CS44800 Project 2: The RDBMS Lower-Level Layers

Due: 11:59 PM ET, Sunday, March 2, 2025. Submit using BrightSpace.
Total Points: 100
(There will be a 10% penalty for each late day. After 5 late days, the project will not be accepted.)

Notes:

- This project (and those to come) should be carried out in groups of three. Choose your partners as soon as possible.
- This project (and those to come) will be based on Minibase, a small relational DBMS, structured into several layers. In this project, you are required to implement parts of the lower-level layers: BM - the buffer manager layer and HFPage - the heap file layer. Layers should be tested and turned in separately.
- This is a relatively long hand-out but it is imperative that you read from top to bottom.

Learning Objectives

1. Understand the design and architecture of lower-level database systems components.
2. Implement the buffer manager component in a simplified database system.
3. Implement a replacement policy for the buffer management layer.
4. Implement Heap File structure and use it alongside the implantation of Heap Scans

PART 1: Buffer Management [50 Points]

1. Introduction

The goal of this section of the project is to implement a simplified version of a Buffer Manager layer, without support of concurrency control or recovery. For this assignment, you will be given the code for the lower layer (which is the Disk Manager layer).

This project is based on Minibase, a small relational DBMS, structured in several layers in order to allow a modular, abstract approach in implementing a DBMS. This approach allows for ease of implementation, as no layer relies on any specific implementation of any lower layer – they only have to make the appropriate calls to functions defined in an API. **The documentation is provided inside the *javadoc* directory in the provided skeleton code base.**

In this part of the project, you will be implementing one of the lower levels of a DBMS: the Buffer Manager level that is responsible for bringing pages into and out of main memory from the disk. The actual disk access functionality is already implemented for you in the Disk Manager layer of Minibase - the source files for the Disk Manager are included in the *diskmgr* package if you would like to investigate them (and we strongly encourage you do so). Do not make any changes to the *diskmgr* package.

You should begin by reading the chapter on Disks and Files to get an overview of buffer management. In addition, HTML documentation is available for Minibase. In particular, you should read the description of the DB class, which you will call extensively in this assignment. The Java documentation for the *diskmgr* package can be found inside the *javadoc* directory that has been provided in the zip file of the project. You should also read the code under diskmgr/ carefully to learn how the package is declared and how exceptions are handled in Minibase.

2. The Buffer Manager Interface

The simplified Buffer Manager interface that you will implement in this assignment allows a client (a higher level program that calls the Buffer Manager) to allocate/de-allocate pages on disk, to bring a disk page into the buffer pool and pin it, and to unpin a page in the buffer pool.

The methods that you have to implement are described below (and are also available in the comments in your skeleton codebase). Also, you can (and are encouraged to) create helper classes for the replacement policy defined later. However, any class you make should be under *bufmgr* package.

```
public class BufMgr {
/**
* Create the BufMgr object.
* Allocate pages (frames) for the buffer pool in main memory and
* make the buffer manage aware that the replacement policy is
* specified by replacerArg (e.g., FIFO, LH, Clock, LRU, MRU, LIRS, etc.).
*
* @param numbufs number of buffers in the buffer pool
* @param replacementPolicy Name of the replacement policy
*/
public BufMgr(int numbufs, String replacementPolicy) {};

/**
* Pin a page.
* First check if this page is already in the buffer pool.
* If it is, increment the pin_count and return a pointer to this
* page.
* If the pin_count was 0 before the call, the page was a
* replacement candidate, but is no longer a candidate.
* If the page is not in the pool, choose a frame (from the
* set of replacement candidates) to hold this page, read the
* page (using the appropriate method from {\em diskmgr} package) and pin it.
* Also, must write out the old page in chosen frame if it is dirty
* before reading new page.__ (You can assume that emptyPage==false for
* this assignment.)
*
* @param pageno page number in the Minibase.
* @param page the pointer point to the page.
* @param emptyPage true (empty page); false (non-empty page)
*/
public void pinPage(PageId pageno, Page page, boolean emptyPage) {};
```

```
/**
* Unpin a page specified by a pageId.
* This method should be called with dirty==true if the client has
* modified the page.
* If so, this call should set the dirty bit
* for this frame.
* Further, if pin_count>0, this method should
* decrement it.
*If pin_count=0 before this call, throw an exception
* to report error.
*(For testing purposes, we ask you to throw
* an exception named PageUnpinnedException in case of error.)
*
* @param pageno page number in the Minibase.
* @param dirty the dirty bit of the frame
*/
public void unpinPage(PageId pageno, boolean dirty) {};

/**
* Allocate new pages.
* Call DB object to allocate a run of new pages and
* find a frame in the buffer pool for the first page
* and pin it. (This call allows a client of the Buffer Manager
* to allocate pages on disk.) If buffer is full, i.e., you
* can't find a frame for the first page, ask DB to deallocate
* all these pages, and return null.
*
* @param firstpage the address of the first page.
* @param howmany total number of allocated new pages.
*
* @return the first page id of the new pages.__ null, if error.
*/
public PageId newPage(Page firstpage, int howmany) {};

/**
* This method should be called to delete a page that is on disk.
* This routine must call the method in diskmgr package to
* deallocate the page.
*
* @param globalPageId the page number in the data base.
*/
public void freePage(PageId globalPageId) {};

/**
* Used to flush a particular page of the buffer pool to disk.
* This method calls the write_page method of the diskmgr package.
*
* @param pageid the page number in the database.
```

```
*/e
public void flushPage(PageId pageid) {};

/**
* Used to flush all dirty pages in the buffer pool to disk
*
*/
public void flushAllPages() {};

/**
* Returns the total number of buffer frames.
*/
public int getNumBuffers() {}

/**
* Returns the total number of unpinned buffer frames.
*/
public int getNumUnpinned() {}

};
```

