



Eclipse b3

The penultimate guide

Henrik Lindberg, Cloudsmith Inc.

Eclipse b3: The penultimate guide

by Henrik Lindberg

0.3 - Second Major Draft - describing first implementation of b3.

Copyright © 2010 Cloudsmith Inc. All Rights Reserved.

This book and the accompanying materials are made available under the terms of the Eclipse Public License v1.0 which is available at <http://www.eclipse.org/org/documents/epl-v10.html>.



Dedication

This guide is dedicated to all software developers who have voiced their frustration with manually putting software build systems together, and to all early adopters that have voiced their frustration over the lack of examples and documentation when trying to construct an automated system.



Stew has his first build tool experience.

Table of Contents

Preface	viii
Why use b3?	viii
Why read this book	ix
This book's audience	ix
Conventions used in this book	ix
Getting examples from this book	x
Request for comment	x
Acknowledgements	x
I. Introduction	1
1. Introduction to b3	2
Functional Overview	2
Getting Components	3
The Build Unit	8
Builders	9
Builder source, input and output	10
More about builders	10
Builder functions	11
Turning something into a build unit	11
Advising units	12
Summary	13
II. b3 reference	15
2. The b3 expression language	16
Intro	16
General information	16
Files	16
Structure	16
Comments and documentation	17
Types	18
Literals	18
Importing	22
What can be imported	23
Functions	23
Defining a function	23
Function examples	24
Expressions	25
Expression and Expression Block	25
Operators - precedence	26
The '.' operator — feature access	27
The [] operator - indexed/keyed access	27
Call expression	27
Increment and Decrement	28
Not operator	28
New expression	28
Sequence operator	29
*, % and /	30
+ and -	30
Relational operators	30
Matches operator	30
instanceof operator	31
Logical connectives && and 	31
Variables and Constants	31
Assignment operations	31
If expression	31
Switch expression	31
Try expression	32

Throw expression	33
Cache expression	33
Typecast	33
With context expression	33
Bitwise operations	34
Properties	34
Property sets	34
Loading properties from files	36
Property sets are concerns	36
Accessing properties	37
Concern	37
Function concern context	38
Proceed expression	39
With expression	40
Type system	41
System functions	41
Evaluation	42
Looping functions	42
Set functions	42
Assert function	44
3. The Build Unit	46
Build Unit	46
Unit body overview	47
Capabilities	48
Repositories	50
Containers	53
Synchronization	53
Builders	54
Input	56
Source	61
Output	62
Annotations in input, source and output	62
The builder's logic	63
The BuildSet	63
Unit & Builder Concern	64
Adding or overriding builders	64
Unit Concern	64
Builder Concern	65
Predicates in concern context	68
Concern examples	69
4. Versions	70
Omni Version introduction	70
b3 and omni version	70
b3's named formats	71
Version ranges	71
III. Examples	72
5. Example 1 - TBD	73
IV. Appendix	74
A. Installation	76
Installing for Eclipse SDK	76
Installing the Headless Product	77
Connectors	79
Subversion (SVN)	79
B. Eclipse	80
Eclipse technology	80
Equinox	80
Platform	80
Java Development Tools (JDT)	80

Plugin Development Environment (PDE)	80
Rich Client Platform (RCP)	81
p2	81
The Eclipse component types	81
Plugins, features and OSGi bundles	81
Fragments	82
Products	82
The Workspace	82
The Target Platform	83
Launch configuration	83
ANT	83
C. p2	85
The Installable Unit	85
Metadata repository	86
Artifact repository	86
Combined / co-located repositories	86
Profile	86
p2 internals	86
Categories	87
Publishing	87
Installing	88
The SDK agent	88
The director application	89
The p2 Installer	89
The EPP wizard	89
The Buckminster installer	89
Shipping	90
Summary	91
D. Omni Version Details	92
Introduction	92
Background	92
Implementation	93
Version	93
Comparison	93
Raw and Original Version String	94
Omni Version Range	94
Other range formats	94
Format Specification	95
Format Pattern Explanation	97
Examples of Version Formats	99
Tooling Support	101
More examples using 'format'	102
FAQ	103
Resources	105

List of Figures

1.1. b3 from 10.000 ft	2
1.2. Transitive Materialization	3
1.3. Repositories	3
1.4. Federation of Repositories	4
1.5. Dynamic repository selection	5
1.6. Materialization Types	5
1.7. Telling b3 what to get	6
1.8. Ordering at “Bill’s Better Burgers (b3)”	6
1.9. Ordering at the B3 Deli	7
1.10. Getting units — summary	8
1.11. Build Unit	8
1.12. Builders	9
1.13. b3 Summary	13
1.14. b3 headless	13
C.1. Anatomy of an IU	85
C.2. p2 in action	87

List of Tables

2.1. regexp options	19
2.2. Operators	26
2.3. bitwise funtion	34



List of Examples

4.1. An OSGi version expressed in raw	70
---	----



Preface

Software development is becoming software assembly, with components sourced from around the world based on a wide range of implementation technologies. Building, assembling, packaging, and provisioning software is getting increasingly more complex while the tools in the domain have undergone little change.

A point solution, such as the Eclipse Plug-In Development Environment (PDE) does a great job of streamlining development componentized plug-ins and feature-sets when using the Eclipse IDE interactively. However, the PDE manages only those components implemented as Eclipse plug-ins, and uses a different way of building when automating builds in “headless fashion”. There is also only limited support in Eclipse for materializing the project workspace per se — i.e. fulfilling all external and internal component dependencies.

The objective for Eclipse b3 is to simplify software build and assembly by leveraging and extending the Eclipse platform to make mixed-component development as efficient as plug-in development, and to make automated building as simply a choice of invoking the one and only build definition from within the graphical user interface, or from the command line. To accomplish this, b3 focuses on the following:

- introduces technology neutral way of describing a development project’s component structure and dependencies based on the Eclipse Modeling Framework EMF.
- provides a mechanism for materializing source and binary artifacts for a project of any degree of complexity and...
- builds the end result by orchestrating the execution of built-in and user-provided build and test actions.

Why use b3?

As a developer, you want to stay focused on the construction of your code, you expect it to be built interactively giving you instant error feedback. Once your code compiles, you expect to instantly be able to run/debug it — and when you make changes to the code it hot deploys into the running instance. At some point the edit/debug cycle is over — you have a set of components, and unit tests.

But you’re not really done, of course. You still need to share what you’ve done so it can be integrated and built on a build server, tested, fixed, rebuilt, retested etc. The vision for b3 is simple — the system should just take care of all this for you automatically!

Most of the information needed is already formally expressed in your code, so b3 can figure out a lot about the components and how things should be put together. There are certain choices you made as a developer that are almost impossible for b3 to figure out on its own. So, a little work is still required on your part. But a lot less than before b3. Another important set of benefits comes from b3’s ability to run the same actions both interactively in the IDE and headlessly on a server. This is particularly useful for organizations implementing continuous build integration and test automation, as well as for open source development where anyone should be able to build the source.

What is special about b3?

- b3 is build with EMF (Eclipse Modeling Framework), and XText - this has many advantages, most importantly:
 - The use of EMF models makes it easy to transform build related information to and from b3. All aspects of the b3 build is captured in models (including the build logic).
 - The use of XText gives the b3 model a concrete human friendly textual syntax for the parts of the build system that are not automatically generated. You can enjoy working with an editor with

syntax coloring, code completion, and all the nice features you expect from an Eclipse based text editor.

Why read this book

We've attempted to make this book a clear, concise and definitive reference. We've tried to cover the bases regarding using b3 in the most typical usage scenarios. We've also tried to provide enough detail to serve as a starting point for more specialized scenarios, including customizing b3 itself. Following are the key topics we address:

- **The general nature of b3.** Not everyone wants to jump into b3 syntax right away, so we will quickly get you up to speed on b3's architecture and what it can do for you.
- **How to get and install b3.**
- **How to get software components from various sources.** b3 provides the mechanisms to get software components in source and binary form from a variety of sources such as source code repositories, Eclipse p2 update sites, and Maven.
- **How to invoke actions** that perform builds and other common tasks.
- **Best practices** when working with Eclipse plug-in projects, and when building RCP applications.
- **Publishing** the built result so it can be consumed by users.
- **Solution cookbook** with examples of how to solve various common issues when building software.
- **Setting up continuous integration with Hudson and b3.**
- **Unit testing.**
- **Extending b3.**
- **Reference documentation.**

This book's audience

We expect that most readers have familiarity with Eclipse in general. When describing b3 features that directly related to developing Eclipse plugins, OSGi bundles in general, writing complete RCP applications, managing p2 repositories, or using b3 for C++ development, we expect the reader to have an understanding of development using the respective technology. Although we do provide introductions to the technologies surrounding b3, as it would otherwise be difficult to understand the full picture, these introductions are by no means intended to serve as anything but starting points for further explorations.

Conventions used in this book

Most books show you all the conventions used, but there are only a few things that needs to be mentioned...

Manually inserted line breaks

Examples tends to get quite wide, and line breaks must be inserted or the lines will be truncated. When this is the case, and where line break matter, we include a ↵ where the line is broken, and one or several ↵ characters on the subsequent line to denote that what follows is a continuation of the previous line. Here is an example:

```
http://somewhere.outthere.com:8080/with/long/path/and/parameters/like↵
```

```
-?thisOne=withAValue&andThisOne=withAnotherValue.↓  
-&thisThirdParameter=withYetAnitherValue.↓  
-~soForth=untilTheLineNeedsToBeBrokenUpAgain&andThenSome=extraStuffAtTheEnd
```

If you type in one of these examples, you should remove everything from the ↓ to the last ~ (inclusive) on the next line and and have no line breaks.

Replaceables

Replaceables denote text that is variable in nature — the replaceable part is something you would type, or that is generated by the system. We use the guillemots characters « and » around the part that should be replaced e.g., copy «fromName» «toName».

Getting examples from this book

The examples in this book can be obtained from the b3 source code repository. Up to date information is found at the general b3 project page at Eclipse.

The b3 project page is located at <http://www.eclipse.org/modeling/emft/b3>.

Request for comment

Please help us improve future revisions of this book by reporting any errors, bugs in examples, confusing or misleading statements, or examples that you would like to see included.

Please report issues with this book in the Eclipse Bugzilla under the category Modeling →EMFT.b3 → documentation. The b3 Bugzilla is found at <https://bugs.eclipse.org/bugs/>.

Acknowledgements

Eclipse b3 is strongly influenced by the Eclipse Buckminster project, and the experiences gained from using Buckminster for a variety of builds.

We are also very grateful to Cloudsmith Inc, our current employer, and its investors for making it possible for us to work on b3.

TO-DO: It is not yet possible to acknowledge those that helped putting this documentation together...

Part I. Introduction

This part is intended as a quick introduction to b3's functional domain which includes provisioning, building, sharing, testing and publishing software components.

If you need a primer on some of the central Eclipse concepts such as the Eclipse workspace and target platform, OSGi and the Eclipse Plugin Development Environment (PDE), and the Eclipse provisioning platform (p2), you may want to start by reading [Appendix B, *Eclipse*](#), and [Appendix C, *p2*](#), where eclipse technology is explained and put into a build domain context.

Specifically, this part discusses how b3 works.



1

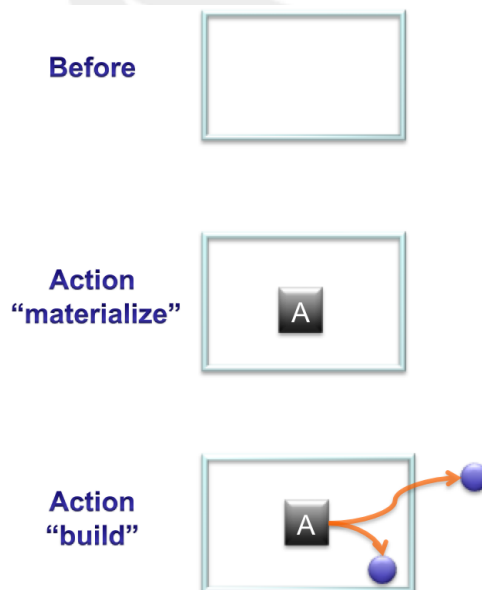
Introduction to b3

This chapter is an overview of the functionality in b3.

Functional Overview

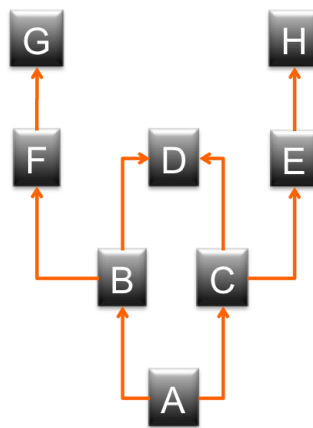
The highest level description of what b3 does is simply as follows. You want to build something, and have nothing of the material you want to build. You tell b3 to *materialize* the *units* you are going to build, and then you tell b3 to build it. This produces output within your workspace, or somewhere on disk.

Figure 1.1. b3 from 10.000 ft



Materialization fetches components so they can be worked on. *Actions* such as build can then be performed.

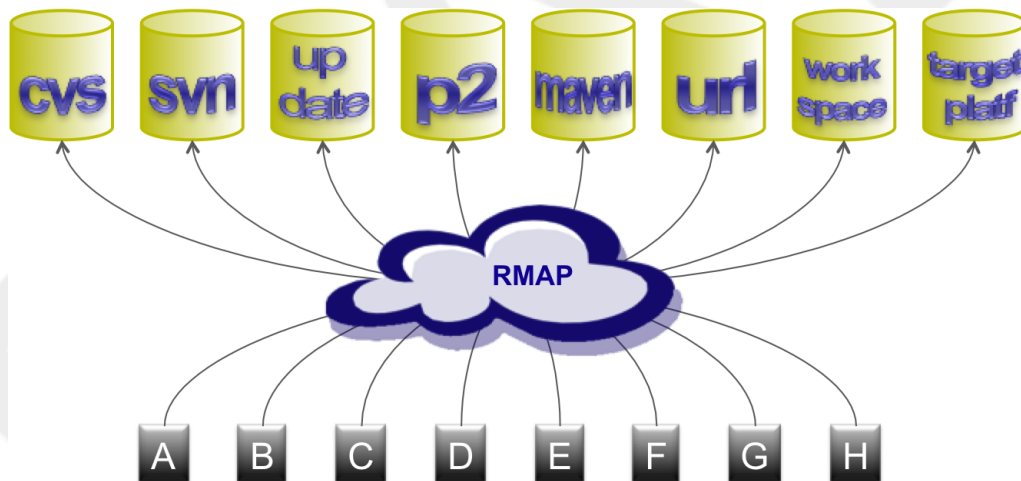
When you request the unit to build (A, in the example above), b3 will not only fetch this unit, but also resolve all of its dependencies transitively.

Figure 1.2. Transitive Materialization

When requesting unit A, it in turn requires B, and C — they both require D, B requires F, and F in turn requires G, similarly C, requires E and H.

Getting Components

The first two questions are usually, *Where does b3 get the units?* and *Where does b3 store them?*

Figure 1.3. Repositories

Units are looked up in a repository configuration which holds the rules for accessing different types of repositories. *TBD - REMOVE RMAP TEXT IN CLOUD*

When b3 needs a unit, a lookup is performed in a repository configuration. This configuration contains rules how to translate a request for a unit of some particular type and version to a repository location of a particular repository type, and how to address the component within this repository.

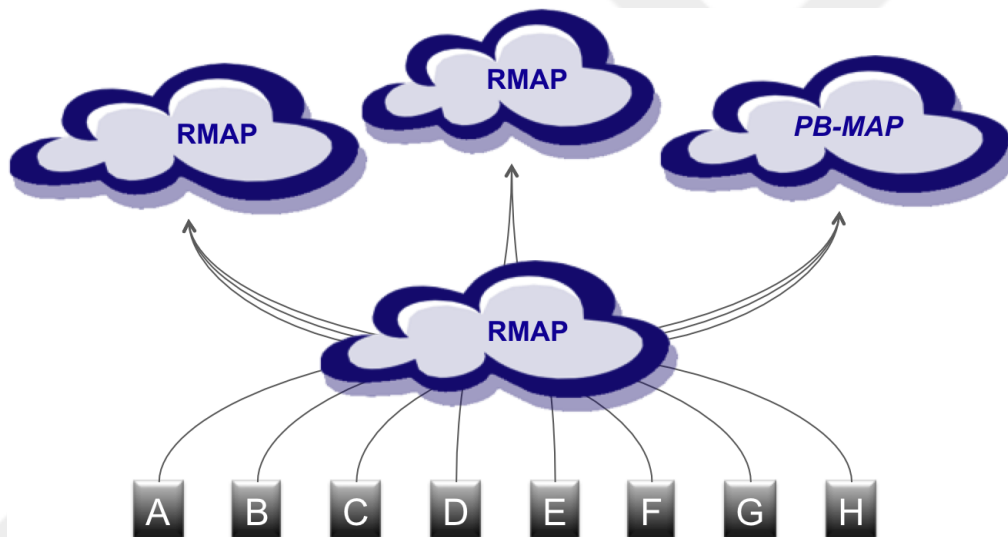
Eclipse b3 supports a wide variety of repositories, and it is possible to extend b3 with new types.

- **CVS** — it is possible to reference components found in HEAD, in branches and via timestamps.
- **SVN** — it is possible to reference components found on trunk, branches, and named tags.
- **GIT** — *specific git features worth mentioning*
- **Update Site** — components published on a Eclipse Update Site in the format specified by the Update Manager (in use up to Eclipse 3.5).

- **p2** — components available in a p2 repository can be fetched.
- **Maven** — components stored in a maven repository can be fetched.
- **URL** — a single component can be fetched from a given location.
- **Workspace** — the components currently in the workspace (probably in source form) are also available to b3's resolution process — naturally there is no need to actually fetch them, but their presence may override resolving to the same component in binary form in some other repository.
- **Target Platform** — the components in a target platform are available to b3's resolution process — these are also not fetched, but affect the resolution process.

The repository configuration does not have to be a singleton — it is possible to reference other configurations.

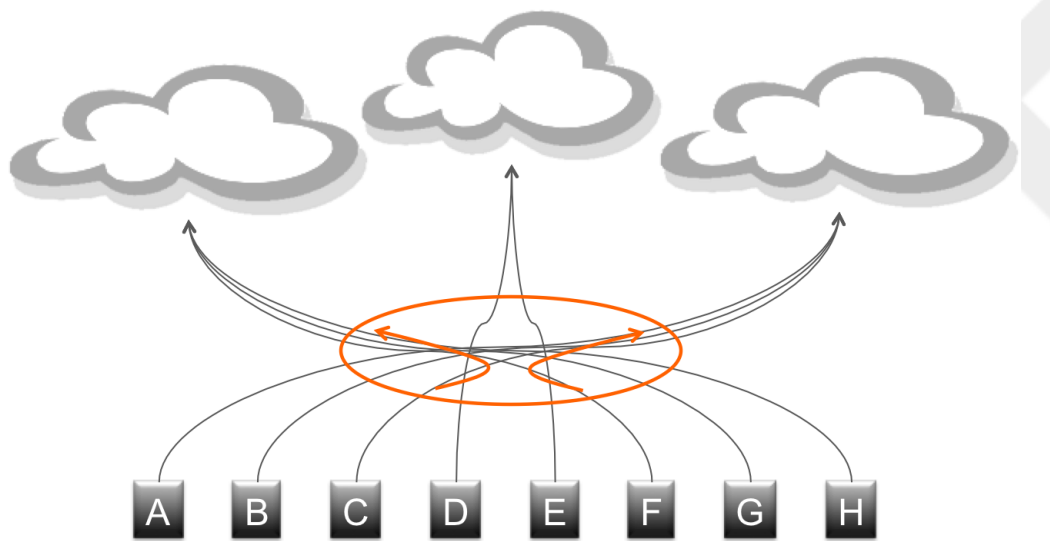
Figure 1.4. Federation of Repositories



A federation of repository configurations, including a platform base builder map.

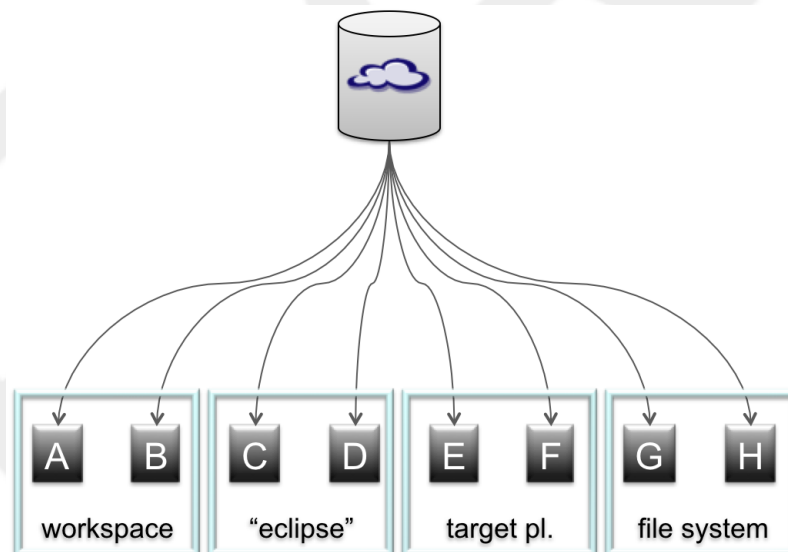
It can be useful to organize the overall repository configuration in a distributed fashion. You may want that different projects maintain their own configuration — which is especially important if projects are following different naming standards, and when they are performing refactoring of repositories. An important feature for projects at Eclipse is that the platform base builder maps are directly supported. This is important because many Eclipse projects include components from the Eclipse Orbit repository and a platform base builder map is provided for this repository, and it can be directly used. Some projects, that are currently building with the platform base builder naturally also benefit as it is easier to transition to b3 by directly being able to use existing maps¹.

¹Although not required, if you are using the platform base builder maps it is recommended that you switch to a b3 repository configuration as it is easier to maintain if you are following typical project naming standards.

Figure 1.5. Dynamic repository selection

Resolution can take different routes depending on rules and parameters.

When b3 resolves a request for a unit it can take parameters and rules into account when selecting the repository to use. You can for instance organize the repositories so that users look up units from a local repository rather than always going to a central repository, and you can do this dynamically using b3 expressions e.g., when the component name matches a regular expression (and much more).

Figure 1.6. Materialization Types

Eclipse b3 can materialize (store) fetched components in different types of containers.

When b3 materializes components, they can be stored in different types of locations. Eclipse b3 supports Eclipse related locations, and the file system, but can be extended with other types of locations.

- **Workspace** — typically projects are materialized to the workspace, but it is also possible to bind binary components (this was common practice prior to Eclipse 3.5 because of difficulties with managing the target platform)
- **Eclipse** — i.e., installing tools into an Eclipse based product such as the Eclipse SDK or an RCP application. Prior to Eclipse 3.5, this was done by using the Update Manager. Since 3.5 this is performed using p2.

- **Target Platform** — i.e., installing into a definition against which components are built. Prior to Eclipse 3.5 the target platform had to be created separately, and then referenced in later operations. In 3.5, a target platform can be dynamically created and installed into.
- **File System** — i.e., storing the component on disk.

Now you have seen how b3 gets components, and where they are stored when materialized. But how do you tell b3 what you want?

TBD - CHANGE IMAGE BELOW TO UNIT QUERY

Figure 1.7. Telling b3 what to get

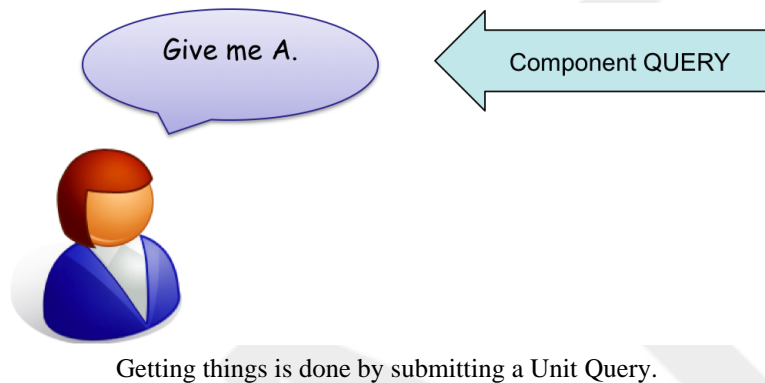
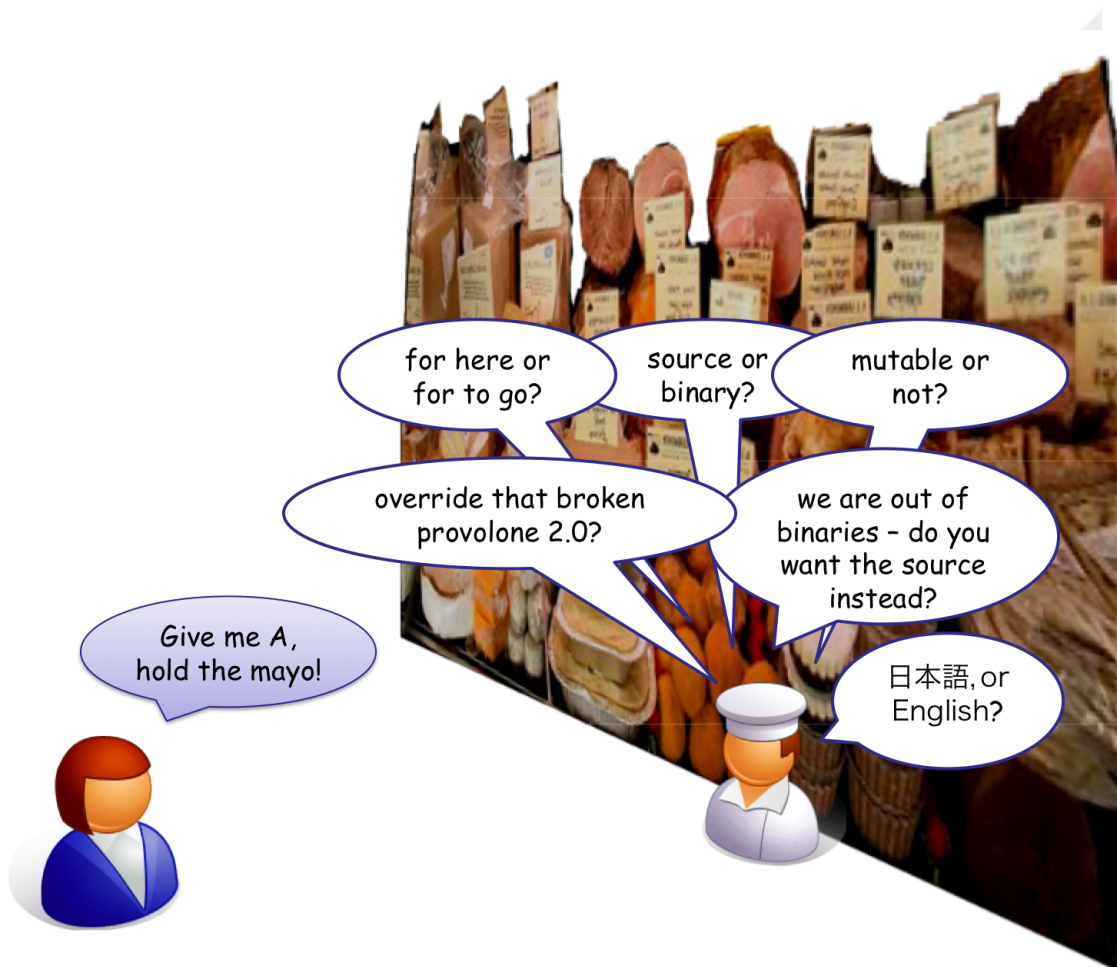


Figure 1.8. Ordering at “Bill’s Better Burgers (b3)”



Telling b3 what you want can be as easy as ordering a meal at Bills Better Burgers...

Most of the time, the only thing needed is to state the name of the component you want. Eclipse b3 will then find the latest version of the component. But sometimes you may have very detailed requirements on your meal.

