BlueJ Blackbox Data Collection

Researchers' Handbook

Getting Started

This document acts as a manual for the BlueJ data collection project, "Blackbox". It is intended to explain, in detail, the schema of the collected data, and how the data relates to the BlueJ IDE. It assumes a basic familiarity with BlueJ's capabilities, such as the debugger, codepad, object bench and so on.

Chapter 1 explains the concepts, assumptions and caveats associated with the data recording. It is important to read this chapter before carrying out any detailed analysis, so that you do not proceed with false assumptions.

Chapters 2 and 3 contain technical detail on the data schema.

Chapter 4 contains details on how to actually access the data (MySQL details, etc), and chapter 5 contains the data-use policies which we expect all researchers to read. Finally, chapter 6 has details on the example application we have provided, and chapter 7 has a few brief notes about the beta phase.

Contents

1 Concepts and Caveats **6**
- 1.1 Users . . . 6
- 1.2 Projects . . . 6
- 1.3 Packages . . . 7
- 1.4 Sessions . . . 7
- 1.5 Participants . . . 7
- 1.6 Events . . . 8
- 1.7 Interrupted Connections . . . 8
- 1.8 Spam . . . 9
- 1.9 Source Code Consistency . . . 9
- 1.10 The Debug VM . . . 9
- 1.11 Extensions . . . 10
- 1.12 Database Structure and Notes . . . 10
  - 1.12.1 Interning . . . 10

2 Master Events Table **11**
- 2.1 Start/Finish Events . . . 12
  - 2.1.1 `"bluej_start"` and `"bluej_finish"` . . . 12
- 2.2 Project Events . . . 12
  - 2.2.1 `"project_opening"` . . . 13
  - 2.2.2 `"project_closing"` . . . 13
- 2.3 Package Events . . . 13
  - 2.3.1 `"package_opening"` . . . 13

2.3.2 "package_closing" . . . 13
2.4 Editing . . . 13
2.4.1 "edit" . . . 14
2.4.2 "file_add" . . . 14
2.4.3 "file_delete" . . . 14
2.4.4 "rename" . . . 14
2.5 Compilation . . . 14
2.5.1 "compile" . . . 14
2.5.2 Related Tables . . . 15
2.6 Miscellaneous Events . . . 16
2.6.1 "resetting_vm" . . . 16
2.6.2 "terminal_open" and "terminal_close" . . . 16
2.7 Debugger Events . . . 16
2.7.1 "debugger_open" and "debugger_close" . . . 17
2.7.2 "debugger_terminate" . . . 17
2.7.3 "debugger_continue" . . . 17
2.7.4 "debugger_hit_breakpoint", "debugger_stepinto", "debugger_stepover" and "debugger_halt" . . . 17
2.7.5 "debugger_breakpoint_add" and "debugger_breakpoint_remove" . . . 18
2.8 Codepad . . . 18
2.8.1 "codepad" . . . 18
2.9 Method Invocation . . . 19
2.9.1 "invoke_method" . . . 19
2.10 Inspector . . . 20
2.10.1 "inspector_show" and "inspector_hide" . . . 20
2.11 Object Bench . . . 21
2.11.1 "bench_get" . . . 21
2.11.2 "remove_object" . . . 21
2.12 Executing Tests . . . 21
2.12.1 "run_test" . . . 21
2.13 Recording Tests . . . 22
2.13.1 "start_test", "cancel_test" and "end_test" . . . 22
2.13.2 "assertion" . . . 22
2.13.3 "bench_to_fixture" and "fixture_to_bench" . . . 23
2.14 Version Control . . . 23
2.14.1 "vcs_history" and "vcs_share" . . . 24
2.14.2 "vcs_commit" and "vcs_update" . . . 24
2.14.3 "vcs_status" . . . 24

3 Other Database Tables **25**
3.1 Users . . . 25
3.2 Sessions . . . 25
3.3 Installation Details . . . 26
3.4 Extensions . . . 26
3.5 Projects . . . 27
3.6 Packages . . . 27
3.7 Source Files . . . 27
3.8 Source Histories . . . 28

3.9 Stack Entries . . . 28
3.10 Bench Objects . . . 29
3.11 Participant Information . . . 29
3.12 Schema Diagram . . . 31

4 Data Access **33**
4.1 SSH Access . . . 33
4.2 MySQL . . . 33
4.3 Feed . . . 33
4.4 Analysis Process . . . 34
4.5 Data Snapshot . . . 34

5 Data Use Policies **35**
5.1 Data Caching/Copying . . . 35
5.2 Data Use . . . 35
5.3 Sharing and Publication . . . 35

6 Analysis Example: CodeView **36**

7 Beta **37**
7.1 Aims . . . 37
7.2 Artificialities . . . 37
7.3 Notes . . . 37

List of Tables

1 Possible `master_events.name` values 5

2.1 `master_events` schema 11
2.2 `source_hashes` schema 13
2.3 `compile_events` schema 15
2.4 `compile_inputs` schema 15
2.5 `compile_outputs` schema 16
2.6 `debugger_events` schema 17
2.7 `breakpoints` schema 17
2.8 `codepad_events` schema 18
2.9 `invocations` schema 19
2.10 `inspectors` schema 20
2.11 `test_results` schema 21
2.12 `tests` schema 22
2.13 `assertions` schema 22
2.14 `fixtures` schema 23
2.15 `bench_objects_fixtures` schema 23
2.16 `version_control_events` schema 23
2.17 `version_control_files` schema 24

3.1 `users` schema 25
3.2 `sessions` schema 25
3.3 `installation_details` schema 26
3.4 `extensions` schema 26
3.5 `projects` schema 27
3.6 `packages` schema 27
3.7 `source_files` schema 27
3.8 `source_histories` schema 28
3.9 `stack_entries` schema 28
3.10 `bench_objects` schema 29
3.11 `participant_identifiers_for_experiment` schema 30
3.12 `sessions_for_experiment` schema 30

Table 1: List of different values allowed for the `name` column in the `master_events` column, with links to sections in which that event is described.

`"assertion"` ..... Section 2.13.2 on page 22
`"bench_get"` ..... Section 2.11.1 on page 21
`"bench_to_fixture"` ..... Section 2.13.3 on page 23
`"bluej_finish"` ..... Section 2.1.1 on page 12
`"bluej_start"` ..... Section 2.1.1 on page 12
`"cancel_test"` ..... Section 2.13.1 on page 22
`"codepad"` ..... Section 2.8.1 on page 18
`"compile"` ..... Section 2.5.1 on page 14
`"debugger_breakpoint_add"` ..... Section 2.7.5 on page 18
`"debugger_breakpoint_remove"` .... Section 2.7.5 on page 18
`"debugger_close"` ..... Section 2.7.1 on page 17
`"debugger_continue"` ..... Section 2.7.3 on page 17
`"debugger_halt"` ..... Section 2.7.4 on page 17
`"debugger_hit_breakpoint"` ..... Section 2.7.4 on page 17
`"debugger_open"` ..... Section 2.7.1 on page 17
`"debugger_stepinto"` ..... Section 2.7.4 on page 17
`"debugger_stepover"` ..... Section 2.7.4 on page 17
`"debugger_terminate"` ..... Section 2.7.2 on page 17
`"edit"` ..... Section 2.4.1 on page 14
`"end_test"` ..... Section 2.13.1 on page 22
`"file_add"` ..... Section 2.4.2 on page 14
`"file_delete"` ..... Section 2.4.3 on page 14
`"fixture_to_bench"` ..... Section 2.13.3 on page 23
`"inspector_hide"` ..... Section 2.10.1 on page 20
`"inspector_show"` ..... Section 2.10.1 on page 20
`"invoke_method"` ..... Section 2.9.1 on page 19
`"package_closing"` ..... Section 2.3.2 on page 13
`"package_opening"` ..... Section 2.3.1 on page 13
`"project_closing"` ..... Section 2.2.2 on page 13
`"project_opening"` ..... Section 2.2.1 on page 13
`"remove_object"` ..... Section 2.11.2 on page 21
`"rename"` ..... Section 2.4.4 on page 14
`"resetting_vm"` ..... Section 2.6.1 on page 16
`"run_test"` ..... Section 2.12.1 on page 21
`"start_test"` ..... Section 2.13.1 on page 22
`"terminal_close"` ..... Section 2.6.2 on page 16
`"terminal_open"` ..... Section 2.6.2 on page 16
`"vcs_commit"` ..... Section 2.14.2 on page 24
`"vcs_history"` ..... Section 2.14.1 on page 24
`"vcs_share"` ..... Section 2.14.1 on page 24
`"vcs_status"` ..... Section 2.14.3 on page 24
`"vcs_update"` ..... Section 2.14.2 on page 24

Chapter 1

Concepts and Caveats

There are several concepts involved in the Blackbox project, and it is very important when doing analysis on the data to understand the concepts involved, and particularly the limitations of the data collection. If you naively assume that one virtual user is one real person, you may draw false conclusions from your analysis.

1.1 Users

BlueJ stores its properties file in a per-user location. For example, on Linux, it uses `~/.bluej/bluej.properties`, whereas on Windows 7 it is (on a typical home installation) `C:\Users\Joe\bluej\bluej.properties`.

This properties file is the only mechanism that BlueJ has to have any sense of identity. If two physical people use the same account, they will appear to BlueJ as a single user (this is still quite common on home Windows systems). If someone uses BlueJ on a university machine, and on a laptop, and on a home machine, they will appear to be three users. If the computer is set up to not maintain a user's profile correctly (e.g. a school system which wipes a user's profile on logout – we know some systems do this!), they will get a new profile each time they load BlueJ, and thus will always appear to be a new user. If the same bluej.properties file is shared between many physical users (e.g. they all get the same profile on login), they will all appear to be the same user (operating with many parallel sessions).

The user's unique identifier or opt-out preference is stored in this properties file. Thus the concept of a "user" is tied to the properties file. It is important to understand this when reasoning about users.

1.2 Projects

BlueJ deals with projects. A project is a directory with a BlueJ project file, and source files (and class files, and maybe a README, and whatever else the user stores in the directory). In Blackbox terms, projects are associated with particular user ids. They are identified only by the associated user, and their full path (hashed, for anonymity).

If a user makes a copy of a project and opens it, it will appear to be an entirely separate project (though you could post-process the data to see that the project has the same source code as another of the user's projects). If a user has the project on a USB memory stick,

and it gets assigned a different drive letter next time they plug the stick in, the project will appear to be different. If a person loads the project from two different machines, even if the project has the same path on both machines, they will appear as separate projects belonging to separate users.

1.3 Packages

A project will have at least one package (the unnamed, default Java package), and potentially many. BlueJ displays a main interface window for each package that the user opens within the project. Thus there can be as many codepads as there are packages in the project. Some BlueJ features are per-package (codepad, object bench), while others are per-project (debugger, version control).

1.4 Sessions

Each time that BlueJ is opened, we deem the time from the opening to the closing to be "a session". Each time BlueJ is loaded, the client generates a unique session identifier to label all events from that session with.

If a user loads many projects at once in a single instance of BlueJ, these will all count as one session. If a user loads BlueJ twice in parallel (e.g. by clicking the shortcut too many times), both sessions will be recorded. But for this reason, it is usually a good idea to use sessions and projects to process events, rather than solely relying on event timestamps (which will interleave events from the two sessions). If a user accidentally closes BlueJ and immediately re-opens it, that will count as two sessions (though you could spot this using the timestamps on the events in post-processing).

There is no time limit on how long a session can last – the client-generated identifier is relied upon to differentiate sessions. It is possible for a session to last for weeks if a user keeps BlueJ open on their always-on machine. One potentially common case is that if a user never shuts down their laptop, and instead always makes it sleep, the session can last for a long time if they leave BlueJ open on their laptop (but see Section 1.7 on page 8 for why they may not send us much data this way, if the session stops sending data at any point).

1.5 Participants

All the data in the BlueJ data collection is anonymous. Part of the intention of the project is to support local experiments, where you, the local researchers, collect additional information about users. The additional information should be held by you, and the experiment approved and conducted according to your local ethics committee.

To allow you to link your participant data with our data, we have a mechanism for BlueJ users to specify an experiment identifier and a participant identifier (strings, up to 32 characters). These two identifiers should be assigned by you, the local researcher. We request that you assign an experiment identifier that is unlikely to clash with other researcher's identifiers. The participant identifier is entirely up to you, although we would expect that you use a generic identifier (e.g. a number), and that you hold the mapping between the participant's real identity and the participant number, as is standard practice.

(That is: do not ask the participants to store their name, or university login or similar, in the participant identifier.)

As described in section Section 3.11 on page 29, you will be able to retrieve participant identifiers for a specific experiment identifier, and retrieve session data for known experiment identifiers and participant identifiers. This means that you will not be able to see experiment or participant identifiers for experiments for which you do not know the identifier. This is deliberate, to help protect information about experiment participation and participant identification.

1.6 Events

The BlueJ client sends a stream of events to the BlueJ server, which acknowledges them. Ultimately, all the information recorded on the server came by way of these discrete events. Things like users or sessions are reconstructed from the individual events that have been sent. Every event that the server receives, without exception, is recorded with an entry in the `master_events` table (see chapter 2 on page 11).

1.7 Interrupted Connections

There are potential problems with missing events or entire sessions in the event stream for a user. You should never assume you have a user's complete history of interaction with BlueJ. Notably, entire sessions might be missing due to users using BlueJ without an internet connection (e.g. on an aeroplane). Similarly, you should not assume that you can see all interactions with a project – source code could be edited outside BlueJ.

Within a session, you can assume that you have all events, sequentially consistent, from the first `bluej_start` event, up until the last event that is recorded in the session. This may not be the actual final event that the user performed in their real session – the user's internet connection may drop, or BlueJ may crash or get killed before a batch of events is sent, and so on. But there will never be a "hole" in the middle of a session where events are lost, then resumed again. **Any event that fails to send will cause the BlueJ client to stop sending for the rest of the session** – each event is only sent once the event before it in the session has succeeded.

It is a deliberate decision to not attempt to resume sending data if previous data has failed to send. Part of this decision is technical – we do not want to keep retrying a send if the user has no Internet connection (or if our own server is inaccessible for some reason), and we do not want to cache lots of events on the client in the hope that the connection one day re-appears.

The other reason is that analysis becomes much harder if events within a session may be missing – data consistency is a nightmare. For example, you may see that a user has closed an inspector, but not how that inspector appeared. Or you may see an execution take place, but not know what breakpoints are set in the code. If gaps are allowed in the event stream, we would have to send a lot more state information with each event to make analysis possible.

1.8 Spam

The BlueJ data collection project server is an open web server to which anyone can submit, and the BlueJ client is an open-source program. There is no technical way to guarantee that the data coming in is from an unmodified BlueJ client – it could be modified, or it could simply be made-up spam data being submitted. It remains to be seen whether this will be an issue.

