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?
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)

```ml
module Foo = struct
    type t = int
    let v = 5
end

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

module type View = sig
    type t
    val v : t
end

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

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)
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.
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.

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)
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.

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)
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)

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)
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

Key:
- package
- requires

XMLParser
xml.parser

XSLT
xslt

ServletEngine
engine

Cache
google

Key

WebCalendar
webcalendar

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)
The example

Module definitions

```
XMLParser
  xml.parser

XSLT
  xslt

ServletEngine
  engine

Cache
  engine.cache

WebCalendar
  webcalendar
```

Key

- package
- module definition

imports

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)
The example

The source for the module definitions written in module files.

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

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

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

```java
superpackage WebCalendar {
    member webcalendar;
    import ServletEngine; import XSLT;
}
```
The example

Module definitions

- XMLParser
  - xml.parser
- XSLT
  - xslt
- WebCalendar
  - webcalendar
- ServletEngine
  - engine
  - engine.cache
- Cache
- Key:
  - package
  - module definition
  - imports

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)
The example

Module definitions and their instances

XMLParser

XSLT

WebCalendar

ServletEngine

Cache

diagram of module definitions and instances:

- XMLParser
- XSLT
- WebCalendar
- ServletEngine
- Cache

Key:
- package
- module definition
- module instance
- linked to
- imports
- instance of
The example

High-level abstraction

XMLParser

XMLParser
xml.parser

XSLT

XSLT
xslt

WebCalendar

WebCalendar
webcalendar

ServletEngine

ServletEngine
engine

XSLT ServletEngine
WebCalendar
WebCalendar
webcalendar
ServletEngine
gen
XSLT
xslt
Cache
gen.cache

Key

- package
- module definition
- module instance
- instance of
- linked to
- sees

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)
The example

High-level abstraction

No DLL/JAR hell

Key

- package
- module definition
- instance of
- module instance
- linked to

XMLParser<3.0>

XMLParser
(version 3.0)
xml.parser

XMLParser<2.0>

XMLParser
(version 2.0)
xm.parser

ServletEngine

Cache
engine.cache

WebCalendar

WebCalendar
webcalendar

XSLT
xslt

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)
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

```
XMLParser (exports: Parser)
"Parser"
Parser

XSLT
"Parser"
Parser
```

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)

Adding a Module System to Java
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

<table>
<thead>
<tr>
<th>IMPORT OPTION</th>
<th>SHORT DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>import ( m )</td>
<td>Uses JAM’s default sharing policy. Can be overridden by replicating — see below.</td>
</tr>
<tr>
<td>import shared ( m )</td>
<td>Explicitly requests a shared instance of ( m ).</td>
</tr>
<tr>
<td>import own ( m )</td>
<td>Requests a separate instance of ( m ).</td>
</tr>
<tr>
<td>import ( m ) as ( amn )</td>
<td>Requests an instance (shared under name ( amn )).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ANNOTATION</th>
<th>SHORT DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>(no annotation)</td>
<td>Instantiation depends on the importer’s policy.</td>
</tr>
<tr>
<td>replicating</td>
<td>Default import results in a new instance.</td>
</tr>
<tr>
<td>singleton</td>
<td>Always shares a single instance.</td>
</tr>
</tbody>
</table>
Our proposals to fix JAM’s problems

replicating superpackage XMLParser {...}
### Reviewing desired modularity properties

<table>
<thead>
<tr>
<th></th>
<th>ML</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstraction</td>
<td>types, modules</td>
<td>access, interface</td>
</tr>
<tr>
<td>composition</td>
<td>inner, functors</td>
<td>inheritance, inner</td>
</tr>
<tr>
<td>static reuse</td>
<td>per file</td>
<td>per class (JAR)</td>
</tr>
<tr>
<td>dynamic reuse</td>
<td>no</td>
<td>per object</td>
</tr>
<tr>
<td>separate compilation</td>
<td>per file</td>
<td>no</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>JAM (LJAM)</th>
<th>iJAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstraction</td>
<td>[Java], exports</td>
<td>[JAM], renaming</td>
</tr>
<tr>
<td>composition</td>
<td>[Java], imports</td>
<td>[JAM], renaming</td>
</tr>
<tr>
<td>static reuse</td>
<td>[Java], mod. def.s</td>
<td>[JAM]</td>
</tr>
<tr>
<td>dynamic reuse</td>
<td>[Java], forced per module</td>
<td>[Java], optional per module</td>
</tr>
<tr>
<td>separate compilation</td>
<td>no*</td>
<td>no*</td>
</tr>
</tbody>
</table>

*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 $\rightarrow$ \LaTeX/Coq/HOL/Isabelle
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' (\( \text{config} \rightarrow \text{config}' \)) rule:

\[
\begin{align*}
L(x) &= \text{oid} \\
H(\text{oid}, f) &= v \\
(P, L, H, \text{var} = x \cdot f ; \overline{s}_i) &\rightarrow (P, L[\text{var} \mapsto v], H, \overline{s}_i') \\
\end{align*}
\]

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)
## Formalization overview

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Ott LOD</th>
<th>No. of rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>LJ</td>
<td>an imperative fragment of Java</td>
<td>1381</td>
<td>85</td>
</tr>
<tr>
<td>LJAM</td>
<td>formalisation of core JAM</td>
<td>2502</td>
<td>164</td>
</tr>
<tr>
<td>iJAM</td>
<td>LJAM with fixes</td>
<td>2671</td>
<td>180</td>
</tr>
</tbody>
</table>

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)  
Adding a Module System to Java
We proved type-soundness for all three formalisations by proving:

**Theorem (Progress)**

\[ \Gamma \vdash (P, L, H, \overline{s}) \land \overline{s} \neq [] \quad \Rightarrow \quad \exists \text{config.} \quad (P, L, H, \overline{s}) \rightarrow \text{config} \]

and

**Theorem (Type Preservation)**

\[ \Gamma \vdash \text{config} \land (\text{config} \rightarrow \text{config}' \lor \text{config} \xrightarrow{a} \text{config} ') \quad \Rightarrow \quad \exists \Gamma'. \quad \Gamma \subseteq_m \Gamma' \land \Gamma' \vdash \text{config}' \]
Definition & proof script reuse

Definitions (lines)
- LJ (1381)
- LJAM (2502)
- iJAM (2671)

Proof Scripts (lines, lemmas)
- LJ (2741, 254)
- LJAM (4386, 382)
- iJAM (4508, 393)

Relative area corresponds to relative number of lines.

R. Strniša (Joint work with P. Sewell and M.J. Parkinson)
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: