The Ateji™ PX 1.2 Manual (Revision 22)

Patrick Viry

The Ateji PX 1.2 Manual (Revision 22)

Patrick Viry

This manual was prepared for Ateji PX 1.2 plugin (September 2011). This manual is frequently updated to reflect changes in the product. The latest version can always be found at: http://www.ateji.com/px/documentation.html

Table of Contents

Chapter 1. Download and Installation of Ateji PX for Java

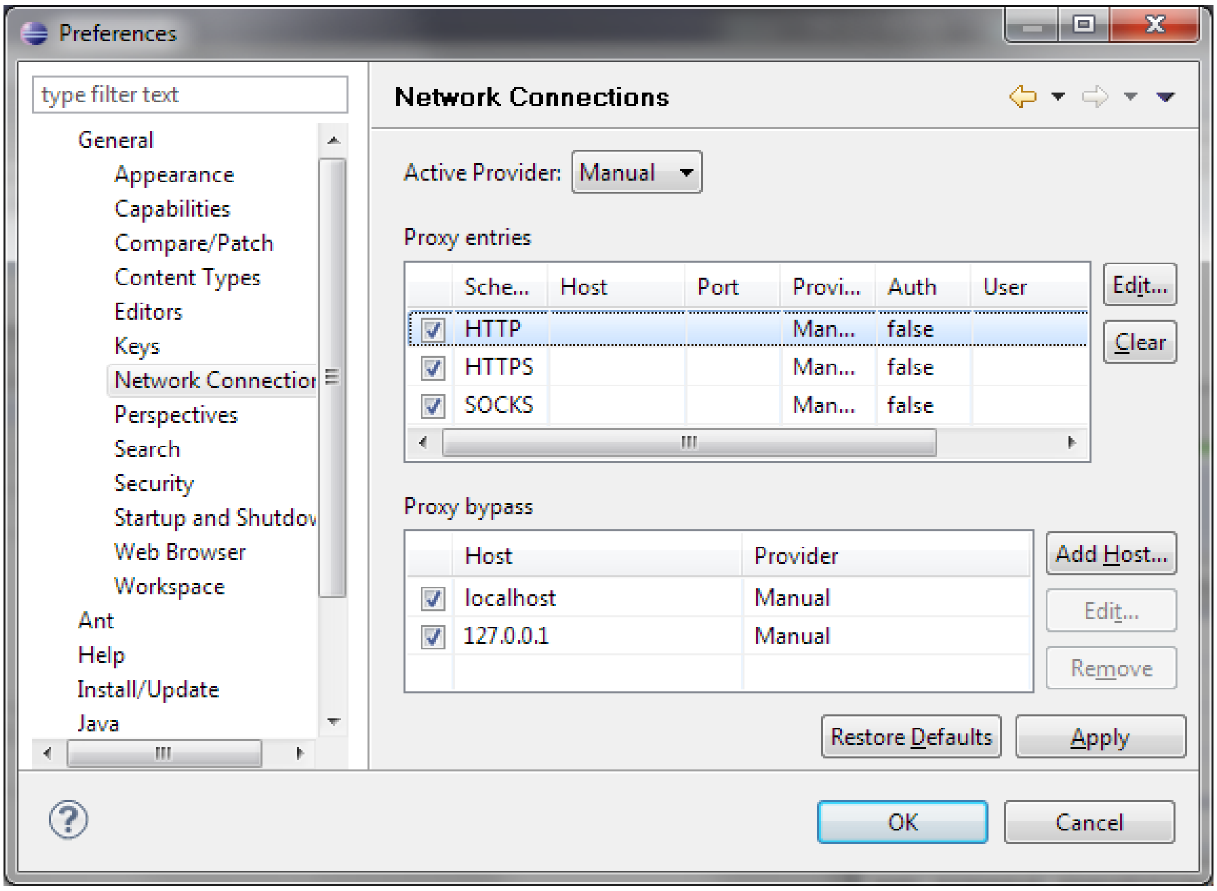
If you do not have a direct Internet access, you must configure Eclipse with your network proxy information. Otherwise you may skip to the next sections.

Ateji PX requires Eclipse SDK version 3.4 (Ganymede), version 3.5 (Galileo), version 3.6 (Helios) or version 3.7 (Indigo).

1. Using a network proxy

Follow this procedure to configure the proxy settings:

- Select `Window > Preferences`
- In the left pane, select `General > Network Connections` and then select `Manual` proxy configuration.
- Configure the settings with your web proxy information.



Network Connections

Active Provider: Manual

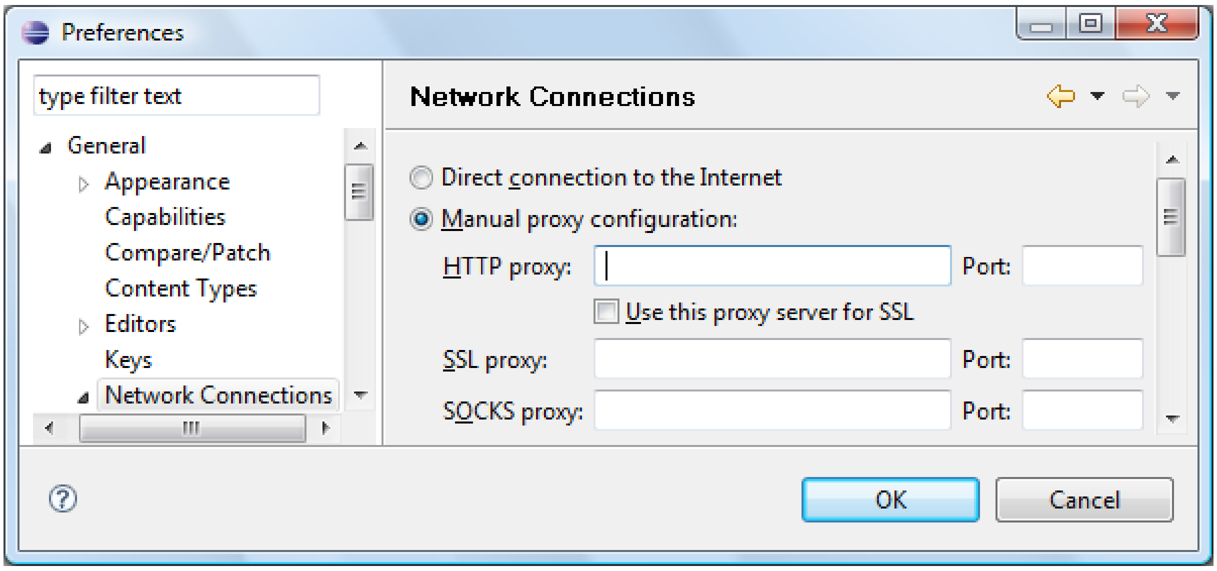
Proxy entries

	Sche...	Host	Port	Provi...	Auth	User
✓	HTTP			Man...	false	
✓	HTTPS			Man...	false	
✓	SOCKS			Man...	false	

Proxy bypass

	Host	Provider
✓	localhost	Manual
✓	127.0.0.1	Manual

Eclipse 3.5/3.6/3.7 Network Connections preference page



Network Connections

Direct connection to the Internet [✓ Manual proxy configuration:]

HTTP proxy: Port:

Use this proxy server for SSL

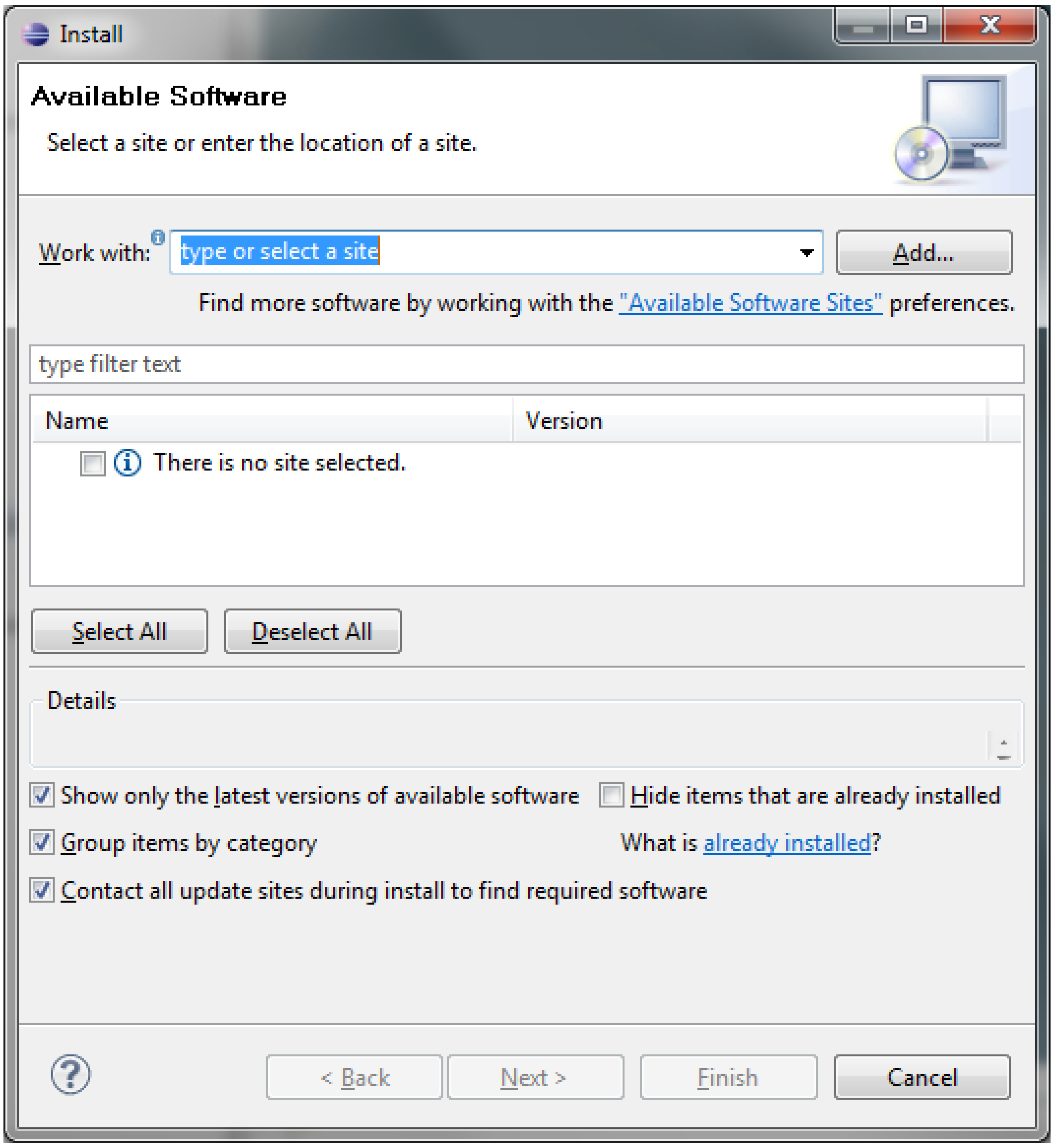
SSL proxy: Port:

SOCKS proxy: Port:

Eclipse 3.4 (Ganymede) Network Connections preference page

2. Instructions for Eclipse 3.5/3.6/3.7

Select `Help > Install New Software...` to open the Install dialog:


Install

Available Software

Select a site or enter the location of a site.

Work with: type or select a site [Add...]

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☐ There is no site selected.	

[Select All] [Deselect All]

Details

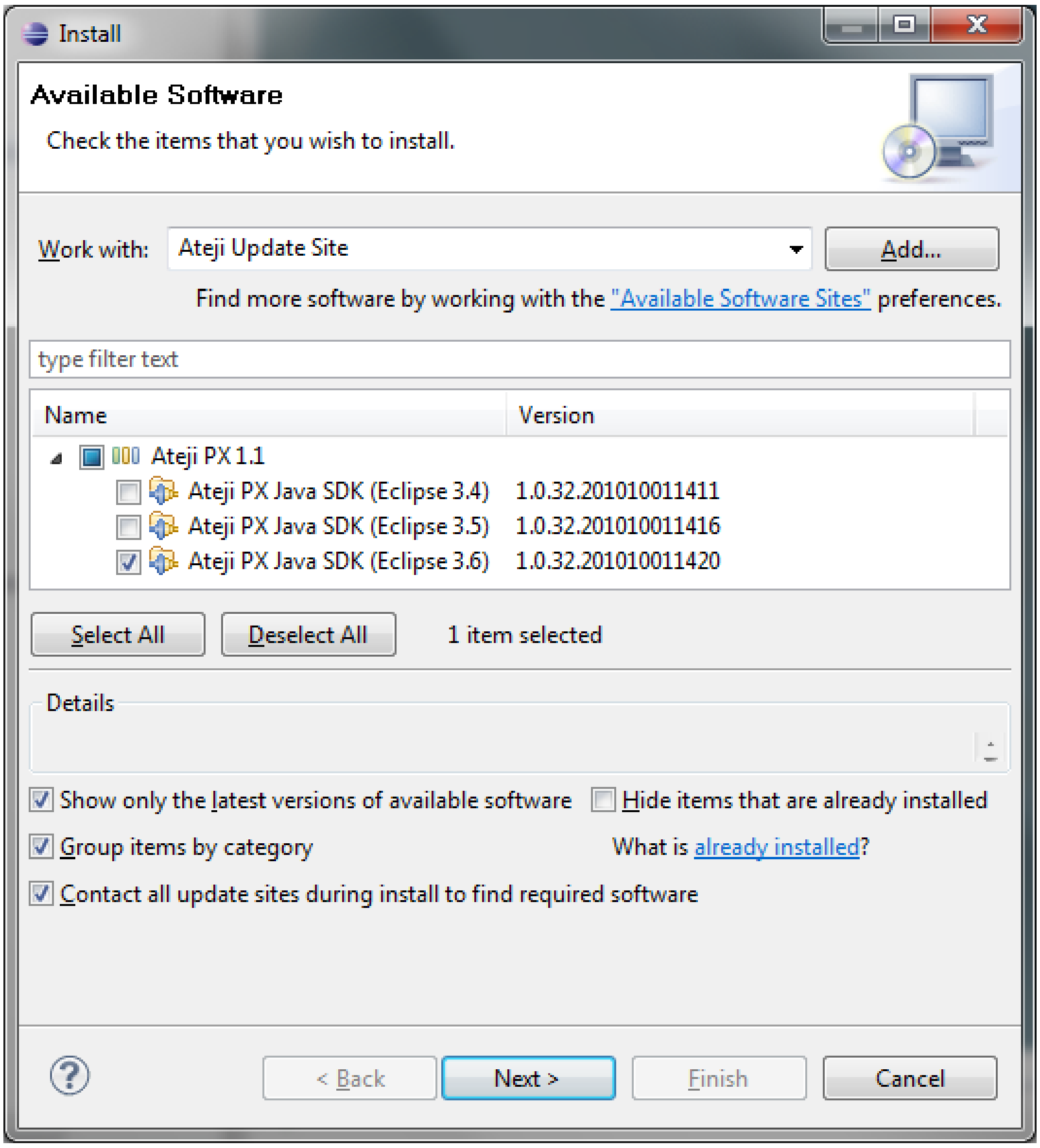
☑ Show only the latest versions of available software ☐ Hide items that are already installed

☑ Group items by category What is already installed?

☑ Contact all update sites during install to find required software

[< Back] [Next >] [Finish] [Cancel]

Enter `http://www.ateji.com/px/eclipse` in the `Work with` field and hit the enter key to validate the input.


Install

Available Software

Check the items that you wish to install.

