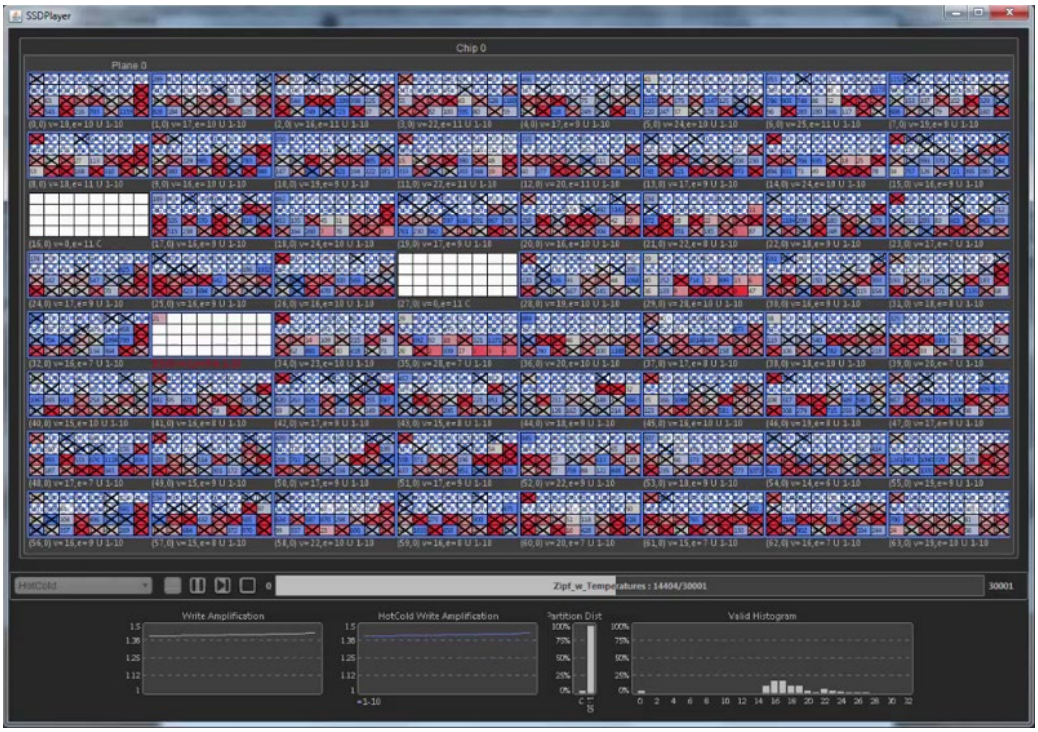
TECHNION
Israel Institute
of Technology

# SSDPlayer Visualization Platform

*Programmer's Guide for Version 1.3.0*




**Roman Shor**
Gala Yadgar
Or Mauda
Dolev Hadar
Roee Matza
Lior Zelikman

# Contents

# 1. Introduction

## 1.1. SSDPlayer overview

SSDPlayer is an open source graphical tool for visualizing data layout and movement on flash devices. It is designed to give a better understanding of how data gets from one place to another and why.

SSDPlayer supports two modes of operation. In simulation mode, it simulates the chosen device on a raw I/O trace or on a synthetic workload generated by the built in workload generator, illustrating the SSD state at each step. This illustration forms a "video" of the data movements that take place during execution. This mode is useful for testing and analyzing various features without, or before, implementing them in a full-scale simulator or hardware platform.

In visualization mode, SSDPlayer illustrates operations that were performed on an upstream simulator or device. The input in this mode is an output trace generated by a simulator, hardware evaluation platform, or a host level FTL, describing the basic operations that were performed on the flash device— writing a logical page to a physical location, changing block state, etc. This mode is useful for illustrating processes that occur in complex research and production systems, without porting their entire set of features into SSDPlayer.

The SSDPlayer display is organized into chips, planes, blocks and pages, as specified by the user at startup. Colors and textures are used to represent page and block properties, such as data 'temperature' or valid page count. A page's properties and state determine its fill color, texture, and frame color. A block's properties determine its background and frame colors. Users can control all the display parameters by editing the configuration file before starting SSDPlayer.

## 1.2. This Guide

This guide is intended as a starting point to users who wish to add or change features within the SSDPlayer implementation. Thus, we refer the reader to the SSDPlayer User's Guide for a more detailed description of the available features and on installing and using SSDPlayer.

The SSDPlayer homepage contains the full SSDPlayer documentation, executables, demos, and links to the SSDPlayer repository on GitHub.

# 2. Design principles

We describe the design principles of SSDPlayer to help the reader understand our motivation for structuring the code the way we have done. We also encourage future contributors to follow those principles in the design of their own additions.

## 2.1. General

SSDPlayer was designed to provide the most general SSD functionality, in order to allow easy extensions and additions for a wide range of capabilities. The basic flash components – e.g., page, block, page mapping and garbage collection – are implemented as abstract classes that can be extended according to the desired FTL functionality.

## 2.2. Platform Independent

SSDPlayer needs to be a portable software and supply testing and simulating capabilities to a wide range of users, in the academic and industry communities. The idea was to implement the simulator so that it will not be limited to a certain architecture or online use.

SSDPlayer is designed as an open source project in order to be helpful in researching SSD. Anyone who is interested in visualizing their use case can use the SSDPlayer source and extend it for their own purposes.

## 2.3. Easily Extendible

One of the more important goals of the project is to allow future research of different FTLs. Thus, we designed the simulator to be easily extendible and configurable for the specific needs of any future work.

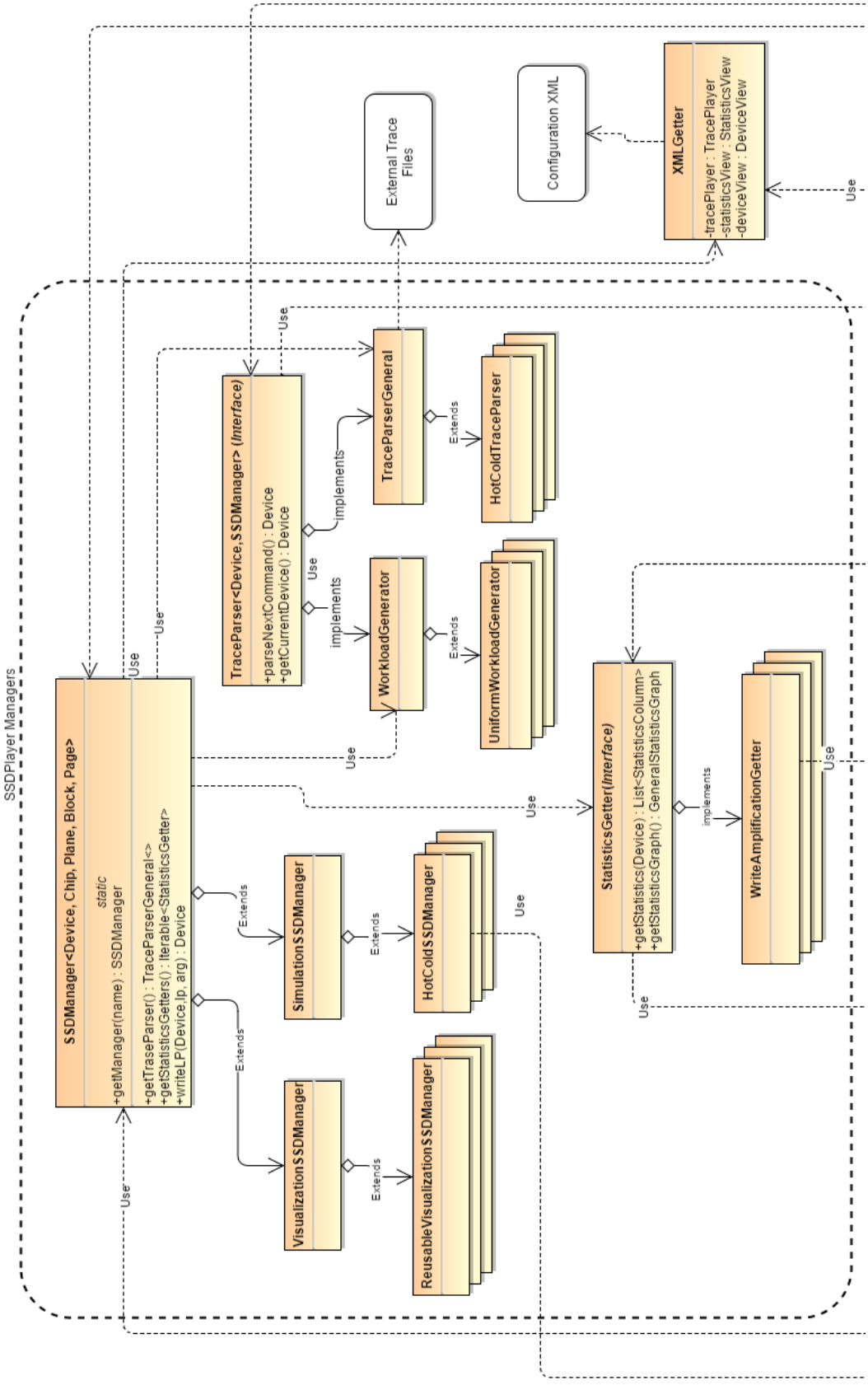
All the simulator components are easily extendible. The *Trace Parser* can be extended for a new use case, additional aggregated statistics may be added to the basic *histograms*, and new synthetic access distributions can be used as *Workload Generators*.

