(Have to format the pdf properly. Will update this soon. (best viewable in ms-word))

THINGS TO CONSIDER

Function:

Common Function Usage:

- Clear screen of console in windows command prompt
    - To clear previous statements
- Thread usage to sleep
    - Used thread.sleep() func
- Input Integrity Check (**IIC**) (**Applies to wherever possible**)
    - Numbers
    - Strings

New Function Added?

- isRoomExists()
    - returns true if the room already exists.

Main Function:

- Choice selection (IIC)
    - Accepts only numbers but not any string value
        - If numbers that are not available in the given choices being inputted, the system will not accept and redirects back to main menu
-

Add Room Choice Option:

- Room Capacity arg
    - Maximum allowed room capacity is 10. If user inputs negative or above 10, the system will not allow and redirects to main menu
- What if room already exists and user trying to add again, the same room name with capacity
    - Can two rooms have same name but different capacity?
        - Naaaah!!! Should not allow. Confusing. Unique room name with its capacity

List Room Choice Option:

- Whether room exists condition

Remove Room Choice Option:

- Whether room exits condition
- Check whether the room is scheduled by anyone if so
    - Override
    - Do not override and exit

Schedule Room Choice Option:

- IIC
- TimeStamp Integrity check
    - Time + day integrity
    - Note:
        - Refer Today's date time stamp at top left corner and schedule accordingly.
        - Input of Date should be in "YYYY-mm-dd" - (e-g): 2018-02-28
        - Input of Time should be in "HH:MM" - (e-g): 10:10
        - Start and End Time should have minutes such that minutes are of round figures like HH:00 or HH:15 or HH:30 or HH:45
        - Start and End Time difference should be minimum of 15 minutes and maximum of 60 minutes
    -
- Timezone based scheduling check (ultimate)
- Daylight saving consideration (ultimate)
- Should not schedule
    - Past timings
    - Very far future timings (lets keep limit like 30 days from the booked day)
    - Scheduling for same day same timing
    - Scheduling for removed rooms which were previously used for a meeting (have to handle this well)
    - Leap year + month end day 31 or 30 thingy + minutes hour thingy (12:60:60)
- Conflict scheduling timings of same date

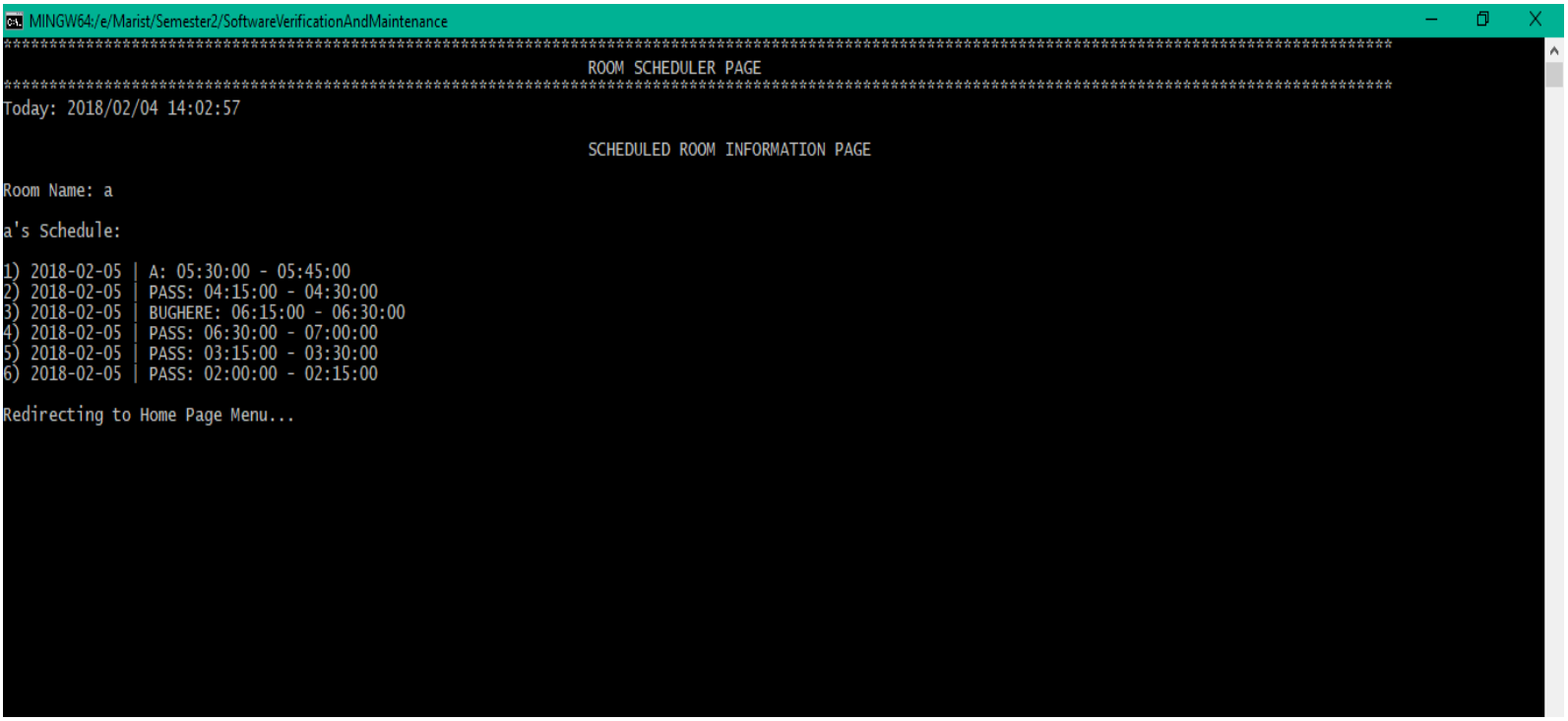
Scheduled Date: 2018-02-05 || Timing: 06:00 to 06:45 (startTime to stopTime)=> diff: 45 mts		
Case A (target1 – target2)	06:00 – 06:45	Same timing | diff => 45 mts
Case B	06:00 – 07:15	More than 60 minutes | diff => 75 mts
Case C	05:45 – 06:15	Target1 before startTime and Target2 before stopTime | diff => 30 mts
Case D	05:45 – 06:45	Wrapping already booked timing
Case E	06:15 – 06:30	Target1 & Target2 inside the startTime and stopTime such that Target1 after startTime and Target2 before stopTime
Case F	06:30 – 07:15	Target1 after startTime and Target2 after stopTime
Case G		

	startTime	stopTime
Target1	Before	Before
Target1	Before	After
Target1	After	Before
Target1	After	After
Target2	Before	Before
Target2	Before	After
Target2	After	Before
Target2	After	After

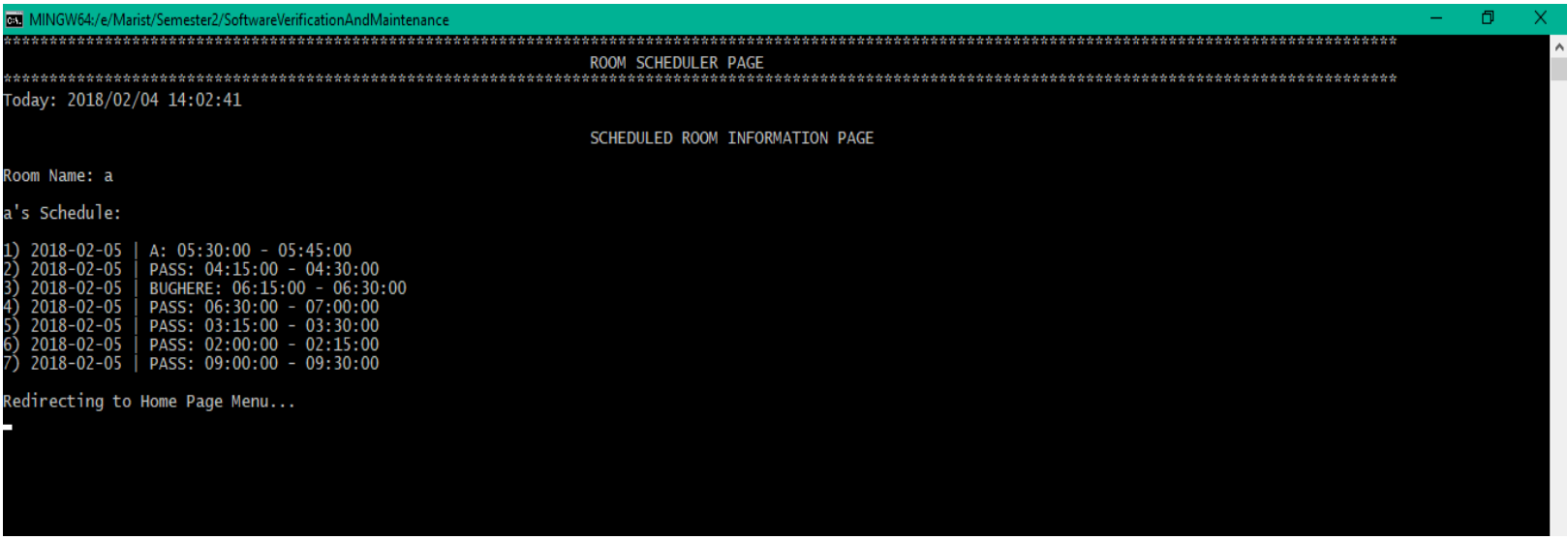
TEST CASES CHART to have consistent Room Scheduling

