MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

Software Design and Construction

159.251

Orthogonality and Separation of Concerns

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

References

[PP] Andrew Hunt and David Thomas:
The Pragmatic Programmer: From Journeyman to Master.
Addison-Wesley, Oct 1999.

[CC] Robert Martin:
Clean Code: A Handbook of Agile Software Craftsmanship.
Prentice Hall 2009.

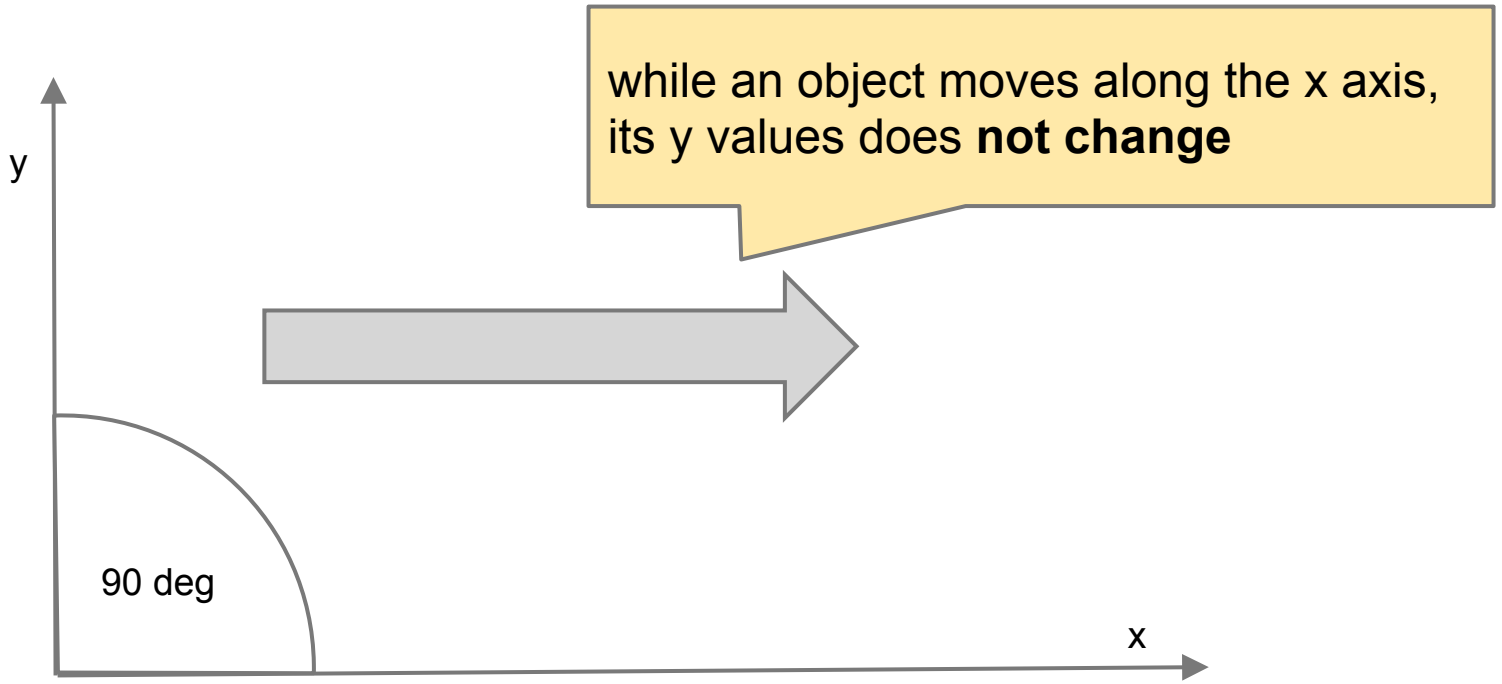
[EJ] Joshua Bloch:
Effective Java Second Edition.
Sun Micro 2008.

Additional Readings

- Ceki Gülcü: A Short Introduction to Log4J.
  http://logging.apache.org/log4j/1.2/manual.html
- Robert Martin: OO Design Quality Metrics - An Analysis of Dependencies.
  http://www.objectmentor.com/resources/articles/oodmetrc.pdf
- Robert Martin: The Dependency Inversion Principle.
  http://www.objectmentor.com/resources/articles/dip.pdf
- Jens Dietrich: Orthogonality By Example. 2013
  http://www.javaworld.com/javaworld/jw-05-2013/130501-jtip-orthogonality-by-example.html

Orthogonality

- a metaphor from geometry
- independence / no interference between orthogonal axis



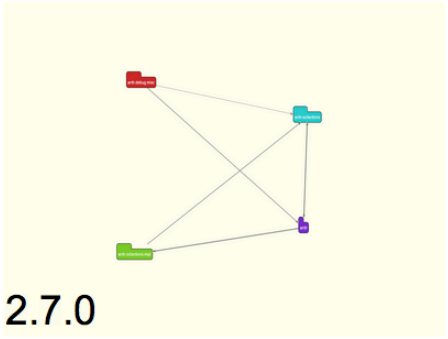
Orthogonality in Software

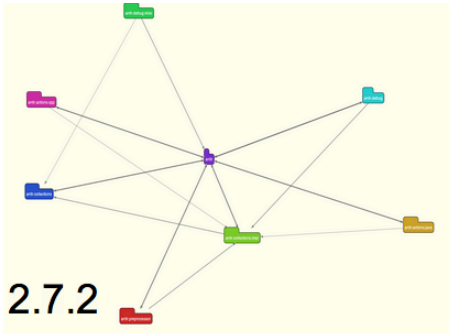
- aim: to change something, without having to change other things
- control the ripple effects when software changes
- aka **decoupling -** opposite of (tight) coupling: many dependencies between two artefacts (classes, methods, libraries) - high probability that changing one results in changing the other

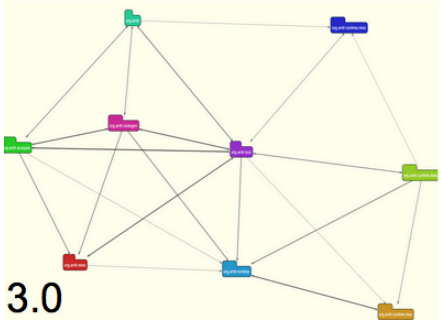
- leads to **localising change**
- needed: eliminate effects between unrelated parts of the software

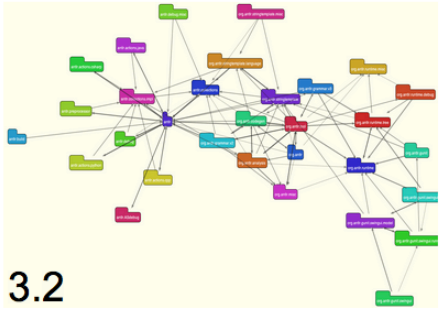
Example: Ripple Effects

Software is Highly Connected !