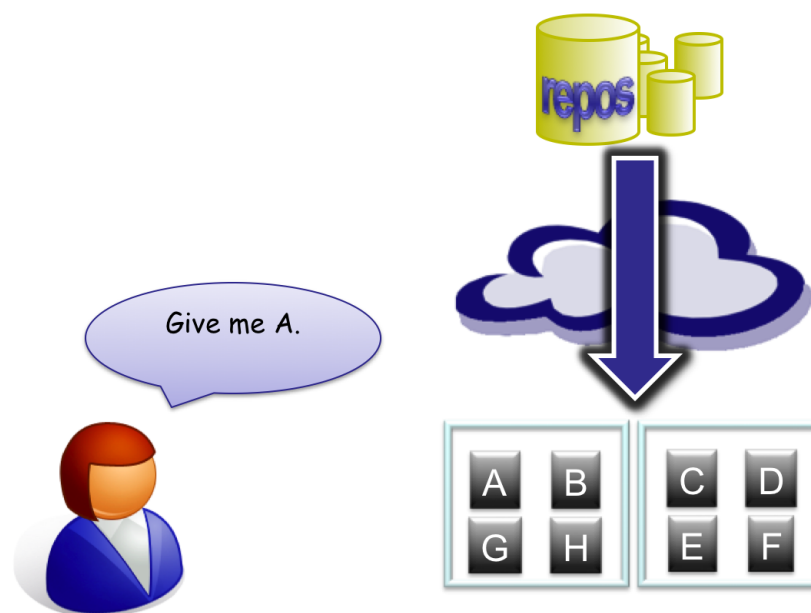
Figure 1.9. Ordering at the B3 Deli

Getting picky at the b3 Deli. (Are you sure that pepper is south Brazilian?)

As you will see later, b3 has a very powerful query mechanism, and aspect oriented programming techniques where you can specify many options and perform advanced overrides:

- Do you require source, or prefer source, but can work with binary, or only require binary form.
- Do you require source that can be modified and checked in (given that you have authority to do so naturally).
- Do you want to load some units from a branch or tag and override the default.
- Do you want to override certain unit-version combinations even if requirements in the unit say otherwise.
- You may want to specify that a search should use a particular repository for certain components — perhaps loading them from a central repository instead of a local mirror.
- You may want some units from a release repository, but some should be picked from a nightly build repository.

Eclipse b3 queries are entered and edited in the b3 text editor (just like everything else in b3). *TBD - REFERENCE TO THE EDITOR - AND SPECIFICALLY HOW TO WRITE A b3 SCRIPT THAT QUERIES.*

Figure 1.10. Getting units — summary

Summary of getting a component — a query is resolved and units fetched from repositories, and materialized into different containers.

The Build Unit

We have already introduced the term *Unit* without any further explanation. Now is the time to look a bit closer at what is meant by a unit in b3, and more specifically, a *Build Unit*, which is the primary type of unit used by b3 (the other type is p2's *Installable Unit*).

Figure 1.11. Build Unit

A Build Unit is an abstraction — a named and versioned piece of content.

A build unit is an abstraction of a (buildable) unit in a software system having a name, one or several types and a version. A build unit typically has content²— and it can exist in multiple forms — such as source or binary. When b3 obtains the definition of a unit, and subsequently its content, a *translator* matching the unit instance's physical shape is used to interpret the component's meta data and translate it into a b3 *Build Unit*. This translation takes place each time the component is requested — there is no need to save the result. This has some important benefits:

- No round trip engineering is required. The meta data at the source is used directly.
- Does not require restating already expressed facts such as dependencies.

A build unit is not tied to any particular implementation technology — b3 works just as fine with Java, C, PHP, as it does with any collection of files. Even if it is possible to turn just about anything into a leaf unit, in order to be really useful however, there must be some meta data available that describes the component and its dependencies.

²A build unit without content functions as a configuration or grouping mechanism.

Eclipse b3 has discover and translators for several meta data formats, and it is possible to add extensions for additional types. And in case you wonder, it is possible to combine different types of repositories with different types of component readers (although some combinations are nonsensical as certain type of meta data may only exist in certain types of repositories). Here is a list of available translators:

- Eclipse types: plugin, feature, product, fragments
- OSGi types: bundle
- Maven: maven POM (version 1 and 2)
- Buckminster: Buckminster's CSPEC and Component Specification Extension (CSPEX).

A Build Unit is a model. Since a Build Unit is an instance of a model, it is not difficult to write translations — there are many tools available in the modeling domain for such tasks. One especially suited is the MoDisco (model discovery) project which allows discovery of structure from source code - which can be used to detect the buildable units and their dependencies even if they are not expressed in concrete meta data.

Other ways of using modeling is to transform a high level model of your system's building blocks into executable build units.

Authoring a Build Unit. You can also author build units directly using the concrete b3 domain specific language DSL and XText based editor. Here is a very simple example of a leaf component called "apple" containing a text file and an image in a folder called docs.

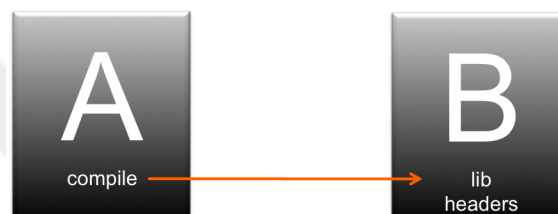
```
unit apple version 1.0.0 {
  builder content {
    source { docs/ [ facts.txt, picture.gif ] }
  }
}
```

You will see a lot more examples using the b3 DSL, as this is the easiest way to illustrate the capabilities of b3 — even if most of the units you are going to be using (at least if you are working with Eclipse and OSGi based technology) are automatically and dynamically translated into b3 models.

Builders

Build Units have *Builders* which are similar to methods of a class. A builder can be thought of as a getter-function which returns either static data, or dynamically computed / built result.

Figure 1.12. Builders



Unit A's compile builder requires the result from the lib and headers builders from unit B.

The return type of a builder is always a *Build Set* which represents a collection of files (actually collection of URI), and named/typed values (called *annotations*). So, when the lib builder in [Figure 1.12, "Builders"](#) is evaluated by the compile builder it will get a collection of the lib files in component B. These lib files could either have been statically declared, or be computed dynamically (component A does not know, nor care).

A builder can also aggregate the build results of other builders as you will see in the next section.

Builder source, input and output

A builder has three different features for declaring a build result:

source

A source declaration defines references to files (using URI notation), and includes the ability to declare additional typed annotations. If a builder has no other declared build result, the source declaration is also what is returned by the builder.

```
builder content {
  source { docs/ [ facts.txt, picture.gif ]; }
}
```

input

An input declaration declares that the builder requires input of build results from other builders. These result can come from other builders declared for the same unit, a unit having a particular name and version, or a unit that satisfies a particular requirement. If a builder has no other declared build result, the input declaration is also what is returned by the builder.

```
builder bookOfFruits {
  input {
    unit/apple/1.0.0#content;
    unit/banana/1.0.0#content;
  }
}
```

output

An output declaration looks the same as a source declaration (a set of URI, and typed annotations), but this is a declaration of what the builder's logic will (is supposed to) produce. You can think of the output declaration as a declaration of “derived files” if you like. Having just an output declaration does not do much however, expressions that actually builds (creates the derived result) must naturally also be stated (in the example below, the input is placed in a zip file as specified by the output declaration).

```
/**
 * Produces a zip file with all the content
 * from the apple and banana units.
 */
builder bookOfFruits {
  input {
    unit/apple/1.0.0#content;
    unit/banana/1.0.0#content;
  }
  output { bookOfFruits.zip; }
  zip(input, output);
}
```

More about builders

The previous section showed some very simple examples of what you can do with builders. There is much more you can do with builders, and this is explained in detail in [the section called “Builders”](#). Here are some useful things to know as you read on:

- The builder's logic uses a comprehensive expression language borrowing much from languages like Ruby, Scala, and OCL.
- The expression language is fully integrated with Java, anything that can be called from Java can also be called from the b3 expression language.
- Declaration in source, input, and output can be filtered using expressions (to reduce issues with combinatory explosion).

- Requirements can use version ranges (example above has single version requirements).
- Builders can take parameters, be marked as private to limit their visibility, and final to limit the ability to redefine them.
- Builders have asserts (pre/post and post input conditions) that can be used to assert that (typically) external processes (like a compiler) has produced what it is supposed to.
- Builders can run in sequence or in parallel (they are in fact Eclipse Jobs, and use Eclipse Job synchronization).
- Builders can have default properties, can create new properties and pass these on to downstream builders (selectively and dynamically).
- The expression language pointcut/advice mechanism allows builders to override just about anything in the builders it is calling (and what they in turn are calling). As an example, expensive asserts can be introduced dynamically when there are issues with a build.
- Builders can be declared outside of units, and applied to units using comprehensive point cut rules (unit name, version, type(s), provided and required capabilities, etc. etc.). This means that it is easy to introduce new ways to build things, troubleshoot and experiment without having access to or modifying the translation of source to model, and perhaps more importantly, it separates the concerns of “build logic” from the concerns of “structure of build material”.

Builder functions

You may have wondered how the body of a builder actually builds anything specific, like compile java code, an RCP application or a p2 repository. You have already learned that b3 has a comprehensive expression language and that it can call on anything that is available in Java — this is good news as it means that you can very easily add support for building things in new ways, but if this was the only thing available, it would also be almost as hard as writing all your build logic directly in Java.

In order to make things simple — b3 comes with support for build functions in the following domains:

- Java — compile, jar, etc.
- PDE — build bundles, features, fragments and products, pack, sign, and zip the result.
- p2 — build a repository (or aggregate several repositories).
- General — fetch files and execute system commands
- ANT — invoke ANT tasks.

Turning something into a build unit

As you have seen earlier, there is nothing you have to do if the software unit you are interested in already has meta data for which there is a translation into a build unit available (as it is for all the Eclipse related types; bundle/plugin, feature, and product).

When this is not the case what you need to do depends on if there is meta data available at all, and if the meta data is rich enough to be useful — if that is the case, you are probably best of by adding a new translator. For more information about how to extend b3 see [???](#). If however, the meta data is missing, or is poor, or you just don't want to create an extension, it is recommended to use the b3's DSL format³.

The default b3 translators expects to find an optional file inside the unit with the name of “this.b3”. This file is created with the b3 editor. The content of this b3 file is evaluated as the last step of discovery, having first translated any available meta data into a b3 model. The this.b3 then acts as additional

³Although you can use the EMF technology of your choice to produce your build models.

advice to the discovered unit. (And naturally, if there was no meta data to discover, the `this.b3` contains the full specification).

There is also a hybrid solution possible, for some reason it may not be possible to insert the `this.b3` meta data into the actual unit, and then you can author a build model in some other unit and still refer to the content in the original unit(s). This would resemble the traditional way of having a build script that contains all the knowledge of how to build, but you still benefit from the power of using b3.



Note

It is only in fairy tales a frog turns into prince by a mere kiss.

Advising units

Eclipse b3 has an extension mechanism for that allows you to “decorate” a unit with additional advice. This is useful in several situations:

- adding additional builders to a unit
- overriding faulty meta data
- adding dependencies to underspecified units
- hooking actions that should be executed throughout the various processing steps
- wrapping existing logic to do some additional work before or after the original action

Handling advice is an integral part of the b3 model and is easy to use in the b3 DSL. Here is a simple example, where a new builder is introduced:

```
/**
 * Used to advice 'Fruit' that the waste (seeds and peel)
 * can be built.
 */
concern FruitProcessing {
  builder waste(Fruit unit) {
    input { unit#peel; unit#seeds; }
  }
}
unit myunit {
  builder makeStuff() {
    input { with (FruitProcessing) unit/apple/1.0.0#waste; }
  }
}
```

You will see all the details later, but a brief explanation is that this example introduces a named concern `FruitProcessing`, which defines a new builder called `waste` that operates on build units of type `Fruit` - when evaluated, this builder combines the `peel` and `seeds` from any fruit. The new concern is used in the unit `myunit` by using a `with` expression before the reference to a particular fruit (an apple). The introduced concern is in effect for all downstream processing (i.e., it is possible to invoke the `waste` builder on all fruits (even if the example above does not illustrate this).

Here is a more advanced example:

```
context unit requires fruit.product/_/[1.0.0, 2.0.0] {
  context builder input nutrition.supplement/vitamin.C {
    - input unit/orange;
  }
}
```

In this example, the selection of build units is based on a predicate (all build units that require a `fruit.product` capability of any name (the `_` means any) in the version range `1.0.0` to `2.0.0` where the unit has an applicable builder that requires input of a *vitamin C nutritional supplement*). When such a builder is encountered, any input requirement on a build unit called `orange` is removed.

Summary

Eclipse b3 gets the building blocks of a software system called units from repositories, reading and translating them into a common build model, and then materializing them into different containers such as the workspace or target platform. When the units have been materialized b3 runs builders defined in the units (or via advice) such as building a product or a repository of plugins.

Figure 1.13. b3 Summary

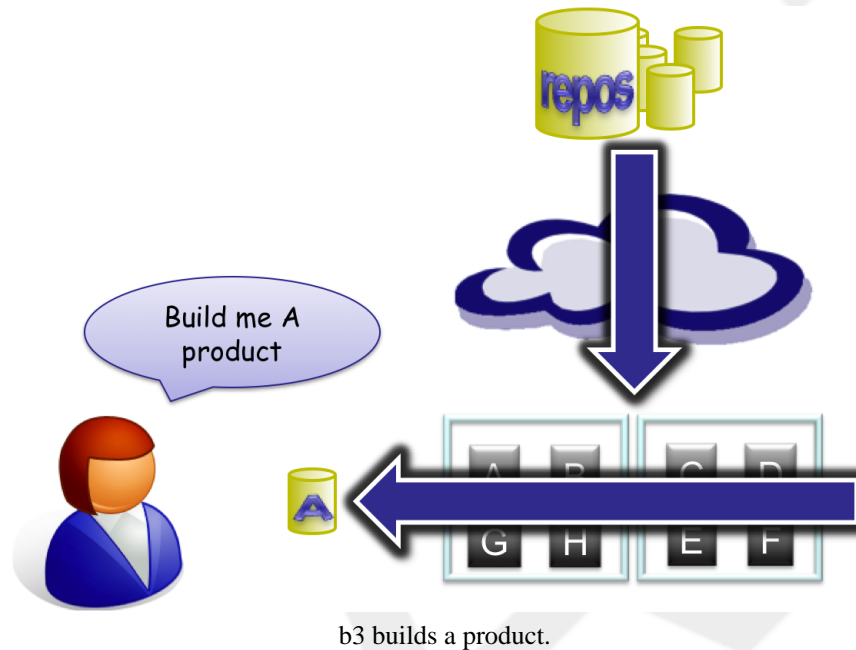
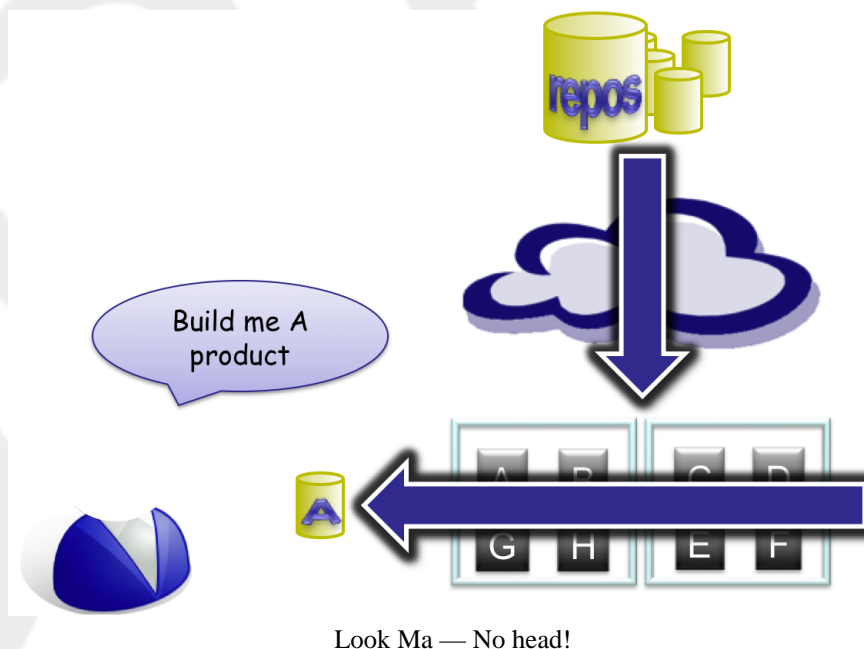


Figure 1.14. b3 headless



The [Figure 1.14, “b3 headless”](#) illustrates the most important feature of them all — the ability to build exactly the same thing in a headless configuration without having to do any additional authoring!

Reading on. You have now seen an overview of b3 and how it relates to other Eclipse technologies. You should now have a high level understanding of the capabilities. The rest of this book is not

intended to be read from start to finish (although you may still want to), but instead provide detailed drill down in the various parts, as well as presenting examples, and reference material.

DRAFT

Part II. b3 reference

In this part, we take a deeper look into b3. The chapters are not intended to be read in sequence, although we try to follow a logical sequence — starting with the general expression language capabilities (even if these has the least to do with actually building stuff, they are at the core of the b3 model). Alternatively, you may want to start by installing b3 as described in [Appendix A, *Installation*](#), and then running through some of the examples in [Part III, “Examples”](#).



2

The b3 expression language

In this chapter we take a closer look at the generic part of the b3 expression language. If you like to read things in top-down fashion you probably want to start by reading [Chapter 3, *The Build Unit*](#) and skip back to this chapter when encountering generic things (expressions and functions) that are not immediately understandable or when you are interested in the details.,

Intro

The b3 expression language is a Domain Specific Language (DSL) for constructing executable build models. It is based on ideas from multiple sources - Xtend, Xtext, Scala, Java, Ruby and OCL.

The language is powerful, but is not considered to be a general purpose programming language. However, when constructing builds, we feel that it should be possible to do so using a modern function/object oriented type of language that gets the job done with as little noise as possible.

This chapter focuses on the concrete syntax and basic principles of the b3 language. Later chapters cover the build specific aspects of b3.

Although not required for reading and understanding the syntax, it is recommended to install b3, and open up the editor. You can immediately run the b3 code from within the editor and try things out.

General information

Files

The b3 language uses the file extension `.b3`, and the filename typically reflects the name of a build unit described in the file, but this is not an absolute requirement. A filename of `this.b3` has special meaning in b3's meta data translation where the information in the `this.b3` file is applied after discovered meta data has been translated into the b3 model - other than that, the `this.b3` file is just like any other b3 language file.

Structure

A typical b3 file describes a single *Build Unit*, which is either a control mechanism for orchestrating build-actions, or a representation of a more concrete buildable thing (e.g., things like a bundle, library, product or application).

In addition to import statements, and the definition of build units, a b3 file may also contain definitions of reusable property sets, general purpose functions and cross cutting concerns¹.



Note

We said that a *typical* b3 file describes a single build unit, but it may in fact contain none, one or several build units. A b3 file without a build unit may be used to only describe

¹An aspect oriented term denoting the possibility to override or amend the system. See [the section called "Concern"](#).

general purpose functions and concerns, and a b3 file with multiple build units is useful both at the top level where it is used to separate

Here is a sample b3 file with some general processing (keywords in bold):

```
function square(Double x) : x * x;

function hypotenuse(Double a, Double b) : Math.sqrt(square(a) + square(b));

function List<String> imageFileNames(List<String> fileNames) {
    fileNames.select( f | switch f
        case ~/*.*\.(jpg| : true
        case ~/*.*\.(gif| : true
        case ~/*.*\.(png| : true
        _ : false
    endswitch);
}

function String aTrigonometricFact() {
    var a = 3.0;
    var b = 4.0;
    "A triangle with the sides %f, and %f has the hypotenuse %f".format(a, b,
    hypotenuse(a,b));
}
```

And here is a sample with a build unit:

```
unit fruitsallad version 1.0.0 {
    builder make {
        input {
            org.myorg.apple / 2.3.0 #dice;
            org.myorg.banana / [1.0,2.0] #slice;
            org.myorg.orange #dice;
        }
        source {
            src/ [ vanilla.xxx, syrup.xxx ];
        }
        output {
            salladir/mysallad.zip;
        }
        myFruitProcessor(input, source, output);
    }
}
```

How all of this works is explained later (and with more realistic examples).

Comments and documentation

The b3 language supports both single line comments, and multi line comments (like in Java and many other languages).

```
// this is a single line comment
/* This is a multiline
comment */
```

Documentation is written in the style of Java Doc, where a multi line comment starting with more than one star after the leading / indicates that the comment is documentation. Documentation can only be used in certain places - one of them is for the unit itself.

```
/**
 * This is documentation for the build unit 'myunit'.
 */
unit myunit { }
```

**Note**

Any whitespace before leading '*' characters, and any trailing whitespace is removed from the documentation string before it is stored in the build model. For lines that do not have a leading '*' all leading whitespace is kept.

Types

The b3 language is a statically typed language, but uses type inference to reduce the amount of type declarations (compared to Java). This means that you can write the following statements without declaring the type of the variables as the b3 language detects the type from the value being assigned.

```
var a = "hello";  
var b = 10;  
var c = 1.0e+3;
```

In an untyped (or dynamically typed) system variables can refer to different types of values over time. This is not possible in b3 — once a variable is created, its type is defined. It may however refer to subclasses of its declared (or inferred) type.

To declare a variable and its type, the keyword `var` is replaced by the type name.

```
Number a = 42;  
Object b = "hello";
```

The typing system of Java is supported with full use of generics. This is important to be able to integrate well with function written in Java. When authoring typical b3 language build constructs, typing is typically inferred.

You can read more about the typing system in [the section called “Type system”](#).

Literals

The b3 language supports conventional literals for integers and real (i.e., floating point) numbers, and strings, but also for several other literal types. These are explained in the following sections.

Numbers

Integers

Decimal integers are written as you expect. These are all decimal integers:

```
0 1 2 3 4 5 6 7 8 10 100 123 4567839393
```

Octal integers are written with a leading 0 digit, and may be followed by the digits 0 to 7. These are all octal numbers:

```
01 023 07 0567
```

The expectation is that these are rarely used, but are valuable when writing logic where octal numbers are typically used (like in Unix file permissions).

Hexadecimal integers are written with a leading 0x (or 0X), and may be followed by a sequence of 0 to 9, a to f, or A to F (lower and upper case letters may be mixed). These are all hexadecimal numbers:

```
0x1 0xabc 0X1 0X0 0x7FFFFe
```

Internally, a b3 language literal integer is represented as a `java.lang.Integer` (if nothing else is stated).

Floating point numbers

Floating point numbers are supported in plain, or scientific notation. The integer part may be omitted. The following are all floating point numbers.

```
0.0 .1 3.14 .14 1.0e+2 .2e2 2e-2 1.0E+10
```

Internally floating point numbers are represented by `java.lang.Double` if nothing else is stated.

Strings

Strings are enclosed in `"` or `'`. A string enclosed in `"` may contain unescaped `'` characters, and vice versa. Special characters are escaped with a backslash `\` character. The b3 language literal string supports the same escapes as a `java.lang.String` (e.g., `\t`, `\n`, `\"`).

```
"This is a string"
'I am a string that says "hello"'
"I also say \"hello\""
```

Regular expressions

The b3 language supports a literal regular expression. A literal regular expression is written with a leading `~` (tilde) followed by the regular expression enclosed in `/`. Options may follow the ending slash character.

Regular expressions can also be constructed using the Java regular expression classes and methods. Using these gives more control over the use of the regular expression, such as making it possible to access the various parts of a match, but in many cases, a regular expression is used for simple boolean matching, and it is convenient to be able to declare literal regular expressions.

Here are some examples of literal regular expressions

```
~/[a-zA-Z_][a-zA-Z0-9_]*/
~/[a-z]*/i
~/^.*$/g
~/org\.myorg\..*/
```

A literal regular expression is represented by an instance of the `java.util.regex.Pattern` class. See the documentation of the `java Pattern` class for an explanation of the regular expression syntax, and additional functionality available.

Regular expression options. The options to a regular expression consist of single character flags with the following meaning:

Table 2.1. regexp options

Option Character	<code>java.util.regex.Pattern</code> Equivalence	Meaning
i	CASE_INSENSITIVE	upper and lower chars are considered equal
m	MULTILINE	makes <code>^</code> and <code>\$</code> also match line beginning and ending
u	UNICODE_CASE	makes case insensitive matching unicode aware
c	CANON_EQ	compares unicode using full canonical equivalence
d	DOTALL	makes <code>.</code> also match line terminators

Use of literal regexp. Literal regexp can be used in matches expression as in:

```
name ~= ~/[a-z]*/
```

or in switch expressions as in:

```
switch aString case ~/[a-z] : ...
```

and in many places where predicates are used for matching.



Note

Since literal regular expressions are compiled to an instance of `Pattern` by the b3 parser any regular expression syntax errors are caught and displayed in the b3 editor, and the literal `regexp` can be used as an argument in calls to `java` when more functionality than matching is needed.

Literal list and map

Literal lists are enclosed in `[]` and may contain any other object.

```
var a = [1, 2, 3, 9, 456];
var b = ["hello", "world"];
var c = ["hello", 42, "world"];
var d = ["hello", ["world 1", 20], ["world 2", 20]];
```

The type of the list elements is inferred, but it is possible to specify the element type by providing a type prefix enclosed in `<>`, as the first entry in the list. (For empty lists, the type must be stated, or if a more generic type than the inferred is wanted).

```
[<String> ]; // an empty list of strings
```

Literal maps (i.e. key/value pairs) are written with enclosing `[]` where the key is written with a `:` suffix. The key may be an identifier, a property identifier, or a string enclosed in `"`. Here are examples of some maps. (Again, the type of the resulting map is inferred).

```
var a = [name: "Fred Upstairs", occupation: "Dancer", shoesize: 9.5];
var b = [course: "Advanced b3" participants: ["mary", "john", "fred" ]];
// with more exotic keys
var c = [{target.platform: "win32", "har egen mjödbägare": true}];
```

The values are expressions, but the keys in maps are literal. The type of the key and value are inferred, but they may be specified by providing a type prefix inside the map.

```
[<String, Integer> key: ...]
```

Using expressions as keys. If you want to use maps with expressions as keys, you can use the `java` Map API (i.e., `put(Object key, Object value)`).

Evauation and mutability. A literal list or map is constructed each time it is evaluated. Thus, a literal map can be seen as a call to `new`, followed by a sequence of `add/put` operation. If you want to create it only once you can use the `cache` expression, but you then have to be aware of that the created list/map is mutable. To make the list/map immutable use `java`'s unmodifiable collection api.

Empty list or map

When an empty list or map is wanted the type information *must* be supplied as there is no way to infer the type of the list, nor is it possible to know if an empty list or an empty map should be created — they look exactly the same i.e., `[]`.

```
var anEmptyList = [<String>];
var anEmptyMap = [<String, Object>];
```

Literal functions

Literal function, also called *lambdas* can be thought of as ‘function as data’ which means that they can be assigned to variables (or constants), passed around as parameters, be stored in literal arrays

or maps etc. A literal function is also a *closure* which means that it has access to the definitions in the scope where it is declared. A literal function can also accept parameters. The parameter types and return type can also be declared. This is explained in detail in [the section called “Functions”](#), and in [the section called “Type system”](#).

A literal function is enclosed in { } with | (pipe) characters separating the parameters from the rest of the lambda. The leading pipe character may be omitted if the result is not ambiguous (which it can be when lambdas are written using the short form explained below). A single pipe character is required when there are no parameters, without it, the construct would simply be an expression block (sequence of expressions).

Here are some examples using literal functions:

```
var x = 10;
var a = { | x + 2 };
var b = a(); // b is set to 12

var add = {x, y| x + y}; // no leading pipe

var add = {|x, y| x + y}; // optional leading pipe (not required)
var d = add(5,4); // d is set to 9.
```

Literal Functions as parameters. Since it is quite common to pass literal functions as parameters, there is a shorthand notation where the enclosing { } may be omitted. Here is an example that shows both the long, and the shorthand notation:

```
// long/standard notation
aCollection.select({x | x instanceof Number});

// shorthand notation
aCollection.select(x | x instanceof Number);
```

More information about short hand notation is found in [the section called “Call expression”](#).

Return type. The return type of a lambda can be specified by a <> enclosed type ref after the opening bracket as in this example:

```
{<Integer> a, b | a + b };
```

Parameter types. Parameter types can be specified the same way as in a regular function, but this is not required when the type(s) can be inferred.

```
{Integer a, Integer b | a + b};
// with return type specified as well
{<Integer> Integer a, Integer b | a + b};
```

Other special literals

null

The special value `null` stands for unknown/no value (just like in Java).

unit

The special variable `unit` has a similar meaning to java’s `this` when used in expressions inside a builder function. The keyword `unit` is also used when declaring a build unit, to denote a requirement on something of build unit type, and when specifying advice/modifications to be applied to build units.

source, input, output

The special variables `source`, `input` and `output` refers to the result of evaluating the corresponding declarations inside builder function. The keywords `source` and `output` are also used to declare the base locations for source and output for build units.

The use of these are explained in [the section called “Input”](#), [the section called “Source”](#), and [the section called “Output”](#).

Wildcard ('_')

The special value `_` (*any*) is a wildcard comparable value — it compares to `true` with any other value. It is also used in some special cases to represent an unnamed variable in a mechanism referred to as *currying*.