3. Internal Design

[A more detailed description of this is provided in the Appendix section of the end of the handout]

The *buffer pool* is a collection of *frames* (page-sized sequence of main memory bytes) that is managed by the Buffer Manager. Conceptually, it should be stored as an array bufPool[numbuf] of Page objects. However, due to the limitation of the Java language, it is not feasible to declare an array of Page objects and later on writing strings (or other primitive values) to the defined Page. To get around the problem, we have defined our Page as an array of bytes and deal with the buffer pool at the byte array level. Note that the size of the Minibase pages is defined in the interface *GlobalConst* of the *global* package. Before jumping into coding, please make sure that you understand how the Page object is defined and manipulated in Java Minibase.

In addition, you should maintain an array FrameDesc[numbuf] of *descriptors*, one per frame. Each descriptor is a record with the following fields:

page number, pin_count, dirtybit.

The *pin_count* field is an integer, *page number* is a PageId object, and *dirtybit* is a boolean. This describes the page that is stored in the corresponding frame. A page is identified by a *page number* that is generated by the DB class when the page is allocated and is unique over all pages in the database. The PageId type is defined as a wrapper class that includes an integer type named pid (in *global* package).

Maintaining a custom Hash Table (*important*): A simple *hash table* should be used to figure out what frame a given disk page occupies. You should implement your own hash table class and not use existing HashTable Java classes. The hash table should be implemented (entirely in main memory) by using an array of pointers to lists of *<page number, frame number >* pairs. The array is called the

directory and each list of pairs is called a *bucket*. Given a *page number*, you should apply a *hash function* to find the directory entry pointing to the bucket that contains the frame number for this page, if the page is in the buffer pool. If you search the bucket and do not find a pair containing this page number, the page is not in the pool. If you find such a pair, it will tell you the frame in which the page resides.

The hash function must distribute values in the domain of the search field uniformly over the collection of buckets. If we have HTSIZE buckets, numbered 0 through M-1, a hash function *h* of the form *h*(*value*) = (*a***value*+*b*) *mod HTSIZE* works well in practice.

When a page is requested, the buffer manager should do the following *[The following instructions also appear in the beginning of the skeleton code methods]*:

1.

Check the buffer pool (by using the hash table) to see if it contains the requested page. If the page is not in the pool, it should be brought in as follows:
(a) Choose a frame for replacement, using the FIFO policy described below.
(b) If the frame chosen for replacement is dirty, *flush* it (i.e., write out the page that it contains to disk, using the appropriate DB class method).
(c) Read the requested page (again, by calling the DB class) into the frame chosen for replacement; the *pin_count* and *dirtybit* for the frame should be initialized to 0 and FALSE, respectively.
(d) Delete the entry for the old page from the Buffer Manager's hash table and insert an entry for the new page. Also, update the entry for this frame in the *FrameDesc* array (defined above) to reflect these changes.