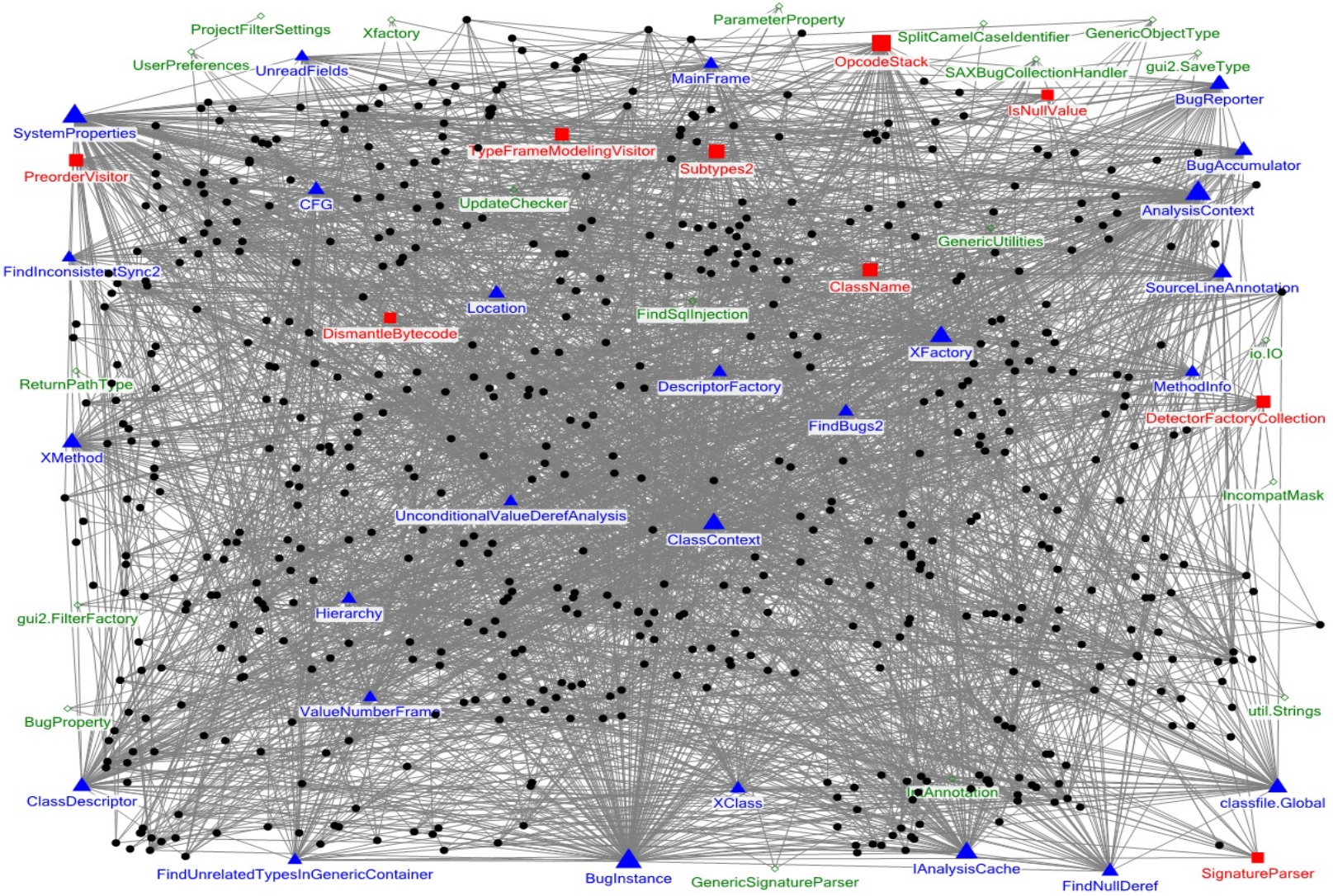






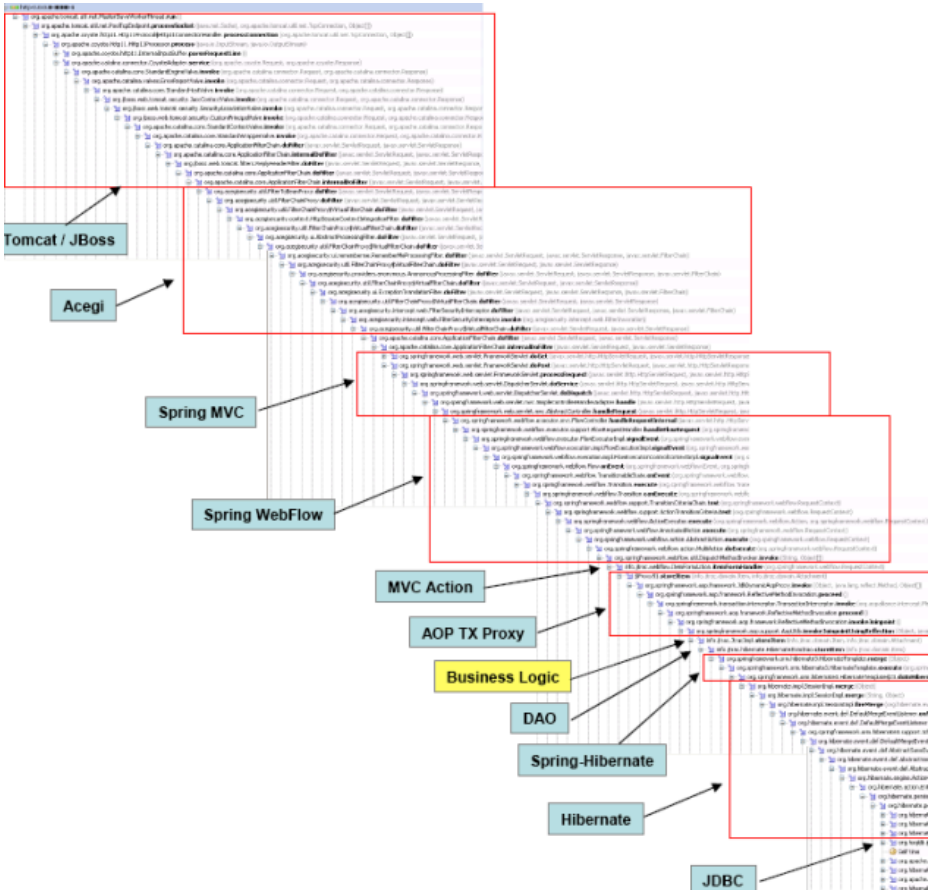
java packages and their dependencies in several versions of ANTLR

Dynamic dependencies (FindBugs)
(when you run the program)



Example: Ripple Effects

An example callstack



source:
http://ptrthomas.wordpress.com/2006/06/06/java-call-stack-from-http-upto-jdbc-as-a-picture/

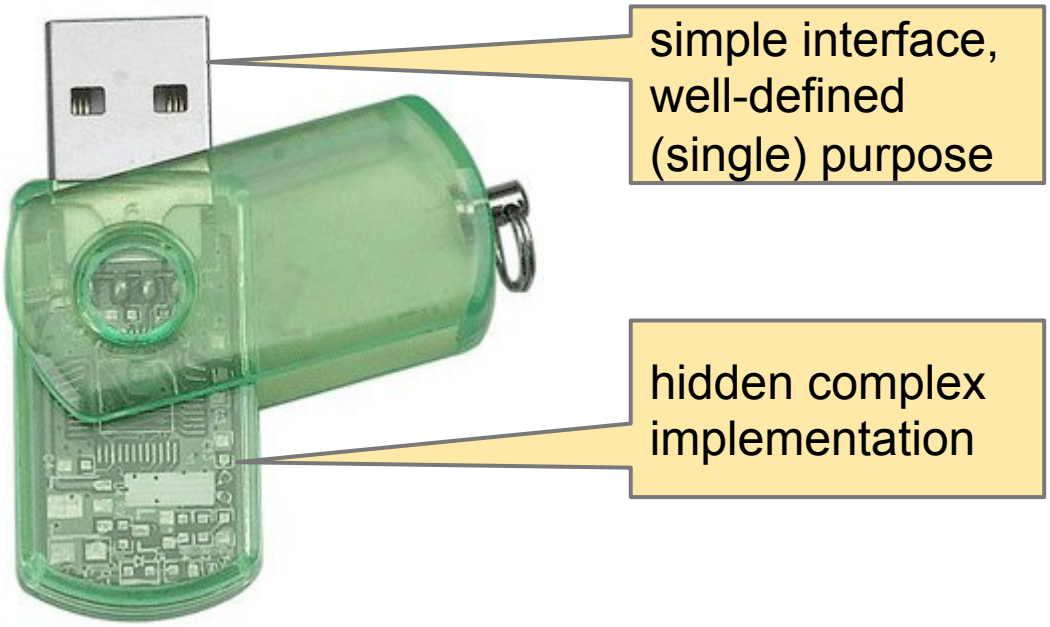
Designing for Orthogonality

- every piece of software (method, class) should have a **single, well-defined purpose**
- the public APIs expose functions related to this purpose, and hide the rest
- complex code necessary to achieve this functionality is hidden (**encapsulated**)
- in languages with access modifiers, this can be enforced (by the compiler)

Every piece of software should have a **single, well-defined purpose**

Designs like this do not work well!

Encapsulation



The "public interface" is defined here:
Universal Serial Bus Mass Storage Class Specification.
http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf

Reaping the Benefits

Recap: Java Access Modifiers

- for members (methods and fields):
  - Private?
  - Default?
  - Protected?
  - Public ?

Recap: Java Access Modifiers

- for members (methods and fields):
  - Private - only visible within class
  - Default - visible within package
  - Protected - visible within package and to subclasses
  - Public – visible

- for (top-level) classes:
  - only public and default for top-level classes
  - inner classes can also be private

- *Advanced: control visibility of entire packages using classloaders - this is done in OSGi (e.g., Eclipse plugins are based on OSGi)*

- OSGi (Open Service Gateway Initiative): https://en.wikipedia.org/wiki/OSGi

Cohesion and Coupling

- By modules we mean: classes/packages
- coupling between modules should be loose to localise change
- but within modules there should be cohesion - everything should "work together"
- several cohesion metrics have been proposed to measure (lack of) cohesion
- these concepts were first proposed in

  W. Stevens, G. Myers, L. Constantine, "Structured Design", IBM Systems Journal, 13 (2), 115-139, 1974.

Case Study: Log4J

- log4J is a log package for Java
- it provides an expressive alternative for console logging
- console logging - basic:
  `System.out.println("Hello World");`
- console logging - improved:
  - use two **loggers** `System.out` and `System.err` to separate debug info and exception reporting
  - `System.out` and `System.err` are print streams, they can be redirected to write to files
  - use `System.setOut()` and `System.setErr()` to replace default streams
- source code:
  https://bitbucket.org/jensdietrich/oop-examples/src/1.0/log4j/

Case Study: Log4J (ctd)

- purpose of case study:
  - a good example of orthogonal design
  - logging is a **much** better alternative to `System.out.println`!

