Adding a Module System to Java

Rok Strniša

Computer Laboratory, University of Cambridge
Email: `Rok.Strnisa@cl.cam.ac.uk`
URL: `http://www.cl.cam.ac.uk/~rs456/`

May 8, 2008 @ The British Computer Society

Joint work with Peter Sewell and Matthew Parkinson

1. What is a module system?
2. Why Java needs a module system
3. Current solutions
4. The Java Module System (JAM)
5. improved JAM (iJAM)
6. Formalizations & proofs
7. Conclusion

What is a module system?

Definition

module system	A system in a programming language, which allows structuring programs of that language in modules.

Definition

module system	A system in a programming language, which allows structuring programs of that language in modules.
module	A piece of software that *should* implement a specific function in a largely independent manner.

Desirable properties

abstraction	hiding the implementation behind an interface.

Desirable properties

abstraction	hiding the implementation behind an interface.
composition	combining many modules.

Desirable properties

abstraction	hiding the implementation behind an interface.
composition	combining many modules.
static reuse	using module's code at many different places.

Desirable properties

abstraction	hiding the implementation behind an interface.
composition	combining many modules.
static reuse	using module's code at many different places.
dynamic reuse	sharing module's types and data.

Desirable properties

abstraction	hiding the implementation behind an interface.
composition	combining many modules.
static reuse	using module's code at many different places.
dynamic reuse	sharing module's types and data.
separate compilation	ability to compile a module in isolation.

Quick look at the ML module system

- Abstraction (abstract types, module types)

```
module Foo = struct            (* implementation *)
  type t = int
  let v = 5
end

let x = 42 + Foo.v   (* OK *)

module type View = sig         (* interface *)
  type t
  val v : t
end

module Bar : View = Foo        (* abstraction *)
let y = 42 + Bar.v   (* COMPILE ERROR *)
```

Quick look at the ML module system

- Composition

  Can be structured. Functors = functions over modules.
- Static reuse

  On the scale of each file (module).
- Dynamic reuse

  Not supported.
- Separate compilation

  Each file (module) compiled separately.

Why Java needs a module system

Java's current "modularity" constructs

class A low-level software unit. Gives abstraction with private members, limited composition with inheritance and inner classes, and low-level static reuse.

Java's current "modularity" constructs

class	A low-level software unit. Gives abstraction with private members, limited composition with inheritance and inner classes, and low-level static reuse.
object	An instance of a class. Provides dynamic reuse for classes.

Java's current "modularity" constructs

class	A low-level software unit. Gives abstraction with private members, limited composition with inheritance and inner classes, and low-level static reuse.
object	An instance of a class. Provides dynamic reuse for classes.
interface	A more explicit form of the abstraction provided by classes.

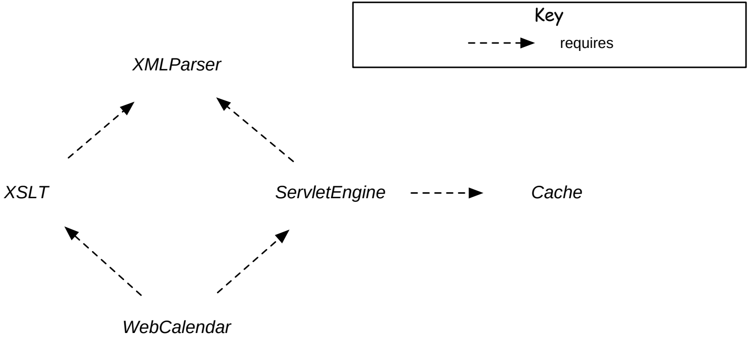
Java's current "modularity" constructs

class	A low-level software unit. Gives abstraction with private members, limited composition with inheritance and inner classes, and low-level static reuse.
object	An instance of a class. Provides dynamic reuse for classes.
interface	A more explicit form of the abstraction provided by classes.
package	Part of class namespace. Serves as barrier for default access restriction, and so enables package-wide abstraction for class members.

Java's current "modularity" constructs

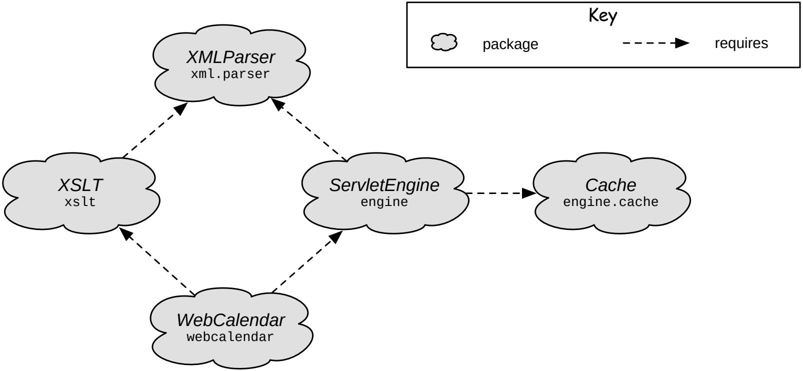
class	A low-level software unit. Gives abstraction with private members, limited composition with inheritance and inner classes, and low-level static reuse.
object	An instance of a class. Provides dynamic reuse for classes.
interface	A more explicit form of the abstraction provided by classes.
package	Part of class namespace. Serves as barrier for default access restriction, and so enables package-wide abstraction for class members.
JAR file	A group of classes/interfaces. Provides only high-level static reuse, nothing else.

An example



Servlet is a small program that runs on a web-server.

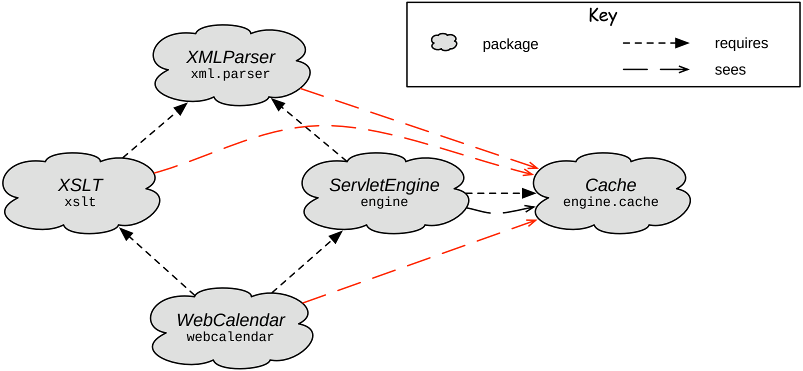
An example



Servlet is a small program that runs on a web-server.

An example

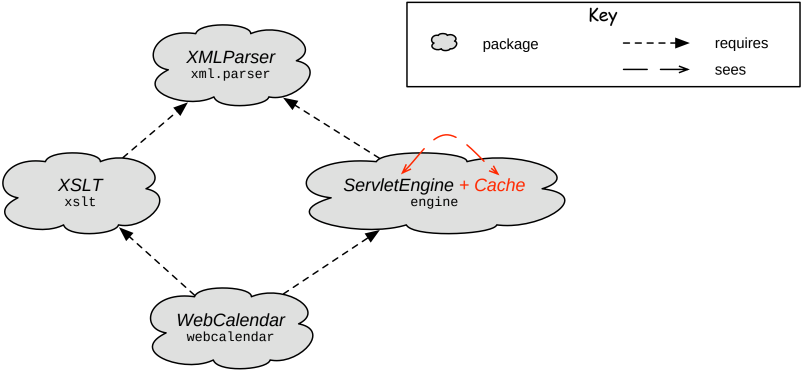
No high-level abstraction



Servlet is a small program that runs on a web-server.

An example

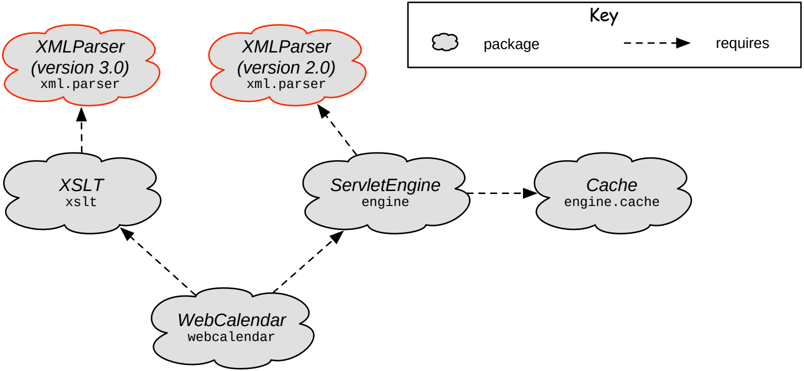
No high-level abstraction



Servlet is a small program that runs on a web-server.