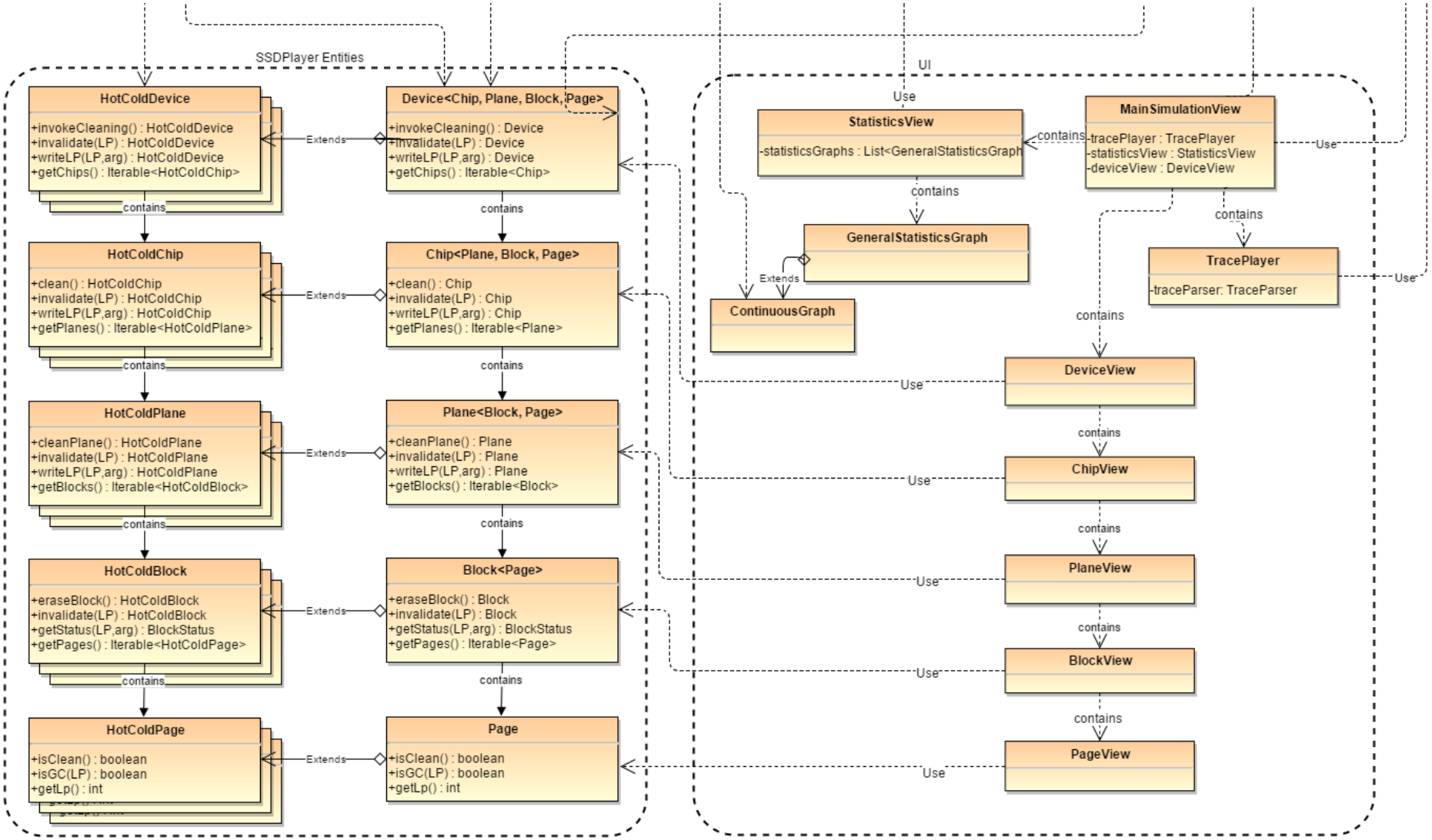
# 3. Creating an executable jar File

To create a jar file with IntelliJ, use the provided pom.xml file, and create a run configuration such as BUILD_JAR.xml. Then run the program. The jar file will be created in the “target” directory.

# 4. Architecture

## 4.1. Class Diagram



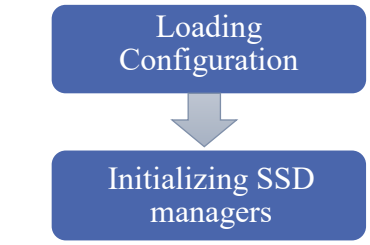

SSDPlayer Entities
HotColdDevice
+invokeCleaning() : HotColdDevice
+invalidate(LP) : HotColdDevice
+writeLP(LP,arg) : HotColdDevice
+getChips() : Iterable<HotColdChip>
Extends
Device<Chip, Plane, Block, Page>
+invokeCleaning() : Device
+invalidate(LP) : Device
+writeLP(LP,arg) : Device
+getChips() : Iterable<Chip>
contains
HotColdChip
+clean() : HotColdChip
+invalidate(LP) : HotColdChip
+writeLP(LP,arg) : HotColdChip
+getPlanes() : Iterable<HotColdPlane>
Extends
Chip<Plane, Block, Page>
+clean() : Chip
+invalidate(LP) : Chip
+writeLP(LP,arg) : Chip
+getPlanes() : Iterable<Plane>
contains
HotColdPlane
+cleanPlane() : HotColdPlane
+invalidate(LP) : HotColdPlane
+writeLP(LP,arg) : HotColdPlane
+getBlocks() : Iterable<HotColdBlock>
Extends
Plane<Block, Page>
+cleanPlane() : Plane
+invalidate(LP) : Plane
+writeLP(LP,arg) : Plane
+getBlocks() : Iterable<Block>
contains
HotColdBlock
+eraseBlock() : HotColdBlock
+invalidate(LP) : HotColdBlock
+getStatus(LP,arg) : BlockStatus
+getPages() : Iterable<HotColdPage>
Extends
Block<Page>
+eraseBlock() : Block
+invalidate(LP) : Block
+getStatus(LP,arg) : BlockStatus
+getPages() : Iterable<Page>
contains
HotColdPage
+isClean() : boolean
+isGC(LP) : boolean
+getLp() : int
Extends
Page
+isClean() : boolean
+isGC(LP) : boolean
+getLp() : int
UI
Use
StatisticsView
-statisticsGraphs : List<GeneralStatisticsGraph
contains
MainSimulationView
-tracePlayer : TracePlayer
-statisticsView : StatisticsView
-deviceView : DeviceView
Use
contains
GeneralStatisticsGraph
Extends
ContinuousGraph
TracePlayer
-traceParser: TraceParser
Use
contains
DeviceView
Use
contains
ChipView
Use
contains
PlaneView
Use
contains
BlockView
Use
contains
PageView
Use



7

## 4.2. Basic Workflow

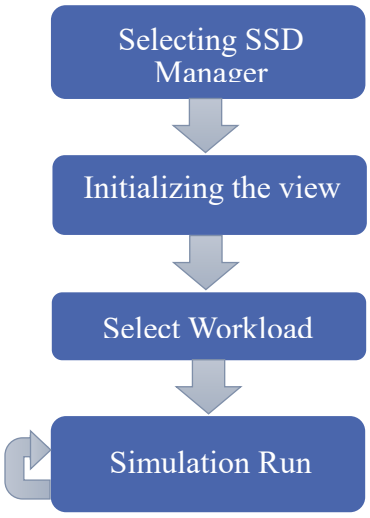
SSDPlayer Loading Process:




- Parsing the command line arguments (in CLI mode)
- Opening ssd_config.xml

- Creating all the Simulation SSD managers using reflectrion.
- Creating all the Visualization SSD managers using reflectrion
- Each manager is reading its configuration from the config reader

Simulation workflow:




- Selecting either simulation or visualization manager from a list

- Get the *TraceParser* from the selected manager
- Get the *StatisticsGetters* from the selected manager
- Get the *WorkloadGenerators* from the selected manager

- Either one of the *WorkloadGenerators is selected*
- Or an input file is loaded

- *TraceParser* is reading and parsing a line from the trace.
- Selected *SSD manager* is writing the LP on the current *Device* and sending the new *Device* it to the view.
- *StatisticsGraphs* are updated with latest statistics.

## 4.3. General Entities

In core of SSDPlayer are 5 entities (in the package entities): *Page, Block, Plane, Chip,* and *Device.*

We designed these five abstract classes to facilitate the development of new use case FTL managers for SSDPlayer. Each abstract class has unique fields and methods suitable for the entity it represents.

When an operation needs to be performed on a page (for example, invalidate page), the manager will call a method (invalidate page) in the Device. The Device will call a method in the correct Chip object, which will call a method in the correct Plane object, which will call a method in the correct Block object, and then at last the operation will take place in the correct Page.

These abstract classes support the builder pattern, for easy creation of new objects. Thus, you will have to implement a builder class if you wish to extend one of these entities.

### 4.3.1. Adding a new FTL use case

You can either use the existing entities with a new use case SSD manager, or implement a new use case SSD manager, with new entities. Adding new use case manager is possible by extending the *SSDManager* and then implementing methods like getTraceParser and initStatisticsGetters. Implementing new entities for your use case is possible by extending *Page, Block, Plane, Chip,* and *Device.*

## 4.4. RAID Entities

All the RAID use case SSD managers extend the *RAIDSSDManager* class (in the package *entities*).

The *RAIDSSDManager* class is based on RAID entities (in the package *entities.RAID*): *RAIDPage, RAIDBlock, RAIDPlane, RAIDChip,* and *RAIDDevice.*

RAID entities classes extend the matching abstract entities explained above, and are shared between all the RAID managers (except *RAIDDevice,* as detailed below).

*RAIDDevice* is the most important class in the *RAIDSSDManager*. It takes care of writing the data pages to the device and updates the matching parity pages.

Each RAID manager (RAID 1, RAID 5 and RAID 6) is composed of two classes:

1. Manager class (in the package *manager)* that extends the *RAIDSSDManager* class.
2. Device class (in the package *entities.RAID)* that extends the class RAIDDevice from the package *entities.RAID*.

For example, the RAID 1 manager is composed of the classes RAID1SSDManager and RAID1Device.

### 4.4.1. Adding a new RAID use case

As explained above, you will have to create your own Manager class and Device class. The methods you will have to implement in each class are as follows.
In the Manager class:

- *void setParitiesNumber()* – determines the number of parities in a stripe.
- *void setStripeSize()* – determines the stripe's size.
- *RAIDDevice getEmptyDevice(List<RAIDChip>)* – returns a new, empty device. You can look at the existing methods in RAID 1/RAID 5/RAID 6 managers for more help.

In the Device class:

- *boolean parityNeedUpdate(int lp, int parityNumber)* – determines whether a parity page need to be updated (Usually it's true for every parity).
- *int getChipIndex(int lp)* – returns the chip index of a logical page.
- *int getPageStripe(int lp)* – returns the stripe of a logical page.
- *int getParityChipIndex(int lp, int parityNumber)* – returns the parity chip index of the parity *parityNumber* of the logical page *lp*.
- *Builder getSelfBuilder()* – returns a new Builder of the Device.

In addition to creating these two classes, you will have to add the related fields in the configuration file:

1. name
2. clean_color
3. parity_color (maybe more than one field)
4. data_color
5. stripe_frame_color
6. stripe_frame_step

For a more detailed description of these fields, please consult the SSDPlayer User's Guide.

