

# BuildBot: Robotic Monitoring of Agile Software Development Teams

Ruth Ablett, Ehud Sharlin, Frank Maurer, Jörg Denzinger, and Craig Schock

Department of Computer Science, University of Calgary, Alberta, Canada  
e-mail : {ablettr, ehud, maurer, denzinge, schock}@cpsc.ucalgary.ca

**Abstract** - In this paper, we describe BuildBot, a robotic interface developed to assist with the continuous integration process utilized by co-located agile software development teams. BuildBot's physical nature allows us to engage the agile software development team members through vision, hearing and touch. In this way, BuildBot becomes an active part of the development process by bringing together human-robot interaction, human group dynamics and software engineering concepts through a number of interaction modalities.

In this paper we describe the design and implementation of the BuildBot prototype, a robotic interface that can sense virtual stimuli, in this case the state of a software build, and react accordingly in a physical way via vision, sound and touch. We present an early evaluation comparing BuildBot to two other tools used by an agile team to monitor the continuous integration process. We also show preliminary results indicating that BuildBot may be more noticeable to the developers and contribute to a fun and lighthearted atmosphere.

We argue that by increasing awareness of the state of the software build, BuildBot can assist in the self-supervision of agile software engineering teams and can help the team achieve its goals in a more engaging and sociable manner.

## I. INTRODUCTION

Robots can easily interact, perceive and act in both the virtual domain of computers and the physical realm of humans, and therefore promise to offer an effective interface between the two. Robots allow for visual, audio and tactile interaction modalities, which can be used to enhance the interaction experience humans have with either physical or virtual entities. We believe that agile software engineering, with its human-centric practices, can benefit from the use of a robot that can assist with two major agile concepts: *continuous integration* and *knowledge sharing*.

*Continuous integration* is used by agile teams working on a common piece of software. Every time developers check new code into the shared source code repository, the entire software is built, deployed and tested against a suite of automated regression tests. Ideally, these kinds of check-ins occur frequently. *Continuous integration with regression testing* and frequent check-ins of tiny increments ensure that existing functionality is not broken by the new code. A simple bug which may take only a few minutes to repair immediately after it is introduced into the code base may end up costing significant numbers of person-hours once more code has been written around it. *Continuous integration* encourages clean design and prevents problems later on in the development process, since bugs can be caught earlier [9].

After finding newly checked-in code in a repository, a *continuous integration* server builds and deploys the software, then executes all tests. If one of these tests fails, the *continuous integration* server sends an indication to the last person who checked-in code. This entire process occurs in the virtual domain. It is the responsibility of that team member to ensure that a reported bug is resolved immediately, for example by reverting to an older version or by fixing the problem in another way. If the bug fix is delayed, other team members might synchronize their code with a broken version of the system – resulting in even more effort required to resolve the problem. We are interested in an engaging and sociable monitoring technique that will encourage team members to “do the right thing”, avoiding dryer and perhaps more threatening approaches such as those requiring the project manager’s involvement.

Excepting face-to-face communication, *knowledge sharing* is achieved in agile teams through information radiators. These are openly displayed artifacts or other means by which developers can gain knowledge about the state of the project without explicitly seeking it [1]. A multimodal system can enhance the way the knowledge is shared without becoming a distraction to the intended audience. Human-Computer interaction has a related concept, *ambient data displays*, which present information so 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.

We believe a robot has the potential to be an effective assistant to an agile team. Robots possess the ability to physically respond to virtual stimuli, bringing awareness information from the digital realm into the physical and vice-versa. Robots are equipped with input sensors and output actuators that can allow them to become a believable physical component of the agile team physical setting. We believe robotic embodiment of the state of the software build can help an agile team collaborate more effectively, especially if the robot is allowed to physically interact with the team members. In this environment, a BuildBot would act as a *dynamic information radiator*, that changes according to the circumstances, rather than a static radiator, which tends to get ignored [1]. Information exchange thus occurs in an ambient manner, in the physical context of humans [12].