An example

No high-level abstraction DLL/JAR Hell



Servlet is a small program that runs on a web-server.

Java's unsolved software engineering problems

- No high-level software units
  Have clear boundaries for high-level logical software units.

Java's unsolved software engineering problems

- No high-level software units

  Have clear boundaries for high-level logical software units.
- No high-level abstraction

  Allow different interfaces for different software units.

Java's unsolved software engineering problems

- No high-level software units

  Have clear boundaries for high-level logical software units.
- No high-level abstraction

  Allow different interfaces for different software units.
- DLL/JAR hell

  Allow software units of different versions to co-exist.

Java's unsolved software engineering problems

- No high-level software units
  Have clear boundaries for high-level logical software units.
- No high-level abstraction
  Allow different interfaces for different software units.
- DLL/JAR hell
  Allow software units of different versions to co-exist.
- No separate compilation
  Allow software units to be compiled in isolation.

Current solutions

Custom classloaders

Classloaders are Java's mechanism for loading code (at runtime).

How to use:

1. Extend `Classloader` class;

Custom classloaders

Classloaders are Java's mechanism for loading code (at runtime).

How to use:

1. Extend `Classloader` class;
2. implement `loadClass` (or `findClass`);

Custom classloaders

Classloaders are Java's mechanism for loading code (at runtime).

How to use:

1. Extend `Classloader` class;
2. implement `loadClass` (or `findClass`);
3. make an instance of the new classloader; and

Custom classloaders

Classloaders are Java's mechanism for loading code (at runtime).

How to use:

1. Extend `Classloader` class;
2. implement `loadClass` (or `findClass`);
3. make an instance of the new classloader; and
4. use it to resolve a class reference.

Custom classloaders

Classloaders are Java's mechanism for loading code (at runtime).

How to use:

1. Extend `Classloader` class;
2. implement `loadClass` (or `findClass`);
3. make an instance of the new classloader; and
4. use it to resolve a class reference.

If `loadClass` is implemented, then the *parent classloader* is consulted first. Overriding `findClass` allow one to change this behavior.

Custom classloaders

Classloaders are Java's mechanism for loading code (at runtime).

How to use:

1. Extend `Classloader` class;
2. implement `loadClass` (or `findClass`);
3. make an instance of the new classloader; and
4. use it to resolve a class reference.

If `loadClass` is implemented, then the *parent classloader* is consulted first. Overriding `findClass` allow one to change this behavior.

Each classloader has its own namespace — multiple versions!

Custom classloaders

Classloaders are Java's mechanism for loading code (at runtime).

How to use:

1. Extend `Classloader` class;
2. implement `loadClass` (or `findClass`);
3. make an instance of the new classloader; and
4. use it to resolve a class reference.

If `loadClass` is implemented, then the *parent classloader* is consulted first. Overriding `findClass` allow one to change this behavior.

Each classloader has its own namespace — multiple versions!

Hard to understand/use/debug. Each company makes their own.

Design patterns

Factory Pattern is a good example:

- Uses interfaces & reflection to find/generate classes;

Design patterns

Factory Pattern is a good example:

- Uses interfaces & reflection to find/generate classes;
- gives some low-level abstraction, and dynamic reuse.

Design patterns

Factory Pattern is a good example:

- Uses interfaces & reflection to find/generate classes;
- gives some low-level abstraction, and dynamic reuse.

Not very type-safe. Ad hoc.

Frameworks

OSGi is currently the most widely used framework:

- Service-oriented:
  Modules (bundles) provide services (access to a module's capability). Services can be registered and discovered through the service registry.

Frameworks

OSGi is currently the most widely used framework:

- Service-oriented:
  Modules (bundles) provide services (access to a module's capability). Services can be registered and discovered through the service registry.
- very customizable.

Frameworks

OSGi is currently the most widely used framework:

- Service-oriented:
  Modules (bundles) provide services (access to a module's capability). Services can be registered and discovered through the service registry.
- very customizable.

Not in language (not enforced). Not enough is enforced, e.g. no high-level abstraction or dynamic reuse.

The Java Module System (JAM)

Sun & friends are addressing the problem

- The Java Module System (JAM) is in development.

JMS already stands for Java Messaging Service.

Sun & friends are addressing the problem

- The Java Module System (JAM) is in development.
- JSR-277 (runtime system) and JSR-294 (developer's view) have started in 2006.

JMS already stands for Java Messaging Service.

JSR stands for Java Specification Request.

Sun & friends are addressing the problem

- The Java Module System (JAM) is in development.
- JSR-277 (runtime system) and JSR-294 (developer's view) have started in 2006.

Goals of JAM:

- provide high-level abstraction & remove JAR hell;

JMS already stands for Java Messaging Service.

JSR stands for Java Specification Request.

Sun & friends are addressing the problem

- The Java Module System (JAM) is in development.
- JSR-277 (runtime system) and JSR-294 (developer's view) have started in 2006.

Goals of JAM:

- provide high-level abstraction & remove JAR hell;
- be as simple as possible to use (have modules with packages);

JMS already stands for Java Messaging Service.

JSR stands for Java Specification Request.

Sun & friends are addressing the problem

- The Java Module System (JAM) is in development.
- JSR-277 (runtime system) and JSR-294 (developer's view) have started in 2006.

Goals of JAM:

- provide high-level abstraction & remove JAR hell;
- be as simple as possible to use (have modules with packages);
- be available to everyone automatically (put into the language);

JMS already stands for Java Messaging Service.

JSR stands for Java Specification Request.

Sun & friends are addressing the problem

- The Java Module System (JAM) is in development.
- JSR-277 (runtime system) and JSR-294 (developer's view) have started in 2006.

Goals of JAM:

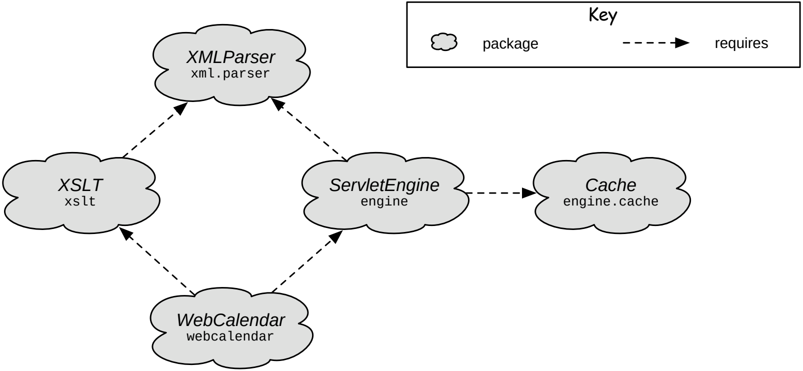
- provide high-level abstraction & remove JAR hell;
- be as simple as possible to use (have modules with packages);
- be available to everyone automatically (put into the language);
- be compatible with all Java sources (but only on JVM 7+).

JMS already stands for Java Messaging Service.

JSR stands for Java Specification Request.

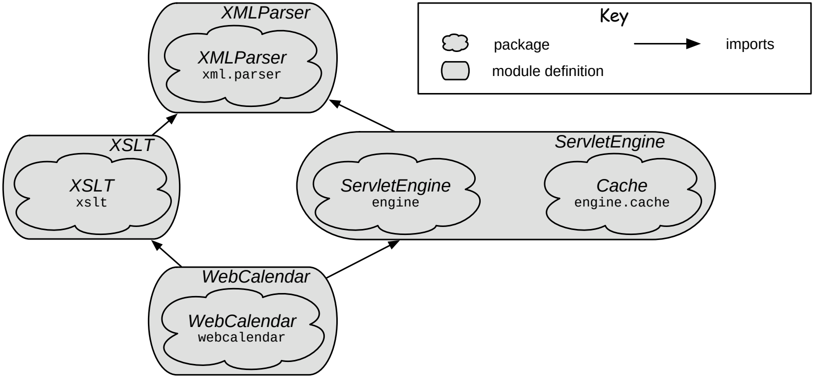
The example

Original Example with Packages



The example

Module definitions



The example

The source for the module definitions written in module files.

```
superpackage XMLParser {
  member xml.parser;
  export xml.parser.Parser;
}
```