# 5. Features

## 5.1. Statistics Getters

Statistics are gathered and displayed throughout the simulation and the visualization runs. *StatisticsGetter* is the interface for returning statistics information given an SSD device, as a *List<StatisticsColumn>.*

The basic *StatisticsGetter* is meant to collect statistics for the basic device type. To collect and access statistics information for specific devices, use casting when passing the requested device as input. See, for example, the *HotColdStatisticsGetter* which handles a HotCold Device.

Each *SSDManager* contains a list of statistics getters that will be displayed as histograms during the simulation run. *StatisticsGetter* also specifies the type of graph (histogram) to be displayed by extending *GeneralStatisticsGraph*.

### 5.1.1. Adding a Statistics Getter

Follow these steps to add a new statistics view to a new or existing SSDManager:

1. Extend the class *StatisticsGetter*. You will have to implement the following functions:
   - *int getNumberOfColumns():* returns the number of lines or columns to be presented in the graph. This property cannot change dynamically, and will affect the way the graph is formatted during initialization.
   - *GeneralStatisticsGraph getStatisticsGraph()*: returns a graph (histogram) to be displayed. This function is called once before the simulation/visualization start running.
   - *List<StatisticsColumn> getStatistics(Device device)*: this function will be called whenever the state of the device changes, as long as the simulation/visualization is running.
   - *Entry<String, String> getInfoEntry(Device device)*: This is an entry for the info window.
2. Add your new custom statistics to the method *initStatisticsGetters* in the *SSDManager* you are modifying.

## 5.2. Trace Parsers

The trace parser is the functionality responsible for converting the input file into operations handled by the *SSDManager*. The *FileTraceParser* reads data from a specifically structured text file (file extension can be specified). Every SSDManager, either for simulation or visualization, must have a *FileTraceParser* that is responsible for generating input commands with the required fields.

### 5.2.1. Adding a File Trace Parser

Each *SSDManager* has one *FileTraceParser*. If you add new *SSDManager* that expects a workload with new parameters (that are not handled by any existing *FileTraceParser*), you will need to add a new *FileTraceParser*. To do, follow these steps:

1. *FileTraceParser* is generic and is configured using *SSDManager* and *Device* types. You will have to extend *FileTraceParser* and implement the following functions:
   - *String getFileExtensions():* Specify file extensions expected by the parser.

- *Device parseCommand(String command, int lineNo, Device device, SSDManager manager):* The main parsing method which receives an input line, a manager, and a device to run the operation on. It parse the input line and generates an I/O command (operation), runs it by the manager and returns the new, possibly modified, device.

2. Override the *getFileTraseParser* method in your new SSDManager and return the new *FileTraceParser.*

## 5.3. Workload Generators

The workload generator is a method for generating synthetic input traces for a simulation run from within SSDPlayer. The *WorkloadGenerator* generates I/O commands (operations) using the distribution of your choice. For example, the *ZipfWorkloadGenerator* generates write operations which are distributed across the device's pages according to a Zipf distribution. A *WorkloadWidget* creates an entry for the generator, where the user can choose the parameters of the workload, such as length, request sizes, random seed, etc.

Note that this type of input is only applicable for simulation *SSDManager*s.

### 5.3.1. Adding a Workload Generator

To add a new Workload Generator, follow these steps:

1. Extend *WorkloadGenerator* and implement:
   - *int getLP():* get logical page for next write.
   - *int getLPArg(int lp):* you can also override this method if you need to supply some argument for the write of the current logical page.
2. Extend *WorkloadWidget:* add fields for user's input (by calling *super.addField(Component input, String label)*) and implement:
   - *WorkloadGenerator createWorkloadGenerator():* which will return your new custom workload generator.
3. Add your new workload widget to the workload widgets list in method *getWorkLoadGeneratorWidgets()* in the *SSDManager* of your choice.

## 5.4. Breakpoints

Breakpoints allow users to pause the simulation at interesting points in which a specific state or condition are of interest.
The list of active break points is loaded from ssd_breakpoints.xml or added by the user using the dialog box. After parsing a command and updating the device state, the SSDplayer checks for breakpoint hits and if a hit occurs it pauses the simulation.
For example, to check for a "Clean Blocks in Chip" breakpoint hit, we check the previous clean block count in the chip using previous device state and the new clean block count in the chip using new device state and return true if clean block count has changed and now is equal to breakpoint value.

### 5.4.1. Adding a breakpoint

1. Extend the class *BreakpointBase*. You will have to implement the following functions:

- *boolean breakpointHit(Device previousDevice, Device currentDevice):* Given previous and current device state return true if there is a breakpoint hit.
- *String getDisplayName()*: Return the name of the breakpoint
- *String getDescription()*: Return description of the breakpoint including its parameters.
- *void addComponents()*: Adding UI components to the breakpoint edit dialog.
- Parameters getters and setters.
- *Boolean isEquals(IBreakpoint)*: compare two breakpoints.
- *String getHitDescription()*: return description on break point hit.
- *boolean isManagerSupported(SSDManager)*: Given an SSDManager return *true* if the breakpoint supports this manager.

2. Add an example to resources/ssd_breakpoints.xml. This file contains predefined breakpoints. Its structure is:

```
<root>
    <breakpoints>
        <breakpoint type={breakpoint class name}>
            <{parameter1 name}> value </ {parameter1 name} >
            < {parameter2 name}> value </ {parameter2 name}>
            < {parameter3 name}>value</ {parameter3 name}>
        </breakpoint>
</root>
```

## 5.5. Zoom Levels

Zoom levels allow users to change the level of details presented on the device, in order to view larger devices entirely on the display.

Each manager defines the list of zoom levels it supports. The supported zoom level is shown in the Zoom dialog and pressing OK applies the zoom level. The Zoom levels listed in the dialog are grouped according to the defined zoom level group and ordered according to the order they were added to the manager.

### 5.5.1. Existing zoom levels:

1. Detailed - standard zoom level. Page numbers and counters are displayed, written pages are colored and deleted pages are crossed.
2. Pages - counters are removed, the sizes of pages and the spacing between them is reduced to half of the original. Invalid pages are marked with a thin line instead of the current bold cross, and the “moved” page pattern is converted to a lighter shade of the original page.
3. Blocks - pages are no longer visible, the color of the block represents the state of its pages. Color meanings:
   - Valid count – darker color represents more valid pages.
   - Erase count – darker color represents “older” blocks.
   - Average temperature - average temperature of pages (for the HotCold manager).
   - Average write level - average write level of pages (for the Reusable manager).
   - Small Blocks - Same as Blocks, but blocks are smaller so that more can fit on the screen. Color meanings are the same as in Blocks level.

### 5.5.2. Adding a new Zoom Level

4. Create a class that implements ZoomLevel.
    - String getName(): returns the name of the zoom level.
    - String getGroup(): returns the group that the zoom level belongs to (or null if it doesn't belong to any).
    - applyZoom(SSDManager, VisualConfig): the function is called upon pressing the OK button in the zoom level dialog. The function should change the appropriate parameters in the visual configuration passed to it.
5. Add the zoom level to the list of supported zoom levels of the desired managers.

## 5.6. Info screen

The info screen enables detailed examination of the simulation state whenever it is paused (due to user action or breakpoint hit). The info screen presents statistics and information about the simulation's entities, in an expandable list.

Each entity in the simulator (devices, chips, planes, blocks, pages and statistics getters) implements the method *public EntityInfo getInfo()*. This method returns an *EntityInfo* which contains information about the entity in the form of ordered key-value pairs.

The *InfoDialog* is initialized at startup with the device configuration. This creates the *JTree* (the model used for the expandable list in the GUI) which represents the device and its entities.

Whenever the device is updated, the *TraceParser* invokes *InfoDialog.setDevice(Device<?, ?, ?, ?> currentDevice, int currFrameCounter).* This method keeps the current state of the device updated in the dialog. When the user clicks the info-screen button, the dialog is made visible. When the user chooses an item in the expandable list, the function *InfoDialog.setEntityInfo(TreePath selectedNodePath)* is called. In this function, the entity that is represented by the item in the expandable list is found, the entity's *getInfo()* function is called, and the result *EntityInfo* is displayed in the right-hand side of the dialog.

To add or change the information displayed about an entity, you should override or edit its *getInfo()* method. You can add new information by using the method *EntityInfo.add(String desc, String value, int order)* adding a new key-value pair. The *order* parameter determines the order of the new key-value pair in the *EntityInfo*. Pairs are ordered in increasing order according to their *order*, breaking ties according to addition order (the pair that was added first will be displayed first).

## 5.7. Error messages

This part of the code enables the programmer to display messages to the user in the error-log window located in the lower right-hand side of the screen.

In order to display a message in the error log you can use the method *MessageLog.log (Message message)*. There are currently three types of messages in the simulator:

- ErrorMessage – informs the user about an error in the simulator. The error text is colored in red.
- InfoMessage – displays general information about the simulator run. The message text is colored in blue.
- BreakpointMessage – indicates that the simulator run was paused because of a breakpoint hit. The message text is colored in yellow.

Error messages can also be displayed before the simulator is loaded, in case an error prevents the simulation from starting. This kind of message is used in order to inform the user of an error that occurred during initialization.

In order to show a dialog containing an error message use the *displayErrorFrame(String string)* method in the *MainSimulationView* class.

You can add a new type of message by extending the Message type.

## 5.8. Trace Player

The trace player is the functionality responsible for executing the commands of the chosen trace when in GUI mode. It is called from the main simulation view after it is done initializing. It then opens the trace file or uses a generator. It parses the commands one-by-one, executes them, and updates the UI if necessary. It is also responsible of presenting the statistics and histograms to the user and listening for requests such as pause/play/stop trace, open new trace, generate new workload, manage breakpoints, next frame, zoom level, info, sample view. The most important method is *parseNextCommand*, on which the program iterates until it finishes parsing all the commands in the input file or that were generated.

## 5.9. Sampling rate

The sample view allows you to choose the rate at which the simulation display is updated. That is, if the sampling rate is *X*, then the display will update itself after every frame whose frame number can be divided by *X*. TracePlayer takes the value of *X* into consideration in the parseNextCommand method, and only updates the display if the current frame number can be divided by *X*.