Here is an example:

```
var aTenAsThird = [_, _, 10, _];
// does list have 4 elements and a 10 in the 3d position?
theTruth = [1,2,10,3] == aTenAsThird;
```

Identifiers

Identifiers (ID) must begin with a letter (a-z, A-Z) or underscore ‘_’, followed by an optional sequence of letters, numbers and the underscore character.

If an identifier happens to be a reserved word (such as `function`), the identifier can be escaped with a preceding `^` character² - i.e., `^function`. Using keywords as identifiers should naturally be avoided, but since identifiers are used to reference java code, or any EMF model, these have a different (or no) set of reserved words, and clashes may occur.

Importing

Imports are done in the outermost scope of a b3 file before any other expressions. Here is the syntax for imports:

```
reexport? import (QualifiedName | URIStrng) (as ID)? ;
```

`reexport`

This optional keyword makes the import available in a file that imports this b3 file as if it had been stated in that file. However, imported/reexported elements are overridden by explicit imports. *Not yet implemented.*

`import`

This keyword is required to indicate that an import is wanted.

«*QualifiedName*»

A qualified name is a series of ‘.’ separated IDs - like a Java package name. A qualified name is used to import a Java class. It is not possible to import everything in a package (i.e., ending the reference with ‘.*’ is not allowed).

«*URIStrng*»

A URI string is a URI written as a string (enclosed in ""). The evaluator will use all available knowledge to interpret what the URI is referring to. Typically this will be another b3 file, or an EMF model. *Not yet implemented.*

`as` «*ID*»

The optional `as` «*ID*» is used to assign the import to an immutable variable. This variable is available in all scopes in the b3 file. By default, a Java import will make the final part of the qualified name available, and for a URI it is the final part of the path with any suffix removed. If some other name is wanted, it should be stated with `as` «*ID*».

Here are some examples:

```
import java.lang.Math;
```

²This construct is used by most XText based DSLs.

```
import java.lang.Math as Calculator;
reexport import "platform:/resource:/myStandardStuff.b3" as common;
import "http://www.myorg.org/b3/globalSettings.b3" as myorgSettings;
```



Note

In the first prototype implementation of b3, only import of java classes is supported (alias is supported).

What can be imported

It is possible to import b3 files, and java classes. You can also import EMF models. In addition there is a mechanism for importing a properties file as a property set — see [the section called “Properties”](#).

Functions

Functions are general purpose, callable elements of the b3 language. They can be used in any expression.

Functions can be declared outside of the main body of a unit, or in a concern.

Function calls are resolved using polymorphism, the function with the most specific type matching the input parameters is selected.

You may think of functions as just being methods on objects, but the concept is far more powerful as you can define new function that appear “as if they were methods”. Lets say you want to add the method `camelCase` to a `String` — this is impossible in Java, but in the b3 language you can do this easily (as shown in [the section called “Function examples”](#)).

Defining a function

The definition of a function has the following syntax:

```
Function :
  DOCUMENTATION? Visibility? final? ReturnTypeInfo?
  FunctionName TypeParameterDeclarations?
  (( ParameterDeclarationList ))?
  (when ( : Expression ; ) | ( { ExpressionList? } ))?
  (( : Expression ; ) | ( { ExpressionList? } ))
  ;

ParameterDeclarationList :
  (ParameterDeclaration (, ParameterDeclaration)* (, ParameterDeclarationEllipse)?)
  | ParameterDeclarationEllipse
  ;
ParameterDeclaration : TypeRef? ID ;
ParameterDeclarationEllipse :
  ...
  TypeRef? ID ;
ReturnTypeInfo : // shown later - basically a reference to a Class
TypeParameterDeclarations : // shown later - this is for Java generics
```

«DOCUMENTATION»

This is Java doc style documentation for the function.

«Visibility»

The visibility is one of `private` or `public` with a default of `public`.

`final`

The keyword `final` makes it impossible to redeclare/override the function using the exact same parameter declaration.

«ReturnTypeRef»

This is a declaration of the return type and is required if the type can not be inferred. A more detailed explanation is found in [the section called “Type system”](#).

«FunctionName»

This is the name of the function/method. The name is an unqualified name (compare to name of a Java Method).

«TypeParameterDeclaration»

This optional part is for Java type generics. This is explained in [the section called “Type system”](#).
NOTE: Not yet supported.

«ParameterDeclarationList»

This is a list of parameter declarations, which is a list of `«TypeRef»? «ID»`, where the last (or only) entry can be a variable arguments parameter declaration `... «TypeRef»? «ID»`. Although the type declaration is optional, it can only be inferred in contexts where a literal/lambda function is being used in an expression that passes parameters to the function. If the type is omitted in a named function the type of `Object` is assumed. *NOTE: In the first implementation of b3, the type inference is limited, and parameter types must always be declared.*

when: «Expression»; | { «ExpressionList» }

The keyword `when` is used to specify a *guard expression*. Guards are very useful when writing system functions, you will probably not need to use them in your regular b3 logic.

The guard expression is used to determine the functions applicability to a given set of parameter types when performing polymorphic function selection. The guard expression(s) are evaluated with the types of the parameters in a call bound to the corresponding parameter names. As an example, a function with parameters `(Number a, Number b)` that is called with the declared types `Integer` and `Double`, will have `a` bound to the class `Integer`, and `b` bound to the class `Double` in the guard expression (as opposed to `a` and `b` being bound to the `integer` and `double instances` when the actual call is being made). Note that the guard function only has access to the types, the actual values are not known when the evaluation of the guard takes place.

A guard must return a boolean value, and if the guard returns false, the function will not be selected even if argument types otherwise match the declared function parameters.

: «Expression»; | { «ExpressionList» }

The function body is either `a` : followed by a semicolon terminated `«Expression»`, or a list of semicolon terminated expressions enclosed in `{ }`.

**Note**

The value of the function is always the last evaluated expression. There is no explicit ‘return’ expression in b3.

Function examples

Here are some simple examples:

```
function square(Number x) : x * x ;
function hypotenuse(Number a, Number b) : square(a)+square(b) ;
function timeRightNow : System.timeOfDay() ;
```

```
private final function toUpperAndSplit(String a) {
  a.toUpperCase();
  a.split('_');
}
```

Here is a more interesting example:

```
import java.lang.String;
```

```
/**
 * Turns a string with _ separators into a camel case string.
 * Input "non_camel_case" becomes "nonCamelCase"
 */
function camelCase(String s) :
    s.split( ~/_/.inject( "", _, | r, s | r + s[0].toUpperCase() + s.substring(1));
```



Tip

The method `inject` (used in the previous example) takes three arguments, a starting value to inject, and a wildcard that is a so called *curried* parameter that is replaced by each element in the collection on which the method is invoked. The third argument to `inject` is a literal function — it is invoked for each element in the collection, and `inject` returns the value of the last invocation as the value of the method. In the example, this means that on first invocation the literal function will receive the value `""` in the parameter `r`, and the first element from the `split` in the parameter `s`. It then turns the first character in `s` to upper case, and concatenates this with the rest of `s` to the result `r`. On the next invocation, `inject` uses the returned value as the value for parameter `r`, and `s` is set to the next value from `split`. This continues until the result has been collected and `inject` returns.

As explained in detail in [the section called “Call expression”](#), functions can be invoked using either *function call style*, or *feature call style*, but since you just learned about declaration of functions, this fact is important for understanding how functions are declared and used. The following two invocation are equivalent:

```
camelCase("i_am_not_in_camel_case"); // function style
"i_am_not_in_camel_case".camelCase(); // feature style
```

This works with multiple parameters too. Imagine an `unCamel` which takes an additional parameter that defines a separator string as an extra parameter. These invocation are then equivalent:

```
"iAmInCamelCase".unCamel('_'); // produces "i_am_in_camel_case"
unCamel("iAmInCamelCase", '_');
```

Expressions

The b3 language is based on expressions (as opposed to a mix of statements and expressions). As an example in Java there is an if-then-else *expression* (i.e., `expr ? expr : expr`), and an if-then-else *statement* (i.e., `if (expr) expr; else expr;`). In the b3 language you will find corresponding expression for if-then-else, switch-case, try-catch, etc. but they are all just expressions.

Loops are notably missing from the language itself. Loops are instead supported as functions/operations on collections and often make use of literal functions. This reduces the syntactic noise dramatically. If you really need to iterate a specified number of times, or need an iteration with an index variable, this is easily achieved by using a sequence expression which can be thought of as a virtual collection of numbers in a sequence.

The b3 language is based on functions, but writing expressions for `+`, `-`, `*`, `/` etc using functions is not human friendly, and the b3 language therefore supports operators (that translate into functions by the b3 engine). Since operators are supported, the b3 language also has a definition of operator precedence and left/right associativity. The precedence used is comparable to Java (but there are some new operators that are borrowed from other languages).

Expression and Expression Block

An *Expression* is something simple like `1+1`, `"hello"`, or something more complex like `var a = if 1>2 then 3 else 4 endif`. An expression can also be an *Expression Block* — a list of semicolon terminated expressions enclosed in curly brackets.

```
var a = "this is an expression";
```

```
var a = {
  var b = "this is an expression in an expression block";
  b.toUpperCase();
}
```

The first expression assigns a string literal to a variable `a`. The second example assigns the value of the last expression in the expression block to the variable `a` (i.e., the string `"this is an expr..."` in upper case).

Since an expression block is an expression it can be used wherever an expression can be used. A notable difference from java, is that it can be used as an argument in a function call. This reduces the need for local variables, and reduces the risk of leaving dangling/unused expression when editing/refactoring. Keep use within reason to increase readability of the code.

```
toUpperCase( {Logger.log("calling toUpper on 'abc123'"); "abc123";});
```

Operators - precedence

Here is a table with the operators, their Left (L), Right (R) associativity (in column A), and precedence (from highest to lowest).

Table 2.2. Operators

A	Operator	Operation Performed
-	if-, switch, try, (expression), literal, var / val reference, with, new, throw, literals	primary expressions - see following sections
L	.	feature access
L	[]	array index or keyed access
L	()	function/method call
R	++	increment pre/post (unary)
R	--	decrement pre/post (unary)
R	!	logical complement (unary) / not
L	..	sequence
L	*, /, %	multiplication, division, modulo
L	+, -	addition, subtraction
L	<, <=	less, less or equal
L	>, >=	greater, greater or equal
R	instanceof	type comparison
L	==	equal
L	~=	matches
L	!=	not equal
L	===	identical
L	!==	not identical
L	&&	logical and
L		logical or
R	var, val	variable, value (constant) definition
R	=	assignment
R	*=, /=, %=, +=, -=	assignment operations

Internally operators are implemented as system function with names corresponding to the operators. The same polymorphic binding mechanism used for regular function calls is naturally used for oper-

ators as well. The exact definition of what operators do, and what types they operate on is therefore determined by the system library.

The `'.` operator — feature access

There are two operators that both deal with keyed access; the *feature* operator `'.`, and the *at* operator `[]` (explained in the next section).

When using the feature operator, there is a difference between just accessing a feature as a value, and when calling a feature. This distinction is important when dealing with java objects as most java object have their features declared as private members and access is via getter and setter methods.

Accessing a feature value. When a feature value is requested the resolution depends on which type of object the feature operator is applied to — for b3 and model objects, the name of the feature is used as is. For java objects, the request is changed to a getter method call using java beans conventions (i.e., `x.foo` is modified to `x.getFoo()` (or `isFoo()` for boolean features), if not available with the name as given.

Setting a feature value. Setting feature values work the same way as access; for java objects feature assignment is modified to a setter method call. In the case of a setter, the name as given is never assumed to be a setter, it must follow the beans convention of `setXXX(someObject)`.

Calling a feature value. Calling features is done by polymorphic function resolution that takes all defined functions as well as the methods defined on the instance on which the feature call is being performed into account. (Defined functions may mask methods defined in a java class.)

Since feature calls are made with verbatim feature name, it is possible to invoke getter and setter methods on a java object should you prefer to use such calls directly, or if the java object does not follow the bean conventions. Here are example of some feature calls:

```
someObject.setSize(10);
someOtherObject.length(10);
```

The `[]` operator - indexed/keyed access

The *at* operator `[]` is used to access instances in collections. For list elements and strings, the index is numeric (integer), starting with 0. For Map elements, the index is any type of object (by default literal maps in b3 use strings as keys).

When `[]` is applied to a string, a one-character string is returned.

For lists, and strings, an `IndexOutOfBoundsException` exceptions is thrown on attempt to access items outside the available range. For map elements, `null` is returned for missing elements.

Call expression

Calling functions is done the conventional way; arguments are passed as a `()` enclosed comma separated list of expressions.

Calls have three different forms:

- **Named function call** — as in `toString(42)`
- **Feature call** — as in `42.toString()`
- **Expression call** — as in `(40 + 2).toString()`

The three examples above results in exactly the same call to the `toString` function.

**Note**

b3 makes no distinction between functions implemented in b3 and methods available in Java — Java methods are simply seen as functions with a first parameter of the particular type where the method was declared.

Calling static Java methods

Calling static Java methods is supported in b3. Here is an example:

```
var a = Math.sin(0);
var b = sin(Math, 0);
```

**Note**

b3 makes instances of classes available as instances of (b3) meta class types and the static methods are simply handled the same way as all other function calls. It is even possible to introduce new functions for classes.

Lambda parameter shorthand

There are many cases where a function takes one or more lambda functions as parameters. A syntactic shorthand is available for this case. The following two expressions are equivalent, and the second shows the shorthand notation:

```
func({x | x + x});
func(x | x + x);
```

This also works for multiple lambda parameters:

```
func(x | x + x, y | y - y);
```

In case there is an ambiguity where the parameters of the lambda starts, an optional bar ‘|’ can be used. Look at the following example, where the first declaration is an error because of ambiguity on parameter a, and the second where a bar is used to correct the ambiguity.

```
foo(a, b, c | b + c); // error, both foo and the lambda expected to have 2 params
foo(a, |b, c | b + c); // corrected
```

The ambiguity would not occur if the lambda was fully typed as in this example:

```
foo(a, <Integer> Integer b, Integer c | a + b);
```

Increment and Decrement

Increment ++ and Decrement -- is supported in both prefix and postfix notation with the same semantics as in Java. It is possible to use increment and decrement with any expression resulting in an assignable value (i.e variables, feature access, and indexed/keyed access). The type of the value must be numeric.

Not operator

The logical not operator ‘!’ negates a boolean expression.

New expression

The new expression is used to create new instances. Here is the syntax for new:

```
new TypeRef ( (ParameterList)? ) (as ID)? ContextBlock? ;
```


In its simplest form, the syntax is the same as for Java (the parentheses may be omitted if there are no arguments to pass to the constructor). Here are some examples:

```
var a = new Person("John", "Smith");
var a = new Person();
var a = new Person;
```

The parameter list is the same as for any function call, so shorthand notation for passing a lambda is supported. The optional `as ID`, makes it possible to refer to the not yet constructed instance inside the optional context block. The `as ID` has no other effect. The context block makes it possible to access the features of the newly created object as variables (without preceding them with the created object and a period). Here are some examples:

```
var johnSmith = new Person {
  firstName = "John";
  lastName = "Smith";
};
var cookieSmith = new Person(johnSmith) as child {
  firstName = "Eulalia";
  nickName = "Cookie";
  lastName = child.parent.lastName;
};
```

```
// equivalent to:
var cookieSmith = new Person(johnSmith);
cookieSmith.firstName = "Eulalia";
cookieSmith.nickName = "Cookie";
cookieSmith.lastName = cookieSmith.parent.lastName;
```

Sequence operator

The sequence operator `..` creates an iterable sequence from *lhs* to *rhs* as in 1..9 with a default increment of 1. The sequence operator works on integer or double numbers. The from value may be larger than the to value — this creates a sequence in descending order.

For double numbers, if the difference between to and from is less than 2.0 the default increment is 0.1.

The sequence can be further controlled using the sequence functions. They all return the sequence itself to allow convenient chaining. The functions `includeFrom(Boolean)` and `includeTo(Boolean)` are used to control if the stated from and to values respectively are included in the sequence or not (both are by default `true`). The function `step(Number)` is used to specify the step value if something other than the default is wanted. The step is an absolute value (i.e., is always positive even if the sequence is descending).

The sequence operator produces an iterable result which means that the system functions operating on sets (e.g., `collect`, `inject`) can be directly used, but it is also possible to get the iterator by calling the `iterator` function.

Here is a simple example adding the numbers 1 to 9, setting the variable `a` to 45.

```
var a = inject(1..9, 0, | a, b | a + b );
// or alternatively
(1..9).inject(0, | a, b | a + b);
```

In the following example, the step is set to 2 to sum the odd values. This also illustrates the different ways functions can be invoked.

```
var a = inject(step(1..9, 2), 0, | a, b | a + b );
// or
var a = inject((1..9).step(2), 0, | a, b | a + b );
// or
var a = (1..9).step(2).inject(0, | a, b | a + b );
// or
```

```
var a = step(1..9, 2).inject(0, | a, b | a + b );
```



Note

The use of parenthesis around (1..9) when using the form (1..9).inject is required since the feature call expression .inject has higher precedence.

The left and right hand sides of a sequence expression does not have to be literal values — complex expressions can be used, but since the precedence of the sequence operator is quite high, the expressions have to be enclosed in parentheses. (This is by design, because the typical use of sequence operator is for literal sequences as all loops over collections have specially designed functions that does not require iteration using index variables). Here is an example using expression:

```
var a = 1;
var b = ((a + 2)..(1+2+3)).inject(0, |a,b|a+b);
```

This examples will set the variable b to 18.

*, % and /

The multiplication *, modulo %, and division / operators operates on numerical values. They can be used with all numerical types in Java.

+ and -

The addition +, and subtraction - operators operate on numerical values. They can be used with all numerical types in Java.

The + operator can also be used to concatenate strings (all CharSequence instances can be used, which includes String, and StringBuffer).

Relational operators

The relational operators <, >, <=, >= compares two objects (implementing Comparable, or Number) and returns a boolean result.

The relational operators == and != tests the equality of two objects, and the === operator tests if two objects are identical (the same instance), and finally !== tests if two objects are not identical (the same as !(a === b)).

Matches operator

The matches operation ~= matches the *lhs* against a pattern on the *rhs*. The pattern can be a literal regular expression, or a simple pattern string. The simple string pattern can use * to denote 0 or more characters, and ? to denote any single char.

The ~= operator also matches lists, maps, numbers and objects. For numbers and objects, the result is the same as when using the == operator. When matching lists the comparison of list elements is performed using ~=. When matching maps, the values (i.e., not keys) are matched using ~=.

Here are some examples that all evaluate to true:

```
"hello" ~= "?ello";
"hello" ~= "h*o";
"hello" ~= ~/.ello/;
"hello" ~= ~/h.*o/;
["hello", "goodbye"] ~= ["h*o", "*oo*"];
[greeting: "hello", farewell: "goodbye"] ~= [greeting: "h*o", farewell: "*oo*"];
```

instanceof operator

The `instanceof` operator is used as in Java to test if an object is an instance of a particular type (class or interface).

Logical connectives `&&` and `||`

The logical connectives for *and* `&&`, and *or* `||` works the same way as in Java. Expressions are evaluated from left to right until the truth or falsehood of the overall expression is known.

Variables and Constants

Variables are defined by using the keyword `var` or by stating the type of the variable. Constants are defined by using the keyword `val`. Variables and constants may be marked as `final` to prevent redefinition in inner scopes. Definition of a constant or definition of a variable with inferred type always requires that a value is assigned. Uninitialized variables have a default value of `null`.

Here are some examples:

```
var a = 10; // type is inferred
Integer a = 10; // type is declared
Integer b; // initialized to null
final var a = 10;
final Integer a = 10;

val c = 10; // type inferred
val Integer c = 20;
final val c = 10;
final val Integer c = 10;
```

Assignment operations

Assignment operation works the same way as in Java. The assignment operator `=` assigns a value to something assignable (also known as a L-value for “left hand side value”). Assignable items are variables, constants (for initialization), features, indexed, and keyed access.

The assignment operators `+=`, `-=`, `*=`, `/=`, and `%=` also works the same as in Java - they are shorthand notation for `a = a «op» b`.

If expression

If-then-else expressions are written on the form:

```
if «Expression» then «Expression»
  (elseif «Expression» then «Expression»)*
  (else «Expression»)?
endif
```

Note that there is no need to terminate the expressions with a semicolon if you want to use the result as in:

```
var a = if french == true
  then "'fin'"
  else "'the end'"
  endif + " is shown at the end of a movie";
```

Switch expression

The switch-case expression allows switching by comparing an expression against multiple expression cases, or simply evaluating expression cases in turn until one returns true. In contrast to a C or Java switch statement, the b3 language switch does *not* fall through from one case to the next, and as a consequence, the b3 language does not have ‘break’ or ‘continue’ expressions.

A switch expression is written on the form:

```
switch «Expression»?
  (case «Expression» : «Expression»)*
endswitch
```

Notice the lack of a default case. If you want a default case, simply enter a case `_` : last in the list (the `_` expression is a wildcard that compares true against anything).



Note

‘`switch true case «e» :`’ is equivalent to ‘`switch case «e» :`’

If either the result of the switch expression, or the result of the case condition implements the java interface `Comparable` the comparison is performed using `compareTo`, otherwise the comparison is performed using `equals`. A literal regular expression implements this interface, and can thus be used directly as case expressions.

If none of the cases match, the switch expression evaluates to `null`.

Remember that switch-case is an expression and it is possible to write expression like this:

```
var a = "The condition is " + switch condition
  case 1: "critical" case 2: "severe" case _: "ok"
endswitch;
```

Try expression

The b3 language has support for exceptions and try-catch-finally expression — they are written on the form:

```
try «Expression»
  (catch «TypeRef» «ID» : «Expression»)*
  (finally «Expression»)?
entry
```

Here is an example:

```
var a = "First line in file " + try "is: " + file.readLine()
  catch IOException e :
    "could not be read due to ERROR: " + e.getMessage()
endtry;
```

This example will set `a` to a message string either containing a line read from a file, or an error message.



Note

The b3 language itself does not have checked exceptions. All exceptions are treated as unchecked.

The value of the try expression is the value of the try block if no exception was caught, and the value of the triggered catch block if an exception is caught. The value of the finally block is never used.



Implementation Note

The current implementation performs type inference by taking the common supertype of the try expression, and all catch expressions. The rationale behind this is that you either do not care because the value of the try/catch is never used (as in java), or you *do* care, in which case you will be returning something that makes sense in all catch expressions (or the try/catch would never have been used in this fashion in the first place). The alternative would be to have catch expression return `null`, and let type be inferred only from the try expression.

Throw expression

The `throw` expression works like Java's `throw` statement. Here is an example:

```
throw new SomeExceptionClass("Ouch, that hurt!");
```

It is also possible to throw any object, in which case b3 will wrap it in a `B3UserException`. This exception has an `Object getData()` method that returns the object being thrown. The exception produces a message on the format "User data exception: «data.toString()»". Here is an example:

```
throw "Ouch, that hurts";
```



Implementation Note

Exceptions are very much part of Java and required for interfacing with Java. Unfortunately it also makes the b3 language dependant on Java for Exception classes, the distinction between checked and unchecked exceptions etc. There is no defined mechanism (at least not yet) in the b3 language to create new Exception classes (in fact there is currently no way to create any new types or classes).

Cache expression

The `cache` expression evaluates an expression, and if it is the first time the expression is evaluated in the current invocation (i.e., a run of a b3 engine) the expression is cached. Subsequent evaluation of the same expression returns the result from the cache instead of re-evaluating the expression.



Note

The value in the cache is mutable (i.e., if it is a mutable object such as a `List` or `Map`). If an immutable instance is wanted, Java utility methods should be used to create an unmodifiable version of the cached value.

The `cache` expression is useful if your logic contains large static data structures that are required many times.

Here are some examples:

```
cache 1 + 1;
var a = cache [someString + someOther, y.getMessage(), z.expensive() ];
```



Warning

The data in the cache is only cleared when the b3 engine has finished executing. The cache may get a cache eviction policy in the future so you should not rely on the fact that calls inside a cached expression only gets executed once.

Typecast

In b3 a typecast is performed by calling an instance of a meta class as in this example:

```
if(x instanceof SomeClass)
    (SomeClass)(x).aSomeMethod();
```

Note the subtle difference from Java (where the casted expression is not enclosed in parantheses).

With context expression

The `with context` expression, is used to evaluate a block expression in the context of another object. Applying a context means that there is no need to restate the expression that constitutes the context. Compare the two equivalent examples:

```
// example 1 - not using with context
document.chapters[3].title = "Title of chapter";
document.chapters[3].authors = ["mary", "john"];

// example 2 - using with context
with context document.chapters[3] {
  title = "Title of chapter";
  authors = ["mary", "john"];
}
```

The value of a `with context` expression is the context instance (i.e., `document.chapters[3]` in the example above).

The `with context` expression is very similar to the `new` expression (except that no new object is created). Just as in the case with the `new` expression, it is possible to give the context instance object an alias (i.e., a temporary name if there is the need to refer to it in the context block).

The `with context` expression is useful to avoid having to use regular variables to act as the context.

Bitwise operations

Notably missing from the set of operators in b3 are bitwise operators as found in C++ or Java (as bitwise operations are probably not that commonly used in build scripts). Should you ever have to use bitwise operations it would be very complicated if there were no support at all, and the b3 language therefore supports bitwise operations via system functions, as shown in the table below. Bitwise operations are available for integral values (i.e., non floating point).

Table 2.3. bitwise funtion

Function	java equivalence
<code>bitwiseShiftLeft(i, shift)</code>	<code>i << shift</code>
<code>bitwiseShiftRight(i, shift)</code>	<code>i >> shift</code>
<code>bitwiseUnsignedRightShift(i, shift)</code>	<code>i >>> shift</code> (i.e., right shift with zero extension)
<code>bitwiseAnd(i, i)</code>	<code>i & i</code>
<code>bitwiseXor(i, i)</code>	<code>i ^ i</code>
<code>bitwiseOr(i, i)</code>	<code>i i</code>
<code>bitwiseComplement(i)</code>	<code>~i</code>

Properties

The b3 language has extensive support for property management. The rationale behind this is that many build related utilities rely on the use of properties to control their behavior and it quickly gets very complex to manage properties that are loosely defined in various property files. This is solved in b3 by defining property sets that are either specified in b3 directly or loaded from properties files. The b3 property sets can extend other property sets, and can be filtered. All in a structured way.



Tip

Loading properties from files is valuable when sharing property definitions between b3 and other tools.

Property sets

Properties are (semi) globally scoped variables that may be hidden/redefined in inner scopes. When returning from a scope that defined properties, the property values return to their previous values.

Properties are declared in property sets, and these come in two flavors — *regular* property sets, and *default* property sets. The difference is that when a default set is evaluated, only those properties that are not already set will be defined. A regular property set will define and set all values defined in the property set irrespective of if they already have a value or not).

Property sets can be named and referenced from other property sets. The named property sets can be referenced both by default, and regular property sets. The evaluation depends on where and how they are referenced (i.e., a regular property set referenced from a default property set will be evaluated with default property set semantics).

A property set may extend another property set, and it may extend a property set imported from a properties file.

A default property set is declared like this:

```
default properties «PropertyBody»
```

And named property sets are declared like this:

```
properties «ID» «PropertyBody»
```

A *PropertyBody* has the following syntax

```
PropertyBody : ((extends ID)? { PropertyOperation* }) | URI
PropertyOperation
  : FilteredProperty
  | PropertyDefinition
  | PropertyBody
  ;

FilteredProperty
  : when (BooleanExpression) (PropertyDefinition | PropertyBody)
  ;

PropertyDefinition
  : final? mutable? TypeRef PropertyName = Expression ;
  ;
```

Property names are qualified names prefixed with \$. Properties loaded from files automatically gets the \$ prefix in b3 when loaded (i.e., properties in files should not be stated with a leading \$). Here is an example of a named property set:

```
properties PlatformAgnosticProperties {
  $target.platform = "*";
  $target.os = "*";
  $target.ws = "*";
}
```

Naming property sets. The name of the property set must be unique among all named concerns in the b3 file (i.e., other named property sets, and named concerns). The name of a set is an ID.

Extending a property body. The expression following `extends` must evaluate to a property set. Here is an example:

```
properties myProperties {
  $this.is.a.standard.property = 10;
}

unit {
  default properties extends myProperties {
    $this.is.an.extra.property = true;
  }
}
```

Filtering properties. A property definition may be filtered by preceding the property definition with `when(«Expression»)`, where the *Expression* is a boolean expression.

The expression may reference any property already defined in the same scope, or in a property set it extends (directly or indirectly). Here is an example:

```
when ($target.platform == "linux") $compiler.xyz.optimize = true;
when ($target.platform == "linux") {
    $extra.property1.for.linux = true;
    $extra.property2.for.linux = true;
};

when ($target.platform == "linux && $build.type == "server")
    extends standard.linux {
    $extra.server.property.for.linux = true;
};
```

Properties are defined once. Note that it is not possible to define the same property more than once in a property set. Properties are by default created as constant values, and their value can not be modified once the property is created. It is however possible to make a property behave as a variable by using the keyword `mutable`, but it still only possible to declare each property once.

The property operations are evaluated in the order they are stated. This makes it possible to involve already defined properties in subsequent statements. Here is an example:

```
properties mySet extends somePropertySet {
    $compilerFlags = "-a -b -c";
    when ($target.platform == "linux")
        $compilerFlagsOptimized = $compilerFlags + " -O4"
}
```

Property value expressions may use the full set of expressions available in the b3 language, and properties may refer to any type of object. Care must be taken though for properties that are later used as system properties (input parameters) to external actions, as passing of properties is typically done using string values and property values will be converted using `toString()` operation on non string values before they are used in such contexts.

Final properties. Properties can be declared as `final`. This means that it is not possible to redefine the property in an inner property scope. An attempt to redefine a final property will throw a runtime exception.

Mutable properties. Properties are by default immutable, and its constant value is assigned when the property is defined. Other property scopes can however define a property with the same name and a different value. A property declared as `mutable` is different — it makes the property behave more like a global variable as it can be modified from any inner scope.

Loading properties from files

A property set can also be defined by loading it from a standard java properties file. A property set that is loaded from a file can not at the same time extend other property sets. The file to load is specified with a URI. Here is an example:

```
properties ServerBuildProperties "http://somewhere.com/server.properties" ;

properties LocalBuildProperties "workspace:/my.build.stuff/localBuild.properties";
```

Other property sets can extends a set from a properties file. And since a property set can include other property sets, and these in turn can extend other, or be loaded from properties files, there is really no limit to how the property sets can be composed.

Property sets are concerns

You will learn more about concerns in [the section called “Concern”](#), but it is good to know that a property set is a simple kind of *concern* — something named that can be applied “after the fact” as in “apply this property set when calling this function”. Here is an example:

```
with LocalBuildProperties : doSomeStuff();
```


Accessing properties

Properties can be accessed in expressions the same way variables are accessed. If a property is mutable it may be modified.

```
var a = "You are running on " + $target.os;
```



Note

Since properties have qualified names, care must be taken when using a feature or feature call expression in combination with a property. As an example the expression `$target.os.toUpperCase()` will try to find a property value for the property `$target.os.toUpperCase`, and then call this value as a lambda. This will most likely fail as there is no such property. The correct way is to either separate the feature reference from the property with a space `$target.os .toUpperCase()`;, enclose the property reference in parentheses `($target.os).toUpperCase()`;, or use a named function call `toUpperCase($target.os)`;

Concern

A *concern* is an aspect oriented programming concept which groups a set of *advice*. An advice is something that is dynamically woven into the fabric of the logic, and can be as simple as modifying a value, or complex overrides of functions found using query patterns with calls to the overridden function after having modified arguments used in original call.

A concern can be defined for the purpose of applying it via reference (via its name), or for the purpose of extending it. It is also possible to both define and apply a concern at the same time. using anonymous advice. How these anonymous concerns are defined and applied is explained in the section [the section called "With expression"](#).

The referenceable form of a concern must have a name, and can optionally extend other concerns. The name of a concerns must be unique among all concerns in the same b3 file (i.e., other concerns, and named property sets).

The syntax for concern is as follows:

```
Concern :
  concern ID (extends ID (, ID)* )? {
    ( (properties ...)
      | (default properties ...)
      | (function ...)
      | (builder ...)
      | Context
    )*
  }
;

Context : context ContextSelector ContextBlock ;

ContextSelector : function | unit | builder ;
```

`concern` `«ID»`

Marks that a concern is being declared and gives it a name. The name is an unqualified name.

`extends` `«ID» (, «ID»)*`

Makes the concern extend other concerns. (When concerns are applied they are applied in the order they appear in the extends list, and all extended concerns are applied before the extending concern).

`properties`, `default properties`

These keywords are followed by property set definitions (with the same syntax as when they are not inside of a concern). Property sets are evaluated in the order they are specified.

`function ... , builder ...`

These keywords are followed by a function or builder definition. (The same syntax as when they are not inside a concern is used — with one exception for builder definitions which is explained in [the section called “Adding or overriding builders”](#)).



Note

A function or builder introduced this way (as a direct child of `concern`), will simply override any existing function or builder with the same parameter signature. This is thus primarily intended as a mechanism for introducing new function or builders. There is no relationship between the introduced function and the overridden and it is not possible to use the `proceed` expression to call the overridden function. A `context` should be used if weaving is wanted.

`context «ContextSelector» «ContextBlock»`

This declares that the following advice is for a specific context. The *ContextSelector* consists of a context type (`function`, `unit` or `builder`), and predicate(s) to specify which instance(s) of the selected type to apply the following context block on. The *ContextBlock* contains the dynamic advice to weave. The context blocks are different for the various types of contexts (`function`, `unit` or `builder`). The function context block is described in the following section, and the contexts for unit and builder are explained in [the section called “Unit & Builder Concern”](#).

Function concern context

A function concern context is used to specify advice / weaving of functionality for already declared functions (as well as future functions dynamically introduced in a scope while the advice is in affect). This is done by specifying a query with predicates for function name and parameter types. When the concern is in effect, all preexisting matching functions as well as all matching functions introduced in the future will be advised.

Since b3 treats methods for Java classes as functions, it is possible to advise these as well. Such advice is however only applied when calling java methods from b3³.



Warning

There is currently no support for matching on return/produced value type. Care must be taken to return a value compatible with the advised function's return type or a runtime exception is thrown. (This is not as difficult as it may sound, since there is a `proceed` expression that calls the original function — see “`proceed` expression” below for more information).

Here is the syntax for the function concern context:

```
function «NamePredicate» ( ( «ParameterPredicateList» ) )?
  «BlockExpression»
```

«NamePredicate»

A name predicate is one of:

- **literal name** — an ID or String which specifies an exact match.
- **regular expression** — a literal regular expression (such as `~/to(Upper|Lower)Case/`)
- **wildcard** — a literal wildcard (i.e., `_`) used to match any name (typically used when there is also parameter predicates).

«ParameterPredicateList»

A comma separated list of parameter predicates as explained below.

³If you need to advice calls from java to java, you need to use other tools such as AspectJ).

«*BlockExpression*»

A b3 list of expressions enclosed in { }. This list may include the `proceed` expression as explained in [the section called “Proceed expression”](#).

Parameter Predicate List. The parameter predicate list consists of a comma separated list of parameter predicates. It is possible to match single or multiple parameters with one predicate. When matching a single parameter, it is also possible to give this parameter a name. A named predicate provides access to the corresponding argument value when the advised function is called. Modification of the arguments have effect on the values seen in the advised function when using `proceed`.

- `_(? | * | +)` — any type, 01, 0M, or 1M times. Also matches varargs. As an example:

```
context function foo(_*)
```

matches all functions having the name “foo”.

- «*typeref*» (? | * | +) — given type, 01, 0M, or 1M times. As an example:

```
foo(Integer+)
```

matches all function having the name `foo`, and one or more `Integer` parameters.

- «*typeref*» «*name*» — given type (occurs once). The argument is available in the block expression by referencing the specified name (the name does not have to match the name in the original function). As an example:

```
foo(Integer p)
```

matches a function called `foo` that has a single `Integer` parameter, the parameter’s value is made available in the variable ‘`p`’.

- `... _` — any vararg (may only appear last in the parameter predicate list). Here is an example:

```
_(_* , ..._)
```

matches any function that is declared with varargs of some type.

- `... «typeRef»` — vararg of given type (may only appear last in the predicate list). Here is an example:

```
_(_* , ...Integer)
```

matches any function that is declared with varargs of `Integer` type.

- `... «typeRef» «name»` — vararg of given type. The argument is available in the block expression. As an example:

```
_(_* , ... Integer x)
```

matches any function that is declared with varargs of `Integer` type, and the varargs list is available via the variable ‘`x`’



Note

The multiplicity rules are not greedy, thus `Integer *`, `Integer a` will give access to the last integer in a series of integers (including a single integer argument).

Proceed expression

It is possible to call the advised function by using a `proceed` expression⁴. The `proceed` expression behaves like a call, but does not take any arguments. Instead, the original arguments are passed (possibly

⁴Requiesive advice and `proceed` can be used.

modified by the advice). The value of the `proceed` expression is the value produced by the advised function, and it is thus possible to define processing before, after, or around the advised function.



Note

If a `proceed` expression is not used, the advised function's body is never evaluated.

Here is an example of a *before* advice:

```
context function _(Person p, _) {
  System.out.println("A person was called");
  proceed;
}
```

Here is an example of a before advice that alters an argument:

```
context function _(Person p, _) {
  p = p.copy();
  p.name = if p.name == null then "unknown" else p.name endif;
  proceed;
}
```

In this example, a copy is made of the argument `Person p`, and the copy is then modified (if the name is null, it is set to "unknown"). The original person object is left unmodified.

In both of the examples above, the `proceed` expression is the last expression, and thus, the value produced by the original function becomes the value produced by the advised function.

Here is an example using an *after* advice:

```
context function _(Person p, _) {
  var result = proceed;
  if p.name =~ ~/John.*/
    then System.out.println("A person named John was processed")
  endif;
  result;
}
```

Processing performed both before and after the `proceed` is called *around*-advice, but there is really no difference between *before*, *after* or *around* advice — it all depends on where and how the `proceed` expression is used and what is being returned.

With expression

The `with` expression is used to apply concerns that have effect in the inner scopes of the `with` expression. The `with` expression may apply already defined concerns, extend such concerns, and declare new (anonymous) concerns. Here is a simple example:

```
var a = with aConcern : aCallToSomething();
```

Here is a more elaborate example:

```
with aConcern
  properties {$a = 10;}
  concern {
    function foo(Number x) : x-3 ;
  } {
    var x = foo($a);
    var y = foo(x);
  };
```

In this example `with` is used to apply 'aConcern' that is defined somewhere else (just shown for syntax illustration). It also defines a property set, and an anonymous concern with a function `foo` that returns its parameter - 3. As a result, the variable `y` will be set to 4, and this is also the value of the entire `with` expression.

Type system

The type system of the b3 language is based on Java. Most of the time, the b3 type inference system is capable of inferring the type, but there are some cases where the type must be defined.

- Parameters to general functions — there is simply no way to know all possible locations from which a function may be called.
- When the inferred type becomes Object (other other generic type) but the returned values share some other trait (interface) that is more suitable as the return type.

The type system is based on Java, but has an additional type construct borrowed from Scala for declaring parameter and return types. The type system syntax is as follows:

```
TypeRef : SimpleTypeRef | ClosureTypeRef ;
SimpleTypeRef : ID (:: ID)* (< TypeParam (, TypeParam)* >)? ;
ClosureTypeRef : ( ( TypeRef (, TypeRef)* ))? => TypeRef ;
TypeParamDeclaration : ID ((extends TypeRef (& TypeRef)* ) | (super TypeRef))?
TypeParam : TypeRefParam | WildcardRefParam ;
TypeRefParam : TypeRef ;
WildcardRefParam : ? ((extends TypeRef (& TypeRef)* ) | (super TypeRef))?
```

This looks very complicated because Java generics are complicated. You don't have to use generics, but when your code interfaces with Java, you may have to. Using generics has many benefits as more inferences can be made.

Here are some examples using the above syntax in function definitions:

```
/**
 * Complicated generics example (borrowed from an xtend wiki page)
 */
function List<M> sort<T extends Comparable<T>, M> (List<M> toSort, (M)=>T closure){
}

/**
 * Example from java tutorial. Note that the constraints on T are after the function
 * name as proposed in the future xtext wiki doc.
 */
function T foo <T extends Annotation> (Class<T> annotationType) {
}

/**
 * Same example, but using closure type style
 */
function (Class<T>) => T foo <T extends SomeClass> (aClass) {
}
```

DISCUSS: - One wiki article about a possible future xtext language showed type constraints written differently than in Java. In Java the type specification of `<T extends Annotation> T foo(Class<T> annotationType)` instead of as in the article where placing the `<T extends Annotation>` after the function name. Wonder about the rationale for this?

The current implementation of b3 does not have full support for java generics. Type variables are notably missing. Eclipse b3 work on type system started before the Java integration was made in XText 0.8. It will possibly change to benefit from the new features in XText 0.8.

System functions

The b3 language has a default set of system functions. Among the things supported are functions that handle loops and set operations. This section documents these functions.

**Note**

The wildcard card character `_` is used to denote that the type is inferred. This notation was chosen rather than using full java generics declarations as these declarations are quite complex to read.

Evaluation

```
_ evaluate(function, ...arguments);

(_*)=>_ function;
Object ...arguments;
```

Evaluates the `function` lambda passing the variable number of arguments. The number and type of arguments must be compatible with the signature of the function. The result of evaluating the function is returned. This is essentially the same as using the call expression, but may be clearer to use in some contexts.

Looping functions

```
_ whileTrue(condition, body);

()=>Boolean condition;
()=>_ body;
```

Evaluates the `condition` lambda, and if it evaluates to `true`, the `body` lambda is evaluated. The process is repeated until the `condition` lambda no longer returns `true`. The last result of evaluating the `body` is returned. (This is the while loop in the b3 language).

```
_ whileFalse(condition, body);

()=>Boolean condition;
()=>_ body;
```

Evaluates the `condition` lambda, and if it evaluates to `false`, the `body` lambda is evaluated. The process is repeated until the `condition` lambda no longer returns `false`. The last result of evaluating the `body` is returned.

```
_ whileTrue(body);

()=>_ body;
```

Evaluates the `body` lambda until it does not return `true`. (This is the do-while loop in the b3 language). The result of the function is the non `true` value that terminated the repetition.

```
_ whileFalse(body);

()=>_ body;
```

Evaluates the `body` lambda until it does not return `false`. The result of the function is the non `false` value that terminated the repetition.

Set functions

```
_ do(collection, body);

Iterable<_> collection;
(_)=>_ body;
```

Iterates over the `collection` and evaluates the `body` lambda with each value from the collection as a parameter. The last result of evaluating the `body` is returned.

```
List<_> select(collection, body);
```

```
Iterable<_> collection;
(_)=>Boolean body;
```

Iterates over the `collection` and evaluates the `body` lambda with each value from the collection as an argument. If the `body` lambda returns `true`, the current value is added to a resulting list. The resulting list is returned.

```
List<_> reject(collection, body);
```

```
Iterable<_> collection;
(_)=>Boolean body;
```

Iterates over the `collection` and evaluates the `body` lambda with each value from the collection as an argument. If the `body` lambda returns `false`, the current value is not added to a resulting list (all other values are). The resulting list is returned.

```
Boolean exists(collection, body);
```

```
Iterable<_> collection;
(_)=>_ body;
```

Iterates over the `collection` and evaluates the `body` lambda with each value from the collection as a parameter. If the `body` returns `true`, the iteration stops and `true` is returned. If the iteration reaches the end of the collection without the `body` having returned `true`, `false` is returned.

```
Boolean all(collection, body);
```

```
Iterable<_> collection;
(_)=>_ body;
```

Iterates over the `collection` and evaluates the `body` lambda with each value from the collection as a parameter. If the `body` returns `false`, the iteration stops and `false` is returned. If the iteration reaches the end of the collection without the `body` having returned `false`, `true` is returned.

```
List<_> collect(collection, body);
```

```
Iterable<_> collection;
(_)=>_ body;
```

Iterates over the `collection` and evaluates the `body` lambda with each value from the collection as an argument. Each returned value from the `body` is added to the resulting list. The resulting list is returned.

```
_ inject(collection, startValue, body);
```

```
Iterable<_> collection;
_ startValue;
(,)=>_ body;
```

Iterates over the `collection` and evaluates the `body` lambda for each value from the collection with the previous value of the body evaluation and the current value from the iteration as arguments. The parameter `startValue` is used for the first iteration. The last result of evaluating the body is returned.

The typical use is as in the following example which returns the value 45 (sum of 1 to 9):

```
(0..8).inject(1, {sum, x | sum + x; }
```

Currying

Currying means that a call is curried/spiced with parameters. The set operation that takes lambdas as arguments use lambdas that take only on argument (or two in the case of `inject`), but what if you want to call a function that takes more than one argument? In these cases, you can use currying. To do this,

simply state the parameters that should be passed to the lambda and use a wildcard for the parameter(s) that should receive the value from the current value from the iteration. Here is an example:

```
function (Number, Number, Number) => Boolean
  between(x, min, max) : x >= min && x <= max ;

aCollection.select(_, 1.0, 2.0, between);
```



Note

Currying an `inject` is special as there are two anonymous values being passed. `Inject` can be called with `_`, `val`, `func` (which is also the default if currying is not used), or `val`, `_`, `func`, or `val`, `func`, `_`. The practical value of this is only if calling a function that was not designed to be used with `inject`.

Remember that the parameter list is only evaluated once when the set function is invoked.

Assert function

The system functions include a number of assert functions. The primary use of these are for writing tests, but they are also very useful when advising code when trying to pinpoint issues in a complex build. All assertion functions either return `true`, or throw an assertion exception. The given message is used as the message text for the exception along with information about the asserted condition's reason for failing.

```
assertEquals (message, expected, actual);
```

```
String message;
_ expected;
_ actual;
```

Asserts that `actual` is equal to `expected`.

```
assertTrue (message, actual);
```

```
String message;
_ actual;
```

Asserts that `actual` is true.

```
assertFalse (message, actual);
```

```
String message;
_ actual;
```

Asserts that `actual` is false.

```
assertNull (message, actual);
```

```
String message;
_ actual;
```

Asserts that `actual` is null.

```
assertNotNull (message, actual);
```

```
String message;
_ actual;
```

Asserts that `actual` is not null.

```
assertType (message, expected, actual);
```

```
String message;
```



```
Type expected;  
_ actual;
```

Asserts that `actual.getClass()` is equal to `expected` (i.e., exact class). This function is mainly useful when testing b3 itself.

```
assertAssignable (message, expected, actual);
```

```
String message;  
Type expected;  
_ actual;
```

Asserts that `actual` can be assigned to a variable of type `expected`. This function is mainly useful when testing b3 itself.



3

The Build Unit

In this chapter we take a closer look at the b3 Build Unit. As mentioned earlier in the introduction ([the section called “The Build Unit”](#)) a Build Unit is an abstraction of the buildable aspect of a part in a software system (i.e., components, modules, plugins, platforms, repositories, bundles, etc.).

Build Unit

A build unit is declared with a unit preamble, followed by the unit’s body enclosed in curly brackets. Here is an example:

```
/**
 * 'myunit' is a build unit, this is its documentation.
 */
unit myunit version 1.0.0 {
    // body of unit
}
```

The preamble has the following syntax:

```
«DOCUMENTATION»?
«ExecutionMode»?
unit «UnitName»? (version «Version»)?
(implements «InterfaceName» (, «InterfaceName»)*)?
```

«DOCUMENTATION»

The build unit can have optional Java Doc style documentation.

«ExecutionMode»

The execution mode is either `parallel` or `sequential`. The default is `parallel`. The execution mode defined for a unit defines the execution mode for builders operating on the unit. In addition, global b3 preferences can specify the execution mode for all units / builders. The mode used is `parallel`, unless the builder is specified as `sequential`, the unit is specified as `sequential`, or the global execution mode preference is set to `sequential`.

Sequential execution means that all dependencies are built in sequential order, when running in parallel all dependant builders may run in parallel, but concurrency is controlled via synchronization rules (locking of resources). This is explained further in [the section called “Synchronization”](#)

The execution mode is not the same as being thread safe — all b3’s actions are thread safe in respect to how synchronized in Java behaves.

unit

This keyword indicates that this is a build unit declaration.

«UnitName»

Although most build units should declare a name, the name of a unit is in fact optional in the b3 model, since a unit can be used to define additions or overrides. When a unit is declared in a `this.b3` file with the purpose of decorating/amending the meta data translation of the real unit it is embedded in, the name is typically omitted as the resulting unit most likely should keep its

original name. If a unit name *is* stated in a `this.b3`, it overrides the original name generated by default. All other units should have a name.

The unit name is an *escaped qualified name* which means that it is either a qualified name (i.e., java package naming standard), or a string enclosed in `" "`. The rationale for this naming standard is that units should retain their original names and that names in different name spaces may follow very different naming conventions than the java package naming standard. The scheme used by b3 should be convenient for typical naming types, while not preventing more exotic schemes from being used.

Simply follow the rule, “if the name is not compliant with the java package naming standard write it as a string”. This also enables full use of NLS names.

`version` `«Version»`

A unit may have an optional version declaration. The *version* is written in the *Omni Version* format which natively supports the OSGi versioning scheme, but can also represent many other versioning schemes with very different semantics. The b3 language allows unescaped version literals in most version schemes, but some exotic scheme may use keywords or special characters that interfere with the reserved words and characters in the b3 language. If that is the case, the version can be entered as a string enclosed in `" "`.

The version string is parsed by the b3 model and feedback in the form of an error marker is displayed if there are errors.

Details about b3’s use of the omni version is found in [Chapter 4, Versions](#).

`implements` `«InterfaceName»` (`«InterfaceName»`)*

A unit may optionally have a declaration of implemented (typically build related) interfaces. The interface name is an ID of an imported interface. Although possible to use any interface the intent is interfaces specifically constructed for building are used¹Support for building in a particular domain makes such interfaces available. As an example support for building eclipse related artifacts such as bundles, eclipse features and RCP products are supported via the interfaces `org.eclipse.b3.build.OsgiBundle`, and `org.eclipse.b3.build.EclipseFeature`. ***SUBJECT TO CHANGE - interface names subject to implementation.***

Having the capability to implement multiple interfaces is important as it allows a unit to formally state that it is multiple things as the same time - for instance: being an `osgi.bundle`, being in binary or source form, being in the form of projects checked out from a particular type of source code repository etc.

When a build unit implements an interface, all functions, and more importantly all builders declared for this interface are applicable for the the unit.

Unit body overview

The body of a unit is written within enclosing `{ }`. The body can contain:

- **default property set** — see [the section called “Properties”](#) for details regarding properties and default properties. The default properties of a unit will always have been evaluated in the context used to evaluate a builder for the unit (before the builder’s default properties are evaluated, if any). Example:

```
default properties {
    $target.platform = "*";
    $compiler.optimization = "-O4";
}
```

- **required capabilities** — states dependencies on the prescense of units (and other things) required in order to make the build unit buildable. Example:

¹If you are familiar with Eclipse Buckminster, the interfaces mostly resembles Buckminster's component type.

```
requires osgi.bundle/org.myorg.mybundle/[1.0.0,2.0.0];
```

- **environment required capabilities** - states dependencies on the presence of capabilities in b3 itself — i.e., things that are required in order to understand/act on the b3 file itself (e.g., support for a particular meta data translator), and thus such requirements must be resolved before other processing can take place.

By default, all java imports, and implements declaration are treated as environment required capabilities. There is no need to restate these as requirements. Typically, there is no need to use environment requirements, but in some cases, the dependency on something in the environment does not tell the full story - as an example, a resolver may be operating against a SVN repository, and normally it does not matter if the subversion support comes from Subversive or Subclipse, but if it does, it is possible to specify this as an environment requirement. Example:

```
requires env osgi.bundle/org.myorg.b3processing.special/[1.0.0,2.0.0];
```

Syntax supported, but functionality not implemented in the first version of b3.

- **provided capabilities** — states (general) capabilities that this unit is providing. Typically, a unit that is providing capabilities that should be discovered by the resolution process do so via the builder actions as these also declare the concrete artifacts that manifest the capability. Some capabilities may however not require any manifestation (other than the presence of the unit itself) and these can be declared in the unit directly. A unit is always providing itself as a unit capability without any specification being required. Example:

```
provides org.myorg.documentation.stylesheets/plainhtml;
```

- **synchronization of build actions** — when running parallel build, it may not be enough to state that individual actions or units must be processed sequentially. It is therefore possible to advice the engine to process actions in different units sequentially (even if they on their own may be safe to execute in parallel). *Synchronization syntax and semantics are not yet fully defined. The prototype introduces syntax that is subject to change. Builders are implemented as Eclipse Jobs, and should use the Jobs synchronization capabilities (syncing on resources, but also needs mechanism to synchronize on things external compilers are doing, perhaps all such synchronization are just via URIs as this provides a hierarchy - there seems to be no requirement that the URIs reference something existing, only that it is possible to compute if one resource contains another.*
- **repositories** — states which repositories to use when resolving and materializing units into the environment.
- **builders** — declarations of build functions that the b3 engine will schedule and execute. The builder functions have declarative syntax that makes this possible, but are otherwise very similar to general purpose functions.
- **named property sets** — declaration of additional named property sets that can be referenced from other property sets in the unit, or when executing builders and function. Example:


```
properties WindowsProperties {
    $zip.processor = "WinZip.exe";
}
```
- **concerns** — declarations of concerns ('overrides') that can be referenced from other concerns in the unit, or when executing builders and functions.

Capabilities

Capabilities are used during resolution where *required capabilities* are matched with *provided capabilities*. This section explains the parts common to both provided and required capabilities.

Capabilities are written on a common form:

```
«NameSpace» / «CapabilityName»
```

Where «*Namespace*» is a qualified name indicating the capability type. This typically corresponds to the interfaces that a unit implements, but a capability name space does not have to be represented by a java interface, they can be arbitrarily invented.

All units provides a capability in the `org.eclipse.b3.unit` name space — i.e the capability of ‘being a build unit’, where the capability name being the name of the unit. Since this is a capability that is commonly used — this name space is used as default if no name space is stated, and it can optionally be declared using the the keyword `unit` to make it more clear what is wanted. As an example — the capability “a build unit called ‘orange’” can be written as:

```
orange
unit/orange
org.eclipse.b3.unit/orange
```

The capability name follows the same rules as a unit name; it is either a qualified name, or a string enclosed in " ". Here are some examples:

```
osgi.bundle/org.myorg.mybundle
se.myorg.song/"Å janta å ja"
nutritional.supplement.vitamin/C
```

Provided capabilities

Provided capabilities are declared using the keyword `provides` followed by a list of capabilities. Provided capabilities may be filtered by using the keyword `when` when followed by a boolean expression in parentheses.

Provided capabilities can optionally (but typically) declare the version of what is being provided. The version follows the same semantics as the `Version` value for the unit (i.e., it is an `OmniVersion`). The version is specified with a separating / after the capability.

Here are some examples:

```
provides osgi.bundle/org.myorg.mybundle/1.0.0;
provides org.myorg.settings/serverDefaults/1.0.0;

provides when ($build.type == "server")
  org.myorg.settings/serverDefaults/1.0;
provides example/someunit, example/someotherunit;
```

Syntax is subject to change. The combination of `when()` and a list makes you think the `when()` is applied to everything in the list, but the filter is for one capability only. It is better if the filter is applied to one thing, being either one capability, or a capability block - as shown in the example below. Provided capability specification then looks like the corresponding requirements in builders.

```
// Syntax will change to this:
provides example/someunit;
provides when (cond) example/someunit;
provides { example/someunit; example/someotherunit; }
provides when ($someProperty == "some value") {
  org.myorg.settings/serverDefaults/1.0;
  org.myorg.awsomness/pizzazz;
}
```

Unit required capabilities

Required capabilities declared at the unit level can be used for several purposes:

- Declaring dependencies on the execution environment.
- Declaring dependencies that are common to multiple builders (to make it easier to modify a particular dependency as it is in one location only).
- Declaring general dependencies that affect the overall resolution, but that are not directly referenced by builders (e.g., unit has dependency on something that acts as a configuration that when resolved provides many different capabilities required by builders).

Typically, a unit does not declare any required dependencies — these are instead declared in the builders that have the concrete need. This makes it easier to maintain a build unit's dependencies.

Required capabilities are declared the same way as provided capabilities but with the following differences: the keyword is `requires` (instead of `provides`), and the optional version, is an optional version *range*. The b3 parser creates an instance of the version range and any errors in version range syntax is flagged as an error in the b3 editor with an explanation. A version range is one of:

- A single `«Version»` which means *a version* `>= stated version`.
- Two, comma separated `Version` instances indicating the *min* and *max* version enclosed with a prefix of `[` or `(`, and a suffix of `]`, or `)`. The use of `[` indicates inclusion of the min version, `]` indicates inclusion of the max version. The `(` and `)` does not include the stated min and max values.

A required capability declared in the build unit can optionally have an alias which makes it possible to reference the requirement in builders. Here is an example of using an alias:

```
requires osgi.bundle/org.myorg.runtime/[1.0.0, 2.3.4) as runtime;
```

Here are more examples of `requires`:

```
requires osgi.bundle/org.myorg.mybundle/[1.0, 2.0];
requires {
    eclipse.feature/org.eclipse.executableFeature;
    osgi.bundle/org.myorg.mybundle/[1.0, 2.0];
}
```

Environment Requirements

Requirements on the environment are written like regular requirements, but with `requires env` instead of just `requires`.