- alternatives to log4j:
  - java.util.logging package, part of Java
  - apache commons logging is an abstraction for different logging frameworks

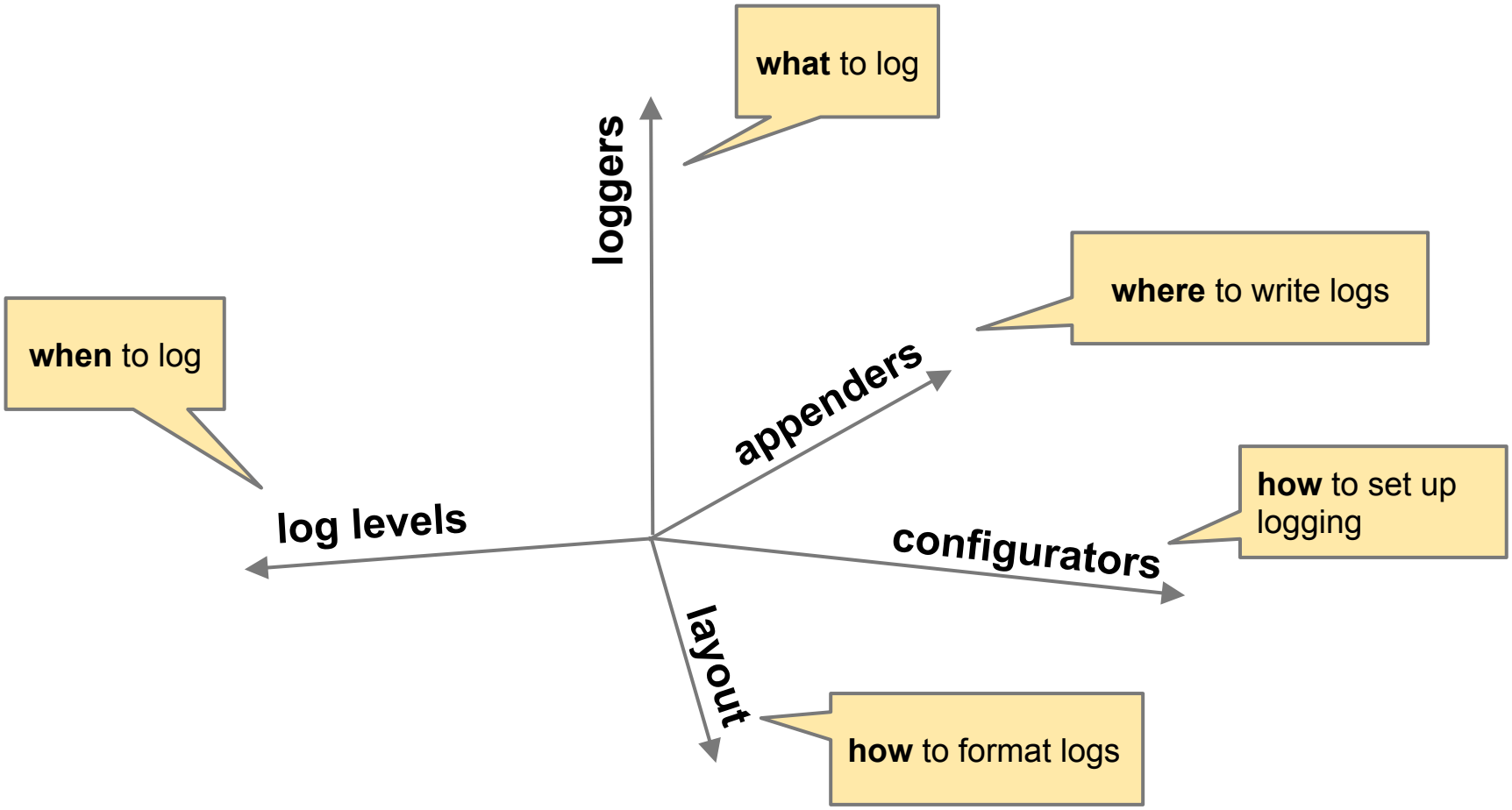
Limitations of Console Logging

- invasive code, difficult to switch on/off as needed
- often used as poor replacement for debugging
- writing directly to a file is slow, some buffering is needed
- dangerous practise of composing strings when logging:

```
System.out.println("this action took " + t + " ms to execute");
```

  - string concatenation slow in Java (although optimised in more recent versions)
  - log levels to coarse (only two)
  - need different loggers for different parts of applications (e.g., enable logging for UI only)

The Five Dimensions of Log4J



Out of context: how can you visualise 5D !?

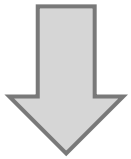
Log4J Hello World

```
BasicConfigurator.configure();

Logger logger = Logger.getLogger("Foo");

logger.debug("Hello World");
logger.warn("it's me");
```

set up logging - basic

create a named **logger**

log something



console output

```
0 [main] DEBUG Foo  - Hello World
1 [main] WARN Foo  - it's me
```

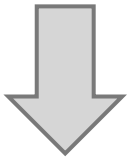
Loggers

- loggers are used for logging - they abstract from `System.out` and `System.err`
- loggers have names
- loggers form a hierarchy defined by hierarchical names - a logger named `com.sample` is **parent** of the logger named `com.sample.MyClass`
- often, loggers are created for packages and classes
- if a message is sent a logger, it is also sent to its parent
- there is a root logger on the top of the hierarchy

Setting the Log Level

```
BasicConfigurator.configure();
Logger logger = Logger.getLogger("Foo");

logger.setLevel(Level.INFO);

logger.debug("Hello World");
logger.warn("it's me");
```

set log level to INFO:
includes WARN,
excludes DEBUG



console output

```
0 [main] WARN Foo  - it's me
```

debug statement not logged!

Log Levels

- allows to configure how much to log
- reconfigurable at runtime
- e.g., an application can be set to "debug mode" to trace problems without restarting it
- sequence with decreasing priority:

```
OFF > FATAL > ERROR >WARN > INFO > DEBUG
> TRACE > ALL
```

- levels are defined as constants in [org.apache.log4j.Level]

Log Levels (ctd)

- semantics:
  - `OFF` - all off, `ALL` - all on
  - `FATAL` - before JVM exits with error
  - `ERROR` - application error
  - `WARN` - critical condition
  - `INFO` - app info
  - `DEBUG`, `TRACE` - for debugging
- in `Logger`, there are method for each level (`warn()`, `debug()`, ..)
- these method are **overloaded**, e.g.:
  - `warn(Object)` - logs a message (usually a string)
  - `warn(Object,Throwable)` - logs a message and a stack trace of throwable (exception)

Appenders

- **appenders** define what happens to the logs
- logs can be written to multiple appenders
- appenders are configured per logger, different loggers can have different appenders
- appenders are inherited from parent loggers

Adding an Appender

```
BasicConfigurator.configure();
Logger logger = Logger.getLogger("Foo");

logger.addAppender(
    new org.apache.log4j.FileAppender(
        new org.apache.log4j.TTCCLayout(), "logs.txt"
    )
);
logger.debug("Hello World");
logger.warn("it's me");
```

add a second appender

```
0 [main] DEBUG Foo  - Hello World
1 [main] WARN Foo  - it's me
```

logs.txt

now logs are added to the console and to a log file

Selected log4j Appenders

appenders	description
`org.apache.log4j.ConsoleAppender`	write to the console (System.out or System.err)
`org.apache.log4j.FileAppender`	writes logs to a file
`org.apache.log4j.DailyRollingFileAppender`	write to files that are frequently rolled over to avoid the creation of log files that are too large
`org.apache.log4j.jdbc.JDBCAppender`	write logs to a (relational) database
`org.apache.log4j.net.SocketAppender`	write logs to a network
`org.apache.log4j.AsyncAppender`	buffers logs, and then writes them to other appenders - this is a "wrapper"
`org.apache.log4j.nt.NTEventLogAppender`	logs to the windows OS logging system (on windows only)