Our BuildBot is used as an ambient & interactive data display and is applied to *continuous integration*. Not only does it provide important information pertaining to the current build status, but it also keeps the team focused on the

goal of finishing the project and verifies the different parts of the software integration properly. In addition, BuildBot increases accountability and enhances the team’s sense of accomplishment because the project is being successfully tested.

In Section II, we will describe previous work in the sub-domain of agile development and related human-robot interaction (HRI) efforts. In Section III, we will introduce BuildBot, our proposed solution. Section IV describes the high-level design approach for the overall BuildBot system, and in Section V, the low-level implementation of the system is explained in detail. In Section VI, we will present the results of a preliminary evaluation performed on a group of participants comparing BuildBot with two other build notification mechanisms. Future work is outlined in Section VII.

## II. PREVIOUS WORK

Kidd [11] discusses different uses for sociable robots. He describes the transition between using robots as tools or simple entertainment devices (such as an industrial robot or a children’s toy) to becoming partners that interact with humans. While still limited in their capabilities, emerging robots are designed to interact with their users sociability, and to integrate in a social setting as peers and teammates [11-13]. The goal of BuildBot is to develop a near-autonomous robotic partner or colleague, designed and tuned for a focused agile software development social setting.

A study by Saff and Ernst [10] evaluated continuous integration when used by a single developer to ensure new code passed regression and unit tests. They found that continuous integration had a positive effect on the completion of programming tasks. Their study shows that individual developers benefit from continuous testing of their own code. However, we are interested in ways that different Continuous Integration tools affect Agile teams as a whole. Agile methods (such as eXtreme Programming [3] or Scrum [4]) refer to human-centric software engineering methodologies that advocate the development of high-quality software using 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] This can be achieved using ambient data displays. The benefits can be seen by outside observers and the development team members. They can quickly assess the state of the project, and this provides encouragement and an incentive to improve.

Savoia [6] has created an ambient feedback device for continuous integration, the *Java Lava Lamp*. There are two lava lamps involved, one green and one red (Figure 1). The green lamp represents a successful build, and the red lamp a build in which one or more tests have failed. Timing is also significant; because the lamps take a few minutes to warm up, the bubbles mean the lamp has been on for at least several minutes. For a green lamp, this is good, however, bubbles in the red lamp indicate a problem.



Fig.1 The two Java Lava Lamps used to display the build status.

To integrate Java Lava Lamps into the development process, the continuous integration server is connected to a controller that sends a signal, 0 or 1, to an electrical power outlet receiver. The receiver is connected to two lava lamps. Only one of which has power at any given time. In the case of a successful build, the green lamp is turned on. In the event of a build break, the red lamp is turned on. Since lava lamps take a few minutes to heat up, it is possible to tell how long the build has been broken. This delay affords the developers the opportunity to fix a broken build before the lava lamp indicates that the build has been broken for a long time. The playful nature of the lava lamp encourages the developers to fix the build without requiring the involvement of the project manager.

While this approach is simple, it only makes use of visual information and this requires developers to poll the ambient display. A better solution would eliminate the need for polling and instead utilize information osmosis. In this way, information would become available for developers without causing a distraction.

## III. BUILDBOT

The main design goal behind BuildBot was to use the agile team’s collective awareness to create an engaging and fun tool that helps the team fix broken builds as quickly as possible.

If new code integration causes one or more tests to fail, the build is considered broken. In this case, BuildBot walks to the individual whose code is responsible for test failure and displays to the team that it is not happy with that developer, in a friendly, funny, and playful way (Figure 2). The timing of this walk is important – BuildBot’s deliberately slow and dramatic walk serves two purposes. It alerts the team to the broken build via ambient sound and visual cues. By sending an e-mail to the responsible individual, BuildBot alerts him/her to the failed tests and gives him/her time to fix the problem. After a few minutes, the dog begins its walk to the workstation. This creates a playful tension as the other team members wonder where the dog is going.

By giving the responsible individual a lighthearted and friendly ‘punishment’, BuildBot introduces more targeted accountability. It avoids potentially unpleasant confrontations (for example, between a team leader and a developer) and it introduces a mild social incentive to fix the problem without having to involve a superior.



Fig.2 BuildBot arriving at a developer's desk.

#### IV. DESIGN APPROACH

There are four different components, excluding the robot, involved in communicating with BuildBot. The architecture and inter-component communication are shown in Figure 3 below.

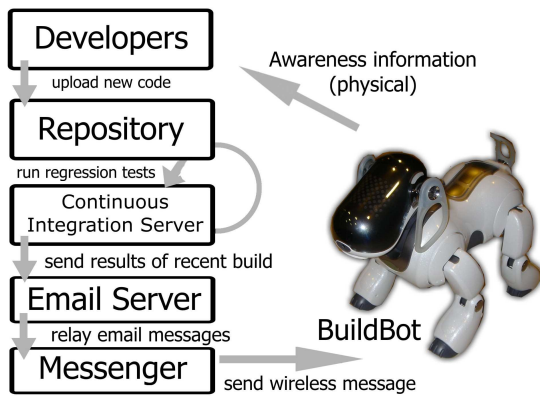


Fig.3 BuildBot system architecture.

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. 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.

The final component, the *messenger*, is timed to check email from the *email server* and to download messages sent from the *repository*. If there is a new message on the *email server* regarding the build, the *messenger* opens a socket to the robot and sends the appropriate command over the wire-

less connection. This command contains the build status, name of the build-breaker, and other pertinent information.

#### V. ROBOTIC INTERFACE INFORMATION

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 needs to be able to walk with stability. It also needs to gain information about its environment through vision. In order to act as an information radiator and provide ambient build information, sound or LED lights (or both) are necessary. To communicate with the continuous integration server, the robot needs wireless capability. Lastly, the robot must be easily programmable.

Fortunately, advances in robotics have resulted in robots that are powerful, affordable 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 [8].

##### A. Hardware

The AIBO is ideal for an ambient information radiator as it combines motion, visual, audio, and tactile components. It can move its head and four legs, and comes equipped with a camera, two microphones, a speaker, LEDs in various colors, wireless capability, touch-sensitive buttons, and sensors for temperature, vibration, distance and acceleration. The robot is also zoomorphic, in the shape of a cute puppy, and that adds to the feeling of fun and lightheartedness.

##### B. Robotic Behavior

BuildBot's behavior is event-based. The robot stays in its starting position until it receives a message from the *messenger* regarding the build status. If the build is broken, BuildBot identifies a developer by moving to his/her workstation.

When creating the first prototype of BuildBot, it was necessary to simplify complex details such as the vision and planning algorithms. In order to allow the dog to walk to the team member's desk, we designed a vision algorithm analyzing the streaming video from the robot's eye (a camera integrated into the end of its nose). For simplicity, a tree structure of guide-lines were placed on the floor. In this tree, each leaf terminated at a developer's desk and this allowed BuildBot to easily navigate to a specific workstation using the the simplest route. The lines on the floor have junctions which branch at 90 degrees (Figure 4). Perpendicular junctions simplify the junction detection algorithm so that BuildBot can easily distinguish junctions from curves.



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

When walking, the dog tracks junctions and consults an internal map, in the form of a matrix, which gives directions to each workstation. For example, in Figure 5 below, in order to get to Naomi’s workstation, BuildBot will turn left at the first junction, go straight at the second, and turn right at the third. A ‘0’ in the next junction means that there are no more junctions.

```
{LEFT, STRAIGHT, LEFT, 0}, // Greg
{LEFT, STRAIGHT, RIGHT, 0}, // Naomi
{LEFT, STRAIGHT, STRAIGHT, LEFT}, // Kenji
{LEFT, STRAIGHT, STRAIGHT, RIGHT}, // Wei
{LEFT, RIGHT, 0, 0}, // Ivor
{RIGHT, STRAIGHT, LEFT, 0}, // James
{RIGHT, STRAIGHT, RIGHT}, // Ryan
{RIGHT, RIGHT, 0} // Tom
```

Fig.5. An example of BuildBot’s internal map.

Once BuildBot reaches the end of a line and has passed the correct number of junctions, 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 when the build is fixed, or when the robot senses a touch on its head sensor.

### C. Low-Level Implementation

The robot’s behavior is controlled using the C++ programming language and the Tekkotsu development framework [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 straightforward – every 500 milliseconds, the midpoint of the binary one values (representing the line) in every horizontal row of the binary array is computed and the mean of these values is computed. If this midpoint is off center, the robot will rotate itself accordingly. The robot can follow curves, as long as they are not too sharp. The line color must be sufficiently contrasting with that of the floor – the best choice of line color is fluorescent pink. Pink has the highest intensity and is sufficiently rare that there is a reduced chance of the robot mistaking some external brightly-colored object as the line. In our test environment, our lines were eggshell white and the line tracking algorithm proved to be stable enough for our purposes, even under the inconsistent and changing light conditions in our lab.

After implementing the line-following component, the next step was to include support for the detection of junctions. After the line direction has been detected, BuildBot attempts to detect an intersection to the left, right, or in both directions.

The internal map inside BuildBot contains directions on how to get to each team members’ desk by following the junctions. For example, a certain developer’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.

There are several circumstances under which the robot can determine that it has become lost. If the robot loses the line (or if the lighting is too dim), the robot will check if it

has passed the correct number of junctions to be at the correct desk, if not, the robot is assumed to be lost. Also, if the robot detects a type of junction at which its next turn instruction is not possible (for example, to go left at a right-only junction), it also becomes lost. If the robot encounters a junction whose junction number is higher than the number of junctions (a result of perhaps having wrongly identified a junction where there was none, for example), BuildBot assumes it is lost. When it becomes lost, BuildBot stops and calls for help by playing a quiet whining sound until it is placed back on its power station.

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

Occasionally, a developer will fix the broken build before BuildBot arrives at their workstation. In this case, the robot is able to turn around and return to its starting point, following the line. The algorithms for going to a desk and returning to the power station are almost totally identical, except for that the junctionCount variable is decremented rather than incremented. Also, when turning, BuildBot will turn in the opposite direction to its map matrix when returning to the power station.

Similarly, if the robot senses its battery power is at 30%, it will immediately return to its power station. It will continue to growl until it senses the build has been fixed.

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 BuildBots email messages from Google’s Gmail, and parses the latest message for the build status. It then sends the build status (and, if broken, the last person to upload code) to BuildBot through a wireless socket.

## VI. PRELIMINARY EVALUATION

Although a formal, thorough user study was not yet performed, two preliminary evaluations were done with a peer group. The first was done with 4 developers, and the second a group of 10 developers. Both groups were a mix of male and female graduate students between 23 and 35 years of age. The experiment was performed in a laboratory with chest-level walls and with a layout as illustrated in Figure 6 below.

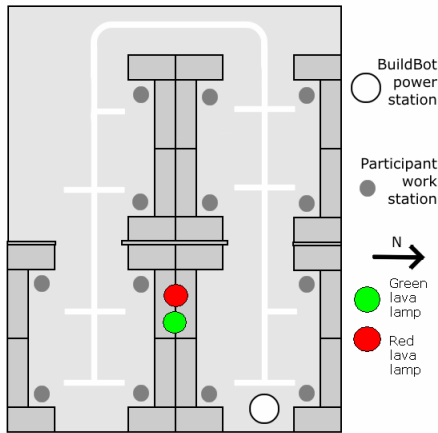


Fig.6 The laboratory layout for the study.

Even though the students were not involved in the same project, the purpose of this study was to assess how effective certain notification mechanisms were, and what effect they had on the participants. These notification mechanisms would be potential candidates for use in a development environment for notifying the developers of a build break.

The case in which the build is successful, where BuildBot does not move from its starting location, was not tested with developers. This is because the absence of the robot's movements is an indication that the build is fine. This study was aimed at determining the effectiveness of different mechanisms to notify developers that there is a problem.

Three such mechanisms were compared: email (only), the lava lamps (see the Previous Work section), and the deploying of BuildBot to one of the developers (plus e-mail).

The first notification mechanism, email, was sent with the message "BUILD BROKEN" to all developers. No physical notification of any kind accompanied this email.

The lava lamps were displayed in a prominent position in the laboratory so that any developer could see it from their desk (some had to turn around to see them, as they were facing the opposite direction). They were switched off or on (switching the illuminated lamp from red to green or vice-versa), and this was combined with an email alerting all developers. The mechanism controlling the lava lamps' power source makes a clicking sound when the switch occurs, so this also alerts the developers via sound.

The third mechanism, BuildBot, was deployed to a developer, selected randomly, and this was accompanied by an email only to that developer.

While the students worked at their workstations on their individual tasks (unrelated to each other), a notification (of one of the three kinds mentioned above) was sent out. The participants were instructed to notify the study facilitator privately, using an instant messaging program when they noticed the robot, received an email, or noticed the lava lamps had changed their state. Sending an instant message ensured that no participants became aware of a change through this communication.

This study was performed on two different days. On the first day, the four participants had not seen the robot or the lava lamps in action. On the second day, the ten participants had seen the robot and lamps in action before and were used

to them. These two studies were done on these two days to measure the effect of novelty on the developers.

The results of the first evaluation were as follows: When the email was sent, only the two who had email notification programs noticed (within a reasonable amount of time). In one case, just after the last run, one person checked for email and saw all the email notifications at once.

Generally, when the lava lamps were switched in conjunction with the email, only one person heard the click of the power switch (and this person did not have a clear view of the lava lamps from their desk). Of the people who responded to the change, half had noticed the lamp had changed, and half had only received the email. The two respondents who responded to the sight of the lava lamps were getting up and walking around the lab when they noticed the lamps had changed.

When BuildBot was deployed, all four participants were alerted by the sight or sound of the robot walking. Three out of the four participants noticed when the robot was near their desks.

The participants didn't seem to notice the lava lamps, even though they are in a prominent location. However, when BuildBot was deployed, the combination of sight and sound resulted in everyone noticing the robot when it was near their desk. As a result of this, every one of the participants agreed that the robot was a distraction. However, every one of the participants also either agreed or strongly agreed that the robot also contributed to a fun atmosphere. Qualitatively, we observed that the energy in the room increased noticeably when the robot was walking to a developer's desk – the participants talked to each other more excitedly, were more alert and got up from their desks to check on the robot's location.

During the second evaluation, the results were more polar. The lamps were switched six times during the course of the second study, and only one of the participants noticed only one of these changes (and that was when he returned to the room and found the red lava lamp lit). The robot, on the other hand, was sent out twice, and both times, eight out of the ten respondents noticed it. The respondents who did not notice the robot were out of the room when it ran.

During both days, the developers nearest to BuildBot noticed it when it was near them, even if it was on the other side of a cubicle wall. On the first day, it took participants an average of 5.1 minutes to notice BuildBot. On the second day, it took participants an average of 3.3 minutes.

For some participants, especially those listening to music or talking to other people during the study, they did not notice the robot until it was near their workstation. Some participants were more distracted by the robot's movements because their desks face outwards, towards aiseways the robot used to arrive at different desks. Participants who did not find the robot's movements distracting had desks that faced the walls of the lab.

Figure 7 shows the data collected from the post-study questionnaire.

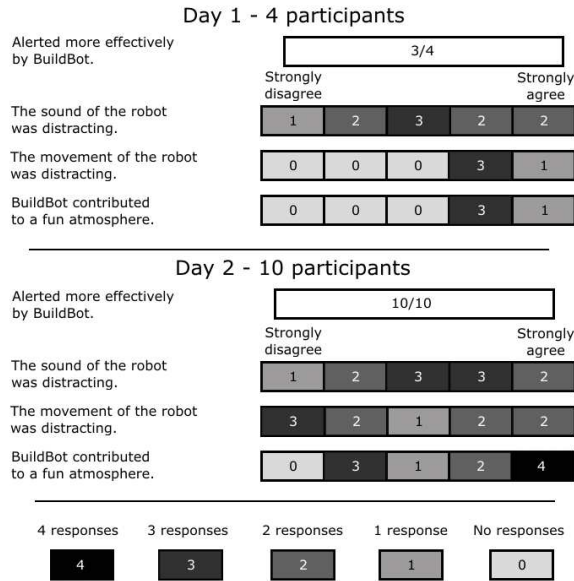


Fig.7 The post-questionnaire results from both studies.

Developers were easily alerted by the movement of BuildBot, but found it somewhat distracting when the robot was in motion. Despite this, most developers felt that the robot contributed to a fun atmosphere.

From these results, we conclude that BuildBot clearly was more effective at alerting developers when a change occurred. Although it was somewhat distracting, the fact that most developers felt that BuildBot contributed to a feeling of fun indicates that it supports the Agile methodology.

## VII. FUTURE WORK

Although an informal evaluation was done in our lab, the next step is to perform a formal evaluation of our approach with a group of developers in industry in order to determine if BuildBot is actually helping the team.

BuildBot should also be tested in a longer study to determine if it helps a team in the longer term. Future studies could also involve multiple BuildBots alerting different teams located in different rooms, different floors, or even different buildings or cities.

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.

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 (instead of the default walk, in which the dog walks on its elbows).

## VIII. CONCLUSION

We believe BuildBot has great potential to play a significant role in an agile software engineering environment. It

keeps the development team focused on the task at hand, that is, to efficiently develop software whose components work together seamlessly. We believe that BuildBot may give the team a sense of accomplishment and progress toward a shared goal. The robot has the potential to increase morale in this way and to keep the social environment more engaging and fun for the agile developers. BuildBot’s visual and audible presence provides a sense of project dynamics and progress to any developer collocated with it. The sight and sound of the dog approaching a team member’s workstation will provide that developer with a positive social incentive to fix the build – avoiding situations of potential embarrassment, resentment or discomfort that may result if the same developer was to be approached by the project manager.

We believe that a team practicing Agile Methods could benefit from a robot that could continuously access this virtual information and express it in a physical way, beyond simple ambient data. Since robots, perhaps similarly to human software developers, actively exist in both the physical and virtual realms, we believe that they are well-suited for various tasks in the agile software development sociable setting.

## IX. REFERENCES

- [1] A. Cockburn, *Agile Software Development: The Cooperative Game*, Agile Software Development Series, 2001, pp 70-80.
- [2] M. Weiser, J. S. Brown. "Designing Calm Technology", *PowerGrid-Journal*, v 1.01, <http://powergrid.electricity.com/> 1.01, July 1996.
- [3] K. Beck, *Extreme Programming Explained*, Addison Wesley, 2000.
- [4] L. Rising, N. 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, "eXtreme Feedback for Software Development", 2004. <http://www.developertesting.com/archives/month200404/20040401-eXtremeFeedbackForSoftwareDevelopment.html>.
- [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>.
- [8] Sony AIBO home page. <http://www.sony.net/Products/aibo/>.
- [9] M. Fowler, "Continuous Integration". May 2006. <http://www.martinfowler.com/articles/continuousIntegration.html>
- [10] D. Saff, M.D. Ernst. An Experimental Evaluation of Continuous Testing During Development, In *International Symposium on Software Testing and Analysis (ISSTA '04)*, p76-85, Boston, USA. July 11-14, 2004.
- [11] C.D. Kidd, Sociable Robots: The role of presence and task in human-robot interaction. MSc thesis, Massachusetts Institute of Technology, June 2003.
- [12] M. Xin., E. Sharlin Exploring Human-Robot Interaction Through Telepresence Board Games. In Proc. ICAT 2006, Lecture Notes in Computer Science, Volume 4282/2006, Springer (2006).
- [13] C. L. Breazeal. *Designing Sociable Robots (Intelligent Robotics and Autonomous Agents)*. The MIT Press (2002).

