TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

SWEN 301 : Scalable Software Development

Software Architecture

Jens Dietrich **(jens.dietrich@vuw.ac.nz)**

Overview

- what is architecture ?
- views and stakeholders
- 4+1 model
- component diagrams
- architectural patterns: layered, plugin/microkernel, service-oriented
- architecture recovery

What is Architecture ?

- software architecture deals with the design and implementation of the **high-level structure** of the software*
- it is a **model**: an abstraction (simplification) of the actual system
- it may have multiple abstractions, corresponding to different aspects and stakeholders: **views**
- it is usually there **before** a system is built, but can also be reverse engineered
- it deals with **decomposition and composition** of a system into **parts**: tackling complexity with **divide and conquer**
- it uses styles / patterns

* Kruchten, Philippe B. "The 4+ 1 view model of architecture." IEEE software 12.6 (1995): 42-50.

Parts

- on an abstract level, a part is a model element
- its granularity is undefined, but for an architecture to be a meaningful abstraction, it should be rather course
- when a system is implemented, a part may correspond to objects, services deployed on dedicated servers, packages, module or components, or large coherent subsystems

Architecture as Model

- abstraction = certain aspects are ignored
- example: how a part of the system is implemented: particular programming languages, libraries, protocols to be used may not be part of the architecture
- trade offs:
  - architecture guides construction, and constraints it
  - it is possible to reason about (expressive) architecture in order to predict system properties
  - architecture facilitates communication, the level of abstraction must be suitable for the various stakeholders

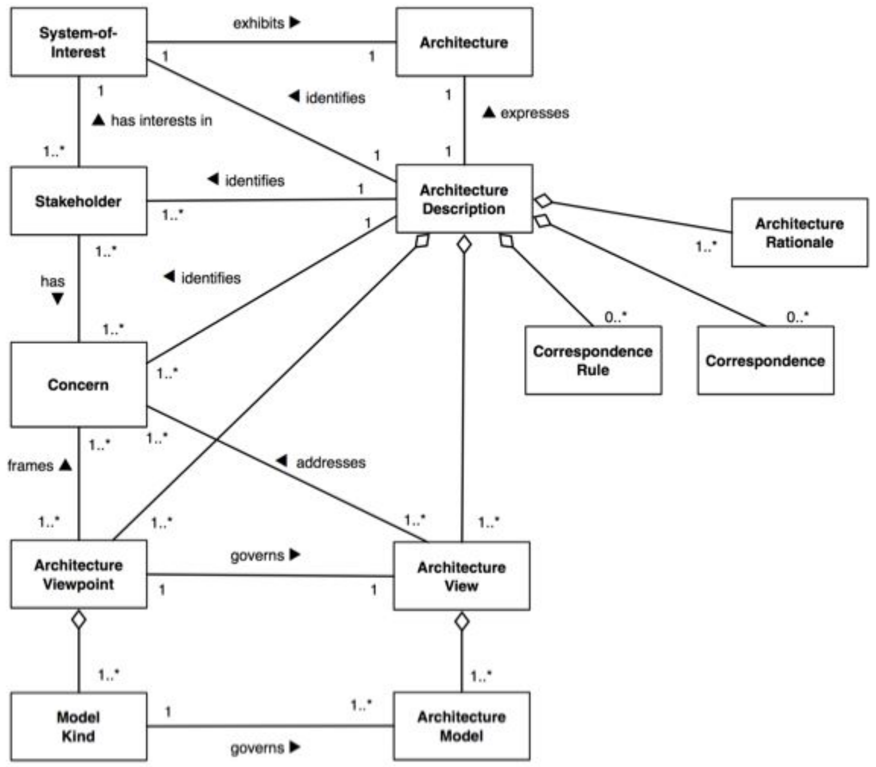
Describing Architectures

- explicite architecture description languages (ADLs) have been developed
- UML contains several models suitable to describe architecture

ISO/IEC/IEEE 42010

- standardises terms to describe architecture
- provides “meta model” for architecture
- very high-level, standard is paywalled

ISO/IEC/IEEE 42010



The 4+1 Model

proposed by Philippe Kruchten at Rational in 95 (coinciding with UML and RUP)

four **core views** on architecture

- **logical** -- corresponds to end user view, using end user terminology, can be expressed using a UML class diagram
- **process** -- captures the concurrency and synchronization aspects of the design, can be expressed using UML activity diagram
- **physical** -- which describes the mapping(s) of the software onto the hardware (deployment), can be expressed using UML deployment diagrams
- **development** -- the static organization of the software, can be expressed by UML component diagrams

Scenarios

- 5th element of the 4+1 model
- glues together other views
- describes sequences of interactions between objects and processes
- can be used to identify elements, to illustrate and validate the architecture

The 4+1 Model

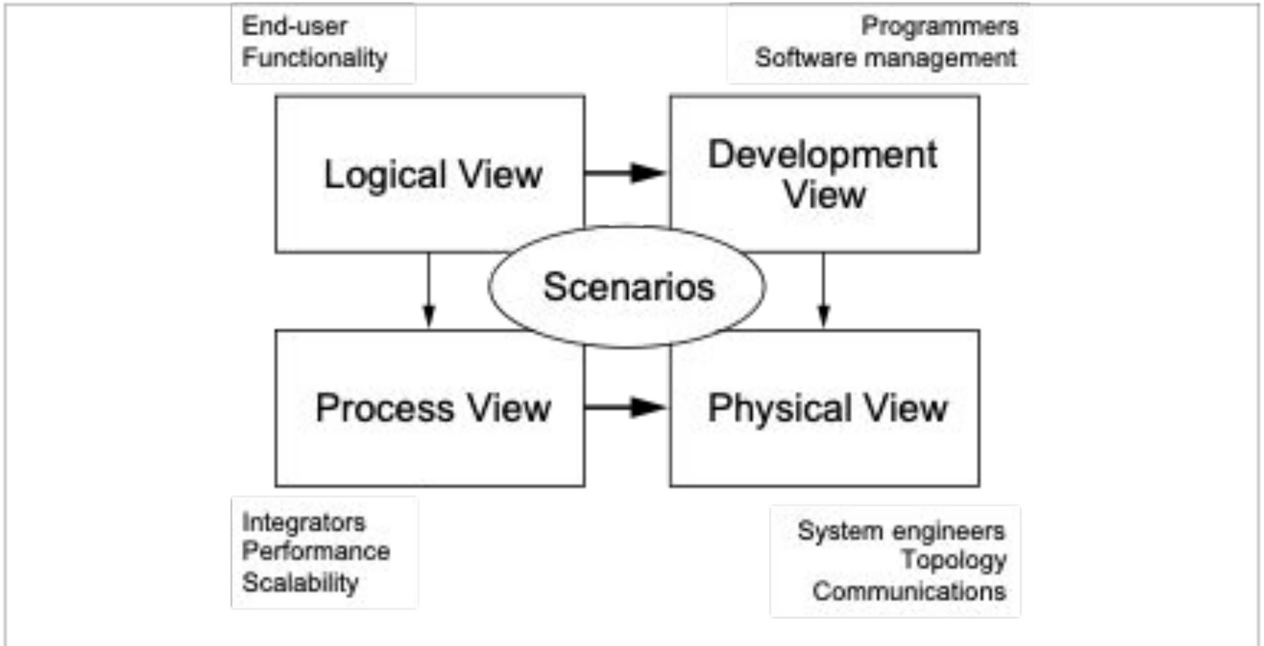


Figure 1 — The "4+1" view model

Kruchten, Philippe B. "The 4+ 1 view model of architecture." *IEEE software* 12.6 (1995): 42-50.