Repositories

The b3 engine has support for resolution from the workspace and target platform by default. This means that whenever a build unit has a requirement on something else, it needs to be present in these locations. When evaluating builds in a build unit, it is possible to declare that other locations should be used - this is done with a `repositories` statement inside the build unit.

Typically, it is only top level build units (i.e., build units that a user interacts with in the IDE, on the command line, or from a build server) that define repositories, but using repository statements inside “downstream” build units is a very powerful and important mechanism to enable separation of concerns. First, let's look at how repositories are declared.

Declaring repositories

A build unit's repository configuration is declared using a `repositories` statement with the following syntax:

```
RepositoriesDeclaration : repositories { «RepositoryConfiguration»* } ;
```

```
RepositoryConfiguration :
```

```
    «SelectFirst»
  | «SelectBest»
  | «SelectSwitch»
  | «Repository» ;
```

```
SelectFirst : select-first { «RepositoryConfiguration»* } ;
```

```
SelectBest : select-best { «RepositoryConfiguration»* } ;
```

```
SelectSwitch : select-switch «Expression»
    ( case «Expression» : «RepositoryConfiguration» )+
  ;
```

```
Repository : ( «URI» | repository «ID» ) «ContextBlock»? ;
```

repositories

This keyword marks that this is a declaration of repositories to use from this point forward. A list of repository configurations are enclosed in curly brackets. The list of repository configurations is a *select-first* configuration (see below).

RepositoryConfiguration

A repository configuration is one of the selection strategies *select-first*, or *select-best* or a concrete repository definition.

select-first

A *select first* strategy is defined with this keyword. It means that the first returned resolution from the list of enclosed repository declarations will be used.

select-best

A *select best* strategy is defined with this keyword. It means that all the repositories in the list of repositories is given a chance to resolve a request and that the resolution that scores highest against the resolution options for the request (source/no source, mutable source/not mutable source, etc.) will be used.

select-switch

A *select switch* strategy is defined with this keyword. It means that a selection is made using a switch-case expression where the expression following *select-switch* is compared against each *case* in turn. The repository declaration for the first matching case is selected to be used for resolving a request. The final constant value *request* is bound to an instance of *RequiredCapability* thus enabling the switch expression and cases to make comparisons on *request.name*, *request.nameSpace*, *version range* etc.

The *select-switch* configuration is useful when resolution requests are slow, and you really just want to test against one particular repository for a matching request.

Repository

A concrete repository entry is either a URI string, or the keyword *repository* followed by a type reference to a repository handler class. The URI is a b3 specific URI that starts with a repository scheme name that has been registered with the b3 engine. The repository scheme name maps to a repository handler, and the rest of the URI is interpreted by the specific handler. Here is a simple example:

```
"p2:http://downloads.eclipse.org/galileo"
```

ContextBlock

The URI-string, or the repository ID is optionally followed by a with context block (described in detail in [the section called "With context expression"](#)) allows the repository handler instance to be configured / initialized. The capabilities vary between different types of handlers, and some may use only simple literal values, but some may also provide advanced features using lambda functions.

Repositories examples

The simplest repositories to declare are p2 repositories. The default configuration of a p2 repository handler will resolve against the repository and when units are required, they will be installed into the current target platform. If that is what you want, all that has to be stated is the URL of the repository. Here is an example, using a first found strategy and some p2 repositories:

```
repositories { select-first {  
    "p2:http://downloads.eclipse.org/galileo"  
    "p2:http://downloads.eclipse.org/myproj-updates-3.5/"  
}  
}
```

If you also want to look for things in source code repositories, and provide the ability to resolve against either binary repositories or source code repositories, you need a *select-best* strategy and declaration of the source code repositories.

```

import org.eclipse.b3.repositories.svn; repositories { select-best {
  select-first { // list of p2 repositories as before
  }
  repository "svn:https://somewhere.org/somerepo/trunk" {
    mutable = false; // do not use this repo for mutable source requests
    // set more options
  }
  repository "svn:svn+ssh://myorg.org/ourrepo/trunk" {
    mutable = true; // the default, but restated for clarity
    // set more options
  }
}
}

```

The settable options naturally vary with the type of repository that is being used, you need to consult the reference documentation for each type of repository handler to get the complete documentation, but since b3 knows about the type of repository being used, its features, and any documentation available for those features, you will find what you need using code completion.

The repository handlers are designed to allow for the most commonly used configuration by default, variations on common themes by setting options, which includes using literal functions / lambdas when something needs to be computed. As an example, the default translation of the final URL to use in a SVN repository is done by taking the location URI (which always should refer to the SVN trunk) and appending the unit name. If something else is wanted, it is possible to set the boolean options `moduleBeforeTag`, and `moduleBeforeBranch`, and the option `subModule` - if these are declared, the resulting URL will be automatically constructed. Finally, if these options are not enough, by assigning a lambda with the signature `(Request)=>URI` to the option `unitURI` can be implemented to return the wanted URL.

It is up to the implementer of a particular repository handler to decide on available options, and if advanced option using lambdas can be used.

Here is an advanced exmple using mapping between units and maven identifiers for a maven repository²:

```

repository "maven:http://..." {
  // option like location etc. omitted...

  /**
   * Map a.b.c.d to a/b.c.d, and everything in org.myorg.
   * as org.myorg/*. Use default mapping for all other requests
   * (by producing null).
   */
  mapRequest = {<MavenId> Request request |
    switch request.name
      case "a.b.c.d" : new MavenId() {
        groupId= "a"; artifactId= "b.c.d";
      }
      case ~/org\.myorg\.*/:
        new MavenId() {
          groupId="org.myorg";
          artifactId = name.substring("org.myorg.".length());
        }
    endswitch;
  }
}

```

Since b3 uses qualified names for units without distinction of the maven `groupId`, and `artifactId`, a mapping between a fully qualified name and the maven concepts is required. The maven repository handler has a default way of doing this, and by using a lambda, this default mapping can be augmented. In the example above a switch statement is used to detect the special "a.b.c.d" unit, and all units under `org.myorg` — for these, appropriately configured `MavenId` instances are returned. For all other instances the switch produces nothing, and null is returned, which tells the maven repository handler .that the default translation should be used.

²This example is subject to the actual implementation of the maven repository handler - it is not available when this was written.

Containers

The term *container* is used to denote a writeable place where build units can be stored. A build unit may define containers that are used to store build units that are found by the resolution process. The b3 engine has one predefined container called `workspace`, and when PDE support is installed there is also a `targetPlatform` container. These refer to the running instance's workspace location, and the active target platform.

It is possible to override the default definition, and also to add other containers/locations.

The syntax for defining containers is:

```
ContainersDefinition : containers { «ContainerDefinition»* } ;
ContainerDefinition : container «ID» agent «ID» «ContextBlock»? ;
```

The syntax is straight forward. A list of containers can be defined. Each container has a name, and a reference to an agent class. The agent can be configured/specialized by providing a context block. The b3 engine has three agent types available in the default configuration of b3:

- `workspace`
- `p2`
- `filesystem`

Container selection. The default is to place all units in source form in the container named `workspace`, and all artifacts from a p2 repository into a container called `targetPlatform` (and these by default are references to the running instance's workspace and active target platform). This gives two possibilities — all repository configurations can be modified to use container IDs that refer to new containers that you have defined, or you simply redefine the `workspace` and `targetPlatform` containers to suit your needs.

When this document was prepared, the details of the workspace and p2 agents were not finished, but based on the corresponding support in Buckminster, it will look something like in the following examples:

```
containers {
  container targetPlatform agent p2 {
    location = new Path($targetPlatformDir, "standard");
    conflictResolution = ConflictResolution.UPDATE;
  }
  container workspace agent workspace {
    location = new Path($user.home, "workspaces/myworkspace");
    conflictResolution = ConflictResolution.UPDATE;
  }
  container auxWorkspace agent workspace {
    location = new Path($user.home, "workspaces/myauxworkspace");
    conflictResolution = ConflictResolution.FAIL;
  }
}
```

DISCUSS: How should the active target platform be switched while running b3?

Synchronization

TODO: Synchronization is currently specified as synchronizing different builders across different units. Synchronization syntax and semantics are not yet fully defined. The prototype introduces syntax that is subject to change. Builders are implemented as Eclipse Jobs, and should use the Jobs synchronization capabilities (syncing on resources, but also needs mechanism to synchronize on things external compilers are doing, perhaps all such synchronization are just via URIs as this provides a hierarchy - there seems to be no requirement that the URIs reference something existing, only that it is possible to compute if one resource contains another.

Builders

A *Builder* is a function with additional declarative syntax used by the b3 engine to plan, schedule and execute a build. In addition to the traits of a general purpose function, a builder has declarative input (dependencies on other builders), declaration of source, and declaration of output, a predetermined return type, the ability to declare provided capabilities, and asserts triggered before the builder is run, once its dependencies have been processed, and after the builder is finished.

Builders also differ from regular functions in that they are called asynchronously — calling a builder produces a `B3BuildJob` (a specialization of an Eclipse Job) that when scheduled will run the builder's logic and produce a build result. This is however all taken care of by the b3 engine.

Here is the syntax for a builder:

```
Builder :
  «DOCUMENTATION»?
  «Visibility»?
  «ExecutionMode»?
  final?
  builder
  «BuilderName» (( «ParameterDeclarationList» ))?
  ( provides «ProvidedCapabilityList» )?
  ( precondition ((: «Expression» ;) | ({ «ExpressionList» })))?
  ( postinputcondition ((: «Expression» ;) | ({ «ExpressionList» })))?
  ( postcondition ((: «Expression» ;) | ({ «ExpressionList» })))?
  {
    ( default properties «PropertyBody»)?
    ( «Input» )?
    ( «Source» )?
    ( «Output» )?
    «ExpressionList»?
  }
  ;
```

«DOCUMENTATION»

Java Doc styled documentation for the builder.

«Visibility»

The visibility is one of `private` or `public` with a default of `public`.

«ExecutionMode»

The same as execution mode for a unit; `sequential` or `parallel`, with a default of `parallel`. Parallel means that if the builder has multiple input dependencies these will be executed in parallel. The unit and the global execution mode must also be set to `parallel` in order for parallel execution to take place.

final

The keyword `final` makes it impossible to override the builder.

builder

The keyword `builder` states that a builder is being defined.

«BuilderName»

The name of a builder follows the same rules as a unit name (i.e., a qualified name, or a name enclosed in " ").

«ParameterDeclarationList»

The builder can optionally have a comma separated list of *ParameterDeclaration* (see below) thus creating a *parameterized builder*. As shown later, the parameters can be passed to the builder when the builder is used in an *Input* expression.

«ParameterDeclaration»

The parameter declaration is the same as for a general function i.e., *Type? ID*, with the last declaration in a parameter declaration list is optionally prefixed with `...` to indicate a variable number of arguments of the specified type (this type is optional and defaults to `Object`).

`provides` *«ProvidedCapabilityList»*

The keyword `provides` declares that the result of this builder is a manifestation of the stated capability. The stated provided capabilities automatically become capabilities provided by the unit. The *ProvidedCapabilityList* is a comma separated list of *ProvidedCapability* (as shown elsewhere).

conditions

The keywords `precondition`, `postinputcondition`, and `postcondition` are used to declare assertions. All conditions are entered as either a `:` followed by a single expressions, or as a block expression. The intent is to write assertion expression that throw exceptions if condition are not met. Several assertion functions are available for this purpose — see *REF: Assertion function*.

In addition to what is described per condition type below, the conditional expressions also have access to any declared builder parameters.

The purpose of the assertions is to enable early detection of build related problems. A builder action may produce files, but there are cases when empty files are generated, expected files are missing, or have the wrong checksum etc.



Note

An alternative to using the `postcondition` assertion is to simply perform the checks as expressions in the builders body and throw exceptions instead of returning the output, but this requires a bit more coding, and makes it more complex to turn such assertions on/off from a calling context. This is important as assertions may take a long time to execute, and are probably only turned on when some problem is detected with the build.

`precondition` *«Expression»*

The `precondition` is evaluated before the builder is evaluated, but the default properties in the unit and builder have been evaluated.

The `precondition` has access to the arguments passed to the builder, but does not have access to the input, source, nor output.

`postinputcondition` *«Expression»*

The `postinputcondition` is evaluated after all input has been evaluated, and it has access to evaluated input, source, and output. The variables are always bound to a result even if the corresponding declaration is missing, in which case the structures are empty.

`postcondition` *«Expression»*

The `postcondition` expression has access to input, source and output. The post condition also has access to the builder's result via the variable `builder`. (Note that the default annotations in output are only evaluated if the output also is the produced result of the builder).

`default properties` *«PropertyBody»*

The keywords `default properties` are used to declare a default property set. The *PropertyBody* is the same as shown in [the section called "Properties"](#). The default properties come in effect if the corresponding properties are not already set. The default properties are evaluated before any other declaration in the builder is evaluated.

«Input»

The input declares input from other builders. This optional list of other builders is automatically processed by the b3 build engine in the correct order. The input is described in detail later. The input declaration is where a builder's dependencies are declared. If the builder only consists of an input declaration, the result of merging the result is also the produced result of the builder.

Source

The source declares the resources considered to be the builder's source (if it has any). If a builder has source declared (but no output), and contains no processing, the source is the result produced by the builder. Using source declaration is described in detail later.

«Output»

Output is the declarative output of a builder that performs processing of source (or additional processing of aggregated input, possibly in combination with source). The output declaration consists of a list of resource references and annotations. In addition to being used as a description to processing logic, the output is also used when computing if the result of a builder is up to date or not. An output declaration can even be used when there is no input or source (the result is derived from something else) for the purpose of conveniently creating a result from the builder. If output is declared, and the builder has no processing, the output is the result produced by the builder. The output is described in detail later.

«ExpressionList»

The expression list is a semicolon separated list of expressions evaluated in order. The expression list is optional if the builder has declared input, output, or source. The output is returned if declared, and if no output is declared, source is returned (if declared), and finally the input is returned (if declared). If none of input, source or output is declared, the expectation is that the builder's list of expressions produce a `BuildSet`³ containing the wanted result. If there is no such expression an empty result is returned (an empty `BuildSet`).

**Note**

To produce a result, any b3 language expression can be used, and the evaluated input, source and output are available via the variables `input`, `source` and `output`. These are all of type `BuildSet`. This allows for advanced processing of the evaluated result of the corresponding declarative statements, as well as free form creation of a result by instantiating a `BuildSet` object, or modifying a `BuildSet` bound to either `input`, `source`, or `output` (changes to a build bags has no effect on the underlying declarative model). The `BuildSet` is further explained in [the section called “The BuildSet”](#).

Before reading more about the details, here are some builder examples:

```
/** simple builder with just source */ builder documents {
  source { docs/ [ intro.html, faq.html]; }
}

/** a builder that aggregates the result of several other builders */
builder everything {
  input {
    #documentsZip;
    unit/org.myorg.core#someStuff;
    unit/org.myorg.myapp#everything;
  }
}

/** a builder that processes the result of another builder */
builder documentsZip {
  input { #documents; }
  output { documents.zip; }
  MyZipUtil.create(output, input);
  output;
}
```

Input

The `input` declaration states requirements on evaluation of other builders (in the same build unit, or in other build units). The declaration can be filtered, and the evaluation of other builders can be advised. The declarative input has the following syntax:

```
Input : input { «Prerequisite»+ } ;

Prerequisite :
```

³`BuildSet` is the return type of all builders.

```

    ( when ( «Expression» ) )?
    ( «WithClause» :)?
    «BuilderReference»
    ( as «ID» )?
    ;

WithClause :
    with ( «concernID» ( , «concernID»)* )?
    ( ( default properties ...
      | properties ...
      | concern «ConcernBlock»
    )
    )*
    ;

BuilderReference
: «DirectBuilderReference»
| «IndirectBuilderReference»
| «CapabilityReferencedBuilder»
| «CompoundReference»
;

DirectBuilderReference :
    unit? # «BuilderName» «ParameterList»? ;

IndirectBuilderReference :
    «requiredCapabilityID» # «BuilderName» «ParameterList»? ;

CapabilityReferencedBuilder :
    «RequiredCapability» # «
        BuilderName»
    «
        ParameterList»
    ? ;

CompoundReference : { «Prerequisite»* } ;

```

input

This keyword is used to declare dependencies on other builders that should be evaluated before this builder starts its processing.

«Prerequisite»

Each entry in the input is a reference to another builder. It is possible to filter the list using a when expression. The evaluation of a builder can be made in a new closure using a with expression. The entry can be *aliased* using an as expression which makes it possible to refer to an individual builder's result, or an aggregation of builder results (i.e., multiple entries having the same alias) in the builders logic.

when «Expression»

The optional when expression is a boolean expression, that if it evaluates to true will include the entry in the processing.

«WithClause»

The with clause makes it possible to declare that the evaluation should be made with referenced property sets, and other concerns in effect. The with clause also makes it possible to state new properties and concerns.

«BuilderReference»

A reference to another builders is one of:

- A # followed by the name of a builder in the same unit, optionally followed by a list of arguments⁴. This may also be written with the keyword `unit` before the #.
- An ID being the alias of a required capability declared in the unit being processed, followed by # and a builder name, optionally followed by a list of arguments.

- A specification of a required capability followed by a # and the name of a builder (applicable to a unit being the resulting resolution of the requirement) optionally followed by a list of arguments.
- Or a list of multiple references as described above enclosed in { }.

as «ID»

It is possible to alias the input requirement using the keyword `as` followed by a name. This creates a local variable in the builder's context that may be referenced in the logic that follows the declarative input/source/output statements. This is a valuable mechanism as it allows for post processing of individual parts of the resulting input before returning the result, or for returning only parts of the input. See below for examples.



Note

Note that these aliases are not available in the declarative input itself — you can not declare an alias and then use it as a reference to the result inside input.



Tip

You can use the same alias for multiple builders. The produced result of all builders with the same alias will be merged and made available via a local variable having the alias as its name. See [the section called “Input examples using 'as ID'”](#) for how this can be used.

Input examples

An input specification is typically very simple, like in this example which references the result from a builder in another build unit: *osgi.bundle (etc) subject to change to an interface name*

```
input { osgi.bundle/org.myorg.mybundle/1.0#anotherBuilder ; }
```

If the referenced builder is defined for the same unit, you just have to reference it via its builder name (Or if you prefer to be more explicit, the keyword `unit` has the same function as “this” in java - the example below shows both forms).

```
input { #anotherBuilder; } input { unit#anotherBuilder; }
```

It is possible to reference a required capability declared in a build unit via its alias. You may for instance use the same requirement in many builders, and rather than repeating the same requirement multiple times, you can declare it at the unit level, and then reference it as in this example:

```
unit UsingAnAlias {
  requires osgi.bundle/org.myorg.mybundle/1.0 as MyBundle;

  builder exampleBuilder {
    input { MyBundle#someBuilder; }
  }
}
```

Input examples using ‘when’

Here is a simple example using a `when` expression to filter the input if different bundles should be used on different platforms (and this is not already handled by bundles with appropriate filters):

```
input {
  when ($target.plaform == "win32")
    osgi.bundle/org.myorg.myWinBundle/1.0#someBuilder;
  when ($target.platform != "win32")
    osgi.bundle/org.myorg.myDefaultBundle/1.0#someBuilder;
}
```

Using a compound list makes it possible to filter several requirements at the same time:

```

input {
  when ($target.plaform=="win32") {
    osgi.bundle/org.myorg.myWinBundle/1.0#someBuilder;
    osgi.bundle/org.myorg.myWinBundle2/1.0#someBuilder;
  }
}

```

Input examples using ‘with’

It is possible to use a `with` clause to evaluate a particular builder or set of builders with a specified set of concerns (properties and other advice). The mechanism allows reuse of already declared property sets and concerns as well as direct declaration of such.

As an example, we want to make it possible to run obfuscation on jar files in bundles. To do this, we first declare a concern called `Obfuscation` that injects/weaves this capability into every `osgi.bundle`. (You may want to do more than just provide the obfuscated bundles via a separate builder — it could for instance be useful to be able to make all references to the jars of a bundle always get the obfuscated version - see [the section called “Concern examples”](#). Here is the example where a builder called `obfuscatedJars` is added to all OSGi bundles.

```

import org.someorg.Obfuscator; concern Obfuscation {
  context unit implements osgi.bundle {
    builder obfuscatedJars {
      input { unit#jars }
      Obfuscator.obfuscate(input);
    }
  }
}

```

The defined concern can then be applied when evaluating input. Here is an example, where some advanced graphics code in multiple bundles are obfuscated. To keep things simple, we assume that this builder is declared in the same unit as the `Obfuscator` concern.

```

builder graphicSubsystem {
  input {
    with Obfuscation : {
      osgi.bundle/org.supergraph.layout/[1.0,3.0]#obfuscatedJars;
      osgi.bundle/org.supergraph.animation/[1.0, 3.0]#obfuscatedJars;
    }
  }
}

```

The use of the `with` expression will weave the `Obfuscation` code into the model, and then evaluate the builders. It is now possible to use the `obfuscatedJars` builder on these bundles. Once the input has been processed, the `Obfuscation` is unwoven⁵.

The `with` expression accepts references to declared concerns, and property sets. As an example, you may need to feed the obfuscater logic with different properties for different platforms.

```

import org.someorg.Obfuscator;
concern Obfuscation { /* as in the previous example */ }

properties ObfuscationProperties {
  $org.supergraph.obfuscation = "default";
  when ($target.platform == "win32")
    $org.supergraph.obfuscation.type = "full";
}

builder graphicSubsystem {
  input {
    with Obfuscation, ObfuscationProperties : {
      osgi.bundle/org.supergraph.layout/[1.0,3.0]#obfuscatedJars;
      osgi.bundle/org.supergraph.animation/[1.0, 3.0]#obfuscatedJars;
    }
  }
}

```

⁵actually, the context containing weaved/wrapped elements just goes out of scope.

If the advice and properties are only required in one particular place, you can state them directly in the `with` part, or in combination with referenced concerns and properties. The examples shows declaration of a property set (but it is also possible to declare any concern).

```
builder graphicSubsystem {
  input {
    with Obfuscation
    properties {
      $org.supergraph.obfuscation = "default";
      when ($target.platform == "win32")
        $org.supergraph.obfuscation.type = "full";
    }
    : {
      osgi.bundle/org.supergraph.layout/[1.0,3.0]#obfuscatedJars;
      osgi.bundle/org.supergraph.animation/[1.0, 3.0]#obfuscatedJars;
    };
  }
}
```

Here is a typical combination of a concern adding a builder and a default property set:

```
import org.someorg.Obfuscator;
concern Obfuscation {
  context unit implements osgi.bundle {
    builder obfuscatedJars {
      input { unit#jars }
      Obfuscator.obfuscate(input);
    }
  }
  default properties {
    $org.supergraph.obfuscation = "default";
    when ($target.platform == "win32")
      $org.supergraph.obfuscation.type = "full";
  }
}
```

This declaration means that whenever the `Obfuscation` concern is brought into effect the properties defined in the default property set will be set if not already set from the command line, or in an outer scope.



Note

The `with` expression is a powerful general expression that you may also use in the non builder logic. There are many more possibilities using concerns, and properties (they can extend a concern or property declaration, you can place a property set inside a concern etc.).

Input examples using 'as ID'

Aliases in the input are useful to enable referencing (and merging) different parts of the input in the subsequent logic. Here is a simple example, where several builders are evaluated, but only a subset is returned.

```
builder example {
  input {
    unit#anotherBuilder;
    { unit#builderA; unit#builderB; } as important;
  }
  important;
}
```

In this example, the b3 engine will resolve all of the references and evaluate them (i.e., `anotherBuilder`, `builderA`, and `builderB`). An alias `important` is defined for `builderA` and `builderB`. The final expression `'important;'` defines the produced result of the builder — the combined result of `builderA` and `builderB`.

You can merge disjunct results by using the same alias as in this example:


```

builder example {
  input {
    unit#anotherBuilder as regular;
    { unit#builderA; unit#builderB; } as important;
    with SomeConcern : unit#yetAnotherBuilder as regular;
  }

  // variables 'regular' and 'important' refers to the respective merged results
}

```

The ability to merge disjunct results reduces the need to introduce intermediate builders to group certain results.

Using parameterized builders

It is possible to use parameterized builders. This is not much different than having functions with parameters.

It is expected that top level builders (i.e., builders invoked from the command line, or from a user interface) are declared without parameters to make it easy to invoke them - a user interface may filter them out for instance to avoid having to ask the user for input in a complicated dialogue.

Calling functions with parameters is a general feature in the b3 language, and the topic is explored in detail in the section about functions. Here is a simple example using a parameterized builder:

```

builder bob(String brandName) {
  /* bob does something interesting with brandName, like using it to filter,
  passing the value to an external builder, etc.
  */
}
builder pat {
  input { bob("Super Zlide - the ultimate gaming experience"); }
}

```

There is always one implicit parameter — `unit`, which always refers to the build unit being processed. When defining builders inside a build unit, this parameter is never declared (it is implicit that these builders will be used only for the unit in which they are declared). When declaring builders outside the scope of build unit (in concerns), the `unit` parameter is explicit.

Source

The source of a builder can be declaratively stated. The simplest form is a reference to a top level source folder as in:

```
source { src; }
```

The source declaration can however be more complex, listing sets of references to files, and use filters to only include references under certain conditions. The source declaration can also include annotation — properties set when the source declaration is evaluated.

A source declaration evaluates to a `BuildSet`, the same structure that is produced as the result of a builder. This means that the source declaration can simply be used as the result of a builder if all that is wanted is a set of references to the source.

Here is the syntax for input:

```

Source : source { URIVectorElement* Annotations? } ;

URIVectorElement : FilteredElement | UnbasedElement | BasedElement ;
UnbasedElement : URI (, URI)* ; ;
BasedElement : URI [ URI (, URI)* ] ; ;
FilteredElement :
  (when ( Expression ) (UnbasedElement | BasedElement | CompoundElement) ) ; ;
CompoundElement : { URIVectorElement* } ;

Annotations : annotations { AnnotationsPropertyStatements } ;
AnnotationsPropertyStatements : ... // the same as a property set

```

The URI elements used are resolved before use; the URI fragments (enclosed in [] in the *BasedElement*) are resolved against the base URI (the URI before the []), and all (resulting) URI elements are resolved against the `sourceLocation` URI found in the build unit being processed.

Output

It is possible to declaratively state the output of a builder — i.e., where the result of the build action logic is supposed to be placed (or rather “is placed” in the case of invoking some external action that outputs things in some location).

The absolutely simplest use of `output` is when the result is already existing in the file system (produced by some action outside of b3’s direct control), then all that is needed is to return the declared output. In other situations, some logic is processing the input or source and needs to know where generated artifacts are supposed to end up in the file system. The output thus has dual use — as a specification of what to return, and as a parameter to the build logic.

The declaration of `output` is exactly the same as the declaration of `input` except for the keyword `output` (instead of `input`). The resolution of URI elements in the `output` declaration is performed against the processed unit’s `outputLocation`.

Here are some examples:

```
output { plugins/; features/; }

output { tmp/ [ plugins/, features/]; }

// using filters
output { when ($target.platform == "win32")
          win/ [ plugins/, features/];
         when ($target.platform != "win32")
          nonWin/ [ plugins/, features/];
        }

// nested construction with filters
output {
  when ($target.os == "linux") {
    when ($target.ws == "motif")
      xwindows [x11/, motif/];
    when ($target.ws != "motif") gtk/;
  }
}
```

Annotations in input, source and output

Annotations are stored as typed key value pairs in the `BuildSet` instances resulting from evaluating `input`, `source`, and `output` (as well as in the merged local aliased results). Typically annotation are produced by the building logic — to hold information about the built result that would not easily be obtainable otherwise (such as resolution of version qualifiers), but it is possible to declare annotations directly using the same syntax as for property sets.

When annotations are declared they are evaluated as follows:

- In `input`, the annotation act as default properties applied on the merged annotation of all the produced result. Aliased merged sub results however only contain the annotations merged from the included builders. *Note: when this document was written, the ability to declare (default) annotations in the input was missing from the implementation.*
- In `source`, the annotations act as regular properties as they are applied when the source declaration is evaluated.
- In `output`, the annotations act as default properties and are evaluated if the output is the produced result from he builder. This enables the builder’s logic to manipulate the `outputBuildSet`’s an-

notations until it is returned. See [the section called “The BuildSet”](#) for a description of available BuildSet functions.



A note about the rationale for annotations

You may wonder about the rationale to use the property like annotations instead of a more type safe and strict approach, and this deserves a comment. One goal with b3 is to interface well with many different kinds of external (existing) build systems, and these are often of command line type and script based tools that have limited ability to deal with parameters in calls. Using properties to communicate with such tools have proved to be successful in Eclipse Buckminster and we extended the support for properties to also be able to return them in the form of annotations.