Date: 2018-02-05 || Timing (start – stop): 06:00 – 06:30 || (dev mindset to break the code cases)						
	startTime	stopTime	Cases (target1 – target2 for Target1 Focussed) (target2 – target1 for Target2 Focussed)	Status	Remarks (NP – No Probs|| Bug - Bug)	Code Review and Output Review
Target1 Focussed	Target1 before	Target2 before	05:30 – 05:45 || 04:15 – 05:30	PASS || PASS	NP || NP	
	Target1 before	Target2 after	05:30 – 06:30	FAIL	NP	
	Target1 after	Target2 before	06:15 – 06:30	FAIL	**Bug**	**False Alarm. Not a bug (**06:00 – 06:30 was not booked prior to this test case**). Rechecked and verified (**09:00 – 09:30 booked || then 09:15 – 09:30 case done -> result PASS || FAIL**).**
	Target1 after	Target2 after	06:15 – 06:45 || 06:30 – 07:00	FAIL || PASS	NP || NP	
Target2 Focussed	Target2 before	Target1 before	05:45 – 05:30 || 08:30 – 08:15	FAIL || FAIL	NP || NP	
	Target2 before (03:15 – 03:30)	Target1 after (03:15 – 03:30)	03:30 – 03:00	FAIL	NP	
	Target2 after	Target1 before	06:30 – 06:15	FAIL	NP	
	Target2 after	Target1 after	06:45 – 06:15 || 07:00 – 06:30	FAIL	NP || NP	
Common	Target1 equals	Target2 equals	06:00 – 06:30	FAIL	NP	
	Target2 equals	Target1 equals	06:30 – 06:00	FAIL	NP	
Booked range between target1 and target2			Book 1st 02:00 – 02:15 || Then 01:45 – 02:30	PASS || FAIL	NP	

Console output Verify:



- Only one bug.. good start.
    - False alarm. Not a bug



- Handle if someone changed their local time in their system and try to schedule. How do I handle it???? Hmmmmm

List Scheduled Room Choice Option:
- isRoomExists()
- Existed Room have schedules?
- IIC

Old date time api vs new one:
- ZoneId.getAvailableZoneIds()
    - To print all time zone
- Date (old):
    - Java.sql and java.util
    - Not Thread safe (multiple threads working on same date object, then if one thread completes work faster, the other suffers)
    - Import java.text.*; to format the date
- (new):
    - Immutable (changing existing value will create a new object for you??? **Ask someone to explain this well.. unable to understand well**)
    - Simple to use
    - LocalDate
        - System defined date java.time.LocalDate;
    - LocalTime
        - Java.time.LocalTime;
        - LocalTime.now()
        - LocalTime.now(ZoneId.of("Asia/Kuwait"))
    - Instant
        - Instant.now()
    - LocalDateTIme

EXTRAS

File Write Operation Options

- https://www.journaldev.com/878/java-write-to-file
- Let's have a brief look at four options we have for java write to file operation.

    - FileWriter: FileWriter is the simplest way to write a file in java, it provides overloaded write method to write int, byte array and String to the File. You can also write part of the String or byte array using FileWriter. FileWriter writes directly into Files and should be used only when number of writes are less.
    - BufferedWriter: BufferedWriter is almost similar to FileWriter but it uses internal buffer to write data into File. So if the number of write operations are more, the actual IO operations are less and performance is better. You should use BufferedWriter when number of write operations are more.
    - FileOutputStream: FileWriter and BufferedWriter are meant to write text to the file but when you need raw stream data to be written into file, you should use FileOutputStream to write file in java.
    - Files: Java 7 introduced Files utility class and we can write a file using it's write function, internally it's using OutputStream to write byte array into file.

JSON Selection

- https://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/
- Parsing speed isn't the only consideration when choosing a JSON library, but it is an important one. Upon running this benchmark test, what we found was that there is no one library that blows the others away on parsing speed across all file sizes and all runs. The libraries that performed best for big files suffered for small files and vice versa.

- Choosing which library to use on the merit of parsing speed comes down to your environment then.

    - If you have an environment that deals often or primarily with big JSON files, then Jackson is your library of interest. GSON struggles the most with big files.
    - If your environment primarily deals with lots of small JSON requests, such as in a micro services or distributed architecture setup, then GSON is your library of interest. Jackson struggles the most with small files.
    - If you end up having to often deal with both types of files, JSON.simple came in a very close 2nd place in both tests, making it a good workhorse for a variable environment. Neither Jackson nor GSON perform as well across multiple files sizes.
    - As far as parsing speed goes, JSONP doesn't have much to recommend for it in any scenario. It performs poorly for both big and small files compared to other available options. Fortunately, Java 9 is reportedly getting native JSON implementation, which one would imagine is going to be an improvement over the reference implementation.

- So there you have it. If you're concerned about parsing speed for your JSON library, choose Jackson for big files, GSON for small files, and JSON.simple for handling both. Let me know if you have any thoughts on this benchmark in the comments.

The JSON file we got at first try:
Room Name: a


```
[
 {
  "name": "a",
  "capacity": 6,
  "meetings": [
   {
    "startDate": "2018-02-15",
    "startTime": "10:15:00",
    "stopTime": "10:30:00",
    "subject": "ShortMeeting"
   },
   {
    "startDate": "2018-02-20",
    "startTime": "11:00:00",
    "stopTime": "12:00:00",
    "subject": "LongMeeting"
   },
   {
    "startDate": "2018-02-25",
    "startTime": "01:15:00",
    "stopTime": "02:00:00",
    "subject": "FunMeeting"
   }
  ]
 }
]
```


Redirecting to Home Page Menu...

- File autoclose using try with resource block.

Things to do:

- File path independent resolve
- Room export suggestion thingy
- Lock file before writing

STATIC TOOL ANALYSIS RESULTS

SonarLint eclipse plugin results:

RoomScheduler.java

Issue	Issue Summary	Issue Description	Issue Correction
// Advantages include -> @ExposeAnnotation || serializing nulls || custom instance creators || set version support || pretty printing || custom serialize & deserialize	This block of commented-out lines of code should be removed.	Sections of code should not be "commented out" (squid:CommentedOutCodeLine) Code smell Major Programmers should not comment out code as it bloats programs and reduces readability. Unused code should be deleted and can be retrieved from source control history if required.	`// Advantages include ExposeAnnotation - serializing nulls - custom instance creators - set version support - pretty printing - custom serialize and deserialize`
ArrayList<Room> rooms = new ArrayList<Room>();	Replace the type specification in this constructor call with the diamond operator ("<>").	The diamond operator ("<>") should be used (squid:S2293) CODE_SMELL Code smell MINOR Minor Java 7 introduced the diamond operator (<>) to reduce the verbosity of generics code. For instance, instead of having to declare a List's type in both its declaration and its constructor,	`ArrayList<Room> rooms = new ArrayList<>();`

		you can now simplify the constructor declaration with <>, and the compiler will infer the type. Note that this rule is automatically disabled when the project's sonar.java.source is lower than 7	

while (!end) {...}	Remove this expression which always evaluates to "true"	Boolean expressions should not be gratuitous (squid:S2589)
		
		Code smell Major
		
		If a boolean expression doesn't change the evaluation of the condition, then it is entirely unnecessary, and can be removed. If it is gratuitous because it does not match the programmer's intent, then it's a bug and the expression should be fixed.
			`case 8:`
			
			`Log.info("Quitting the program...");`
			
			`utility.Utility.sleepFor(2000);`
			
			`utility.Utility.clearScreen();`
			
			`end = true;`

case 8:
System.out.println("Quitting the program...");
System.exit(0);
default:
	End this switch case with an unconditional break, return or throw statement.	Switch cases should end with an unconditional "break" statement (squid:S128)
		
		Code smell Blocker
		
		When the execution is not explicitly
			`case 8:`
			
			`Log.info("Quitting the program...");`
			
			`utility.Utility.sleepFor(2000);`

		terminated at the end of a switch case, it continues to execute the statements of the following case. While this is sometimes intentional, it often is a mistake which leads to unexpected behavior.	`utility.Utility.clearScreen();` `end = true;`
`System.out.println("Quitting the program...");`	Replace this use of System.out or System.err by a logger.	Standard outputs should not be used directly to log anything (squid:S106) Code smell Major When logging a message there are several important requirements which must be fulfilled: • The user must be able to easily retrieve the logs • The format of all logged message must be uniform to allow the user to easily read the log • Logged data must actually be recorded • Sensitive data must only be logged securely If a program directly writes to the standard outputs, there is absolutely no way to	`Log.info("Quitting the program...");`

		comply with those requirements. That's why defining and using a dedicated logger is highly recommended.	
e.printStackTrace();	Use a logger to log this exception.	Throwable.printStackTrace(...) should not be called (squid:S1148) Vulnerability Minor `Throwable.printStackTrace(...)` prints a `Throwable` and its stack trace to some stream. By default that stream `System.Err`, which could inadvertently expose sensitive information. Loggers should be used instead to print `Throwables`, as they have many advantages: • Users are able to easily retrieve the logs. • The format of log messages is uniform and allow users to browse the logs easily. This rule raises an issue when `printStackTrace` is used without arguments, i.e. when the stack trace is	`Log.trace(e);`

		printed to the default stream.	

`System.out.print("Room Name: ");`	Define a constant instead of duplicating this literal "Room Name: " 3 times.	String literals should not be duplicated (squid:S1192)
		
		Code smell Critical
		
		Duplicated string literals make the process of refactoring error-prone, since you must be sure to update all occurrences.
		
		On the other hand, constants can be referenced from many places, but only need to be updated in a single place.
			`private static final String ROOM_NAME = "Room Name: ";`
			
			……..
			
			`Log.info(ROOM_NAME);`

`if (roomList.size() == 0 || !isRoomExists(roomList, name)) {`	Use isEmpty() to check whether the collection is empty or not.	Collection.isEmpty() should be used to test for emptiness (squid:S1155) Code smell Minor Using `Collection.size()` to test for emptiness works, but using `Collection.isEmpty()` makes the code more readable and can be more performant. The time complexity of any `isEmpty()` method implementation should be `O(1)` whereas some	`if (roomList.isEmpty() || !isRoomExists(roomList, name)) {`

		implementations of `size()` can be `O(n)`.	

String minuteStamp = time.split(":")[1];
if (minuteStamp.equals("00")) || minuteStamp.equals("15") || minuteStamp.equals("30") || minuteStamp.equals("45")) {
	Replace this if-then-else statement by a single return statement.	Return of boolean expressions should not be wrapped into an "if-then-else" statement (squid:S1126)
		
		Code smell Minor
		
		Return of boolean literal statements wrapped into `if-then-else` ones should be simplified.
			Boolean status = true;
			if (!(minuteStamp.equals("00") || minuteStamp.equals("15") || minuteStamp.equals("30") || minuteStamp.equals("45"))) {
			
			status = false;
			}
			return status;

protected static boolean isSameRoomAndTimeBooked(ArrayList<Room> roomList, String name, String startDate, String startTime, String endTime) {	Remove this unused method parameter "startDate".	Unused method parameters should be removed (squid:S1172) Code smell Major Unused parameters are misleading. Whatever the values passed to such parameters, the behavior will be the same.	protected static boolean isSameRoomAndTimeBooked(ArrayList<Room> roomList, String name, String startTime, String endTime) {

for (Meeting m : currentRoom.getMeetings( )) { ...}	A "NullPointerException" could be thrown; "currentRoom" is nullable here.	Null pointers should not be dereferenced (squid:S2259)
		
		Bug Major
		
		A reference to `null` should never be dereferenced/accessed. Doing so will cause
			if (currentRoom == null) {
			
			return false;
			}

		a `NullPointerException` to be thrown. At best, such an exception will cause abrupt program termination. At worst, it could expose debugging information that would be useful to an attacker, or it could allow an attacker to bypass security measures. Note that when they are present, this rule takes advantage of `@CheckForNull` and `@Nonnull` annotations defined in JSR-305 to understand which values are and are not nullable except when `@Nonnull` is used on the parameter to `equals`, which by contract should always work with null.	
`protected static void scheduleRoom(ArrayList<Room> roomList) { .. }`	Refactor this method to reduce its Cognitive Complexity from 38 to the 15 allowed.	Cognitive Complexity of methods should not be too high (squid:S3776) Code smell Critical Cognitive Complexity is a measure of how hard the control flow of a method is to understand. Methods with high Cognitive Complexity will be difficult to maintain.	

		See • Cognitive Complexity	

Room curRoom = *getRoomFromName*(roomList, name);
 	
Meeting meeting = new Meeting(startDate, startTime, endTime, subject);
 	
curRoom.addMeeting(meeting);
	A "NullPointerException" could be thrown; "curRoom" is nullable here.	Null pointers should not be dereferenced (squid:S2259)
		
		Bug Major
		
		A reference to `null` should never be dereferenced/accessed. Doing so will cause a `NullPointerException` to be thrown. At best, such an exception will cause abrupt program termination. At worst, it could expose debugging information that would be useful to an attacker, or it could allow an attacker to bypass security measures.
		
		Note that when they are present, this rule takes advantage of `@CheckForNull` and `@Nonnull` annotations defined in JSR-305 to understand which values are and are not nullable except when `@Nonnull` is used on the parameter to `equals`, which by contract should always work with null.
			if (curRoom != null) {
			
			curRoom.addMeeting(meeting);
			
			Log.info("\nSuccessfully scheduled meeting!");
			
			} else {
			
			Log.error("Error occurred while scheduling the meeting...");
			
			}

`if (getRoomFromName(roomList, roomName).getMeetings().size() == 0) {` `..}`	A "NullPointerException" could be thrown; "getRoomFromName" is nullable here.	Null pointers should not be dereferenced (squid:S2259) Bug Major A reference to `null` should never be dereferenced/accessed. Doing so will cause a `NullPointerException` to be thrown. At best, such an exception will cause abrupt program termination. At worst, it could expose debugging information that would be useful to an attacker, or it could allow an attacker to bypass security measures. Note that when they are present, this rule takes advantage of `@CheckForNull` and `@Nonnull` annotations defined in JSR-305 to understand which values are and are not nullable except when `@Nonnull` is used on the parameter to `equals`, which by contract should always work with null.	

Utility.java

Issue	Issue Summary	Issue Description	Issue Correction

`public class Utility {`
`...}`
	Add a private constructor to hide the implicit public one.	Utility classes should not have public constructors (squid:S1118)
		
		Code smell Major
		
		Utility classes, which are collections of `static` members, are not meant to be instantiated. Even abstract utility classes, which can be extended, should not have public constructors.
		
		Java adds an implicit public constructor to every class which does not define at least one explicitly. Hence, at least one non-public constructor should be defined.
		
		...
		
		Exceptions
		
		When class contains `public static void main(String[] args)` method it is not considered as utility class and will be ignored by this rule.
			`private Utility() {`
			`  throw new IllegalStateException("Utility class");`
			`}`

`public static void clearScreen() { try { if (System.getProperty("os.name").contains("Windows")) try { new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor(); } catch (InterruptedException e) { // TODO Auto-generated catch block e.printStackTrace(); } ... }`	Extract this nested try block into a separate method.	Try-catch blocks should not be nested (squid:S1141) Code smell Major Nesting `try/catch` blocks severely impacts the readability of source code because it makes it too difficult to understand which block will catch which exception.	
`// TODO Auto-generated catch block`	Complete the task associated to this TODO comment.	Track uses of "TODO" tags (squid:S1135) Code smell Info `TODO` tags are commonly used to mark places where some more code is required, but which the developer wants to implement later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed.	deleted
`e.printStackTrace();`	Use a logger to log this exception.	Throwable.printStackTrace(...) should not be	`Log.trace(e);`

		called (squid:S1148) Vulnerability Minor `Throwable.printStackTrace(...)` prints a `Throwable` and its stack trace to some stream. By default that stream `System.Err`, which could inadvertently expose sensitive information. Loggers should be used instead to print `Throwables`, as they have many advantages: • Users are able to easily retrieve the logs. • The format of log messages is uniform and allow users to browse the logs easily. This rule raises an issue when `printStackTrace` is used without arguments, i.e. when the stack trace is printed to the default stream.	
`try { new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor(); } catch (InterruptedException e) {`	Either re-interrupt this method or rethrow the "InterruptedException".	"InterruptedException" should not be ignored (squid:S2142)	`} catch (InterruptedException e) { // Restore interrupted state...`

`log.trace(e);`
`}`
		Bug Major
		
		`InterruptedExceptions` should never be ignored in the code, and simply logging the exception counts in this case as "ignoring". The throwing of the `InterruptedException` clears the interrupted state of the Thread, so if the exception is not handled properly the fact that the thread was interrupted will be lost. Instead, `InterruptedExceptions` should either be rethrown - immediately or after cleaning up the method's state - or the thread should be re-interrupted by calling `Thread.interrupt()` even if this is supposed to be a single-threaded application. Any other course of action risks delaying thread shutdown and loses the information that the thread was interrupted - probably without finishing its task.
		
		Similarly, the `ThreadDeath` exception should also be propagated. According to its JavaDoc:
		
		If `ThreadDeath` is caught by a
			`Thread.currentThread().interrupt();`
			
			`log.trace(e);`
			`}`

		method, it is important that it be rethrown so that the thread actually dies.	
String jsonPrettyPrint = gson.toJson(obj);	Immediately return this expression instead of assigning it to the temporary variable "jsonPrettyPrint".	Local variables should not be declared and then immediately returned or thrown (squid:S1488) Code smell Minor Declaring a variable only to immediately return or throw it is a bad practice. Some developers argue that the practice improves code readability, because it enables them to explicitly name what is being returned. However, this variable is an internal implementation detail that is not exposed to the callers of the method. The method name should be sufficient for callers to know exactly what will be returned.	

Room.java

Issue	Issue Summary	Issue Description	Issue Correction

`public ArrayList<Meeting> getMeetings() { ...}`	The return type of this method should be an interface such as "List" rather than the implementation "ArrayList".	Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList" (squid:S1319)
		
		Code smell Minor
		
		The purpose of the Java Collections API is to provide a well defined hierarchy of interfaces in order to hide implementation details.
		
		Implementing classes must be used to instantiate new collections, but the result of an instantiation should ideally be stored in a variable whose type is a Java Collection interface.
		
		This rule raises an issue when an implementation class:
		
		• is returned from a `public` method.
		• is accepted as an argument to a `public` method.
		• is exposed as a `public` member.
			`import java.util.List;`
			`public class Room {`
			
			`private String name;`
			`private int capacity;`
			`private List<Meeting> meetings;`
			`.....`
			`public List<Meeting> getMeetings() {`
			`return meetings;`
			`}`
			`public void setMeetings(List<Meeting> meetings) {`
			
			`this.meetings = meetings;`
			`}`
			`}`

Meeting.java

Issue	Issue Summary	Issue Description	Issue Correction
`public String toString() { return getStartDate() + " | " + getSubject() + ": " + this.getStartTime().toString() + " - " + this.getStopTime(); }`	"getStartTime" returns a string, there's no need to call "toString()".	"toString()" should never be called on a String object (squid:S1858) Code smell Minor Invoking a method designed to return a string representation of an object which is already a string is a waste of keystrokes. This redundant construction may be optimized by the compiler, but will be confusing in the meantime.	`public String toString() { return getStartDate() + " | " + getSubject() + ": " + this.getStartTime() + " - " + this.getStopTime(); }`

