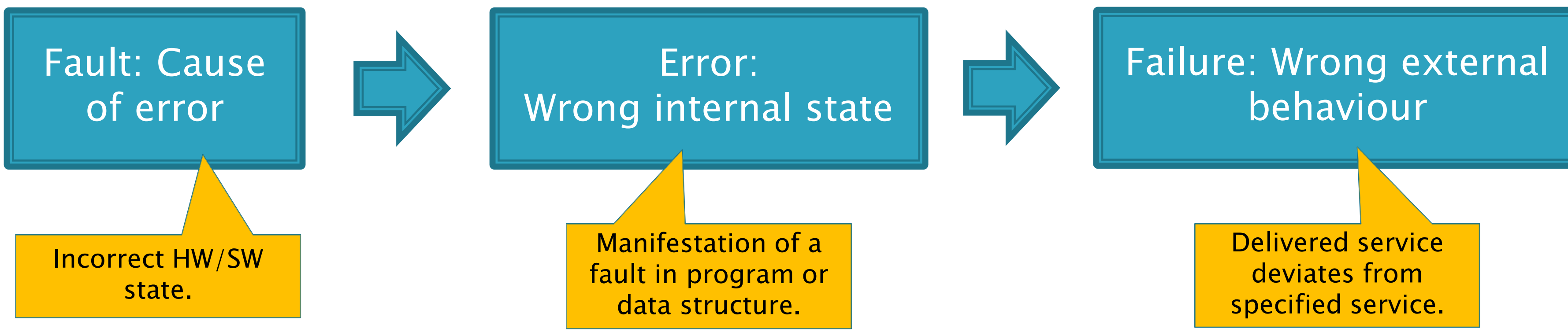
How Developers can Improve Reliability

Reliability and Failures

- Reliability is a dynamic system characteristic which is a function of the number of software failures.
- Software failure: An event where software behaves in an unexpected way
  - Behaviour not according to specification
  - Behaviour not according to expectation
- Possible reasons:
  - faults in the program
  - faulty or incomplete specifications
  - unanticipated user interaction
  - problems in the hardware or the external environment

Faults, Errors, Failures



Reliability in ISO 20000 Standard – Software Quality Requirements & Evaluation

- *Reliability*
  - degree to which a system/product/component performs specified functions under specified conditions for a specified period of time
  - *Maturity* – degree to which a system, product or component meets needs for reliability under normal operation
  - *Availability* – degree to which a system/product/component is operational and accessible when required for use
  - *Fault Tolerance* – degree to which a system/product/component operates as intended despite presence of hardware/software faults
  - *Recoverability* – degree to which, in the event of an interruption/failure, a product/system can recover the data directly affected & re-establish the desired state of the system

Example Reliability Metrics

- *Availability* – measures likelihood that the system is available for use, e.g.
  - The network is available at least 98% of the time
  - The server serves at least 98% of requests correctly (service availability)
- *Rate of occurrence of failure* – measures of frequency of occurrence with which unexpected behaviour is likely to be observed
  - E.g. if the ROCOF is 2/1000 this indicates that no more than 2 failures are likely to occur for each 1000 transactions
- *MTTF: Mean time to failure* – measures the time between observed failures
  - Used for a stable system that undergoes no changes to indicate of how long the system will remain operational before a failure occurs
- *MTBF: Mean time between failures* – measures the time between failures for a system that can recover

Reliability Specification

- Reliability requirements are often expressed in an informal, qualitative, untestable way
  - *The system should be as reliable as possible*
  - *The software shall exhibit no more than N faults/1000 lines*
- Use Failure Classes to help understand faults
  - Transient – Occurs only with certain inputs
  - Permanent – Occurs with all inputs
  - Recoverable – System can recover without operator intervention
  - Unrecoverable – Operator intervention needed to recover system
  - Non-Corrupting – Failure does not corrupt data
  - Corrupting – Failure corrupts data
- Some other failure classes:
  - Byzantine – Everything is possible
  - Timing – Response is delayed
  - Crash – No response

What is a failure/fault for this system?

Different ways for classifying faults exist.

Reliability Specification Example:
Cash Point Machine Network

Failure Class	Example	Reliability
Permanent, non-corrupting	No magnetic stripe data read from any card	1 in 300,000 transactions
Transient, non-corrupting	Failure to read magnetic stripe data on a particular card	1 in 500 transactions
Unrecoverable, non-corrupting	Software failure resulting in card return	1 in 100,000 transactions
Recoverable, corrupting	Loss of users input - users re-enter input	1 in 30,000 transactions

Assume 1,000 transactions per day

Specifics of Software Reliability

- In Software reliability, faults are usually permanent
- Reliability depends on how system is used
  - Different users use a system in different ways.
    - Users with different roles require different interactions with the system
    - There may be different features for novice, intermediate and expert users
  - A user may see problems that do not appear to others who use the system in a different way
  - Users may work around a fault or avoid features known to be faulty
- Certain types of failure are more important than others
  - Single failure in mission-critical aircraft system vs. multiple failures in ticket barcode reader

Making Software Reliable

- Avoid faults
- Deal with faults

Avoiding Software Faults

Main causes of software faults (and how to address them)

Specification errors
- Specification faulty and/or incomplete

Addressed by →

- Validate specification against requirements to ensure there are no errors or omissions

Developer errors
- Software does not meet specification

Addressed by →

- Verify implementation against specification – does the implementation fulfill the specification?
- Apply good software-engineering principles

Changes
- Changes to specification
- Changes to project

Addressed by →

- Use change management and configuration management

External Environment
- Faulty external SW component
  Faulty external SW environment
- Faulty hardware
- Faulty or malicious user

Addressed by →

- Testing – load, stress, security
- Fault-tolerant design
- Defensive programming
- Fault-tolerant programming

Choice of Software Development Approach

- Waterfall
  - Testing is done late in the lifecycle so reliability issues are not found until most of the code has been developed
    - Can be difficult and time consuming to find faults
    - Testing may be compromised to meet deadlines
  - Difficult to deal with changes
- Throwaway/Rapid Prototyping/Evolutionary Prototyping
  - Focuses on functionality rather than quality issues such as reliability
  - Can help avoid specification errors

Choice of Software Development Approach

- Incremental Development
  - Produces working software in increments so it is possible to see new faults
  - Can focus on an architecture that emphasises reliability
- Rational Unified Process (RUP)
  - Has business modelling and requirements processes to gather requirements
  - Design focusses on architecture so can emphasise reliability
  - Controls changes to software through configuration and change management process and tools
  - Has test workflow running throughout the project to verify and validate the product
  - Produces working software in increments
  - Produces UML designs and documentation to help maintain a reliable system

Choice of Software Development Approach

- Open Source
  - Many people working on and testing software as it evolves so problems get fixed
  - Theory: The more people look at software, the more likely it is that problems will be found early
  - Very much dependent on developer community
- Agile Software Development
  - Early and continuous delivery of working software in short timescales
  - Can easily adapt if changes are needed
  - Involvement of the customer – prevents specification errors

Choice of Software Development Approach

- XP (eXtreme Programming)
  - Test driven development approach with automated testing provides framework to see if an error has been introduced
  - Continuous integration and frequent builds means there is little code to fix if a problem occurs
  - Involvement of customer in developing user stories can capture customer needs
  - Weekly cycle delivers working software to meet these needs
  - Customer involved in acceptance test of user stories show system meets business needs
  - Pair programming (two people developing code on same machine) produce better quality code with less bugs
  - Developers work in same room so can easily communicate if they need to fix a problem
  - Lack of formal documentation so it may be difficult to maintain a reliable system
  - Can focus on functionality rather than designing an architecture that is more reliable

General rules for avoiding faults

- Use of iterative design and thorough testing
- Use modular design
  - Develop good structure for the whole program
  - Hide information
  - Encapsulate functionality
- Design algorithms before coding
- Comment
- Comment
- Comment
- Understand common errors (and avoid them)

Common Errors– Data Types

- Different data types have different characteristics
- Know the kinds of errors associated with each type (and handle them)
  - Number types:
    - All:
      - Division by zero
      - Overflows
      - Sign
    - Floating point: rounding errors, equality comparisons
  - Strings:
    - Buffer overflows
    - Upper/lower case
    - String encoding
  - Arrays
    - Off-by-one errors
    - Trying to access an element that is out of bounds
    - Using the wrong index in a multidimensional array

Guidelines for Conditional Code

- `If-else` statements
  - Make sure the branch condition is correct
  - Ensure normal path through code is clear
    - Put normal case in the `if` rather than `else`
    - Do not mix up normal flow and exception flow
  - Avoid complex conditions
  - For chains of `if` statements
    - Put most common case first
    - Make sure all cases are covered
    - Separate `if` statements if they are independent
    - If appropriate, use `switch/case` instead of chained `if` statements
      - Make sure you use `break` at end of each case

Guidelines for Loops

- `For` loops – when loop executes a specified number of times
  - Should not change loop control mechanisms
    - E.g avoid changing loop variable within loop
- `While` loops – a condition decides how loop terminates
  - Put loop control mechanism either at top or bottom of the loop
  - Make sure the loop terminates under all conditions
    - E.g. do not forget incrementing counters
- General advice
  - Ideally the loop should perform one and only one function
  - Readability – see whole loop on single screen
  - Avoid more than 3 levels of nesting loops
  - Use `break` and `continue` (wisely) to produce cleaner code

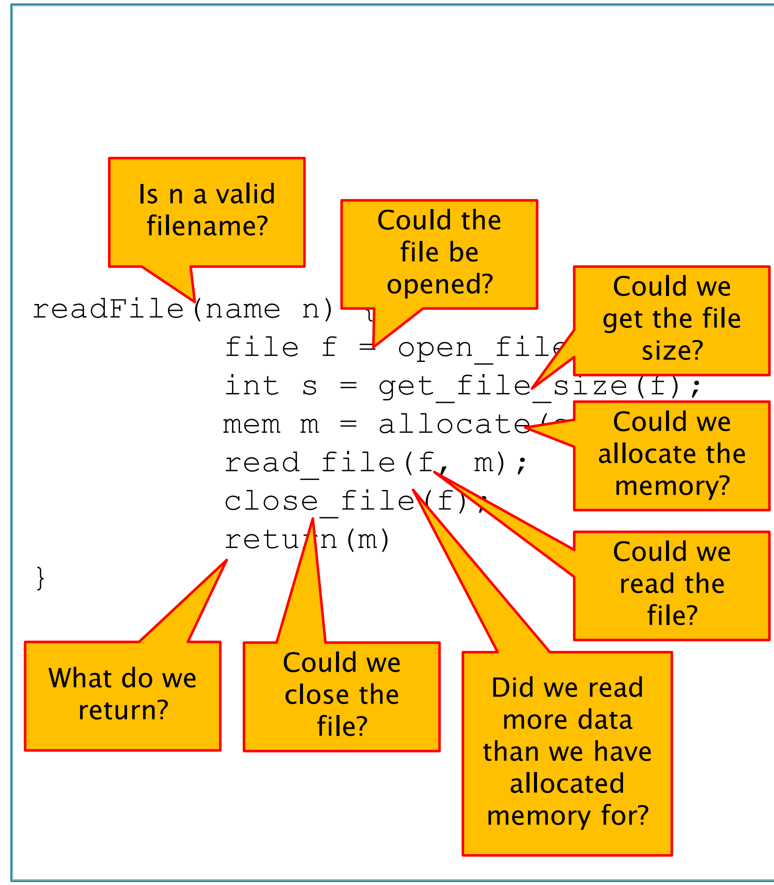
Dealing with faults

Defensive Programming

- "Garbage In does not mean Garbage Out" (McConnell, p. 97)
- Good programs do not output garbage regardless of the input
- Anticipate faults
- Detect faults
- Handle faults:
    - Correct errors (if possible)
    - Report
    - Fail fast

Detecting

- Check values of all data input from users or external systems and devices
- Check values of all input parameters to a method or function
- Detect errors in calling other modules
- Detect environmental errors
- Make sure return values are reasonable



Handling

- If possible, correct faults, e.g.
  - Correct obviously faulty data
  - Recover corrupted data, if possible
  - Retry if transient error is likely (e.g. network)
- Report faults
  - Log
  - Error message to user
  - A well-constructed error message
    - should identify the program that is posting the error message
    - should alert the user to the specific problem
    - should provide some information as to how solve the problem
    - should suggest where the user may obtain further help
    - should not contain unhelpful, redundant, incomplete, or inaccurate information
    - should provide an identifying code to distinguish it from similar messages
    - should be polite and inoffensive
    - should include a timestamp
- Fail fast
  - If you cannot correct or tolerate the fault, at least avoid causing more harm

```
readFile(name n) {
   if (is_file_name(n)) {
    file f = open_file(n);
    if (is_open(f)) {
      int s = get_file_size(f);
      if (s > 0) {
       mem m = allocate(s);
       if (length(m) == s) {
        int r =
          read_file_max_length(f, m,
                    length(s));
        if (s > r) {
          log("File size changed.")
        }
       } else {
          log("Could not allocate memory.");
       }
      } else {
         log("Could not get file size.");
      }
      close_file(f);
      if (is_open(f)) {
        log("Could not close file.");
      } else {
        return(m);
      }
  } else {
     log("Could not open file.");
  }
}
```

Handling: Exceptions

- Mixing normal and error-handling code results in unreadable programs
- Exceptions
  - Provide dedicated error-channel through program
  - Separate normal and error-handling code
  - Group and differentiate error types
- Definition: *An event which occurs during the execution of a program that disrupts the normal flow of the program's instructions*

  The Java Tutorials

```
readFile(name n) {
   if (is_file_name(n)) {
    file f = open_file(n);
    if (is_open(f)) {
      int s = get_file_size(f);
      if (s > 0) {
       mem m = allocate(s);
       if (length(m) == s) {
        int r =
          read_file_max_length(f, m,
                    length(s));
        if (s > r) {
          log("File size changed.")
        }
       } else {
          log("Could not allocate memory.");
       }
      } else {
         log("Could not get file size.");
      }
      close_file(f);
      if (is_open(f)) {
        log("Could not close file.");
      } else {
        return(m);
      }
  } else {
     log("Could not open file.");
  }
}
```

Handling: Exceptions

- Exceptions
  - Get thrown when an error is detected
  - Are propagated up the call stack
  - Get caught by an exception handler
  - Are objects with data and behaviour
- Java syntax:

```
if (error) {
  throw new ExceptionTypeA();
}
```

Somewhere else:

```
try {
// normal code
} catch (ExceptionTypeA e) {
// executed for Type A
} catch (ExceptionTypeB e) {
// executed for type B
} finally {
// always executed
}
```

```
readFile(name n) throws FileReadFailed,
                 NoFileName {
  try {
    if (! is_file_name(n)) {
      throw new NoFileNameException();
    }
    file f = open_file(n);
    int s = get_file_size(f);
    mem m = allocate(s);
    int r = read_file_max_length(f, m,
                    length(s));
    if (s > r) {
       log("File size changed.")
    }
    close_file(f);
    return(m);
  } catch(OpenFailed e) {
     log("Could not open file.");
     throw new FileReadFailed(e);
  } catch(AllocateFailed e) {
     log("Could not allocate memory.");
     throw new FileReadFailed(e);
  } catch(CloseFailed e) {
     log("Could not close file.");
     throw new FileReadFailed(e);
  } finally {
     close(f);
  }
}
```

Handling: Exceptions (ctd.)

- Python syntax:
-

```
if error:
  raise ExceptionTypeA()
```

Somewhere else:

```
try:
 normal code
except ExceptionTypeA as err:
 # handle Type A
except ExceptionTypeB as err:
 # handle Type B
else:
 # execute without exception
finally:
 # always run
```

Fault Masking

- Design system such that faults can be masked
- Faults do not need to be detected
- Examples:
  - Redundancy in time: Repeat computations several times and use majority vote for result
  - Redundancy in space: Perform same operation on independent copies, use majority vote

Fault Masking: N-Version Programming

- Development time:
  - Hand same specification (or same requirements) to an odd number of independent development teams
  - Have all teams develop software
- At run-time:
  - Run computation on all versions of the software
  - Use reliable voting mechanism to select result
- Disadvantages:
  - Extremely expensive – only applicable for very high-risk areas
  - Independence difficult to ensure
    - Same practices and mindsets
    - Common frameworks

Conclusion: What Good Programmers Know

- The principles underlying programming languages
- A wide range of data structures and their characteristics
- A wide range of algorithms and their characteristics
- A good programming style
- Kinds of error commonly made by themselves in particular and programmers in general
- Ways of dealing with errors when they occur

References and recommended reading

- Steve McConnell (1993) "Code Complete", Microsoft Press
- Sun/Oracle: The Java Tutorial, especially the lesson on exceptions: https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html (last accessed 5 February 2019)
- Said van de Klundert: Python Exceptions: An Introduction: https://realpython.com/python-exceptions (last accessed 5 February 2019)