## 5.10. CLI mode

If the program is run form the command line, and it is given command line parameters, the program will run in CLI mode. The parameters are supposed to be in one of the following formats:

1. -C <config file name> -M <manager Name> -F <trace file name>.<trace file extension> -O <output file name>

2. -C <config file name> -M <manager Name> -G -U <workload length> <seed> -O <output file name>

3. -C <config file name> -M <manager Name> -G -U <workload length> <seed> <max write size> <is write size uniform> -O <output file name>

4. -C <config file name> -M <manager Name> -G -Z <workload length> <seed> <exponent> -O <output file name>

5. -C <config file name> -M <manager Name> -G -Z <workload length> <seed> <exponent> <max write size> <is write size uniform> -O <output file name>

The program parses the parameters and passes them together with the visual configuration to *MainCLI*, which calls *TracePlayerCLI* in order to execute the trace commands without any display. *TracePlayerCLI* works just like *TracePlayer*, but simpler, since it doesn't take care of updating the display and having interactive windows with the user.

# 6. Important Classes

## 6.1. SSDManager<P, B, T, C, D> - Package manager, Extends java.lang.Object

Generic Type Parameters: P - Page, B - Block, T - Plane, C - Chip, D – Device

### *General description*

SSDManager is an abstract base class for every FTL use case. All the non-abstract subclasses of this class will be loaded using *Reflections*[1] library. The UI classes use the static methods of this class to get all the possible use cases in the simulator. This class includes static methods like getManagerByName, getAllManagerNames etc. The static members are: *simulatorsList* - list of names of the simulation SSD managers.
*visualisationsList* - list of names of the visualization SSD managers.
*managersMap* - holding all of the managers by their name (specified in the config file)

The non-static part presents the interface which every use case SSDManager will have to implement. Methods like getTraceParser, initStatisticsGetters, etc. The type is Generic in terms of the entities it uses so there will be static typing as strict as possible. Every SSDManager is defined to use a very specific set of entities, which will be defined with the implementation of the use case.

There is an abstract sub type of the SSDManager for the **visualization** managers. There is not much difference to them, but the **visualization** managers are harder to generalize because they are more tailor made for the specific a specific input. Needless to say that the trace parser for **visualization** managers will be more complicated, also the manager will need more access methods than just *writeLp*, for example see *ReusableVisualizationSSDManager*.

### *Method Details*

#### # initializeManager

***static*** *void initializeManager(XMLGetter xmlGetter)*

Initialize SSD manager using given configuration. First loads all of the subclasses of the SSDManager. After creates instance of the non-abstract ones, calls their initialization method and adds them to the managersMap, by their name (specified in the config file).

**Parameters:** xmlGetter - - configuration getter

#### # getManagerByName

*public* ***static*** *SSDManager<?,?,?,?,?> getManager(java.lang.String managerName)*

**Returns:** the use case manager with the specified name, or null if not found

#### # getAllSimulationManagerNames

***static*** *java.lang.Iterable<java.lang.String> getAllSimulationManagerNames()*

**Returns:** all the use case simulation managers

#### # getAllVisualizationManagerNames

***static*** *java.lang.Iterable<java.lang.String> getAllVisualizationManagerNames()*

**Returns:** all the visualization managers

[1] *https://code.google.com/p/reflections/* Open source library, WTFPL license. Reflections scans *classpath*, indexes the metadata, allows to query it on runtime and may save and collect information for modules within a project.

# getAllVisualizationManagerNames
***static*** *java.lang.Iterable<java.lang.String> getAllVisualizationManagerNames()*

**Returns:** all the visualization managers

# getTraceParser
***abstract*** *TraceParserGeneral<D,? extends SSDManager<P,B,T,C,D>> getTraceParser()*

**Returns:** get trace parser for this manager

# getEmptyPage
***abstract*** *P getEmptyPage()*

**Returns:** empty page for device initializing.

# getLpRange
*int getLpRange()*

Calculates and returns the logical page addresses range of the device. Calculates using the physical sizes and the Over-Provisioning.

**Returns:** the size of the device in pages

# getManagerName
*java.lang.String getManagerName()*

**Returns:** name of the SSDManager specified in the configuration file.

# getOP
*double getOP()*

**Returns:** Over-Provisioning - specified in the configuration file

# getReserved
*int getReserved()*

**Returns:** number of blocks reserved for Over-Provisioning

# getGCT
*int getGCT()*

**Returns:** Garbage Collection threshold, specified in the configuration file as percent, returned in blocks number.

# getChipsNum
*int getChipsNum()*

**Returns:** Number of Chips in the Device, specified in the configuration file.

# getPlanesNum
*int getPlanesNum()*

**Returns:** Number of Planes in each Chip, specified in the configuration file.

# getBlocksNum
*int getBlocksNum()*

**Returns:** Number of Blocks in each Plane, specified in the configuration file.

# getPagesNum
*int getPagesNum()*

**Returns:** Number of Pages in each Block, specified in the configuration file.

# getCleanColor
*java.awt.Color getCleanColor()*

**Returns:** Color of a clean Page.

# getStatisticsGetters
*java.lang.Iterable<StatisticsGetter> getStatisticsGetters()*

Each use case manager initializes a list of statistics it would like to present.

**Returns:** Returns the statistics getters

# writeLP
*D writeLP(D device, int lp, int arg)*

This method simulates the normal write procedure in SSD device. In order to change the basic writing algorithm overload this method. This operation may invoke the GC as a side effect.

**Parameters:** *device* - - the device to write on**,** *lp* - -logical page to write

**Returns:** the new device after the write.

# getWorkLoadGeneratorWidgets
*java.util.List<ui.WorkloadWidget<D,SSDManager<P,B,T,C,D>>> getWorkLoadGeneratorWidgets()*

Each use case SSDManager may have specific workload generators it is applicable to. This is the method to overload in order to add workload generators of your liking.

**Returns:** List of Workload Generators applicable with current SSDManager.

## 6.2. StatisticsGetter - interface

### *General description*

This interface represents statistics gathered by SSD manager. Every gathered statistics type in this simulator implements this interface.

### *Method Details*

# getNumberOfColumns
*int getNumberOfColumns()*

The Graphs cannot dynamically change the number of columns or lines they are presenting. For that reason the statistics getter must explicitly state number of gathered columns or lines.

**Returns:** number of gathered columns or lines

# getStatistics
*java.util.List<entities.StatisticsColumn> getStatistics(entities.Device<?,?,?,?> device)*

This is the method in which the statistics getter calculates the statistics, for the current Device state.

**Parameters:** device - - to calculate the statistics on

**Returns:** list of statistics columns for the current Device's state.

# getStatisticsGraph
*ui.GeneralStatisticsGraph getStatisticsGraph()*

**Returns:** the widget to present gathered statistics.

## 6.3. Page - Package entities, Extends java.lang.Object

### *General description*

The smallest measure in SSDPlayer. It contains the basic info on the physical content of a page in the simulator: isClean, isValid, isGC, logical page.

This entity is built using a Builder class (Builder design pattern). Builder is a nested class of the Page and it allows to create a Page and validates that the initialization is correct. In order to extend the Page entity you will have to extend the Builder class as well (see HotColdPage).

The Page is immutable, so any change to an existing page will create a copy of the page with this change. This is done so it will be easy to save device state as a frame in a video.

### *Method Details*

#### # getBGColor

*public abstract java.awt.Color getBGColor()*

**Returns:** Background color of the page.

#### # getSelfBuilder

*public abstract Page.Builder getSelfBuilder()*

**Returns:** a Builder initialized to create a copy of this page.

#### # isClean

*public boolean isClean()*

**Returns:** whether the page is clean

#### # isGC

*public boolean isGC()*

**Returns:** whether the page was written as Garbage Collection write (copied from erased block)

#### # getLp

*public int getLp()*

**Returns:** the Address of the Logical Page written in this page (Identifier of LP).

#### # isValid

*public boolean isValid()*

**Returns:** whether this page is valid.

#### # getTitle

*public java.lang.String getTitle()*

**Returns:** Formats the string to display on this page (Usually just the LP Address).

#### # getPageTexture

*public java.awt.TexturePaint getPageTexture(java.awt.Color color)*

**Returns:** Page texture (different textures may be displayed in the simulator).

## 6.4. Block <P> - Package entities, Extends java.lang.Object

Generic Type Parameters: P – Page type the block stores

### *General description*

Block basically is ordered collection of pages, but this entity stores more information like eraseCounter, block status, etc.

Block is also immutable and has a builder same as Page.

### *Method Details*

#### # getSelfBuilder

*public abstract Block.Builder getSelfBuilder()*

**Returns:** a Builder initialized to create a copy of this block.

#### # getPages

*public java.lang.Iterable<P> getPages()*

**Returns:** iterable pages in this block, this is done in order to avoid changes in the collection outside the block, we regard iterable as immutable.

#### # getStatus

*public entities.BlockStatus getStatus()*

**Returns:** block status. *BlockStatus* is an interface which can be implemented in order to extend the number of block statuses in specific use case.

#### # getEraseCounter

*public int getEraseCounter()*

**Returns:** Erase Counter of this Block, the block is updating this field on a copy after erase.

#### # getPage

*public P getPage(int i)*

**Returns:** Page in the index i. Throws IllegalArgument for invalid index.

#### # isInGC

*public boolean isInGC()*

**Returns:** This is for visualization mode, it returns whether the block is moved by GC currently, in simulation mode this happens in between frames.

#### # getValidCounter

*public int getValidCounter()*

**Returns:** Number of valid Pages in the block, aggregated data.

#### # getNewPagesList

*public java.util.List<P> getNewPagesList()*

**Returns:** New list of the same pages, this is a utility method for copying a block. A new list is created after an edited page is inserted and an edited copy of the block can be produced by a Builder.

#### # getCleanPageIndex

*public int getCleanPageIndex()*

**Returns:** Index of the next clean page in the block, returns -1 if none exists.

#### # getBGColor

*public java.awt.Color getBGColor()*

**Returns:** Background color of the block.

#### # getStatusName

*public java.lang.String getStatusName()*

**Returns:** Format of the block status string to be displayed. May contain status, counters, etc.

# getStatusColor
*public java.awt.Color getStatusColor()*

**Returns:** get color of the written block status color. Used in order to mark some states.

# invalidate
*public Block<P> invalidate(int lp)*

**Parameters:** lp - - Logical Page to be invalidated.

**Returns:** a new block in which the Logical Page given is invalidated, if doesn't contain the given LP returns itself (this). Creates a copy.

