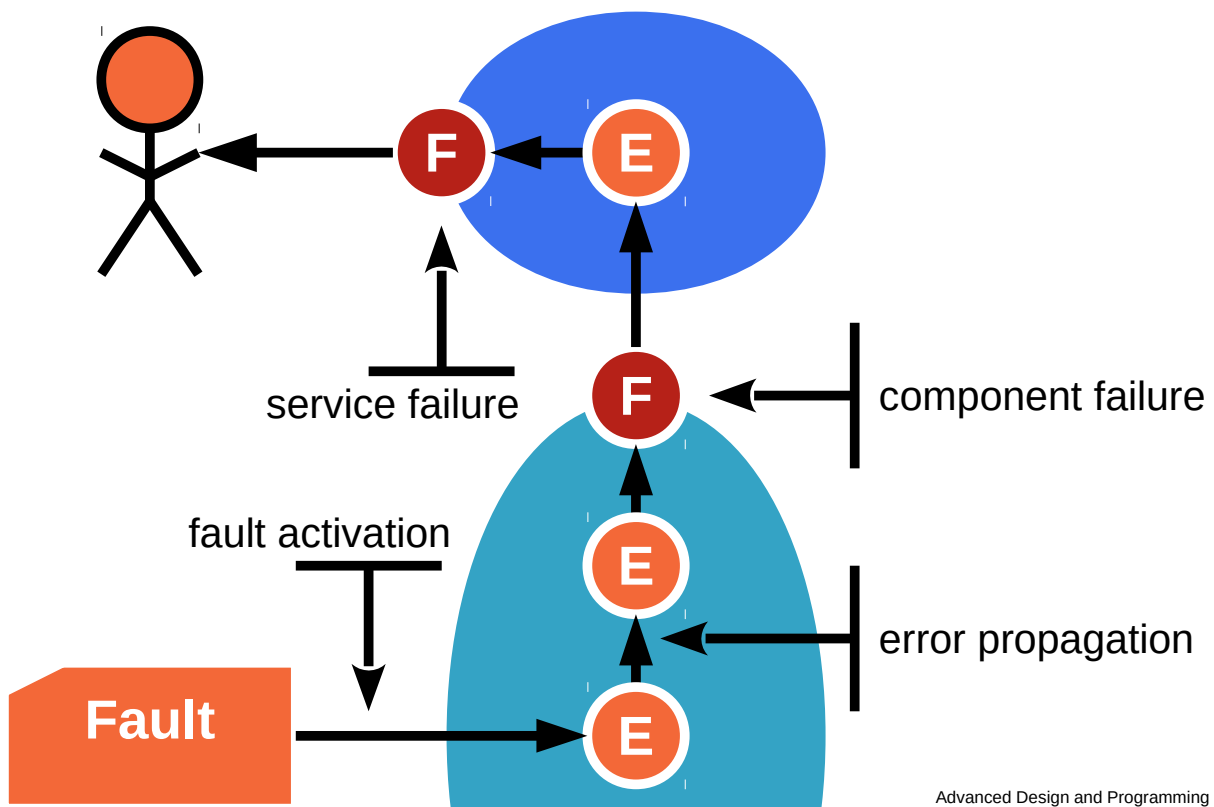
Error

- An error
 - Is a state of the system that may lead to a failure
 - Has been detected, if it has been indicated by error message or signal
 - An error is **latent**, if it has not been detected
 - Can be categorized by the failures it may cause

Failure

- A failure
 - Is an event that transitions the system from correct to incorrect service
 - 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)

Process Leading to Service Failure



- 1. Detection**
- 2. Signaling**
- 3. Handling**

Error Detection

- If an error is an incorrect state of a component, how to detect it?
 - You can't (in normal programming)
- But you can detect a failure to deliver the promised service
 - How? Using design by contract!
- Error detection = recognizing inability to provide service
 - Failing preconditions, class invariants, or post conditions
- Don't think in error states, think in failure to provide service
 - Remember: No need for defensive programming

Examples of Error Detection

```
public void insert(int i, String c) {  
    assertClassInvariants();  
  
    // assert preconditions  
    assertIsValidIndex(i, getNoComponents() + 1);  
    assertIsNonNullArgument(c);  
  
    // prepare assertion of postconditions  
    int oldNoComponents = getNoComponents();  
  
    doInsert(i, c);  
  
    // assert postconditions  
    assert (oldNoComponents + 1) == getNoComponents() : "...";  
  
    assertClassInvariants();  
}
```

Error Capture / Representation

- Information to be captured
 - Error ID
 - Error type
 - Source objects
 - Affected objects
 - Explanatory message
- Representation of information
 - Error codes
 - Error objects
 - Exception objects