Work with: Ateji Update Site

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
Ateji PX 1.1	
☐ Ateji PX Java SDK (Eclipse 3.4)	1.0.32.201010011411
☐ Ateji PX Java SDK (Eclipse 3.5)	1.0.32.201010011416
☑ Ateji PX Java SDK (Eclipse 3.6)	1.0.32.201010011420

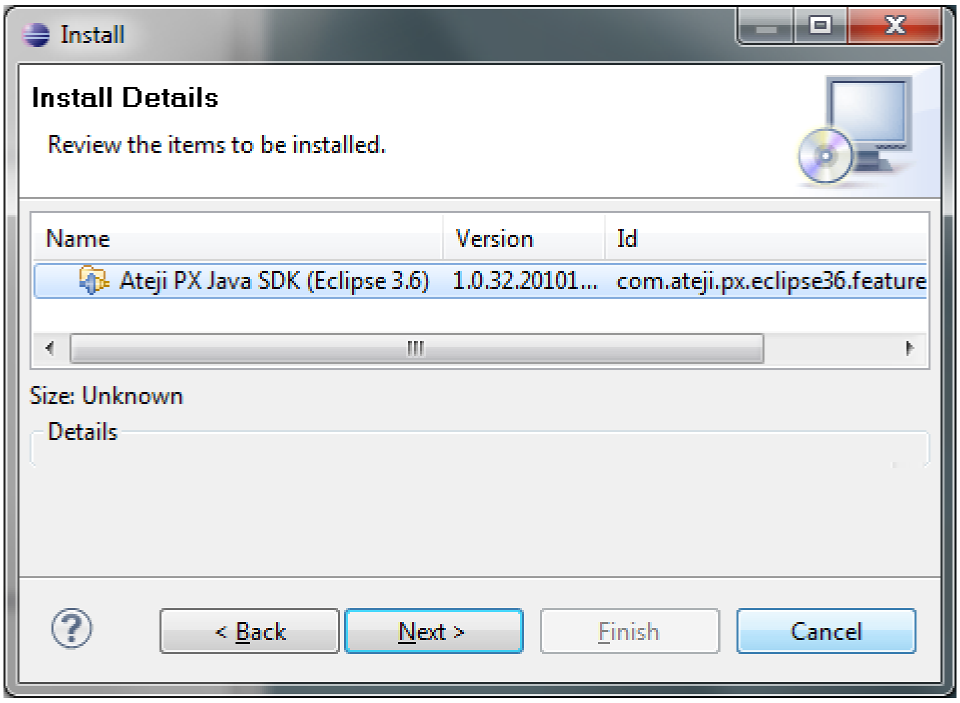
1 item selected

Details

☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☑ Contact all update sites during install to find required software

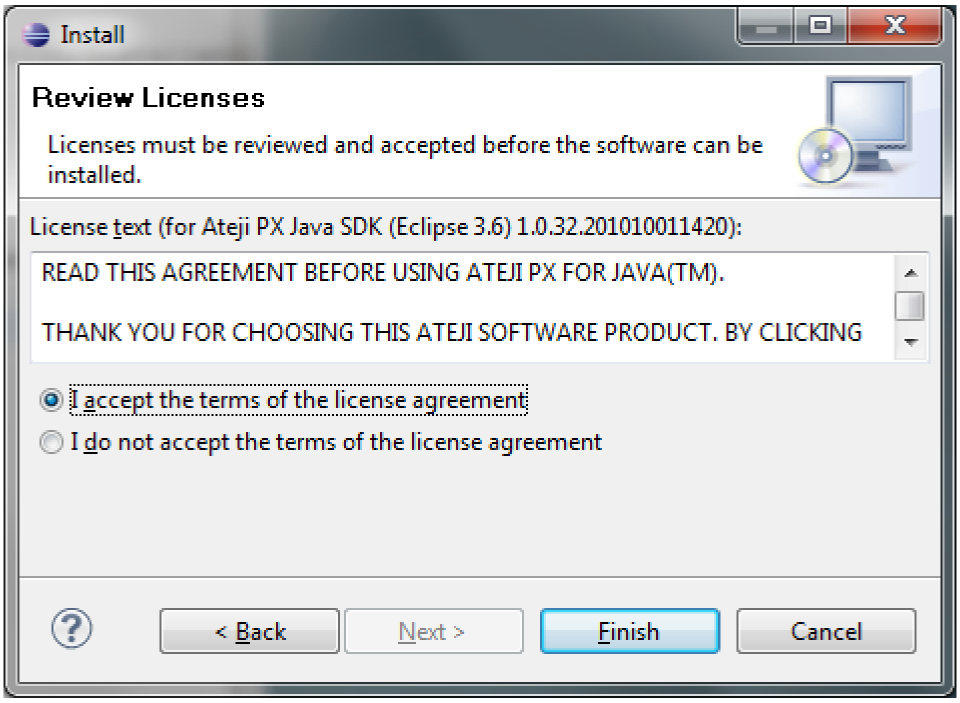
The update site contains Eclipse plug-ins for multiple Eclipse versions. Unfold the `Ateji PX 1.2` category and select `Ateji PX Java SDK (Eclipse 3.5)` if you are using Eclipse 3.5 (Galileo), `Ateji PX Java SDK (Eclipse 3.6)` if you are using Eclipse 3.6 (Helios), or `Ateji PX Java SDK (Eclipse 3.7)` if you are using Eclipse 3.7 (Indigo), then click on `Next`.

Once the dependencies are resolved and verified a confirmation dialog appears:


Install
Install Details
Review the items to be installed.
Name
Version
Id
Ateji PX Java SDK (Eclipse 3.6)
1.0.32.20101...
com.ateji.px.eclipse36.feature
Size: Unknown
Details
< Back
Next >
Finish
Cancel

Press `Next` if you are using Eclipse 3.6/3.7 or `Finish` if you are using Eclipse 3.5 (Galileo) to proceed with the installation.

In the following dialog, you can review the licence agreement.


Install
Review Licenses
Licenses must be reviewed and accepted before the software can be installed.
License text (for Ateji PX Java SDK (Eclipse 3.6) 1.0.32.201010011420):
READ THIS AGREEMENT BEFORE USING ATEJI PX FOR JAVA(TM).
THANK YOU FOR CHOOSING THIS ATEJI SOFTWARE PRODUCT. BY CLICKING
I accept the terms of the license agreement
I do not accept the terms of the license agreement
< Back
Next >
Finish
Cancel

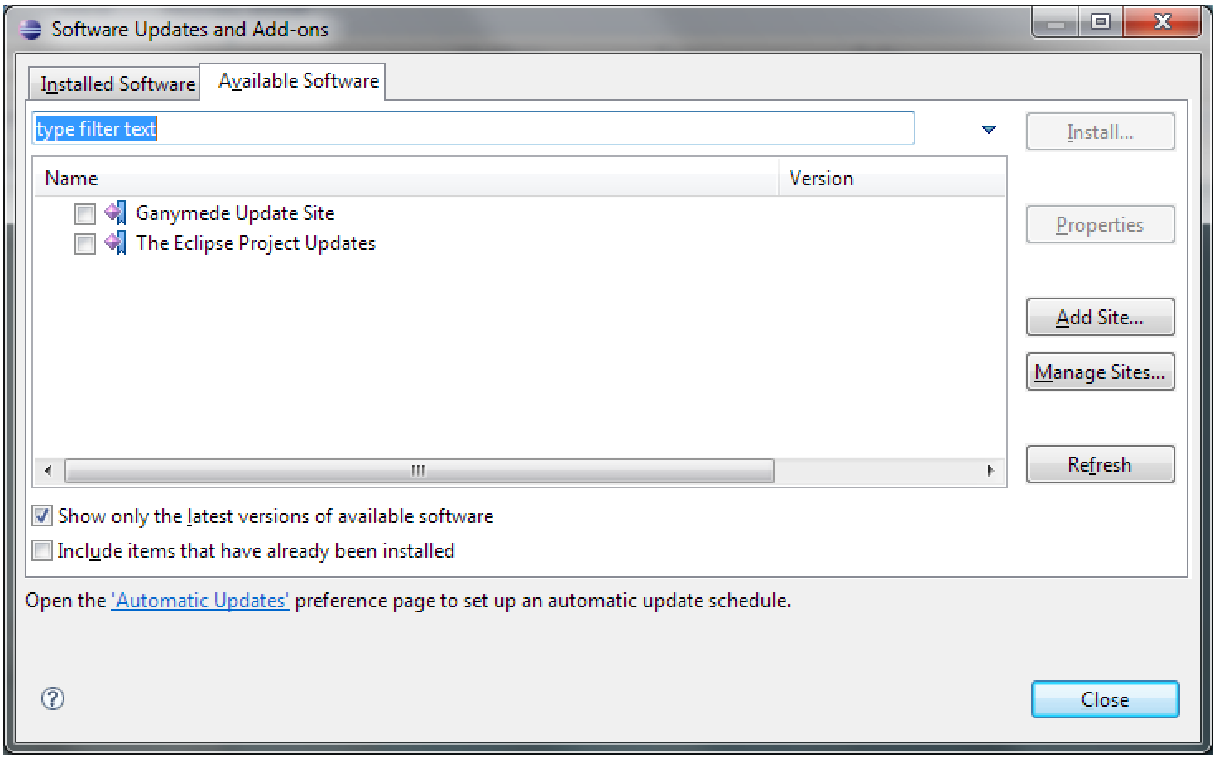
Select `I accept the terms of the license agreement` and press `Finish` to start the download.

Press `OK` when asked confirmation to install the unsigned Ateji PX plug-in.

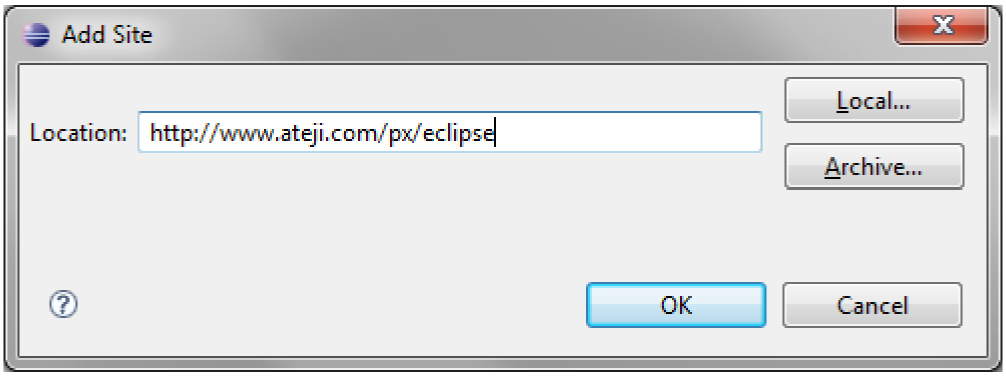
Choose to restart Eclipse immediately when proposed to do so.

3. Instructions for Eclipse 3.4 (Ganymede)

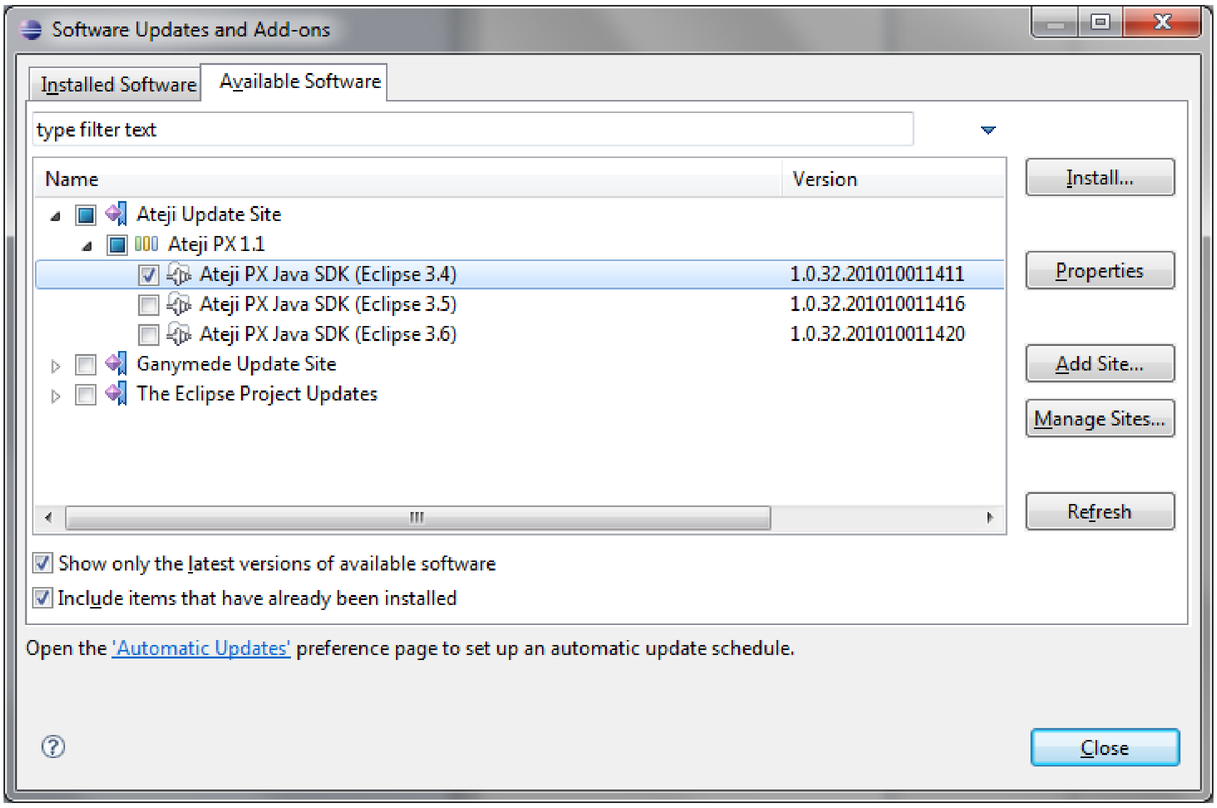
Select `Help > Software Updates...` and ensure that the `Available Software` tab is selected:


Software Updates and Add-ons
Installed Software
Available Software
type filter text
Install...
Name
Version
Ganymede Update Site
The Eclipse Project Updates
Properties
Add Site...
Manage Sites...
Refresh
Show only the latest versions of available software
Include items that have already been installed
Open the 'Automatic Updates' preference page to set up an automatic update schedule.
Close

Press `Add Site...` and enter `http://www.ateji.com/px/eclipse` in the `Location` field:



Click `OK` to close the dialog. The Ateji update site appears in the list of available software sites:



Software Updates and Add-ons

Installed Software | Available Software

type filter text

Name	Version
☐ Ateji Update Site	
☐ Ateji PX 1.1	
☑ Ateji PX Java SDK (Eclipse 3.4)	1.0.32.201010011411
☐ Ateji PX Java SDK (Eclipse 3.5)	1.0.32.201010011416
☐ Ateji PX Java SDK (Eclipse 3.6)	1.0.32.201010011420
☐ Ganymede Update Site	
☐ The Eclipse Project Updates	

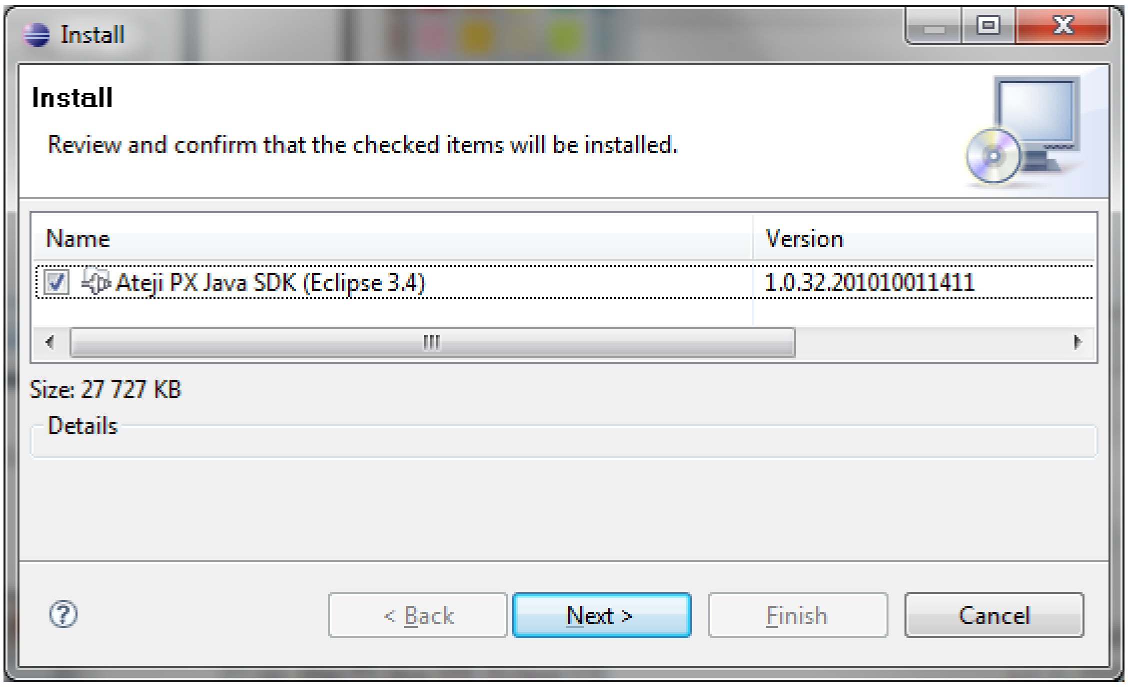
☑ Show only the latest versions of available software

☑ Include items that have already been installed

Open the 'Automatic Updates' preference page to set up an automatic update schedule.

The update site contains Eclipse plug-ins for multiple Eclipse versions. Unfold the `Ateji PX 1.2` category and select `Ateji PX Java SDK (Eclipse 3.4)` then click on `Install...`.

Once the dependencies are resolved and verified the Install dialog appears:



Install

Install

Review and confirm that the checked items will be installed.

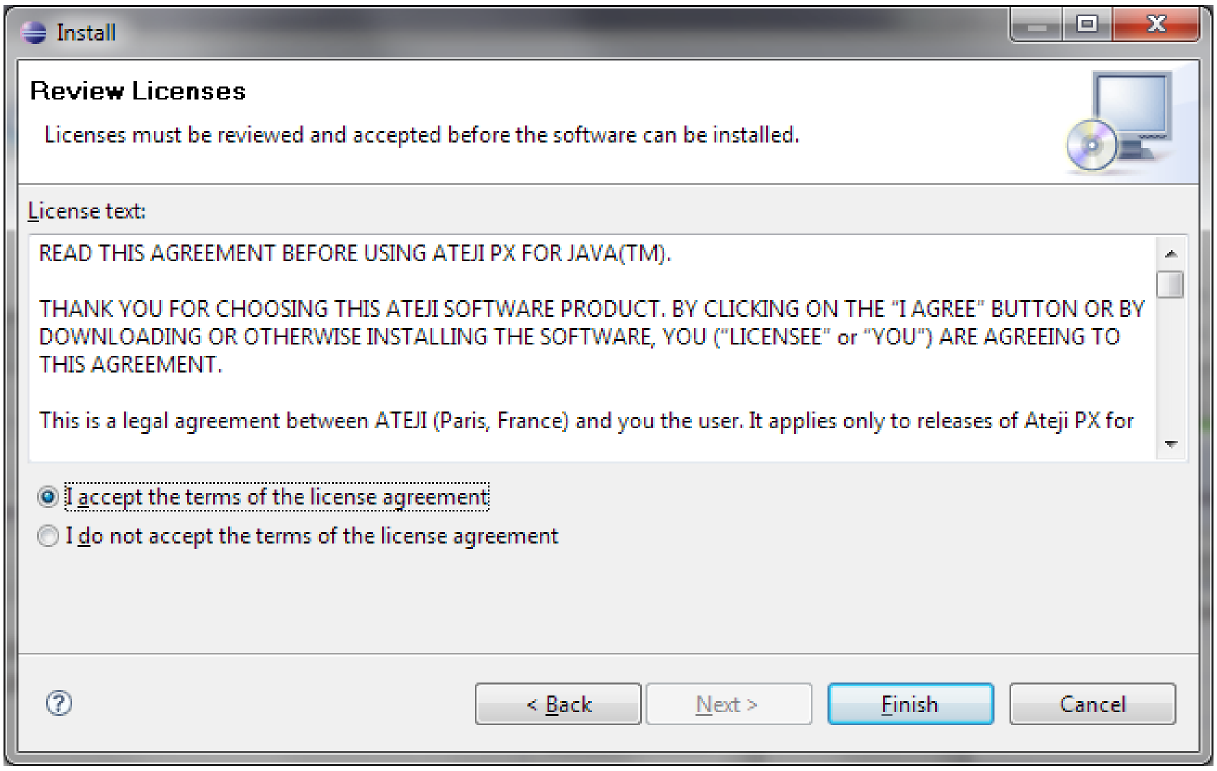
Name	Version
☑ Ateji PX Java SDK (Eclipse 3.4)	1.0.32.201010011411

Size: 27 727 KB

Details

Press `Next` to proceed with the installation.

In the following dialog, you can review the licence agreement.



Licenses must be reviewed and accepted before the software can be installed.

License text:

READ THIS AGREEMENT BEFORE USING ATEJI PX FOR JAVA(TM).

THANK YOU FOR CHOOSING THIS ATEJI SOFTWARE PRODUCT. BY CLICKING ON THE "I AGREE" BUTTON OR BY DOWNLOADING OR OTHERWISE INSTALLING THE SOFTWARE, YOU ("LICENSEE" or "YOU") ARE AGREEING TO THIS AGREEMENT.

This is a legal agreement between ATEJI (Paris, France) and you the user. It applies only to releases of Ateji PX for

[✓ I accept the terms of the license agreement] I do not accept the terms of the license agreement

Select `I accept the terms of the license agreement` and press `Finish` to start downloading the plug-in.

Press `OK` when asked confirmation to install the unsigned Ateji PX plug-in.

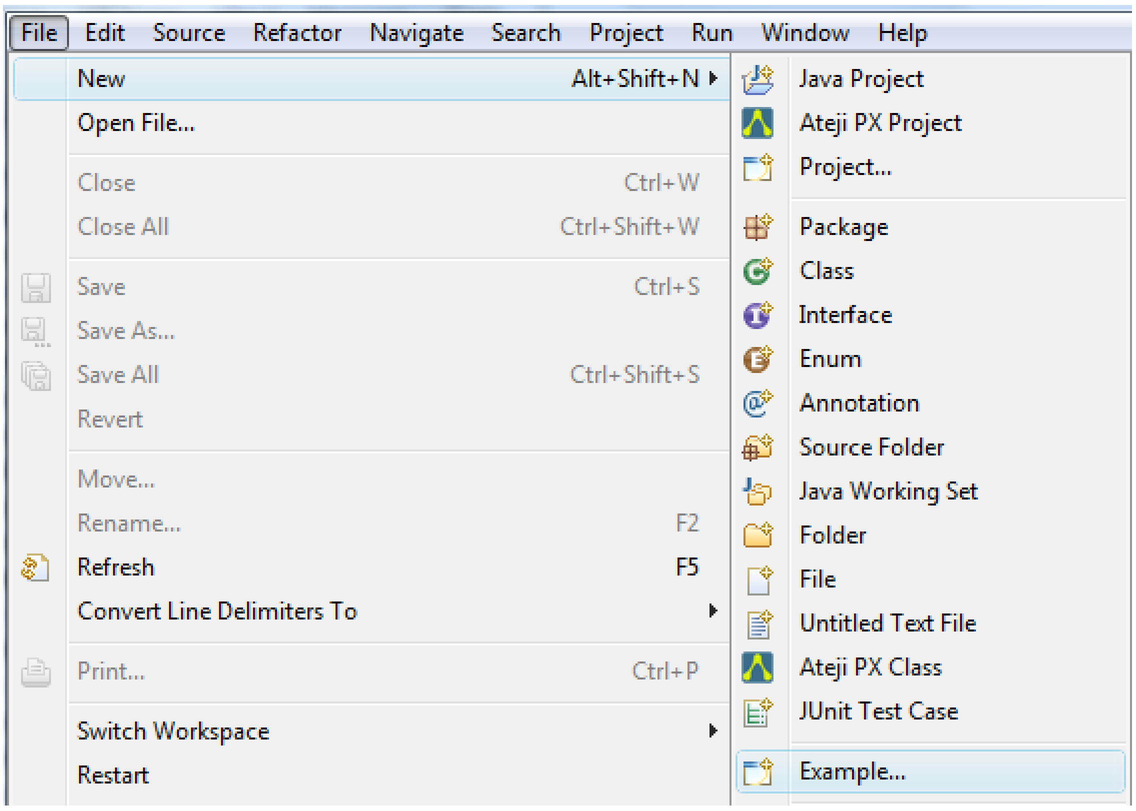
Press `Yes` when asked whether you want to restart immediately.

4. Samples installation

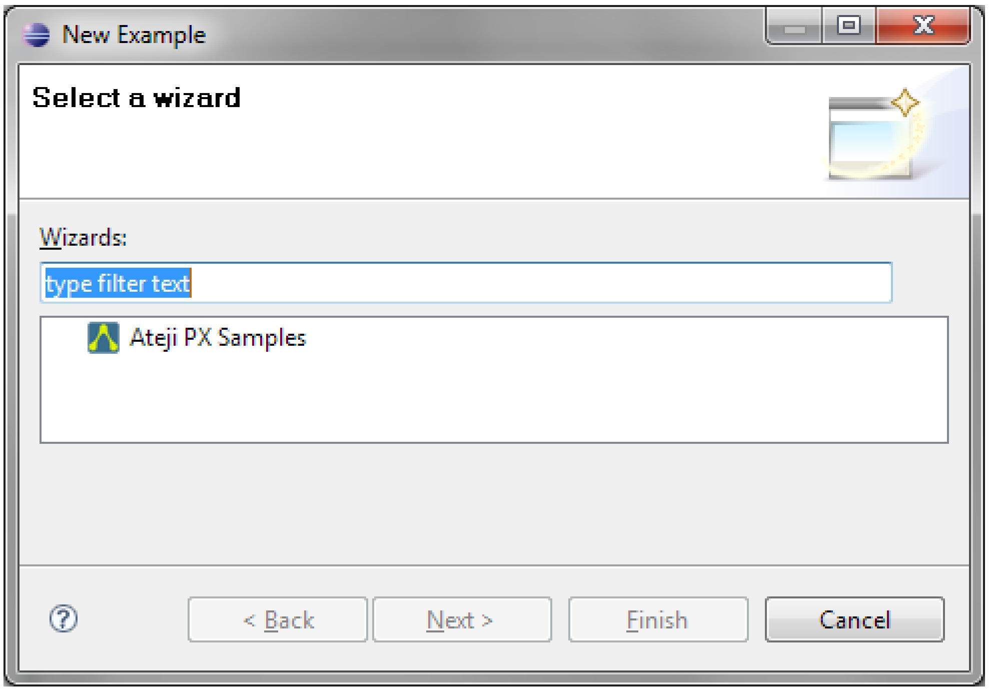
Ateji provides a comprehensive samples library to help you. We strongly recommend that you write your first Ateji PX programs by starting from a sample and adapting it to your needs.

Follow this procedure to install a sample project in your workspace:

Select `File > New > Example...`



Select `Ateji PX Samples` and click on `Finish`



After completion of the wizard the Ateji PX Samples are available in your workspace.

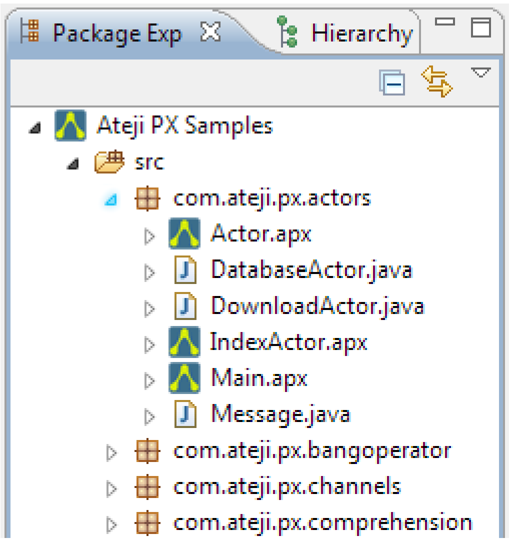
Chapter 2. The Ateji PX plug-in

Ateji PX is provided as an Eclipse plug-in. This chapter will explain how to use it.

1. Overview

As your first Ateji PX project, we recommend you to install the included Ateji PX samples. See chapter "Installation" for details.

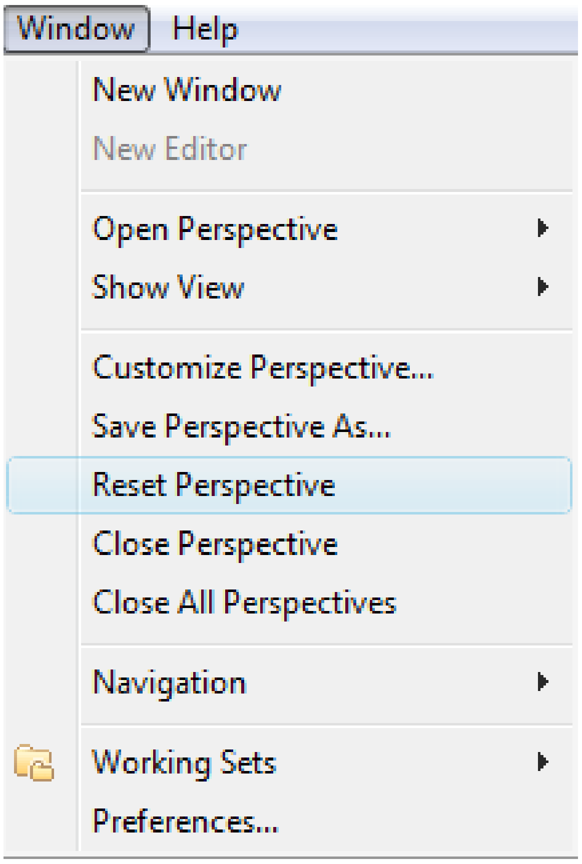
When the Ateji PX plug-in is properly installed and enabled, Ateji PX projects and Ateji PX source files (.apx files) appear with the Ateji icon:



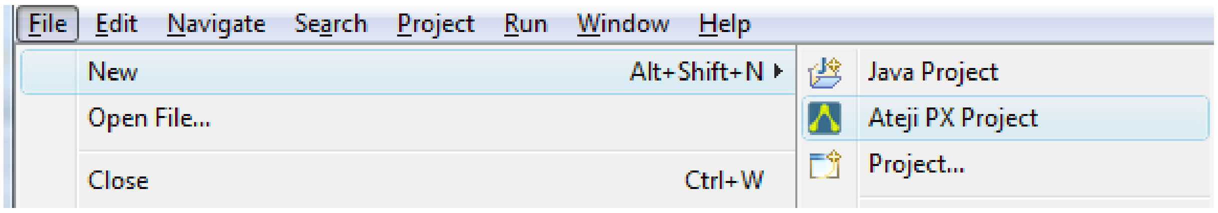
The following sections will explain how to write your own parallel code.

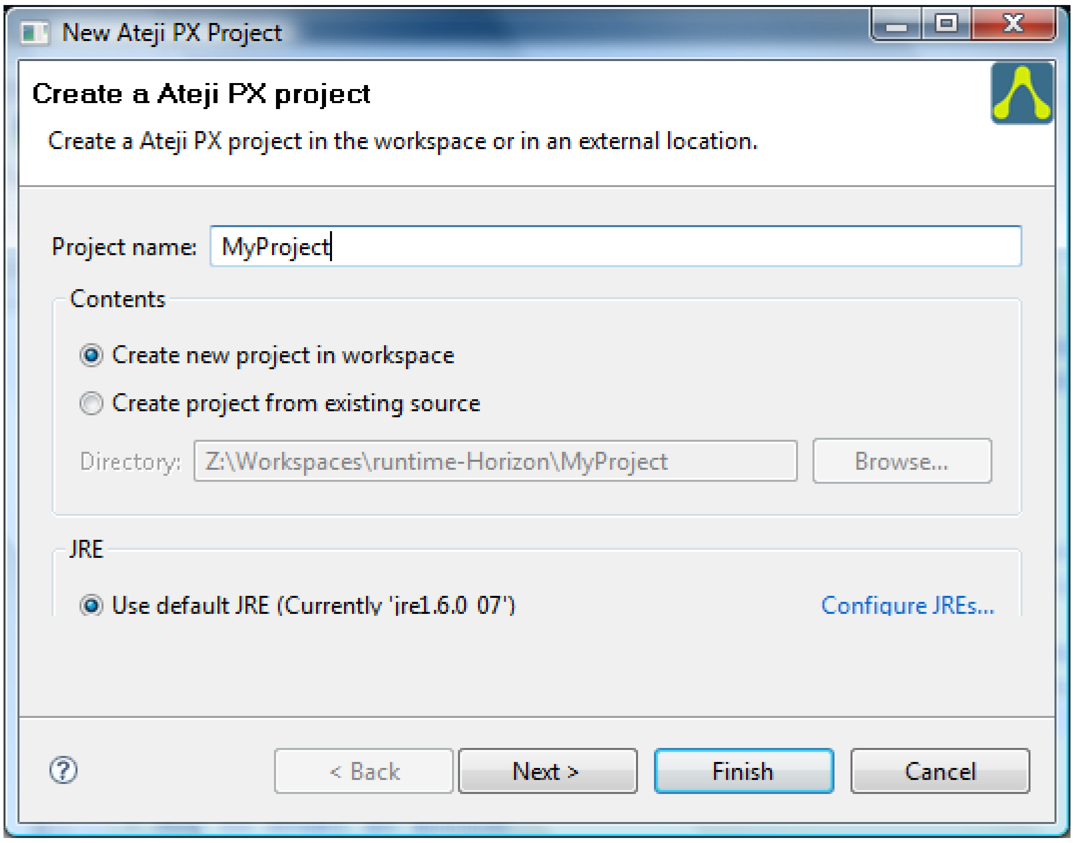
2. Create a new Ateji PX project

At first reset the Eclipse perspective. Select `Window -> Reset Perspective` and then accept the confirmation:



Then select `File > New`, then choose `Ateji PX Project` (if you can not choose `Ateji PX Project` then you must reset the perspective):

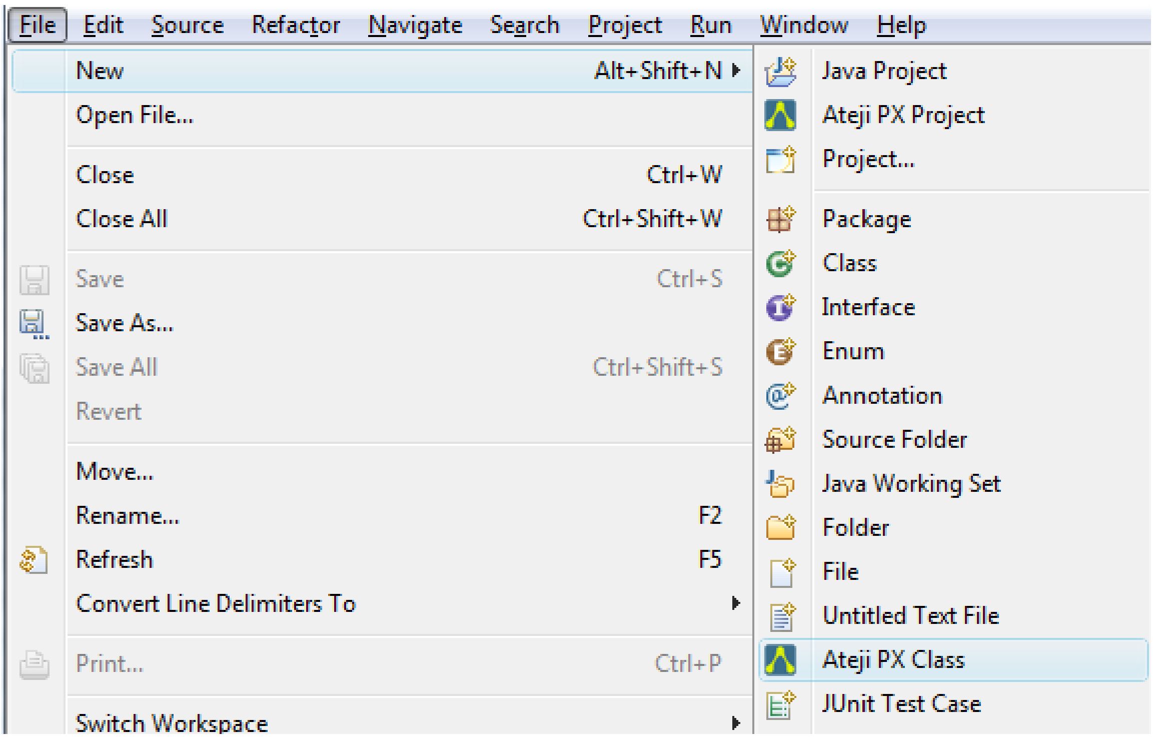



New Ateji PX Project
Create a Ateji PX project
Create a Ateji PX project in the workspace or in an external location.
Project name: MyProject
Contents
Create new project in workspace
Create project from existing source
Directory: Z:\Workspaces\runtime-Horizon\MyProject
Browse...
JRE
Use default JRE (Currently 'jre1.6.0_07')
Configure JREs...
< Back
Next >
Finish
Cancel

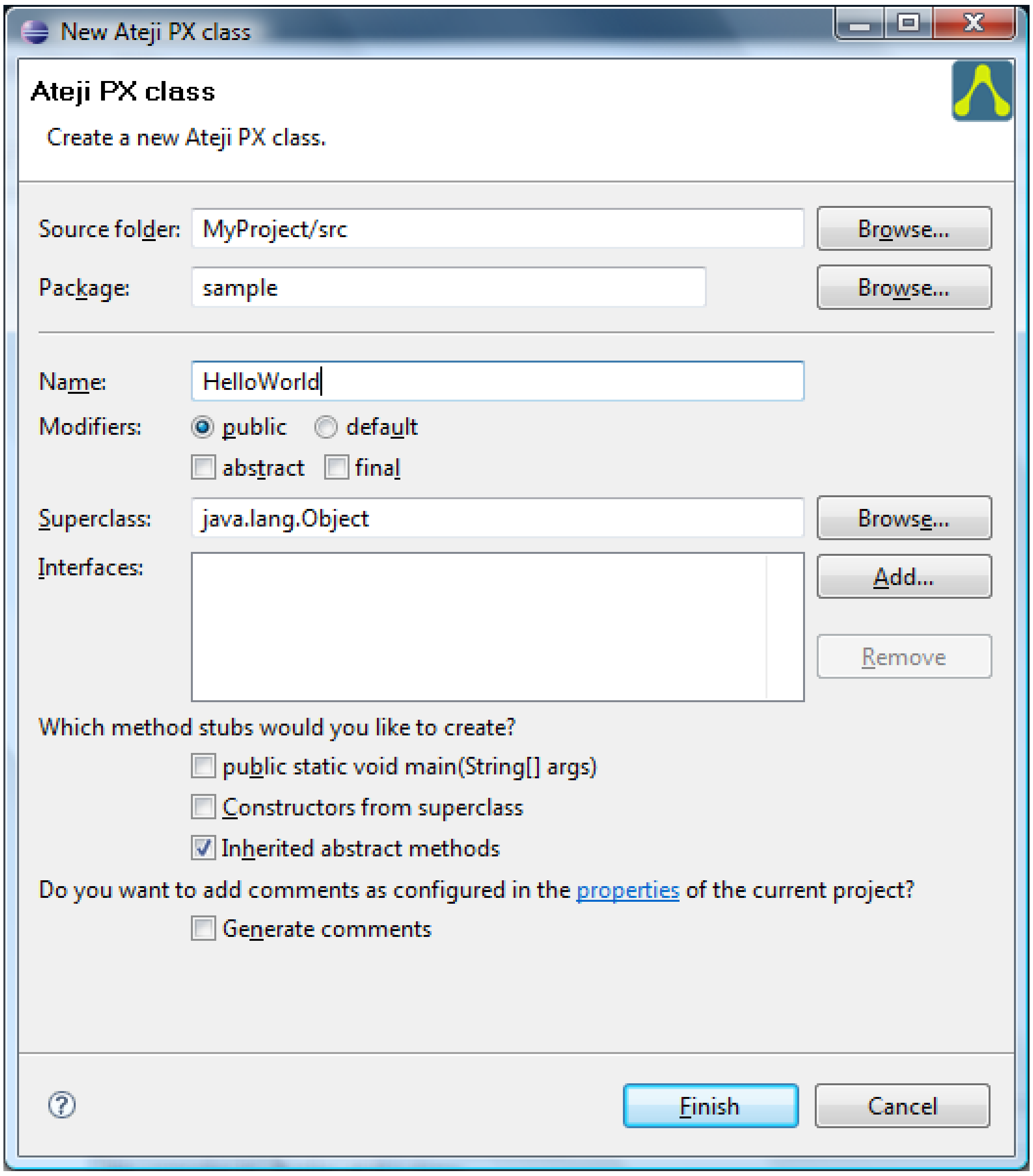
Enter the new project name and click on `Finish`.

3. Create an Ateji PX class

Select `File > New`, then choose `Ateji PX Class` :



Fill the form and click on `Finish`.



New Ateji PX class

Ateji PX class

Create a new Ateji PX class.

Source folder:	MyProject/src
Package:	sample
Name:	HelloWorld
Modifiers:	[✓ public] default; abstract, final
Superclass:	java.lang.Object
Interfaces:	

Which method stubs would you like to create?

- [ ] public static void main(String[] args)
- [ ] Constructors from superclass
- [✓] Inherited abstract methods

Do you want to add comments as configured in the properties of the current project?

- [ ] Generate comments

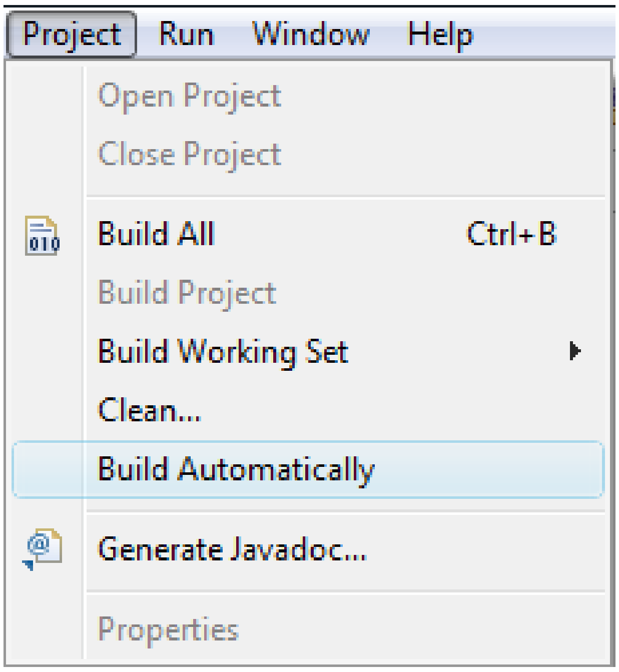
4. Write parallel code

Copy and paste the following code in the `HelloWorld.apx` source file:

```
package sample;

public class HelloWorld
{
  public static void main(String[] args)
  {
    [
      || System.out.println("Hello");
      || System.out.println("World");
    ]
  }
}
```

5. Compile code

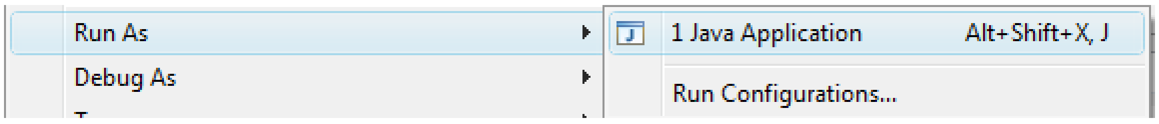
In order to compile your sources automatically you have to ensure that the option "Build Automatically" is set. This option is located at `Project > Build Automatically`.



Now save the `HelloWorld.apx` file if you have not done already and the Ateji PX compiler will build your project.

6. Launch your code

To run the main method of the HelloWorld class right click on `HelloWorld.apx` and select `Run As > Java Application`:



Running this program will print either

```
Hello
World
```

or

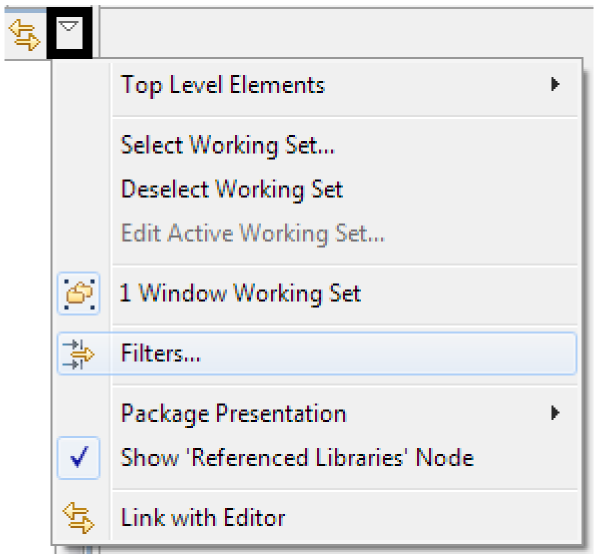
```
World
Hello
```

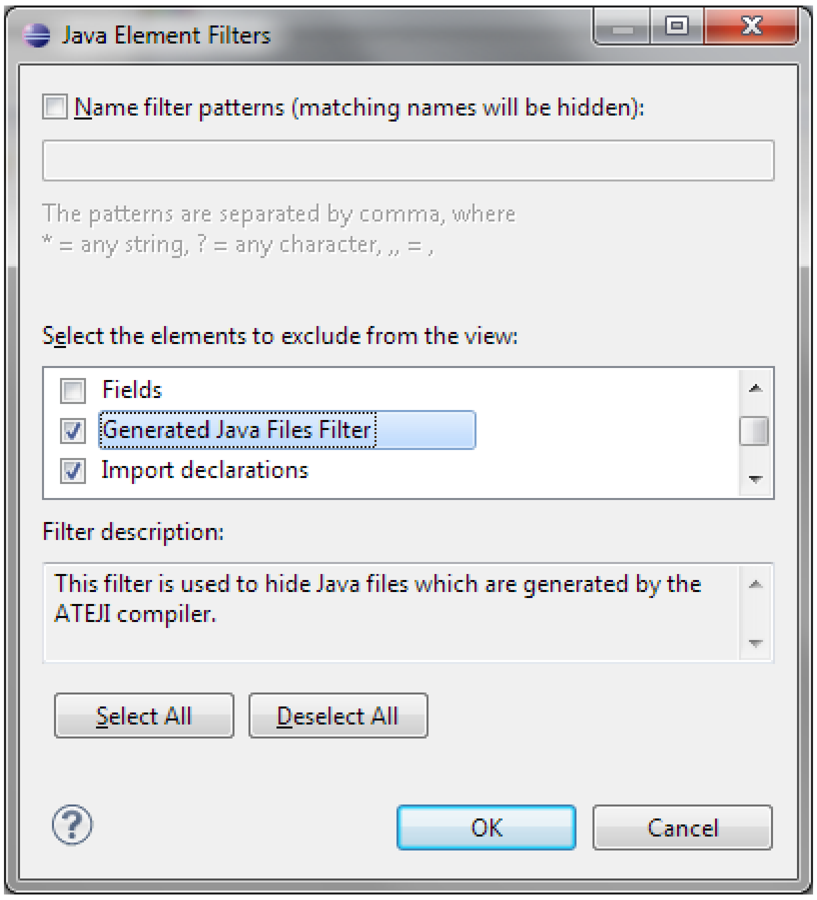
7. Links between Java and Ateji PX

Ateji PX projects are Java projects with an additional nature that handles the compilation of Ateji PX source files. Java and Ateji PX sources lie side by side in the same project.

7.1. See the generated Java files

The Ateji PX compiler is implemented as a source-to-source compiler. This is essential for making Ateji PX compatible with standard productivity tools such as Javadoc that require a Java source file as input. You can see the generated Java files by disabling the "Generated Java Files" filter in the Package Explorer view.




Java Element Filters

☐ Name filter patterns (matching names will be hidden):

The patterns are separated by comma, where
* = any string, ? = any character, ,, = ,

Select the elements to exclude from the view:

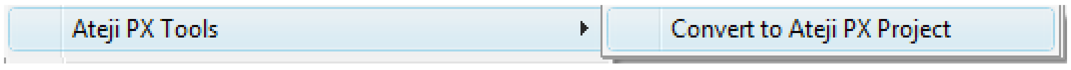
- ☐ Fields
- ☑ Generated Java Files Filter
- ☑ Import declarations

Filter description:

This filter is used to hide Java files which are generated by the ATEJI compiler.

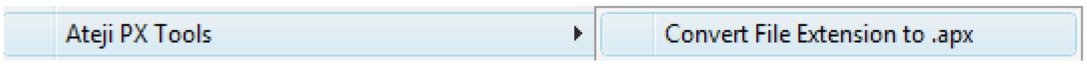
7.2. Convert an existing Java project to an Ateji PX project

You can convert one of your existing Java projects into an Ateji PX project by adding it the Ateji PX nature. Right-click on a Java project and select:



7.3. Convert an existing Java class to an Ateji PX class

You can also convert one of your existing Java classes into an Ateji PX class.


Ateji PX Tools
Convert File Extension to .apx

Chapter 3. Language overview

1. Hello World

If you already know the Java™ language, you'll only need to learn a few additional constructs to write parallel programs. The first one is the parallel composition:

```
class HelloWorld
{
  public static void main(String[] args)
  {
    [
      || System.out.println("Hello");
      || System.out.println("World");
    ]
  }
}
```

The square brackets `[ ... ]` introduce a parallel block. A parallel block is an indication given by the programmer that the branches it contains *may* be run in parallel. Whatever the actual implementation, the set of possible program outcomes will remain the same.

Each `|| ...` inside the block introduces a branch. The parallel block terminates when all branches it contains have terminated.

Parallelism in Ateji PX is *compositional*: you can think of `[ || ... || ... ]` as a parallel composition operator, the counterpart of the sequential composition operator `{ ...; ...; }`.

Running this program will print either

```
Hello
World
```

or

```
World
Hello
```

The two branches are run in parallel, in no particular order. When multiple processors or processor cores are available, Ateji PX tries to maximize performance by allocating branches to available processing resources. Parallel blocks do not guarantee any kind of fairness in the execution of branches. In particular, branches may possibly be run in sequence one after the other in any order.

2. How parallel branches are executed

There are many possible options for executing branches, ranging from totally sequential to totally threaded (each branch in its own thread). The default option tries to use about as many threads as there are available processors.

In most cases, the default option should suit your purposes. One notable exception is when you need responsiveness, such as when writing user interface code. In this case, you must specify that you want each branch to be run in its own thread, so that branches can be scheduled preemptively. Block-level indications are the way to specify this:

```
  [
    // states that these branches *must* run in their own thread
    || (#Thread()) userInterfaceCode();
    || (#Thread()) longRunningTask();
  ]
```

Without this indication, Ateji PX would be free to execute branches sequentially, first the long running task, then the user interface code, making the program totally unresponsive.

Additionally, branch-level indications make it possible to name branches. This name will be displayed in the debugger:

```
  [
    || (#Name("Branch1")) branch1();
    || (#Name("Branch2")) branch2();
  ]
```

3. Decomposing work for parallel processing

The most straightforward case is when you want to apply an independent computation to all elements of a collection. The following example increments each element of an array:

```
int[] array = new int[N];
[
  || (int i : 0 .. N-1, if (i % 2 == 0) ) array[i]++;
]
```

Here the parallel branch is quantified by a qualifier list that may contain an arbitrary number of generators (`i : 0 .. N-1`) and filters (`i % 2 == 0`). One branch is created for each odd value of i.

Another typical approach is recursive decomposition. Consider the Fibonacci sequence defined as `fib(n) = fib(n-1)+fib(n-2)` for `n>1`:

```
int fib(int n) {
  if(n <= 1) return 1;
  int fib1, fib2;
  [
    || fib1 = fib(n-1);
    || fib2 = fib(n-2);
  ];
  return fib1 + fib2;
}
```

Ateji PX will recursively create parallel branches (there are however better algorithms for computing the Fibonacci sequence).

4. Controlling parallelism

Creating a branch, regardless of how it is implemented, incurs a small run-time overhead. As a rule of thumb, you may assume about 10µs per branch on a standard desktop PC.

When creating many branches, this overhead may become significant.

```
 [
   || (int i: 0 .. I-1, int j: 0 .. J-1) {
        ... do something with element indexed by i,j
      }
 ]
```

This will create `I*J` branches. Generator-level indications make it possible to reduce this overhead:

```
 [
   || (#BlockCount(N), int i: 0 .. I-1, int j: 0 .. J-1) {
        ... do something with element indexed by i,j
      }
 ]
```

With the `BlockCount(N)` indication, exactly `N` branches will be created. Within each of the `N` branches, computation will proceed using sequential `for` loops. Typically you would set `N` to be the number of available processors.

```
[
  || (#BlockSize(N), int i: 0 .. N-1, int j: 0 .. N-1) {
       ... do something with element indexed by i,j
     }
]
```

The indication `BlockSize(N)` is similar, but will create just enough branches to iterate over `i` in bunches of size `N`.

Since the inner loop is sequential, these indications should not be used for branches that need to exchange messages with each other, as they may block the whole parallel block.

5. Sharing memory between branches

In imperative languages, the main mechanism for exchanging data between different parallel branches is access to shared memory.

Ateji PX allows branches to communicate using shared memory. In this example, the local variable sum is accessed in reading and writing by both branches:

```
    // sum must be updated atomically - using an int here would be incorrect
    AtomicInteger sum = new AtomicInteger(0);
    [
      || sum.addAndGet(1); // instead of sum += 1
      || sum.addAndGet(2); // instead of sum += 2
    ]
```

Shared memory introduces a whole spectrum of problems such as data races, visibility problems, deadlocks, and performance hits because of poor cache usage. Since branches may run in different threads, you must take the usual care and use standard techniques such as locks, atomic operations, volatile keywords, etc., in order to guarantee a well-defined behavior.

6. Exchanging messages

Message passing is an alternative to shared memory for communication between branches. Message passing has many advantages over shared memory:

- under some simple assumptions (such as avoiding sharing channels), programs based on message passing avoid most of the problems related to shared memory and are amenable to automatic analysis and verification
- programs based on message passing are much more likely to be scalable over future hardware architectures, using thousands of processing units and non-uniform memory access
- programs based on message passing can be distributed over clusters of computers across a network

Shared memory programs remain more efficient than message passing programs for some heavy data-parallel algorithms on current multi-core processors. However they often require fine tuning and may not scale to future hardware architectures.

Ateji PX provides message passing at the language level. This enables synchronization and data exchange between concurrently running branches without any reference to shared memory:

```
Chan<String> c = new Chan<String>();
[
  || c ! "Hello World";      // send a message on channel c
  || c ? String s;           // receive a message on channel c
     System.out.println(s);  // and print the received value
];
```

The first branch sends the string `"Hello World"`, the second branch receives it and prints it.

Channels of type `Signal` do not carry values and are used for synchronization purposes only.

Asynchronous (buffered) communication is straightforward to implement on top on synchronous message passing, by making the buffer explicit. Ateji PX also provides predefined asynchronous channels.

A channel being an object, it is possible to send a channel over another channel.

6.1. Synchronous vs. Asynchronous channels

A channel can be synchronous or asynchronous:

- `Chan<T>` : synchronous channel carrying values of type T
- `AsyncChan<T>` : asynchronous channel carrying values of type T

Synchronous channels implement rendezvous synchronization. The sender and the receiver must synchronize before the message passing operation completes. This is unlike usual read/write or send/receive methods pairs, where read is blocking but write is not.

Asynchronous channels are allowed to buffer messages. Sending does not block as long as the buffer is not full, and receiving does not block as long as the buffer is not empty.

This example shows the difference in behavior:

```
  c ! 1; c ? v; // send then receive on channel c
```

If c is a synchronous channel, the program will block since the send operations cannot complete unless there is a matching receive operation. If c is an asynchronous channel the program does not block: the send operation can proceed before the receive operation take place.

6.2. Sharing vs. Copying channels

Channels may carry values or references:

```
IChan<int[]> sharingChannel = new Chan<int[]>();
IChan<int[]> copyingChannel = new Chan<int[]>(new SerializationCloner<int[]>());
```

The difference is meaningful if you ever *modify* an object after it has been sent or received over a channel:

```
  Object o = new Object();
  [
    || c ! o;
    || c ? value; System.out.println(o == value);
  ]
```

If `c` is a shared channel, the program will print `true`. Any modification of the object in one branch will be visible to the other branch (standard care about shared memory accesses applies here). If `c` is a copying channel, the program will print `false`. Modifications in one branch will not be visible in the other branch.

Shared channels are provided for performance reasons, and can be used on shared-memory architectures only. Copying channels must be used when no memory is shared, such as remote communication over a network. In cases where both types of channels can be used, the balance is between performance (shared channels are faster) and reliability (copying channels induce less hidden bugs).

Most channel constructors can take an argument of type `Serializer` or `Cloner` that specifies the channel behaviour regarding copying and sharing:

`Chan` and `AsyncChan` use `Cloner` objects for defining their share/copy behaviour:

- `ShareCloner`: values are shared
- `SerializationCloner`: values are copied using the standard Java serialization mechanism

Other channels use `Serializer` objects for defining their share/copy behaviour:

- `CopySerializer`: values are copied using the standard Java serialization mechanism

Additionally, you can define your own `Cloner` or Serializer objects for specifying the serialization mechanism to be used. For instance, channels bound to web services would require XML serialization.

Note

Additional predefined Cloner and Serializer types will be provided in future releases. Use the support forums to express your wishes.

6.3. I/O mapped channels

Almost any class featuring a pair of read()/write() or send()/receive() methods can be mapped to an Ateji PX channel. This has many advantages:

- Synchronization is managed by Ateji PX, which ensures in particular that the whole program does not get blocked because a task is waiting on I/O (as it is the case for instance with the Java Executor framework)
- Powerful Ateji PX constructs such as `select` can be used together with I/O devices
- The same program can be reused to target different libraries, or use indifferently Ateji PX channels when running on a single machine or a communication library when running on different machines across a network, with minimal or no changes in the source code.

I/O mapped channels are typically asynchronous and copying. The following I/O mapped channels are available:

- `InputStreamChan` : wraps an `InputStream` as an Ateji PX channel
- `OutputStreamChan`: wraps an `OutputStream` as an Ateji PX channel
- `ReaderChan`: wraps a `Reader` as an Ateji PX channel
- `WriterChan`: wraps a `Writer` as an Ateji PX channel

6.4. The IChan interface

When writing code that is suitable for any kind of channel, whether synchronous or asynchronous, whether copying or sharing, use `IChan<T>` as the channel type.

`IChan` is an interface implemented by all channels.

7. Starred branches

A starred branch represents a potentially infinite number of branches, whose creation is controlled by a message passing operation. Whenever a message is sent or received, an additional branch will be created, running in parallel with the existing branches:

```
[
  ||* c ? int i: System.out.println(i);
]
```

This creates one branch for each value received.

Starred branches enable idioms such as the listen/accept mechanism for sockets.

8. Example: data-flow programming

An adder is a process that indefinitely reads two values on its two output channels, and outputs their sum on its output channel:

```
void adder(Chan<Integer> in1, Chan<Integer> in2, Chan<Integer> out)
{
  for(;;) {
    int v1, v2;
    [ in1 ? v1;  || in2 ? v2; ]   // receive v1 and v2 in no particular order
    out ! v1+v2;                  // send their sum
  }
}
```

The input and output channels are passed as parameters to the method. The two reads are run in parallel branches, making it explicit that no specific order is required for the arrival of input values. (a purely sequential language would be unable to express this)

This adder must be run in parallel with producer processes that send values to its input channels and a receiver process that reads values from its output channel.

```
Chan<Integer> in1 = new Chan<Integer>();
Chan<Integer> in2 = new Chan<Integer>();
Chan<Integer> out = new Chan<Integer>();
[
  || producer(in1); // sends values on in1
  || producer(in2); // sends values on in2
  || adder(in1, in2, out);
  || consumer(out); // received values on out
]
```

9. Selecting

Selection provides a way to bind the behaviour of a program to external events, by committing atomically to a pair of message send/receive operations that is ready for communication.

```
select {
  when c1 ? v1: { System.out.println("received value " + v1 + " on channel c1"); }
  when c2 ! v2: { System.out.println("sent value " + v2 + " on channel c2"); }
}
```

This will either receive a value on c1 or send a value on c2, but never both, depending on whichever channel is ready first. When the two channels are ready, the choice is non-deterministic. Select blocks do not guarantee any kind of fairness when multiple channels are ready.

Message reception in a select may either bind an existing variable, or bind a new variable:

```
select {
  when c1 ? v1: { ... }   // v1 has been declared before
  when c2 ? T v2: { ... } // declares a local variable v2 of type T
}
```

Like branches of parallel blocks, choices of a select statement can be quantified using qualifier lists:

```
select {
  when (int i: 0 .. N-1) channel[i] ? value : { System.out.println(value); }
}
```

This will select a ready channel among all those in the array. This idiom is typically used when waiting for incoming connections.

Filters in qualifier lists make it possible to implement guards:

```
select {
  when ( if (condition) ) c1 ? v1: { ... } // communication will be attempted
                                           // only when condition is true
}
```

10. Example: stopping an infinite process

A typical use case of the select statement is the termination of an infinite process.

```
for(;;) {
  select {
    // either receive an input value and perform some work
    when in? int value: { out ! value+1; }
    // or receive a stop signal and stop the process
    when stop? : { return; }
  }
}
```

11. Comprehensions

Comprehension expressions can express algebraic calculations in a simple way.

For instance, this is how the sum of the first n integers would be expressed in Ateji PX using a comprehension:

```
int firstIntegers = `+ for(int i : N) i;
```

The last part "`i`" is called the target, it may contain an arbitrary expression. The following line computes the sum of the first N squares:

```
int firstSquares = `+ for(int i : N) (i*i);
```

The part "`int i : N`" is called the qualifier list, it may contain an arbitrary number of generators and filters. The following line computes the sum of all elements from a IxJ matrix not lying on the diagonal:

```
int sumExceptDiagonal = `+ for(int i : I, int j : J, if (i != j)) matrix[i][j];
```

11.1. Monoids

In the previous code snippets, the `` `+ `` operator was used to aggregate values. This operator is called a *monoid*.

All associative Java binary operators can be used instead of plus. For example, a multiplication can be written by using the `` `* `` monoid; a boolean sum can be written by using the `` `| `` monoid:

```
int prod = `* for (int i : 1 .. 10) i;
```

```
// Check whether an element satisfies the following property: (i%129 == 0)
boolean some = `| for (int i : 1 .. n) (i%129 == 0);
```

Moreover, other monoids are provided by Ateji. They are in the library. The monoids calculating min and max are included as well as monoids building collections such as `HashSet` and `ArrayList`.

```
int min = Monoids.min() for (int i : 1 .. n) i;
```

```
Set<Integer> set = Monoids.hashSet() for (int i : 1 .. 10) i;
```

The last type of monoids are those developed by users; they are presented in Section 13, “User-defined monoids”.

11.2. Parallel comprehensions

To parallelize the calculation, simply add the symbol `||` after the keyword `for`:

```
int sumExceptDiagonal = `+ for || (int i : I, int j : J, if (i != j)) matrix[i][j];
```

However, not all comprehensions can be parallelized; only comprehensions using a *commutative* monoid can be parallelized. For instance the following comprehension building an `ArrayList` cannot be parallelized:

```
List<Integer> set = Monoids.arrayList() for (int i : 1 .. 10) i;
```

Indeed, the list concatenation operator is not commutative; adding `0` then `1` does not build the same list than by adding `1` then `0`. In this example, attempting to parallelize the comprehension by adding the parallel bar `||` will lead to a compile-time error.

11.2.1. Controlling comprehension parallelism

Much like a regular parallel branch, the parallelism of a parallelized comprehension can be controlled. The generator-level indications described in Section 4, “Controlling parallelism” can be used in the qualifier-list of a parallelized comprehension to achieve the same result.

11.3. Comprehension evaluation

In order to evaluate a comprehension expression, the Ateji PX compiler transforms them into pure Java code. The algorithm is described in the following article: *Towards an Effective Calculus for Object Query Languages - L. Fegaras and D. Maier - ACM SIGMOD International Conference on Management of Data - 1995*.

A comprehension expression can be seen as a shorthand for a number of embedded `for` loops and `if` statements. Take for example the previously shown computation of the sum of all elements from a IxJ matrix not lying on the diagonal:

```
int sumExceptDiagonal = `+ for(int i : I, int j : J, if (i != j)) matrix[i][j];
```

Using embedded for loops and if statements the line above would be written in Java as:

```
int sumExceptDiagonal = 0;
for(int i = 0; i < I; i++) {
  for(int j = 0; j < J; j++) {
    if(i != j) {
       sumExceptDiagonal  += matrix[i][j];
    }
  }
}
```

12. Example: parallel reductions

The Ateji PX syntactic construction of generalized comprehensions includes in particular the notion of parallel reduction as it can be found for instance in OpenMP.

Parallel reduction is a common operation whereby a binary operation is applied in parallel over all elements of a given collection (see e.g. http://en.wikipedia.org/wiki/OpenMP#Reduction for the case of OpenMP).

Ateji PX provides a concise and efficient notation for parallel reductions. Here is the parallel sum of all squares from 0 to N-1:

```
int firstSquares = `+ for || (int i : N) (i*i);
// sumOfSquares = 0*0 + 1*1 + ... + (N-1)*(N-1)
```

Since addition can be performed in any order (addition is associative and commutative), this leads to a natural parallel decomposition scheme: have a number of parallel tasks each working on one part of the collection, summing the intermediate results in a local variable, then sum all those intermediate results in a shared global result variable.

Ateji PX takes care of introducing local variables and managing all the necessary synchronization. Try it on your own computer and see what speedup can be obtained by introducing a single parallel bar.

13. User-defined monoids

A monoid is the "operator" used in comprehensions as shown in Section 11, “Comprehensions”. It is possible to define custom monoids besides the ones provided by Ateji, for instance:

- a monoid computing a set of statistics such as minimal value, maximal value, number of elements, average value, and so on; or
- a monoid building a non-standard data structure; or

- a monoid concatenating strings by inserting a separator between each element.

In Ateji PX, a typical monoid is a subtype of the following abstract class:

```
apx.util.monoid.Monoid<C, E>
```

Defining a new monoid is a matter of extending the class `Monoid<C, E>` and providing an implementation for the few abstract methods: `zero`, `unit`, `merge` and `add` (optional).

13.1. Monoid type parameters

The type parameters `C` and `E` of a `Monoid` corresponds respectively to the type of the results built by the monoid and to the type of the elements processed by the monoid; `C` is called the *collection type* and `E` is called the *element type*.

For instance the monoid building a set of `String` from `String` elements would have `C=Set<String>` and `E=String`.

Also, when a given monoid is used in a comprehension expression, the type of the comprehension expression is the same as the collection type of the monoid, and the target expression must be convertible to the element type of the monoid.

13.2. Monoid methods

Of all the `Monoid` methods, the conceptually most important ones are `merge` and `zero`:

- The `merge` method defines how to merge two collections into one. The function implemented by the `merge` method must be *associative*, that is that `merge(merge(a, b), c)` and `merge(a, merge(b, c))` both return the same result.
- The `zero` method must return the *neutral element* of the monoid. The neutral element is the element such that its application to an arbitrary collection returns the same collection, that is `merge(zero(), c)` always returns `c` for any collection `c`.

These two properties are what defines the mathematical concept of monoid: an associative operator with a neutral element.

The `unit` method is used to bridge the collection type and the element type; given an element, it builds the single-element collection with this element.

The `add` method is used to directly add one element to a collection. Implementing this method is not mandatory; there is an obvious default implementation which uses the `unit` and `merge` methods. However, it is recommended to override the default implementation whenever it is inefficient. For instance adding elements to an existing `List` is much more efficient by using the `Collection.add(...)` method from the Java collections framework.

Finally, we recommend to take a close look at the included samples which contains a detailled tutorial on how to define your own monoids.

13.3. Commutative monoids

In order to be used in a parallel comprehension (see Section 11.2, “Parallel comprehensions”) a monoid must be *commutative*.

A commutative monoid is a monoid whose `merge` method is commutative: `merge(a, b)` and `merge(b, a)` both return the same result.

In Ateji PX, a monoid is commutative if it is a subtype of the following interface:

```
apx.util.Commutative
```

Defining a commutative monoid is a matter of implementing the interface `Commutative`. It is the responsability of the implementor to ensure that the `merge` method is actually commutative.

13.4. Other base implementations

Ateji PX provides other base implementations which can be extended instead of `Monoid`. Some of them actually supports a special handling from the Ateji PX compiler in order to provide better performance; others are just classic base implementations which require less work to implement.

13.4.1. Mutable monoid

Some monoids build their result by modifying and returing an existing collection rather than building a new one. Those monoids are called *mutable monoids* because they require their collection type to be a mutable type. Such a monoid should extend the following class instead of `Monoid`:

```
apx.util.monoid.MutableMonoid<C, E>
```

For a mutable monoid, the return value of the `merge` method is irrelevant, since the work is already done by side-effects. This is reflected by the abstract methods required to be implemented by `MutableMonoid`.

Also the Ateji PX compiler uses this information to provide a slightly more efficient translation than for `Monoid` where the return value of the `merge` method is relevant and must not be discarded.

13.4.2. Collection monoid

The most typical mutable monoids are monoids building collections of objects. The Ateji PX library provides a base implementation for such *collection monoids*:

```
apx.util.monoid.CollectionMonoid<E>
```

`CollectionMonoid` defines `C` as being `java.util.Collection<E>`. and provides reasonably efficient base implementations of the monoid methods using this property. In particular, the `add` method is implemented by directly adding the element to the collection by using the `Collection.add(...)` method.

13.4.3. Void monoid

It is possible to define a monoid whose only purpose is to make side-effects. Such a monoid is called a *void monoid*: a monoid which actually does not build any collection.

```
apx.util.monoid.VoidMonoid<E>
```

Most monoid methods become irrelevant for a void monoid since there is no collection to build. Thus the only abstract method that needs to be implemented when extending `VoidMonoid` is the `VoidMonoid` equivalent of the `unit` method: what must be done for each given element.

13.4.4. Primitive monoid

The `Monoid` class can not be used with primitive types, because the type parameters of `Monoid<C, E>` must be reference types; this is a hard constraint from the Java language specification. Using primitive values thus relies on wrapper types (for instance `Integer` for `int`) which can lead to a significant overhead at run-time.

For this specific case, *primitive monoids* can be used. A primitive monoid has the following properties:

1. the collection type and the element type are both the same;
2. the collection/element type is a primitive type; and
3. there is no `unit` method because `unit` must be the identity, that is `unit(x)` is `x`.

In Ateji PX, a primitive monoid is a subtype of any of these interfaces:

```
apx.util.monoid.PrimitiveBooleanMonoid
apx.util.monoid.PrimitiveByteMonoid
apx.util.monoid.PrimitiveCharMonoid
apx.util.monoid.PrimitiveDoubleMonoid
apx.util.monoid.PrimitiveFloatMonoid
apx.util.monoid.PrimitiveIntMonoid
apx.util.monoid.PrimitiveLongMonoid
apx.util.monoid.PrimitiveShortMonoid
```

Defining a new primitive monoid is a matter of implementing any of these interfaces. These interfaces are not mutually-exclusive and are compatible with the `Monoid` abstract class; for instance it is possible to define a single monoid type which would support `Integer` reference values (by extending `Monoid<Integer, Integer>`) as well as both `byte` and `int` primitive values (by implementing `PrimitiveByteMonoid` and `PrimitiveIntMonoid`).

14. Non-local exits and termination

Branches in a parallel block may contain non-local exit statements such as `break`, `continue`, `return`, or may throw exceptions. When a branch perform a non-local exit, all other branches in the same parallel block are interrupted (politely asked to terminate), then the parallel block terminates with the exit status provided by the interrupting branch.

This can be used for instance to implement speculative parallelism by running two different algorithms in parallel and stopping when one of them terminates:

```
[
  || result = algorithm1(); return result;
  || result = algortihm2(); return result;
]
```

The first branch that terminates will interrupt the other, and return its result.

Although a branch is not necessarily run in its own thread, the interruption mechanism used by Ateji PX is the standard Java thread interrupt mechanism. When a branch is interrupted during a blocking operation, an exception is thrown in the branch. You can also test the interrupted status to make a branch more responsive to interruption, as in this example:

```
[
  ||
  // first branch
  for(;;) {
    // infinite loop, responsive to interruption
    doSomething1();
    // test if we've been asked to terminate
    if(Thread.currentThread().isInterrupted()) {
      break; // stop the loop and terminate the branch
    }
  }

  ||
  // second branch
  doSomething2();
  return; // return interrupts the first branch
]
```

A blocking operation will typically react to interruption by throwing `java.lang.InterruptedException` or `apx.lang.ApxInterruptedException`. Both have the same meaning, but the latter is an unchecked exception. Ateji PX provides this facility because Ateji PX programs are typically littered with interruptible statements, such as message sending and receiving, and requiring a checked exception would make programs unnecessarily verbose.

You normally do not need to care about interrupted exceptions unless you want to clean resources before stopping an interrupted branch. In this case, catch the exception, but do not forget that the branch has been asked to stop, and that it will only be asked once. At the end of the catch block, you must take an action that will stop the branch, such as exiting a loop, jumping to an end label, or by default re-throwing the same exception.

15. Ranges

We have already seen ranges in a few examples, they are used to express ranges of integers in generators. They are written `min .. max`, where `min` and `max` are integral values:

```
[
  // create N branches, for i ranging from 0 to N-1
  || (int i: 0 .. N-1) doSomething(i);
]
```

As a shorthand, a single integer used as generator stands for a 0-based range:

```
[
  // same as above : i ranges from 0 to N-1
  || (int i: N) doSomething(i);
]
```

16. Compatibility with legacy Java programs and libraries

The Ateji PX model of parallelism is compatible with the standard Java models, such as plain threads or fork/join. Both can be mixed as long as you follow these simple guidelines:

- Remember that a branch in a parallel block may or not run in its own thread, depending on the implementation and the hardware architecture. Never assume that one branch = one thread.
- Use thread-level methods, such as `start()`, `join()`, etc., only with threads that you have created yourself, and never directly in an Ateji PX parallel branch.
- Since different branches may run in different threads, you must take the usual care for protecting memory accesses shared across branches (volatile keywords, atomic instructions, locks, etc.).
- As long as you follow these guidelines, you can safely create threads within Ateji PX branches, and branches within threads, and use thread-based libraries.

Blocking methods in branches require a special handling. Ateji PX ensures that calling a blocking method will block only the relevant branch, not the whole program, but the compiler has no way to know if a user-defined method may block or not.

Chapter 4. Tutorials

1. How to parallelize a Java "for" loop

You're probably familiar now with the Ateji PX parallel block construct:

```
 [
  || aSingleBranch();
  || (int i: 1..10) aQuantifiedBranch();
]
```

Ateji PX provides an alternative syntax when the block contains exactly one quantified branch, that mimics a Java "for" loop:

```
for || (int i : 1 .. 10) ...
```

The purpose of this syntax is to simplify the parallelization of "for" loop : simply add a parallel bar after the "for" keyword.

The iterations of for loop are partitioned among all available cores. Specifically, the Ateji PX compiler automatically splits the range 1 .. 10 into n blocks, where n is the number of available processors, and then executes the n blocks in parallel.

Matrix multiplication sample:

Matrix multiplication is a standard benchmark for parallel programming. The following Ateji PX code computes the product matrix A = B * C sequentially :

```
for (int i : 1 .. I)
 for (int j : 1 .. J)
   for (int k : 1 .. K)
     A[i][j] += B[i][k] * C[k][j];
```

To parallelize this code, simply add the symbol '||' at the **outer** loop:

```
for || (int i : 1 .. I)
 for (int j : 1 .. J)
   for (int k : 1 .. K)
     A[i][j] += B[i][k] * C[k][j];
```

Now try this pattern on your computer. You should notice an important speed-up between the sequential and the parallel version. You can also find the complete version of matrix multiplication in our samples library.

2. How to run Ateji PX in batch mode

Automating the build process is an important requirement for most IT projects. This is typically done from a script that performs a full checkout of the code, recompiles everything, packs the executable files and then launches the test suite. A one step process is the basis for daily builds and will help ensuring that no breakage goes unnoticed.

The Ateji PX compiler can be run independently in batch mode. This makes it possible to use Ateji PX in combination with build tools like Ant and integrate the Ateji PX compiler in your build process. You can now easily integrate Ateji PX with your build and deployment architecture.

This tutorial explains how to use the Ateji PX compiler in command-line mode to compile the file `HelloWorld.apx`. It is assumed that you are familiar with command-line tools. Moreover, you are expected to know how to run Java applications.

Please note that even in command-line mode, the Ateji PX compiler still needs the Eclipse plug-in. You will have to set the classpath by yourself.

2.1. Setting the classpath

You must explicit the classpath that has to be used to run the Ateji PX compiler so that it matches your Eclipse installation.

First, define some environment variables.

The variable `ECLIPSE_PATH` will be the path to the Eclipse installation in which Ateji PX has been installed; this is the root folder of Eclipse where the Eclipse binary can be found alongside the `plugins` and `features` sub-folders. Here for the example we are using `C:\eclipse` under Windows and `~/eclipse` under Linux but your Eclipse installation can be somewhere else; adjust the command accordingly.

```
(under Windows) > set ECLIPSE_PATH=C:\eclipse

(under Linux)   > ECLIPSE_PATH=~/eclipse
```

The variable `APX_PATH` will be the path to the Ateji PX folder containing the JAR that we need to use.

```
(under Windows) > set APX_PATH=%ECLIPSE_PATH%\plugins\com.ateji.bonza.lang.horizon_VERSION

(under Linux)   > APX_PATH=$ECLIPSE_PATH/plugins/com.ateji.bonza.lang.horizon_VERSION
```

where `VERSION` must be the version of your latest installation of Ateji PX -- when in doubt, select the highest version number.

The variable `APX_JAR` will be the path to the Ateji PX command-line JAR.

```
(under Windows) > set APX_JAR=%APX_PATH%\commandline-noeclipseplugins.jar

(under Linux)   > APX_JAR=$APX_PATH/commandline-noeclipseplugins.jar
```

This JAR depends on other plug-ins which can be found inside the `plugins` folder inside the Eclipse installation. The variables `ECLIPSE_COMMON_JAR`, `ECLIPSE_RESOURCES_JAR` and `ECLIPSE_JOBS_JAR` will be the path to the JAR of these plug-ins. Here for the example we are using the plug-ins that ships with Eclipse 3.6.1 (Eclipse Helios SR1) but if you are using another version of Eclipse the JAR versions will differ; adjust the commands accordingly.

```
(under Windows) > set ECLIPSE_COMMON_JAR=%ECLIPSE_PATH%\plugins\org.eclipse.equinox.common
                _3.6.0.v20100503.jar
(under Windows) > set ECLIPSE_RESOURCES_JAR=%ECLIPSE_PATH%\plugins\org.eclipse.core.resour
                ces_3.6.0.R36x_v20100825-0600.jar
(under Windows) > set ECLIPSE_JOBS_JAR=%ECLIPSE_PATH%\plugins\org.eclipse.core.jobs_3.5.1.
                R36x_v20100824.jar
(under Windows) > set ECLIPSE_JDT_JAR=%ECLIPSE_PATH%\plugins\org.eclipse.jdt.core_3.6.1.v_
                A68_R36x.jar

(under Linux)   > ECLIPSE_COMMON_JAR=$ECLIPSE_PATH/plugins/org.eclipse.equinox.common_3.6.
                0.v20100503.jar
(under Linux)   > ECLIPSE_RESOURCES_JAR=$ECLIPSE_PATH/plugins/org.eclipse.core.resources_3
                .6.0.R36x_v20100825-0600.jar
(under Linux)   > ECLIPSE_JOBS_JAR=$ECLIPSE_PATH/plugins/org.eclipse.core.jobs_3.5.1.R36x_
                v20100824.jar
(under Linux)   > ECLIPSE_JDT_JAR=$ECLIPSE_PATH/plugins/org.eclipse.jdt.core_3.6.1.v_A68_R
                36x.jar
```

The variable `APX_CLASSPATH` will contain the whole classpath needed to run the Ateji PX compiler in command-line mode.

```
(under Windows) > set APX_CLASSPATH=%APX_JAR%;%ECLIPSE_COMMON_JAR%;%ECLIPSE_RESOURCES_JAR%
                ;%ECLIPSE_JOBS_JAR%;%ECLIPSE_JDT_JAR%

(under Linux)   > APX_CLASSPATH=$APX_JAR:$ECLIPSE_COMMON_JAR:$ECLIPSE_RESOURCES_JAR:$ECLIP
                SE_JOBS_JAR:$ECLIPSE_JDT_JAR
```

Also, do not forget to set the classpath of the compiler. It must contain the standard library for the Ateji PX language. Otherwise the Ateji PX compiler won't be able to compile Ateji PX code successfully. The variable `APX_RT` will be the path to this library.

```
(under Windows) > set APX_RT=%APX_PATH%\lib\apxrt.jar

(under Linux)   > APX_RT=$APX_PATH/lib/apxrt.jar
```

The main entry-point for the Ateji PX JAR is: `com.ateji.bldt.internal.compiler.batch.Main`

It is now possible to compile the `HelloWorld.apx` file. The `-cp` option of java must be used to explicit the classpath. From the folder where `HelloWorld.apx` is:

```
(under Windows) > java -cp %APX_CLASSPATH% com.ateji.bldt.internal.compiler.batch.Main Hel
                loWorld.apx -cp %APX_RT%

(under Linux)   > java -cp $APX_CLASSPATH com.ateji.bldt.internal.compiler.batch.Main
                 HelloWorld.apx -cp $APX_RT
```

To run `HelloWorld`, do not forget to add the standard library for the Ateji PX language in the classpath:

```
(under Windows) > java -cp %APX_RT%;. HelloWorld

(under Linux)   > java -cp $APX_RT:. HelloWorld
```

3. How to model actors in Ateji PX

The Actor model is a mathematical model of concurrent computation. An actor application is broken up into completely independent processes. They communicate with each other through messages, thus eliminating shared state, and thus eliminating problems of data corruption and deadlocks. See wikipedia entry for more details: http://en.wikipedia.org/wiki/Actor_model

3.1. How to model an actor in Ateji PX

An actor is an object with a unique mailbox (an asynchronous input channel) on which it receives messages. In response to a message, the actor performs an local action described by a private method. More precisely, the actor executes an infinite loop :

- await a message
- select the action to perform based on the message
- perform the action

A particular message, the stopMessage, stops this infinite loop.

Actor objects are not supposed to have public fields or methods, and should avoid sharing state. This is not enforced by Ateji PX.

3.2. How to model an action in Ateji PX

An action being uniquely determined by its name, each action is represented by a private method with the same name. It may have parameters corresponding to the parameters of the private method.

3.3. How to model a message in Ateji PX

A message is an object that contains the name and the parameters of the action to be performed.

3.4. Code sample

The common behaviour of actors explained above is implemented once and for all in a specific Actor class. Users need only specify actions :

```
// Here is an example of action. Its purpose is to compute a sum
```

```
private sum = 0;
private void act_add(int x) {
  sum += x;
}
```

```
// Here is an example of a message requesting an agent to calculate its sum
sumActor.mailbox ! new Message("add", 5);
```

A complete example featuring actors is provided with the Ateji PX release in the included samples.

Chapter 5. Ateji PX Tips

1. Programming style

The programming style that best fits the Ateji PX approach is to have a lot of small concurrent tasks that communicate and synchronize with message-passing. This helps Ateji PX makes the best use of the underlying parallel hardware, and avoids the programming errors and performance problems that are common with shared memory.

Parallel blocks can be controlled by annotations. This means that whenever you identify potential parallelism, it is always good to make it explicit with the introduction of a parallel block. In the case your tests show that there is too much parallelism in your program for the underlying hardware, it is easy to limit or entirely disable some parallel blocks with a #BlockCount or #BlockSize annotation. Should you later switch to a different hardware that can accommodate a larger degree of parallelism, you will simply remove some of these annotations.

2. Read the samples

The best way to quickly master a new programming language is to start from samples. Ateji PX comes with an extensive samples library. Installation of these samples is described in the "Download and Installation of Ateji PX for Java" chapter.

Start with an overview of all samples. When you're ready to write your first program, copy and paste a similar example, and modify it to fit your needs.

3. Getting used to the compositional parallelism model

If you have ever written parallel programs using a traditional language, you are probably familiar with the "spawn" model, such as Java threads, Unix fork/join or Cilk spawn. The following pseudo-code shows how you typically run two tasks in parallel:

```
 spawn task1
 run code for task2
 wait for task1 to terminate
```

Ateji PX has no concept of spawning a task or waiting for a task to terminate. In fact, it does not have a concept of task at all. The same program would be written in Ateji PX using a parallel composition operator (hence the name compositional):

```
[
  || code for task1
  || code for task2
]
```

Similarly, there is no concept of barrier in Ateji PX. Instead, you simply write two parallel blocks one after the other.

When you need to dynamically create additional branches in response to external events, use the bang operator.

4. Parallelizing legacy Java applications for performance

The main reason for parallelizing existing code is to improve performance by enabling the use of multiple processors or cores. But remember that correctness always comes first: there is no point in improving the performance of an incorrect program.

Start by improving the sequential program. There may be important performance to be done by choosing the right algorithm and the right implementation for it.

- The right algorithm: Remember the Fibonacci sequence example. While the naive algorithm is exponential, there also exists a linear algorithm (in matrix form). In contrast, parallelizing the naive algorithm can only lead to an speedup factor bounded by the number of available cores.
- The right implementation: Consider the classical matrix multiplication example, using three nested loops (available in the Ateji PX samples library). Reordering the loops has a strong impact on cache performance.

This being said, here are the general steps to follow when parallelizing an existing Java application:

1. Transform your Java project into a Ateji PX project (right-click on the Java project name in the package explorer, select `Ateji PX Tools -> Convert to Ateji PX project`)
2. Select the source files that contain code candidate for parallelization and transform them into Ateji PX source files (in an Ateji PX project, right-click on a Java file name in the package explorer, select `Ateji PX Tools -> convert file extension to .apx`). You may want to keep a copy locally or on a teamwork server such as CVS or Subversion.
3. If not already done, write unit tests for this code
4. Identify potential parallelism, starting at the highest level, and replace sequential composition with parallel composition whenever appropriate.
5. Identify existing calls to `Thread.start()` and `Thread.join()`, and replace them with parallel blocks.
6. Run the unit tests and measure performance.
7. It may happen that a parallel program actually runs *slower* than its sequential equivalent. The two most likely culprits are too much parallelism or poor cache usage.
   - When there is too much parallelism, the overhead of creating branches may become noticeable. Use parallel block indications such as `BlockCount` and `BlockSize` to reduce the actual number of branches.
   - Poor cache usage is due to an excess of memory sharing between branches, where every write access invalidate the caches of other branches. When this happens, identify which objects or variables are shared between branches, and try to remove sharing. It is often possible to replace shared variables by message passing.
8. Visualize the run-time behavior of the code with a simple tool such as Windows task manager. If the code does not keep all processors busy, there is probably room for more improvement. Start again and identify potential parallelism at lower levels.

5. Testing parallel programs

Parallel programs are inherently non-deterministic, and this has strong implications for unit testing. Remember the "Hello World" example: you cannot use directly the output of the program as an indicator of correctness, since the program may have different outputs and still be correct.

A standard approach is to memoïze the results in a sorted data structure, then display the results. For instance, this is you could test the "Hello World" program:

```
class HelloWorld
{
  public static void main(String[] args)
  {
    int printedHelloCount = 0;
    int printedWorldCount = 0;
    [
      || printedHelloCount++;
      || printedWorldCount++;
    ]
    System.out.println(printedHelloCount + " " + printedWorldCount); // should print 1 1
  }
}
```

Chapter 6. Ateji PX vs. OpenMP

This chapter illustrates the major similarities and differences between Ateji PX and OpenMP. Its aim is to provide a short learning path to developers already familiar with OpenMP. If this is not your case, you may safely skip this chapter.

1. Ateji PX vs. OpenMP

OpenMP is a common implementation of data parallelism with the Fortran and C/C++ languages. Ateji PX provides a similar level of expressiveness as OpenMP, although with a different underlying model.

- OpenMP relies on an underlying sequential language with no notion of parallelism. The OpenMP pragmas attempt to extract parallelism from commonly used patterns. In contrast, Ateji PX introduces parallelism as a first-class language notion.
- Because OpenMP relies on a sequential language, little can be done by the compiler to guarantee the correctness of parallel programs. Ateji PX will provide in a next version a proof system that can guarantee the correctness of parallel composition.
- The basic elements of parallelism in OpenMP are threads. Ateji PX has no notion of threads, but instead provides composable operators for expressing parallelism and non-determinism.
- OpenMP preserves compatibility and stability by using pragmas that are simply disregarded by non OpenMP compilers. Ateji PX provides compatibility and stability by source-to-source translation to the underlying language.

In addition, Ateji PX provides constructs for task parallelism, message-passing, data-flow and distributed models of computation, and is not limited to shared-memory architectures.

2. Threads

> “Concurrency in software is difficult. However, much of this difficulty is a consequence of the abstractions for concurrency that we have chosen to use. The dominant one in use today for general-purpose computing is threads. But non-trivial multi-threaded programs are incomprehensible to humans.”
>
> In "The Problem with Threads", Edward A. Lee, Technical Report EECS-2006-1, University of California at Berkeley

There are no threads in Ateji PX. Unlike the threads model, parallelism in Ateji PX is compositional, and takes the form of parallel blocks containing parallel branches. The Ateji PX language specifies the behavior of parallel branches, which remains the same for all implementations.

Ultimately, parallel branches are mapped to processors threads in some way, because threads is the only parallelism construct offered by current mainstream processors at the hardware level. The difference between Ateji PX branches and processor threads is akin to the difference between a high-level programming language and a hardware-level assembly language.

The mapping from parallel branches to hardware threads may be complex and dependent on a specific implementation. In consequence:

- you cannot assume that an Ateji PX parallel branch runs in its own thread
- you cannot assume that an Ateji PX parallel branch does not run in its own thread

You normally won't ever need to even think about threads, except in one case: when using Java thread primitives together with Ateji PX code.

The actual number of hardware threads is handled by the Ateji PX runtime, which tries to make the best use of the hardware. When needed, it is possible to control the level of parallelism at the parallel block level using indications (see ...).

There are no thread ID in Ateji PX, since there are no threads. When you need to identify different branches, use a variable introduced in the qualifier list for the branch:

```
[
  || (int i : N) {
      System.out.println("Hello from branch " + i);
  }
]
```

Similarly, all thread-related OpenMP API methods are irrelevant for Ateji PX.

3. The 'parallel' directive

The basic OpenMP parallel loop:

```
 #pragma omp parallel for
 for (i = 0; i < N; i++) {
      a[i] = 2 * i;
 }
```

is written in Ateji PX as:

```
[
  || (int i : N) {
      a[i] = 2 * i;
  }
]
```

Syntactic similarities aside, the OpenMP version creates threads while the Ateji PX version creates branches. This has many implications on correctness and simplicity of programs.

Branches may be indexed with qualifier lists containing more than one generator, and possibly containing filters:

```
[
   // one branch for each (i, j) s.t. i > j
   || (int i : N, int j : M, if (i > j) ) {
       a[i][j] = 2 * i;
   }
]
```

Ateji PX can have multiple branches in a single parallel block:

```
[
   || System.out.println("Hello"); // a single branch
   || (int i : N) {                // N branches, indexed by i
       a[i] = 2 * i;
   }
   || (int j : M) {                // M branches, indexed by j
       b[j] = 3 * i;
   }
]
```

4. Schedule

The Ateji PX equivalent of OpenMP schedules are parallel block annotations.

BlockSize and BlockCount control the amount of parallelism. Without indications, this code

```
[
  || (int i: I, int j: J) { // creates I*J branches
```

```
        ... do something with element indexed by i,j
    }
]
```

creates `I*J` threads. With a `#BlockCount(N)` indication, exactly `N` branches will be created:

```
[
  || (#BlockCount(N), int i: I, int j: J) { // creates N branches
        ... do something with element indexed by i,j
  }
]
```

5. Private

There is no specification of private variables in Ateji PX, since the compiler already knows which variables are local to the branches. The scoping rules are the same as in the underlying language, namely variables introduced inside a branch are local to the branch.

6. Synchronization

Ateji PX uses the standard Java constructs for synchronization: locks, synchronized statements, volatile and atomic variables, etc. Since two Ateji PX branches may possibly belong to different threads, you must take the same care when sharing memory between branches than between threads.

7. Loop-carried dependencies

Future releases of Ateji PX will analyze loop-carried dependencies.

8. Tasks

Ateji PX makes no difference between data parallelism and task parallelism. Both are modeled by branches in a parallel block. A close approximation to an OpenMP task is an instance of `Runnable` that can be passed around and executed in Ateji PX parallel branches.

Chapter 7. Troubleshooting

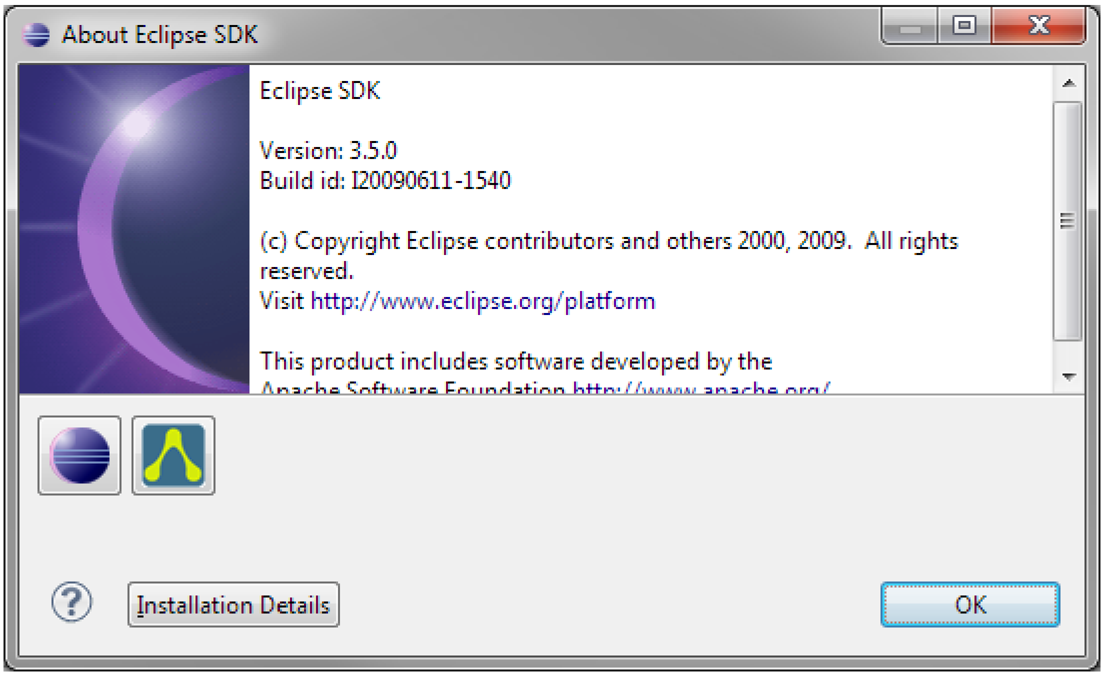
If you face any trouble with Ateji PX, you can obtain support from the Ateji team by using the Ateji PX forums at http://forums.ateji.com/atejipx.

But, please, first complete the following verification steps.

1. Make sure your Eclipse version matches Ateji PX system requirements

Ateji PX requires Eclipse SDK version 3.4 (Ganymede), version 3.5 (Galileo), version 3.6 (Helios) or version 3.7 (Indigo) running on a Java Runtime 1.5 or later.

To check your Eclipse version, select `Help > About Eclipse`:



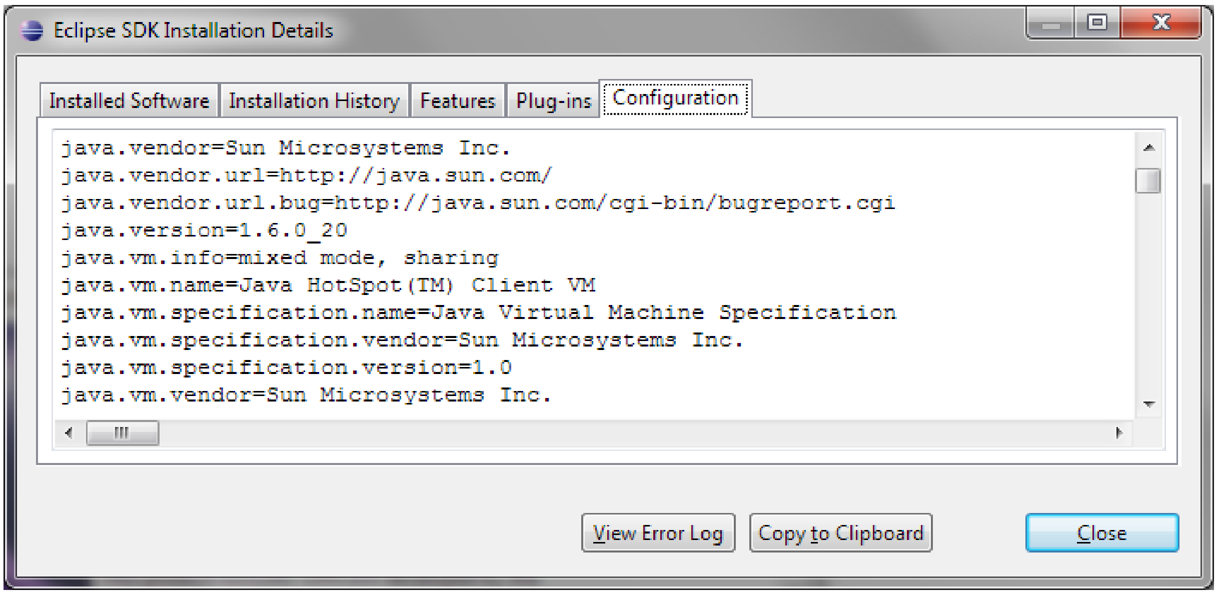
Eclipse SDK

Version: 3.5.0
Build id: I20090611-1540

(c) Copyright Eclipse contributors and others 2000, 2009. All rights reserved.
Visit http://www.eclipse.org/platform

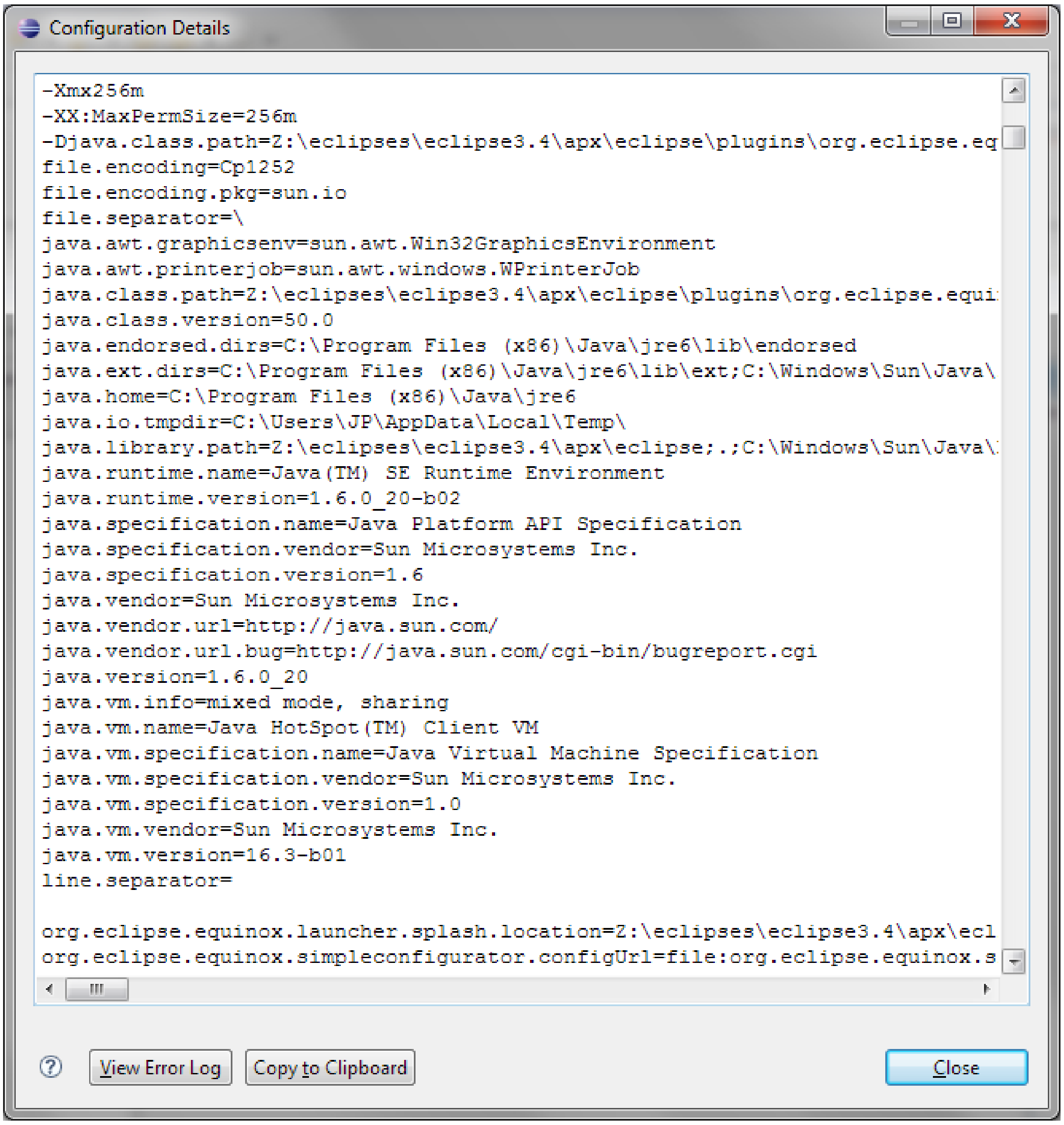
This product includes software developed by the

If you are using Eclipse 3.5 (Galileo), 3.6 (Helios) or 3.7 (Indigo) you can then click on the "Installation Details" button to see details about the Java Runtime on the `Configuration` tab. Look for `java.version`.



```
java.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport.cgi
java.version=1.6.0_20
java.vm.info=mixed mode, sharing
java.vm.name=Java HotSpot(TM) Client VM
java.vm.specification.name=Java Virtual Machine Specification
java.vm.specification.vendor=Sun Microsystems Inc.
java.vm.specification.version=1.0
java.vm.vendor=Sun Microsystems Inc.
```

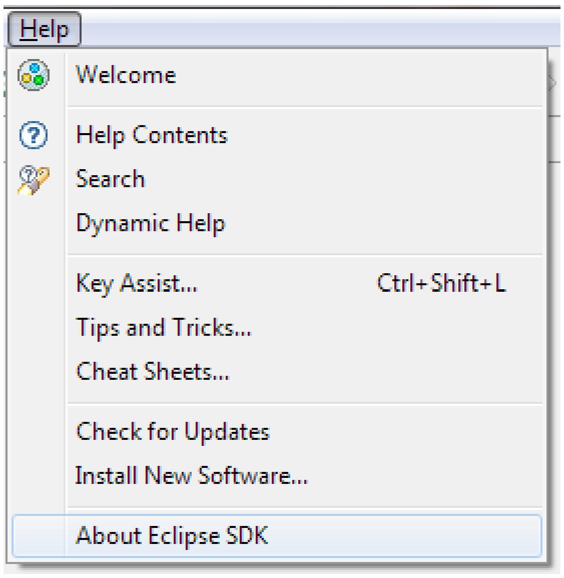
If you are using Eclipse 3.4 (Ganymede) you can then click on the "Configuration Details" button to see details about the Java Runtime. Look for `java.version`.



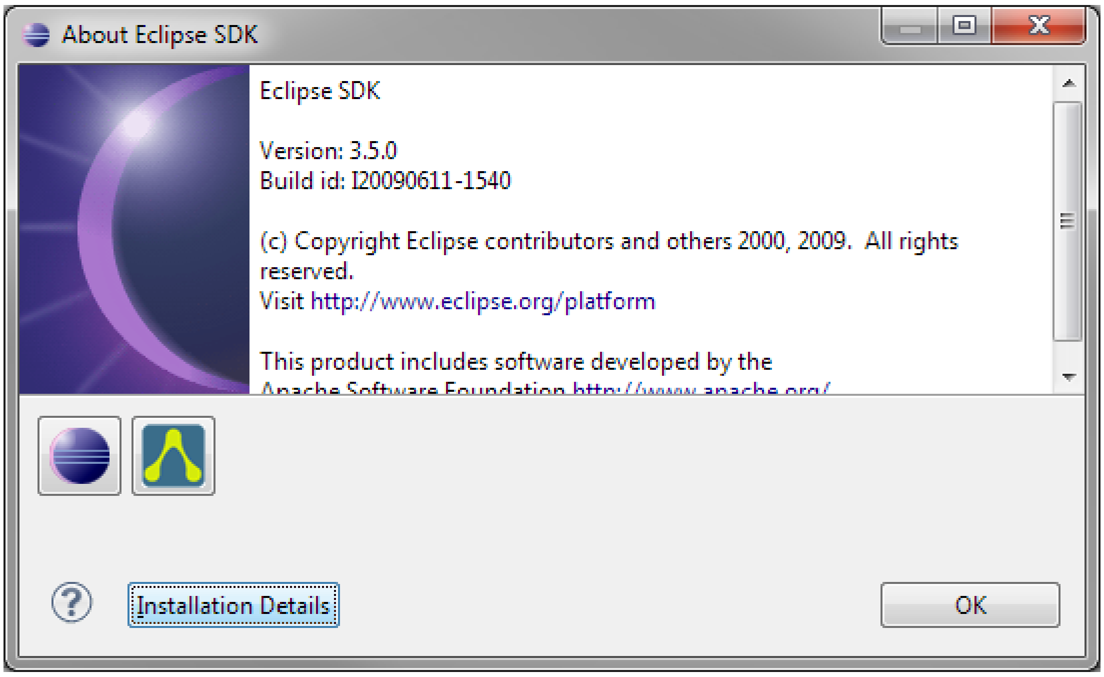
```
-Xmx256m
-XX:MaxPermSize=256m
-Djava.class.path=Z:\eclipses\eclipse3.4\apx\eclipse\plugins\org.eclipse.eq
file.encoding=Cp1252
file.encoding.pkg=sun.io
file.separator=\
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
java.awt.printerjob=sun.awt.windows.WPrinterJob
java.class.path=Z:\eclipses\eclipse3.4\apx\eclipse\plugins\org.eclipse.equi
java.class.version=50.0
java.endorsed.dirs=C:\Program Files (x86)\Java\jre6\lib\endorsed
java.ext.dirs=C:\Program Files (x86)\Java\jre6\lib\ext;C:\Windows\Sun\Java\
java.home=C:\Program Files (x86)\Java\jre6
java.io.tmpdir=C:\Users\JP\AppData\Local\Temp\
java.library.path=Z:\eclipses\eclipse3.4\apx\eclipse;.;C:\Windows\Sun\Java\
java.runtime.name=Java(TM) SE Runtime Environment
java.runtime.version=1.6.0_20-b02
java.specification.name=Java Platform API Specification
java.specification.vendor=Sun Microsystems Inc.
java.specification.version=1.6
java.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport.cgi
java.version=1.6.0_20
java.vm.info=mixed mode, sharing
java.vm.name=Java HotSpot(TM) Client VM
java.vm.specification.name=Java Virtual Machine Specification
java.vm.specification.vendor=Sun Microsystems Inc.
java.vm.specification.version=1.0
java.vm.vendor=Sun Microsystems Inc.
java.vm.version=16.3-b01
line.separator=

org.eclipse.equinox.launcher.splash.location=Z:\eclipses\eclipse3.4\apx\ecl
org.eclipse.equinox.simpleconfigurator.configUrl=file:org.eclipse.equinox.s
```

2. Make sure the Ateji PX plug-in is correctly installed in Eclipse 3.5/3.6/3.7 (Helios)

Click on `Help > About Eclipse SDK`:



Then click on `Installation Details`:



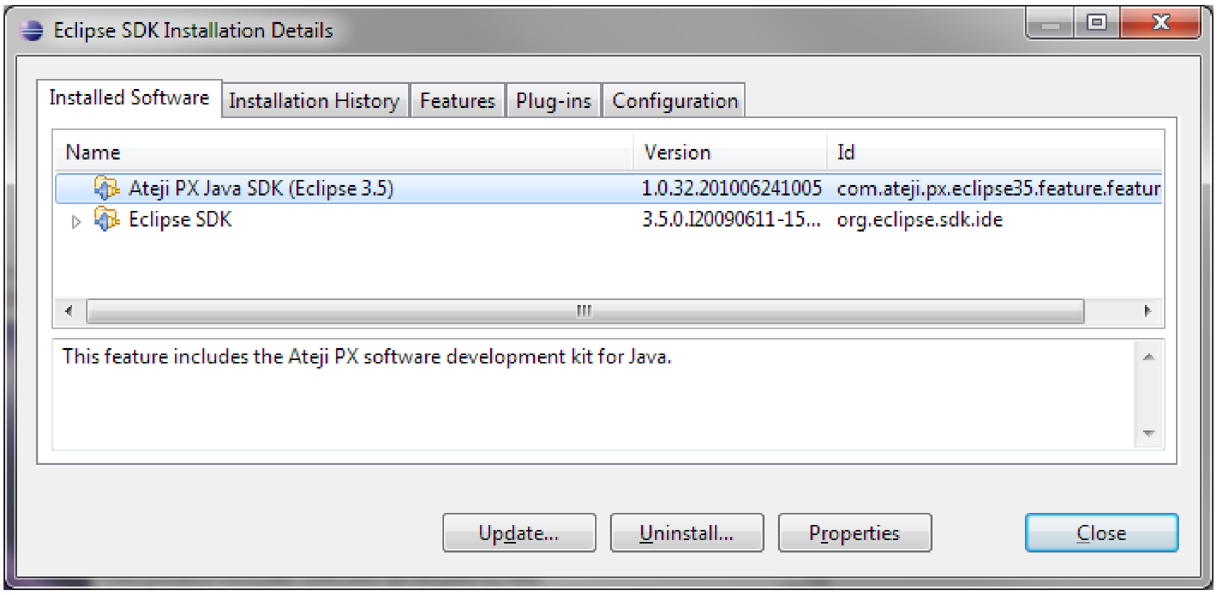
Eclipse SDK

Version: 3.5.0
Build id: I20090611-1540

(c) Copyright Eclipse contributors and others 2000, 2009. All rights reserved.
Visit http://www.eclipse.org/platform

This product includes software developed by the
Apache Software Foundation http://www.apache.org/

In the `Eclipse SDK Installation Details` dialog, the Ateji PX plug-in should appear in the `Installed Software` tab:

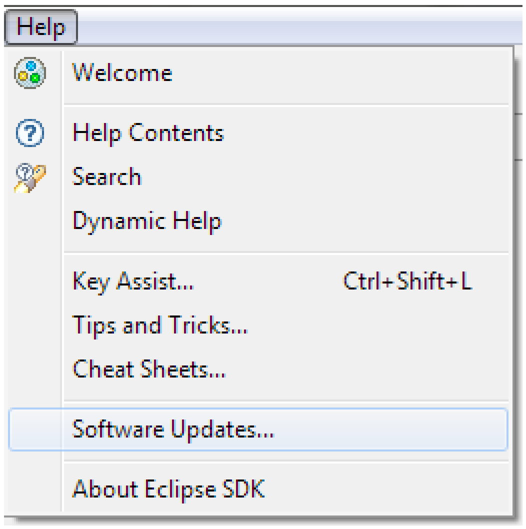


Name	Version	Id
Ateji PX Java SDK (Eclipse 3.5)	1.0.32.201006241005	com.ateji.px.eclipse35.feature.featur
Eclipse SDK	3.5.0.I20090611-15...	org.eclipse.sdk.ide

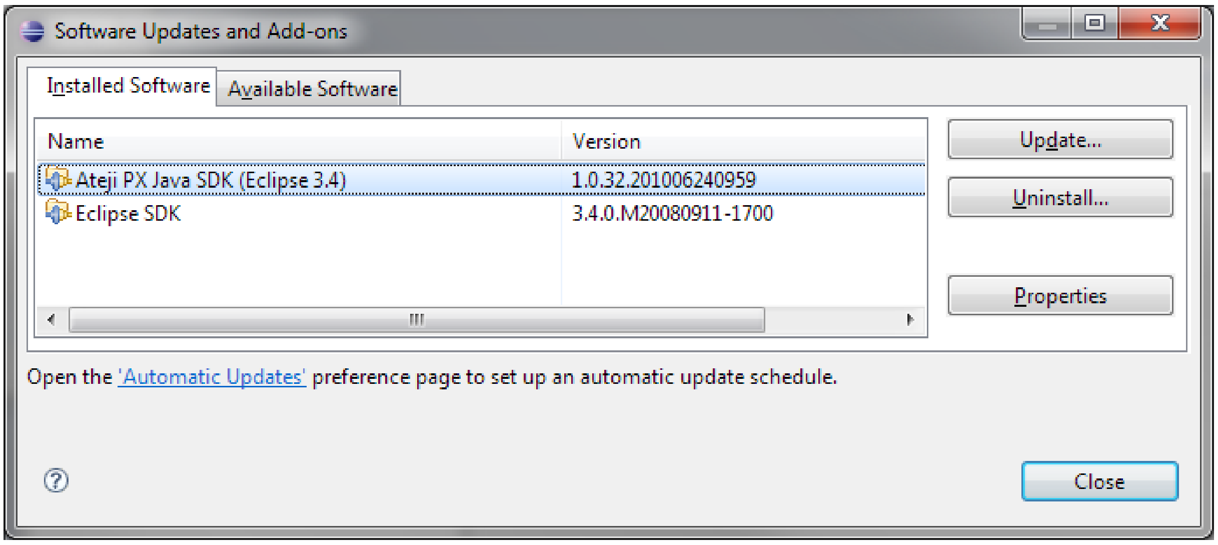
This feature includes the Ateji PX software development kit for Java.

3. Make sure the Ateji PX plug-in is correctly installed in Eclipse 3.4 (Ganymede)

Click on `Help > Software Updates...`:



In the `Software Updates and add-ons` dialog, the Ateji PX plug-in should appear:


Software Updates and Add-ons

Installed Software | Available Software

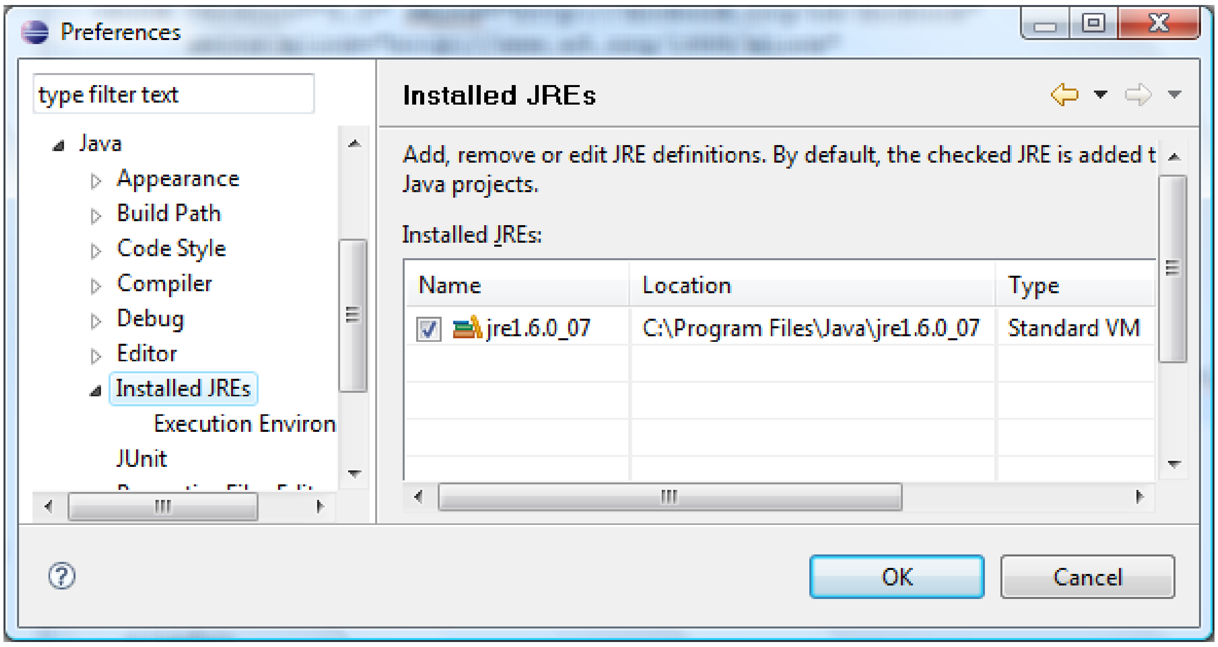
Name	Version
Ateji PX Java SDK (Eclipse 3.4)	1.0.32.201006240959
Eclipse SDK	3.4.0.M20080911-1700

Open the 'Automatic Updates' preference page to set up an automatic update schedule.

4. Make sure your Java compiler matches Ateji PX system requirements

Ateji PX requires Java 1.5 or later.

- Select `Window > Preferences...`, then choose `Java > Installed JREs`. Then check that the selected JRE is version 1.5 or more.

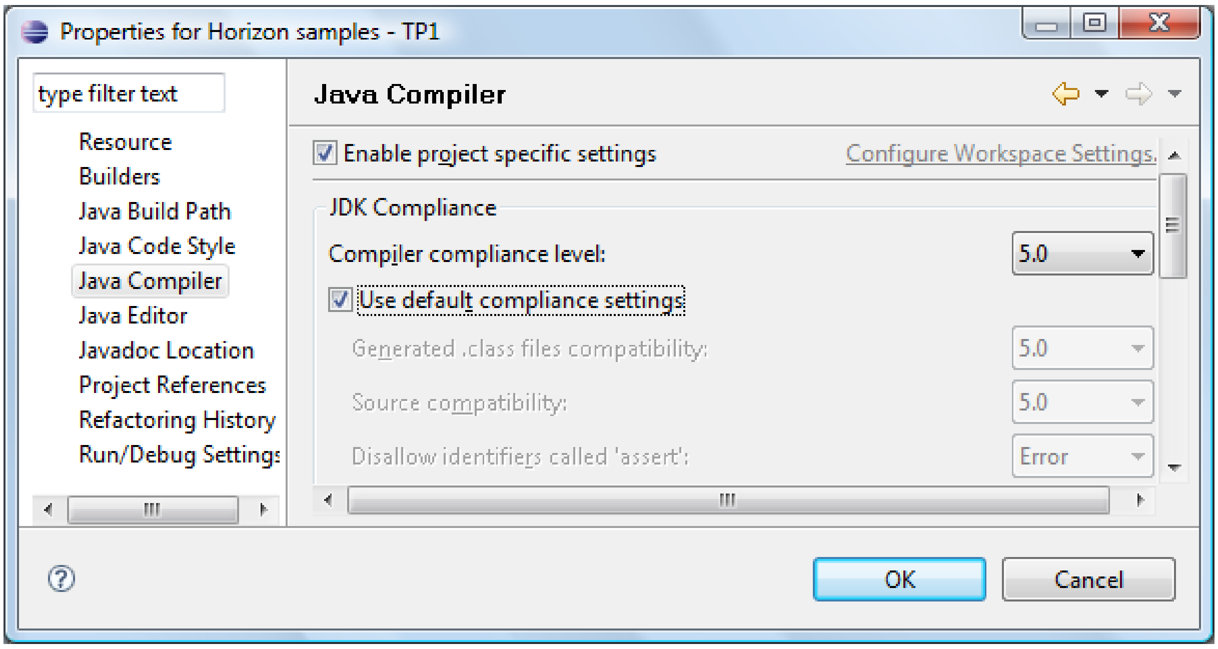

Installed JREs

Add, remove or edit JRE definitions. By default, the checked JRE is added t
Java projects.

Installed JREs:

Name	Location	Type
✓ jre1.6.0_07	C:\Program Files\Java\jre1.6.0_07	Standard VM

- Then right-click on your project. Select `properties` then choose `Java Compiler`. Make sure that the project compliance is `5.0`:



Properties for Horizon samples - TP1

type filter text

- Resource
- Builders
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Javadoc Location
- Project References
- Refactoring History
- Run/Debug Settings

Java Compiler

☑ Enable project specific settings — Configure Workspace Settings...

JDK Compliance

Compiler compliance level: 5.0

☑ Use default compliance settings

Generated .class files compatibility: 5.0

Source compatibility: 5.0

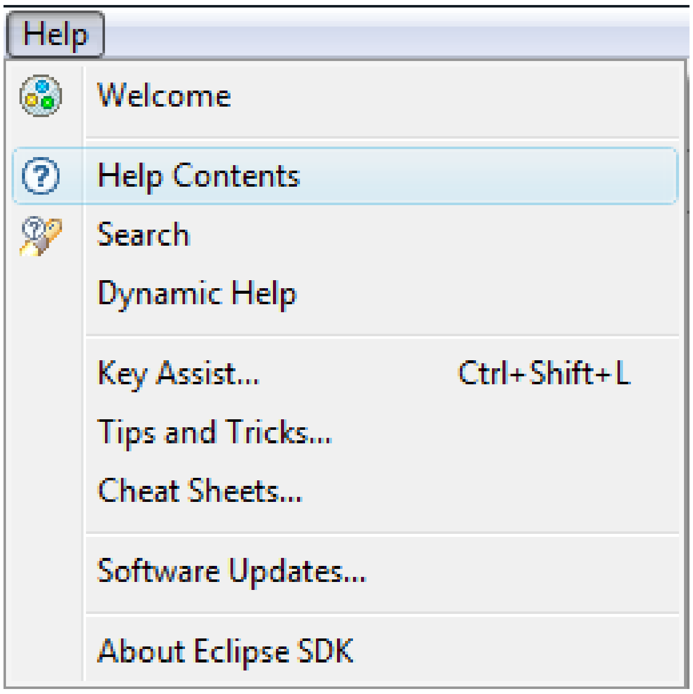
Disallow identifiers called 'assert': Error

OK Cancel

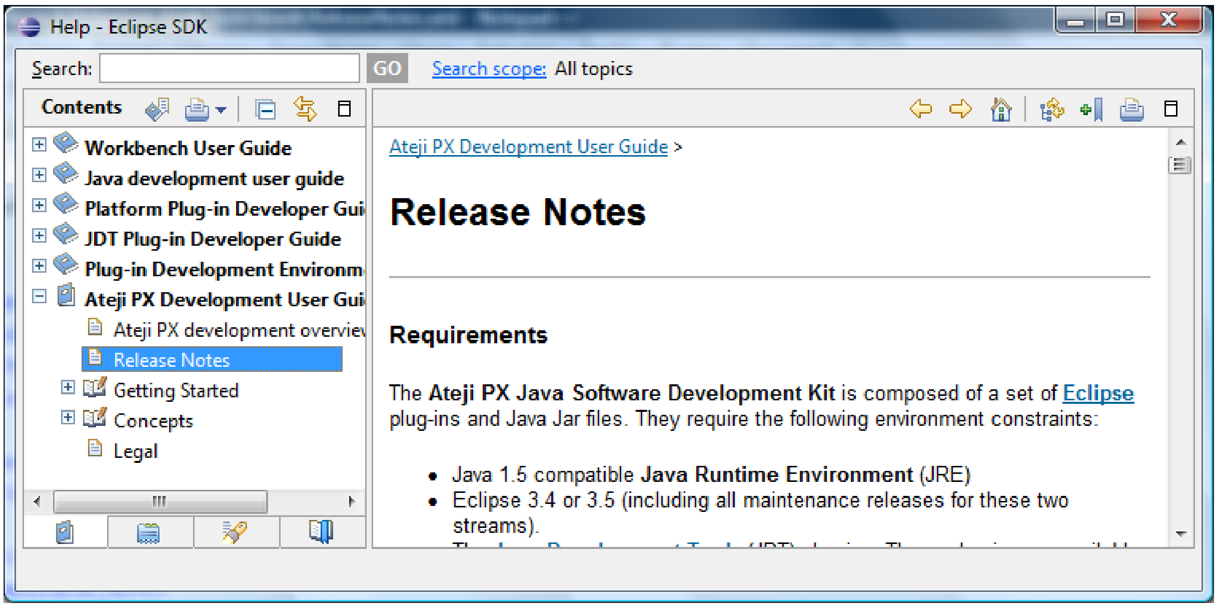
Chapter 8. Release notes

Release notes are included in the Ateji PX plug-in.

Select `Help > Help Contents`:



Unfold `Ateji PX Development User guide` and select `Release Notes`:


Help - Eclipse SDK
Search:
GO
Search scope: All topics
Contents
Workbench User Guide
Java development user guide
Platform Plug-in Developer Gui
JDT Plug-in Developer Guide
Plug-in Development Environm
Ateji PX Development User Gui
Ateji PX development overvie
Release Notes
Getting Started
Concepts
Legal
Ateji PX Development User Guide >

Release Notes

Requirements

The **Ateji PX Java Software Development Kit** is composed of a set of Eclipse plug-ins and Java Jar files. They require the following environment constraints:

- Java 1.5 compatible **Java Runtime Environment** (JRE)
- Eclipse 3.4 or 3.5 (including all maintenance releases for these two streams).

Chapter 9. The Ateji PX Community

The Ateji PX forum at http://forums.ateji.com/atejipx is the place to get support from the Ateji team, to exchange ideas with the community of Ateji PX users, and to share your experience.

- Technical Support: The official Ateji PX technical support board answered by the Ateji team.
- Success Stories: Share the highlights and successes you've had while using Ateji PX.
- Suggestions and Requests: Make your suggestions and/or requests for new features.
- Bug Reports: Notify what you believe to be bugs or errors.
- Announcements: Come here to read about anything related to Ateji PX.

Chapter 10. Legalese

1. Names and Trademarks

Ateji and Ateji PX are trademarks of Ateji SAS. Java is a trademark of Sun Microsystems Inc. Eclipse is a trademark of the Eclipse Foundation.

2. Patents

Ateji PX is covered by international patent applications.

3. Third party licenses

3.1. Eclipse Project

This offering is based on technology from the Eclipse Project.
Visit http://www.eclipse.org

A copy of the EPL is available at http://www.eclipse.org/legal/epl-v10.html

3.2. Apache Software Foundation

This product includes software developped by the Apache Software Foundation http://www.apache.org.
The BCEL library can be obtained from http://jakarta.apache.org/bcel/

3.3. Beaver

This product uses Beaver - a LALR Parser Generator (http://beaver.sourceforge.net).