# eraseBlock
*public Block<P> eraseBlock()*

**Returns:** new clean block with updated erase counter. Creates a copy.

# hasRoomForWrite
*public boolean hasRoomForWrite()*

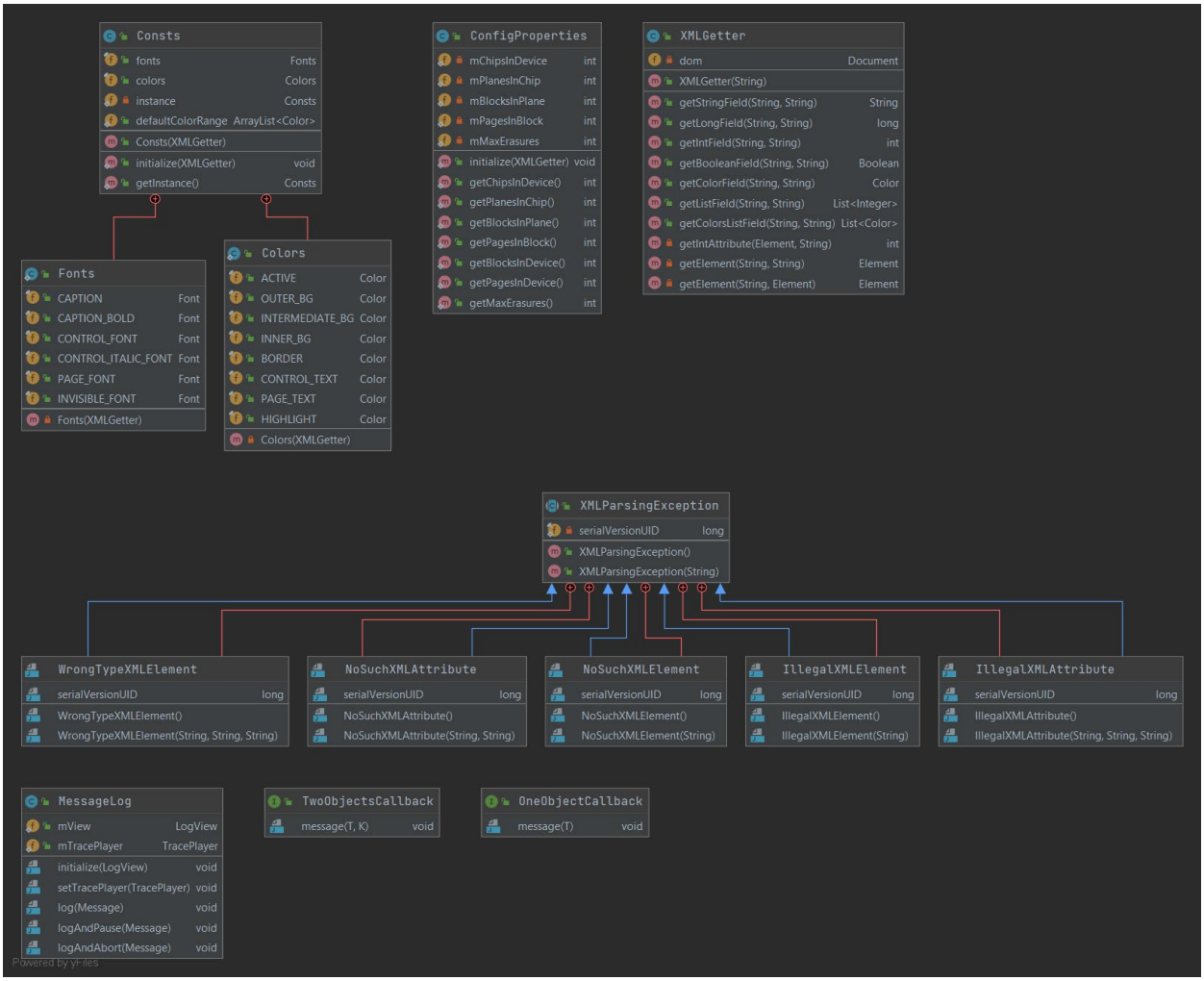
**Returns:** whether the block has free page to write on.

# setStatus
*public Block<P> setStatus(entities.BlockStatus status)*

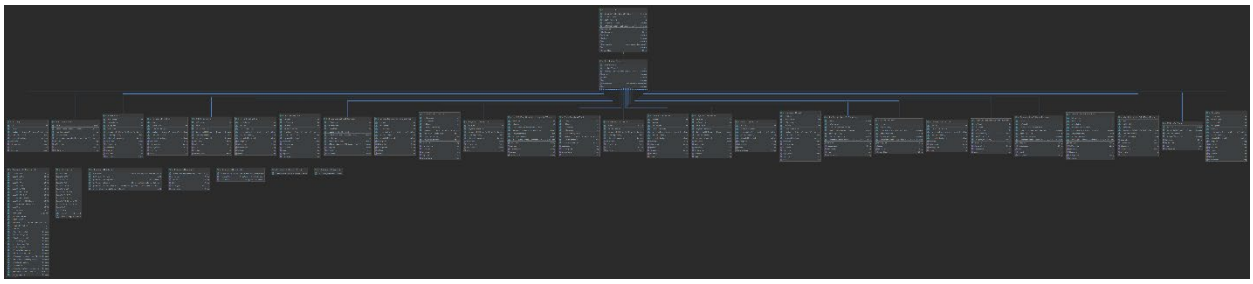
**Returns:** whether the block has free page to write on.

