Course "Debugging"

Logging and Tracing

Prof. Dr. Lutz Prechelt
Freie Universität Berlin, Institut für Informatik
http://www.inf.fu-berlin.de/inst/ag-se/

- Debugging as search in space-time
- Pros and cons of debuggers, The role of logging
- java.util.logging
- Inserting logging calls via aspects

Debugging as a search in space-time

Freie Universität Berlin

- Debugging means locating the defect **D** that leads to an observed failure **F**

- The relevant aspect of the defect is its influence on program execution
  - Either an incorrect modification of a variable ("infects" the program state)
  - or incorrect control flow

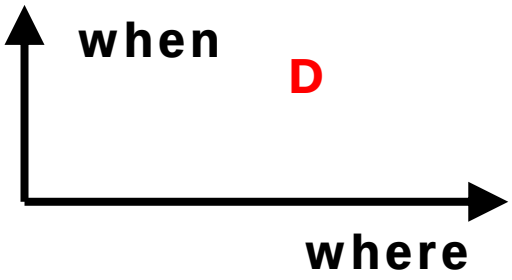
- The location **D.s** in the program that is wrong is the **space dimension of the defect**

Debugging as a search in space-time (2)

- However, the same location may execute correctly hundreds of times (even in the same run) before it executes incorrectly
  - depending on the program inputs

- The time **D.t** at which D.s executes incorrectly is the **time dimension of the defect**

- Finding a defect by observation is thus a two-dimensional search in
  1. program space and
  2. execution time



  - Remark: This is why reviews are so powerful. They attempt to consider all executions at once. If successful, this reduces the search space to only the space dimension.

Defects from data flow or control flow: example

Freie Universität Berlin

Assume we have observed **y** happening, where **x** should have happened

- We trace **y** to be the call in the following statement:
  - *if (**a** || **b**) **x()** else **y()**;*

- If we find that the statement ought have been
  - *if (**a** || **b**) **y()** else **x()**;*
- then the defect is a control flow defect in this line.
- However if we find that **a** was *false*, but ought have been *true*
- then the defect is an incorrect modification (or lack thereof) at some previous assignment to **a**
  - This might again be control-flow-related. Or not.

Strengths of debugger programs

Freie Universität Berlin

Where debuggers can help a lot:

- Observing program execution in great detail
  - → single-stepping
    - feasible only for short sections of execution
- Investigating program state in great detail
  - → looking at contents of variables; following references
- Catching events that are understood precisely
  - → conditional breakpoints

Weaknesses of debugger programs

Where debuggers are hardly useful:

- Observing long stretches of program execution
  - *"What is the program doing all that time?"*
- Going backwards in time
  - *"How did we get to this point?"*
  - *"What happened before this?"*
- Catching odd events not known in advance
  - *"Is anything unusual happening?"*

Solution: An execution protocol

In the above-mentioned cases, we need a protocol of the program execution to do the job:

- Observing long stretches of program execution
  - → Skim the protocol
- Going backwards in time
  - → Once you found something interesting, investigate preceding protocol entries
- Catching odd events not known in advance
  - → Search for violations of patterns in the protocol

Such protocols are commonly known as *log* or *trace*

- Traces tend to be detailed and complete
- Logs may be selective
- Most people use the terms interchangeably

Print-debugging

- Beginners in programming tend to spontaneously invent logging during their first debugging episodes
  - They insert 'print' statements indicating
    - the values of certain variables or
    - that the program has reached a certain point

- Disadvantages of this approach:
  - Output must be tailored to the specific debugging purpose
    - or the output may quickly overwhelm the programmer
  - Output tends to be hard to read
    - or creating it is cumbersome
  - The investment is very short-term
    - In particular, statements need to be taken out for serious use of the program

Solution: Configurable logging or tracing

Freie Universität Berlin

The state-of-the-art solution to logging involves a special-purpose logging library:

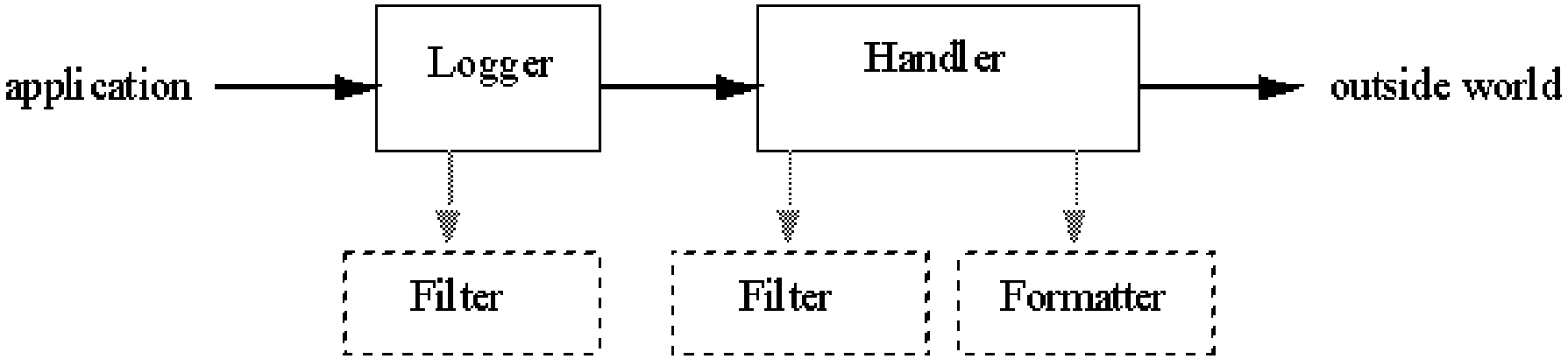
- Insert calls to the library at many points in the program
  - often purely precautionary
- Configure which calls should actually produce output
  - types/categories, priorities/levels, sources, etc.
- Configure how the output should look like
  - formatters
- Configure where the output should go to
  - sinks, targets, handlers, etc.

Large projects use such logging routinely (as a precaution)

java.util.logging (j.u.l.)

- Java 1.4 introduced a logging library (JSR 47)
  - *java.util.logging*

- Basic concepts:
  - The application makes calls to a ***Logger*** object.
  - Loggers suppress some calls based on ***Levels*** and ***Filters***.
  - Unsuppressed calls are propagated to a ***Handler***,
  - which applies a ***Formatter*** and determines a sink.



j.u.l.Logger

- *Loggers* have an arbitrary name
  - convention: use a dot-separated, hierarchical namespace according to the package names
    - deviations are possible as needed
  - *Loggers* are parent-child-related according to the namespace (the root Logger has name "")
- *Loggers* have a *level*
  - and ignore all calls for lower levels rightaway
    - other calls will allocate a *LogRecord*
  - level is defined by *LogManager* (→config file) or by *setLevel*
  - if level is null, level is inherited from parent *Logger*
- Loggers are obtained via
  static Logger getLogger(String name)
  - delegates to LogManager (see below):
    result is unique-per-name

j.u.l.Logger

- Loggers have a large number of logging methods
  Examples:
  - *log(LogRecord record)* the basic logging operation
  - *log(Level level, String msg)* log simple message
  - *log(Level level, String msg, Object[] params)*
    log message with parameters
  - *fine(String msg)* log simple message with Level FINE