Models

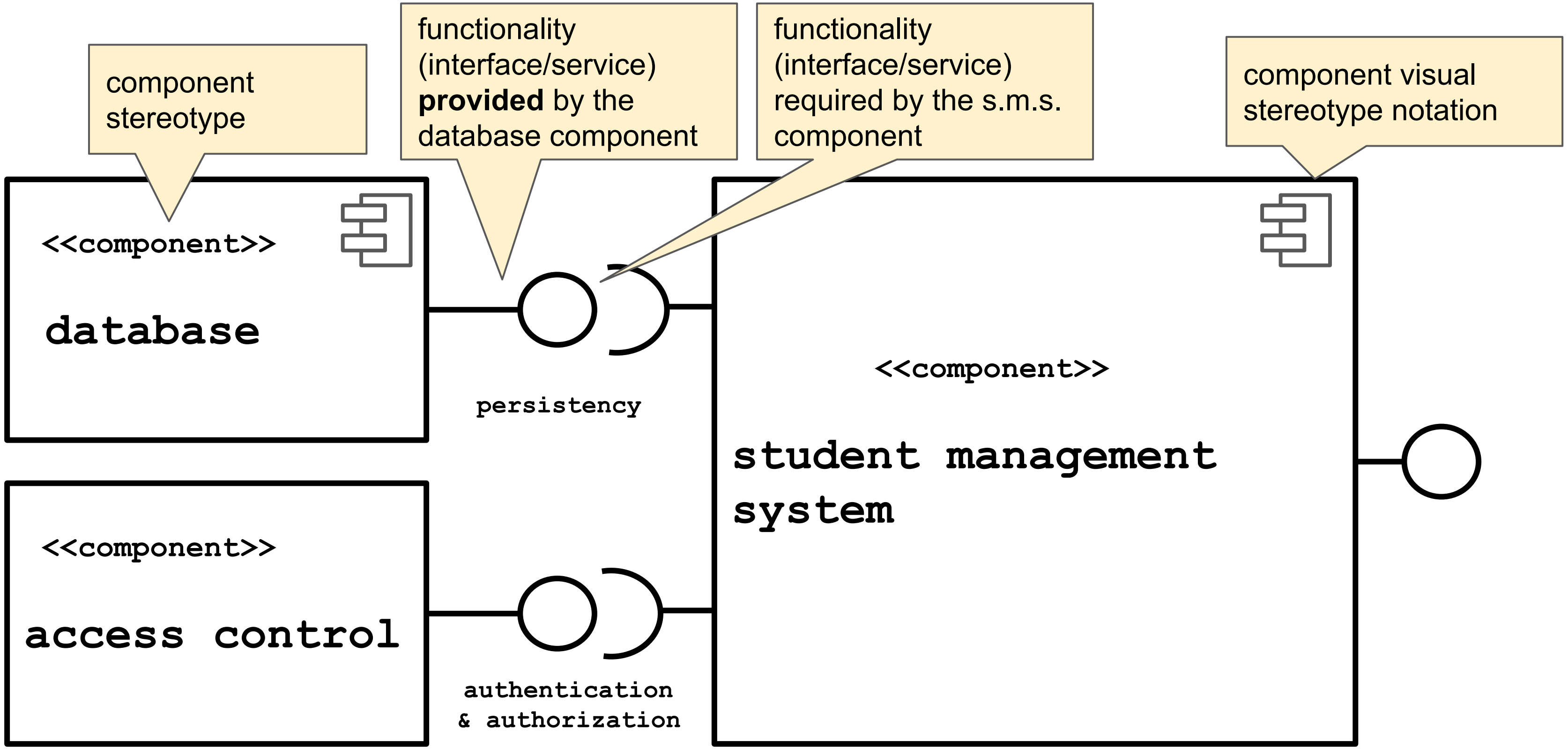
- note that models are not always separated
- consider a university management system, consisting of
  - student management system (SMS), a course management system (CMS), and a room booking system (RBS)
  - these systems may form coherent parts in different models: they are logical components (logical), the respective systems are deployed on different servers (physical) and form code modules maintained in different repositories (development)
- in this case, fewer (one) model can be used with additional annotations (e.g. UML stereotypes) representing aspects of the other views
- *"The "4+1" view model is rather "generic": other notations and tools can be used, other design methods can be used, especially for the logical and process decompositions, but we have indicated the ones we have used with success."* P Kruchten

UML Component Diagrams

- high-level description of how systems are composed from parts
- parts are called **components**: they provide and require **services**
- *“a component represents a **modular part of a system** that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. ”* *

* [UML Spec 2.2] OMG Unified Modeling Language^{TM} (OMG UML), Superstructure Version 2.2