# 7. Appendix: package UMLs

## 7.1. General

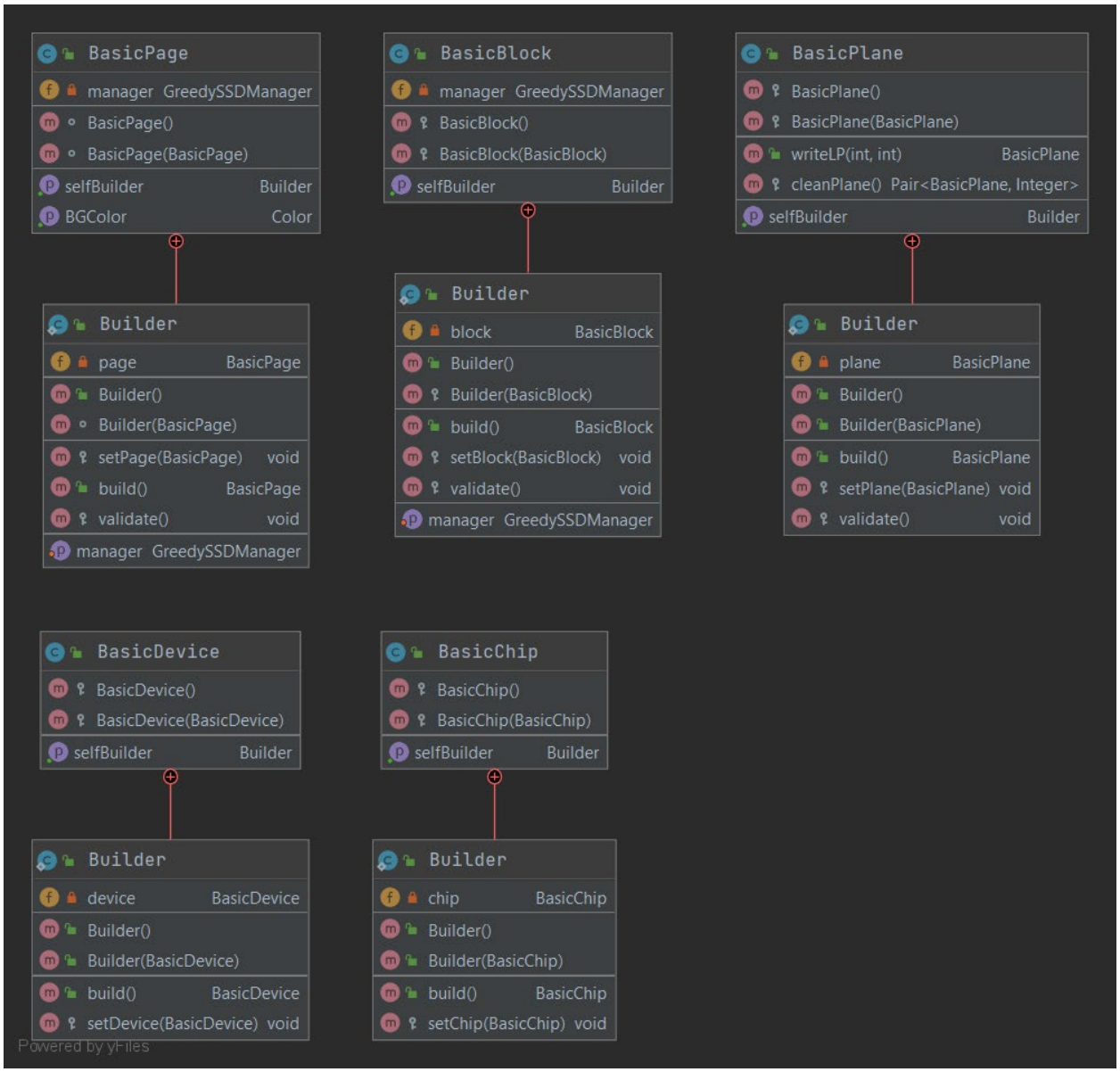



## 7.2. Breakpoints

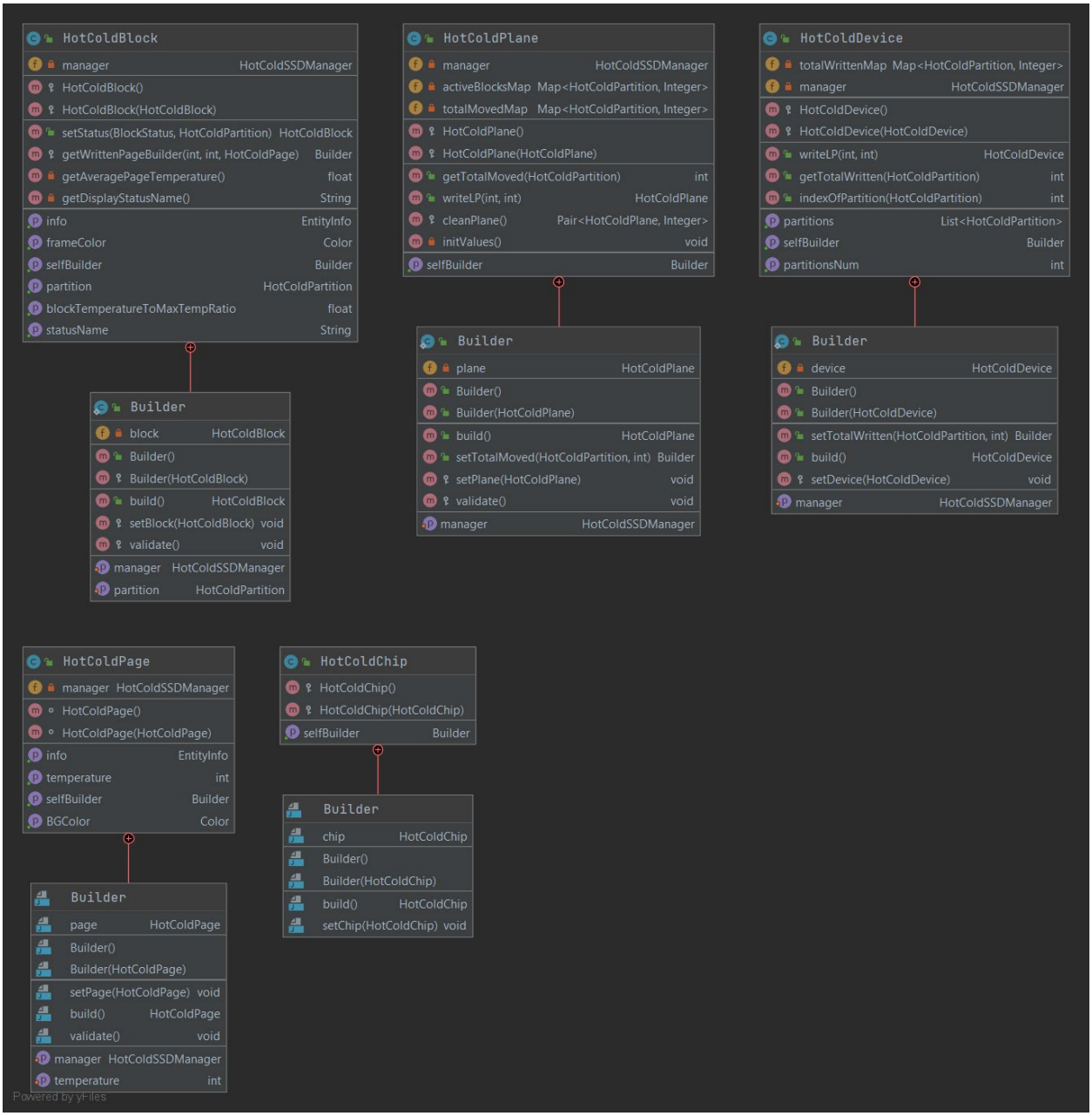


## 7.3. Entities

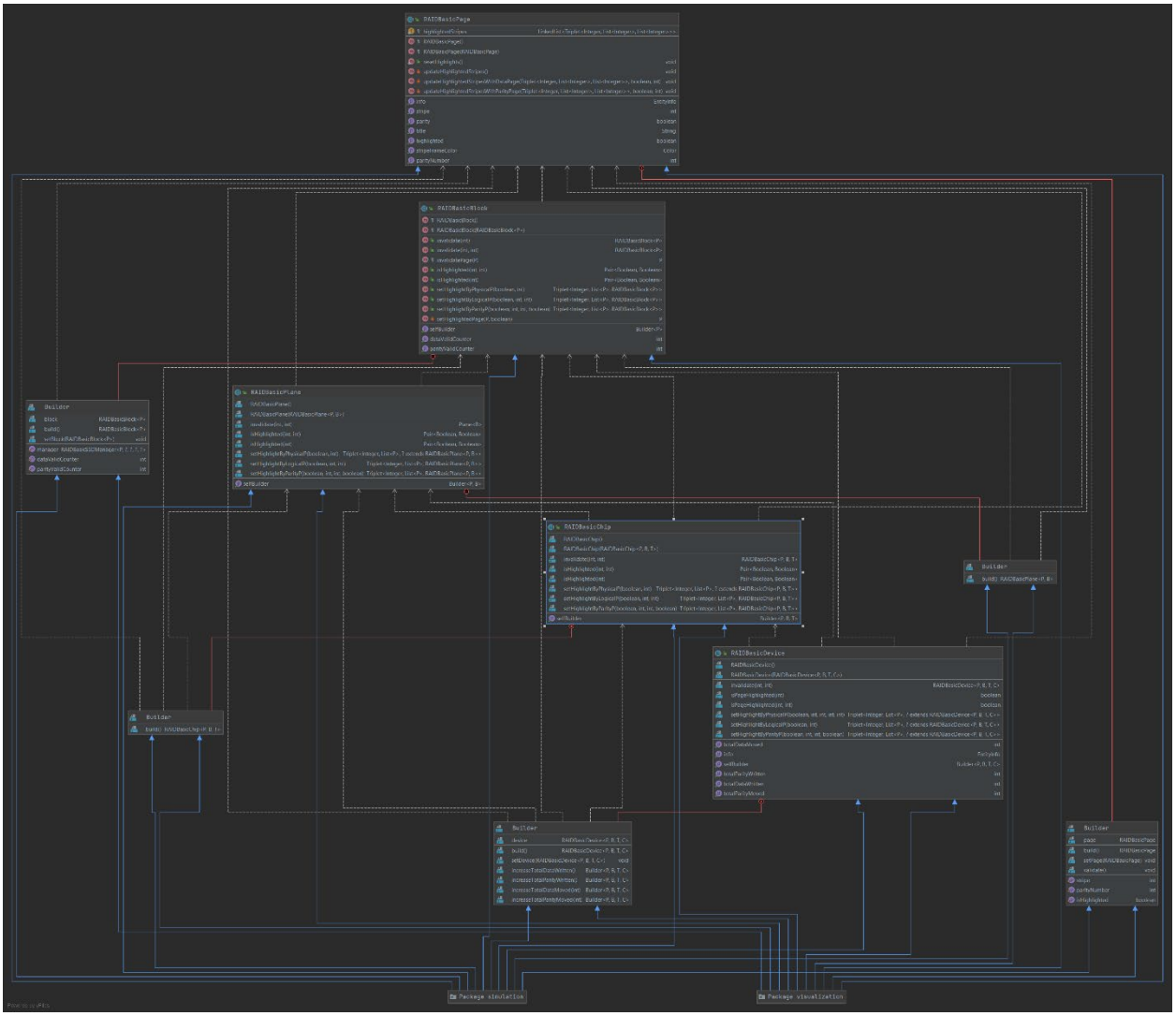
### 7.3.1. Entities basic



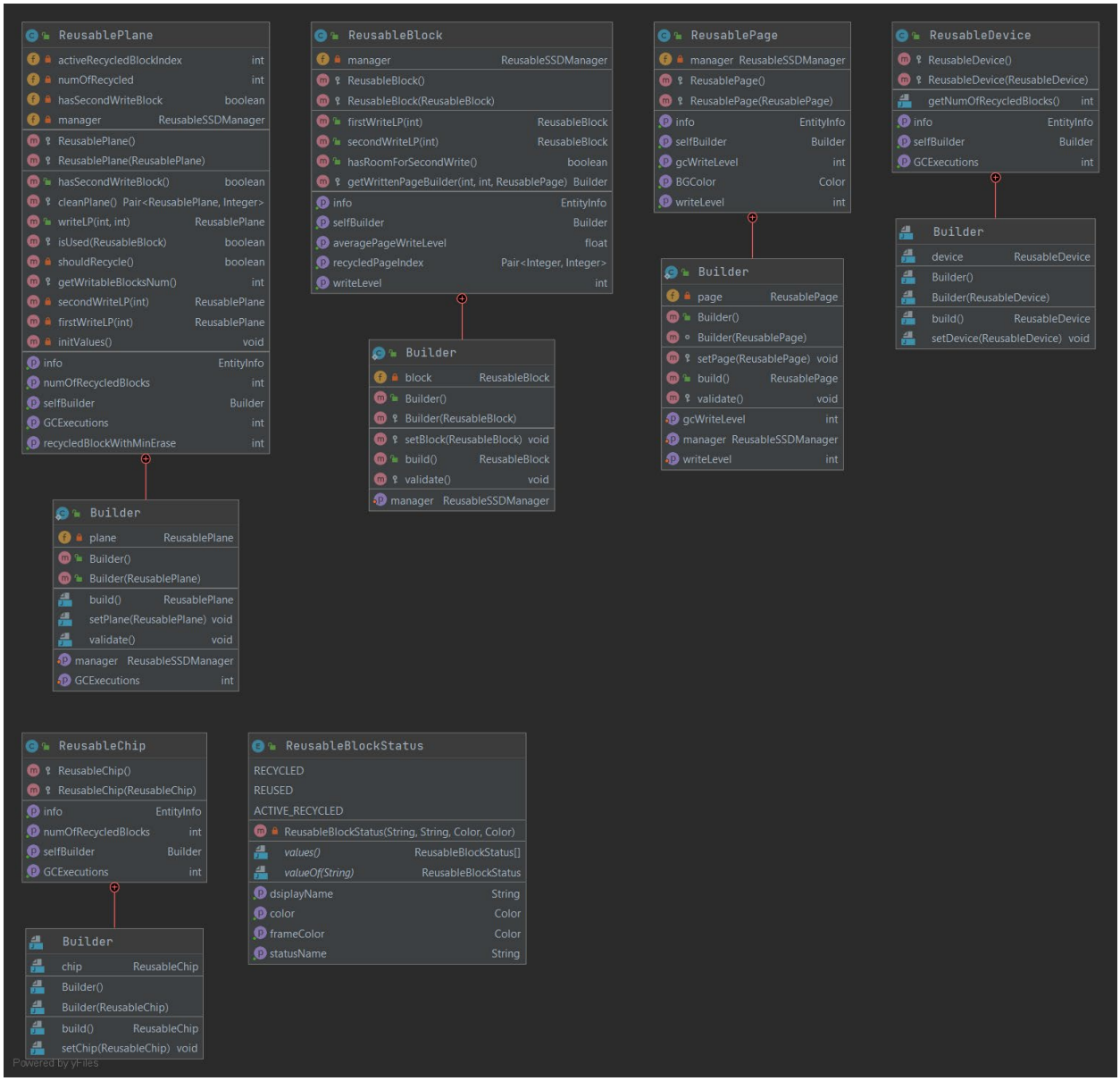