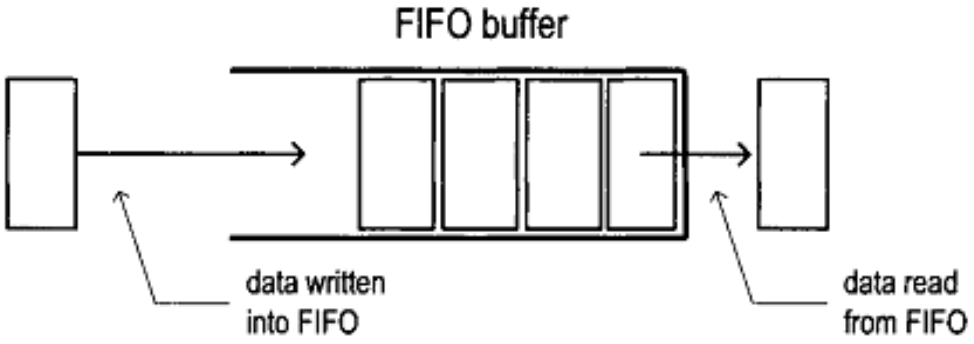
2.

Pin the requested page by incrementing the *pin_count* in the descriptor for this frame and return a pointer to the page to the requestor.

The First In First Out (FIFO) Replacement Policy

FIFO replacement strategy reads the data pages sequentially into the buffer, replacing the oldest page in the buffer (as long as it is not pinned). First in First Out (FIFO) is one of the simplest replacement policies, operating like a basic queue structure for page management. In FIFO, pages are evicted strictly based on their arrival time - the oldest page is removed first, regardless of how frequently it has been accessed. FIFO retains pages based solely on their entry time, keeping the most recently added pages in the buffer. While simple to implement, FIFO may be less efficient than other strategies because it does not consider page usage patterns - it will evict pages even if they are frequently accessed, as long as they are the oldest in the buffer.

If you picture the buffer as a queue of frames, then the new page is placed at the input end of the FIFO buffer. As more pages are written into the buffer, older pages move toward the output end, from where they will be read and eventually removed on a first-in-first-out basis.



4. Error protocol

Though the Throwable class in Java contains a snapshot of the execution stack of its thread at the time it was created and also a message string that gives more information about the error (exception), we have decided to maintain a copy of our own stack to have more control over the error handling.

We provide the *chainexception* package to handle the Minibase exception stack. Every exception created in your *bufmgr* package should extend the ChainException class. The exceptions are thrown according to the following rule:
Error caused by an exception caused by another layer:

For example: (when try to pin page from diskmgr layer)

```
try {
    Minibase.BufferManager.pinPage(pageId, page, true);
}
catch (Exception e) {
      throw new DiskMgrException(e, "DB.java: pinPage() failed");
}
```

Error not caused by exceptions of others, but needs to be acknowledged:

For example: (when try to unpin a page that is not pinned)

```
if (pin_count == 0) {
      throw new PageUnpinnedException (null, "BUFMGR: PAGE_NOT_PINNED.");
}
```

Basically, the ChainException class keeps track of all the exceptions thrown across the different layers. Any exceptions that you decide to throw in your *bufmgr* package should extend the ChainExceptionclass. For testing purposes, we ask you to throw the following exceptions in case of error (use the exact same name, please):

BufferPoolExceededException

Throw a BufferPoolExceededException when an attempt is made to pin a page to the buffer pool with no unpinned frame left.

PagePinnedException
Throw a PagePinnedException when an attempt is made to free a page that is still pinned.

HashEntryNotFoundException
Throw a HashEntryNotFoundException when the page specified by PageId is not found in the buffer pool.

Feel free to throw other new exceptions as you see fit. But make sure that you follow the error protocol when you catch an exception. Also, think carefully about what else you need to do in the *catch* phase. Sometimes you do need to unroll the previous operations when failure happens.

5. Where to Find Makefiles, Code, etc.

Please copy the file P2-1-Buffer-Skeleton.zip (that has been provided in the zip file of the project) into your own Unix local directory. The directory lib contains the jar file that implements the disk manager, you will use this file to compile your code. The content of the directory src is:

under *bufmgr* directory
You should make all your code for *bufmgr* package implemented under this directory.

under *diskmgr* directory
You should study the code for *diskmgr* package carefully before you implement the *bufmgr* package. Please refer to the java documentation of the packages.

under *tests* directory
TestDriver.java, *BMTest.java*: Buffer manager test driver program. Note also that you may need to change the test file *BMTest.java* to select the replacement policy you will implement.