Required and Provided Interfaces

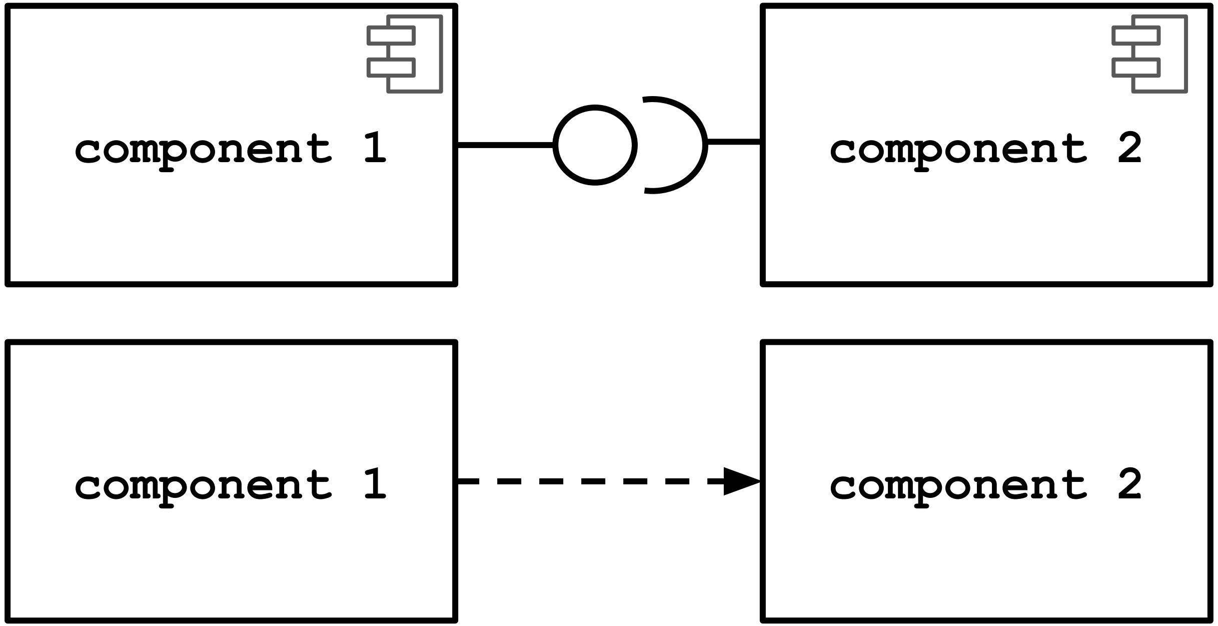

component stereotype
functionality (interface/service) **provided** by the database component
functionality (interface/service) required by the s.m.s. component
component visual stereotype notation
<<component>>
database
persistency
<<component>>
student management system
<<component>>
access control
authentication & authorization

Beyond the Basics

- components can be nested (see [UML Spec 2.2], fig 8.12)
- special connectors can be used between ports and lollipops (required and provided interfaces)(see [UML Spec 2.2], fig 8.14)
- component internals can be modelled using class diagrams (see [UML Spec 2.2], fig 8.14)
- there is a generic notion of component dependency (see [UML Spec 2.2], fig 8.14)

The Nature of Collaboration and Dependencies

What does this mean ?



The UML View

An interface declares a set of **public features and obligations** that constitute a **coherent service** .. An interface does not specify how it is to be implemented, but merely what needs to be supported by realizing instances.

[UML Spec 2.2] 7.3.24

The interface compatibility between provided and required ports that are connected enables an existing component in a system **to be replaced by one that (minimally) offers the same set of services**. ”

[UML Spec 2.2] 8.3.3

Example 1: Objects as Components

- Java objects as components
- Java interfaces **as services** / interfaces
- consistent with vision of early versions of Java: Java Beans component model, later Java Enterprise Beans (EJB)
- often too fine-grained for architectural modelling
- Example:
  - a student management system uses a **Set** component to manage students
  - a set implementation (like **HashSet**) is the component, it provides itself as a set service
  - substitutability (replaceability) means that it could be replaced by an instance of another **Set** implementation (like **TreeSet**)
- **`Set set = new HashSet();`**

Example 2: Service Loaders

- interfaces (or other abstract types) are services
- consumers and providers of services are libraries (= jars files containing compiled Java code)
- the Java service loader mechanism is used to connect consumers and providers

The Service Abstraction

```
package com.foo.services;
public interface StudentLookup {
   Student findStudent(String id);
}
```

The Provider Perspective (in provider.jar)

```
package com.foo.services.db;
import com.foo.services.StudentLookup;
public class StudentLookupImpl implements StudentLookup {
   Student findStudent(String id) {
      // lookup info in database,
      // create and return Student instance
   }
}
```

- include file **`META-INF/services/com.foo.services.StudentLookup`** in jar
- content is one line comprising **`com.foo.services.db.StudentLookupImpl`**

The Consumer Perspective (in myapplication.jar)

```
ServiceLoader<StudentLookup> loader =
    ServiceLoader.load(StudentLookup.class);
Iterator<StudentLookup> lookups = loader.iterator();
StudentLookup firstLookup = lookups.next();
```

- service loaders are an utility to collect service implementations from libraries in the classpath
- this avoids code dependencies in the consumer from concrete providers !
- this therefore facilitates strong encapsulation

Services in Action: JDBC

- JDBC = Java database connectivity
- standard to connect Java application to relational database
- provides a framework for standardised adapters, provided by database-specific drivers
- drivers handle:
  - mapping of generic JDBC / SQL to database-specific features
  - connection to DB (usually via a network protocol)
- drivers are managed by a driver manager

JDBC Usage Example

```
import java.sql.*;

Class.forName("com.mysql.jdbc.Driver");

String databaseUrl = “jdbc:mysql://localhost/EM”;
Connection conn = DriverManager.getConnection(databaseUrl);

Statement stmt = conn.createStatement();
String sql = "SELECT * FROM Students";
ResultSet rs = stmt.executeQuery(sql);

while(rs.next()){
     int id  = rs.getInt("id");
     ..
}

rs.close();
stmt.close();
conn.close();
```

load driver

connect to database

fire query

analyse row data for each row in query result

close result set

JDBC Usage Example Discussion

- core types `Connection`, `Statement`, `ResultSet` are all abstract (interfaces), concrete types are provided by driver but never directly used by application
- this is to avoid dependencies to particular implementations , and therefore **vendor lock-in**
- the abstract factory pattern is used to instantiate objects
- `Class.forName(..)` triggers loading the driver class and execution of the static block, this will register a driver instance with the `DriverManager`
- when the driver manager is asked to connect, it will ask registered drivers who can provide a connection for a database with this URL

JDBC 4 Service Loading - Drivers as Plugins

- in JDBC 4 , the obscure `Class.for(..)` call has been deprecated
- drivers now advertise **services** in the jar file metadata
- the mysql driver jar contains a file `meta-data/services/java.sql.Driver` with a single line "`com.mysql.cj.jdbc.Driver`" -- i.e. `com.mysql.cj.jdbc.Driver` is a concrete (insatiable) class implementing `java.sql.Driver`
- an application (such as the driver manager) can ask for available services in classpath components using methods in `java.util.ServiceLoader`
- this makes it possible to install drivers as "plugins" or rather "dropins"
- this pattern is called **service locator** (https://martinfowler.com/articles/injection.html#UsingAServiceLocator)

Example 3: OSGi

- OSGi component model: component (aka bundle) = jar + classloader
- components declare provided and consumed services as Java interfaces
- widely used in application servers, and Eclipse plugins
- similar: Eclipse plugins -- services are provides as extensions, extension points are ports to plugin-in required services
- interesting for several reasons:
  - implementation is very close to architectural principles
  - used to engineer very large systems (WebLogic, WebSphere)
- see https://github.com/jensdietrich/se-teaching/tree/main/osgi

OSGi Concepts

- bundles are packages as standard jar files with extended manifests
- bundles are executed within an OSGi container (Felix, Equinox, Knopflerfish)
- a bundle activator is a class that has lifecycle callbacks when the bundle is started or stopped by the container
- the activator can use the **BundleContext** to post new services it offers, remove services, and subscribe to services offered by other components
- services are identified by name, e.g. by the name of the Java interface used to define the service

OSGi Design Patterns

- OSGI uses a popular design pattern: observer
- when a component providing a service becomes available, other components consuming the service are notified by the OSGi framework
- they can then start using the service (for instance, using a DateFormatter to render `Date` instances)

OSGi - Defining a Service

A service is defined using a plain old Java interface, such as **`DateFormatter`**, with a sole method **`String format(java.util.Date date)`** .

OSGi - Providing a Service (Short Date Formatter)

```
public class Activator implements BundleActivator  {

    public void start(BundleContext bc) throws Exception {
        DateFormatter service = new ShortDateFormatter();
        bc.registerService(DateFormatter.class.getName(),service);
    }
```

- The bundle context is a reference to the framework
- The service definition is defined by the simple **`DateFormatter`** interface
- The actual service is the **`ShortDateFormatter`** implementation of **`DateFormatter`**
- **`start`** is called by the framework when the component is activated
- full code: https://github.com/jensdietrich/se-teaching/blob/main/osgi/dateformatter-provider-short/src/nz/ac/vuw/jenz/osgi/df_short/Activator.java

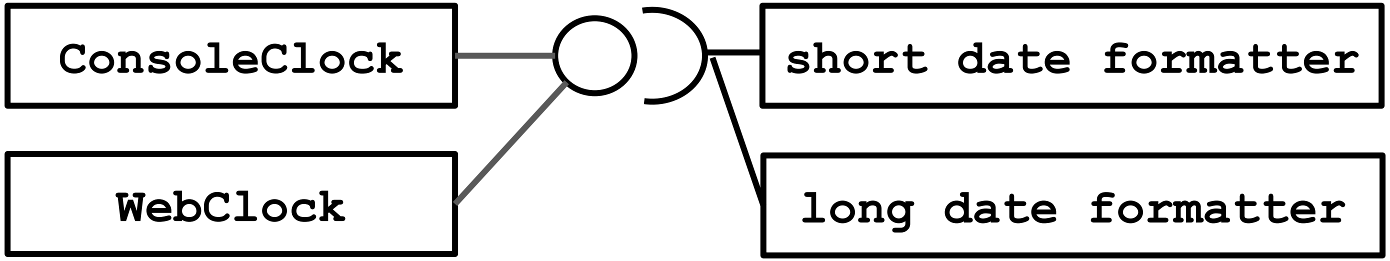
OSGi - Consuming a Service (ConsoleClock)

- when a consumer starts, it registers a **`ServiceListener`** with the framework
- whenever a service changes (becomes available or unavailable), the listener is modified
- this can then be used to update references to **`DataFormatters`** to be used
- full code: https://github.com/jensdietrich/se-teaching/blob/main/osgi/dateformatter-consumer-consoleclock/src/nz/ac/vuw/jenz/osgi/consoleclock/Activator.java

Many2Many Abstraction

- the service can be provided by multiple components: ShortDateFormatter / LongDateFormatter
- the service can be consumed by multiple components: ConsoleClock / WebClock
- WebClock will start an HTTP server, accessible with http://localhost:8080/date

Many2Many Abstraction



Issues & Extensions

- when multiple service providers are available, the consumer must make a choice
- example: put services in list, use top-most
- consumers have to be careful caching services as this interferes with GC !
- this could then prevent services to be removed or replaced (updated)
- possible solutions: respond to unregister events, use weak or soft references
- OSGI mechanism can be made more elegant by more declarative service layers, examples: Eclipse extensions, OSGi declarative services

Example 4: Web Services

- service is described by URLs, and properties of the URL transaction
- often, a particular description language is used for this purpose, such as SWAGGER
- this approach to system modelling is part of a service-oriented architecture
- the focus is shifting from component details to interface / service details (e.g. defined using OpenAPI)

`foo.com` provides an exchange rate service.
`GET foo.com/xrate?from=%1&to=%2` will return the following result:

1. **418 I'm a teapot** if %1 is “beans” and %2 is “coffee”
2. **404 Not Found** if rule 1 does not apply and %1 and %2 aren't currency symbols as defined in ISO 4217
3. **200 OK** otherwise, with a JSON - encoded response body with the following structure: ..

Example 4: Little Teapots ..

- status code 418 is an easter egg in HTTP
- https://www.google.com/teapot

Comparison of Approaches to Composition

Approach	**Language Agnostic ?**	**Composition**	**Service Abstraction**	**Locality**	**Runtime Recomposition / Upgrades**	**Platform**
Plain Old Java	Java and other Languages compiling into JVM byte code	Compiletime	Java abstract types	Same JVM	no	JVM
Service Loaders	Java and other Languages compiling into JVM byte code	Runtime	Java abstract types	Same JVM	no	JVM
OSGI	Java and other Languages compiling into JVM byte code	Runtime	Java abstract types	Same JVM	yes	OSGi container
Web Services	any	Runtime	API Specs (Swagger etc)	Web	yes	HTTP Server & Client

Architectural Patterns

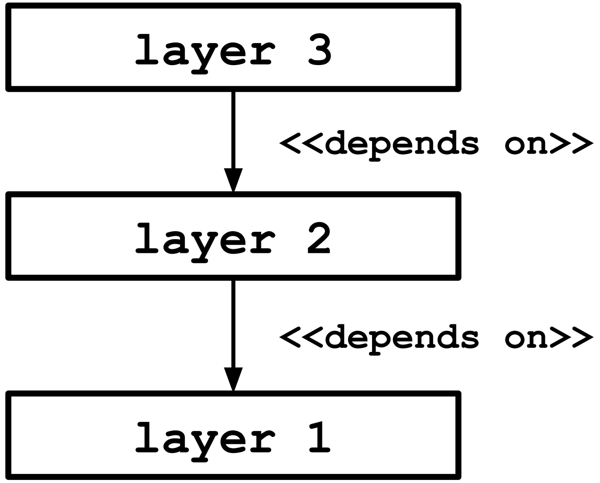
- pattern represent **re-usable best practice** to solve certain problems
- inspired by work on design patterns (GangOf4, ..)
- used to communicate design
- can be described using template (pattern language)