### 7.3.2. Entities hot_cold



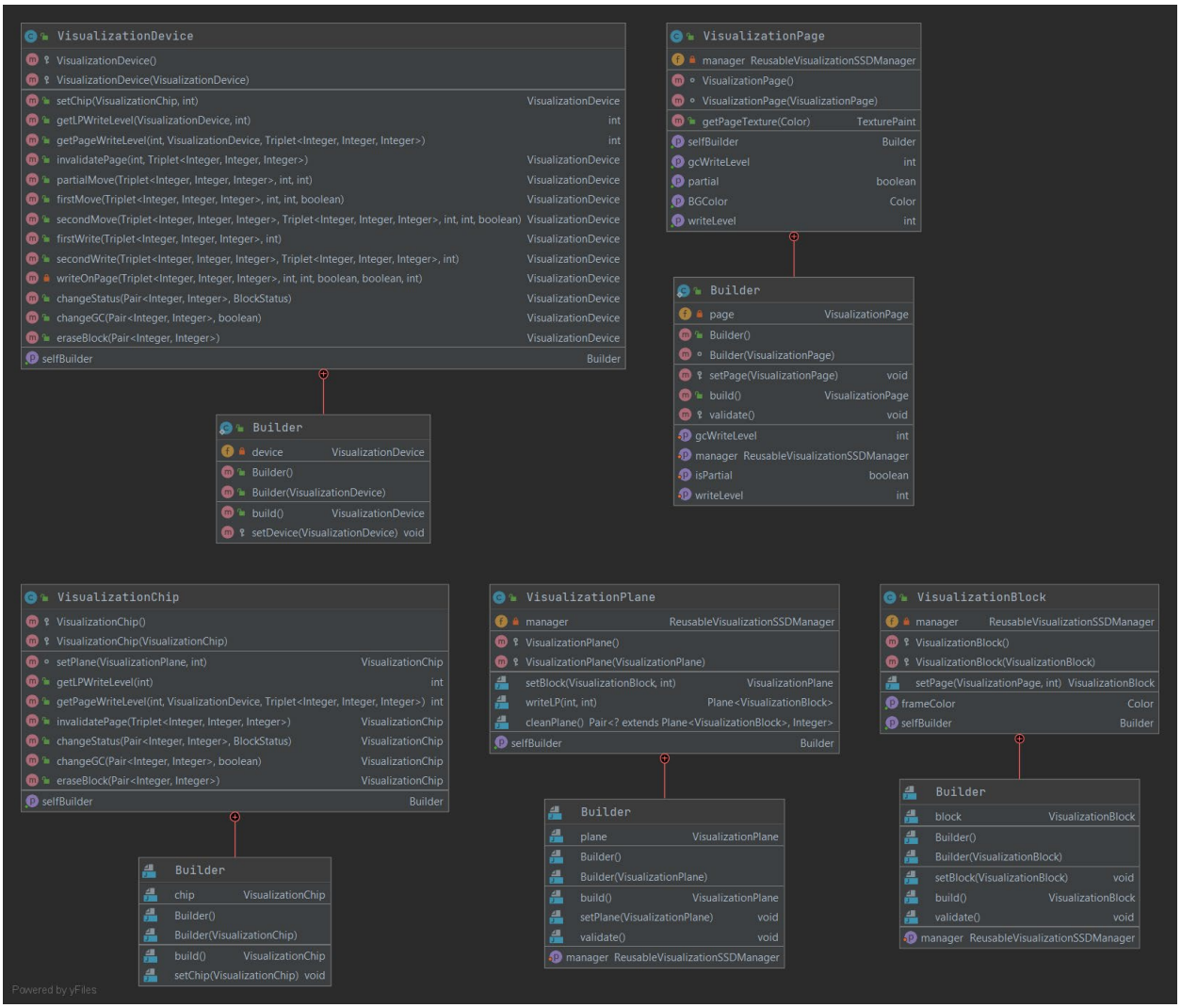
### 7.3.3. Entities RAID



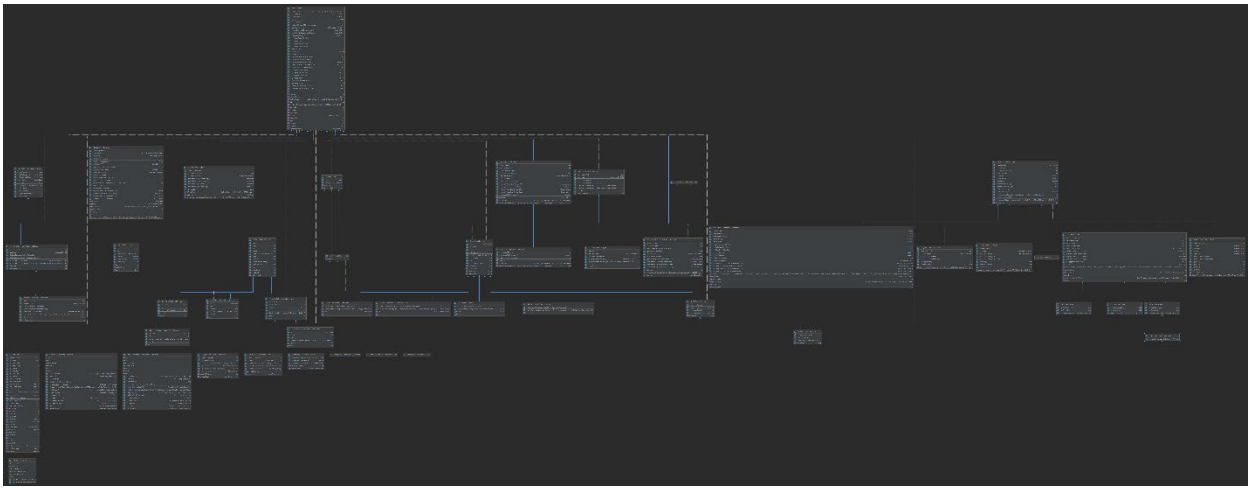
### 7.3.4. Entities reusable



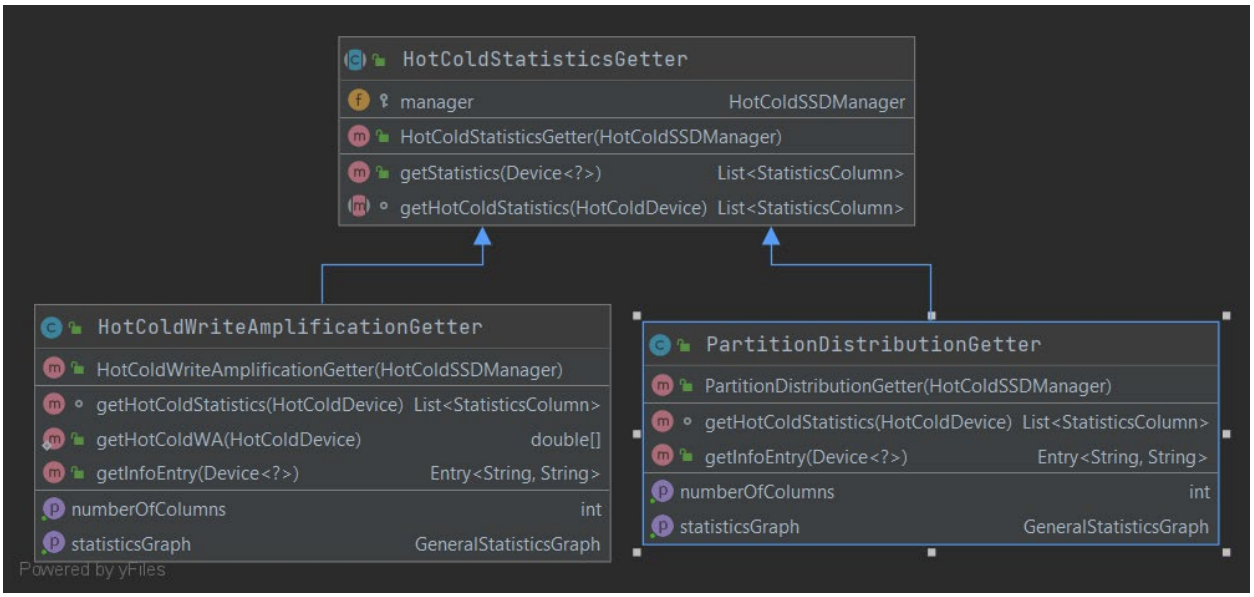
### 7.3.5. Entities reusable visualization