Examples of Error Representation

```
protected void assertIsValidIndex(int i) throws IndexOutOfBounds... {  
    if ((i < 0) || (i >= getNoComponents())) {  
        throw new IndexOutOfBoundsException("invalid index = " + i);  
    }  
}
```

```
public class RegExpParseException extends ParseException {  
    protected String regExp = "";  
  
    public RegExpParseException(String msg, String exp, int offset) {  
        super(msg, offset);  
        regExp = exp;  
    }  
  
    public String getRegExp() {  
        return regExp;  
    }  
    ...  
}
```

Error Logging

- Possibly log the error information using system logger
 - May be helpful in case (poor) using code drops the error
 - Be slow to make assumptions about context
- Using the (system) logger
 - Write error object to appropriate logging level
 - Further functionality depends on the logger

Error Signaling

- A detected error needs to be (logged and) signaled
- Transitions the system from normal to abnormal program state

Normal vs. Abnormal Program State

- Normal program state (NPS)
 - Method performs its duties
 - Control flow returns to caller via return statement
- Abnormal program state (APS)
 - Method failed to provide service
 - Control flow returns to caller
 - via return error code
 - via thrown exception

Methods for Error Signaling

- Using normal control flow (via return)
 - Error information can be passed using
 - Return value
 - Method argument
 - Mailbox object
- Using abnormal control flow (via raising an exception)
 - Error information is passed using
 - Exception object as part of raised exception

Exercise for Error Detection and Signaling

- How to implement a basic buffer read method?

Solution Using Error Codes

```
public class File {
    public static final int ERROR_END_OF_FILE = 1;
    public static final int ERROR_PARITY = 2;

    public int readBytes(Buffer buf, int no) {
        while (no-- >= 0) {
            int err = readByte(buf);
            if (err != 0) return err;
        }
        return 0;
    }

    protected int readByte(Buffer buf) {
        if (handle.isEOF()) return ERROR_END_OF_FILE;
        byte next = handle.getNextByte();
        boolean parity = handle.getParity();
        if (parity != calcParity(next) return ERROR_PARITY;
        buf.add(next);
        return 0;
    }

    ...
}
```

Solution Using Exceptions

```
public class File {
    public byte[] readBytes(int no) throws IOException {
        byte[] buffer = new byte[no];
        for(int i = 0; i < no; i++) {
            byte next = readByte();
            buffer[i] = next;
        }
        return buffer;
    }

    protected byte readByte() throws IOException {
        if (handle.isEOF()) throw new EOFException(...);
        byte next = handle.getNextByte();
        boolean parity = handle.getParity();
        if (parity != calcParity(next)) throw new IOException(...);
        return next;
    }

    ...
}
```

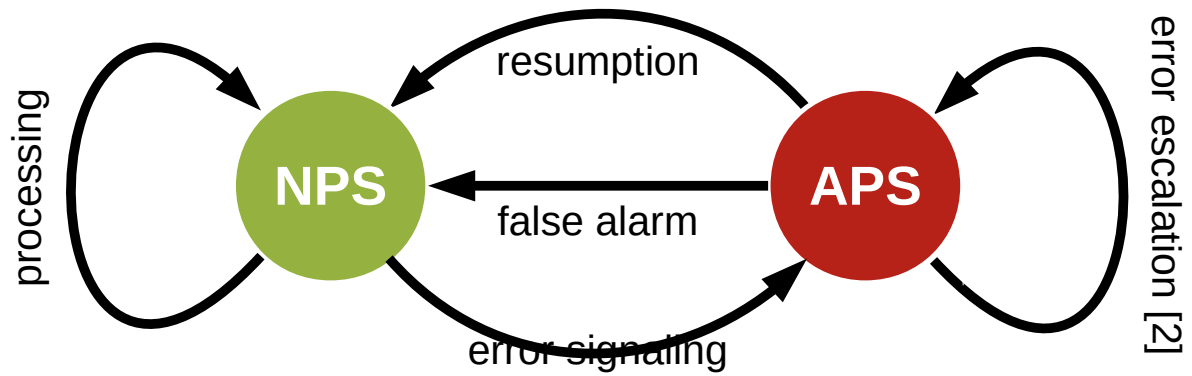
Error Code Conventions

- 0 typically indicates “no error”
- -1 typically indicates a generic error
- 1..onwards indicate specific errors

Error Codes vs. Exceptions

- Error codes are a poor error signaling mechanism
 - Mix normal with abnormal program state code
 - Separate error signal code from error information object
 - **In Java, avoid using error codes if possible**
- Exceptions were designed for error signaling
 - They separate normal from abnormal program state
 - Specifically support passing error information in exception object
 - Corollary: Don't use exceptions to make a regular return

Error Handling State Model [M92] [1]



- [1] Adjusted terminology to [A+04]
[2] Called "organized panic" in [M92]

- 1. False alarm**
- 2. Resumption**
- 3. Escalation**

Exercise of Error Handling

- How to handle error signaled by File component?