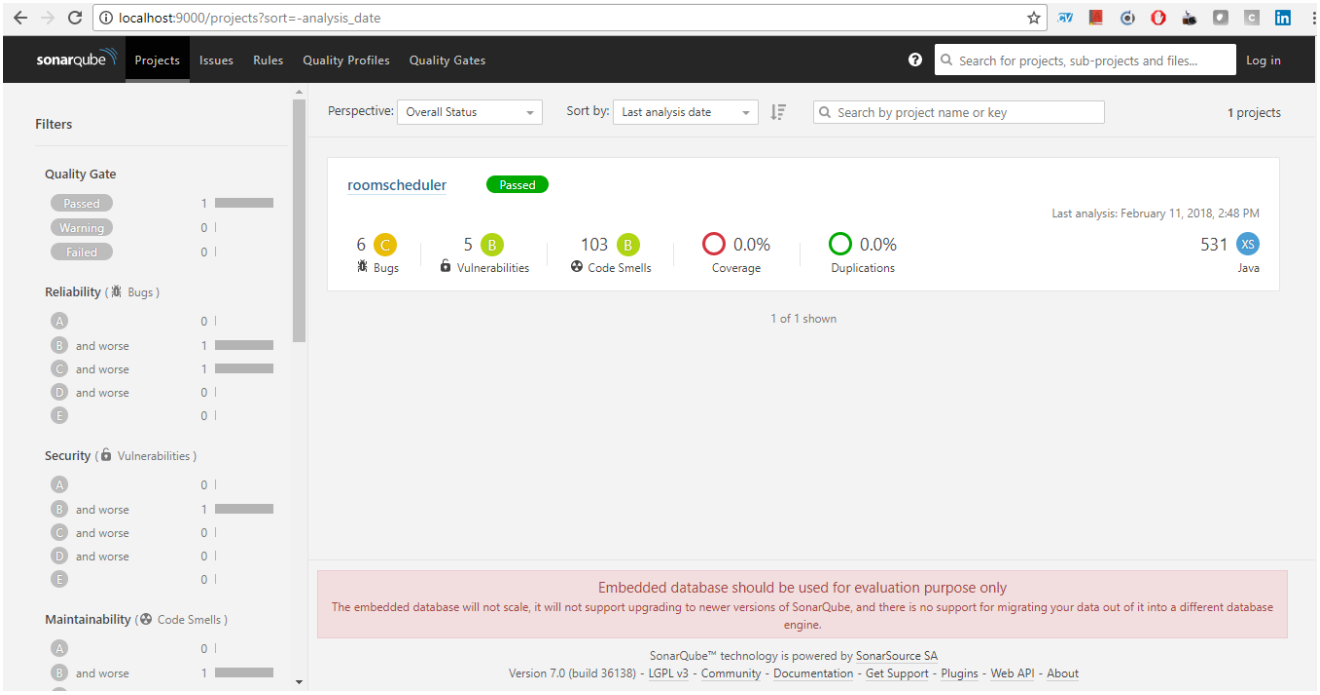
Code Coverage Pro

PMD

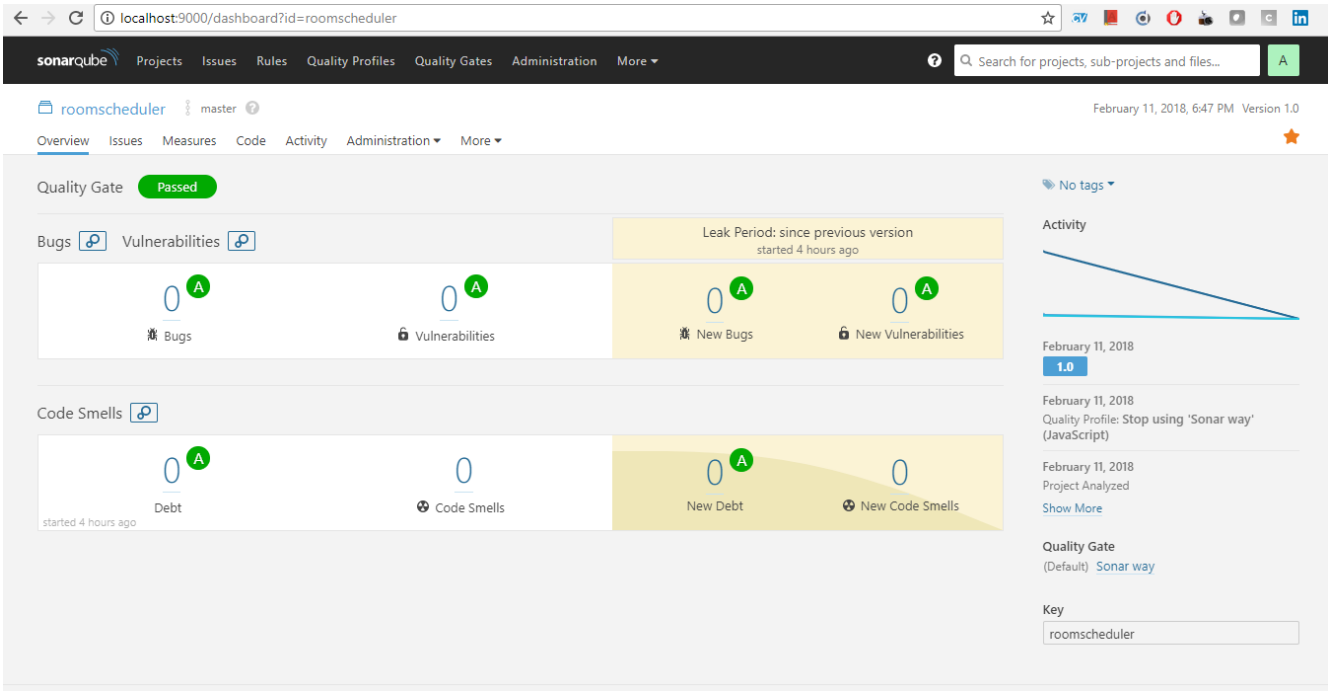
- Report being generated.
    - Link:

SonarQube Analyis

Take 1: Analysis result:



Take 2: Analysis after modification:



Take 1 vs Take 2 Analysis after code modification: