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

SWEN 301 : Scalable Software Development

Components and Evolution

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

Overview

- component models
- contracts between components
- dependencies and dependency resolution
- compatibility issues
- semantic versioning

My thesis is that the software industry is weakly founded, and that one aspect of this weakness is the absence of a software components subindustry.

McIlroy, Malcolm Douglas (January 1969). "Mass produced software components". Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968. Scientific Affairs Division, NATO. p. 79.

Why Components ?

- achieve economy of scale through reuse
- better quality of code due to it being reused: faults are discovered faster as code is used in different contexts

What is a Component ?

1. Multiple-use and Non-context-specific
2. Composable with other components
3. Encapsulated (non-investigable through its interfaces)
4. A unit of independent deployment and versioning

Clemens Szyperski: Component Software - Beyond Object-Oriented Programming – 2nd Ed. Addison-Wesley / ACM Press, 2002.

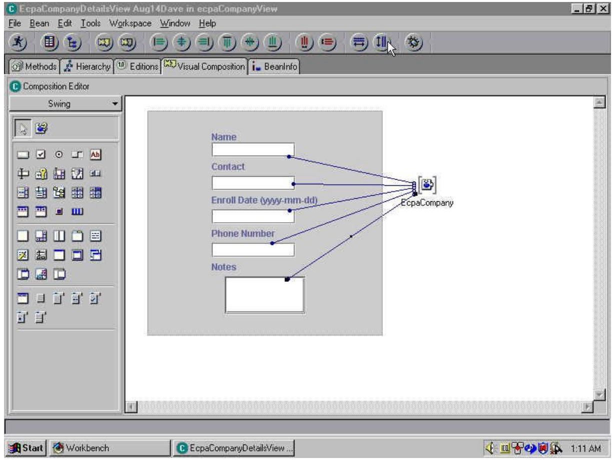
Components AKAs

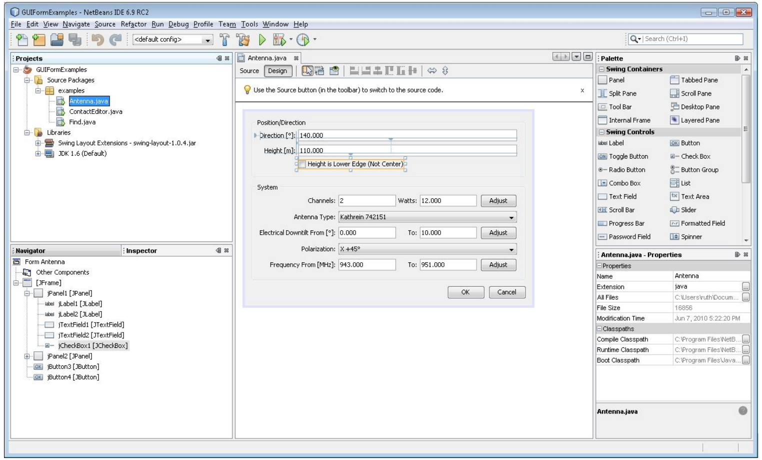
- packages, modules, bundles, parts, libraries ..
- some names have a slightly different meaning (emphasising certain aspects, or adding additional properties)
- often language- / product- specific

Approaches to Components in Java: Java Beans

- components are called **beans**
- components are instance of plain Java classes with some additional rules
- properties are defined by getters and setters following canonical naming patterns -- this facilitates introspection (service discovery)
- public constructors without parameters allow dynamic instantiation
- a simple event model makes beans observable (by registering property change listeners)
- tooling supports reflection and persistence of beans
- beans are typically not units of separate deployment and versioning
- use case: user interface composition via UI builders

User Interface Builders





facilitates approaches like 4GL programming (90ties), “low code” (2020ties)

Approaches to Components in Java: OSGi

- components are called **bundles**
- bundles run in special OSGi containers (felix, equinox, knopflerfish)
- bundles are jar files with additional metadata (extended jar manifests)
- bundles have private classloaders, this can be used to separate them (e.g., multiple versions)
- the container can deploy, undeploy, start, stop and dynamically connect (wire) bundles, use case: dynamic upgrade without restarting application

Approaches to Components in Java: OSGi (ctd)