Solution Using Exception Handling

```
public class Document {
    protected byte[] buffer;
    public void loadFromFile(File file) throws DocumentException {
        ...
        int no = file.getLength();
        buffer = new byte[no];
        int tries = 0;
        for(int i = 0; (i < no) && (tries < 3); i++) {
            try {
                buffer[i] = file.readByte();
            } catch EOFException eofex {
                throw new DocumentException(..., eofex);
            } catch IOException iex {
                tries++;
            }
        }
        if (tries == 3) {
            throw new DocumentException(...);
        }
    }
    ...
}
```


Error Escalation (“Organized Panic”)

- Error escalation
 - Is the process of cleaning-up and delegating error handling to caller
 - Basically, your code has exhausted its options and gives up

Steps in Error Escalation

- Clean-up
 - Always leave the current component in a viable state
 - Make sure you restore class and component invariants
 - Restore and/or release relevant resources
 - Use finally block in exception handling to ensure this
- Escalation
 - Enhance original error information with new insights
 - Do not hide your attempts to handle the error
 - Typically, chain exceptions
 - Attach prior error information (exception) to new one

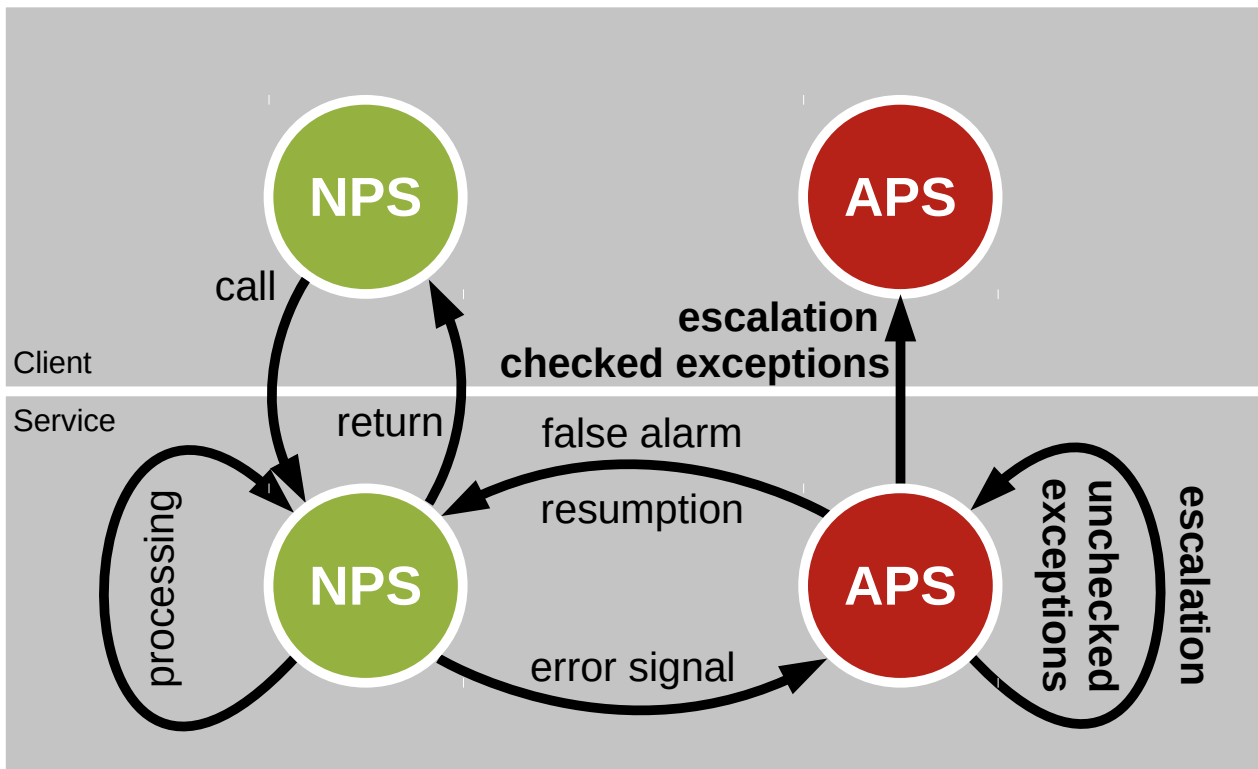
Checked vs. Unchecked Exceptions [DR]

- Checked exceptions
 - Are exceptions that must be declared in a method signature
 - Are intended to force user to take notice of the exception
 - Works well if error handling code is close to where exception was raised
 - Are a pain to handle if code is far removed from origin of exception
 - **Use checked exceptions (or error codes) in component interface**
- Unchecked exceptions
 - Are exceptions that don't need to be declared
 - Are intended to pass through client code by default
 - May make you miss an error signal that you should have handled
 - Are the only way to not complete clutter your component code
 - **Use unchecked exceptions only within your component**

Component Failure

- Error signals
 - Are part of the component interface
 - Should be specific to the component
 - **Use only checked exceptions in interface**
- **Do not let an unchecked exception escape**

Failure Handling at Component Boundary

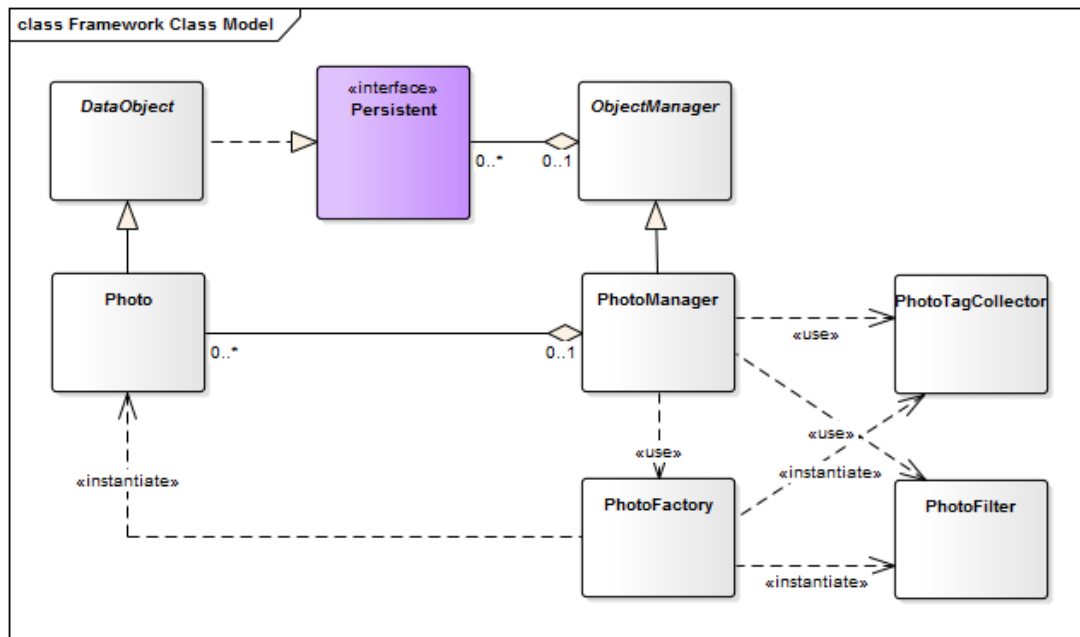


Methods for Failure Signaling

- Failure clean-up
 - Like clean-up for error signaling
 - Restore invariants
 - Release resources
- Failure escalation
 - Like error escalation but
 - Catch all unchecked exceptions
 - Escalate using checked exception
 - Provide exception chain with new one

Exercise for Component Failure

- How to handle an ObjectManager failure (to load an object)?



Solution to Component Failure Exercise

- Within component
 - Error detection
 - Catch error signal from storage layer (file or database exceptions)
 - Handle error to the extent possible; eventually, give up
 - Error signaling
 - Capture prior error signal; create new unchecked exception
 - Throw exception about inability to load object
 - Error escalation
 - If method can handle exception, do so
 - If not, let the exception pass through
- At component boundary
 - Capture internal error signal, wrap it in component-specific exception
 - Throw checked exception about component failure to environment

Service Failure (User Interface)

- A service failure
 - Is a component failure with the user as the client
 - User interface is the final system boundary
- Handling a service failure
 - Log the service failure (error)
 - Don't throw a checked exception
 - Convert the error into human-readable form and display it

Handling Faulty Components

- Well-behaved (but faulty) components
 - Follow error handling strategy as discussed
- Component of unclear quality
 - Wrap component in defensive code
 - Follow error handling strategy as discussed

Final Example of Raising an Exception [S11]

```
Exception up = new Exception("Something is wrong.");  
throw up; // ha ha
```

Review / Summary of Session

- Error handling
 - System structure: Class, component, service
 - Terminology: Fault, error, exception
- Error handling processes
 - Detection, signaling, handling
 - Life-cycle model of processing state

Thanks! Questions?

dirk.riehle@fau.de – <http://osr.cs.fau.de>

dirk@riehle.org – <http://dirkriehle.com> – [@dirkriehle](#)

DR

Credits and License

- Original version
 - © 2012-2018 [Dirk Riehle](#), some rights reserved
 - Licensed under [Creative Commons Attribution 4.0 International License](#)
- Contributions
 - ...