Layouts

- appenders use **layouts** to format log events
- information that can be displayed: event count, timestamp, thread, message, level, logger
- layout examples: formatted strings, xml, html

Using Layouts

```
.....
FileAppender appender = new FileAppender();
....
// configure the appender here, with file location, etc
        appender.setFile("trace.html");
        appender.setName("trace");
        appender.setLayout(new org.apache.log4j.HTMLLayout());
        appender.setAppend(false);
        appender.activateOptions();
        logger.addAppender(appender);
        logger.addAppender.warn("Error in page??!!!");
```

Access file appender

change layout to HTML

```
<tr>
<th>Time</th>
<th>Thread</th>
<th>Level</th>
<th>Category</th>
<th>Message</th>
</tr>
```

first log, uses default layout

second log, uses layout that formats log events as an HTML table row

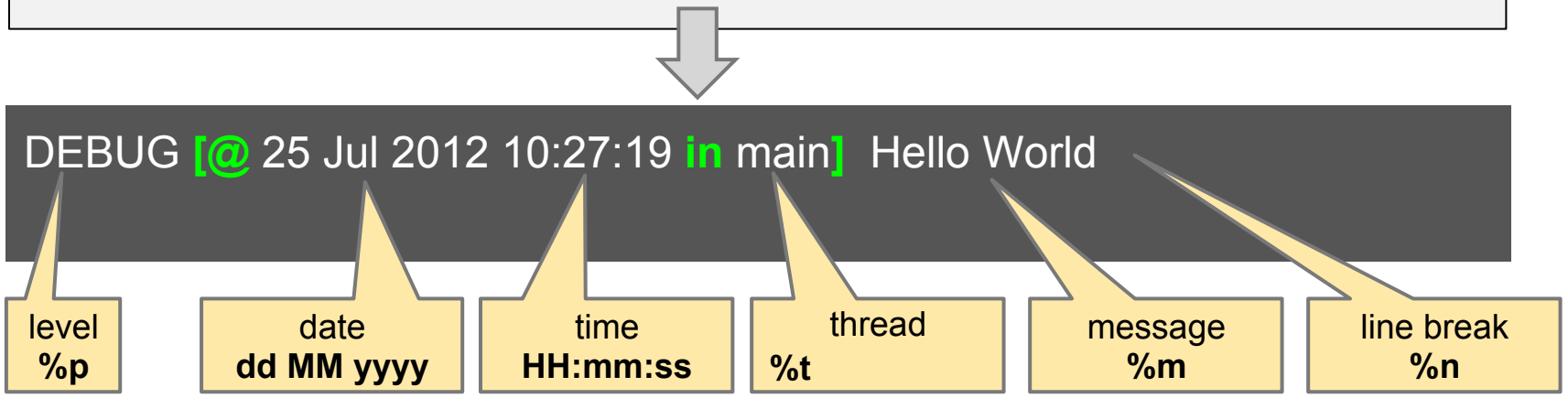
Layout Patterns

- it is difficult to support the composition of complex patterns
- this leads to either many classes, or many parameters (properties) in classes and it is hard to predict the outcome (i.e., the strings generated)
- a better way is to use a template or pattern: a string that defines the structure of the outputs
- variables like `%t` are used that are then instantiated (bound) when a log event is printed
- often, patterns are transformed in an object representation that facilitates binding (aka pattern compilation)
- this is an example of a little language or domain specific language

Using Layouts - Patterns

pattern defines how a log string is generated

```
..
String pattern =
    "%p [@ %d{dd MMM yyyy HH:mm:ss} in %t] %m%n";
Layout layout =
    new org.apache.log4j.PatternLayout(pattern);
appender.setLayout(layout);
logger.debug("Hello World");
```



green: constants, these symbols have no special meaning in the pattern language

Configurators

- **configurators** are used to set up log4j
- define loggers, levels, appenders, layouts
- `BasicConfigurator` - set defaults, log to console
- `PropertyConfigurator` - read configuration from property file (keys-values)
- more configurators exist

Using PropertyConfigurator

```
PropertyConfigurator.configure("log4j.config");
Logger logger = Logger.getLogger("Foo");
logger.debug("Hello World");
logger.warn("it's me");
```

read and apply configuration

```
log4j.rootLogger = DEBUG, ROOT

#set the root appender to be a console appender
log4j.appender.ROOT=org.apache.log4j.ConsoleAppender

#set the layout for the ROOT appender
log4j.appender.ROOT.layout=org.apache.log4j.PatternLayout
log4j.appender.ROOT.layout.conversionPattern=%p [%t] -  %m%n
```

create logger, set level and name

set appender

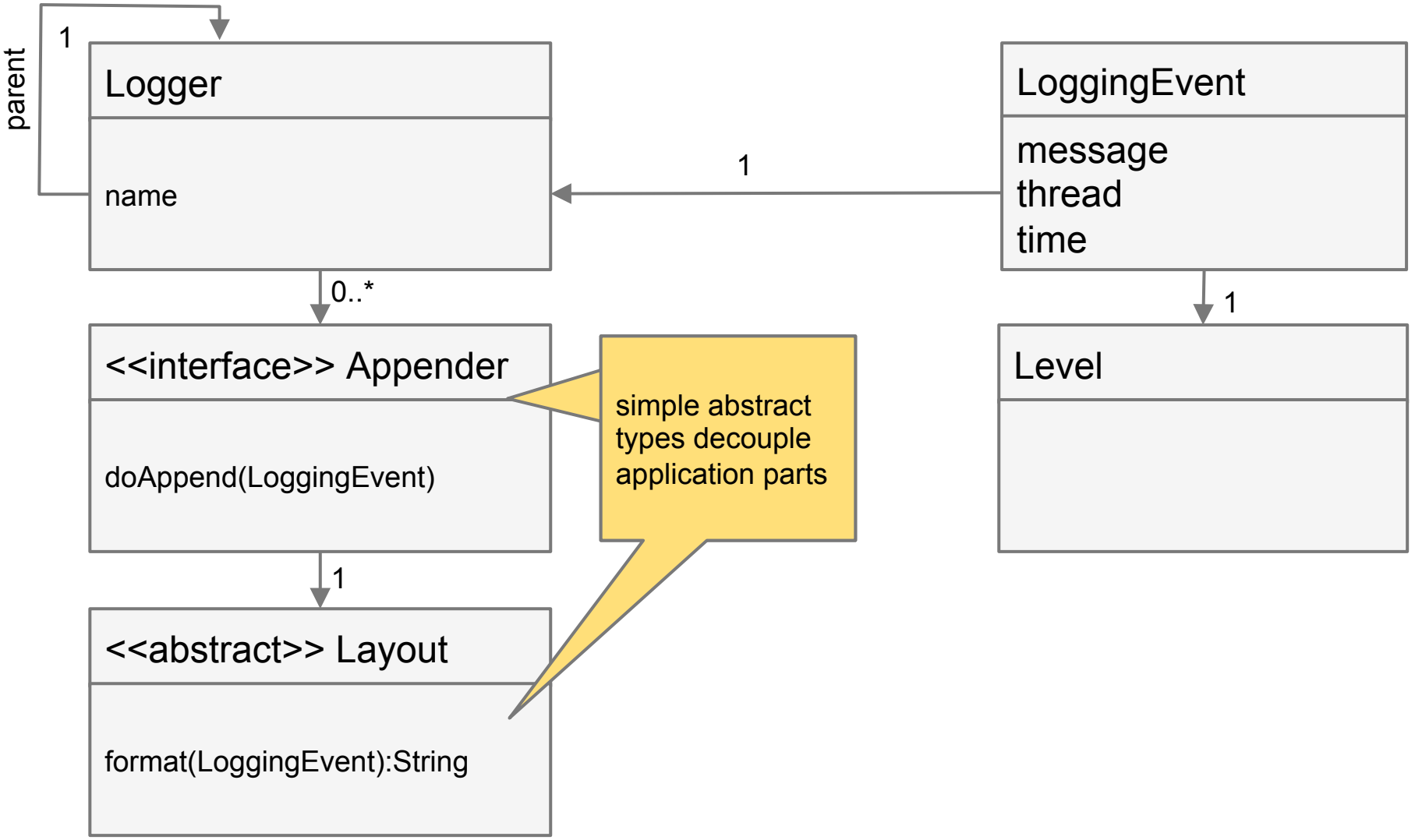
set layout

log4j.config

Designing for Orthogonality

- the log4j design aims at separating loggers, levels, appenders and layouts
- one aspect can change, without interfering with others
- the key is a design that separates (**decouples**) the several aspects of logging
- this is done through **abstract types** (abstract classes and interfaces)
- these types have simple interfaces, and are strictly separated from implementation classes
- these abstract types and their methods form the **Application Programming Interface (API)** of log4j

Log4J Design (Simplified)



Dependency Analysis

- lets try to quantify this: **interfaces should be stable**
- stability of an artefact is required if many other artefacts (classes, types) depend on it
- this implies that change can be propagated to dependents and cause "ripple effects"
- example: imaging the impact of changing `size()` in `java.util.Collection` (and therefore in all sets and lists) to `getSize()` - almost all existing Java code would have to be changed!

Sidetrack: Deprecating APIs

- problem: changing an API while trying not to break existing solutions depending on it
- practical solution: keep old API as well
- mark old API (methods and classes) as deprecated:
  - use the `@deprecated` annotation
  - use the `@deprecated` tag in the comments, and explain why this API is deprecated, and what should be used instead
- the compiler will produce warnings if deprecated APIs are encountered
- popular: see deprecated API list in Java 1.6

Deprecated Example

```
745     /**
746      * Forces the thread to stop execu
...
792     @deprecated This method is inherently unsafe.  Stopping a thread with
793     *       Thread.stop causes it to unlock all of the monitors that it
794     *       has locked (as a natural consequence of the unchecked
795     *       <code>ThreadDeath</code> exception propagating up the stack).  If
796     *       any of the objects previously protected by these monitors were in
797     *       an inconsistent state, the damaged objects become visible to
798     *       other threads, potentially resulting in arbitrary behavior.  Many
799     *       uses of <code>stop</code> should be replaced by code that simply
800     *       modifies some variable to indicate that the target thread should
...
810      */
811     @Deprecated
812     public final void stop() {
813                 stop(new ThreadDeath());
814     }
```

explain to the user why this has been deprecated

use annotation to tell compiler to produce warning

http://goo.gl/BiQIW

Deprecated APIs - Semantics

A program element annotated @Deprecated is one that programmers are discouraged from using, typically because it is dangerous, or because a better alternative exists. Compilers warn when a deprecated program element is used or overridden in non-deprecated code.

http://docs.oracle.com/javase/7/docs/api/java/lang/Deprecated.html

Valid reasons to deprecate an API include:

- It is insecure, buggy, or highly inefficient
- It is going away in a future release
- It encourages bad coding practices

http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/deprecation/deprecation.html

Deprecated APIs: Examples (from Java 1.6)

java.util.Date.parse(String)

As of JDK version 1.1, replaced by DateFormat.parse(String s).

improve expressiveness: parsing dates is complex and depends on local settings and use preferences - a separate class to manage data formats became necessary

java.awt.Window.show()

As of JDK version 1.5, replaced by Window.setVisible(boolean).

add standard compliance: revamp API to comply to Java Beans standards (use getters/setters to read/ write property visible)

java.lang.Thread.stop()

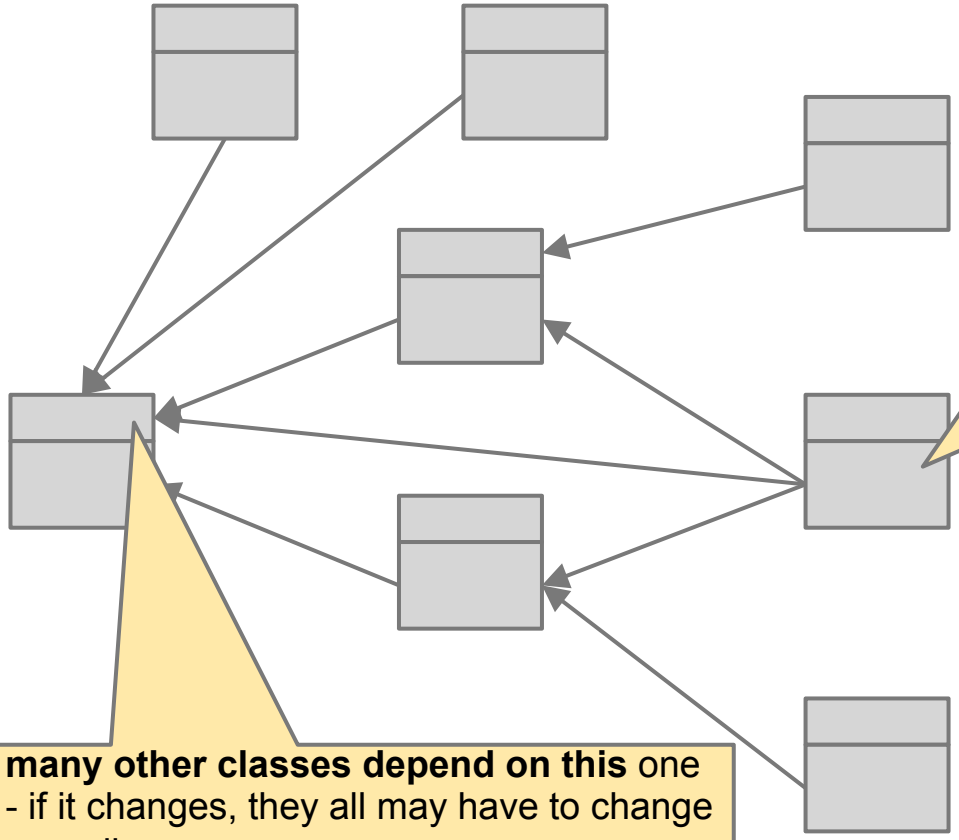
This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack).

bugfix: old concurrency API can lead to deadlocks and other concurrency problems

Measuring Stability

- count incoming dependencies of artefacts (methods, classes, packages)
- many incoming dependencies: should be more stable (change less often)
- more stability if we have more abstractness: declarations change less often than implementations

Measuring Stability (ctd)



this class **depends on many other classes** - if they change, this class may have to change as well - it is therefore **unstable**
This is known as Efferent coupling (Ce) or Import Coupling (IC)

many other classes depend on this one - if it changes, they all may have to change as well.
it should therefore be **abstract**
This is known as Afferent coupling (Ca) or Export Coupling (EC)

Note that the same approach can be applied at package level as well

Tooling: JDepend

- JDepend is a tool that measures dependency metrics for Java packages
- based on set of metrics developed by Robert Martin in OO Design Quality Metrics - An Analysis of Dependencies [alternative link]
- JDepend analyses dependencies between packages:
  - package 1 depends on package 2 if any type (class) in package 1 depends on any type in package 2
  - type 1 depends on type 2 if type 1 cannot be compiled when type 2 is removed (simplified, this includes the use of type 2 as supertype, field type, method return types etc in type 1)

Tooling: JDepend (ctd)

- JDepend is free: https://github.com/clarkware/jdepend
- integrations:
  - command line
  - Eclipse plugin
  - ANT integration

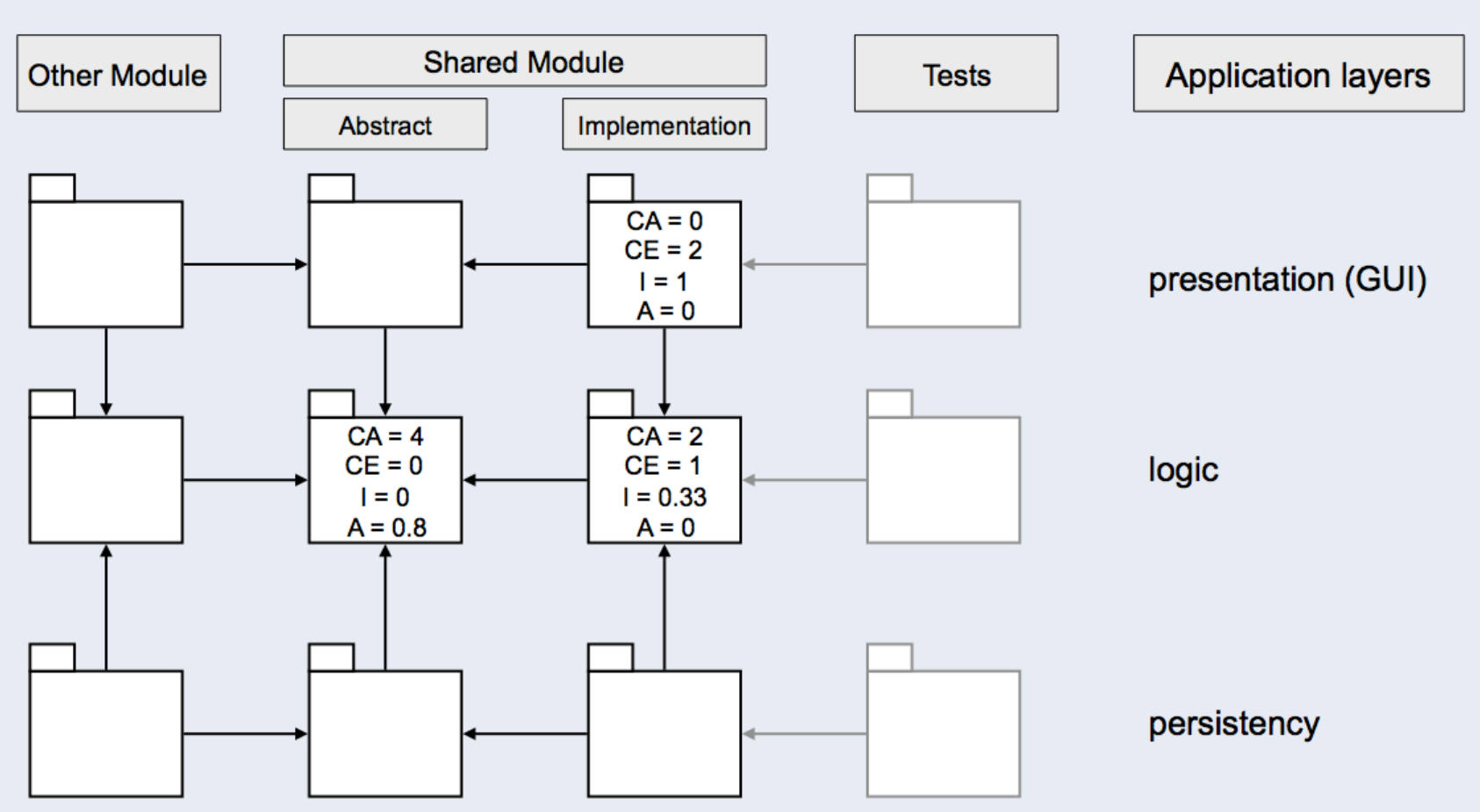
Tooling: JDepend (ctd)

- **Afferent Couplings** (Ca) - how many other packages depend upon a package, an indicator of the package's **responsibility**
- **Efferent Couplings** (Ce) - on how many other packages a package depends, an indicator of the package's **independence**
- **Abstractness** (A)-
  interfaces+abstract classes / #classes relative
  number of abstract type in package- between 0 and 1
- **instability** (I) = CE / (CE + CA) - an indicator of the package's resilience to change
- instability is high if there are relatively many dependencies on other packages – this package is more likely to change!

JDepend (ctd)

- abstract packages should be more stable
- implementation packages are generally more unstable
- good packages should have a good balance between stability and abstractness
- i.e., A+I should be close to 1
- purely abstract packages - instability should be 0
- pure implementation packages - instability should be 1
- (coarse) metric to assess this: measure distance to main sequence (D) - difference from A+I to 1
- D should be low!

JDepend Example



JDepend Example: log4j

org.apache.log4j.spi: "contains part of the System Programming Interface (SPI) needed to extend log4j"

- CA=13, CE=3, I=0.18, A=0.58
- many packages depend on this package, but this package only depends on a few other packages
- this means low instability
- many abstract types in package
- I+A=0.76 close to 1 (D = 0.24)
- **good example of a (mainly) specification package**

JDepend Example: log4j

org.apache.log4j.jdbc: "The JDBCAppender provides sending log events to a database.."

- CA=0, CE=2, I=1, A=0
- depends on a few other packages, no package depends on it
- maximum instability
- no abstract types
- I+A=1 perfect! (D = 0)
- **good example of an implementation (only) package**

Achieving Orthogonality: Coding

- use encapsulation features of the language
- avoid **god objects** and classes
- use abstract types where possible
- advanced techniques:
  - dependency injection
  - *service locator*
  - *aspect-oriented programming*

God Objects/Class

- this is an example of an **antipattern** (a bad structure in code/design)
- Also known as Brain Class or Super Class
- aka monster object
- a god objects knows too much, and is responsible for too many things
- it contradicts the "*divide and conquer*" approach of OO - a god objects is a central hub
- it needs to change, and causes change often
- often one can recognise a god class (instantiated by a god object) through naming patterns used: Manager, Admin, Util, ..

JDeodornt

- Just like a *deodorant* !!
- JDeodorant finds smells in your code and refactor them
- Eclipse plugin : **Help-> Eclipse Marketplace** then search for **JDeodorant** and **Install**
-
- More on “code smells” later in the course!

Achieving Orthogonality: Design

- follow OO design principles
- layered architecture
- separate specification from implementation artefacts (classes vs interfaces, abstract and implementation packages)
- group related artefacts together
- actively manage dependencies (document and test them!)
- use a component model

Principles of OO Design

- set of 11 principles representing widely accepted best practices
- online: http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign
- contains Liskovs Substitution Principle (already discussed)

The Dependency Inversion Principle (DIP)

1. High level modules should not depend upon low level modules. Both should depend upon abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

http://www.objectmentor.com/resources/articles/dip.pdf

controls dependencies by constraining their **direction**

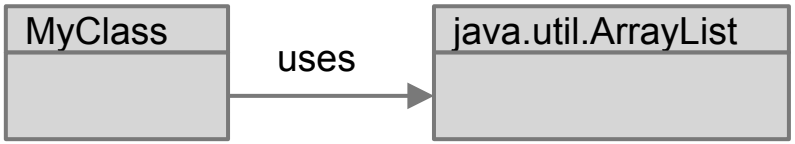
The Interface Segregation Principle (ISP)

> The dependency of one class to another one should depend on the smallest possible interface.

http://c2.com/cgi/wiki?InterfaceSegregationPrinciple

controls dependencies by constraining their **target**

ISP Example



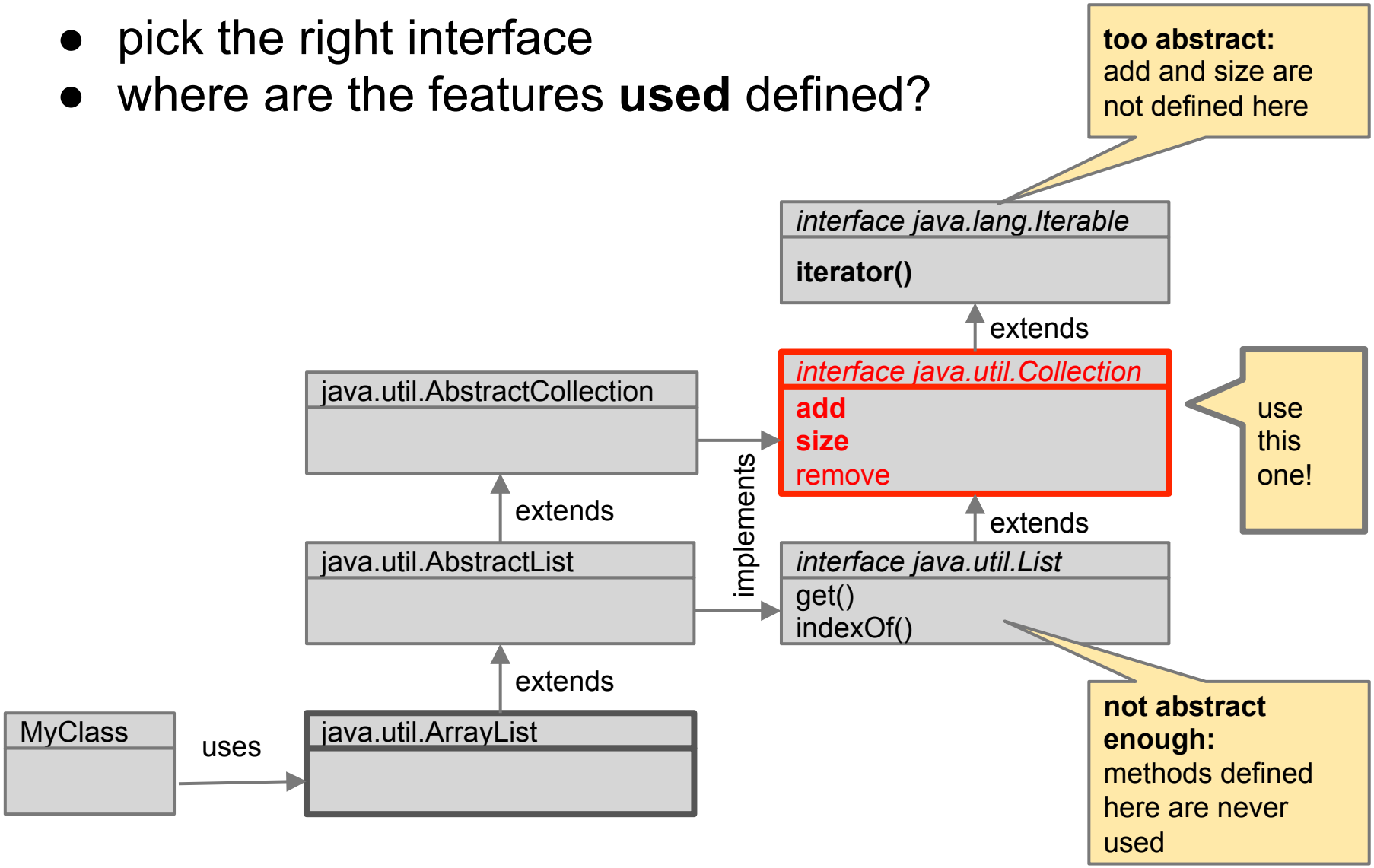

```
MyClass {
        ArrayList list = new ArrayList();
    ...
    public void foo() {
        list.add("object1");
        list.add("object2");
        System.out.println(list.size());
        for (Object e:list) {
                ..
        }
    }
    ...
}
```

use **add** and **size** methods on list

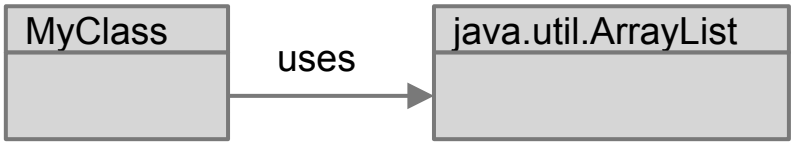
this (implicitly) uses another method on list: iterator(). The compact iteration is actually a while loop over the iterator returned by iterator() !

ISP Example (ctd)

- pick the right interface
- where are the features **used** defined?



ISP Example (ctd)




```
MyClass {
        Collection list = new ArrayList();
    ...
    public void foo() {
...
```

- the Collection interface should be used for collaboration between MyClass and ArrayList
- it defines **all methods needed** (referenced), and is the **most abstract type** doing so
- advantage: we can easily switch to a *linked list*, *hash set* or *tree set* when needed

ISP Example (ctd)

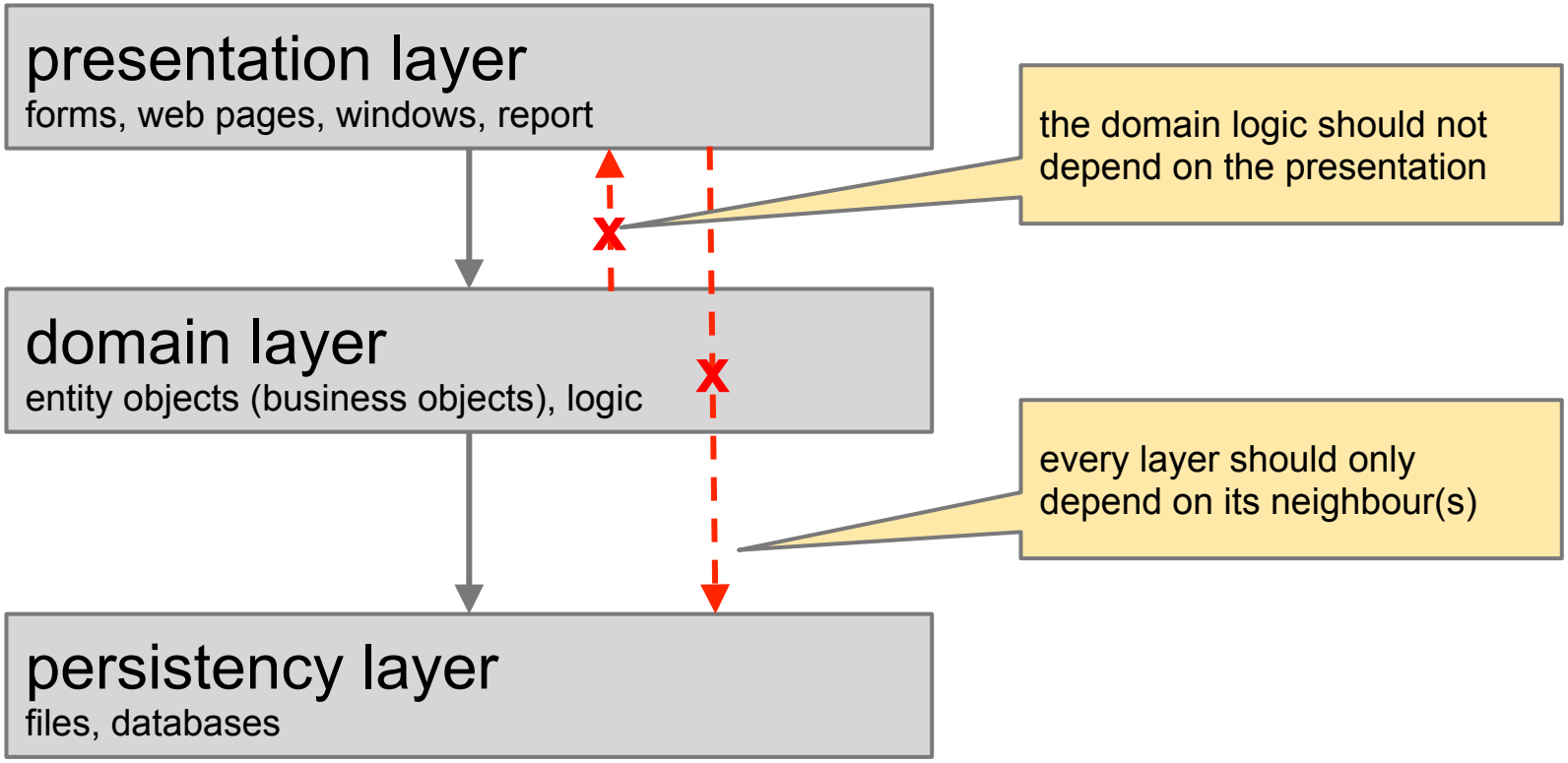
```
MyClass {
        Collection list = new ArrayList();
    ...
    public void foo() {
...
```

- this still leaves us with a dependency to **`ArrayList`** from the object instantiation
- there are different strategies to avoid this dependencies:
  - initialise field as null, and set it via a setter or constructor parameter (aka dependency injection)
  - use a factory
- Both mechanism to be discussed later

Layered Architecture

- organise the application is layers (aka tiers)
- each layer only knows its neighbours
- classical example: 3-layer architecture
- successful application of layered architecture: the OSI model defining network layers

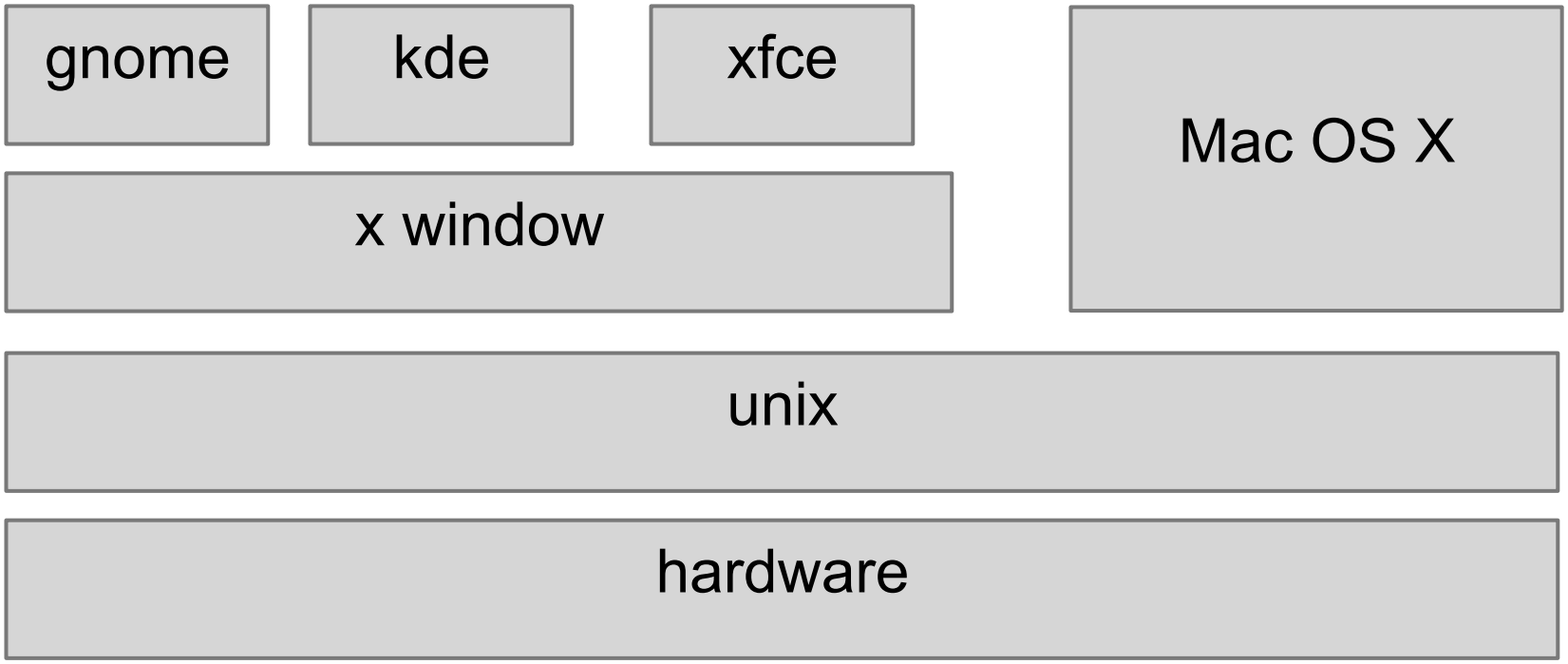
The 3-Tier Architecture



The 3-Tier Architecture Benefits

- easy to replace presentation layer, e.g. migrate desktop application to web application
- easy to replace persistence layer, e.g. migrate from file based persistence to more scalable database-based persistence
- standard protocols and modules can be used for certain layers, such as:
  - HTTP/HTML for communication with web-based presentation layer
  - SQL over TCP/IP for communication with database-based persistence layer
- frameworks exist to ease communication between layers: "middleware" such as object-relational mapping frameworks (ORM)

Layer Architecture in Unix/Linux



flexibility because core OS does not depend on the windowing system!

Wrong Dependencies

```
public class Student {
       private String id = null;
       private String name = null;
       private Date dob = null;
   public boolean equals(Object obj) {..}
       ..
    public void edit () {
       Form form = new Form(this);
       form.display();
    }
}
```

- here the **domain layer** depends on the **presentation layer**!
- **event driven programming** can be used to break this kind of dependency
- often this design flaws are caused by user-interface centric tools (such as VisualBasic or Delphi)

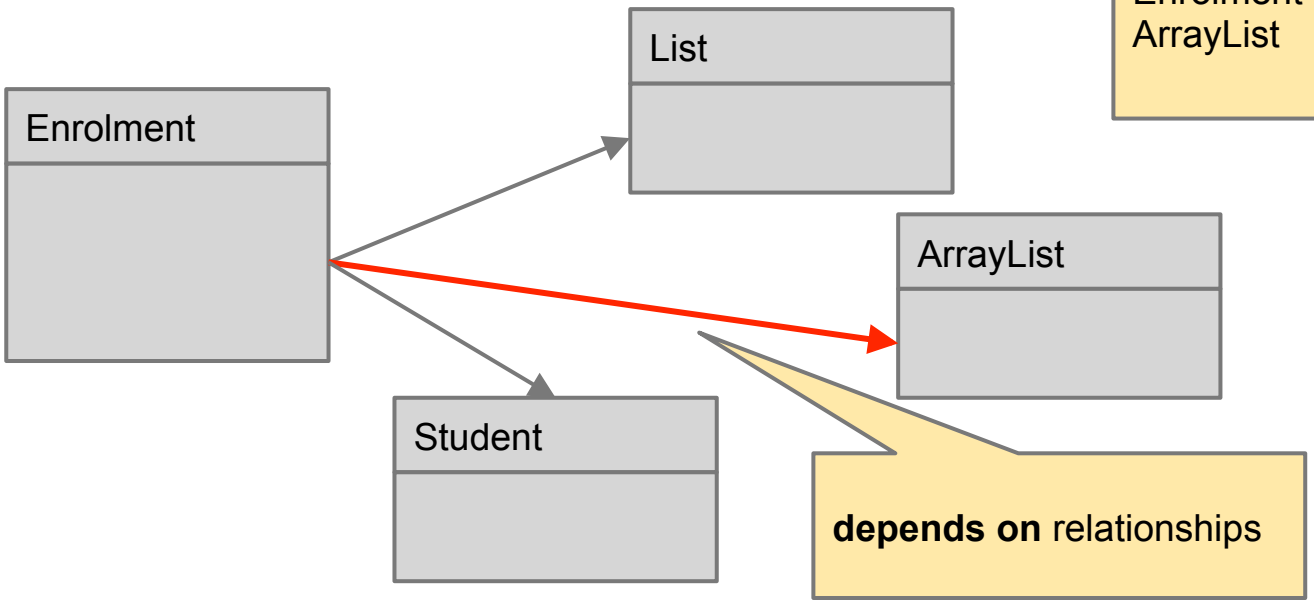
Dependency Injection

- in log4j, an appender does not have a default layout
- they only have a `setLayout` method to set (inject) a layout
- this is done in client code, or during initialisation using a configurator (e.g., the information is read from a config file)
- this technique is called **dependency injection**
- see also:
  Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html
  Read here as well https://en.wikipedia.org/wiki/Dependency_injection
- there are specialised dependency injection framework to facilitate this, such as Spring and Google Guice

Dependency Injection Example

```
public class Enrolment {
    private List<Student> students = new ArrayList<Student>();
    public List<Student> getStudents() {
        return students;
    }
    ...
}
```

Enrolment depends on ArrayList



Dependency Injection Example

```
public class Enrolment {
    private List<Student> students = null;
    public List<Student> getStudents() {
        return students;
    }
    public void setStudents(List<Student> list) {
        this.students = list;
    }
    ...
}
```

Dependency to ArrayList has been removed

setter can be used to **inject** list when program is running

Dependency Injection (ctd)

- this is sometimes also called the **Hollywood Principle**: "don't call us, we will call you."
- the list example seems trivial, but consider the case where:
  - the abstract type is `java.sql.Connection` (a database connection)
  - the concrete types is `MySQLConnection` (to connect to a database of a particular vendor)

Service Locators

- service locators are an alternative to dependency injection
- objects are accessed through interfaces, and the actual creation of types is hidden
- when an object (service) is requested, the service locator either creates one (acts as a factory) or returns an existing one (acts as a registry)

Service Locator Example - JDBC

```
import java.sql.*;

public class MySQLAccess {
  private Connection connect = null;
  private Statement statement = null;
  private PreparedStatement preparedStatement = null;
  private ResultSet resultSet = null;

  public void query(String dbURL,String sql) throws Exception {
    try {
      Class.forName("com.mysql.jdbc.Driver");
      Connection connect = DriverManager.getConnection(dbURL);
      Statement statement = connect.createStatement();

     ResultSet rs = statement.executeQuery(sql);
```

Connection, *Statement* and *ResultSet* are all abstract types, the mysql specific implementation typesare hidden!

Inside JDBC

- the initial reference to a connection is obtained from the `DriverManager` for a database address
- the driver manager has an internal registry of database-specific drivers, and will go through all drivers and ask them whether they can create a connection for this database address
- if the URL starts with "`jdbc:mysql:`", the JDBC driver will respond and create a connection
- the `DriverManager` will return this connection, but the application will not be aware what connection it is

Inside JDBC: Bootstrapping

```
import java.sql.*;

public class MySQLAccess {
  private Connection connect = null;
  private Statement statement = null;
  private PreparedStatement preparedStatement = null;
  private ResultSet resultSet = null;

  public void query(String dbURL,String sql) throws Exception {
    try {
      Class.forName("com.mysql.jdbc.Driver");
      Connection connect = DriverManager.getConnection(dbURL);
      Statement statement = connect.createStatement();

      ResultSet= statement.executeQuery(sql);
```

Magic line of reflection code !!

Inside JDBC (ctd)

- the `Class.forName()` statement is used to load the driver class
- this will trigger the registration of the driver: in the static block of the driver class (aka class constructor), `DriverManager.registerDriver(<newinstance>)` is called
- from JDBC 4.0, there is a more elegant solution using the the built-in `ServiceLoader`

Inside JDBC (ctd)

- the driver is distributed in a library (such as mysql.jar), and the metadata of this library contains a file `META-INF/services/java.sql.Driver` with a list of implementation classes for this interface
- the service loader class can then be used by the driver manager to access and instantiate these classes
- this means that the (slightly obscure) `Class.forName()` statement is not longer necessary for newer drivers, and driver registration is more “plugin like”