1.9 Source Code Consistency

When a package is opened, the full state of the source code of the project is sent to the server. Similarly, when a class is added to the project while BlueJ is running, the full code is sent. For all other edit events within a session, a diff is sent between the last successfully sent source code state (during that session) and the new state. Using a diff minimises the amount of data sent and stored.

You can assume that the source code diffs within a session are consistent. However, you should not assume that you have seen the full state of the source code between sessions. Source code may be edited outside BlueJ or may be edited by BlueJ when the user has no Internet connection. You should be able to detect this by comparing the source code state between the end of one session and the start of the next.

1.10 The Debug VM

BlueJ has two virtual machines. One, generally known as the server VM, handles:

- The graphical interface
- Code editing
- Compilation

No user code is ever run on the server VM. All user code is run on a second VM, known as the client VM or the debug VM. (There is only ever one debug VM at a time, for each project.) So things like:

- Object construction
- Method invocations
- Codepad use
- JUnit tests

Are all run on the debug VM. When the debugger is used, the interface and control code are in the server VM, using hooks into the debug VM to do stepping and so on. Whenever user code is executed, it is possible that the result is that the debug VM terminates, but this does not mean that BlueJ has exited. User code that runs forever will not interrupt BlueJ, but will prevent further use of the debug VM.

1.11 Extensions

BlueJ has an extension mechanism that allows programmatic operation of BlueJ from third-party extensions. (Not to mention that BlueJ is open-source, anyway.) The extensions can trigger many of the events described in this document. We do not annotate which events are triggered by extensions and which are not. This annotation would be a lot of extra technical work, and probably for not much purpose, since we do not believe many BlueJ users use extensions that will trigger these events.

We do record which extensions are loaded in the blackbox data, so you can use this to rule out any extensions which could be problematic (or more likely, to whitelist extensions that are not an issue).

1.12 Database Structure and Notes

The database is used as a persistence layer by our Ruby on Rails server, which uses an object-store model. Every table in the database, except many-to-many link tables, has an integer `id` field which acts as a primary key. Relations in the database are handled in two ways:

1. When the destination table is fixed and known – e.g. a master event has a user – there is a field in the `master_events` table named `user_id` which links to the `users` table.

2. When the destination table may vary, e.g. a master event has a subordinate "event" record which may be from one of many tables, there is both an `event_id` and an `event_type` field. The `event_type` field contains an uppercased camel-cased, non-underscored[1] version of the table name. So if a particular master event links to a compile event, the `event_type` field will have `"CompileEvent"`, and the `event_id` will correspond to an `id` in the `compile_events` table.

1.12.1 Interning

There are some tables in the database which are likely to feature the same information over and over again, such as the installation details. When the server comes to add a record to these tables with duplicate data to a previous record, it *may* reference the existing record instead, to save space, like interning strings. This is a performance optimisation that should have no effect on the semantics. Tables that do this are tagged as such.

[1] This is Rails' doing – don't ask.

Chapter 2

Master Events Table

The master events table is the place where all events are entered. There are lots of different types of events that can be received, each with a variety of different data, but all of them get one entry in the `master_events` table, with links to other tables where necessary.

Table 2.1: Schema for the `master_events` table.

Field Name	Type	Can Be Null?
id	bigint	NO
client_address_id	int	NO
created_at	datetime	YES
event_id	int	YES
event_type	varchar(32)	YES
name	varchar(32)	NO
package_id	int	YES
participant_id	int	YES
project_id	int	YES
sequence_num	int	NO
session_id	int	NO
source_time	datetime	NO
user_id	int	NO

The most important field in the table is the `name` field, which describes which kind of event it is, and determines which of the other fields will be present. The possible values for the `name` field are given in Table 1 on page 5.

Each master event has links (as per Section 1.12 on page 10) to other tables:

- `user_id` – present for all events
- `session_id` – present for all events
- `project_id` – present for all events except `"bluej_start"` and `"bluej_finish"`
- `package_id` – present for some events (see further details below).

- `event_id`, `event_type` – present for some events (see further details below).
- `participant_id` – present for all events, if the user currently has participant information specified.
- `client_address_id` – present for all events.

The `created_at` field is a server-side, trustworthy datestamp of when the event was received. The `source_time` is the client-side time, which is likely (but not guaranteed) to be sequentially consistent, but should not be trusted to be accurate.

The `sequence_num` field is provided by the client to make sure events are received in correct sequence (within a given session). The server ensures that the sequence numbers are valid (i.e. that they begin at 1, and each subsequent event increments the sequence number by 1).

The `participant_id` field links to a hidden `participants` table. If you want to investigate this information, see Section 3.11 on page 29 on how to access this information.

The `client_address_id` field links to a hidden `client_address` table. However, you can reasonably assume that there is a one-to-one mapping between `client_address_id` and IP addresses from which the event is sent.

The subsequent sections in this chapter describe the different possible values for the `master_events.name` field, and what that means for the other fields, especially what is linked to by the `event_id` and `event_type` fields. Each section title has a string, which is the value of the `name` field.

2.1 Start/Finish Events

Two bookend events are sent, one when BlueJ starts (before it has loaded up a project or done anything) and one when BlueJ exits (after it has closed all projects and finished everything). The data will likely contain more start events than finish events, for reasons explained in Section 1.7 on page 8.

2.1.1 `"bluej_start"` and `"bluej_finish"`

These two events have no extra accompanying event. They have no `project_id` and no `package_id`. However, the `"bluej_start"` event will have zero or more attached entries in the `extensions` table (see Section 3.4 on page 26) that link back to this master event – these are extensions loaded from the system directories.

The `"bluej_start"` event will always create a new entry in the `sessions` table (see Section 3.2 on page 25) – the installation details for this copy of BlueJ will be recorded there.

2.2 Project Events

When a project is opened or closed in BlueJ, corresponding events will be sent. See Section 1.2 on page 6 for details on how projects are distinguished. A project-open event will be followed by at least one package-open event.

2.2.1 "project_opening"

This event has no package_id.

This event will have zero or more entries in the extensions table (see Section 3.4 on page 26) linking back to this event, which represent extensions loaded from this project's directory.

2.2.2 "project_closing"

This event will no attached data, and will have no package_id.

2.3 Package Events

When a package is opened or closed in BlueJ, a corresponding event is generated. Note that each window in BlueJ actually belongs to a package.

2.3.1 "package_opening"

This event will have a package_id.

The event will have zero or more entries in the source_histories table (see Section 3.8 on page 28) that link back to this event. These represent the state of the files as the project is opened (as a complete snapshot, not a diff).

The event will link to an entry in the source_hashes table (event_type will be "SourceHash"), which will contain a hash of the source code in the package. The exact details of the hash are unimportant, but this hash can be used to quickly narrow down likely duplicate projects (for example, spotting the starting projects from the BlueJ book).

Table 2.2: Schema for the source_hashes table. This table may be interned, see Section 1.12.1 on page 10.

Field Name	Type	Can Be Null?
id	int	NO
source_hash	varchar(64)	YES

2.3.2 "package_closing"

This event will have a package_id, but no other associated data.

2.4 Editing

We do not capture all the different edit operations (e.g. paste, indent), we only capture the resulting effect on the source code.

2.4.1 "edit"

This event will have a package_id, pointing to the package of the class that is edited.

This event is the central editing event in BlueJ. These events are recorded when multiple lines are edited at once (e.g. via paste, or deleting a selection, or so on), or a single line has been edited and the caret has now left that edited line. An edit event is not generated by every single keystroke.

There is currently no guarantee as to how edits are grouped together into a single edit event (e.g. highlighting a selection and then pasting may appear as one event or two events, auto-indent may appear as one edit or several edits, and many other such examples), but an edit event will never span a compiler event, and all edits will be flushed before a compile event.

The event will always have exactly one associated entry in the source_histories table (see Section 3.8 on page 28), which will be a "diff" event.

2.4.2 "file_add"

This event will have a package_id, which indicates which package the file was added to.

This event is fired whenever a class is added to the BlueJ project. This may be via the New Class menu item, or via the Import Class menu item, or via version control.

The event will always have exactly one associated entry in the source_histories table (see Section 3.8 on page 28), which will be a "complete" event.

2.4.3 "file_delete"

This event will have a package_id, which indicates which package the file was deleted from.

This event is fired whenever a class is removed from a BlueJ project. This may be via the interface, or via version control.

The event will always have exactly one associated entry in the source_histories table (see Section 3.8 on page 28), which will be a "file_delete" event.

2.4.4 "rename"

This event will have a package_id, which indicates which package the file was renamed in.

This event is fired whenever a class is renamed in a BlueJ project. A rename occurs in BlueJ when a Java file is saved, and is parseable, and has a different top-level class name to the name of the file. At this point, BlueJ silently renames the file to match the class name. But note that until this save occurs, many more edits may take place, so the rename of the file may appear some time after the actual relevant source code edit takes place.

The event will always have exactly one associated entry in the source_histories table (see Section 3.8 on page 28), which will be a rename event.

2.5 Compilation

2.5.1 "compile"

This event will have a package_id if and only if all the compiled files were in the same package. However, having a package_id does not necessarily indicate that the user asked for the whole package to be compiled.

This event occurs whenever a compilation of the user's classes is triggered. (This does not include things like codepad interactions).

This master event will always have `event_type` set to `"CompileEvent"`, and `event_id` will reference an entry in the `compile_events` table (see Section 2.5.2 on page 15). (In turn, that entry in the `compile_events` table, will be linked to by one or more `compile_inputs` and zero or more `compile_outputs`, but see the `compile_events` table (see Section 2.5.2 on page 15) for more details.)

2.5.2 Related Tables

The `compile_events` table records whether the compilation was a success (i.e. no errors – but there could be warnings) or not.

Table 2.3: Schema for the `compile_events` table.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`success`	`tinyint`	`NO`

The `compile_events` may then be linked to by entries in the `compile_inputs` and `compile_outputs` table.

Table 2.4: Schema for the `compile_inputs` table.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`compile_event_id`	`int`	`NO`
`source_file_id`	`int`	`NO`

The `compile_inputs` table is really just a link table, linking one compile event to many source files.

Table 2.5: Schema for the `compile_outputs` table.

Field Name	Type	Can Be Null?
id	int	NO
compile_event_id	int	NO
end_column	int	YES
end_line	int	YES
is_error	tinyint	NO
message	varchar(255)	YES
shown	tinyint	NO
source_file_id	int	YES
start_column	int	YES
start_line	int	YES

The `compile_outputs` table has an entry for each warning and each error that occurred. Note that these warnings and errors are reported from the standard Java compiler from the JDK. However, BlueJ may alter the format of some messages, and will only display at most one error, and may ignore some warnings. The message recorded here is the altered format, and the `shown` flag indicates whether or not this error or warning was actually displayed to the user. The `is_error` flag indicates if it was an error (otherwise it was a warning), the `message` field shows the text that would be displayed to the user, and the line and column fields indicate the error's location, as reported by the compiler (lines and columns begin at 1, not 0).

2.6 Miscellaneous Events

2.6.1 `"resetting_vm"`

This event has no `package_id`, because the debug VM is per-project.

The event occurs when the VM is restarted due to an explicit user request (via a menu). Although it has the same effect as the `debugger_terminate` event, it is recorded separately.

2.6.2 `"terminal_open"` and `"terminal_close"`

These events have no `package_id`, because there is only one terminal per project.

These events occur when the terminal window is shown or closed. It may be shown either due to explicit triggered by the user (via a menu item) or because output has been generated by the debug VM (which causes the terminal to be shown if it is not already shown).

2.7 Debugger Events

Some, but not all, debugger events link to the `debugger_events` table, which can store thread names.

Table 2.6: Schema for the `debugger_events` table.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`thread_name`	`varchar(255)`	`NO`

Other debugger events will link to the `breakpoints` table, which references a source file and a line number (line numbers begin at 1, not 0).

Table 2.7: Schema for the `breakpoints` table. This table may be interned, see Section 1.12.1 on page 10.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`line_number`	`int`	`YES`
`source_file_id`	`int`	`YES`

2.7.1 `"debugger_open"` and `"debugger_close"`

These events have no `package_id` (because the debugger is per-project).

These events occur when the debugger window is opened or closed. The debugger may open because the user explicitly opened the debugger (via the menu), or because the code has hit a breakpoint. No further data is attached.

2.7.2 `"debugger_terminate"`

This event has no `package_id` (because the debugger is per-project).

This event occurs when the user clicks the Terminate button in the debugger. It kills the current debug VM (which will be subsequently restarted, but will be idle), and thus all currently-executing user code. No further data is attached.

2.7.3 `"debugger_continue"`

This event has no `package_id` (because the debugger is per-project).

This event occurs when the user clicks the Continue button in the debugger on a thread which is currently not executing. It makes the selected thread resume execution.

The `event_type` field will always be `"DebuggerEvent"`, and the `event_id` will reference an entry in the `debugger_events` table (see Section 2.7 on page 16). The `debugger_events` entry will have the `thread_name` filled in, but there will be no further information.

2.7.4 `"debugger_hit_breakpoint"`, `"debugger_stepinto"`, `"debugger_stepover"` and `"debugger_halt"`

These events have no `package_id` (because the debugger is per-project).

These events occur when the user code in a thread becomes suspended, for different reasons. The `debugger_hit_breakpoint` event indicates that the suspension was due to hitting a breakpoint, while `debugger_stepinto` and `debugger_stepover` indicate that the thread has just completed a step request. (If a step request ends on a breakpoint, only the breakpoint is reported.) The `debugger_halt` is reported in the remaining case, that the user has clicked the Halt button in the debugger (which pauses execution of the thread).

For these events, the `event_type` field will be always be `"DebuggerEvent"`, and the `event_id` will reference an entry in the `debugger_events` table (see Section 2.7 on page 16), which will have the `thread_name` filled in.

There will also be one or more entries in the `stack_entries` table (see Section 3.9 on page 28) which will link to the `debugger_event` (not the master event) via the fields in that table `stack_entries.sub_event_type` and `stack_entries.sub_event_id`.

2.7.5 `"debugger_breakpoint_add"` and `"debugger_breakpoint_remove"`

These events do have a `package_id`, which indicates the package that the involved source file is in.

These events occur when a user adds/removes a breakpoint. The `event_type` field will be `"Breakpoint"` and the `event_id` will reference an entry in the `breakpoints` table (see Table 2.7 on page 17), which details the location of the breakpoint.

2.8 Codepad

2.8.1 `"codepad"`

These events have a `package_id`, which indicates which package window the codepad being used corresponds to – more by accident than by design, each package window has its own codepad.

These events occur each time that the user types something into the codepad window, presses return and gets a response. The codepad event links to an item in the `codepad_events` table (the `event_type` will be `"CodepadEvent"`, and the `event_id` will reference an entry in the `codepad_events` table).

Table 2.8: Schema for the `codepad_events` table.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`command`	`text`	`NO`
`error`	`varchar(255)`	`YES`
`exception`	`varchar(255)`	`YES`
`outcome`	`varchar(255)`	`YES`
`result`	`varchar(255)`	`YES`

The `outcome` field indicates what happened when the code was executed. An `outcome` of `"success"` indicates that the code successfully compiled, and executed without throwing

an exception or without exiting the VM early. In this case, the fields `command` and `result` will be filled in with the input and output respectively.

The `outcome` field being `"error"` indicates a compiler error occurred, or that the VM exited during execution: the `command` and `error` fields will be filled in with the input and error respectively. The `outcome` field being `"exception"` indicates an exception occurred: the `command` and `exception` fields will be filled in with the input and exception respectively.

2.9 Method Invocation

Method invocations reference the `invocations` table.

Table 2.9: Schema for the `invocations` table.

Field Name	Type	Can Be Null?
id	int	NO
bench_object_id	int	YES
code	text	YES
compile_error	varchar(255)	YES
exception_class	varchar(255)	YES
exception_message	text	YES
result	varchar(255)	YES
session_id	int	YES
session_subid	int	YES
test_id	int	YES
type_name	varchar(255)	YES

2.9.1 `"invoke_method"`

This event has a `package_id` field, which indicates which package window the method was invoked from. The method invoked is not guaranteed to be in this package, as the user may have used the call-library-class function to invoke a method on String, or any other class.

This event occurs when the user interactively (i.e. via a menu) invokes a static method on a class or an instance method on an object on the object bench, or uses the call-library-class menu function. The `event_type` will always be `"Invocation"` and the `event_id` will reference an entry in the `invocations` table.

In the `invocations` table, the `code` field will have the code that was generated to execute the invocation. The `result` field will have the result, which will be one of:

- `"success"` – method was invoked successfully, the `bench_object_id` field will link to an entry in the `bench_objects` table (see Section 3.10 on page 29) – if the result was added to the object bench, i.e. if the method was a constructor – and the the `type_name` field will have the type of the result.
- `"compile_error"` – an error occurred while compiling the generated code (e.g. because the user entered invalid parameters), the `compile_error` field will have the full message.

- `"terminated"` – the VM terminated during execution, e.g. due to a `System.exit` call.
- `"exception"` – an uncaught exception occurred during execution. The `exception_class` field will have the class of the exception, the `exception_message` field will have the message, and there will be associated entries in the `stack_entries` table (see Section 3.9 on page 28) linking to this event (the `invocations` entry, not the `master_events` entry).

2.10 Inspector

The inspector events reference the inspectors table.

Table 2.10: Schema for the `inspectors` table.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`bench_object_id`	`int`	`YES`
`class_name`	`varchar(255)`	`YES`
`display_name`	`varchar(255)`	`YES`
`session_id`	`int`	`NO`
`static_class`	`varchar(255)`	`YES`
`unique`	`varchar(32)`	`YES`

2.10.1 `"inspector_show"` and `"inspector_hide"`

This event has a `package_id` field, which indicates which package window the inspection was invoked from. This is transitive, for occasions where users inspect an object in an inspector.

Both of these events link to an entry in the `inspectors` table: the `event_type` field will be `"Inspector"`, and the `event_id` field will be a reference to an entry in the `inspectors` table.

For the `inspector_show` event, there are two cases:

1. The inspection was on a class. In this case, the `static_class` field will have the name of the class, and the other fields will be empty.
2. The inspection was on an object. In this case, the `class_name` field indicates the type of the object, the `display_name` (whatever is displayed on the left-hand side of the title in the object inspector) will be present, and the `bench_object_id` will link to a `bench_object` if the inspected object is on the object bench (regardless of whether that was how this object was reached for inspection).

In both cases, the `unique` will have some string that uniquely identifies an inspector within the session (this is useful during collection, and is of no use afterwards, because the record will be unique per inspector).

For the `inspector_hide` event, there will be a link back to the `inspectors` entry that was hidden.

2.11 Object Bench

2.11.1 "bench_get"

This event has a package_id, linking to which package the object bench is associated with.

The event occurs when a user is inspecting something (an object, class, or method result) and uses the Get button to transfer the object to the object bench – or when they drag a result object over from the codepad.

The event links to an entry in the bench_objects table (see Section 3.10 on page 29) (event_type will be "BenchObject"), indicating the object which was added to the bench.

If a test was being recorded at the time, the bench_objects entry will have a link to the tests table, indicating which test was being recorded.

2.11.2 "remove_object"

This event has a package_id, linking to which package the object bench is associated with.

The event occurs when a user explicitly removes an object from the object bench.

The event links to an entry in the bench_objects table (see Section 3.10 on page 29), indicating the object which was removed from the bench.

2.12 Executing Tests

2.12.1 "run_test"

This event has a package_id, linking to which package the test class is in.

The event occurs when the user executes a test, either by using the Test All command, or individually invoking a test. Each test that is run will get its own run_test event.

The event links to an entry in the test_results table.

Table 2.11: Schema for the test_results table.

Field Name	Type	Can Be Null?
id	int	NO
class_name	varchar(255)	NO
exception_message	text	YES
exception_trace	text	YES
method_name	varchar(255)	NO
result	varchar(16)	NO
run_time	int	NO

The class_name and method_name fields describe which test method was executed. The run_time field has the number of milliseconds that the test took to execute. The result field will have one of three values:

- "success" – the test passed, no further information is supplied.
- "failure" – the test failed with a failed assertion. The exception_message and exception_trace fields have more information.

- `"error"` – the test failed due to an exception (other than an assertion). The `exception_message` and `exception_trace` fields have more information.

2.13 Recording Tests

2.13.1 `"start_test"`, `"cancel_test"` and `"end_test"`

This event has a `package_id`, linking to which package the test class is in.

These events relate to recording a test. The user performs various actions, and at the end of the recording, code is generated in a new test method to mirror the user's actions. Each of these events links to an entry in the `tests` table.

Table 2.12: Schema for the `tests` table.

Field Name	Type	Can Be Null?
id	int	NO
method_name	varchar(255)	YES
session_id	int	YES
session_subid	int	YES
source_file_id	int	YES
status	varchar(255)	YES

Each entry in the `tests` table is one attempt (either ongoing, successful, or aborted) at recording a test. The `source_file_id` links to the source file in which the final test method (named according to `method_name`) will be generated. The `status` field will be `"Recording"`, `"Recorded"` or `"Cancelled"`, according to the test's current status.

The `session_subid` field is an internal identifier used by the client to identify tests, and can be ignored. The `session_id` field links to the session in which the test was recorded.

2.13.2 `"assertion"`

This event has a `package_id`, linking to which package the test was recorded for.

The event links to an entry in the `assertions` table.

Table 2.13: Schema for the `assertions` table.

Field Name	Type	Can Be Null?
id	int	NO
assertion	varchar(255)	YES
invocation_id	int	YES
param1	varchar(255)	YES
param2	varchar(255)	YES
test_id	int	YES

The `assertions` table links to an entry in the `invocations` table that the assertion relates to, and to an entry in the `tests` table (see Section 2.13.1 on page 22) for the test which this assertion will be recorded into.

The `assertion` field is the JUnit function that will be recorded, e.g. `"assertEquals"`. The `param1` field and `param2` fields are the other parameters to the assertion.

2.13.3 `"bench_to_fixture"` and `"fixture_to_bench"`

These events have a `package_id`, linking to the package in which the object bench and test class are.

These events occur when the user captures all the objects on the object bench and turns their creation into a test setup phase (bench to fixture) or runs a test setup to put the objects back onto the object bench (fixture to bench).