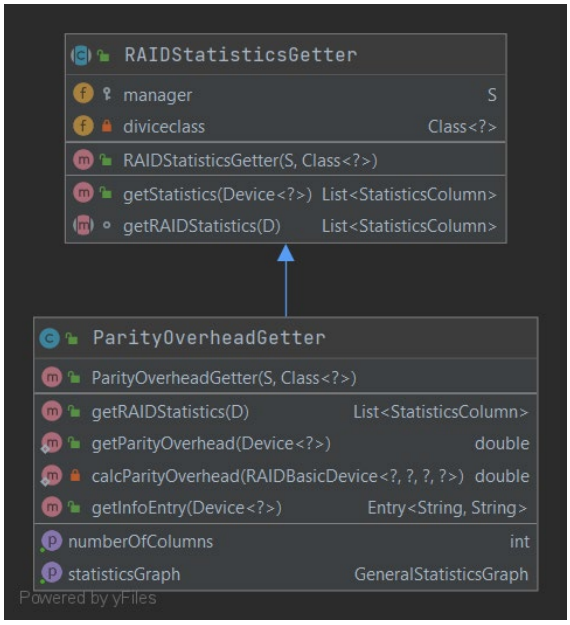
## 7.4. Manager



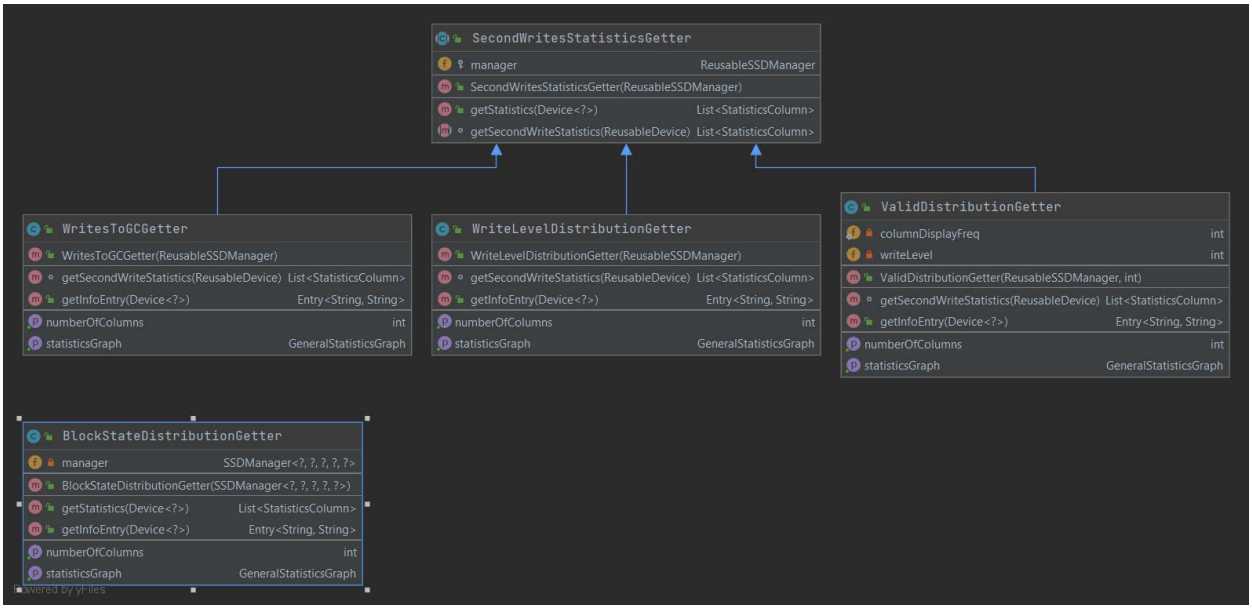
### 7.4.1. Manager HotColdStatistics



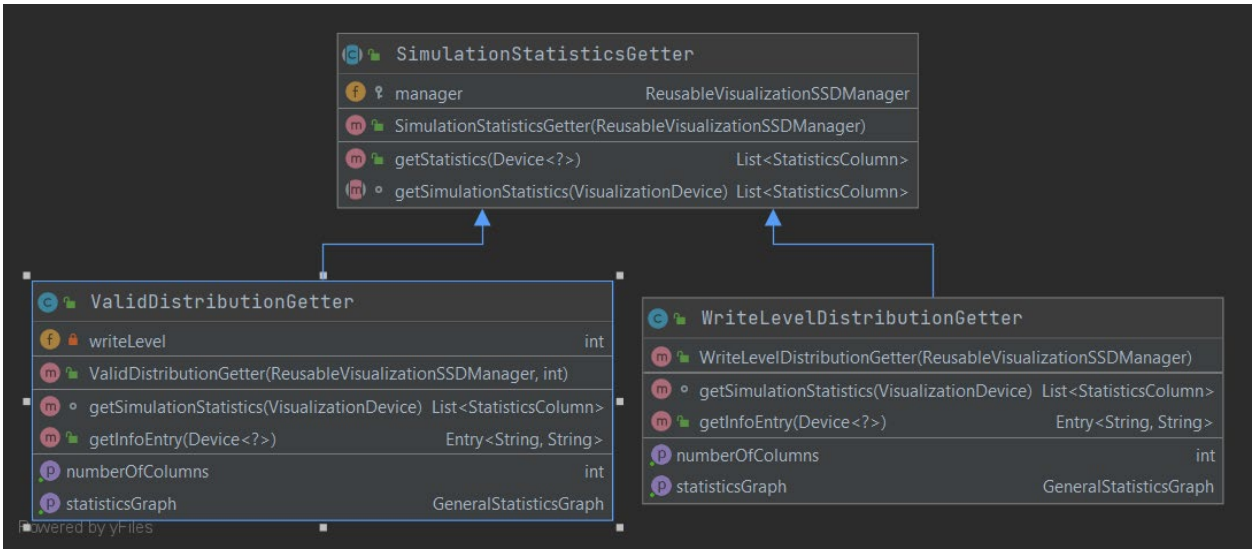
### 7.4.2. Manager RAIDStatistics



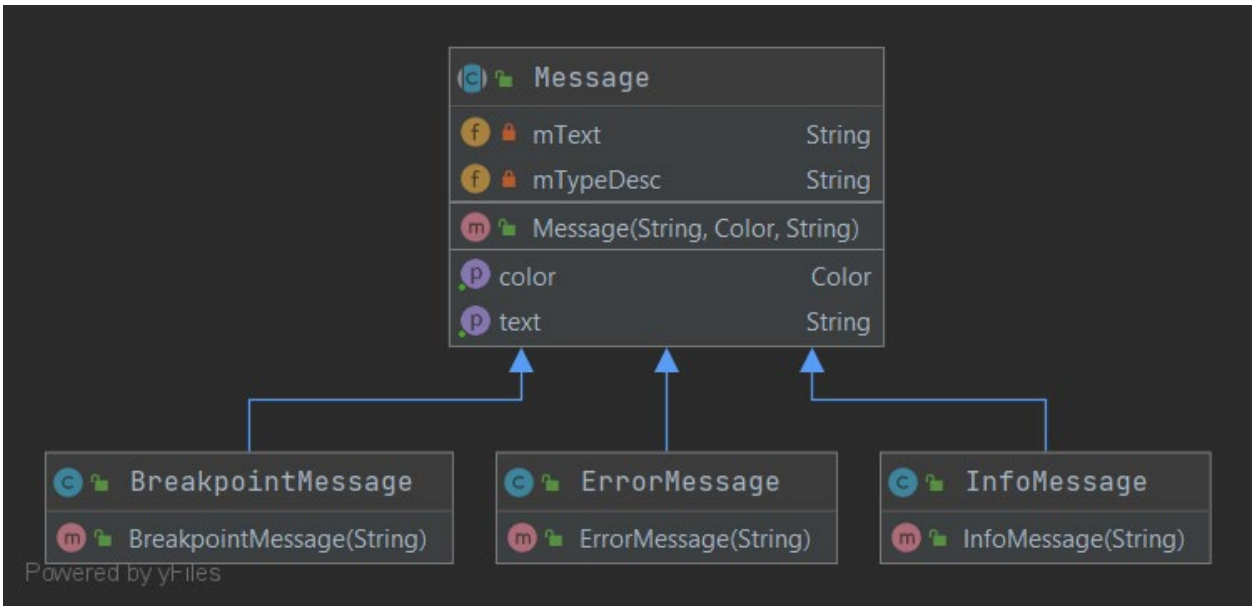

### 7.4.3. Manager SecondWriteStatistics



### 7.4.4. Manager SimulationStatistics



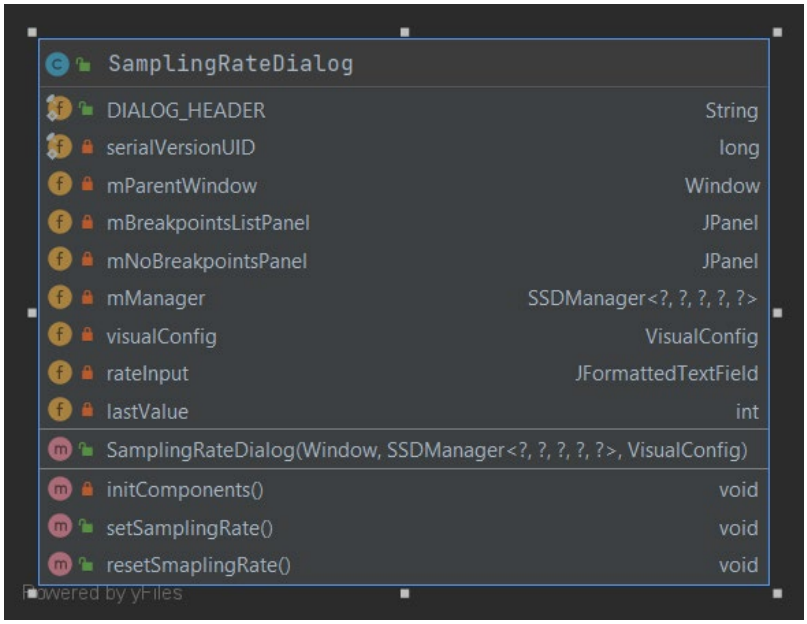

## 7.5. Message



## 7.6. UI

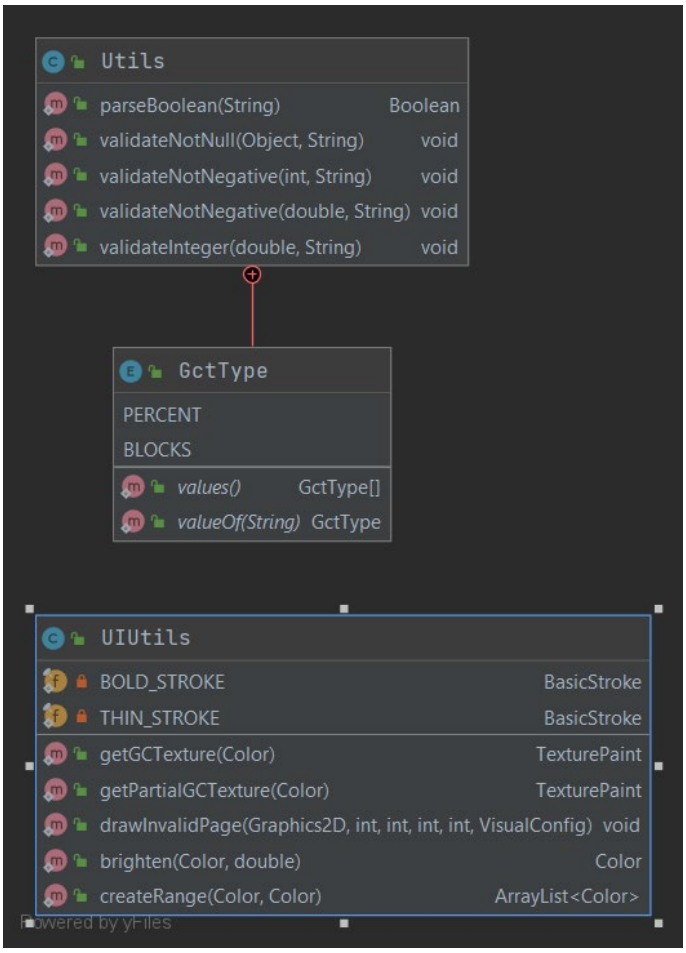
### 7.6.1. UI Zoom

### 7.6.2. UI Sampling



## 7.6.3. UI Breakpoints

## 7.7. Utils




## 7.8. Zoom