We provide a sample *Makefile* to compile your project. You will have to set up any dependencies by editing this file. You may need to design your own Makefile. Whatever you do, please make sure that the classpath is correct. ***A primer on how to set this up on IntelliJ IDEA IDE is given at the Appendix section of the handout.***

You can find other useful information by reading the java documentation on other packages, such as the *global* and *diskmgr* package.

6. What to Turn in

You should turn in copies of your code together with the Makefile and a readme file. All files need to be zipped in a file named: **your_career_login1_your_career_login2_bufmgr.zip.**

In the readme file, put the name of your group members, and the feature you would like us to know. You do not need to include the library file and other files provided. Please make sure to run the test cases so you know if your implementation is working properly. *We should be able to compile/run your program using make even if you are using any IDE of your choice.*

The directory structure of your zip file should be identical to the directory structure of the provided zip file (i.e., having the directory src, the Makefile. Your grade may be deduced by 5% if you do not follow this. In the readme file, make sure to state the roles of each member in the group (i.e., who did what).

We stress on the fact that, regardless of what IDE you prefer to use, your projects will be tested on the university CS Linux machines (e.g., data.cs.purdue.edu). So, we recommend you always do a test run of your solution on the CS machines.

PART 2: Heap Files [25 Points]

1. Introduction

In this part, you will implement the Heap file layer. You will be given the documentation for the lower layers (Buffer Manager and Disk Space Manager), as well as the documents for managing records on a Heap file page. You can find the package index for the above in the included javadoc directory – index.html.

This assignment has three parts. You have to do the following:

1. Familiarize yourself with the Heap file, HFPage, Buffer Manager and Disk Space Manager interfaces.

2. Implement the Heap file class. You can ignore concurrency control and recovery issues, and concentrate on implementing the interface routines. You should deal with free space intelligently, using either a linked list or page directory to identify pages with room for records. When a record is deleted, you must update your information about available free space, and when a record is inserted, you must try to use free space on existing pages before allocating a new page.

3. Run the tests provided.

The major methods of HeapFile.java that you will implement include:

```
public HeapFile(String name)
protected void finalize() throws Throwable
public void deleteFile()

public RID insertRecord(byte[] record)
public byte[] selectRecord(RID rid)
public void updateRecord(RID rid, byte[] newRecord)
public void deleteRecord(RID rid)
```

```
public int getRecCnt()
public HeapScan openScan()
```

2. Available Documentation

You should begin by reading the chapter on Disks and Files, to get an overview of the HF layer and buffer management. The complete java documentation of all the packages are provided in the *javadoc* directory inside the provided zip file.

3. Complexity Requirements

1. **insertRecord**

insertRecord needs to be done in at most O(log(n)) time, which means using a linked list without any additional data structures to manage the pages of free space is not enough. You can assume the size of a record will not exceed the size of a page. Different data structures and different inserting algorithms may have different pros and cons. You will be asked your design and advantages/disadvantages of your design when grading your project. So, it would be wise to choose a data structure appropriately to keep your implementation simpler.

2. **updateRecord, deleteRecord**

these two operations need to be done in at most O(log(n)) time as well. You need to consider the case when updating a record, the updated record may not be stored in its original page.

3. All other operations need to be done in O(1).

4. Classes to Familiarize Yourself With First

There are three main packages with which you should familiarize yourself: *heap, bufmgr, diskmgr*. Note that part of the *heap* package contains implementation for HFPage. The java documentation of HFPage is provided to you. A Heap file is seen as a collection of records. Internally, records are stored on a collection of HFPage objects.

5. Compiling Your Code and Running the Tests

Copy the file P2-2-Heap-Skeleton.zip (that has been provided in the zip file of the project) to your own local directory and study them carefully. The files are:

- In directory src/heap: Again, the java documentation for packpage *bufmgr, diskmgr* and *heap* are online and also included in the skeleton package provided.

Note that you DO NOT have to throw the same exceptions as documented for the heap package. However, for testing purposes, we DO ask you to name one of your exceptions InvalidUpdateException to signal any illegal operations on the record. In other error situations, you should throw exceptions as you see fit following the error protocol introduced in the Buffer Manager Assignment Section.

We should be able to compile/run your program using make even if you are using any IDE of your choice.

- In directory src/tests: This directory contains the source code of the test. Make the required changes to the error protocols (if you see fit).
- In directory lib: *heapAssign.jar:* this is a library that has the implementation of the Disk Manager and Buffer Manager layers. As you know, you will develop the Buffer Manager layer in the first part of this project. We are providing this library to help you start working with the second part as soon as possible and also to help you test your code comparing the behavior of your code with the one obtained used the library. The final goal is to make the system work without using this library and using your code instead.

6. What to Turn in

Part 2 & 3 should be compiled together. See Part 3 to know what to turn in.

PART 3: Heap Scans [25 Points]

After completing the buffer manager and heap file layers, you will be able to implement a basic scan of a given file. A heap scan is simply an iterator that traverses the file's directory and data pages to return all records. The major methods of **HeapScan.java** that you will implement include the following (please see the included javadoc directory -index.html- for detailed descriptions):

```
protected HeapScan(HeapFile hf)
protected void finalize() throws Throwable
public void close()

public boolean hasNext()
public byte[] getNext(RID rid)
```

Internally, each heap scan should consist of (at least) the following:

Directory Position

The current directory page and/or entry in the scan.

Record Position

The current data page and/or RID in the scan.

The Tuple class is the wrapper of an array of data. Internally it should contain a declaration byte[] data and other methods called by the test driver.

Your implementation should have at most one directory page and/or at most one data page pinned at any given time.

What to Turn in

Part 2 & 3 should be compiled together. You should turn in copies of your code together with the Makefile and a readme file. All need to be zipped in a file named:

your_career_login1_your_career_login2_heapfile.zip.

In the readme file, put the name of your group members, and the feature you would like us to know. *We should be able to compile/run your program using make regardless of whichever IDE you choose to use.*

In the readme file, you should also include the roles of each group member in the project, i.e., who implemented what.

The directory structure of your zip file should be identical to the directory structure of the provided zip file (i.e., having the directory src, the Makefile. Your grade may be deduced 5% off if you don't follow this.

You should upload all your zip files using Blackboard.

Note: We stress on the fact that, regardless of what IDE you prefer to use, your projects will be tested on the university CS Linux machines (e.g. data.cs.purdue.edu). So, we recommend you always do a test run of your solution on the CS machines.

Appendix

Additional Explanation on Part 1 (Optional)

Buffer Manager Components

The Buffer Manager consists of several data structures that are used to manage and track pages in-memory:

Buffer Pool: The Frame Pool is an array of Page objects, defined in the skeleton code as *bufPool*. This array consists of *n* elements, where *n* is a parameter defined at the creation of the database (in this case, 100). This is where the contents of pages are actually stored while they are resident in memory. A Page object is essentially an array of bytes. Minibase provides methods to read and write datatypes to the pages – you should not have to change the actual contents of any Page within your Buffer Manager implementation, but you can see these methods used in the test cases.

Frame Descriptors: An array of FrameDescriptor objects, called *frameDesc*. This is also an array consisting of *n* elements (n=number of frames in total – in buffer pool), where each element in *frameDesc* corresponds to an element in *bufPool*. A FrameDescriptor tracks information about the contents of each frame:

- the ID of the Page stored in the frame (-1 if no Page is stored in that frame),
- the pin count of a frame, and
- the dirty bit (true if the page has been written to since being brought into memory, false otherwise).

Replacement Policy: This has been defined in detailed in previous sections of the handout.

Page Representation

A database reads data from disk in units called blocks rather than read one byte at a time. This improves the efficiency of I/O operations to and from the disk. In Minibase, this data is stored in Pages, where a page is the same size as a block that is read from disk.

The Page object in-memory is not the same as a Page stored on disk. It is only a container for the byte data stored in that page. All pages are identified by a **PageId** – *a globally unique value generated by Minibase when pages are allocated*. This PageId is what Minibase uses to actually access pages from the disk – a call to the Disk Manager is made to read or write the page with the given PageId, and then the DiskManager will either read or write data from/to the provided Page object.

PageId:
The PageId class contains an integer "pid" field that stores the page ID. You can change or set this value directly in order to set a PageId to the appropriate page. This is essentially a wrapped for the primitive integer value of pid.

