Code smells are hints that imply the software might have potentially deeper problems than what is visible or present. The purpose of Smellware is to motivate programmers to implement consistent and effective coding styles by detecting code smells and informing users of such violations. There are tools that send visual or auditory feedbacks to the user developed for detecting code smells. However, visual and auditory feedbacks are easy to overlook and the developers are not always inclined to remove code smells.

To reinforce the feedback mechanism, Smellware communicates with the user using both olfactory and visual feedbacks. Smellware is designed to check for code smells in source codes written in Java for the current release, and can be upgraded for other programming languages or as a plug-in for Java IDE’s in the future. The hardware of Smellware consists of 2 different odour dispensers and a fan. The dispensers emit odours when code smell errors are detected and the fan cleans the surrounding air when all code smells are fixed. More than 80% of Smellware participants find that Smellware can mitigate the number of code smell errors they make in their final product.

Categories and Subject Descriptors
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

General Terms
Design, Experimentation, Human

Keywords
Code smells, Olfactory sensors, Human Interface Technology

1. INTRODUCTION
Code smells are indicators of software code having potential design or programming error. They are not errors that will cause compilation failure or runtime crashes, but rather bad coding styles that lead to ambiguity and code inefficiency. With short deadlines and short iteration cycles for software development, software developers often take programming shortcuts in coding to meet requirements and deadlines to satisfy their customers.

These shortcuts usually lead to developing and accumulating technical debts that may potentially disrupt the progression and evolution of the project in the future. To fix this problem, we are proposing a solution to use olfactory devices to redefine how code smells are identified and used to provide feedback to the developers.

The proposed project aims to give user feedback on their current coding style and implementations through an electronic olfactory device. This device as shown in Figure 1 emits commercial off-the-shelf fragrances that allow users to “smell something that might have gone wrong”. This is achieved by having a background process that checks the written code and sends a signal to the olfactory device to emit odours.

Visual and auditory systems have been implemented widely for user notifications. However, there is a limit to the degree of how much information a person can process all the information through visual and auditory feedbacks. Often when under stress and large amount of information, many smaller errors are neglected even if both visual and auditory sensors are employed. A user can simply choose to ignore the growing list of code smell violations to avoid the warnings from interrupting their work at hand. Therefore, there have been many active researches in haptic feedbacks for many human interface technologies.
Olfaction has been one of the important ways that human beings gather information and interact with the environment surrounding them. Our olfactory receptors, our noses, allow us to identify many things in this world including food, signs of danger, and specific events. According to [1], human nose can identify strong smells to reflect olfactory perception similar to vision and audition. However, comparing to visual and audio sensing systems, applications for using human olfactory sensing have not been widely administered in the field of software development. The potential of using olfactory devices to strengthen impression or experience in a code segment maybe be a new approach to new programming practices. Therefore, the proposed project is intended to answer the research question of “Whether olfactory feedbacks can allow developers to produce better programming code”.

In this paper, we will discuss the software and hardware designs of Smellware, user evaluations and results, and possible future work.

2. RELATED WORK

Code smell is a potential error or conflict in code logic or design. There has been extensive research done on developing algorithms for detecting code smells [2][3][4]. Most conventional compilers have the ability to detect bad naming style, duplicated code, etc. Code smell detectors that are built specifically for detecting bad programming style, illegal memory allocation and rule violation suggest taking new approaches towards refactoring programming code.

There has been increasing awareness in the field of olfaction that interacts with the virtual reality. Studies have been done to perform virtual therapy with olfactory [5] and augmented virtual reality with olfactory and gustatory [6]. It shows how important that olfactory can affect the ambient surrounding and how the user perceive the information displayed. However, one major challenge presented in an olfactory display is how an old smell can be removed before the new smell gets emitted. Washburn [7] evaluated a few commercial olfactory displays and designed his own custom olfactory device. His findings allowed us to look at different mechanisms and manipulate our design accordingly to produce a more suitable olfactory system for our use. There have also been projects that attempt to identify real world scents or emotions by electronic olfactory sensing systems: SPOT-NOSED mimics how brain identifies different smells and stores the information inside a robot [8]. All of the aforementioned projects show that olfactory is an important aspect in detecting the situation and surrounding environment hence the motivation for this research project.

3. DESIGN COMPONENTS

This section is to give an overview on how Smellware is implemented. How and why the odours are selected and which code smells are handled by Smellware will be discussed. In addition, the design of Smellware device’s software and hardware component will be studied thoroughly in the following subsections.

In Figure 2 below, the storyboard briefly explains the steps on how Smellware operates during a programming session by the intended user. Each component is designed to work together with another under different circumstances to produce visual and olfactory feedbacks.

3.1 Odour Choice

The original idea of Smellware is to emit unpleasant smell when code smell errors are detected and to increase the strength of the unpleasant smell as the code smell errors accumulates in the programming code. However, with the lack of knowledge in chemistry, we could not produce a formula of an unpleasant odour for the dispenser. The alternative is to purchase any off-the-shelf air freshener sprays. This is sufficient for our project since we are exploiting the fact that the tolerance for concentrated air freshener is low with human nose and it irritates most users regardless of pleasant or unpleasant smell. Another challenge presented is how human nose adapt to smells. To avoid the tendency of growing accustomed and insensitive to a specific scent, our olfactory device will dispense small amounts at the beginning and increase gradually. In addition, we will have two different air fresher odours prepared that will alternate every 5 sprays. The odour selected does not represent any code smell in particular. We considered the possibility of using different odours to represent different code smell violations; however, the human ability to analyze and separate the individual odour from a mixture of smells is relatively low, comparing to other animals such as dogs. Therefore, Smellware only emits one odour at a time. This makes “smelling something wrong” intuitive because, under normal situations, when we smell unrecognized odour, we do not immediately know what it represents and will to find out what causes that smell.
3.2 Software Implementation

The software system of Smellware will include three subcomponents: a graphical user interface (GUI) for source code input, a code-smell detector and a hardware controller.

The GUI is implemented in JSwing as a notepad-like application. The user may open and save as a textfile. There will also be a button to click on the GUI to ignore the detected code smell violation if the user recognizes it as a non-code smell violation. The user may select which types of code smell violations that Smellware should detect and may choose to turn on or off the visual warning aid.

The code-smell detector detects a set of code smell rules by parsing the text in the notepad GUI and provides the user with olfactory and visual notifications. A serial signal is sent to the hardware component to disperse scent. Smellware is implemented in Java since it will be easier to interface with the Arduino hardware controller. Upon detection of rule violations, the microcontroller receives signals of an odour to be emitted. At the same time that the odour signal is being sent, if visual warning aid is turned on, a pop-up textbox will interrupt the current workflow to indicate which identifier name, which method or which class contains violated code smells. If a violation has been delayed from fixing, the strength of the odour will intensify as the spray activates every 20 seconds. This is used to give user a more intuitive feedback as they code; the stronger the odour is, the more severe the problem is to ignore such warnings. When all the violations are fixed, another signal is sent to activate the fan to blow away the emitted odours. The code smell detector detects code smell errors in three different scopes during implementation time: code smell per line, code smell per method, and code smell per class shown on Table 1 below.

<table>
<thead>
<tr>
<th>Per line</th>
<th>Per method</th>
<th>Per class</th>
</tr>
</thead>
<tbody>
<tr>
<td>- identifier too long</td>
<td>- method too long</td>
<td>- class too big</td>
</tr>
<tr>
<td>- identifier too short</td>
<td>- argument list too long</td>
<td></td>
</tr>
</tbody>
</table>

These code smell rules are only a small portion of the traditional code smell violations. They are chosen to illustrate how smell can help motivate the programmers to code better. The user will know beforehand which code smell Smellware handles, so that the user knows what might have gone wrong when they smell an unexpected odour. This is similar to how humans attempt to find out the source of an unexpected odour (olfactory localization). For example, when one smells garbage odour in the house, he or she will have an idea of where and what the odour might be. He or she will then try to confirm these speculations by checking different areas in the house one by one. This is equivalent to the debugging action of a user using Smellware looking for the reason of the smell.

There is an option, “ignore all code smell errors”, on the main GUI that allows the user to clear all odours. This not only clears the smell but will also disable code smell detection unit. Another feature aforementioned is the incorporation of two different smells to reduce the chance of human nose adapting to the smell. Every 5 sprays, the microcontroller will alternate signals internally to activate the other spray in the pair. When all violations are fixed, the detector interface will send signal to start the fan to clear the odours.

The visual indicators of code smells are pop-ups rather than the traditional visual feedbacks that other code smell detectors have. The existing code smell detector software maintains a list of code smells in the background and either displays the list concurrently while the programmers code or when the code is compiled. This method, however, is not reinforcing the urgency of fixing code smells, as they do not interrupt the current workflow and the programmer can easily choose to neglect the warnings and move on with the other tasks. The pop-up textbox warnings are implemented with the intent to stress the importance of fixing code smells immediately and are there to help users locate the reason of odours. The pop-up warnings are shown below in Figure 3 and 4 for demonstration purposes.

3.3 Hardware Implementation

The hardware component of Smellware is interfaced with the software module using the Arduino Microcontroller. A circuit is shown in Figure 5 to illustrate how the Arduino board is connect to the AirWick Freshmeltic automatic sprays as odour emitters. A fan controller circuit is added to the system to clear and reset the odours that are already diffused in air. Figure 6 shows the actual...
pin connections and circuit board designed in Figure 5. Figure 7 shows the logic interpretation of Smellware hardware components.

During the first attempt of building the scent emitter component, it was found that the spray nozzle draws too much current and thus cannot be controlled and directly connected to the Arduino. As a result, the circuit in Figure 5 was built to allow external power support to the spray nozzle while maintaining a way to control the spray activation using MOSFETs. N-channel MOSFETs are used as switches to turn the sprays and the fan on or off. The MOSFET used in Smellware is F12N10L by FAIRCHIL. It withstands current flow up to maximum of 12 amps. The Air-Wick Freshmatic sprays require a current of 2 amps to be powered.

4. RESULTS AND USER EVALUATIONS
Users have been asked to use Smellware to implement a short Java class, and to find and correct code smell violations in Java test classes with pre-planted code smell errors. In the following section, the evaluation procedure, test and user evaluation results will be discussed.

Figure 5. Smellware Circuit Scheme

Figure 6. Circuit Boards

Figure 7. Hardware Component

4.1 User Acceptance Test and Test Results
The pre-requisite criterion for user study participation is being able to implement Java programs. The objective of Smellware is to motivate developers to avoid programming in bad coding style. Smellware does not check for compilation errors nor does it run the program. It is assumed that the participants have the capability to implement in legal Java syntax. Figure 8 below shows the user evaluation in progress. There are 3 parts to this user study: a programming question, a locating and correcting code smells exercise, and a set of quantitative questionnaire for usability and feasibility study. The participants are debriefed with information such as what code smells are, what Smellware does, which code smell violations Smellware check, and how the experiments are going to be executed. The Smellware visual warning aid is turned
off by default and the participants are asked to only turn it on if they have difficulties locating the source of the code smell.

**4.1.1 Programming Question**
The first programming question the participants are asked to implement is a Java program that performs string manipulation, such as reversing the string, removing the vowels and obtaining the string length. This question is designed to check whether the participants would refactor the individual tasks into separate functions to avoid a method being too long or use an integer named “i” to access the character array without using a more descriptive identifier name to indicate which character array it is accessing. Almost every participant, being accustomed to use an integer named “i” in a for conditional loop to access arrays, resulted in a code smell violation with identifier name too short, even when they are already informed that Smellware would catch such violation. A few participants who violated this rule, was unable to realize why the spray mechanisms was activated and had to turn the visual warning on. None of the participants had problems with refactoring the methods properly. This shows that a major part of the reason why code smells exist is that programmers have developed such habits since their early years of software education. Code smells can be efficiently avoided if programmers are educated to not code in a certain way since the beginning of their exposure to the software programming.

**4.1.2 Locating and Correcting Code Smells**
Since it is not easy to design a programming question to lead the programmers into coding a particular code smell, pre-implemented Java classes have been prepared with code smells planted for the participants to indicate them. At first, a large Java file with all five code smells planted was presented to the participants. There were multiple instances of all code smells in the file. However, according to participant feedbacks, this method was inefficient. The file being too long and having multiple instances of the same code smells became too irritating because the spraying action would not stop until all code smells were fixed. The purpose of this part of the test is to determine whether, after what the participants have learned in part one, they would spend more effort on preventing code smells and be more vigilant in detecting the code smells. If the participant was able to locate code smells in a short file, there was no need to do it in a long file. This test was then refined to have only four short Java programs prepared and each of them has only one instance of one code smell planted. The current Smellware release is capable of detecting five code smells, but only four Java programs are presented to the participants for code smell matching to prevent the users from guessing randomly for the last Java file. The four Java example files have the code smell errors pre-planted as shown below in Table 2.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Code Smell Planted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1.java</td>
<td>Method argument list too long</td>
</tr>
<tr>
<td>Test2.java</td>
<td>Identifier name too long</td>
</tr>
<tr>
<td>Test3.java</td>
<td>Method too long</td>
</tr>
<tr>
<td>Test4.java</td>
<td>Identifier name too short</td>
</tr>
</tbody>
</table>

The users are asked to turn off the visual warning before loading the test files and only to turn it on if they have trouble locating the code smell errors. Out of the all participants who needed to turn on visual aid in part one, only 20% of them turned on the visual aid for the test files. This shows that for most participants, Smellware achieved its purpose helping developers avoid code smells.