- bundles can consume and provide services represented by Java interfaces
- the container can dynamically “re-wire” components if the components providing a service change (e.g., become unavailable)
- extensions to make this more explicit: OSGi Declarative Services, Eclipse plugins, Spring Dynamic Modules
- widely used in application servers, and plugin based products (Eclipse !)
- example: https://github.com/jensdietrich/se-teaching/tree/main/osgi

Approaches to Components in Java: Maven

- components are called **artifacts** and organised in namespaces called **groups**
- artifacts are jar files with additional metadata (**`pom.xml`** or similar for gradle & co)
- focus is on describing dependencies, deploying and versioning

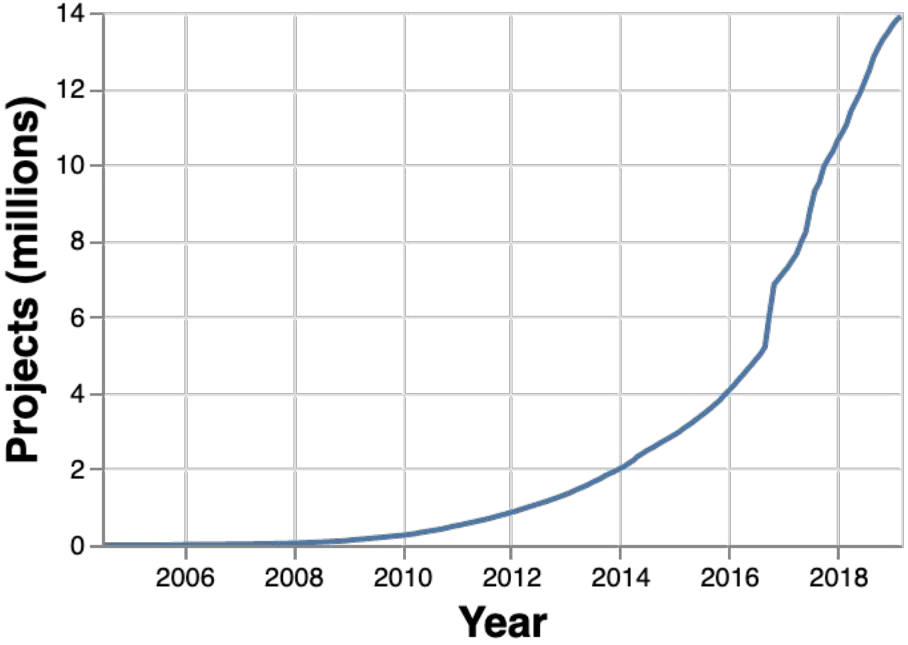
Approaches to Components in Java - Comparison

	multiple-use / non context specific	composable	unit of independent deployment and versioning	needs container	supports modularity at
Java beans	YES	YES	NO	NO	design and run-time
OSGi bundles	YES	YES	YES	YES	run-time
Maven artefacts	YES	YES	YES	NO (not at runtime)	build-time

Dependencies

- when components are being used, a **dependency** to it is added by the client program using it
- as the component itself may depend on other components, **transitive dependencies** are added as well
- this means that the components must be present at buildtime (e.g. to compile) and at runtime
- early reuse: download libraries, copy them into project, manage transitive dependencies manually, build classpath
- state-of-the-art: dependencies are symbolic (e.g., URLs or XML elements), only direct dependencies are declared, transitive dependencies are resolved and actual components fetched from a central repo by a **dependency manager** such as `mvn`

Success Stories: More Components are Available
Evolving Component Markets / Eco-Systems



number of artifacts in Maven repo (mvnrepository.com)

Success Stories: Components are being Used

AVG Number of **Direct** Dependencies by Package Manager

projects, projects with one version

number of **direct** dependencies in any / first / last version within tracked range

	PROJ	ONE	AVG	STDEV	AVG1	STDEV1	AVGL	STDEVL
Cargo	11,251	3,236	6.13	9.56	3.85	3.54	4.86	4.39
Maven	63,497	16,952	9.96	23.23	5.03	6.3	5.3	7.01
CRAN	11,646	3,223	5.56	8.75	3.54	3.87	6.05	5.98
Pypi	4,083	935	8.76	14.87	2.71	2.85	3.15	3.25
CPAN	28,015	5,055	7.49	15.04	7.24	9.33	10.87	14.34
Elm	1,273	352	4.43	6.36	2.5	1.75	2.54	1.79
Homebrew	1,806	1,784	1.01	0.13	2.77	2.61	2.77	2.61
NPM	547,338	153,412	7.34	22.52	8.75	16.16	9.76	15.78
Atom	3,845	600	11.18	22.11	2.92	3.79	4.08	5.43
Haxelib	470	188	6.06	9.84	1.8	1.26	1.89	1.31
NuGet	76,775	19,860	12.28	48.76	2.77	3.54	2.96	3.77
Dub	550	144	8.69	18.76	1.58	1.18	1.85	1.8
Packagist	104,585	28,340	7.59	16.48	3.37	3.97	4.04	4.6
Rubygems	119,942	35,671	6.41	15.35	4.19	3.41	4.89	3.95
Hex	3,667	1,248	5.4	8.01	2.14	1.56	2.33	1.7
Pub	2,867	688	9.06	17.72	3.08	2.45	3.95	3.34
Puppet	3,703	956	5.61	8.08	2.01	1.83	2.29	2.31

programs add dependencies as they evolve

Dietrich, J., Pearce, D. J., Stringer, J., Tahir, A., & Blincoe, K. Dependency Versioning in the Wild. MSR19

Dependency Resolution

- locate package referenced (in a central repository) using a symbolic name, including a version (or version constraint)
- download it (into a local cache)
- recursively resolve dependencies of dependencies
- resolve conflicts

Dependency Resolution with Maven

- necessary for compilation
- dependencies can be listed with **`mvn dependency:resolve`**
- the hierarchical structure of dependencies can be displayed with **`mvn dependency:tree`**

Dependency Resolution with Maven - Example

```
[INFO] nz.ac.vuw.jenz.webfuzz:webfuzz:jar:1.0-SNAPSHOT
[INFO] +- junit:junit:jar:4.12:compile
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.3:compile
[INFO] +- com.pholser:junit-quickcheck-generators:jar:0.8.2:compile
[INFO] +- org.apache.httpcomponents:httpclient:jar:4.5.7:compile
[INFO] |  +- org.apache.httpcomponents:httpcore:jar:4.4.11:compile
[INFO] |  +- commons-logging:commons-logging:jar:1.2:compile
[INFO] |  \- commons-codec:commons-codec:jar:1.11:compile
[INFO] +- log4j:log4j:jar:1.2.17:compile
[INFO] +- com.google.guava:guava:jar:27.1-jre:compile
[INFO] |  +- com.google.guava:failureaccess:jar:1.0.1:compile
[INFO] |  +- com.google.guava:listenablefuture:jar:9999.0-empty-to-avoid-conflict-with-guava:compile
[INFO] |  +- com.google.code.findbugs:jsr305:jar:3.0.2:compile
[INFO] |  +- org.checkerframework:checker-qual:jar:2.5.2:compile
[INFO] |  +- com.google.errorprone:error_prone_annotations:jar:2.2.0:compile
[INFO] |  +- com.google.j2objc:j2objc-annotations:jar:1.1:compile
[INFO] |  \- org.codehaus.mojo:animal-sniffer-annotations:jar:1.17:compile
[INFO] +- commons-io:commons-io:jar:2.6:compile
[INFO] +- org.apache.commons:commons-lang3:jar:3.9:compile
[INFO] +- commons-cli:commons-cli:jar:1.4:compile
[INFO] +- com.pholser:junit-quickcheck-core:jar:0.8.2:compile
[INFO] |  +- org.javaruntype:javaruntype:jar:1.3:compile
[INFO] |  |  \- org.antlr:antlr-runtime:jar:3.1.2:compile
[INFO] |  +- ognl:ognl:jar:3.1.12:compile
[INFO] |  |  \- org.javassist:javassist:jar:3.20.0-GA:compile
[INFO] |  +- ru.vyarus:generics-resolver:jar:2.0.1:compile
[INFO] |  \- org.slf4j:slf4j-api:jar:1.7.25:compile
[INFO] \- org.apache.commons:commons-math3:jar:3.6.1:compile
```

project: running `mvn dependency:tree` on https://bitbucket.org/jensdietrich/webfuzzer/

Caveats

- components have to be generic -- they can be used in different contexts (by many applications)
- this creates complexity, e.g. poor analysability
- high rates of reuse mean that if components are faulty, a large number of applications can be affected
- often, what a component offers does not exactly match requirements:
  - it has to be adapted
  - it bloats an application as unused code (and potentially vulnerable) code is also integrated

JS Leftpad

- caused by deep transitive dependencies in the NPM (JavaScript) eco-system to a trivial package called left-pad
- withdrawn by developer due to legal dispute
- broke thousands of application depending on it
- problems:
  - package manager allowed withdraw of packages other packages depend on (leaving them “orphaned” and causing builds to fail)
  - packages to fine-grained -- simple 11-line function, should have been part of system library
  - programs have too many transitive dependencies
- https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/
- https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/

Typosquatting

- type of attack that emerged around 2017 in JS (NPM) and python package repos
- hackers add package with similar name to a popular package into repo
- this has similar functionality, but also contains some malicious code
- https://thenewstack.io/npm-cleans-typosquatting-malware/

Apache Commons Collections Vulnerabilities

- CVE-2015-7501 and related CVEs
- Java binary serialisation widely used in networked applications
- vulnerable library has generic data structures, including *reflective computing maps* where an arbitrary function (saved in the stream) can be used to compute the value for a key
- with a carefully crafted object, this can be used to execute arbitrary functions when the stream is deserialised, i.e., arbitrary code execution attacks
- exploited in attack on SF public transport

Apache Commons Collections Vulnerabilities (ctd)

- problem: a library providing very generic functionality rarely required but exploitable
- see https://github.com/jensdietrich/xshady/tree/main/CVE-2015-7501 for proof-of-vulnerability code (based on https://github.com/frohoff/ysoserial)
- related DOS attacks: https://drops.dagstuhl.de/opus/volltexte/2017/7260/pdf/LIPIcs-ECOOP-2017-10.pdf

Log4Shell

- CVE-2021-44228 (and similar CVEs)
- top-severity 10/10, injection activated by log statements
- can be easy exploited , e.g. unusual headers / paths in when requests are often logged
- library is pervasive
- led to significant political response
- see https://github.com/jensdietrich/xshady/tree/main/CVE-2021-44228 for proof-of-vulnerability code

DLL Hell & Co

- problem first observed with DLL (Windows) libraries
- different versions of the libraries shared
- lead to compatibility errors - situations where applications use the wrong version of an API: calling non-existing methods, calling methods with incorrect parameters etc
- Java version: jar / classpath hell

Avoiding Hell

Strategy 1: clearly separate multiple versions

- OSGi achieves this by using classloaders
- in Java, two versions of a class can be loaded with different classloaders, and be completely separated

Strategy 2: pick a winner

- Maven tries to do this using **dependency mediation**
- if multiple versions are present in the dependency, the one nearest to the root wins
- if more than one version have the same depth, the first one wins
- https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

Conflict Resolution with Maven

```
[INFO] [dependency:tree]
[INFO] org.apache.maven.plugins:maven-dependency-plugin:maven-plugin:2.0-alpha-5-SNAPSHOT
[INFO] +- org.apache.maven.reporting:maven-reporting-impl:jar:2.0.4:compile
[INFO] |  \- commons-validator:commons-validator:jar:1.2.0:compile
[INFO] |     \- commons-digester:commons-digester:jar:1.6:compile
[INFO] |        \- (commons-collections:commons-collections:jar:2.1:compile - omitted for conflict with 2.0)
[INFO] \- org.apache.maven.doxia:doxia-site-renderer:jar:1.0-alpha-8:compile
[INFO]    \- org.codehaus.plexus:plexus-velocity:jar:1.1.3:compile
[INFO]       \- commons-collections:commons-collections:jar:2.0:compile
```

this version of commons-collections wins
as it is “more direct” at depth 3 (not 4)

Avoiding Conflicts: Shading

- copy libraries into projects and move them into different package
- allows multiple versions of packages to co-exist
- even the JDK uses it: **`jdk.internal.org.xml.sax`** and **`jdk.internal.org.objectweb.asm`** !
- can be automated: maven shade plugin
- caveats:
    - static analysis required may miss dynamic language features – can cause runtime errors
    - software composition analysis tools may miss undeclared dependencies, leads to hidden vulnerabilities, e.g. https://github.com/github/advisory-database/pull/2444 , https://arxiv.org/abs/2306.05534

