BuildBot: A Robotic Software Development Monitor in an Agile Environment

Ablett, Ruth; Maurer, Frank; Sharlin, Ehud; Denzinger, Joerg

BuildBot: A Robotic Software Development Monitor in an Agile Environment

Ruth Ablett, Frank Maurer, Ehud Sharlin, Jörg Denzinger
Department of Computer Science, University of Calgary
{ablettr, maurer, ehud, denzinge}@cpsc.ucalgary.ca

Abstract

In this paper, we describe BuildBot, a robot developed to assist with continuous integration of a software build in Agile development teams. BuildBot can interact physically with individual members of the team and be an active part of the development process by bringing together human-robot interaction with human group dynamics and knowledge about software engineering concepts.

This paper describes the design and implementation of a robot that can sense virtual stimuli, in this case the state of a software build, and react accordingly in a physical way. By increasing awareness of the state of the software build, BuildBot assists in the self-supervision of teams.

1. Introduction

Robots exist in both the virtual domain of computers and the physical realm with humans, and therefore offer an effective interface between the two. We believe that Agile software engineering, with its human-centric practices, can benefit from the use of a robot as an assistant.

There are two Agile concepts central to the idea of this paper: *continuous integration* and *knowledge sharing*. In the first, *continuous integration*, every time new code is checked into the shared source code repository, the entire software is build, deployed and tested against a suite of automated regression tests. Ideally, these kinds of check-ins occur frequently. *Continuous integration* and frequent check-ins of tiny increments ensures that existing functionality is not broken by the new code. A simple bug which may take only a few minutes to repair in the early stages may end up costing huge numbers of person-hours once more code has been written around it. The practice of *continuous integration* encourages clean design and prevents problems later on in the development process, since bugs can be caught and fixed earlier on.

After finding newly checked-in code in a repository, a *continuous integration* server builds and deploys the software. It then executes all tests. If one of these tests fails, the *continuous integration* server sends out an e-mail to the last person or persons who checked-in code. This entire process is occurring in the virtual domain.

Knowledge sharing is the second Agile concept important to this paper. Excepting face-to-face communication, *knowledge sharing* is achieved in agile teams through information radiators which are openly displayed artifacts or other means by which developers can gain knowledge about the project without explicitly seeking it [1].

In human-computer interaction, a related idea is *ambient data displays* (also known as calm technology). Ambient data displays present information in such a way that *"one does not even need to be looking at it or near it to take advantage of its peripheral clues."* [2] These types of displays are often very simple and use color, sound and motion to convey information.

A robot has the potential to be an effective assistant to an agile team. The robot is unique in that it possesses the ability to physically respond to virtual stimuli, bringing awareness information from the digital realm into the physical and vice-versa. We believe robotic embodiment of the state of the software build can help an Agile team collaborate more effectively, especially if the robot can physically interact with the team members. In this environment, BuildBot would act as a *dynamic information radiator*, that changes according to the circumstances, rather than a static one, that tend to get ignored [1].

Ambient data can in this way be applied to *continuous integration*. Not only does it provide important information pertaining to the current build status, but also keeps the team focused on the goal of finishing the project and makes sure the different parts of the software integrate properly. It also gives the team a sense of accomplishment that the project is being tested, and increase accountability toward the project.

2. Related Work

Agile methods (such as eXtreme Programming [3] or Scrum [4]) refer to human-centric software engineering methodologies that advocate developing high-quality software in short iterations. Agile methods rely heavily on automated regression testing to ensure internal software quality.

According to Kent Beck, "*an interested observer should get a general idea of how the project is going in 15 seconds. He should be able to get more information about real or potential problems by looking more closely*." [5] Ambient data is one way that this is achieved. The benefits are not only felt by outside observers, but by the team members themselves. They can quickly assess the state of the project, providing encouragement or an incentive to improve.

Savoia [6] has created an ambient feedback device, known as *Java Lava Lamps*, that helps the entire team keep track of the overall build status. There are two lava lamps involved, one green and one red. The green lamp represents a successful build, and the red lamp a build in which one or more tests have failed. The length of time is also important; because the lamps take a few minutes to warm up, the presence of bubbles means that the lamp has been on for at least several minutes. For a green lamp, this is a favorable state, however, bubbles in the red lamp indicate the presence of a problem.

The continuous integration server is connected to a controller that sends a signal, 0 or 1, to an electrical power outlet receiver. The receiver has two lava lamps, one green and one red, connected to it, and only one of which has power at any given time. In the case of a successful build, the green lamp would be turned on. In the event of a build break, the red lamp would be turned on. Because lava lamps take a few minutes to heat up, it was possible to tell how long the build had been broken, judging by the bubbles of lava on both lamps; whether a build had just been broken, or if it had been that way for a while. Further, the developers were trying to fix the problem before the lamp heated up – this voluntary, playful behavior created a self-supervision of developers instead of having to involve a manager.

We now present BuildBot, a robotic software development monitor in an Agile environment, starting with an explanation of its contribution to Agile software development, the overall high-level design, and the implementation overview.

3. Design Approach

The main design goal behind BuildBot was to create a fun tool that will help an agile team to fix a broken build as quickly as possible.

If an update was successful, meaning the new code changes properly integrate with the build and the entire software passes all the tests, BuildBot provides positive feedback to the entire team by happily barking from its home base. By displaying its contentment, it congratulates everyone in the team for their work in keeping the build stable. A successful build is a team effort, and instead of congratulating every member individually, the BuildBot will show its contentment through ambient feedback.

If the code changes cause one or more of the tests to fail, the build is considered broken. In this case, the BuildBot would walk to the individual who had uploaded the bad section of code and display to the team that it is not happy with that one person, in a friendly, funny, and playful way (Figure 1). Even the timing of this walk is important – BuildBot's deliberately slow walk serves two purposes. It alerts the team to the broken build, creating a kind of playful tension as the team members wonder if the dog will be coming to their own desk. It gives the perpetrator time to fix the offending section of code before it arrives at their desk.

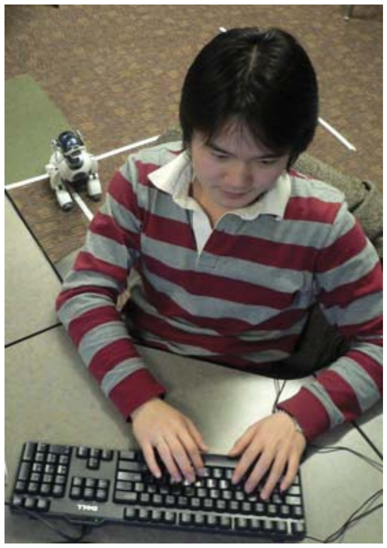


Figure 1. BuildBot arriving at a developer's desk.

Giving the responsible individual a lighthearted and friendly 'punishment' introduces more targeted accountability. It also avoids potentially unpleasant confrontations (for example, between a team leader and a developer) and is instead a mild social incentive to fix the problem without having to involve a manager.

4. Server Implementation

There are four different components, excluding the robot, involved in communicating with BuildBot. The architecture and inter-component communication are shown in Figure 2 below.

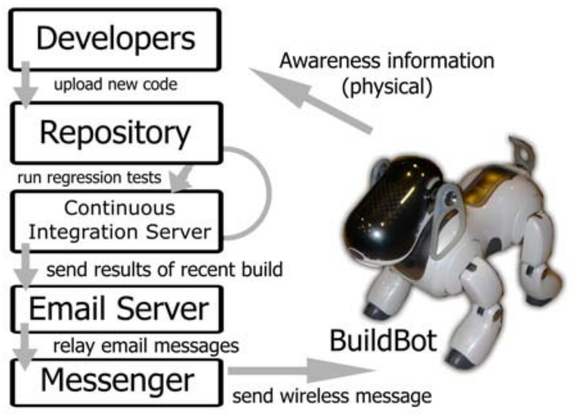


Figure 2. BuildBot system architecture.

The *repository* contains the software currently being developed by the software team. Whenever a change is uploaded to the shared code repository by a team member, the *continuous integration* component runs a script that integrates the entire team's code together.

With BuildBot, the build results are sent via email to the *email server* which stores these messages for retrieval. The *email server* can be external, for example, Google's Gmail or Yahoo Mail.

The final component, the *messenger,* is timed to check email from the *email server* and to download the latest message regarding the build, sent from the *repository*. If there is a new message on the *email server* regarding the build, the *messenger* will open a socket to the robot and send it the appropriate command over the wireless connection. This message will contain the build status, name of the build-breaker, and other pertinent information. The robot receives these wireless signals and acts upon them.

5. Robotic Interface Implementation

In order to create BuildBot, it was necessary to have a robot with several capabilities and features. To physically interact with the team members, the robot would need the ability to walk with stability. As an extension to this, it would also need to gain information about its environment, through vision. In order to act as an information radiator, sound or LED lights (or both) would be necessary. To communicate with the continuous integration server, the robot would need some kind of wireless capability. Using wires would be annoying and dangerous, especially if the dog was on its way to a desk. Lastly, the robot must be easily programmable with a good API.

Fortunately, advances in robotics have resulted in robots that are powerful and compact enough to make BuildBot feasible. The robot used for the BuildBot's implementation is the most recent model of the Sony AIBO robot dog, the ERS-7.

5.1. Hardware

The AIBO walks on four legs and can move its head and legs in various directions. It also comes equipped with a camera, two microphones, a speaker, LEDs in various colors, the capability to communicate over wireless LAN, touch-sensitive buttons, and sensors for temperature, vibration, distance and acceleration. The robot is also zoomorphic, in the shape of a cute puppy, and in that it adds to the feeling of fun and lightheartedness.

5.2. Robotic Behavior

BuildBot's behavior is event-based. The robot will stay in its starting position until it receives a message from the *messenger* regarding the build status. If the build is broken, BuildBot will get up and begin to make its way to its goal.

When creating the first prototype of BuildBot, it was necessary to simplify potentially complicated details such as the vision and way-finding algorithms. For the sake of simplicity, electrical tape was used to form the lines on the floor, leading to each team member's desk in a tree structure. These lines act as a navigation guide, linking BuildBot's base station to this network of lines and allowing it to walk to a developer's desk via the simplest route possible.

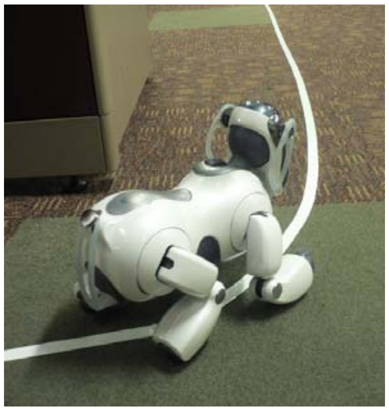


Figure 3. BuildBot following a line on the floor.

When walking to a team member's desk, the dog follows the line (Figure 3) using its camera, which is mounted on the end of its nose. The lines on the floor have junctions which branch off at 90 degrees.

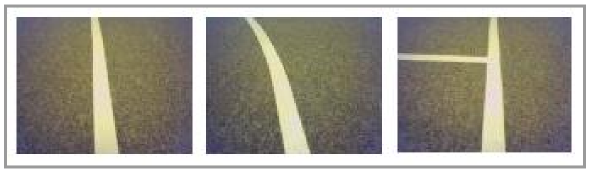


Figure 4. A straight line, a curved line, and a junction.

When walking, the dog keeps track of these junctions and consults an internal map, in the form of a matrix, which gives directions of how to get to each workstation based on the junctions it encounters.

The line detection can handle curves; however, if the curve is at or near 90 degrees, the algorithm detects a junction. Once BuildBot reaches the end of a line, it has arrived at its goal. BuildBot then looks up and gently 'punishes' the team member by barking and growling. This robotic reprimand will cease only when the build is fixed.

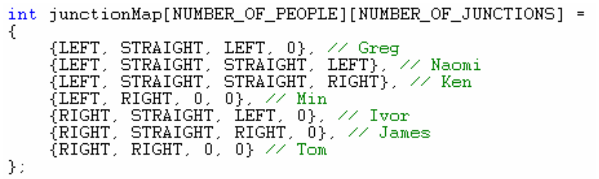
5.3 Low-Level Implementation

The robot's behavior, including LEDs, movement, vision and sound, is controlled using the C++ programming language through the Tekkotsu development framework developed and maintained by Carnegie Mellon University [7].

The first step in implementing BuildBot's behavior was to program the robot to follow a single line on the floor. This was done by acquiring the vision stream via Tekkotsu and thresholding the intensity values to create a binary array.

The line detection algorithm is quite straightforward – every 500 milliseconds, the midpoint of the binary one values (representing the line) in the vertical middle of the binary array is computed. If this midpoint is off center in the field of view, the robot will compensate by rotating itself accordingly. The robot can follow curves, as long as they are not too sharp (if the curve is at or near 90 degrees, BuildBot will either miss it or count it as a junction and may arrive at the wrong desk). The line color must be sufficiently contrasting with that of the floor for this algorithm to function properly.

After implementing the line-following component, the next step was to include support and detection for junctions. The internal map inside BuildBot contains directions on how to get to each team members' desk by following the junctions. For example, in Figure 5 below, in order to get to Naomi's workstation, BuildBot would turn left at the first junction, go straight at the second, straight at the third, and turn right at the fourth, following the line until it ends, meaning it will have arrived at her desk.



```
int junctionMap[NUMBER_OF_PEOPLE][NUMBER_OF_JUNCTIONS] =
{
    {LEFT, STRAIGHT, LEFT, 0}, // Greg
    {LEFT, STRAIGHT, STRAIGHT, LEFT}, // Naomi
    {LEFT, STRAIGHT, STRAIGHT, RIGHT}, // Ken
    {LEFT, RIGHT, 0, 0}, // Min
    {RIGHT, STRAIGHT, LEFT, 0}, // Ivor
    {RIGHT, STRAIGHT, RIGHT, 0}, // James
    {RIGHT, RIGHT, 0, 0} // Tom
};
```

Figure 5. An example of BuildBot's internal map.

A '0' in the next junction means that there are no more junctions to consider – when the robot reaches the end of the line, it has arrived at the workstation. The maximum number of possible junctions to reach any workstation is calculated, and this becomes the width of the array. The 0s act as placeholders – due to the absence of array bounds-checking in the C++ programming language. This prevents an *array out-of-bounds* exception, which will crash BuildBot.

After intersection detection, the major components of the robotic implementation were completed. Aesthetic components, such as posture and barking, were added afterward.

For the *continuous integration* server component, the script was written using Apache Ant. It includes the sending of an email, which contains the person who last updated code to the repository, as well as the build status.

The *messenger* component was written using Java. It retrieves email messages from Google's Gmail, and parses the most recent message for the build status update. It then sends the build status (and, if broken, the last person to upload code) to BuildBot through a wireless socket.

6. Preliminary Evaluation

A preliminary evaluation was performed with a peer group. A formal, thorough user study was not performed; however, BuildBot's behavior was demonstrated in a user-less environment. The network of taped lines was placed on the floor, one computer emulated the broken build and the robot successfully reached the appropriate desk in almost every trial run.

Within the peer group, the initial impression was that BuildBot has the potential to be useful, but there are a few improvements that were suggested.

The threshold value, contained in the C++ code, must be manually changed to adapt to differences in lighting. Moreover, BuildBot cannot function in a workspace with sufficiently fluctuating lighting.

While BuildBot is quite accurate with detecting junctions properly, there is a possibility of an inaccurate junction count if there is an object on the floor with a similar color intensity to the line.

7. Future Work

Our major future effort for this project is to perform a more formal evaluation with a group of developers, even amongst a peer group, and to determine if BuildBot is actually helping the team.

The robot in its current implementation is not able to recharge its own batteries at its base station. This will be essential if BuildBot is to be used for more than a few hours. BuildBot should be able to find its own power station using its camera. As an extension of this, the robot should use an improved planning algorithm to determine whether it is possible, given its remaining battery power, to reach a workstation. In the event that its remaining battery power is not sufficient, it would return to the base station and simply bark while recharging. Also, BuildBot must be able to turn around and walk back to its base station if the build is fixed in time.

Another issue that was mentioned was that BuildBot does not recognize if a developer actually sits at her workstation. Therefore, future work also includes implementing some way of detecting this.

Other aspects of the system that may be implemented in the future include different mechanisms to give the dog a more lifelike demeanor – examples of this include 'sniffing' the air menacingly, or implementing a more realistic-looking gait (the default Tekkotsu walk sees the dog walking on its elbows).

8. Conclusion

BuildBot has a great potential to play many roles in an Agile environment. It keeps the development team focused on the task at hand, that is, to develop software whose components seamlessly work together. We believe that BuildBot may give the team a sense of accomplishment and the developers will feel that the project is progressing. The robot has the potential to increase morale in this way and to keep the environment more fun for the programmers. Anyone who sees the dog can instantly know how the project is progressing. And finally, the sound of the dog approaching a team member's workstation will give that developer a social incentive to fix the build – avoiding situations of potential embarrassment, resentment or discomfort.

We believe that team practicing Agile Methods could benefit from a robot that could access this virtual information and express it in a physical way, beyond simple ambient data. Robots actively exist in both the physical and virtual realms and provide an interface between the two, and therefore they are well-suited for this task.

9. References

[1] A. Cockburn, *Agile Software Development: The Cooperative Game*, Agile Software Development Series, 2001, pp 70-80.

[2] M. Weiser, J. Seely Brown. Designing calm technology, 1995 http://www.ubiq.com/weiser/calmtech/calmtech.htm

[3] K. Beck, *Extreme Programming Explained*, Addison Wesley, 2000.

[4] Linda Rising, Norman S. Janoff, "The Scrum Software Development Process for Small Teams," *IEEE Software*, vol. 17, no. 4, pp. 26-32, Jul/Aug, 2000.

[5] S. Baker. *Agile in Action: Informative Workspace*. http://www.think-box.co.uk/blog/2005/08/informative-workspace.html.

[6] A. Savoia, "On Java Lava Lamps and Other eXtreme Feedback Devices", 2004.

[7] E. Tira-Thompson, N. Halelamien, J. Wales, D. S. Touretzky. *Tekkotsu: Cognitive Robotics on the Sony*

AIBO. http://www.cs.cmu.edu/~tekkotsu/media/tira-thompson-iccm04.pdf.

Project web site at http://www.tekkotsu.org.