DISCUSS: Should we have a more restrictive policy? It is good to have annotations be declared. If an external builder tries to set something else - that could be an error. This to make it possible to detect which annotations that are useful to reference. The downside is when there is an actors that produces a lot of annotation - it has to be declared every time, or the logic that wraps some actor would need to only update annotation that were actually declared. Yet another alternative is to specify an interface that the annotations should comply with. Using reflective modeling is also an option worth exploring.

The builder’s logic

As already described in the previous sections about builder, the builder’s logic does not have to be stated if all that is wanted is the return of the evaluated input, source, or output. Also shown in earlier sections is that the variables input, source, and output refer to the corresponding evaluated result (of type BuildSet). The merged aliased subresults from the input are also available via variables with the same names as the aliases (these are also of type BuildSet).

The variable unit is also bound in the builder’s context. It refers to the build unit being processed by the builder (much like the implicit variable this in Java).

The BuildSet

The BuildSet is a data structure used to carry data between Builders. The same data structure is used for both input and output, and it consists of a collection of URI elements, and typed named annotations.

See the sections called Input, Source, and Output for examples of how BuildSet instances are created and passed between builders. Most of the time the functionality available via the declarative statements are enough, but when authoring more complicated build logic, and interfacing with external tools, there is a need to use the BuildSet API directly.

Here is a selection of the funtions available for BuildSet.

boolean **containsValue** (BuildSet *b*, String *name*)

Returns true if the annotation is defined in the BuildSet (may have the value null)

Object **defineValue** (BuildSet *b*, String *name*, Object *value*, Type *class*)

Defines a constant value have the given name and type. Returns the defined value.

Object **defineFinalValue** (BuildSet *b*, String *name*, Object *value*, Type *class*)

Defines a constant final value have the given name and type. Returns the defined value.

Object **defineVariableValue** (BuildSet *b*, String *name*, Object *value*, Type *class*)

Defines a constant value have the given name and type. Returns the defined value.

Object **defineFinalVariableValue** (BuildSet *b*, String *name*, Object *value*, Type *class*)

Defines a constant final value have the given name and type. Returns the defined value.

```
Iterator<URI> getPathIterator ()
```

Returns an `Iterator<URI>` that iterates over all URI references in the `BuildSet`.

```
Object getValue (BuildSet b, String name) throws B3EngineException
```

Returns the value of the value/variable with the given name. Throws `B3EngineException` (`B3NoSuchVariableException`) if the value was not defined.

```
BuildSet merge (BuildSet b, BuildSet mergedSet)
```

Merges (adds) all the URI references and values from the `mergedSet` into `b`. Returns `b`.

Unit & Builder Concern

As already shown in [the section called “Concern”](#) a *concern* is an aspect oriented programming concept which groups a set of advice that is dynamically woven into the fabric of the logic. Concerns have content that have an effect on everything, or on something in a specified context. One such context is shown in [the section called “Function concern context”](#), where it is explained how functions can be advised. In this section we focus on how build units and builders can be advised.

Advising build units is done in a *unit context*, and builders are advised in a *builder context*. Both of these have queries/predicates that define which units/builders to advise, and then a series of +/- expressions that add or remove features.

Builders can also be introduced directly in a concern for the purpose of adding or overriding an existing builder.

Adding or overriding builders

Builders can be added directly in a concern. Such a builder can override an existing builder with the same signature if the existing builder is not marked to be final. A builder defined this way in a concern has the same syntax as when defining a builder inside a build unit (as shown in [the section called “Builders”](#)), with one difference — the builder must be declared with parameters, and have a first parameter called `unit` which must be of a type that implements or extends the `BuildUnit` interface. Here is an example:

```
concern FruitProcessing {
  builder waste(Fruit unit) {
    input { unit#peel; unit#seeds; }
  }
}
```

In this example, a builder called `waste` is added. When the concern `FruitProcess` is in effect, it is possible to invoke `waste` on any build unit that implements the interface `Fruit`. When invoked, the builder will merge the result of getting the peels and seeds from the unit.

Unit Concern

The unit concern context has the following syntax:

```
UnitConcernContext :
  «DOCUMENTATION»?
  context unit
  «UnitQuery» {
    ( «Builder»
    | «BuilderConcernContext»
    | (+ requires { («RequiredCapability» ;)+ })
    | (+ requires «RequiredCapability» ;)
    | (- «RequiresPredicate» ;)
    | (+ provides { («ProvidedCapability» ;)+ })
    | (+ provides «ProvidedCapability» ;)
    | (- «ProvidesPredicate» ;)
    )*
    (- default properties «PID» (, «PID»)* ;)?
```

```

    (+ default properties «PropertySet» )?
  }
;

```

«DOCUMENTATION»

Java Doc style documentation.

context unit

Declares that this is advice for build units.

«UnitQuery»

The unit query is a predicate expression using &&, ||, !, and () in combination with predicates for unit name, implements, requires, and provides. Some of these predicates are the same as when advising builders, as explained in [the section called “Predicates in concern context”](#). Here is a simple example:

```

~/org\myorg\.* /
  && implements osgi.bundle
  && requires osgi.bundle/org.myorg.old.x

```

«Builder»

This is a declaration of a Builder as shown in [the section called “Builders”](#).. The declared builder will be made available for all units that match the *UnitQuery*.

«BuilderConcernContext»

This is a builder concern (as described in [the section called “Builder Concern”](#)). The builders matched by this nested concern are restricted to builders for units that match the parent unit query.

+ requires ...

Adds one, or several required capabilities to the unit. A required capability is entered the same way as when declared for a build unit directly.

- «RequiresPredicate»

Removes required capabilities declared for matching units. Removal of requirements is performed before requirements are added, so to replace a requirement use a -/+ combination. The syntax for the predicate is shown in [the section called “Predicates in concern context”](#).

+ provides ...

Adds one, or several provided capabilities to the unit. A provided capability is entered the same way as when declared for a build unit directly.

- «ProvidesPredicate»

Removes provided capabilities declared for matching units. Removal of requirements is performed before requirements are added, so to replace a requirement use a -/+ combination. The syntax for the predicate is shown in [the section called “Predicates in concern context”](#).

- default properties «PID» (, «PID»)*

Removes the listed properties from the unit’s set of default properties.

+ default properties «PropertySet»

Adds a property set to the default properties specified for the unit. The property set has the same syntax as described in [the section called “Properties”](#)..

Builder Concern

The builder concern context has the following syntax:

```

BuilderConcernContext :
  «DOCUMENTATION» ?
  context builder (( «ParameterQuery» ) )?
  BuilderQuery {
    ( (- provides «ProvidesPredicate» ; )

```

```

| (+ provides «ProvidedCapability» ; )
)*

(- precondition ; )?
(+ precondition ((: «Expression» ; )|( «BlockExpression»)))?
(- postinputcondition ; )?
(+ postinputcondition ((: «Expression» ; )|( «BlockExpression»)))?
(- postcondition ; )?
(+ postcondition ((: «Expression» ; )|( «BlockExpression»)))?
(- default properties «PID» (, «PID»)* ; )?
(+ default properties «PropertySet» )?

( (+ input «Prerequisite»)
| (- «InputPredicate» ; )
)*

( (+ source «ConditionalURIVector»)
| (- source «SourcePredicate» ; )
)*

(- annotations source «PID» (, «PID»)* ; )?
(+ annotations source «PropertySet» )?

( (+ output «ConditionalURIVector»)
| (- output «OutputPredicate» ; )
)*

(- annotations output «PID» (, «PID»)* ; )?
(+ annotations output «PropertySet» )?

«ExpressionList»?
}
;

```



Note

A builder concern context can appear in a concern, or inside a unit context. A builder concern context wrapped in a unit concern context will only affect builders that are applicable to the units matched in the unit context.

«DOCUMENTATION»

Java Doc style documentation.

context builder

Declares that this is advice for a builder.

«BuilderQuery»

The builder query is a predicate expression using `&&`, `||`, `!`, and `()` in combination with predicates for builder name, builder parameters, requires (in input), and provides. Some of these predicates are the same as when advising units, as explained in [the section called “Predicates in concern context”](#). Here is a simple example:

```
generatedDocs provides doc.doctype.extensionref/_
```

(«ParameterQuery»)

An optional parameter query can be specified in parentheses to restrict the affected builders to those matching the parameter query. The parameter query is the same as when advising function as shown in [the section called “Function concern context”](#).



Note

If a parameter query is used, the first parameter for unit must be included in the query or there will be no matches. This is also true when a `context builder` is used inside a `context unit` since the surrounding unit query can match units of different type.

- + `provides ...`
Adds one, or several provided capabilities to the builder. A provided capability is entered the same way as when declared for a builder directly (as shown in [the section called “Builders”](#)).
- `«ProvidesPredicate»`
Removes provided capabilities declared for matching builders. Removal of requirements is performed before requirements are added, so to replace a requirement use a `-/+` combination. The syntax for the predicate is shown in [the section called “Predicates in concern context”](#).
- `precondition`
Removes all precondition expressions.
- + `precondition ...`
Appends one expression, or a block of expressions to the builder’s existing pre-conditions (which may be empty).
- `postinputcondition`
Removes all post-input-condition expressions.
- + `postinputcondition ...`
Appends one expression, or a block of expressions to the builder’s existing post-input-conditions (which may be empty).
- `postcondition`
Removes all post-condition expressions.
- + `postcondition ...`
Appends one expression, or a block of expressions to the builder’s existing post-conditions (which may be empty).
- `default properties PID (, PID)*`
Removes the listed properties from the builder’s set of default properties.
- + `default properties «PropertySet»`
Adds a property set to the default properties specified for the builder. The property set has the same syntax as described in [the section called “Properties”](#).
- + `input «Prerequisite»`
Adds a prerequisite (which may be compound) to the builder’s input. The prerequisite is entered the same way as for regular input as described in [the section called “Input”](#).
- `input «InputPredicate»`
Removes a prerequisite from the input. The syntax for the predicate is shown in [the section called “Predicates in concern context”](#).
- + `source «ConditionalURIVector»`
Adds a source vector (which may be conditional and compound) to the builder’s source. The vector entered the same way as for regular source as described in [the section called “Source”](#).
- `source «SourcePredicate»`
Removes an URI from the source. The syntax for the predicate is shown in [the section called “Predicates in concern context”](#).
- `annotations source «PID» (, «PID»)*`
Removes the listed properties from the source annotations.
- + `annotations source «PropertySet»`
Adds a property set to the default annotations specified for the source. The property set has the same syntax as described in [the section called “Properties”](#).

- + output «*ConditionalURIVector*»
Adds an output vector (which may be conditional and compound) to the builder's output. The vector is entered the same way as for regular output as described in [the section called "Output"](#).
 - output «*OutputPredicate*»
Removes a URI from the output. The syntax for the predicate is shown in [the section called "Predicates in concern context"](#).
 - annotations output «PID» (, «PID»)*
Removes the listed properties from the output annotations.
 - + annotations output «*PropertySet*»
Adds a property set to the default annotations specified for the output. The property set has the same syntax as described in [the section called "Properties"](#)..
- «*ExpressionList*»
The list of expressions is executed instead of a matched builder's original logic. A `proceed` expression may be used to call the advised builder's logic. This works the same way as for function, and more details can be found in [the section called "Function concern context"](#).

Predicates in concern context

This section contains an explanation of the predicates used in build unit and builder concern contexts.

name

A name predicate is used to match a name. The predicate is one of a qualified name, or a qualified name enclosed in " ", a literal regular expression, or a wildcard '_ '.

builder name

A builder name predicate is a name predicate.

unit name

A unit name predicate is a *name predicate* optionally followed by a / and a version range literal. Example:

```
org.myorg.mybundle/[1.0.0,2.0.0]
```

implements

An implements predicate is used to match build units that implements a particular interface and has the syntax `implements «TypeRef»` as in:

```
implements OsgiBundle
```

provides

A provides predicate is used to match build units or builders that provides a particular capability. The syntax is `provides` followed by a *capability predicate*.

requires

A requires predicate is used to match build units that require a particular capability. The syntax is `requires` followed by a *capability predicate*. Matching environment requirements is done by using `requires env`.

capability predicate

A capability predicate is used in other predicates. It has the following syntax:

```
«NamePredicate» / «NamePredicate» ( / «VersionRangeLiteral»)?
```

Where the first *name predicate* is for the capability name space, the second for the capability name. The optional version range literal specifies the range of matched versions.

input

An input predicate is used to match a builder with a particular input, and to specify removal of input from a builder. The syntax is:


```
(«CapabilityPredicate» | «UnitNamePredicate» | unit?) # «BuilderNamePredicate»
```

i.e., either a full capability predicate, or a predicate for a unit name, or a reference to a builder in the same unit (as the matched builder). (The use of the keyword `unit` is optional). The name predicate following the `#` is a name predicate for the builder name.

source

A source predicate is used to specify removal of a particular source URI. The predicate is a *URI predicate*.

output

An output predicate is used to specify removal of a particular output URI. The predicate is a *URI predicate*.

URI predicate

An URI predicate is part of a source or output predicate and is written on the form:

```
(«Path» [ «Path» (,«Path»)* ])
| «Path»
| «LiteralRegexp»
;
```

where *Path* is a URI fragment (which can be entered without quotes if it is a simple path). Here are three examples of URI predicates:

```
src/ [a, b]
src/c
"resource:/org.myorg.myproj/src" [a, b]
```

Concern examples

Adding a builder to osgi.bundle

```
import org.someorg.Obfuscator;
concern Obfuscation {
    builder obfuscatedJars(OsgiBundle unit) {
        input { unit#jars }
        output { obfuscated;/ }
        Obfuscator.obfuscate(input, output);
        output;
    }
}
```

If you want to replace a property set with another and override a property you can do as follows. This will make all references to the property set `targetProperties` refer to the the property set defined in the concern:

```
/**
 * Repairs an issue where some products have the wrong platform name
 * in their property sets. Using a regular expression as there are too
 * many products with too many versions to enable them to list them.
 */
concern FixProperties {
    context unit osgi.bundle/~/org\.myorg\.product\..*/ {
        properties targetProperties extends super.targetProperties {
            $target.platform="win32"; // override
        }
    }
}
```



4

Versions

Eclipse b3 supports versions and version ranges from different versioning schemes. If you are working with Eclipse and OSGi based components, you are probably already familiar with how they work — and you can continue to use both versions and version ranges expressed just like they are expressed everywhere else in the Eclipse user interface. If you however step outside of the OSGi realm, there are many different versioning schemes in use. Eclipse b3's version handling is based on the *omni version* implementation found in Equinox p2.

If you are only developing for Eclipse and OSGi, you will still benefit from the general overview of version and version ranges, and the handling of version qualifier substitution. If you are using Maven you will also need to learn about how to handle these types. Finally, if you are working on extending b3, or if you want to use b3 in domains that use unique version formats you need to understand more about the full omni version scheme.

The really detailed implementation details are found in [Appendix D, Omni Version Details](#).

Omni Version introduction

The omni version is a canonical format. There is only one implementation and it is capable of describing versions in a wide range of versioning schemes. There is no central registry of version formats — each version or range instance carries the full specification. That each version carries the full definition means that versions can be transmitted between systems without risk of not functioning because of a missing definition. The fact that there is only one implementation means that there is no risk of not functioning because a particular implementation is not available in a system.

The omni version's canonical format is called the *raw* format, and it is constructed by parsing an *original version string* using a *format*. Since the raw format retains the format and original version strings, it is possible to recreate the input.

Omni versions always compare version based using the translated raw format. This creates a strict ordering of all versions across all versioning schemes.

Example 4.1. An OSGi version expressed in raw

```
raw:1.0.0.'r1234'/format(n[.n=0;[.n=0;[.S=[a-zA-Z0-9_-];]]):1.0.0.r1234
```

In [Example 4.1, “An OSGi version expressed in raw”](#) you can see what the OSGi version 1.0.0.r1234 looks like in raw format. Luckily, when using b3, you don't have to use such strings in your input as you will see in the next section.

b3 and omni version

When specifying versions (and version ranges) in b3 textual form, the format pattern is referred to via a defined name. *NOT YET IMPLEMENTED: This feature is not yet implemented. Currently, version and version range string are passed 'as is' to the respective constructors/factory methods. In order to support this, b3 needs extension points that specify the format, and the terminal converters for version and version range needs to interpret and apply the format. See [bug 304948](https://bugs.eclipse.org/bugs/show_bug.cgi?id=304948) [https://bugs.eclipse.org/bugs/show_bug.cgi?id=304948].*

Internally the b3 model is based on omni version. The b3 DSL handles conversion of version and version range in textual form to omni version instances.

New named formats can be introduced via a b3 extension point. No coding is required, but the extension must be provided by a bundle.

b3's named formats

Eclipse b3 has the following named formats¹:

OSGi	The OSGi version format on the format <code>major.minor.micro.qualifier</code> where major, minor, and micro are numeric, and qualifier is a string. Major must be specified, but minor and micro defaults to 0 if omitted. The qualifier is optional.
Triplet	A version format used by Maven, and others, which is similar to OSGi, but where an empty qualifier compares as larger than any qualifier.
String	A single segment version using string comparison as performed by Java String.
Timestamp	A version format where the version is expressed as a timestamp and compared in ascending order.

For more details on the rules, and how these named formats are expressed, please see [Appendix D, Omni Version Details](#).

Version ranges

Version ranges are expressed using the following syntax:

```
{ '[' | '(' } «lower-bounds» [ ',' | «upper-bounds» ] { ']' | ')' }
```


Description:

- The range must start with a literal [or (, and end with a literal] or).
- «*lower-bounds*» is a version in the specified format
- The use of [at the beginning means that the «*lower-bounds*» is included in the range.
- The user of (at the beginning means that the «*lower-bounds*» is excluded from the range such that any $v > \text{«lower-bounds»}$ is included.
- Similarly, a] or) at the end specifies that the «*upper-bounds*» is included or excluded from the range.
- If the optional *upper-bounds* is not specified, the value of the «*lower-bounds*» is used here as well (and in this case both ends must be inclusive).
- The range must be well formed so that «*lower-bounds*» \leq «*upper-bounds*».
- A single version «*x*» can be used where a range is expected — this means any $v \geq \text{«x»}$

¹These formats are compatible with the formats of the same name as used in Eclipse Buckminster.

Part III. Examples

In this part we are showing several examples, from the simple Hello World kind, to a full build of a RCP product and p2 repository. As you probably want to run through these examples live, you should follow the instructions in [Appendix A, *Installation*](#), so you can experiment with the examples yourself.





5

Example 1 - TBD

Placeholder for examples (building POJO, Update Site, RCP, etc.)

DRAFT

Part IV. Appendix

DRAFT

Table of Contents

A. Installation	76
Installing for Eclipse SDK	76
Installing the Headless Product	77
Connectors	79
Subversion (SVN)	79
B. Eclipse	80
Eclipse technology	80
Equinox	80
Platform	80
Java Development Tools (JDT)	80
Plugin Development Environment (PDE)	80
Rich Client Platform (RCP)	81
p2	81
The Eclipse component types	81
Plugins, features and OSGi bundles	81
Fragments	82
Products	82
The Workspace	82
The Target Platform	83
Launch configuration	83
ANT	83
C. p2	85
The Installable Unit	85
Metadata repository	86
Artifact repository	86
Combined / co-located repositories	86
Profile	86
p2 internals	86
Categories	87
Publishing	87
Installing	88
The SDK agent	88
The director application	89
The p2 Installer	89
The EPP wizard	89
The Buckminster installer	89
Shipping	90
Summary	91
D. Omni Version Details	92
Introduction	92
Background	92
Implementation	93
Version	93
Comparison	93
Raw and Original Version String	94
Omni Version Range	94
Other range formats	94
Format Specification	95
Format Pattern Explanation	97
Examples of Version Formats	99
Tooling Support	101
More examples using ‘format’	102
FAQ	103
Resources	105

Appendix A. Installation

This Appendix describes how to install b3 for Eclipse versions 3.6. As this draft of the book describes work in progress, you should consult the b3 webpage for current installation instruction and information about available versions. *W.I.P — INSTRUCTIONS IN THIS PART DO NOT HAVE VALID URLs TO UPDATE SITES. PLEASE FOLLOW INSTRUCTIONS ON THE B3 DOWNLOAD PAGE.*



Tip

It is possible to develop for older Eclipse versions by using a target platform suitable for the applications you are building. Using a target platform is the recommended way of building as you can then update the IDE and headless tools independently of your built applications.

Eclipse b3 comes in two different packagings — for use in the Eclipse SDK (the IDE), and for headless use.



Warning

Do NOT install the headless features into your Eclipse SDK! (There is absolutely no reason to do this, and it will cause instabilities).



Note

The latest download instructions are always found at <http://www.eclipse.org/modeling/emft/b3/download/> [http://www.eclipse.org/modeling/emft/b3/download/]. The information in this appendix describes the instructions for the versions current in March of 2010.

Installing for Eclipse SDK

1. **Check if b3 is already available for install.** You may already have access to a repository that contains Eclipse b3. Check under *Help* → *Install New Software...* for the b3 category. If you can't find the b3 category, you need to add a repository location.
2. **Add repository location (if needed).** Installing into the Eclipse SDK is done by adding the repository you want to install from. This is done in Eclipse 3.6 by adding the repository location either under '*Help* → *Install New Software...*' or under '*Eclipse* → *Preferences...* → *Install/Update* → *Available Software Sites*' and then selecting the wanted features under '*Help* → *Install New Software...*'.

Please consult the b3 download page for an up to date list of available repositories, and alternatives such as downloading a full copy of a repository to facilitate a later local install.

For convenience, here are the current locations — please note that these links are for use with Eclipse p2 installer, and *not* for use in a web browser:

- **b3 update site for Eclipse 3.6 (the 'latest fixes')**. <http://download.eclipse.org/TBD/TBD>
TODO: b3 update site.
- **Select features.** There are several features available. They are categorized into *core* and *optional*. Please note that you are expected to *make a choice* of what optional categories you need. Do *not* select all of them.



Warning

Eclipse b3's support for *Subversive* and *Subclipse* are **mutually exclusive**. Do NOT install both.

3. **Verify.** Eclipse b3 is not highly visible in the Eclipse UI, so you may wonder if your installation was successful. You can naturally try to run one of the examples, but a quick check is to look for the menu entry *File* → *YYY...* *TODO: Command to use to verify that b3 is installed.*

Installing the Headless Product

TODO: The headless packaging of b3 is not yet available.

The *Headless Product* application is based on the Eclipse Runtime. This product is intended to be used when b3's functionality is wanted, but without using a graphical user interface — e.g., from the command line, in automated scripting, etc. The headless application contains only the bare minimum to get a working headless command line utility. To make it useful, you must install the features you need into it, and the result can then be shared as necessary.

1. **Download the director.** The (headless) director is a command line packaging of the p2 director — an installer that is a general purpose installer for software available in p2 repositories. Consult the b3 download page for the current address.
2. **Unpack the zip.** Unpack the zip file to a location where you want the director. Note that the director application is also used in many headless use cases — it is not just for installing the headless b3, so select a location that is reachable from your current `PATH`, or update the `PATH` to include the location. (You don't have to set the path if you are just installing the headless b3 as you can do this from the directory where you unzipped the director).
3. **Perform the install.** You perform the installation by running the director with the following command (type everything as a single line of input):

```
director -r «repo-location»  
-d «install-folder»  
-p b3  
-i org.eclipse.b3.cmdline.product
```

Where the command line option have the following meaning:

`-r «repo-location»`

Replace `«repo-location»` with the URL to the headless b3 repository. The location is currently `http://download.eclipse.org/TBD/` *TODO: URI to headless repository*, but you should check on the b3 download page for the latest information. Alternatively, download the entire archived repository as instructed on the download page, and the use the local URI to the location where you unpacked the repository.

`-d «install-folder»`

Replace `install-folder` with the folder/directory where you want the headless b3 installed.

`-p b3`

Type `-p b3` literally, this is the name of the p2 profile.

`-i org.eclipse.b3.cmdline.product`

Type `-i` and the entire identity literally, this is a reference to the installable unit you are installing.

4. **Install additional features (at least one is required).** The installation in the previous step installed the basic b3 bootstrap and command line shell, the only useful thing it can perform is to

install additional features. You will probably want support for Java and PDE development, and some connectors to source repositories. You can use the director as shown in the previous step to install these features or use the just installed b3 (which has a simpler syntax):

```
b3 install «repository-url» «feature-id» [ «version» ]
```

Where «*repository-url*» is the same as in the previous step, and «*feature-id*» is one of the features listed below. Optionally, a specific version can be installed. Here are the features you can install:

```
org.eclipse.b3.core.headless.feature
```

The Core functionality — this feature is required if you want to do anything with b3 except installing additional features.

```
org.eclipse.b3.maven.feature
```

Maven support. (In case you noticed, there is no special headless needed for maven, this is the same feature that is used with the user interface).

```
org.eclipse.b3.cvs.headless.feature
```

Headless CVS support.

```
org.eclipse.b3.pde.headless.feature
```

Headless PDE and JDT support. Required if you are working with Java based components.

If you use the director to install, use '-i «*feature-id*».feature.group' as the p2 IU identities for features have a 'feature.group' suffix appended to the feature identity.

- 5. Install SVN support (if required).** If you require support for Subversion (SVN), you must install this in a separate step as the required plugins have a license that is not compatible with Eclipse EPL, and they can therefore not be distributed directly from the eclipse.org repositories. Instead, Cloudsmith Inc. has made them available in a repository located at <http://download.cloudsmith.com/b3/external>. *TODO: This update site does not yet exist.*

You install either support for subversive or subclipse by issuing the following command (type everything as a single line of input):

```
director -r http://download.cloudsmith.com/b3/external
-d «install-folder»
-p b3
-i «svn-adapter-id»
```

Where the command line option have the following meaning:

```
-r http://download...
```

Use the literal location <http://download.eclipse.org/tools/TBD/headless-3.6/URITBD>, but you should check on the b3 download page for the latest information.

```
-d «install-folder»
```

Replace «*install-folder*» with the folder/directory where you have installed the headless b3.

```
-p b3
```

Type -p b3 literally, this is the name of the p2 profile.

```
-i «svn-adapter-id»
```

Type -i and then the identity of either the subclipse or the subversive integration feature. You should use `org.eclipse.b3.subclipse` for subclipse, and `org.eclipse.b3.subversive` for subversive.

**Tip**

You can prepare a file with the b3 install commands you want to perform, and tell the initial b3 to execute this file. This saves you work if you are installing the headless b3 on different machines. See *REF: Headless b3 Chapter* for more information about using a script.

Connectors

Eclipse b3 can be extended to support many different types of connectors. Here are notes regarding installation for those that require more than just installing the connector.

Subversion (SVN)

There are different ways to connect to a SVN — the b3 connector distributed from Eclipse is not enough. Unfortunately, the various SVN clients all contain code with licenses that are not allowed for redistribution from eclipse.org. Cloudsmith Inc. provides these bundles from a special repository, and you can also get these bundles directly from the the respective publishers.

Depending on which combination of Eclipse plugins and protocols you select, and which platform you are running on, the instructions are quite different. On Windows it is particularly complicated to set up access over svn+ssh with use of certificates as windows does not have any support for this out of the box (whereas Un*x systems do).

There are currently two connectors for SVN — and you have to make a choice between *Subversion* and *Subclipse*.

**Warning**

Do NOT install support for both Subversive and Subclipse in the same environment!

Appendix B. Eclipse

This chapter contains a brief overview of selected Eclipse technology and how it relates to b3's domain of composing component based systems.

An overview of Eclipse concepts such as the workspace, target platform, component types such as plugins and features, is also found in this chapter.

Eclipse technology

A selection of Eclipse Technology explained.

Equinox

Equinox is the name of the OSGi runtime underlying the Eclipse IDE. It is a general purpose OSGi runtime. Equinox is (among many things) responsible for the loading (and unloading) of components. It functions as the container for the rest of the system.

For more technical information about OSGi — see <http://www.osgi.org/About/Technology>.

Platform

The Eclipse Platform provides the core frameworks and services upon which all plug-in extensions are created. It also provides the Equinox runtime in which plug-ins are loaded, integrated, and executed. The primary purpose of the Platform is to enable other tool developers to easily build and deliver integrated tools.

Java Development Tools (JDT)

Java Development Tools (JDT) is the set of tools build on top of the Eclipse platform for developing in the Java programming language. It includes a rich set of functionality for editing, compiling, debugging and running java code.

When used alone, created projects are “plain java” and management of dependencies is handled in a manual fashion and with this comes all the classic java issues with specifying a class path, and making sure all the required parts are available when running the code.

You will find more information about using b3 with “plain java” in *TBD LINK TO BUILDING PLAIN JAVA*.