Supporting Conflict Revolution: Ranges

- the version element in a Maven dependency is actually a constraint which version to use
- `<version>1.2.3</version>` -- use version 1.2.3 if possible, but if mediation is required, another version can be used
- `<version>[1.2.3]</version>` -- use exactly version 1.2.3, this means an increased probability that dependency resolution will fail
- `<version>[1.2.3,1.4.0)</version>` -- use at least 1.2.3 or any version larger than this but less than 1.4.0, this gives mvn more options to resolve dependencies, and perhaps chose an optimal versions (later is better)

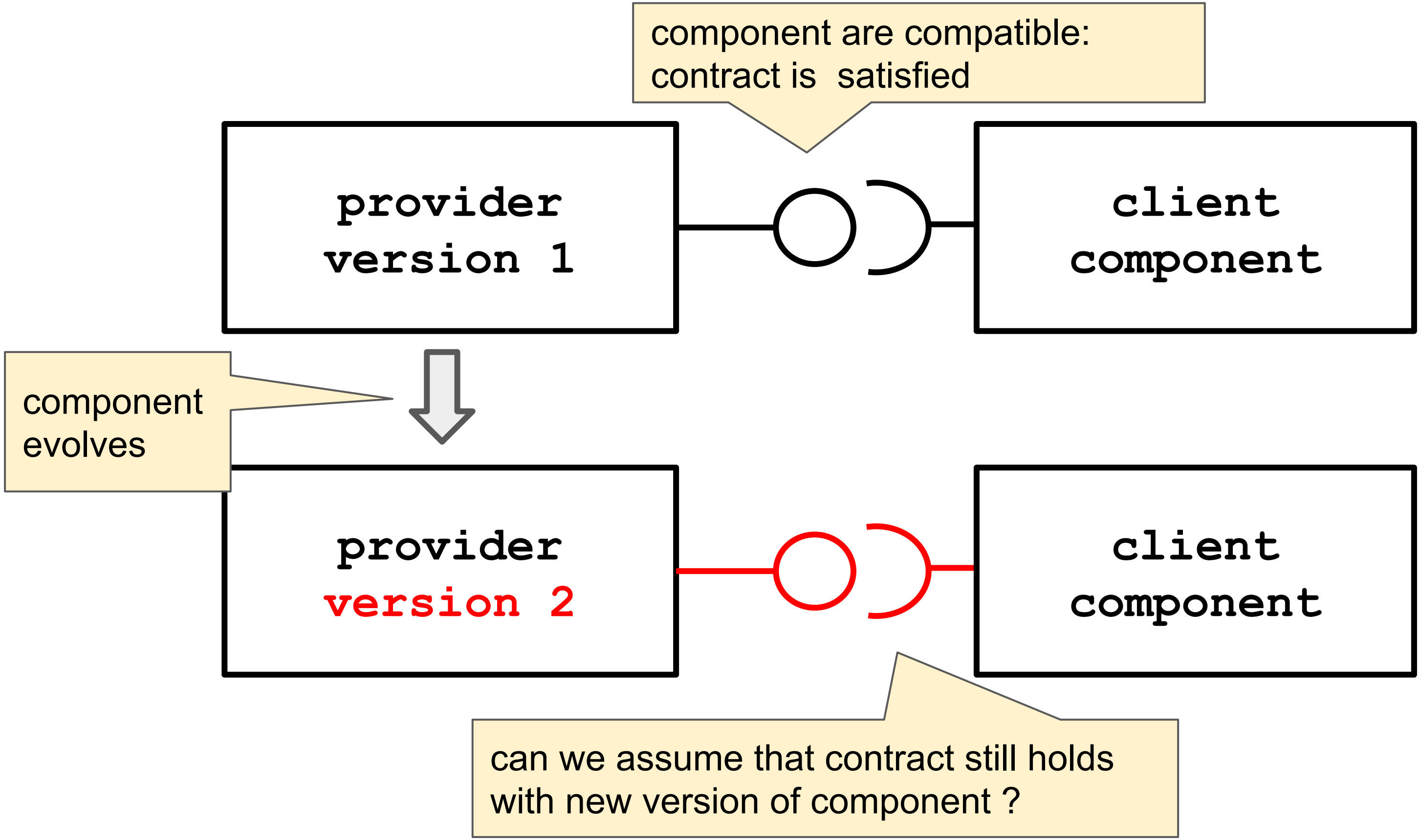
Compatibility and Contracts

- collaborating components have to adhere to a contract
- violating the contract leads to compatibility errors
- example: **`component1`** provides an API **`save(Student)`**, **`component2`** uses (consumes) it
- contract example:
  - syntax-level: component1 must call **`save`** with a **`Student`** instance as parameter, it must not be **`null`**
  - semantic-level: if the **`save`** operation is successful, **`save()`** will return true, false otherwise
  - quality-of-service-level: save is guaranteed to return a result with 200ms
  - license: component2 is not licensed under a viral open source license, such as the GPL
- Beugnard, A., Jézéquel, J. M., Plouzeau, N., & Watkins, D. (1999). Making components contract aware. Computer, 32(7), 38-45.

Evolution

- agile practices have led to (very) short release cycles
- what happens when a package a program depends on changes ?
- it is often desirable to upgrade dependencies, in particular when the change is a bugfix

Compatible Change



Compatible Evolution in Java

- complex rules
- source compatibility: if we can compile with lib-version1, then we can also compile with lib-version2
- binary compatibility: if we can run with lib-version1, then we can also run with lib-version2
- neither does binary compatibility imply source compatibility, nor vice versa
- https://www.slideshare.net/JensDietrich/presentation-30367644
- https://drops.dagstuhl.de/opus/volltexte/2016/6106/pdf/LIPIcs-ECOOP-2016-12.pdf

Compiling vs Linking

- **how to deploy / use a new version of a library**
- **recompile**: replace library version (for instance, update Maven dependency), recompile project
- relates to **source compatibility**
- **relink**: replace library version in classpath, don't recompile
- relates to **binary compatibility**

What Happens when you recompile / relink ?

lib version 1.0

```
import java.util.*;
public class Foo {
  public static Collection getColl() {
    return new ArrayList();
  }
}
```

lib version 2.0

```
import java.util.*;
public class Foo {
  public static List getColl() {
    return new ArrayList();
  }
}
```

client program using lib

```
import java.util.*;
public class Main {
  public static void main(String[] args) {
    Collection coll = Foo.getColl();
    System.out.println(coll);
  }
}
```

Source but Not Binary Compatible

lib version 1.0

```
import java.util.*;
public class Foo {
  public static Collection getColl() {
    return new ArrayList();
  }
}
```

lib version 2.0

```
import java.util.*;
public class Foo {
  public static List getColl() {
    return new ArrayList();
  }
}
```

client program using lib

```
import java.util.*;
public class Main {
  public static void main(String[] args) {
    Collection coll = Foo.getColl();
    System.out.println(coll);
  }
}
```

- source compatible: specialising return type is ok
- byte code contains “descriptor” of invoked method, and changing the return type changes the descriptor
- not binary compatible: results in **`NoSuchMethodError`**

What Happens when you recompile / relink ?

lib version 1.0

```
import java.util.*;
public class Foo {
  public static List<String> getColl() {
    return new ArrayList();
  }
}
```

lib version 2.0

```
import java.util.*;
public class Foo {
  public static List<Integer> getColl() {
    return new ArrayList();
  }
}
```

client program using lib

```
import java.util.*;
public class Main {
  public static void main(String[] args) {
    List<String> coll = Foo.getColl();
    System.out.println(coll.size);
  }
}
```

Binary but Not Source Compatible

lib version 1.0

```
import java.util.*;
public class Foo {
  public static List<String> getList() {
    return new ArrayList();
  }
}
```

lib version 2.0

```
import java.util.*;
public class Foo {
  public static List<Integer> getColl() {
    return new ArrayList();
  }
}
```

client program using lib

```
import java.util.*;
public class Main {
  public static void main(String[] args) {
    List<String> coll = Foo.getColl();
    System.out.println(coll.size);
  }
}
```

- cannot be recompiled as type parameter changes
- in bytecode contains type parameters are erased
- binary compatible !
- but when client code iterates over (non-empty) list, a runtime exception (class cast) occurs

Dealing with Compatibility

- use a tool !
- revapi, clirr etc can detect many incompatible changes between versions

Predictability vs Agility

- build automation facilitates to release often (daily)
- with version ranges, a dependency manager can choose optimal versions, usually the latest satisfying constraints (e.g., latest in range)
- assumption: the latest version will have bug fixes and other improvements (e.g., performance)
- improvements can be propagated as part of the automated release cycle
- clients want guarantees that changing a dependency to a later version does not introduce compatibility errors
- conservative approach: declare a dependency to a fixed version, have **reliable builds** but must upgrade manually or miss out on bug fixes and other improvements

When to Update

(too) early: use cutting edge version, might not be thoroughly tested, integration problems

(too) late: may introduce vulnerabilities discovered

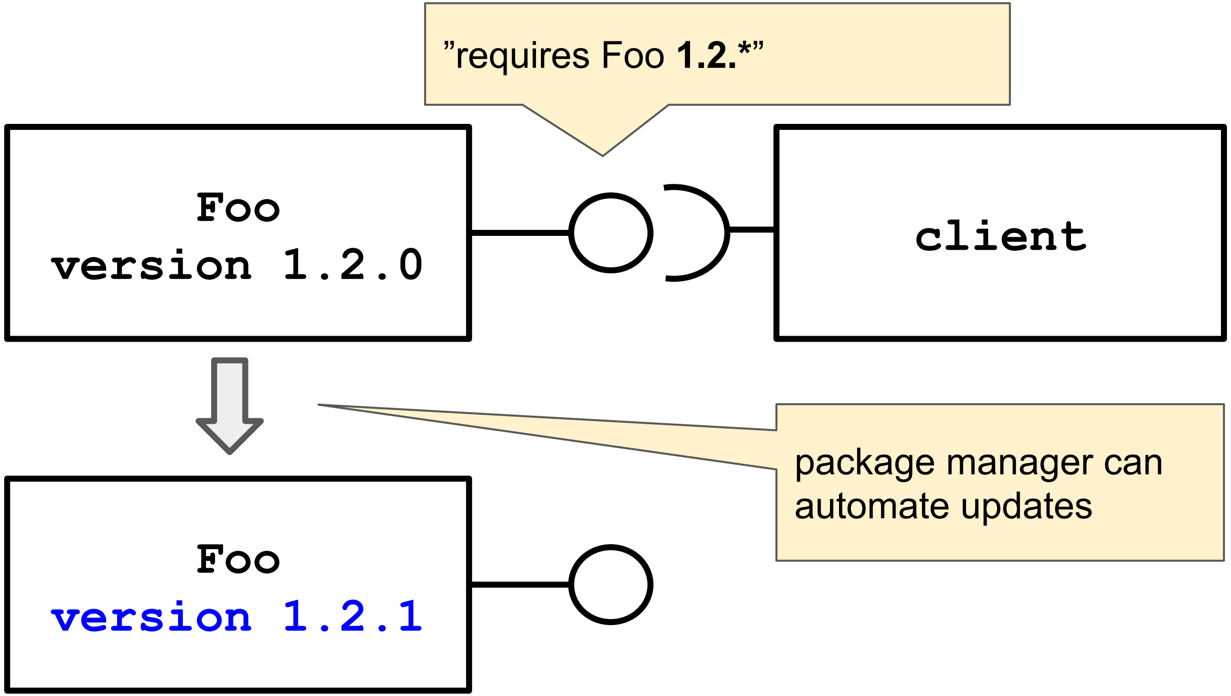
The Equifax Incident

- equifax is credit bureau
- attack stole private records of 147.9 million Americans and records from Canadians and British residents
- exploited a known vulnerability in an outdated version of the struts Java framework used by equifax (CVE-2017-5638)
- indicted members of China's People's Liberation Army
- congressional report: https://republicans-oversight.house.gov/wp-content/uploads/2018/12/Equifax-Report.pdf

Semantic Versioning

- tries to define the sweet spot between agile and predictive
- versions comply to the general structure `<major>.<minor>.<patch>`
- changes in `<patch>` are used for compatible bug fixes
- changes in `<minor>` indicate that APIs are deprecated, or new backwards compatible (not breaking clients) APIs are introduced
- changes in <major> indicate breaking changes
- versions with major version `0` are considered unstable
- https://semver.org/

Semantic Versioning Example



Semantic Versioning in Practice

- slow adaptation, differences between package managers (Dietrich et al MSR19, Decan et al TSE19)
- compatibility rules complex, developers don't understand them (Dietrich et al ESE15)
- semantic version calculators needed to compute / verify version changes
- those calculators are based on (mainly static) program analysis
- theoretical limitations: Rice's theorem
- interesting questions for research