These events link to an entry in the `fixtures` table.

Table 2.14: Schema for the `fixtures` table.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`source_file_id`	`int`	`YES`

Each entry in the `fixtures` table links to the source file holding the test class involved. A new fixture will be created for each `"bench_to_fixture"` or `"fixture_to_bench"` event.

Table 2.15: Schema for the `bench_objects_fixtures` table.

Field Name	Type	Can Be Null?
`bench_object_id`	`int`	`YES`
`fixture_id`	`int`	`YES`

The `bench_objects_fixtures` table is a simple link table that links object bench objects to test fixtures. In the case of `"bench_to_fixture"`, these objects are already on the bench and will be removed by this event. In the case of `"fixture_to_bench"`, these objects will be added to the bench, and are created by this event.

2.14 Version Control

Table 2.16: Schema for the `version_control_events` table.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`protocol`	`varchar(255)`	`YES`
`vcs_type`	`varchar(255)`	`NO`

Table 2.17: Schema for the `version_control_files` table.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`master_event_id`	`int`	`NO`
`source_file_id`	`int`	`NO`
`status`	`varchar(255)`	`YES`

Each version control event has an associated entry in the `version_control_events` table, with a `vcs_type` (built-in options: `"CVS"`, `"SVN"`) and a `protocol` (e.g. `"http"` or `"svn+ssh"`).

2.14.1 `"vcs_history"` and `"vcs_share"`

These events get an associated entry in the `version_control_events` table, but no more data than that.

2.14.2 `"vcs_commit"` and `"vcs_update"`

These events get an associated entry in the `version_control_events` table, and also zero or more entries in the `version_control_files` table, with just the `file` field filled in.

2.14.3 `"vcs_status"`

This event gets an associated entry in the `version_control_events` table, and also zero or more entries in the `version_control_files` table, with the `file` field filled in, and the `status` field filled in.

Chapter 3

Other Database Tables

3.1 Users

Table 3.1: Schema for the `users` table.

Field Name	Type	Can Be Null?
id	int	NO
created_at	datetime	YES
last_update	datetime	YES
uuid	varchar(64)	NO

The `uuid` field is the unique user identifier, as discussed in Section 1.1 on page 6. The `created_at` field shows the time that this user record was created (the first time their UUID was seen), and the `last_update` field shows the last time that data was entered for this user.

3.2 Sessions

Table 3.2: Schema for the `sessions` table.

Field Name	Type	Can Be Null?
id	int	NO
created_at	datetime	NO
identifier	varchar(64)	NO
installation_details_id	int	NO
last_sequence_id	int	NO
participant_id	int	YES
user_id	int	NO

Each session is associated with a specific user. There is a link to the `installation_details` table (see Section 3.3 on page 26) – specifically, the entry that accompanied the `bluej_start` event that began this session. The `identifier` and `last_sequence_id` fields are used for internal book-keeping, and can be ignored. There is a link to an entry in the `participants` table (see Table 2 on page 12) if this user had participant information filled in during the last event that was added to this session.

3.3 Installation Details

Table 3.3: Schema for the `installation_details` table. This table may be interned, see Section 1.12.1 on page 10.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`bluej_version`	`varchar(255)`	`YES`
`interface_language`	`varchar(255)`	`YES`
`java_version`	`varchar(255)`	`YES`
`operating_system`	`varchar(255)`	`YES`

The `installation_details` table contains the BlueJ `interface_language` – the possible options are listed in the bluej.defs file. The `operating_system` field glues together (with slashes) the output of Java's `System.getProperty` call for the keys `"os.name"`, `"os.arch"` and `"os.version"`. The `bluej_version` is the version of BlueJ they are running, and `java_version` is the Java version being used to run BlueJ.

3.4 Extensions

Table 3.4: Schema for the `extensions` table.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`master_event_id`	`int`	`NO`
`name`	`varchar(255)`	`NO`
`version`	`varchar(255)`	`NO`

The `extensions` table contains the (self-reported) name and version of a BlueJ extension. BlueJ comes with the submitter extension installed by default.

3.5 Projects

Table 3.5: Schema for the `projects` table.

Field Name	Type	Can Be Null?
id	int	NO
name	varchar(64)	NO
path_hash	varchar(255)	YES
user_id	int	NO

Each project is associated with a specific user. A project is stored with the md5 hash of its full path (`path_hash`), to try to identify the project across multiple sessions. For anonymity purposes, the full path is not sent to the server (because it will typically have the user-name in the path), but the directory name of the project (`name`) is sent, as this may be informative.

3.6 Packages

Table 3.6: Schema for the `packages` table.

Field Name	Type	Can Be Null?
id	int	NO
name	varchar(255)	YES
project_id	int	YES

Each package belongs to a specific project. A package is identified by its fully-qualified package name. The default, unnamed package will have the empty string as its name.

3.7 Source Files

Table 3.7: Schema for the `source_files` table.

Field Name	Type	Can Be Null?
id	int	NO
name	varchar(255)	NO
project_id	int	NO

Each entry in the `source_files` table corresponds to a particular source path in a project. (That is, if you delete a source file and generate a new file with the same name, it may share the entry of the original file.)

Each source file will have its path relative to the base of the project. So files that are within packages will have a path with the parent package names as directories (which may use forward slash or backslash for separation).

3.8 Source Histories

Table 3.8: Schema for the `source_histories` table.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`content`	`text`	`YES`
`master_event_id`	`int`	`NO`
`source_file_id`	`int`	`NO`
`source_history_type`	`varchar(255)`	`YES`

Each entry in the `source_histories` table belongs to a particular `master_event`, and is associated with a given `source_file`. Based on the `source_history_type` field, there can be different meanings to the `content` field:

- `source_history_type` is `"complete"` – `content` has the complete text of the source file (anonymised). This will be used by file additions, and source histories added on loading a package.
- `source_history_type` is `"diff"` – `content` has the diff (in unified diff format) from the previous version of the file.
- `source_history_type` is `"rename"` – `content` holds the name of the file before the rename. During the rename, the source history will be updated to feature the new name of the file.
- `source_history_type` is `"file_delete"` – `content` will be blank.

3.9 Stack Entries

Table 3.9: Schema for the `stack_entries` table.

Field Name	Type	Can Be Null?
`id`	`int`	`NO`
`class_name`	`varchar(255)`	`YES`
`class_source_name`	`varchar(255)`	`YES`
`entry`	`int`	`YES`
`line_number`	`int`	`YES`
`sub_event_id`	`int`	`YES`
`sub_event_type`	`varchar(255)`	`YES`

Any time that an event is associated with a stack trace, it will have zero or more entries in this table: one for each line in the stack trace. (Some stack traces may be filtered by BlueJ – the recorded trace will always be the filtered version, i.e. the version that the user actually sees.)

The `entry` field will be the index within the stack trace. The top line (typically, the actual location of the thrown exception, or breakpoint, etc) of the trace will have `entry` of zero, with an increasing `entry` index for each subsequent line.

The `class_source_name` field will be the name of the file involved, and `line_number` will be the line number (line numbers start at 1). So if an exception is thrown in an anonymous inner class of class Foo, this field will be `"Foo.java"` – even if Foo is inside a package. The source name loses all package information, but this can be recovered via the `class_name` field, which is the fully qualified name of the class at the specific source location.

The `sub_event_id` and `sub_event_type` fields are what is used to link the stack entries to their parent event – see documentation throughout the manual for events that have associated stack traces.

3.10 Bench Objects

Table 3.10: Schema for the `bench_objects` table.

Field Name	Type	Can Be Null?
id	int	NO
class_name	varchar(255)	YES
name	varchar(255)	YES
package_id	int	YES
session_id	int	NO
test_id	int	YES

Objects on the BlueJ object bench get an entry in the `bench_objects` table. The `name` field is the name/label of the object, and `class_name` is the type of the object. The `package_id` field links to the package which is associated with the object bench that this object is on. The `session_id` field links to the session during which this object existed.

The `test_id` field links to an entry in the `tests` table (see Section 2.13.1 on page 22) – this link is present if (and only if) the given test was being recorded when this object was added to the bench.

3.11 Participant Information

To keep participant information (see Section 1.5 on page 7) hidden, we hide the actual table involved, `participants`, and instead offer three database views of the information.

Table 3.11: Schema for the `participant_identifiers_for_experiment` table.

Field Name	Type	Can Be Null?
participant_identifier	varchar(32)	NO

The `participant_identifiers_for_experiment` view allows you to see the available participant identifiers for a given experiment identifier. You can call the view using a statement such as:

```
select p.participant_identifier from
  (select @experiment:="myexperiment") unused,
  participant_identifiers_for_experiment p;
```

The inner select is a trick to set the `@experiment` parameter to be the experiment identifier. The view then retrieves a list of participant identifiers based on the experiment identifier that you supply.

Table 3.12: Schema for the `sessions_for_experiment` table.

Field Name	Type	Can Be Null?
id	int	NO
created_at	datetime	NO
identifier	varchar(64)	NO
installation_details_id	int	NO
last_sequence_id	int	NO
participant_id	int	YES
participant_identifier	varchar(32)	NO
user_id	int	NO

The `sessions_for_experiment` view allows you to retrieve information for all the sessions associated with a given experiment identifier. A session is associated with an experiment identifier (and similarly with a participant identifier) if during the last event logged in that session, the user had their experiment identifier set to the given value. So if a user sets their identifiers partway through a session (and does not change it again before the end of the session), the whole session is deemed to be part of the experiment.

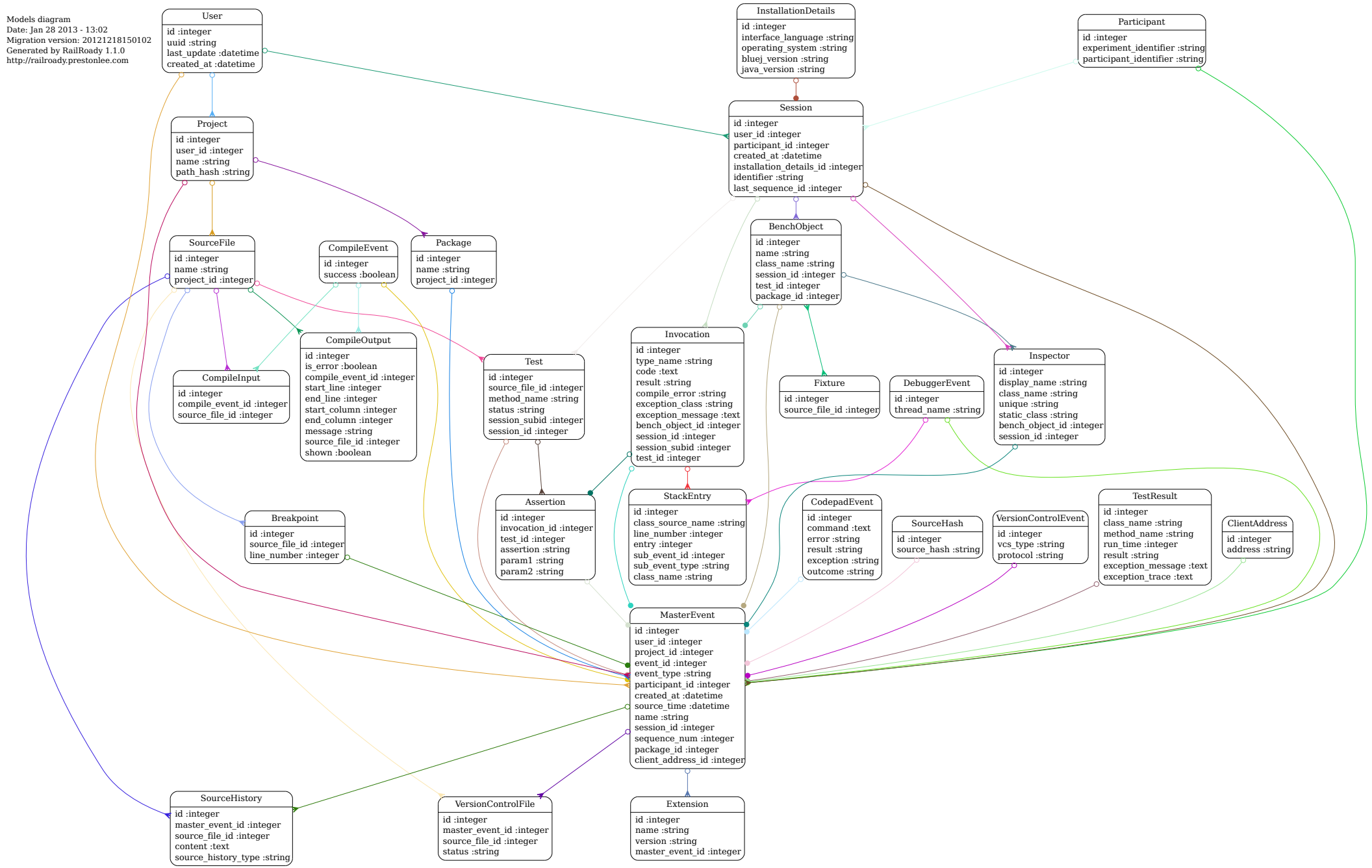
The columns returned are those from the `sessions` table (see Section 3.2 on page 25), plus the `participant_identifier` column.

The view can be called using a statement such as:

```
select s.* from
  (select @experiment:="myexperiment") unused,
  sessions_for_experiment s;
```

3.12 Schema Diagram

The next page has an overview diagram for all the tables in the database, and how they connect


Models diagram
Date: Jan 28 2013 - 13:02
Migration version: 20121218150102
Generated by RailRoady 1.1.0
http://railroady.prestonlee.com
User
id :integer
uuid :string
last_update :datetime
created_at :datetime
InstallationDetails
id :integer
interface_language :string
operating_system :string
bluej_version :string
java_version :string
Participant
id :integer
experiment_identifier :string
participant_identifier :string
Project
id :integer
user_id :integer
name :string
path_hash :string
Session
id :integer
user_id :integer
participant_id :integer
created_at :datetime
installation_details_id :integer
identifier :string
last_sequence_id :integer
SourceFile
id :integer
name :string
project_id :integer
CompileEvent
id :integer
success :boolean
Package
id :integer
name :string
project_id :integer
BenchObject
id :integer
name :string
class_name :string
session_id :integer
test_id :integer
package_id :integer
CompileOutput
id :integer
is_error :boolean
compile_event_id :integer
start_line :integer
end_line :integer
start_column :integer
end_column :integer
message :string
source_file_id :integer
shown :boolean
Test
id :integer
source_file_id :integer
method_name :string
status :string
session_subid :integer
session_id :integer
Invocation
id :integer
type_name :string
code :text
result :string
compile_error :string
exception_class :string
exception_message :text
bench_object_id :integer
session_id :integer
session_subid :integer
test_id :integer
Inspector
id :integer
display_name :string
class_name :string
unique :string
static_class :string
bench_object_id :integer
session_id :integer
CompileInput
id :integer
compile_event_id :integer
source_file_id :integer
Fixture
id :integer
source_file_id :integer
DebuggerEvent
id :integer
thread_name :string
Assertion
id :integer
invocation_id :integer
test_id :integer
assertion :string
param1 :string
param2 :string
StackEntry
id :integer
class_source_name :string
line_number :integer
entry :integer
sub_event_id :integer
sub_event_type :string
class_name :string
CodepadEvent
id :integer
command :text
error :string
result :string
exception :string
outcome :string
SourceHash
id :integer
source_hash :string
VersionControlEvent
id :integer
vcs_type :string
protocol :string
TestResult
id :integer
class_name :string
method_name :string
run_time :integer
result :string
exception_message :text
exception_trace :text
ClientAddress
id :integer
address :string
Breakpoint
id :integer
source_file_id :integer
line_number :integer
MasterEvent
id :integer
user_id :integer
project_id :integer
event_id :integer
event_type :string
participant_id :integer
created_at :datetime
source_time :datetime
name :string
session_id :integer
sequence_num :integer
package_id :integer
client_address_id :integer
SourceHistory
id :integer
master_event_id :integer
source_file_id :integer
content :text
source_history_type :string
VersionControlFile
id :integer
master_event_id :integer
source_file_id :integer
status :string
Extension
id :integer
name :string
version :string
master_event_id :integer