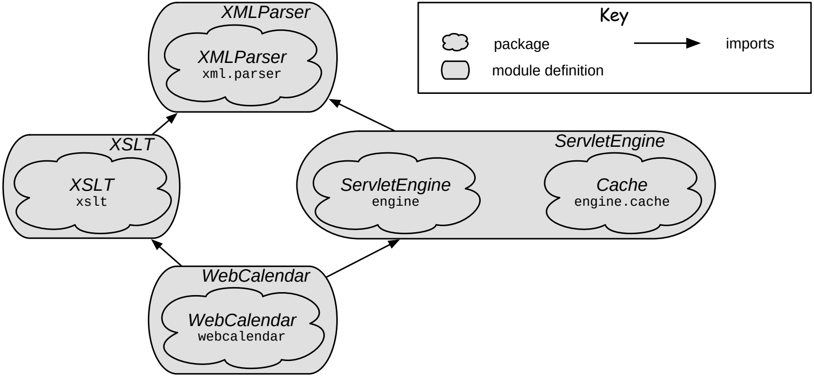
```
superpackage XSLT {
  member xslt;
  import XMLParser;
  export xslt.XSLTProcessor;
}
```

```
superpackage ServletEngine {
  member engine; member engine.cache;
  import XMLParser;
  export engine.ServletEngine;
}
```

```
superpackage WebCalendar {
  member webcalendar;
  import ServletEngine; import XSLT;
}
```