- *isLoggable(Level level)* checks the Logger's Level
- Loggers can be configured:
  - *addHandler(Handler), removeHandler(Handler)*
  - *setFilter(Filter), setLevel(Level), setParent(Logger)*
  - *setUseParentHandlers(boolean)* send outout to parent, too
  - (some operations require the application to have *LoggingPermission* in the *SecurityManager*)

j.u.l.LogManager

A singleton class that manages the hierarchy of *Loggers*

- *static LogManager getLogManager()*

- by default, Logger configuration will be read from the Properties file *$JRE/lib/logging.properties*
  - Entries for example:
    java.util.logging.ConsoleHandler.level = INFO
    java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
    com.xyz.foo.level = SEVERE
- *readConfiguration(InputStream)* re-initializes Logger parameter configuration
- *getLoggerNames()* enumerates all *Loggers*
- *getLogger(String name)* used by *Logger.getLogger*

j.u.l.Level

- *java.util.logging.Level* defines the logging levels:
  - *OFF* (no logging)
  - *SEVERE* (highest value)
  - *WARNING* > *INFO* > *CONFIG* > *FINE* > *FINER*
  - *FINEST* (lowest value)
  - *ALL* (complete logging)
- *intValue()* returns the underlying integer representation
- *new Level("FATAL", Level.SEVERE+1)* creates new *Level*
- *getName()* obtains a String representation
- *getLocalizedName()* adds i18n
  - all other j.u.l. classes can use *ResourceBundles*, too

j.u.l.LogRecord

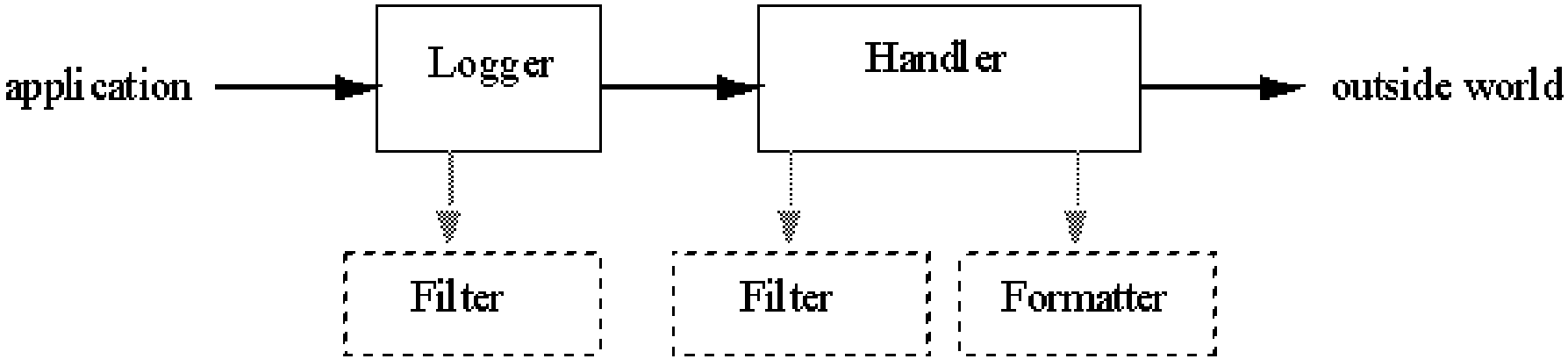
- Represents the results of an unsuppressed *Logger* call
  - Is a pure data class

- Has getters and setters for the following attributes:
  - *level, loggerName, message*
  - *parameters* (an *Object[]*)
  - *millis* (timestamp)
  - *sequenceNumber*
  - *sourceClassName, sourceMethodName, threadID, thrown*
    - class and method are either supplied as parameters at a logging call
    - or will be determined by Logger (as well as the JVM permits)

j.u.l.Filter

- *Loggers* and *Handlers* can have a *Filter*:
  - *interface Filter { boolean isLoggable(LogRecord record); }*
  - can suppress individual log calls based on arbitrary criteria

- *Loggers* will check each call against the *Filter* before creating a *LogRecord*
- *Handlers* will check each incoming *LogRecord*
- There are no predefined *Filters*