**4.1.3 Questionnaire**
A Likert style questionnaire is given to the participants at the end of the user studies. The questions are designed to quantify how effective the participants find Smellware is and how helpful it is to combine olfactory and visual feedbacks for the communication between human and technology. The participants were also asked to input any additional comments that they would like to add. In Table 3, the questions and the average answers are listed, with 1 being “Strongly Disagree” and 5 being “Strongly Agree”.

<table>
<thead>
<tr>
<th>Question</th>
<th>Average</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does the emitted smell motivate you to remove code smell errors? and why?</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td>Does the pop-up visual warning motivate you to remove code smell errors? and why?</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Does the addition of pop-up visual warnings help to locate code smell errors?</td>
<td>4.2</td>
<td></td>
</tr>
<tr>
<td>Does Smellware help in preventing code smell errors to be made in the future?</td>
<td>3.7</td>
<td></td>
</tr>
<tr>
<td>Do you feel the smell intensity varying when you are not fixing the code smell errors?</td>
<td>3.7</td>
<td></td>
</tr>
<tr>
<td>Did you intentionally ignore code smell errors?</td>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td>Are you likely to use this again for future programming?</td>
<td>3.1</td>
<td></td>
</tr>
</tbody>
</table>

The most significant question is the first question, because it shows whether Smellware achieved its purpose of motivating the developers to avoid code smells. However, it is to be noted that the answers were either close to “strongly agree” or “strong disagree”, never “neutral”. The “agree” and “disagree” answers were evenly distributed and led to a neutral average result. The reasons why some participants found Smellware motivating for fixing code smells are:

- The spraying mechanism action is startling.
- The strong smell is overwhelming and irritating.
- The odour emitted, although pleasant, is probably unhealthy if inhaled too much.
- The uncontrollable change in the surrounding, (ie. The odour surrounding the participant) cannot be ignored easily, unlike a list of warning which can be overlooked.

The reasons why some participants found Smellware not motivating for fixing code smells are:

- The Air-Wick Freshmatic odour is pleasant and bearable.
- The participant is unwilling to change the code style that he or she is used to for years. He or she would choose to disable all code smell detections.

In the additional comment section, many participants suggested the spray mechanism to be less frequent, less amount or less intense odour. A participant struggled with using Smellware because he was allergic to strong fragrances.
The pop-up visual warnings were a lot more acceptable by the participants because they helped locating the code smells much faster and less irritating.

4.2 Discussion
After the participants completed the usability study, they were interviewed randomly with any further comments and evaluations. Most participants found the experience stimulating and interesting. It was suggested to deploy this tool for educational purposes, rather for professional users who are already under a lot of stress while developing software. As it has been seen from the first part of the exploratory study, the reason behind why a developer would implement with code smell violations is out of long term habits. If Smellware could have been deployed for beginner programmers and helped them with building up effective coding style, then the code smell violations could be more efficiently avoided. It is still up for further discussion whether it is better to use unpleasant odours, if they were available or reproducible without hazardous substances. It might able to drive those who appreciated the Air-Wick Freshmatic odour to be more motivated into fixing code smells, but it might also create more resentment from the people who found it irritating. The pleasant odour was irritating only to a point that the motivated participants were willing to fix the code smells, however, if the odours were to become unpleasant, those participants might not want to deploy this tool at all. Smellware is also not suitable or practical for the users who have allergy to fragrances.

5. FUTURE WORK AND CONCLUSION
Overall, Smellware is able to motivate developers to fix code smells or to avoid them in the future. Smellware achieved its objective to communicate with the users through olfactory feedbacks. However, there is still a lot to be improved. It currently supports 5 code smell detections and detecting them in programs written in Java. It does not compile or color-distinguish commands like most IDE’s do. Smellware could evolve to support different programming languages or as a plug in for other development IDE’s. By being integrated with the other IDE’s, Smellware will be able to compile and run programs. If Smellware could be configured for different programming language, it could be more widely used for different fields of software developments. As the participants have suggested during the user studies, Smellware could spray less frequently or spray different odours for different categories of code smells. Although humans’ ability to distinguish the different smells when they are all mixed together in air is low, there are some smells that are so natural and distinctive to humans that they can easily tell them apart even when the smells are mixed together, such a citrus smell and a wood smell.

6. REFERENCES
http://www.nature.com/neuro/journal/v14/n11/full/nn.2926.html
http://ieeexplore.ieee.org/xplore/library.ubc.ca/stamp/stamp.jsp?tp=&arnumber=1353185
http://ieeexplore.ieee.org/xplore/library.ubc.ca/stamp/stamp.jsp?tp=&arnumber=1353185
http://medgadget.com/2006/05/novel_olfactory.html