Page:
Although you should not have to manipulate any data in pages directly within your code, you will have to set the value of Page reference parameters in the pinPage() and newPage() methods. You can use the ".setPage(Page aPage)" method to set the data in the calling page to point to the provided aPage's data. Refer to the Javadocs for documentation of the method(s).

The Disk Manager

The Disk Manager provides methods that handle allocation of new pages on disk, reading pages, and writing pages. The Disk Manager stores certain data – *such as the map of allocated/unallocated space on disk* – in pages itself. It will use the Buffer Manager to manage these pages and bring them into and out of memory as necessary.

The different modules of the DBMS are implemented in Minibase as static instances. In order to call the Disk Manager, you need to access this instance as follows:

Minibase.DiskManager.<method_to_call>()

where <*method_to_call*> is whichever method you are trying to execute.

Several methods are provided to enable you to use the Disk Manager. The methods you will need to use are as follows:

void read_page(PageId pageno, Page apage)
Reads the page denoted by the *pageno* PageId from disk and stores the contents of the page in the *apage* parameter.

void write_page(PageId pageno, Page apage)
Writes the contents of the Page *apage* to the page denoted by the *pageno* PageId

PageId allocate_page(int run_size)
Allocates a contiguous run of pages of size *run_size* on disk and returns the PageId of the first page in that run.

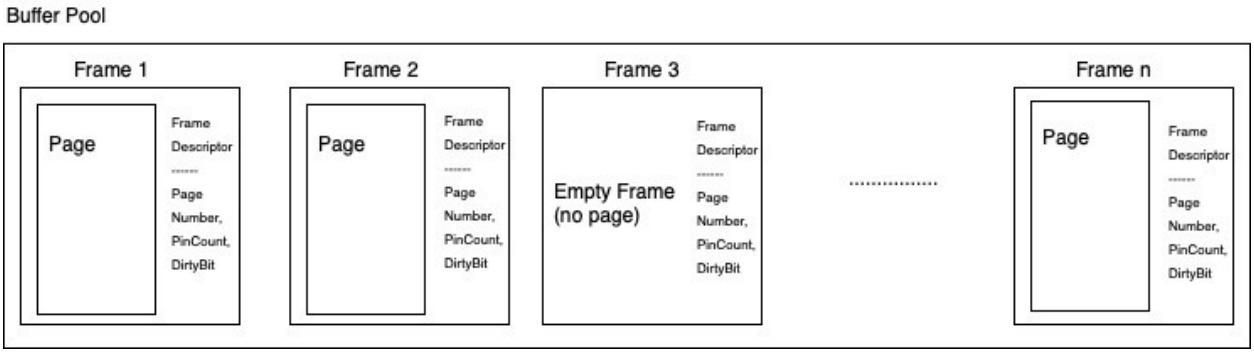
void deallocate_page(PageId start_page, int run_size)

Deallocates a run of pages on disk of size *run_size*, starting with the PageId *start_page.*

void deallocate_page(PageId pageno)
Deallocates a single page.

The Buffer Manager Interface

The simplified Buffer Manager interface that you will implement in this assignment allows a client (a higher-level program that calls the Buffer Manager) to allocate/de-allocate pages on disk, to bring a disk page into the buffer pool and pin it, and to unpin a page in the buffer pool.



Setting up the project in IntelliJ IDEA

While the choice of an IDE is personal preference, we have observed that the most popular IDE among students for Java development is IntelliJ. Therefore, a brief instruction is given below.

NOTE: We stress on the fact that, regardless of what IDE you prefer to use, your projects will be tested on the university CS Linux machines (e.g. data.cs.purdue.edu). So, we recommend you always do a test run of your solution on the CS machines.

Let's set up and run the Part 1 of the project.

Step 1: Download and extract the zip file in a directory. Let's call this "P2-1".

Step 2: Open IntelliJ IDEA, click File->New->Project From Existing Source

Step 3: Browse to and select P2-1 (the directory that hosts the codebase). Click "Choose"

Step 4: In the prompts, click “Next” several times until you get “Finish” (just accept everything on the way as it will catch the dependency in the lib directory automatically). Click “Finish” and “Open in New Window”

Step 5: Now you should be in the project view and can browse through the folder structure on the left panel.

Step 6: To run tests, change the “Project” view to “Packages”

Step 7: Under packages, select “tests” and it will expand.

Step 8: Right-Click on BMTest.java and click on Run BMTest.java

From now on, the regular “Run” command will always execute the test cases.