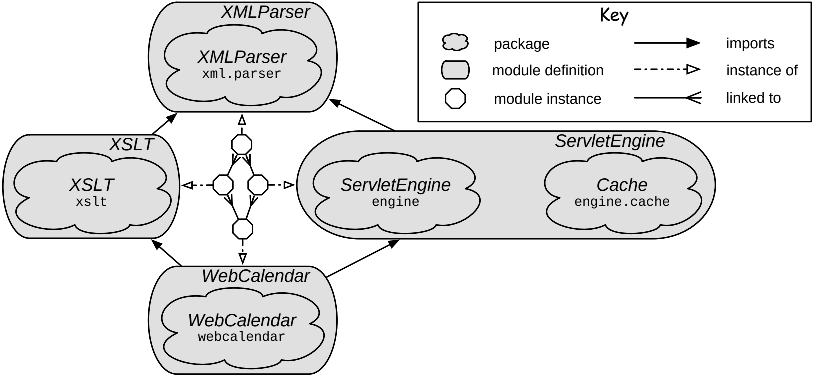
The example

Module definitions



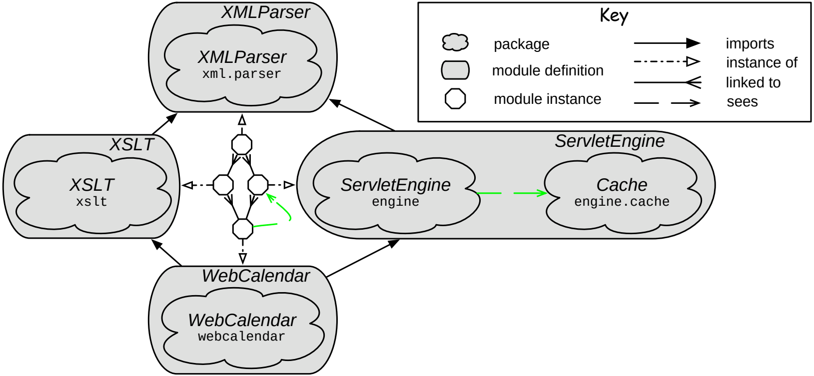
The example

Module definitions and their instances



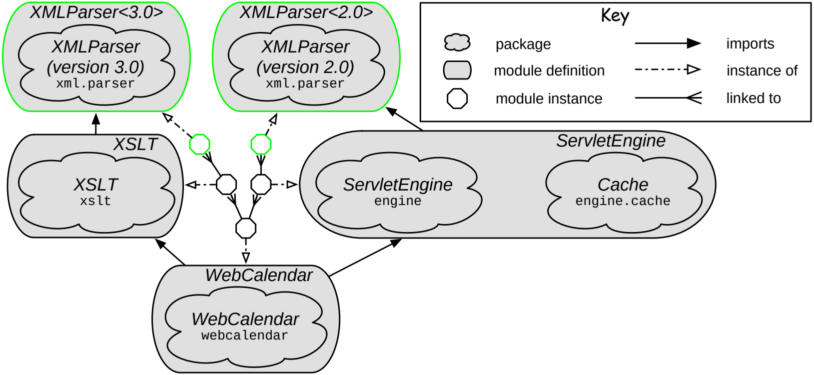
The example

High-level abstraction



The example

High-level abstraction No DLL/JAR hell

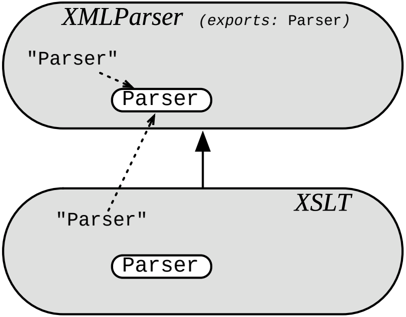


Summary of JAM's concepts

Module file	A file *naming* the packages. Also mentions what is imported/exported.
Module definition	(a.k.a. superpackage) A file *containing* the packages. Compiled from a module file and classes.
Module instance	A runtime instance of a module definition that is linked up to other module instances.
Repository	A runtime entity where an administrator can (un-)install or initialize module definitions.

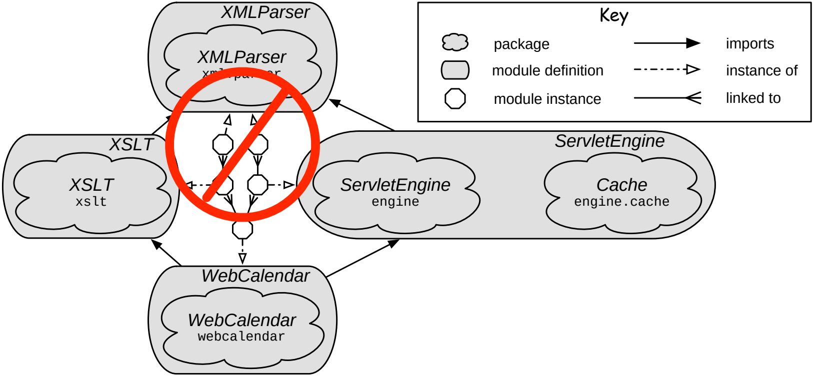
We discovered two major deficiencies with JAM

1. unintuitive class resolution



We discovered two major deficiencies with JAM

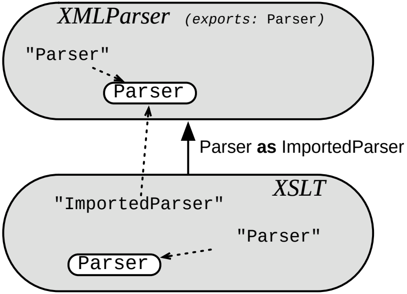
2. only a single instance of each module permitted



improved JAM
(iJAM)

Our proposals to fix JAM's problems

1. adapt class resolution & allow class renaming



superpackage XSLT {
.. **import** XMLParser **with** Parser **as** ImportedParser; ..
}

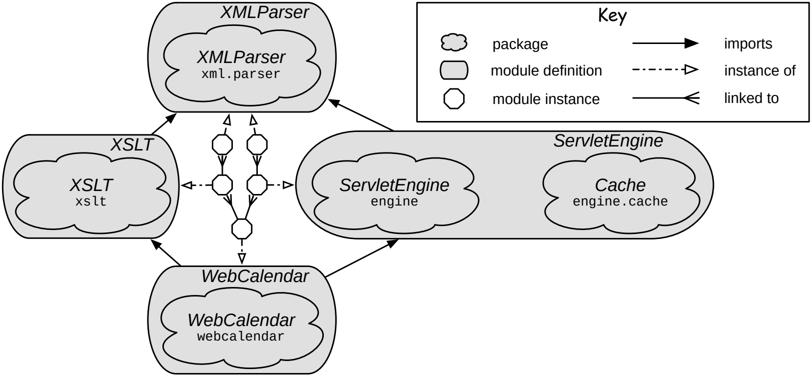
Our proposals to fix JAM's problems

2. allow the user to specify sharing policies

IMPORT OPTION	SHORT DESCRIPTION
import *m*	Uses JAM's default sharing policy. Can be overriden by **replicating** — see below.
import shared *m*	Explicitly requests a shared instance of *m*.
import own *m*	Requests a separate instance of *m*.
import *m* **as** *amn*	Requests an instance (shared under name *amn*).

ANNOTATION	SHORT DESCRIPTION
(no annotation)	Instantiation depends on the importer's policy.
replicating	Default import results in a new instance.
singleton	Always shares a single instance.

Our proposals to fix JAM's problems



replicating superpackage XMLParser {...}

Reviewing desired modularity properties

	ML	Java
abstraction	types, modules	access, interface
composition	inner, functors	inheritance, inner
static reuse	per file	per class (JAR)
dynamic reuse	no	per object
separate compilation	per file	no

	JAM (LJAM)	iJAM
abstraction	[Java], exports	[JAM], renaming
composition	[Java], imports	[JAM], renaming
static reuse	[Java], mod. def.s	[JAM]
dynamic reuse	[Java], forced per module	[Java], optional per module
separate compilation	no*	no*

*SMARTJAVAMOD [FTfJP'05] uses *compositional constraints* [POPL'05] to provide separate compilation for JAM-like modules.

Formalizations & proofs

What we did

- designed & formalized the core of JAM (40% JSR-277 & 80% JSR-294) on top of LJ — obtaining LJAM.
  (where JSRs were ambiguous/incomplete we made reasonable choices and discussed alternatives);

What we did

- designed & formalized the core of JAM (40% JSR-277 & 80% JSR-294) on top of LJ — obtaining LJAM. *(where JSRs were ambiguous/incomplete we made reasonable choices and discussed alternatives)*;
- designed & formalized fixes to JAM — obtaining iJAM;

What we did

- designed & formalized the core of JAM (40% JSR-277 & 80% JSR-294) on top of LJ — obtaining LJAM. *(where JSRs were ambiguous/incomplete we made reasonable choices and discussed alternatives)*;
- designed & formalized fixes to JAM — obtaining iJAM;
- proved type soundness (in Isabelle/HOL) for LJ, for LJAM, and for iJAM.

What we did

- designed & formalized the core of JAM (40% JSR-277 & 80% JSR-294) on top of LJ — obtaining LJAM. *(where JSRs were ambiguous/incomplete we made reasonable choices and discussed alternatives)*;
- designed & formalized fixes to JAM — obtaining iJAM;
- proved type soundness (in Isabelle/HOL) for LJ, for LJAM, and for iJAM.
- tools used:

Isabelle	A tool for writing computer-verified maths. (essential for this scale).
Ott	A tool for writing definitions of PLs [ICFP'07]: ASCII source ⟶ LaTeX/Coq/HOL/Isabelle

Why?

- The JAM JSRs are written in English only (+180 pages), including statements like:

> *"There is at most one module instance instantiated from each module definition per repository instance. A module definition can have multiple module instances through multiple repository instances."*

Why?

- The JAM JSRs are written in English only (+180 pages), including statements like:

  > *"There is at most one module instance instantiated from each module definition per repository instance. A module definition can have multiple module instances through multiple repository instances."*

- Formalisation allows precise discussion.

Why?

- The JAM JSRs are written in English only (+180 pages), including statements like:

  > *"There is at most one module instance instantiated from each module definition per repository instance. A module definition can have multiple module instances through multiple repository instances."*

- Formalisation allows precise discussion.
- Subtle bugs are found early.

Why?

- The JAM JSRs are written in English only (+180 pages), including statements like:

  > *"There is at most one module instance instantiated from each module definition per repository instance. A module definition can have multiple module instances through multiple repository instances."*

- Formalisation allows precise discussion.
- Subtle bugs are found early.
- When available, one can (dis-)prove properties about the specification.

What does it mean *to formalise*?

1. Read the given prose carefully;

What does it mean *to formalise*?

1. Read the given prose carefully;
2. identify the key concepts;

What does it mean *to formalise*?

1. Read the given prose carefully;
2. identify the key concepts;
3. associate those with mathematical entities;

What does it mean *to formalise*?

1. Read the given prose carefully;
2. identify the key concepts;
3. associate those with mathematical entities;
4. give mathematical rules that relate these entities according to the intended semantics.

What does it mean *to formalise*?

1. Read the given prose carefully;
2. identify the key concepts;
3. associate those with mathematical entities;
4. give mathematical rules that relate these entities according to the intended semantics.

An example 'statement reduction' ($config \longrightarrow config'$) rule:

$$\frac{L\,(x) = oid \qquad H\,(oid,\ f\,) = v}{(P,\ L,\ H,\ var = x\,.\,f\ ;\ \overline{s_l}^{\,l}) \longrightarrow (P,\ L\,[var \mapsto v],\ H,\ \overline{s_l}^{\,l})}\ \textsc{R\_FIELD\_READ}$$

Formalization overview

Name	Description	Ott LOD	No. of rules
LJ	an imperative fragment of Java	1381	85
LJAM	formalisation of core JAM	2502	164
iJAM	LJAM with fixes	2671	180

Results

We proved type-soundness for all three formalisations by proving:

Theorem (Progress)

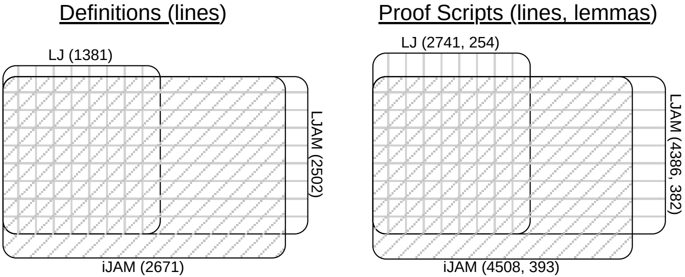
$$\Gamma \vdash (P, L, H, \overline{s}) \ \wedge \ \overline{s} \neq [] \implies \exists \mathit{config}. \ (P, L, H, \overline{s}) \longrightarrow \mathit{config}$$

and

Theorem (Type Preservation)

$$\Gamma \vdash \mathit{config} \wedge (\mathit{config} \longrightarrow \mathit{config}' \vee \mathit{config} \xrightarrow{a} \mathit{config}') \implies \exists \Gamma'. \ \Gamma \subseteq_m \Gamma' \ \wedge \ \Gamma' \vdash \mathit{config}'$$

Definition & proof script reuse



Relative area corresponds to relative number of lines.

Implementation

We wrote a proof-of-concept implementation in Java, which can closely follow the semantics of either LJAM or iJAM.

In particular, we implemented:

- module files (with JavaCC);
- repositories;
- module definitions;
- module instances;
- the module initialization mechanism;
- LJAM's and iJAM's class resolution (with classloaders): *each module is a classloader, which delegates class resolution according to the class lookup semantics.*

Conclusion

Conclusions

- **formalization is useful**: concise, unambiguous def.; enables precise discussion of design; early bug detection!

Conclusions

- **formalization is useful**: concise, unambiguous def.; enables precise discussion of design; early bug detection!
- using the formalization we **found & fixed** some problems;

Conclusions

- **formalization is useful**: concise, unambiguous def.; enables precise discussion of design; early bug detection!
- using the formalization we **found & fixed** some problems;
- this work was completed relatively quickly, on the timescale of language evolution process;

Conclusions

- **formalization is useful**: concise, unambiguous def.; enables precise discussion of design; early bug detection!
- using the formalization we **found & fixed** some problems;
- this work was completed relatively quickly, on the timescale of language evolution process;
- formalizing real PLs and their updates now seems **feasible** (and **recommended**)!

Future plans

- extract implementation of LJ/LJAM/iJAM from its Isabelle definition;
- apply developed ideas to a large project.

More information

For more information, publications, documentation, definitions, and Java implementation of LJAM/iJAM, see

`http://www.cl.cam.ac.uk/~rs456/`

Papers:

- Strniša, R., Sewell, P., Parkinson, M.: The Java Module System: Core Design and Semantics Definition. In: Proc. of OOPSLA'07
- Strniša, R.: Fixing the Java Module System, in Theory and in Practice. *Submitted to FTfJP'08 (ECOOP).*