Chapter 4

Data Access

The data collection project involves two machines, running Ubuntu. Each machine has two 6-core hyperthreaded 2.5Ghz Xeon processors, with 32GB of RAM, and four 2TB drives set up in RAID 5 configuration to hold the database.

One machine, nicknamed black, is responsible for collecting the data. It solely runs the MySQL database, and the Ruby on Rails server.

The second machine, nicknamed white, has a live-cloned database (named `blackbox_production`) from the recording machine. White is the machine to which you will have an account, and be able to log in and perform data analysis. The intention is that this way, nothing that anyone does while examining the database on white can affect the recording on black.

4.1 SSH Access

The white machine can be accessed by SSHing to `white.kent.ac.uk` on the standard port (22). The machine runs Ubuntu. If you need any software installed, then we can install any standard Ubuntu packages for you (email `nccb@bluej.org` and `davmac@bluej.org`).

4.2 MySQL

The white machine runs MySQL on the standard port (3306) on the localhost interface. If you want to access the database directly from your own machine, you can use an SSH tunnel:

```
ssh -L 3306:localhost:3306 white.kent.ac.uk
```

While that SSH session is open, you can access the database by running `mysql -u username -p blackbox_produc`

4.3 Feed

To get familiarity with the data collection, we provide a mechanism to observe a live feed of your own data. The live feed is available, from the white machine, on `http://localhost:3080/feed`. It is probably easiest to port-forward 3080 to your local machine using:

```
ssh -L 3080:localhost:3080 white.kent.ac.uk
```

Then access `http://localhost:3080/feed` on your own machine. You can find your own UUID in your `bluej.properties` file – look for the line `blackbox.uuid=...`. If you paste your UUID into the feed page and click the button, it will refresh every five seconds until closed (be warned!) to show you the data for that UUID. Looking at this webpage while running BlueJ should give a good feel for what is being recorded.

4.4 Analysis Process

We anticipate that researchers will do some of their analysis on the white machine, and some on their own machine. The advantage of using the white machine is that it is high-powered, and has immediate access to all of the data, without needing to copy the data across the Internet. The disadvantage is that it can be a bit restrictive to operate on a remote machine for all analysis. Researchers may therefore want to do a split, with some pre-processing on the white machine, followed by more investigation on their own machine.

4.5 Data Snapshot

When researchers publish their data, they will specify their data source. It is worth remembering that the Blackbox data is continually being added to, so it is also necessary to specify which time period data was used from. We suggest choosing a cut-off date (e.g. the beginning date of the analysis) and ensuring that only data up until that cut-off date is used, so that the source of the data can be correctly specified as "the Blackbox data repository was used, up until 2013-03-31". You can use the date stamps in the `master_events` table (see chapter 2 on page 11) to decide which data to use in your study. (Bear in mind that this may bisect some sessions – make sure you do not process the events in a session which fall outside your cut-off date).

Chapter 5

Data Use Policies

The data that we collect is not public. This is part of our ethics approval – it is only visible to us, and selected researchers (i.e. those to whom we give an account on the white machine).

5.1 Data Caching/Copying

We realise that researchers will ultimately end up taking some data (probably processed) on to their own machine, as a pragmatic step during the analysis. We ask that you act responsibly, and make sure that the raw data is not made public, and also that you take care to not let your account on the white machine be compromised. If you need multiple researchers at your institution to have access, we would prefer giving them all accounts rather than encouraging the practice of sharing accounts (and thus passwords).

5.2 Data Use

The data is intended for education research, and is anonymous. We expect you to not attempt to identify any students or data within the database, except your own, and that of any participants in a study for which you have ethical approval. In particular, we forbid any use of the data for the purposes of plagiarism detection (and similar) in students at your local institution. The participation of the users in the experiment is conditional on the data being anonymous, and should not have any negative impacts for the participants.

5.3 Sharing and Publication

Researchers should not share the data with anyone – any other researchers wanting access should ask us for accounts. Obviously, at some point, researchers may want to publish their findings. We expect that the data will either be published in an aggregated form (which thus retains anonymity), and that any individual case studies/vignettes will be manually anonymised, if necessary.

Chapter 6

Analysis Example: CodeView

We have created a simple proof-of-concept analysis tool ("CodeView") to give a quick idea of how analysis might work. It appears to have a lot of code, but almost all of it (in the `bluej.*` packages) is ripped from BlueJ – code for displaying the editor.

In the default package are five classes. `IdName` is a simple data holder, and `ListDialog` is a class to ask the user to pick an item from a list. The class `DatabaseInterface` has some Java code for interfacing with the database. It connects to localhost (assuming you are running the SSH tunnel as described in section 4.2). It demonstrates running a few queries, given the UUID, to find a user's projects, and the files within a chosen project. For that file, the full history of the file is pulled.

The `SourceHistory` class uses the DiffUtils Java library to apply the necessary diffs to reconstruct the source history. You can also use the command-line `patch` application (available on all Unix-like systems) or any similar library in other languages. The `Viewer` class ties all this together, and displays the source in a BlueJ editor window, and binds the scroll wheel to moving backwards and forwards through the version history.

(Note: the sample has a slight irritation that the cursor is always moved to the end of the file during scrolling, but the point is to demonstrate the database queries and diffs and so on.)

Chapter 7

Beta

7.1 Aims

The aim of the beta is two-fold:

- Firstly, we want to test the software outside of our local setup. We would appreciate if you were able to test the software in a variety of settings, e.g. from your home machine, from a student machine in your labs. This will help pick up any technical problems with slow connections, or firewalls, or unexpected lab setups and so on.
- Secondly, we invite feedback on the data schema (described in this document). Is there something we're not recording that you want to know? Are we recording something in a way that will inhibit easy analysis? Are some of our caveats too big, or our guarantees too weak? The recording mechanism is much easier to alter before the public release than afterwards.

7.2 Artificialities

- UUID is shown in preferences dialog
- Message logged to debug log if there is a problem with event submission

7.3 Notes

During the beta, it may prove necessary to change the data schema. In this case, data may be wiped, and analysis tools may need adjusting. This is in order to provide the best setting for collecting the actual data: once the beta is over, there will be a stable interface to code against, and no data will be removed.