Arch. Pattern: Monolithic Architecture

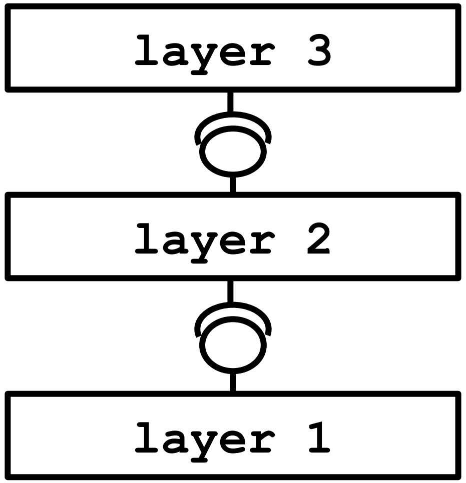
- often seen as antipattern
- lack of internal structure: no parts, or all parts are heavily interconnected and interdependent
- often design erodes into a monolithic architecture
- “The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. ” Big Balls of Mud (B. Foote, J. Yoder 99)

Arch. Pattern: Layered Architecture

- application is stratified into parts that have only linear, one-directional relationships

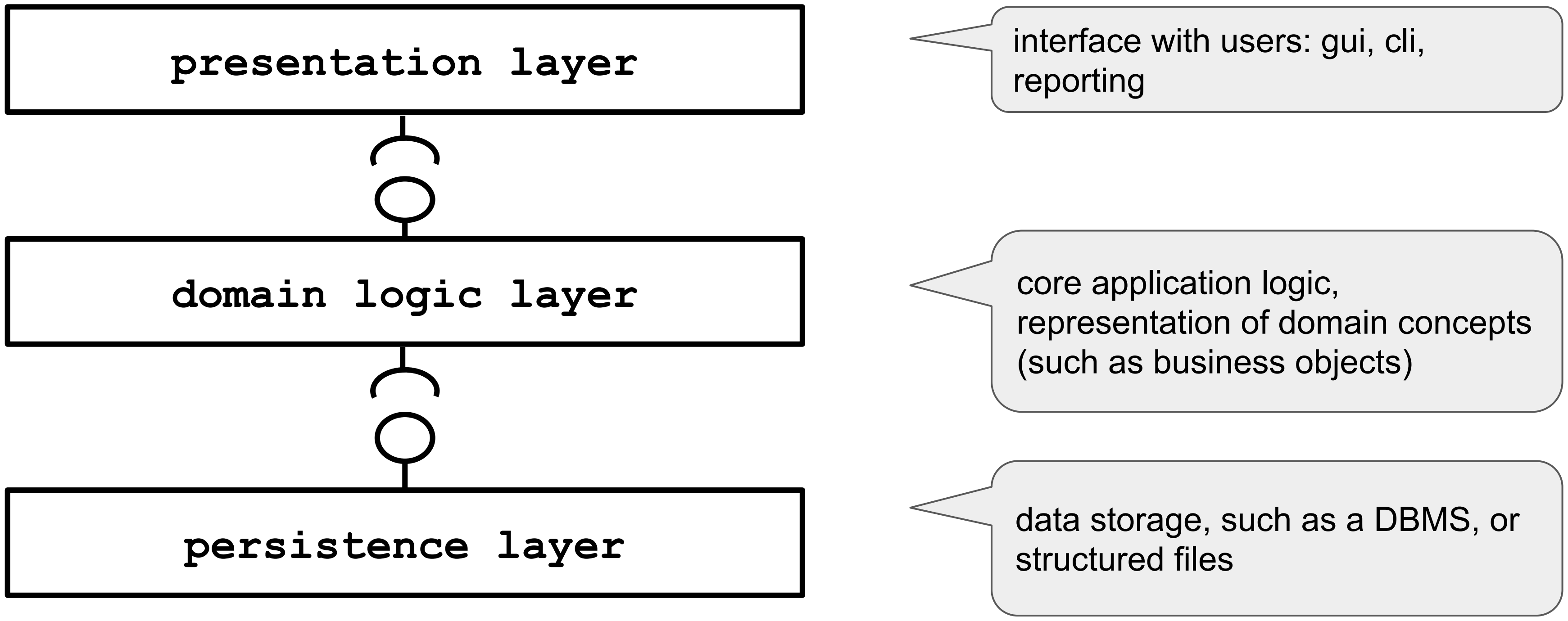


generic boxology



UML component diagram syntax

A Standard Three-Tier Architecture


presentation layer
domain logic layer
persistence layer
interface with users: gui, cli, reporting
core application logic, representation of domain concepts (such as business objects)
data storage, such as a DBMS, or structured files

Benefits: Swapping layers

- Linux provides many alternative presentation layers, such as KDE , Xfce and gnome, this is facilitated by another layer to define graphics primitives: X
- business applications often swap the persistence layer during their lifecycle, to move to an enterprise level DBMS in order to increase scalability
- common domain can support web-based and mobile applications (presentation layer)

Standardising Interfaces between Layers

- to take full advantage of a layered architecture, interfaces must be standardised and facilitate low coupling
- example 1: standardized JDBC + SQL interface between Java applications (domain logic) and RDBMS (persistence)
- example 2: modern applications often use a thin surface layer on top of the domain layer to expose functionality to different presentation layers, such as web uis and mobile apps, and reporting engines

Case Study: Provide Persistence via RDBMS

- use RDBMS to provide storage
- avoid vendor lock-in using standards, standard interfaces to upgrade components with minimal friction (JDBC + SQL92)
- scalability: upgrade to enterprise level DB, tune database (index tables, denormalise), cluster
- fault tolerance: use transaction and recover features of RDBMS
- caveat: the object-relational impedance mismatch

The object-relational Impedance Mismatch

- executing basic CRUD operations on databases through JDBC is easy
- CRUD = **c**reate, **r**ead, **u**pdate, **d**elete
- this is the foundation for providing peristency to objects
- problems due to conceptual differences between OOP and RDBMS

The Basics

```
class Student {
     String id = null;
     String name = null;
}
```

ID	name
42	John
43	Jane

Class	=	Table
Instance	=	Row
Property	=	Column

Identifying Entities

- objects are referenced by address
- database records (rows) are identified by attributes (“primary keys”)
- i.e. it is possible to create multiple objects representing the same row
- state becomes inconsistent: how to sync objects with database ?

Identifying Entities

```
Student student1 = DB.getStudentById("42");
Student student2 = DB.getStudentById("42");

assert student1.equals(student2);
assert student1!=student2;

student1.setName("Tim");
student2.setName("Tom");


DB.save(student1);
DB.save(student2);
```

create two objects representing same record (row)

state becomes inconsistent

persistent state now depends on order of saving

Possible Solutions

- use a cache component
- a cache is basically a map associating keys (identities) with persistent objects: **`Map<String,Student>`**
- instead of creating multiple objects representing records, existing objects are reused
- caches need to solve several problems:
  - be fast (e.g., use concurrent maps)
  - avoid memory leaks: eviction policies
  - make sure cache referenced do not interfere with GC
- example : Google's guava cache component (https://github.com/google/guava/wiki/CachesExplained)

Impedance Mismatch - Other Issues

- references between objects use keys (foreign keys) that must be resolved
- resolution must be lazy to avoid cascading query effects when foreign key references are resolved -- can be solved using the proxy pattern
- OO programs have no native concept of transactions, a transaction manager must be implemented to synchronise application state with database
- RDBMS have no native concept of 1:many references (like `Collection` data structures), this is modelled via backreferences
- RDBMS do not support inheritance, hierarchies must be flattened
- RDBMS use different types for primitive (numeric, string), like varchar

Object Relational Mapping Frameworks

- to address those issues, a special type of **middleware** called **object relational mapping frameworks (ORMs)** is often used
- examples:
  - Hibernate (Java, has been ported to .NET) -- http://hibernate.org/
  - ActiveRecord (Ruby, has been ported to Python, Java and other languages, default framework used by RubyOnRails) https://github.com/rails/rails/tree/master/activerecord

Mapping between Layers: Misc

- ORMs are based on users providing mappings between classes and tables
- this is usually done via config files, annotations etc
- this can also be based on naming conventions: **convention over configuration**, pioneered by Ruby on Rails
- **model-driven architecture (MDA)** is based on the idea that schemas and class models etc are models instantiating particular meta-models, and this can be used to precisely define mappings
- often, one layer is generated: for instance, a database schema (CREATE TABLE statements) and mapping code is generated from inspecting a domain model

Generating Mapping Code Example: JAXB

- the structure of a particular XML document is defined by a schema (XSD)
- many applications consume and produce XML, so need to build an object model to represent the structure of the XML
- XML databinding is a way to facilitate this by statically generating a parser from an XSD that populates an object model resembling the structure of the XML document

Background: Schemas

- a schema refers to a formal description of the data (their structure and relationships)
- examples:
  - a database schema specifies tables and properties like primary and foreign keys, column data types -- it can be queried and manipulated (e.g., create new tables) with SQL, in particular the DDL (“data description language”) subset
  - an XML Schema defined a class of valid XML documents by specifying tags, nesting rules between tags and attributes , is is expressed in XMLSchema (itself XML-based) and tools can be used to check documents for validity
  - HTML can also be described by a schema, and tools like the W3C Validator can use this to check whether web sites are valid HTML
  - Swagger contains a schema language for JSON (to be discussed)
- related:
  - grammars describing valid strings w.r.t. formally defined language (e.g., Java)
  - classes describing valid objects

XML Data Example

```
<?xml version="1.0" encoding="UTF-8"?>
<email id="id1" .. xsi:noNamespaceSchemaLocation="email.xsd">
    <to>
        <email_address>jenz@vuw.ac.nz</email_address>
        <display_name>Jens Dietrich</display_name>
    </to>
    <cc>
        <email_address>studentsSWEN301@vuw.ac.nz</email_address>
        <display_name>SWEN301 student list</display_name>
    </cc>
    <subject>update</subject>
    <body>this lecture notes have been updated</body>
</email>
```

XSD reference

XML Schema: Define Type of XML Documents

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
     <xs:complexType name="participant">
          <xs:sequence>
               <xs:element name="email_address" type="xs:string"/>
               <xs:element name="display_name" type="xs:string" minOccurs="0"/>
          </xs:sequence>
     </xs:complexType>
     <xs:element name="email">
          <xs:complexType>
               <xs:sequence minOccurs="0">
                    <xs:element name="to" type="participant" maxOccurs="32"/>
                    ..
```

Code Generated with xjc

- created domain classes: **`Email`**, **`Participant`**
- created parser: **`ObjectFactory`**
- code: https://github.com/jensdietrich/se-teaching/tree/main/jaxb
- usage example:

```
JAXBContext jc = JAXBContext.newInstance("nz.ac.vuw.jenz.jaxb.generated");
Unmarshaller parser = jc.createUnmarshaller();
File file = new File("email1.xml");
Email mail = (Email) parser.unmarshal(file);
```

Tools to Auto-Map between Layers

Name	Uses Information from:	Target	Method
JAXB (XML data binding)	persistence layer	domain layer	static code generation of Java code
java.beans XML Serializer	domain layer	persistence layer	runtime reflection “on-the-fly”
DDL generators in ORMs (= generating TABLES)	domain layer	persistence layer	static code generation of SQL
Naked Objects / Isis	domain layer	user interface	runtime reflection “on-the-fly”
JSON Databinding (jackson, gson, ..)	domain/persistence layer	persistence/domain layer	runtime reflection “on-the-fly”

Tools to Auto-Map between Layers ctd

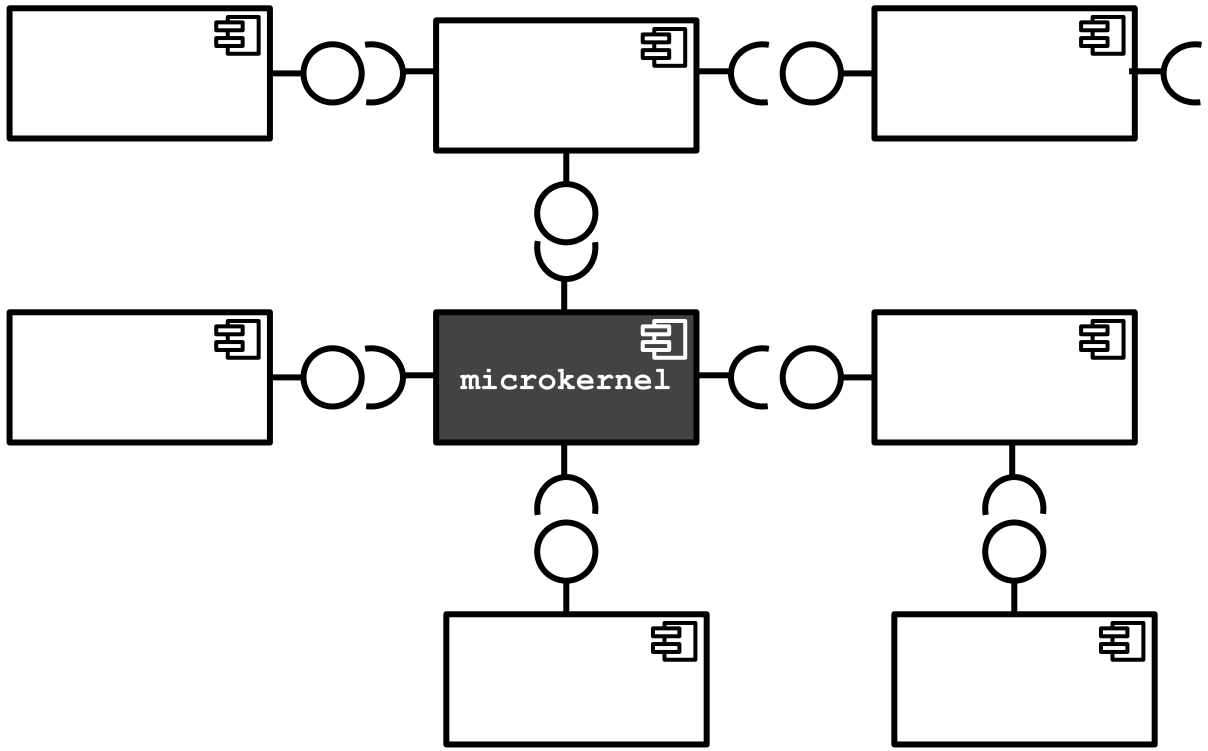
- the economics of using such tools is often superior for large projects
- avoids structural redundancy: helps to uphold DRY (Do Not repeat Yourself)* principle on the design level
- drawbacks: less optimised code (performance, usability)

*Andy Hunt and Dave Thomas: The Pragmatic Programmer: From Journeyman to Master. The Pragmatic Bookshelf 1999.

Pattern: Plugin-Based Architecture / Microkernel

- plugin -- components that extends the functionality of a host program
- pioneered by web browsers to customise behaviour and extend functionality
- this idea can be extended by allowing plugins to extend the functionality of plugins
- then the original application can be reduced to a minimum, a **microkernel**

Plugins and Microkernel



Example: Eclipse

- Eclipse is based on the OSGi component model
- the microkernel is the OSGi framework plus a bundle containing the extension registry
- plugins publish extension points to register services they can consume, and extensions for existing extension points
- examples: extension points to add action buttons to menus, view or editor view components for certain file types, help pages etc
- often (but not always) extension points provide an interface that must be implemented by extensions

Social Aspects

- plugin-based architectures are particularly suitable to promote software ecosystems
- to be successful, those systems need social rules and conventions
- Eclipse can now be used:
    - as a Java IDE
    - as a some PL X IDE
    - as a Torrent client (Vuze/Azureus is Eclipse-based)
    - as a server-side platform

Eclipse Housekeeping Rules (Selection)

- **Invitation Rule:** Whenever possible, let others contribute to your contributions
- **Fair Play Rule:** All clients play by the same rules, even me.

from Beck/Gamma in "Contributing to Eclipse: Principles, Patterns, and Plug-Ins ", Addison Wesley, 2003, see also http://se.inf.ethz.ch/old/teaching/ss2006/0284/slides/inside-eclipse.pdf

Combining Patterns

- large evolving systems often combine several patterns
- E.g. plugins can still be grouped and arranged in layers with additional constraints governing their dependencies

Pattern: Service-Oriented Architecture (SOA)

- evolved from components-based architectures
- approaches in the 90ties to develop distributed object technologies that were language-agnostic: CORBA
- HTTP / XML -based web-service protocol stack then introduced around 2000 to facilitate better interoperability
- modern SOA is based on lightweight protocols, typically consisting of HTTP + JSON, **REST-full services** *

* Roy Thomas Fielding: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, UCI 2000. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

SOA Characteristics

- service abstraction: services are blackbox, they hide implementation details including PL / server specifics
- location transparency: services hide their location (e.g., using DNS)
- service statelessness: services should not hold state
- service contracts and stability: services should have stable contracts that describe their functionality
- discoverability: services can be discovered by service consumers (clients)
- autonomy: service control their functionality at design- and run-time
- composability: services can be composite

SOA Trends and Challenges

- made popular by tech companies offering APIs to use functions “as a service”: SalesForce and eBay (2000), Amazon (2002) https://apievangelist.com/2012/12/20/history-of-apis/
- granularity: microservices (too micro ?)
- contracts: how to describe a service (expressiveness vs simplicity), WSDL, OpenAPI / Swagger, ..
- lifecycle and evolution: no standardised versioning policies
- scalability: scalability is achieved by the ability to parallelise on the cloud (possible due to statelessness), but if granularity is too fine-grained, this can be inefficient

The Bezos Mandate (Amazon, 2002)

1. All teams will henceforth expose their data and functionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols – doesn't matter. Bezos doesn't care.
5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
6. Anyone who doesn't do this will be fired.

Have a nice day!

Architecture and ilities (ISO/IEC 25010)

- one aspect to drive architecture non-functional requirements
- examples:
  - for instance, high scalability requirements may necessitate a SOA with the ability to massively parallelise using cloud computing
  - if requirements focus on robustness and availability in an environment with poor network connectivity, than SOA might not be suitable
  - SOA also has limitation when testability is a priority, since application testing may rely on remote services not controlled by the application

Recovering and Verifying Architecture

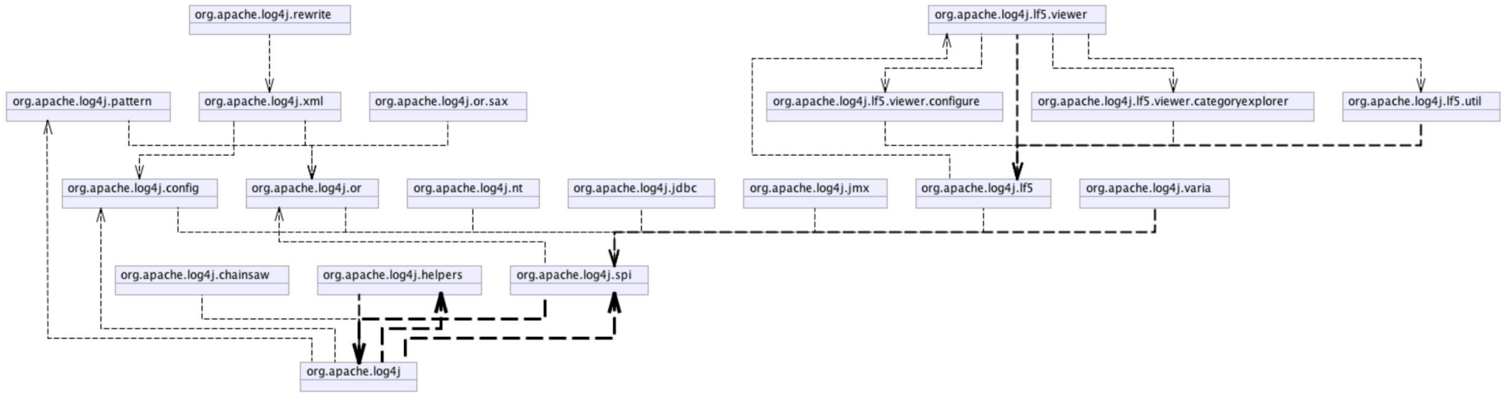
- **architecture recovery** aims at building (architectural) models from code for the following purposes:
  - comprehend the structure in case it was never defined
  - compare it with the original architecture to assess erosion and drift
- this is an example of **static program analysis** : build and reason models from code
- this is also an example of **reverse engineering**
- some case tools try to support roundtrip - engineering, two way synchronisation between models and code
- the opposite is **dynamic program analysis**: execute and observe code (e.g., testing, monitoring, debugging and profiling)
- tools (commercial): structure101, lattix, sotograph

Example: Java 2 UML

- UML class diagrams can be easily computed from Java source code
- there are some limitations, including:
  - character of association (aggregation of not)
  - weak references (via reflection, not in the GC sense)
- but can be integrated in automated build, **models are always consistent with code** (after each build cycle)
- a technical challenge is to create an optimal graph layout that minimises overlaps
- some tools address this be only creating local, navigable models

Example: Java 2 UML -- yWorks UML Doclet

- plugins into javadoc
- UML class diagrams embedded into HTML pages for packages and classes
- clickable image maps (models are navigable)

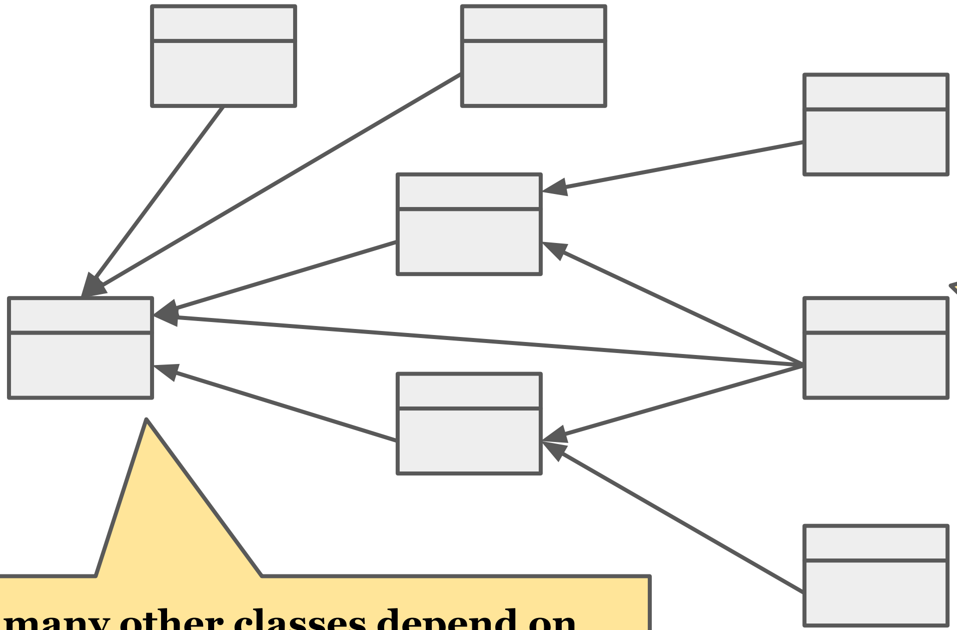


package dependencies in log4j, model extracted with yDocs
https://github.com/jensdietrich/se-teaching/tree/main/java2uml

Example: JDepend

- not just visualisation, but analysis of dependencies between Java packages
- extracts several metrics, and finds circular dependencies
- relate to violations of some SOLID design principles
- this introduces another dimension: abstract specification vs implementation components, with constrained dependencies:
    - concrete should depend on abstract, abstract must not depend on concrete

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**

many other classes depend on this one - if it changes, they all may have to change as well.
it should therefore be **abstract**

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 [alt 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)

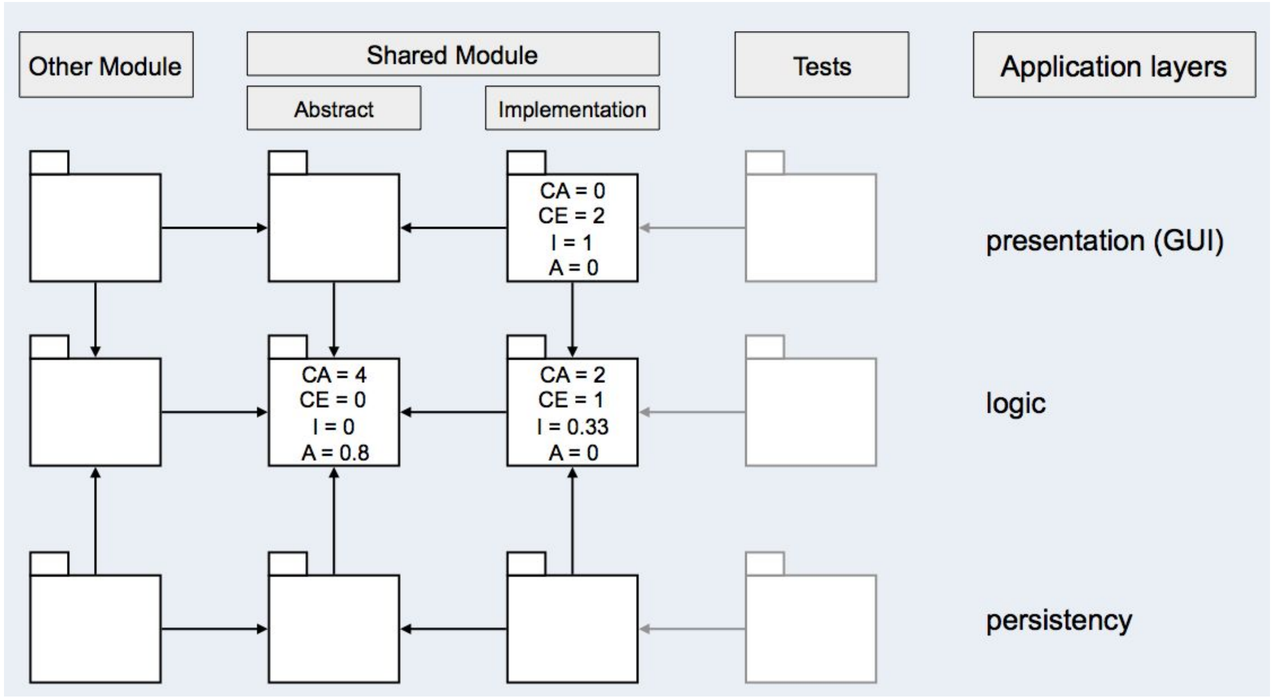
Tooling: JDepend (ctd)

- **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
- **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 for 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**