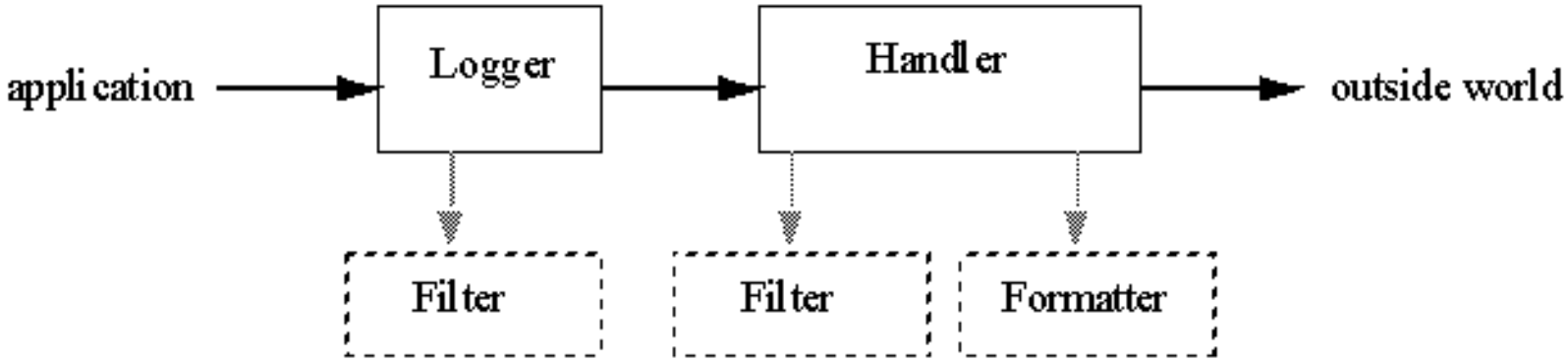


j.u.l.Handler

- Handlers are responsible for producing the actual logging output:
  - perhaps filter some more (*Level, Filter*),
  - format the message (*Formatter*),
  - perhaps do buffering, grouping, or rearranging
  - write to the output channel
  - perhaps report write errors (*ErrorManager*)
- *publish(LogRecord record)*
- There are several predefined *Handlers*:
  - *ConsoleHandler* writes to *System.err*
  - *FileHandler* writes to a file or rotating set of files
  - *MemoryHandler* buffers w/o formatting, forwards
  - *SocketHandler* writes to network, defaults to XML
  - *StreamHandler* writes to any *OutputStream*

j.u.l.Formatter

- Converts a *LogRecord* into a String
  - associated with (and called by) a *Handler*
  - *String format(LogRecord record)*
- *java.util.logging* provides two predefined *Formatters*
  - *SimpleFormatter*
    - creates a compact, human-readable format
  - *XMLFormatter*
    - creates XML according to a standardized DTD
- Projects will often have their own customized replacement for *SimpleFormatter*



j.u.l.: Example

```
package com.wombat;
import java.util.logging.*;
public class Nose {
    private static Logger logger =
            Logger.getLogger("com.wombat.nose");
    public static void main(String argv[]) {
      logger.fine("doing stuff");
      try {
            Wombat.sneeze();
      }
      catch (Error ex) {
        logger.log(Level.WARNING, "trouble sneezing", ex);
      }
      logger.finer("done");
    }
}
```

j.u.l.: Example (2)

- Default configuration see
  - *$JRE/lib/logging.properties*

- Modifying the configuration in program code:

```
Handler fh = new FileHandler("%t/wombat.log");
Logger.getLogger("").addHandler(fh);
Logger.getLogger("com.wombat").
      setLevel(Level.FINEST);
```

Other logging solutions

Other Java libraries for logging are also in wide use:

- Log4J (probably still the market leader, Jakarta)
  - concepts are similar to java.util.logging
    - e.g. Category, Appender, Layout
  - bigger set of formatters and sinks than java.util.logging
  - also available for *Perl* (Log::Log4perl)
- JLog (IBM)
- Protomatter Syslog (sourceforge)
  - basic generic adapter API for plugging in different implementations (e.g. any of the above)

Similar packages are available for many other languages.