Plugin Development Environment (PDE)

The Plugin Development Environment (PDE) is a set of tools built on top of the Eclipse platform and JDT for developing Eclipse Plugins as well as more general OSGi bundles. PDE has a rich set of functionality to work interactively with the additional meta data found in plugins and bundles and supports all required operation from construction to publication.

The relationships between Eclipse plugins, features, and OSGi bundles is further addressed in [the section called “The Eclipse component types”](#).

PDE also includes PDE-build, which consists of generation of ANT scripts that are then used to build software headlessly.

b3 provides a much more convenient way of invoking the various build actions in PDE than the script based PDE-build, as b3 does not generate scripts.

Rich Client Platform (RCP)

The Rich Client Platform (RCP), is the name for the Eclipse technology that makes it possible to write general purpose applications based on the Eclipse platform. The term “RCP application” is often used to denote the top level product such as the Eclipse IDE. Two well known open source applications built on RCP are the bittorrent client Vuze (Azureus), and the RSS reader RSS Owl. There are also many smaller RCP application in the Eclipse family, such as the p2 and Buckminster installers, the p2 agent, i.e., small independently packaged utilities with a user interface.

Many companies build their internal applications using Eclipse RCP.

Eclipse b3 provides support for building complete RCP-products with a minimum of effort.

p2

Equinox p2 is the relatively new provisioning platform (introduced in Eclipse 3.4 *Ganymede*), designed to be a platform for many different kinds of provisioning solutions, and specifically designed to be a replacement for the Eclipse Update Manager. In Eclipse 3.5 *Galileo* p2 is both functionally rich and well tested with over a year of use, and with close to 2000 unit tests having been constructed. In 3.4 it existed in parallel with the Update Manager, and in 3.5, p2 has replaced it completely.

As p2 is heavily used by b3, and p2 also defines the format of the typical end result (an installable system, or plugins to such a system) we have included a somewhat longer description in [Appendix C, p2](#) as we believe this technology may be new to most users.

The Eclipse component types

The Eclipse system contains several types of “components”; OSGi bundles, plugins, features, fragments, and products. In this section we present an overview of what they are, and the role they play in the composition of a software system built on Eclipse.

Plugins, features and OSGi bundles

The terms “plugin”, “feature”, and “bundle” (short for OSGi bundle) refers to Java components containing meta data information that makes it possible to manage their life cycle. The terms “plugin” and “feature” are specific to the Eclipse platform, and “bundle” is the generic software component handled by an OSGi runtime. Since Eclipse is built on the Equinox OSGi runtime, it can make use of all three types; bundles, plugins, and features.

Bundle

A *bundle* is the fundamental type. In addition to being the container for the code it has meta data describing its dependencies on other bundles, and requirements on packages expected to be present when using the bundle.

Plugin

A *plugin*, is also an OSGi bundle. What makes it special is that it also can contain information that makes use of the Eclipse extension mechanism — a declarative way to define that a bundle contains code that extends functionality in some other bundle.

Feature

A *feature* is a grouping of plugins and other features. It defines a unit of what should be installed together. The feature is a configuration — a bundle may specify that it requires that a certain java package must be present, but the bundle says nothing about where this package should come from. This

can be specified in the feature. This separation allows a bundle to be used in different configurations without requiring that the bundle itself needs to be changed.

Fragments

A *fragment* is a special kind of bundle with what could be called a “reverse dependency” on a host bundle. Fragments are typically used to implement optional code that is included in a configuration, often filtered on parameters like installed language, operating system, hardware architecture and user interface technology. As an example, a fragment could contain code that is only needed during testing or debugging, contain features available only on a particular platform, or for a particular language.

A fragment can also have normal dependencies — these come in effect if the fragment is selected for inclusion.

Fragments are included in a configuration by requiring them in a feature.

Products

A *product* is a special grouping mechanism used to define a “top level” product (such as the Eclipse IDE itself). Unfortunately, the tools that help maintain the group aspect of the product definition are somewhat lacking (in comparison to the same functionality for features), and we recommend that the product definition is used only to define the product aspect, and that all grouping is defined in a single feature that is referenced by the product. An examples of how to do this is found in ???.

When a product definition also acts as a grouping mechanism, it is referred to as a “bundle based product”, and when it refers to feature(s) (we recommend using only one) it is said to be “feature based”.

In addition to referring to the feature(s) or bundles being the configuration for the content of the product, the product also has a reference to a “branding bundle” that contains items such as the splash screen and icon for the product.

The Workspace

The Eclipse Workspace contains projects. These projects can be specialized i.e., plugin project, feature project etc. When you are looking at content in the Eclipse Navigator, or Package Explorer you are looking at content in projects.

You can get content into the workspace by:

- creating new projects and importing files manually
- importing a complete projects from somewhere on disk
- importing one or several projects from a source code repository
- linking to content in the correct format somewhere outside of the workspace
- importing from a “team project set” file, which contains a list of projects to check out from a source code repository.
- importing from source bundles (this is primarily used for debugging and patching).

As you can see, there is only one option that is suitable for automation — using the team project set. Many set up their projects to include such a file in a “meta project” and users begin by checking out this project and then importing using the team project set.

The pitfalls is that the team project set must be maintained manually. As dependencies are added or removed, the set of files required in the workspace may differ, and there is no way to control loading some projects from a branch or a tag.

Solving this particular problem was actually one of the very first requirements for Eclipse Buckminster, the precursor to b3 — as you will see later, b3 provides convenient population of the workspace for the typical case, and it is quite easy to load particular parts from branches and tags.

The Target Platform

The *target platform* is a definition of the set of features/plugins to use when running the code being built. You can say that the code is built for a particular target platform. By default, the target platform is defined to be the same as the Eclipse IDE — this means that when you are running your code in the self hosted environment you will not encounter missing bundles. When however you export and run the code separately, you will almost certain be hit by surprises.

Prior to Eclipse 3.5 there was no good way of managing a target platform in the IDE. A target platform was simply an Eclipse configuration in a directory.

In 3.5 the functionality to handle management of the target platform has been added. Multiple *Target Definitions* can be created. A definition can be saved to file (for later loading). It is also possible to make one definition be the active target platform. The new *Target Definition* defines a set of locations. Each location can be one of:

<i>Directory</i>	A directory in the local file system.
<i>Installation</i>	An installation (such as an Eclipse SDK) in the local file system.
<i>Features</i>	One or more features from an installation.
<i>Software Site</i>	Downloads plug-ins from a p2 repository.

The preferred way of handling target platforms in 3.5 is to create one (or several) with the IDE and then save the definition to a file. Eclipse b3 can use such definitions, and you can also materialize a target platform using b3.

Launch configuration

A launch configuration is a definition of how to launch/run/debug something from within Eclipse. There are multiple classes of launchers for Eclipse covering running plugins, OSGi frameworks, tests, etc. Launch configurations can also launch servers or just run external commands.

Don't confuse launch configuration with target platform. The (eclipse type) launcher launches the active target platform definition with the configuration specified in the launch configuration. This makes it possible to switch target platforms, build for that target, and then launch what was built for testing.

Many developers use the Eclipse IDE itself as the target platform, and then define the set of features/plugins to run in the launch configuration. In Eclipse 3.5, where target platform management has been improved, it is better to define an exact target platform and then have a simpler launch configuration that just use what is in the workspace and everything enabled in the target platform. This separation of concerns is valuable as the target platform definitions are reusable across many products/launchers, and makes it easier to migrate components to newer targets.

ANT

Apache ANT is a Java based build tool that is both well known and widely spread. ANT is integrated with Eclipse, and b3.

The b3 integration consists of:

- Eclipse b3 can call ANT scripts.

- Calling b3 from ANT-scripts.

DRAFT

Appendix C. p2

An introduction to the Eclipse provisioning platform

Equinox p2 is the relatively new provisioning platform introduced in (Eclipse 3.4 *Ganymede*) designed to be a platform for many different kinds of provisioning solutions, and specifically designed to be a replacement for the Eclipse Update Manager. In Eclipse 3.5 *Galileo* p2 is both functionally rich and well tested and it has now replaced Update Manager completely.

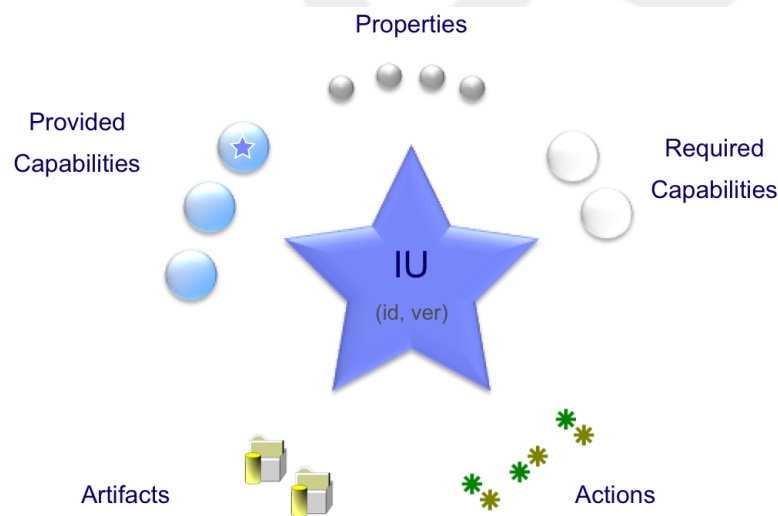
Equinox p2 is still new technology, and does not yet (in Eclipse 3.5) have an official API, and much work remains in utilizing its full potential. Eclipse b3 uses p2 extensively, and as it also steps outside of the OSGi domain (which is the primary focus in p2 and PDE), and adds new use cases to the “p2 as a replacement of Update Manager” there is close cooperation between the teams behind Eclipse b3 and p2. Specifically this has resulted in the Omni Version implementation capable of handling non OSGi versions, and the recent p2 Query Language which enables efficient communication with remote repositories as well as defining more complex dependencies. The Omni Version is covered in [Appendix D, Omni Version Details](#). The p2 Query Language is covered in [TBD REFERENCE TO p2QL](#).

This chapter is only a brief introduction to p2 meant to establish the key concepts.

The Installable Unit

The central concept in p2 is the Installable Unit (IU). It is an entity named in a name-space having a version e.g., the `org.eclipse.equinox.bundle` named `org.myorg.helloworld` having version `1.0.3`.

Figure C.1. Anatomy of an IU



Dependencies are handled by declaring *required capabilities* which are matched with *provided capabilities* also declared in a IU. Specifically, all IUs have a declaration that they provide themselves as a capability. This makes it possible for one IU to require another. The dependency mechanism is very flexible as it allows addition of new capability types. Capability types for Eclipse related types (i.e., plugins, bundles, features, java packages, etc.) have already been defined and are used by p2.

An IU’s artifacts — i.e., the content the IU is describing, is referenced via name and type, and when the artifacts are needed, they are looked up in a p2 *artifact repository*.

The IU also contains *touchpoint instruction*; actions that are invoked in specified phases of a provisioning job e.g., when installing or uninstalling. The instructions can be things like copying files, unzipping an archive, changing startup parameters etc.

If an IU requires special installation instructions these must naturally be installed before an attempt is made to install the IU itself. A mechanism called *meta requirements* allows an IU to declare these, and can then trust p2 to handle resolution and installation of these when an installation of the IU itself is requested.

Metadata repository

The meta data describing components — i.e. the IUs, are stored in a p2 meta data repository. Technically, a meta data repository is an interface and there are several implementations delivered with p2.

- A simple meta data repository stored in a file system directory
- A composite meta data repository that references other meta data repositories
- An Update Site based repository (i.e., the structure used by the older Eclipse Update Manager)
- Specialized repositories that enable the current installation (among other things) to be used as the meta data repository.

Artifact repository

An artifact repository contains the contents of IUs such as files, zip archives, jar files, etc. Technically an artifact repository is an interface and there are several implementations delivered with p2. The available repository implementations are similar to the meta data repositories (i.e., simple, composite, update manager based, and special).

There are many advanced options such as controlling how artifacts are physically stored and sent over the wire; verbatim, packed, or as a delta.

Combined / co-located repositories

Although p2 is capable of handling that meta data and artifact repositories are stored in completely different locations (anywhere addressable by a URI), the most common set up (and the only one supported from the Eclipse SDK's user interface) is a combined (or co-located) repository where a meta data repository and an artifact repository is addressed via a single URI.

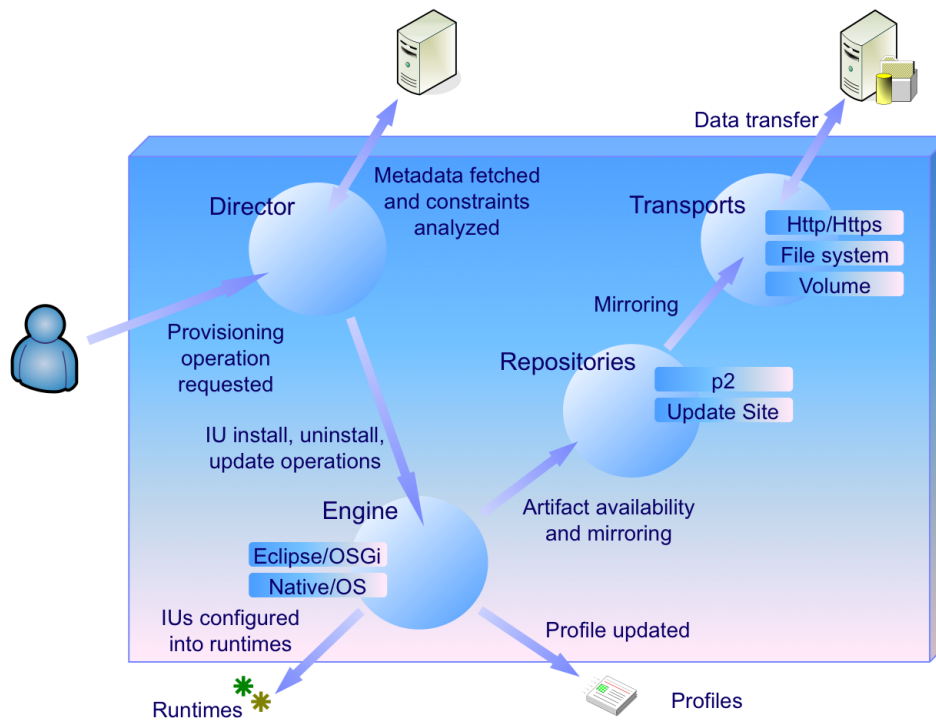
Profile

The p2 *profile* is a central concept — an installation of a product is described by a profile. It contains the meta data for everything that is currently installed. Thus, installation always takes place into a p2 profile.

A profile maintains a history, and it is possible to roll back to a previous configuration. As you may guess, a profile can also function as a repository, making it possible to “copy” parts of an installation from one profile to another.

p2 internals

Internally, the provisioning work is divided up between p2's major parts. The *director* handles provisioning requests such as installing or uninstalling one or several IUs. The director performs the work by using the meta data available in a profile, combined with the meta data in associated meta data repositories (those that have been used to install components from, or repository references just about to be added to the profile). This information is then fed to the *planner* which is responsible for resolving all requirements (dependencies). The resulting plan is fed to p2's *engine* which executes the work in *phases* (in simple terms — it collects items, downloads/mirrors artifacts, installs, and then configures them).

Figure C.2. p2 in action

The planner uses SAT4J to handle the complicated NP-complete problem of resolving requirements. It is interesting to note that there is a guarantee that if there is a solution, it will be found, and it will be an optimal solution (i.e., optimal in the sense of a defined set of weights such as “later version is better”). The use of SAT4J is a major leap forward compared to the old Update Manager (may it rest in peace).

Categories

From an end user perspective, an important part of p2 is the handling of categories. They are used to group related features and arrange them in a structure that makes sense for a human installing software. The features (although one level up above the (to a user) almost incomprehensible very technical plugin names) are still often quite technical in naming, and it can be very difficult for a user to understand the purpose of a particular feature. You have probably already seen the use of categories, as they enable you to browse the content under labels like “java development”, and “modeling” as opposed to just seeing a long alphabetical list with project names.

Prior to Eclipse 3.5, categories were authored in the Update Manager’s `site.xml` file stored in an update site. Such category information is read by p2 when it reads an older update site. When producing new p2 repositories however, the category information needs to be authored differently. In Eclipse 3.5 there are three ways; use the new *Category Editor* which creates a file that PDE makes use of when exporting to a p2 repository, use the (provisional) *p2 publishing advice* which is stored in a `p2.inf` file in the component being published, or use Buckminster or b3 which supports definition of categories in build properties.

Publishing

The act of making components available for consumption by p2 is referred to as “publishing”. It is an area that overlaps with three key technologies; p2, PDE, and the Eclipse platform, and if you look under the covers, you will see that they work in close cooperation.

PDE understands the source components, the meta data that makes the java projects be plugins, features or products. These are translated into the p2 form (IUs), containing information and instructions that makes it possible to install them and control the startup of the equinox environment.

Publishing components as p2 repositories does not require any additional authoring of p2 specific artifacts. More specifically, you do not need to author the IUs — this is done by the PDE specific publisher.

Prior to Eclipse 3.5, publishing was done by first producing an update site, and then generating the p2 combined repository from the output. This is basically what p2 does when it encounters an old style Update Manager site — “publishing” if you like, the update site on the fly. Although this interpretation of update sites is still supported, the recommended way of publishing is via the p2 publisher as it has more information available. (As you will see later, b3 provides a very convenient mechanism to execute p2 publishing of both PDE based projects, but also for those that are not based on Eclipse technology).

Installing

Installing from p2 repositories (or update sites adapted by p2) can be done by a user of the Eclipse SDK directly in the SDK’s “install new software” dialog. With update manager, this was the only (managed) choice — most experienced users simply dropped the required files into the Eclipse installation folder structure (and this worked most of the time). Now, with p2, an install is fully managed to ensure that all requirements are met and that needed actions such as setting startup levels, and modifying initialization parameters take place during installation. This ensures that things actually have a chance of working, be updated, and eventually uninstalled.

With p2, the options are many, especially since p2 does not require that the system being installed into is active when performing the install — it can be done by an external p2 “agent” (there is a utility application called the “p2 agent” which is one example of such an agent. The “p2 installer” is another such example, and the SDK itself also has such an agent).



Note

The various agents all share the same p2 code — the difference is that they are designed to be used in different situations, and thus they expose only information required to support the particular task they were design to handle.

Since users have become accustomed to “dropping in” things that should be installed, this is also supported in p2, but the plugins and features are now dropped in a special folder that is monitored by p2. When it encounters new material in this folder, p2 will perform the same type of managed installation as when installing from repositories. There are several caveats when using drop-ins to install, and it is not the recommended approach as the higher quality meta data provided by publishing is unavailable.

The SDK agent

The p2 SDK agent manages installations into the SDK when used from the user interface. But the functionality of this agent can also be accessed from the command line to perform installation as an external agent. This is referred to as “running the embedded director app”.

Users of Eclipse will typically not use this embedded agent, and instead perform installation work via the user interface. The user interface and backing functionality can also be used in RCP applications, and there are many configuration option available to cater to different installation and update policies (on demand, automatically on startup, completely hidden from the user, update only (no new install, no uninstall), lock down of used repositories, etc).

The SDK agent allows the user to add and remove repositories (under *Eclipse* → *Preferences*), or directly in the “install new software” dialog. The user can see what is installed, select new features to install from selected repositories, perform the installation, and much more.

Since the SDK agent is designed to install into the running SDK itself, many of the advanced features, such as installing into an arbitrary profile, control advanced repository layout through the use of bundle

pooling and shared installs are not present in the user interface. One of the other agents should be used for this purpose.

The director application

The director application is part of every Eclipse SDK and can be invoked from the command line. The director app is also packaged as a separate headless product with a reduced footprint. The headless director application is maintained by the Buckminster project. (See [Appendix A, Installation](#), for how to obtain it). *IS THIS TRUE - OR IS IT NOW PART OF P2?*

The director application makes it possible to control the more advanced features in p2, while still having convenient command line options available for the most common operations.

We will shown examples where the headless separate director application is used and how to get it is explained in [the section called “Installing the Headless Product”](#).

The p2 Installer

The *p2 installer* should be seen as an exemplary implementation of an installer, its user interface is quite unsophisticated, and it lacks many production grade qualities such as detailed progress information, and error reporting. That said, it is still a very useful utility when a user interface based installer is wanted.

The p2 installer in its default configuration is designed to install the Eclipse SDK. It is pre-configured with all the parameters, and when invoked after downloading, all that is required by the user is to tell the installer where it should install the SDK.

It is however possible to feed the p2 installer a different set of parameters by providing a properties file with the information regarding what to install from where, and then modifying the startup of the installer to override the built in default. This requires far less effort than creating a custom installer and may be sufficient for many smaller applications.

The p2 installer is used in one of the examples to install a RCP application — see [???](#).

The EPP wizard

Finally, the Eclipse Packaging Project (EPP) has written an application called the EPP-wizard, a RCP application with a RAP user interface which is driven by meta data to allow a user to select between high level EPP packages such as “Eclipse classic”, or “Web development”, and then add support for optional technologies.

At the end of the process, the EPP-wizard provides the user with a configured p2 installer, that when downloaded and invoked will install exactly what the user picked from the available options.

The Buckminster installer

The Buckminster project also provides an experimental installer. It is designed to be started via Java Web Start or via a Java applet and it gets its initial parameters indirectly via a URL. Originally this installer used Buckminster’s provisioning capabilities and before p2 this was one of very few options available when an external, web startable installer was wanted.

The Buckminster installer also includes a JSON client, and is capable of engaging in a dialog with an smart repository and thereby present more information about what is being installed, manage a sign-in dialog, branding, and much more.

The Buckminster installer is however not yet considered released — its API may need further changes to be suited for general use, and testing is limited. Using this installer requires setting up the server side correctly and this part is not included in the installer, and no documentation is provided.

Shipping

By *shipping* we mean making the published material available to the intended consumers. You may think of this as “publishing” (i.e., making something publicly available), but this term is already used to mean making the internal meta data found inside projects public to the outside world in the form of p2 repositories.

In fact, there is no support in Eclipse to handle the steps required once such publishing has taken place. The resulting folder structure with files in them are simply written to disk, and there everything ends.

The most common way of shipping is making the published result available on a web site. And in cases when what is shipped is supposed to be installed into the Eclipse SDK, or consists of plugins for some other RCP application, this is as simple as just copying the result written to disk by the publisher to the appropriate directory where a web server picks it up.

If creating a complete application however there are more to consider. Users will typically not have the application installed to begin with, so user must start by downloading something. As seen in [the section called “Installing”](#) there are several installers available that can serve as a starting point — from the headless director application, to the interactive Buckminster installer. The benefit of using these is that there is no need to ship the complete application pre-configured for different platforms — as this is handled by the installer. Unfortunately, as the various installers were all created for a specific purpose, and some being more “exemplary”, you may find that they may not suffice if you are going to ship a more high profile application, and you may want to write your own installer.

Your options for shipping includes:

- Pre-configured installations per platform. To do this, you typically run the headless director app (or use b3 to do it) — telling it to install for one particular configuration (operating system, window system, architecture, language, etc.) into a location on disk. The result is then zipped-and-shipped.
- An installer configured to install the application from a remote repository. This has advantages as the initial download is small, and the bulk of the installation is performed by p2 which supports parallel downloading, selection of mirrors, and compressed artifacts. It is also very simple to add download of newer versions as everything is stored in a central repository.
- Zipping up a p2 repository with everything and a configured installer. The benefit is that the user will download everything that is needed to local disk, and can perform the install while not being connected to the Internet. The downside is that the repository contains components that are never used on the platform where it is installed.

This form is suitable if you are shipping on a CD/DVD.

- Delivering application via a Linux package manager such as RPM creating a read only and shared installation that is then extended via an embedded p2 agent.
- Hybrid form, where the basic application is downloaded using one of the above mechanisms, but where bulky extras are installed via a p2 agent embedded in your application (like the Eclipse SDK p2 agent), or via an external installer.

In addition to deciding on how to ship — you must also decide on how you want to compose the required repositories. Your options include:

- Creating a composite repository with a reference to the main Eclipse repository for everything that is used from the Eclipse platform. This has the advantage that “your site” is always up to date with the latest repository content, and you do not have to store copies of everything in your repository.
- Creating an aggregated meta data repository that contains the meta data from the Eclipse main repository as well as your site(s), but uses the existing artifact repositories via a composite artifact repository. This has the advantage over the simplest form in that all of the meta data is obtained

in a single download, and since you are reconstructing the meta data, you also have more control over the categorization of features.

- Mirror everything you need to your repository and then deliver everything from your servers. The benefit is that you have full control, but you do not make use of the Eclipse mirrors, and you must periodically update your mirrors.

Eclipse b3 has support for aggregating sites — this functionality has been used in the Eclipse 3.5 Galileo release to compose the final Galileo repository. See [TBD - LINK TO AGGREGATOR CHAPTER](#).

Summary

Equinox p2, is a provisioning *platform* and as such has a rich and flexible feature set. Being rich and flexible also means that it is complex. It is complex in itself as it is solving a very difficult problem, and it is doing so with OSGi technology that under the covers need to perform complex tasks so developers can focus on the functionality instead of the mechanics of configuring a dynamic system — all in order to provide consumers of the resulting software with a high quality software provisioning experience — simply click install, and run automatic updates.

In the following chapters we will show how b3, p2 and PDE work together, and how you can use b3 to handle some of the complexities.

Appendix D. Omni Version Details

Introduction

This appendix describes the *Omni Version* implementation handling instances of version and version ranges. The omni version implementation resides in equinox p2, and is also used in b3. The omni version was created because of the need to have a version format capable of describing versions using another versioning scheme than OSGi (which was the only versioning scheme supported by p2 prior to Eclipse 3.5 and omni version).

With the omni version contribution to p2 — which fully describes a format, a canonical version comparable against versions with different formats, as well as containing the original version string, it is possible to use p2 for provisioning also for non OSGi based components.

Background

There are other versioning schemes in wide use that are not compatible with OSGi version and version ranges. The problem is both syntactic and semantic.

Many open source projects do their versioning in a fashion similar to OSGi but with one very significant difference. For two versions that are otherwise equal, a lack of qualifier signifies a higher version than when a qualifier is present — i.e.,

```
1.0.0.alpha
1.0.0.beta
1.0.0.rc1
1.0.0
```

The 1.0.0 is the final release. The qualifier happens to be in alphabetical order here but that's not always true.

Mozilla Toolkit versioning has many rules and each segment has 4 (optional) slots; *numeric*, *string*, *numeric*, and *string* where each slot has a default value set to 0 or *max string* respectively for the numeric and string slots if a particular slot is missing).

```
1.2a3b. // yes, a trailing . is allowed and means .0
1.a2
```

Mozilla also allows bumping the version (using an older Mozilla scheme)

```
1.0+
```

This means 1.1pre in Mozilla.

Example of syntax issue

Here are some examples of versions used in Red Hat Fedora distributions.

```
KDE Admin version 7:4.0.3-3.fc9
Compat libstdc version 33-3.2.3-63
Automake 1.4p6-15.fc7
```

And here are some Mozilla toolkit versions:

```
1.*.1
1.0+
1.-1 // negative integer version numbers are allowed, the '-' is not a delimiter
1.2a3b.a
```

These are not syntactically compatible with OSGi versions.

Implementation

The current implementation in p2 uses the omni versions throughout. This means that p2 can create a plan including units that have non OSGi versioning scheme.

One implementation

Equinox p2 has one implementation of `Version` and one of `VersionRange` that are capable of capturing the semantics of various version formats. The advantages are that there is no need to dynamically plugin new implementations, and new formats can be easily be introduced.

One canonical format

The omni version and omni version range are “universal” — all instances of version should be comparable against each other with a fully defined (non ambiguous) ordering. The API is (as today) based on a single string fully describing a version or version range.

The canonical string format is called “raw” and it is explained in more detail below. To ensure backward compatibility, as well as providing ease of use in an OSGi environment, version strings that are not prefixed with the omni version keyword `raw` have the same format and semantics as the current OSGi version format.

As an example the following two version strings are both valid input, and express exactly the same version:

```
1.0.0.r1234
raw:1.0.0.'r1234'
```

Version

The omni version implementation uses an vector to store version-segments in order of descending significance. A segment is an instance of `Integer`, `String`, `Comparable[]`, `MaxInteger`, `MaxString`, or `Min`.

Comparison

Comparison is done by iterating over segments from 0 to n.

- If segments are of different type the rule `MaxInteger > Integer > Comparable[] > MaxString > String` is used — the comparison is done and the version with the greater segment type is reported as greater.
- If segments are of equal type — they are compared — if one is greater the comparison is done and the version with the greater segment is reported as greater.
- All versions are by default padded with `-M` (absolute min segment) to “infinity”. A version may have an explicit pad element which is used instead of the default.
- A shorter version is compared to a longer by comparing the extra segments in the longer version against the shorter version’s pad segment.
- If all segments are equal up to end of the longest segment array, the pad segments are compared, and the version with the greater pad segment is reported as greater.
- If pad segments are also equal the two versions are reported as equal.
- As a consequence of not including delimiters in the canonical format; two versions are equal if they only differ on delimiters.

As an example — here is a comparison of versions (expressed in the raw format introduced further on in the text — ‘p’ means that a pad element follows, and ‘-M’ the absolute min segment):

```
1p-M < 1.0.0 < 1.0.0p0 == 1p0 < 1.1 < 1.1.1 < 1p1 == 1.1p1 < 1pM
```

Raw and Original Version String

The original version can be kept when the raw version format is used, but it is not an absolute requirement as simple raw based forms such as `raw:1.2.3.4.5` could certainly be used directly by humans. Someone (who for some reason does not want to use OSGi or some other known version scheme), could elect to use the raw format as their native format.

A version string with raw and original is written on the form:

```
'raw' ':' raw-format-string '/' format(...):original-format-string
```

The p2 Engine completely ignores the original part — only the raw part is used, and the original format is only used for human consumption.

Example using a Mozilla version string (as it has the most complex format encountered to date)¹.

```
raw:<1.m.0.m>.<20.'a'.3.'b'>p<0.m.0.m>
/format((<n=0;?s=m;?n=0;?s=m;?>(<n=0;?s=m;?n=0;?s=m;?>)*)=p<0.m.0.m>;)
:1.20a3b.a
```

An original version string can be included with unknown format:

```
raw:<1.m.0.m>.<20.'a'.3.'b'>p<0.m.0.m>/:1.20a3b.a
```

See below for full explanation of the raw format.

Omni Version Range

The version range holds two version instances (lower and upper bound). A version range string uses the delimiters `[]`, `()` and `.`. If these characters are used in the lower or upper bound version strings, these occurrences must be escaped with `\` and occurrences of `\` must also be escaped.

The version range is either an OSGi version range (if raw prefix is not used), or a raw range. The format of the raw range is:

```
'raw' ':' ( '[' | '(' ) raw-format-string ',' raw-format-string ( ']' | ')' )
```

The raw-range can be followed by the original range:

```
raw-range '/' 'format' '(' format-string ')'
':' ( '[' | '(' ) original-format-string ','
original-format-string ( ']' | ')' )
```

An original version range can be included with unknown format:

```
raw: [ <1.m.0.m>.<20.m.0.m>p<0.m.0.m> ,
<1.m.0.m>.<20.'a'.3.'b'>p<0.m.0.m> ]
/: [1.20,1.20a3b.a]
```

The p2 Engine completely ignores the original part — only the raw part is used, and the original format is only used for human consumption.

See below for full explanation of the raw format.

Other range formats

Note that some version schemes have range concepts where the notion of inclusive or exclusive does not exist, and instead use symbolic markers such as “next larger“, “next smaller“, or use wild-cards to define ranges. In these cases, the translator of the original version string must use discrete versions and the inclusive/exclusive notation to define the same range.

Some range specifications allows the specification of union, or exclusion of certain versions. This is *not* currently supported by p2. If introduced it could be expressed as a series of ranges where

¹line breaks are inserted for readability

a ^ before a range negates it. Example `[0,1][3,10]^(3.1,3.7)` which would be equivalent to `[0,10]^(1,3)^[3.1,3.7]`

Format Specification

There are two basic formats *default OSGi string format*, and *raw canonical string format*. There are also two corresponding range formats *OSGi-version-range*, and *raw-version-range*.

The raw format is a string representation of the internally used format — it consists of the keyword “raw“, followed by a list of entries separated by period. An entry can be numerical, quoted alphanumerical, or a sub canonical list on the same format. A canonical version (and sub canonical version arrays) can be padded to infinity with a special padding element. Special entries express the notion of ‘max integer’ and ‘max string’.

The OSGi string format is the well known format in current use.

The raw format in BNF:

```

digit: [0-9];
letter: [a-zA-Z];
numeric: digit+;
alpha: letter+;
alpha-numeric: [0-9a-zA-Z]+;
delimiter: [^0-9a-zA-Z];
character: .;
characters .+;

// A sequence of charactes quoted with " or ', where ' can
// be used in a " quoted string and vice versa
quoted-string: ("^[^"]*"|'[^']*');

// a sequence of any characters but
// with ',', ' ', ')' and '\' escaped with '\\'
range-safe-string: TBD;

sq: [''];
dq: [""];

version:
| osgi-version
| raw-version
;
osgi-version:
| numeric
| numeric '.' numeric
| numeric '.' numeric '.' numeric
| numeric '.' numeric '.' numeric '.' .+
;
raw-version:
| 'raw' ':' raw-segments optional-original-version
;
optional-original-version:
| '/' original-version
;
version-range:
| osgi-version-range
| raw-version-range
;
rs: ('[' | '(');
re: (']' | ')');

osgi-version-range:
| rs osgi-version ',' osgi-version re
;
raw-version-range:
| 'raw' ':' rs raw-segments ',' raw-segments re
optional-original-range

```

```

        ;
optional-original-range:
    |
    | '/' original-range
    ;
raw-segments:
    | raw-elements optional-pad-element
    ;
raw-elements:
    | raw-elements '.' raw-element
    | raw-element
    ;
raw-element:
    | numeric
    | quoted-strings // strings are concatenated
    | '<' raw-elements optional-pad-element '>'
        // subvector of elements
    | 'm' // symbolic 'maxs' == max string
    | 'M' // symbolic 'absolute max'
        // i.e., max > MAX_INT > maxs
    | '-M // symbolic 'absolute min'
        // i.e., -M < empty string < array < int
    ;
optional-pad-element:
    |
    | pad-element
    ;
quoted-strings:
    | quoted-strings quoted-string
    | quoted-string
    ;
pad-element:
    | 'p' raw-element
    ;
original-version:
    | optional-format-definition ':' .*
    ;
original-range:
    | optional-format-definition ':' rs range-safe-string
    | ',' range-safe-string re
    ;
optional-format-definition:
    |
    | format-definition
    ;
format-definition:
    | 'format' '(' pattern ')'
    ;

// Definition of parsing patterns
//
pattern:
    | pattern pattern-element
    | pattern-element
    ;
pattern-element:
    | pelem optional-processing-rules optional-pattern-range
    | '[' pattern ']' processing-rules
    ;
optional-processing-rules:
    | optional- processing-rules '=' processing-rule ';'
    | '=' processing-rule ';'
    ;
optional-pattern-range:
    | repeat-range
    ;

pelem
    | 'r' | 'd' | 'p' | 'a' | 's' | 'S' | 'n' | 'N' | 'q'
    | '(' pattern ')'

```

```

    | '<' pattern '>'
    | delimiter
    ;
repeat-range:
    | '?' | '*' | '+'
    | '{' exact '}'
    | '{' at-least ',' '}'
    | '{' at-least ',' at-most '}'
    ;

exact: at-least: at-most: numeric;

processing-rule:
    | raw-element
    | pad-element
    | '!'
    | '[' char-list ']'
    | '[' '^' char-list ']'
    | '{' exact '}' // for character count
    | '{' at-least ',' '}'
    | '{' at-least ',' at-most '}'
    ;
char-list: TBD; // Sequence of any character but
              // with '^', ']' and '\' escaped with '\'
delimiter:
    | [!#$%&/=^,.;:_ ] // Any non-alpha-num that
                        // has no special meaning
    | quoted-string
    | '\' . // any escaped character
    ;

```

Examples:

- OSGi 1.0.0.r1234 is expressed as `raw:1.0.0.'r1234'`
- apache/triplet style 1.2.3 is expressed as `raw:1.2.3.m`
- Mozilla style 1a.2a3c. can be expressed as

```
raw:<1.'a'.0.m>.<2.'a'.3.'c'>p<0.m.0.m>
```

Mozilla's format is complex — see external links at the end of this appendix, for more information.

Format Pattern Explanation

Here are explanations for the rules in `format(pattern)`.

rule	description
<code>r</code>	<i>raw</i> — matches one <i>raw-element</i> as specified by the <code>raw</code> format. The <code>r</code> rule does not match a pad element — use <code>p</code> for this.
<code>'characters'</code>	<i>quoted delimiter</i> — matches one or several characters — the matched result is not included in the resulting canonical vector (i.e., it is not a segment). A <code>\</code> is needed to include a single <code>\</code> . The sequence of chars acts as one delimiter.
<i>non-alphanumeric character</i>	<i>literal delimiter</i> — matches any non alpha-numerical character (including space) — the matched result is not included in the canonical vector (i.e., it is not a segment). A non alphanumeric character acts as a delimiter. Special characters must be escaped when wanted as delimiters.
<code>a</code>	<i>auto</i> — a sequence of digits creates a numeric segment, a sequence of alphabetical characters creates a string segment. Segments are delimited by any character not having the same character class as the first character in the sequence, or by the following delimiter. A numerical sequence ignores leading zeros.
<code>d</code>	<i>delimiter</i> — matches any non alpha-numeric character. The matched result is not included in the resulting canonical vector (i.e., it is not a segment).

rule	description
s	<i>letter-string</i> — a string group matching only alpha characters (i.e., “letters”). Use processing rules = [] ; or = [^] to define the set of allowed characters. It is possible to allow inclusion of delimiter chars, but not inclusion of digits.
S	<i>string</i> — a string group matching any group of characters. Use processing rules = [] ; or = [^] to define the set of allowed characters. Care must be taken to specify exclusion of a delimiter if elements are to follow the s.
n	a <i>numeric</i> (integer) group with value ≥ 0 . Leading zeros are ignored.
N	a possibly <i>negative value numeric</i> (integer) group. Leading zeros are ignored.
p	parses an explicit <i>pad-element</i> in the input string as defined by the raw format. To define an implicit pad as part of the pattern use the processing instruction =p . . . ; . A pad element can only be last in the overall version string, or last in a sub array.
q	<i>smart quoted string</i> — matches a quoted alphanumeric string where the quote is determined by the first character of the string segment. The quote must be a non alphanumeric character, and the string must be delimited by the same character except brackets and parentheses (i.e., (), { }, [], < >) which are handled as pairs, thus q matches <andrea-doria> and produces a single string segment with the text andrea-doria. A non-quoted sequence of characters are not matched by q.
()	indicates a group
< >	<i>array</i> — indicates a group, where the resulting elements of the group is placed in an array, and the array is one resulting element in the enclosing result
?	<i>zero to one</i> occurrence of the preceding rule
*	<i>zero to many</i> occurrences of the preceding rule
+	<i>one to many</i> occurrences of the preceding rule
{ n }	<i>exactly n</i> occurrences of the preceding rule
{ n , }	<i>at least n</i> occurrences of the preceding rule
{ n , m }	<i>at least n</i> occurrences of the preceding rule, but not more than m times
[]	short hand notation for an <i>optional group</i> . Is equivalent to () ?
= processing ;	<p>an additional <i>processing rule</i> is applied to the preceding rule. The <i>processing</i> part can be:</p> <ul style="list-style-type: none"> • a <i>raw-element</i> - use this <i>raw-element</i> (as defined by the raw format) as the default value if input is missing. The default value does not have to be of the same type (e.g., s=123 ; ? produces an integer segment of value 123 if the optional s is not matched. • ! — if input is present do not turn it into a segment (i.e., ignore what was matched) • [list of chars] — when applied to a d defines the set of delimiters. The characters], ^, and \ must be escaped with \ to be used in the list of chars. and Example d=[+ - /] ; One or several ranges of characters such as a-z can also be used. Example d=[a-zA-Z0-9_-] ; • [^ list of chars] — when applied to a d defines the set of delimiters to be all non alpha numeric except the listed characters. The characters], ^, and \ must be escaped with \ to be used in the list of chars. One or several ranges of characters such as a-z can also be used. Example d=[^ \$] ;

rule	description
	<ul style="list-style-type: none"> <code>praw-element</code> — defines “padding to infinity with specified raw-element” when applied to an array, or a group enclosing the entire format. Example <code>format((n.s)=pM;)</code> The pad processing rule is only applied to a parsed array, not to a default value for an array. If padding is wanted in the default array value, it can be expressed explicitly in the default value. <code>{n}</code> <code>{n,}</code> <code>{n,m}</code> character ranges — with the same meaning as the rules with the same syntax, but limits the range in characters matched in the preceding <code>s</code>, <code>S</code>, <code>n</code>, <code>N</code>, <code>q</code>, or a rules. For <code>q</code> the quotes does not count.
<code>\</code>	<code>escape</code> removes the special meaning of a character and must be used if a special character is wanted as a delimiter. A <code>\\</code> is needed to include a single <code>\</code> . Escaping a non special character is superfluous but allowed.

Additional rules:

- if a rule produces a null segment, it is not placed in the result vector

e.g., `format(ndddn):10-/-12` → `raw:10.12`

- Processing (i.e., default values) applied to a group has higher precedence than individual processing inside the group if the entire group was not successfully matched.
- Parsing is greedy — `format(n(.n)*(.s)*)` will interpret `1.2.3.hello` as `raw:1.2.3.'hello'` (as opposed to being reluctant which would produce `raw:1.'2'.'3'.'hello'`)
- When combining `N` with `={...}`; and the input has a negative number, the ‘-’ character is not included in the count — `format(N{3}N{2}): -1234` results in `raw:-123.4`
- When combining `n` or `N` with `={...}` and input has leading zeros — these are included in the character count.
- An empty version strings is always considered to be an error.
- A format that produces no segments is always considered to be an error.

Note about white space in the raw format:

- white space is accepted inside quoted strings — i.e., `1.'a string'` is allowed, but not `1. 2`
- white space is accepted between version range delimiters and version strings
i.e., `[1.0, 2.0]` is allowed.

Note about timestamps Versions based on a timestamp should use `s` or `n` and ensure comparability by using a fixed number of characters when choosing `s` format.

Examples of Version Formats

Here are examples of various version formats expressed as using the format pattern notation.

type name	pattern	comment
osgi	<code>n[.n=0;[.n=0;[.S=[a-zA-Z0-9_-];]]]</code>	Example: the following are equivalent: <ul style="list-style-type: none"> <code>format(n[.n=0;[.n=0;[.S=[a-zA-Z0-9_-];]]):1.0.0.r1234</code> <code>raw:1.0.0.'r1234'</code> <code>osgi:1.0.0.r1234</code>

type name	pattern	comment
		<ul style="list-style-type: none"> 1.0.0.r1234
triplet	<code>n[.n=0;[.n=0;[.S=m;]]]</code>	<p>A variation on OSGi, with the same syntax, but where the a lack of qualifier > any qualifier, and the qualifier may contain any character. The following are all equivalent:</p> <ul style="list-style-type: none"> <code>format(n[.n=0;[.n=0;[.S=m;]]):1.0.0</code> <code>raw:1.0.0.M</code> <code>triplet:1.0.0</code>
jsr277	<code>n(.n=0;){0,3}[-S=m;]</code>	<p>As defined by JSR 277 — but is provisional and subject to change as it is expected that compatibility with OSGi will be solved (they are now incompatible because of the fourth numeric field with default value 0). The jsr277 format is similar to triplet, but with 4 numeric segments and a '-' separating the qualifier to allow input of "1-qualifier" to mean "1.0.0.0-qualifier". As in triplet the a lack of qualifier > any qualifier. The following are all equivalent:</p> <ul style="list-style-type: none"> <code>format(n(.n=0;){1,3}[-S=m;]):1.0.0</code> <code>raw:1.0.0.0.M</code> <code>jsr277:1.0.0</code>
tripletSnapshot	<code>n[.n=0;[.n=0;[-n=M;,.S=m;]]]</code>	<p>Format used when maven transforms versions like 1.2.3-SNAPSHOT into 1.2.3-<buildnumber>.<timestamp> ensuring that it is compatible with triplet format if missing <buildnumber>.<timestamp> at the end (format produces max, max-string if they are missing).</p> <p>Example: the following are equivalent:</p> <ul style="list-style-type: none"> <code>format(n[.n=0;[.n=0;[-n=M;,.S=m;]]):1.2.3-45.20081213:1233</code> <code>raw:1.2.3.45.'20081213:1233'</code> <code>tripletSnapshot:1.2.3-45.20081213:1233</code>
rpm	<code><[n:]a(d?a)*>[-n[dS=!;]]</code>	<p>RPM format matches [EPOCH:]VERSION-STRING[-PACKAGE-VERSION], where epoch is optional and numeric, version-string is auto matched to arbitrary depth >= 1, followed by a package-version, which consists of a build number separated</p>

type name	pattern	comment
		<p>by any separator from trailing platform specification, or the string 'src' to indicate that the package is a source package. This format allows the platform and src part to be included in the version string, but if present it is not used in the comparisons. The platform type vs source is expected to be encoded elsewhere in such an IU. Everything except the build-number is placed in an array as build number is only compared if there is a tie.</p> <p>An example of equivalent expressions:</p> <ul style="list-style-type: none"> format(<[n:]a(d?a)*>[-n[dS=!;]]):33:1.2.3a-23/i386 raw:<33.1.2.3.'a'>.23
mozilla	$(\langle n=0;?s=m;?n=0;?s=m;? \rangle (\langle n=0;?s=m;?n=0;?s=m;? \rangle)^*) = p \langle 0.m.0.m \rangle ;$	Mozilla versions are somewhat complicated, it consists of 1 or more parts separated by period. Each part consists of 4 optional 'fragments' (numeric, string, numeric, string), where numeric fragments are 0 if missing, and string fragments are MAX-STRING if missing. The versions use padding so that $1 == 1.0 == 1.0.0 == 1.0.0.0$ etc.
string	S	a single string
auto	$a(d?a)^*$	serves like a "catch all".

Tooling Support

The omni version implementation is not designed to be extended. An earlier idea was that it should be possible to define named aliases for common formats and that these formats should be parseable by the omni version parser. The reasons for introducing alias was to make it possible for users to enter something like `triplet:1.0.0` instead of entering the more complicated format. This did however raise a lot of questions: Who can define an alias, what if the definition of the alias is changed, where are the alias definitions found. Is it possible to work at all with a version that is using only an alias — what if I want to modify a range and do not have access to the alias?

Instead, the alias handling is a tooling concern. Tooling should keep a registry of known formats. When a version is to be presented, the format string is "reverse looked up" in the registry — and the alias name can be presented instead of the actual format. This way, the version is always self describing. There is still the need to get "well known formats" and make them available in order to make it easier to use non OSGi versions in publishing tools — but there is no absolute requirement to support this in all publishing tools (some may even operate in a domain where version format is implied by the domain) — and there is no "breakage" because an alias is missing.

Tooling support can be as simple as just having preferences where formats are associated with names — the user can enter new formats and aliases. Some import mechanism is probably also nice to have. Further ideas could be that aliases can be published as IU's and installed (i.e install a preference).

Existing Tooling should naturally use the new omni version implementation to parse strings — thus enabling a user to enter a version in raw or format() form. An implementation can choose to present the full version string (i.e., `Version.toString()`), or only the original version.

More examples using ‘format’

A version range with format equivalent to OSGi

```
format(n[.n=0;[.n=0;[.S=[a-zA-Z0-9_-];]])
:[1.0.0.r12345, 2.0.0]
```

At least one string, and max 5 strings

```
format(S=[^.] [.S=[^.] ; [.S=[^.] [.S=[^.] [.S=[^.] ]]]])
:vivaldi.opus.spring.bar5
```

```
format(S=[^.] [.S=[^.] ]{0,4}):vivaldi.opus.spring.bar5
=> 'vivaldi'.opus.spring.bar5'
```

At least one alpha or numerical with auto format and delimiter

```
format(a(d?a)*):vivaldi:opus23-spring.bar5
=> 'vivaldi'.opus.23.spring.bar'.5
```

The texts ‘opus’ and ‘bar’ should not be included:

```
format(s[.opus'n[.bar'n]):vivaldi.opus23.bar8
=> 'vivaldi'.23.8
```

The first string segment should be ignored — it is a marketing name:

```
format(s=!;.n(.n)*):vivaldi.1.5.3
```

Classic SCCS/RCS style:

```
format(n(.n)*):1.1.1.1.1.1.4.5.6.7.8
```

Max depth 8 of numerical segments (limited classic SCCS/RCS type versions):

```
format(n(.n){0,7}):1.1.1.1.1.1.1.4
```

Numeric to optional depth 8, where missing input is set to 0, followed by optional string where ‘empty > any’

```
format(n(d?n=0;){0,7}[a=M;]):1.1.1.4:beta
=> 1.1.1.4.0.0.0.0.'beta'
```

```
format(n(d?n=0;){0,7}[a=M;]):1.1.1.4
=> 1.1.1.4.0.0.0.0.M
```

Single string range

```
format(S):[andrea doria,titanic]
```

Range examples

Examples:

- raw:[1.2.3.'r1234',2.0.0]
- [1.2.3.r1234,2.0.0]
- format(a+):[monkey.fred.ate.5.bananas,monkey.fred.ate.10.oranges]
- [1.0.0,2.0.0] equal to osgi:[1.0.0,2.0.0]
- format(S):[andrea doria,titanic]
- rpm:[7:4.0.3-3.fc9,8:1] - an example KDE Admin version 7:4.0.3-3.fc9 to 8:1
- triplet:[1.0.0.RC1,1.0.0]

FAQ

Is internationalization supported? Alphanumeric segments use vanilla string comparison as internationalization (lexical ordering/collation) would produce different results for different users.

Are users just using Eclipse and OSGi bundles affected? No, users that only deal within the OSGi domain can continue to use version strings like before, there is no need to specify version formats.

How does a user of something know which version type to use? This seems very complicated... To use some non-OSGi component with p2, that component must have been made available in a p2 repository. When it was made available, the publisher must have made it available with a specified version format. The publisher must understand the component's version semantics. A consumer that only wants to install the component does not really need to understand the format, and the original version string is probably sufficient. In scenarios where the consumer needs to know more — what to present is domain specific — some tool could show all non OSGi version strings as “non-OSGi” or “formatted” with drill down into the actual pattern (or if there is an alias registry available, it could reverse lookup the format).

Will open (OSGi) ranges produce lots of false positives? Very unlikely. One decision to minimize the risk was to specify that integer segments are considered to be later than array and string segments. (We also felt that version segments specified with integers are more precise). Note that to be included in the range, the required capability would still need to be in a matching name space, and have a matching name. To introduce a false positive, the publisher of the false positive would need to a) publish something already known to others (namespace and name) b) misinterpret how its versioning scheme works, and publishing it with a format of n.n.n.n (or n.n.n.s.<something>), c) having first learned how to actually specify such a format and how to publish it to a p2 repository and d) then persuaded users to use the repository.

What happens when a capability is available with several versioning schemes? A typical case would be some java package that is versioned at the source using triplet notation, and the same package is also made available using OSGi notation (which btw. is a mistake).

As an example, the following capabilities are found:

- org.demo.ships triplet:2.0.0
- org.demo.ships triplet:2.0.0.RC1
- org.demo.ships osgi:2.0.0
- org.demo.ships osgi:2.0.0.RC1

(Reminder: in triplet notation 2.0.0.RC1 is *older* than 2.0.0).

The raw versions will then look like this:

- 2.0.0.m
- 2.0.0.'RC1'
- 2.0.0
- 2.0.0.'RC1'

And the newest is 2.0.0.m (which is correct for both OSGi, and triplet). When specifying a range, the outcome may depend on if the range is specified with osgi or triplet notation.

- osgi:[1.0.0,2.0.0] == raw:[1.0.0, 2.0.0] => matches the osgi:2.0.0 version only
- triplet:[1.0.0,2.0.0] == raw:[1.0.0.m,2.0.0.m] => matches all the versions, and picks 2.0.0.m as it is the latest.

i.e., result is correct (assuming the bits are identical as different artifacts would be picked)

Now look at the lower boundary, and assume that the following versions are the (only) available:

- org.demo.ships triplet: 1.0.0 == raw: 1.0.0.m
- org.demo.ships triplet: 1.0.0.RC1 == raw:1.0.0.'RC1'
- org.demo.ships osgi: 1.0.0 == raw:1.0.0
- org.demo.ships osgi:1.0.0.RC1 == raw:1.0.0.'RC1'

When specifying ranges:

- osgi:[1.0.0,2.0.0] == raw:[1.0.0, 2.0.0] => matches all the version, and picks 1.0.0.maxs as this is the newest
- triplet:[1.0.0,2.0.0] == raw:[1.0.0.m,2.0.0.m] results in 1.0.0.m as it is the only available version that matches.

i.e., the result is correct and here the exact same version is picked.

The “worst OSGi/triplet crime” that can be committed is publishing an unqualified triplet version as an OSGi version (if the same version is not also available as a triplet) as this would make that version older than what it is even when queried using a triplet range.

What if the publisher of a component changes versioning scheme — what happens to ranges? The order among the versions will be correct as long as the versions are published using the correct notation. The only implication is that users must understand that a query for triplet:1.2.3 means raw:1.2.3.m — e.g., osgi:[1.0.0,2.0.0] != triplet:[1.0.0,2.0.0] (OSGi upper range of 2.0.0 would not match triplet published 2.0.0, and triplet lower range of 1.0.0 would not match OSGi published 1.0.0).

Why not use regexp instead of the special pattern format? This was first considered, and would certainly work if the pattern notation was augmented with processing instructions, or if the regexp is specified as a substitution that produces the raw format. Such specifications would typically be much longer and more difficult for humans to read than the proposed format, except possibly for regexp experts :). Another immediate problem is that regexp breaks the current API requirement. It is not included in execution environment CDC-1.1/Foundation-1.1 required by p2.

Pattern parsing looks like it could have performance implications — what are the expectations here? A mechanism similar to regular expressions is used — when a format is first seen it is compiled to an internal structure. The compiled structure is cached and reused for all subsequent occurrences of the same format. Once parsed, all comparisons are made using the raw vector, which is comparable in speed to the current implementation (in many cases it is faster).

Also note that the Engine does not have to parse and apply the format to the original string unless code explicitly asks for it, and this is not the normal case during provisioning.

Why not just let the publisher deal with transforming the version into canonical form? The proposal allows this — the publisher is not required to make the format available. We think this is reasonable in domains where humans are not involved in the authoring (or the consumption).

There are several reasons why it is a good idea to include the original version string as well as the format:

- the original version strings needs to be kept as users would probably not understand the canonical representation in many cases.
- if the transformation pattern is not available a user would not be able to create a request without hand coding the canonical form

- making the transformation logic used by one publisher available to others would mean that all publishers must have extensions that allow plugging in such logic, and the plugins must be made available

Would it be possible to use the OSGi implementation of version as the canonical form? The long answer is: To be general, the encoding would need to be made in the qualifier string part of the OSGi version. An upper length for segments must be imposed, numerical sections must be left padded with “0” to that length, and string segments must be right padded with space (else string segment parts may overlap integer segments parts). The selected segment length would need to be big enough to allow the longest anticipated string segment. A fixed length string representation of MAX must be invented. A different implementation would still be needed to be able to keep the original version strings. The short answer is: no (and this is the reason for implementing the omni version in the first place).

Why not use an escape in string segments to be able to have strings with a mix of quotes? There are several reasons:

- this would mean that the version string would need to be preprocessed as it would not have \ embedded from the start
- all version strings that use \ as a delimiter would need to be pre-processed to escape the \
- to date, we [...the authors of this proposal] have not seen a version format that requires a mix of quotes
- In the unlikely event that such strings are present it is possible to concatenate several strings in the raw format.
- parsing performance is affected

Which format should I use? If you have the opportunity to select a versioning scheme — stick with OSGi.

Resources

- [mozilla toolkit version format](https://developer.mozilla.org/En/Toolkit_version_format) [https://developer.mozilla.org/En/Toolkit_version_format]
- [rpm version comparison](http://linux.duke.edu/~mstenner/docs/rpm-version-cmp) [http://linux.duke.edu/~mstenner/docs/rpm-version-cmp]
- [sun spec version format](http://java.sun.com/j2se/1.5.0/docs/guide/deployment/deployment-guide/version-format.html) [http://java.sun.com/j2se/1.5.0/docs/guide/deployment/deployment-guide/version-format.html]

Colophon

How to print this book. This book was produced by using the following specifications and tools:

- DocBook 4.5 schema
- Serna 4.1 free — for editing
- Apache FOP 0.95 — for producing PDF output
- Doc Book XSLT style sheets 1.75.1

Parameter settings are required to set the font size for monospaced verbatim areas as code examples otherwise would be truncated. A size of 8pt is required.

Parameters also needs to be set to produce PDF “bookmarks” (i.e., a PDF TOC). This is done on the command line as a directive to `xsltproc`.

Tools used. This book was authored by using the following tools:

- Serna 4.1 free — for DocBook editing
- Inkscape 0.46 — for vector graphics
- LineForm 1.5 — for vector graphics
- graphviz 2.24 — for graph generation