Other advantages of logging

- Logs enable debugging after-the-fact
  - I.e. after the program run has finished
  - Indispensable for remote customer support
- Logs may be available even when debuggers are not
  - e.g. for concurrent programs
  - e.g. for distributed systems
  - e.g. for embedded systems
- Logs support understanding the code of an unknown system
  - if they identify source classes/routines.
  - Great help for a newcomer software engineer

Disadvantages of logging

- Logging statements (even deactivated ones) can slow down an application
  - if used too densely
- Log data can fill disks or DBMSs
  - and lead to system crashes
- Amount of output can overwhelm a user
  - and bury important error messages

Logging pitfalls

Consistency and understandability:

- Just like error messages, log messages should use a consistent syntax and terminology
  - This is not easy to do!
- The Level appropriate for a message is also often unclear

Filtering design:

- Just like with module interfaces, the properties of a log message should be flexible (here: w.r.t. filtering)
  - Should allow for all kinds of filtering desired later
- But often, the same filtering equivalence-class turns out to hold both relevant and irrelevant outputs
  - Problematic if the irrelevant ones have high frequency

Logging pitfalls (2)

Performance:

- Too fine-grained logging calls strangle performance
  - Even if almost all of them are suppressed
  - They always involve a method call and level check
    - java.util.logging is highly optimized, though

- In particular, calls with expensive argument construction strangle performance
  - Arguments are evaluated even if the call will do nothing
  - Beware of String concatenation (→ time, garbage):
    log(Level.FINE, "Values: " + a + ", " + "b" + " (" + paramlist + ")");
    - Use explicit guards using *isLoggable* instead
    - Even better: use calls with separate parameters (to be assembled by the *Formatter*)

Other purposes of logging

- Logging is an important ingredient of system monitoring
  - Often the logging output is funneled into a system management tool such as
    - IBM Tivoli or
    - Computer Associates Unicenter

- Many applications use logging for auditing purposes
  - Not a technical function, such as for debugging
  - But rather a functional requirement of the application
  - Highly relevant in domains with security needs

How to insert logging instructions: Aspects

Freie Universität Berlin

- Inserting logging statements into a large program can be cumbersome
  - and sometimes you even want to modify them later…
- Rather than doing it all manually, ***aspect-oriented programming (AOP)*** may provide some automation
- "Aspect" is a concept that adresses the phenomenon of ***cross-cutting concerns***:
  - Large programs are modularized (e.g. by means of classes)
  - Modularization provides separation of concerns,
  - but works only for functional concerns.
  - Other concerns necessarily affect more than one class,
  - we call them cross-cutting concerns

Examples of cross-cutting concerns

- notifications
  - e.g. for logging, statistics gathering, display updating, etc.
- checks
  - e.g. for security, design contracts, coding conventions, etc.
- error-handling
- distribution concerns
- feature variations
- context-sensitive behavior
- persistence
- testing

The AOP idea

- Aspect-oriented programming aims at
  - languages and methods that allow for
  - designing and programming cross-cutting concerns
  - in a modular fashion.
  - Rather than *scattering* them throughout the program
  - or *tangling* two or more in some small code region

- The basic idea is describing rules for centrally adding code to certain (perhaps many) places in a program
  - These rules are called ***aspects***
  - Aspects are similar to classes
    - At least in some forms of AOP; it can take other forms

- Note: Approaches other than AOP exist that can address (at least some kinds of) cross-cutting concerns:
  - reflective systems, mixin-based inheritance

AOP implementations

- There are a number of tools that extend Java (or other languages such as C#, C++, etc.) with AOP constructs.
  - see http://aosd.net/

- Some examples (for Java):
  - AspectJ http://eclipse.org/aspectj/
    - AOP-to-bytecode compiler, AOP/bytecode-weaver
    - most widely used AOP tool
    - will (as of 2005-01) be merged with AspectWerkz
  - AspectWerkz http://aspectwerkz.codehaus.org/
    - bytecode modification at buildtime, loadtime, or runtime
    - also supports Java 5 annotations for the aspects
  - JBoss AOP http://www.jboss.org/
    - used in the JBoss 4.0 J2EE Application Server
    - includes a set of reusable aspects for caching, asynchronous communication, transactions, security, remoting, etc.

AspectJ

- A reliable, well-balanced AOP implementation
  - Now being further developed as an Eclipse project
- Consists of
  - a compiler (ajc),
  - a small runtime system,
  - an extended Javadoc tool (ajdoc),
  - ant tasks, etc.
- IDE plugin API available (AJDE)
  - integration for Eclipse, NetBeans, JBuilder, Emacs, XEmacs

AspectJ basic concepts

- **Join point**: A well-defined point in the execution of a program
  - e.g. reading/writing a field; executing an exception handler, method or constructor
  - a dynamic concept: refering to runtime, not to code
- **Pointcut**: A description of a set of join points
  - Takes the form of a declaration
- **Advice**: The code that executes before or after each join point in a pointcut

Aspects

- Aspects are much like classes, with these extensions:
  An aspect can have (as members):
  - Pointcuts
  - Advice
  - Inter-type declarations
    - i.e. declare members of other classes (and access them)

- Aspects do not have constructors
  - They apply implicitly based on their scope

AspectJ example

Freie Universität Berlin

- ```
  aspect PublicErrorLogging {
    Logger log = Logger.getLogger("global");
    pointcut publicInterface(Object o):
      call(public * com.xerox.*.*(..)) && target(o);
    after(Object o) throwing (Error e): publicInterface(o) {
      log.log(Level.fine, o, e);
  } }
  ```

- Effect:
  - Whenever any public method of a *com.xerox.** class throws an Error,
    that error is logged before being thrown to its caller
    - Aspects are singletons, so there is only one Logger

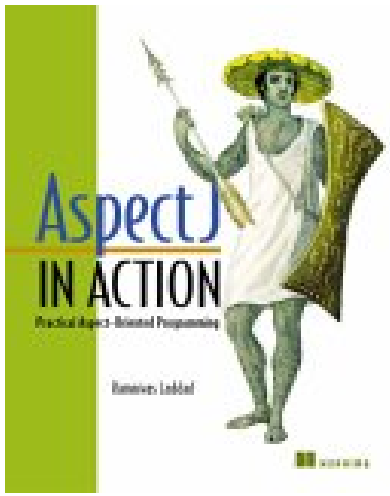
- Conventional implementation: a large number of methods have a try/catch around their body
  ```

Advantage of AOP for debug-style logging

- There can be many separate aspects that insert logging calls,
  - one for each kind of debugging situation,
  - each one rather specific.
  - This greatly improves the overview of the logging structure

- The aspects (and hence logging calls) can be left out of production systems easily.

- Want to know more about AspectJ and AOP?:
  - Ramnivas Laddad: *AspectJ in Action*, Manning 2003



Summary

- Logging is a useful tool for debugging when you need to
  - observe long stretches of program execution
  - go backwards in time (or debug after-the-fact)
  - catch odd events not known in advance
- or when no debugger is available
  - e.g. in some concurrent, distributed, or embedded situations

- Logging is also useful for other purposes
  - auditing, monitoring, etc.

- Since JDK 1.4, *java.util.logging* provides a logging library
- Aspect-oriented programming can help applying it
  - for instance via *AspectJ* or some other tool

The nine rules: Where/how to use logging

When applying this rule	***role of logging***
• Understand the system	watch it run
• Make it fail	observe the failure
• Quit thinking and look	objective and reliable
• Divide and conquer	–
• Change one thing at a time	–
• Keep an audit trail	logging does much of it
• Check the plug	Has X occurred?
• Get a fresh view	–
• If you don't fix it, it ain't fixed	simplify verification

Thank you!