FlyLoop: A Micro Framework for Rapid Development of Physiological Computing Systems

Evan M. Peck, Eleanor Easse, Nick Marshall, William Stratton, L. Felipe Perrone
Bucknell University, Department of Computer Science
Lewisburg, PA. USA
{evan.peck, ele008, nam021, wss012, perrone}@bucknell.edu

ABSTRACT

With the advent of wearable computing, cheap, commercial-grade sensors have broadened the accessibility to real-time physiological sensing. While there is considerable research that explores leveraging this information to drive intelligent interfaces, the construction of such systems have largely been limited to those with significant technical expertise. Even seasoned programmers are forced to tackle serious engineering challenges such as merging data from multiple sensors, applying signal processing algorithms, and modeling user state in real-time. These hurdles limit the accessibility and replicability of physiological computing systems, and more broadly intelligent interfaces. In this paper, we present FlyLoop - a small, lightweight Java framework that enables programmers to rapidly develop, and experiment with intelligent systems. By focusing on simplicity and modularity rather than device compatibility or software dependencies, we believe that FlyLoop can broaden the participation in next-generation user interfaces, and encourage systems that can be communicated and reproduced.

Author Keywords

software engineering; brain-computer interface;
physiological computing system; intelligent user interface

ACM Classification Keywords

H.5.2. User Interfaces: Prototyping

INTRODUCTION

Over the last decade, physiological sensors have slowly transitioned from the exclusive use of researchers and research labs to the commercial sector. For example, commercial brain sensors such as Neurosky, Emotiv, and Muse, have emerged as low-cost, widely-advertised inputs for biofeedback systems or brain-computer interfaces. Extending beyond the brain, the number of sensors on or around our bodies have increased over last decade, whether it through heart-rate monitors embedded in watches or accelerometers in smartphones. However, despite this advancement of technology into the public's hands, the application of physiological signals is largely limited to simple biofeedback or health applications.

In a recent post, Steven Fairclough, a leading researcher in physiological computing systems (PCS), wrote the following: "[research on adaptive systems] place enormous emphasis on the capacity of the machine to monitor and make accurate inferences about the psychological state of the user. As important as this is, it is only half of the story ... The process of linking the detection of user states to the adaptive repertoire of a machine remains the great unspoken and unexplored area of research in the area of physiological computing" [5].

In this paper, we claim that **the lack of usable, flexible programming tools is a significant hurdle to the development of systems that are driven by sensor input**. Given the difficulty of examining adaptive responses without state detection, building an architecture for mapping physiological responses to meaningful output becomes a prerequisite for exploring adaptive mechanisms. Yet when discussing the relationship between programmers and adaptive applications, Magnaudet and Chatty write, "The mappings between sources and actions are where the intrinsic programming complexity resides" [6]. As a direct consequence, building the basic functionality of an intelligent system has become a filter for advancing the expressiveness of adaptive applications.

It is not surprising then that the creation of adaptive systems is difficult. The fundamental architecture behind physiological computing systems, the biocybernetic loop, involves the input and integration of sensor information, a training or calibration period, filters to reduce noise and extract relevant features, a mapping of data to user state, and the real-time output of a state predictions [4]. Each of these phases relies on technical expertise and involves design decisions that could bind the developer to a rigid, single-purpose system.

As a result, developers and researchers often construct custom-built systems that serve the short term goals of their creators. More often than not, these systems turn out to be so inflexible that the creators themselves find it cumbersome to adapt them to different scenarios. This effort comes at a cost. Systems that make assumptions about incoming data sources (either in terms of velocity or modality), for example, make it

EICS'15, June 23 - June 26, 2015, Dusiburg, Germany.
 ISBN 978-1-4503-3646-8/15/06 $15.00
DOI: http://dx.doi.org/10.1145/2774225.2775071

difficult to add additional sources without significant refactoring efforts. Systems that follow a specific data pipeline can be inflexible due to the coupling between components, slowing the incremental design process.

The complexity of these systems has broader consequences as well. Because systems are intricate, the distribution, replication, and validation of physiological computing systems is difficult. In addition, inflexibility has a negative impact on developers' abilities to rapidly construct new prototypes. Taken together, these issues can slow progress in a field that is quickly advancing new sensor technology.

In the tradition of data-flow architectures, **we created a programing micro framework for building physiological computing systems - FlyLoop - that focuses on simplicity, flexibility, and reuse**. Because it is written in Java, it has no inherent hardware restrictions, software dependencies (beyond the Java language), or OS requirements. At the system level, FlyLoop makes no assumptions about the flow of data between sensors, signal filtering algorithms, feature extraction, and machine learning models. To aid developers, we include default behaviors that hide challenging aspects of building physiological computing systems - sensor fusion and the training/testing phase of model building and use. Finally, FlyLoop focuses on usability for the programmer during module creation, minimizing necessary template code and allowing developers to focus on the underlying algorithms.

We believe that a framework like this can:

1. Lower the existing technological barrier for building sensor-driven intelligent systems, making them accessible to a more broad population of developers.
2. Increase the clarity of existing systems to encourage replication and validation.
3. Reduce coupling within systems to encourage extensibility and innovation.

BACKGROUND: ACCESSIBLE PCS

In the past, it required significant technical acuity to build a functioning physiological computing system (PCS) or brain-computer interface (BCI). Aside from the challenge of translating systems to real-time environments, assigning meaning to physiological input required access to expensive sensors, expertise in data acquisition, signal processing, and data modeling [4]. More recently, innovations have reduced the need for expensive equipment or expertise at varying stages of development.

BITalino, for example, is a 'DIY biosignals' kit that increases accessibility at the acquisition and output levels by providing users with a board of built-in inputs designed for interfacing with electrocardiography (ECG), electromyograph (EMG), and electrodermal activity (EDA) [1]. It also includes an associated software platform, OpenSignals, for visualizing incoming data. Similarly, the OpenBCI Board focuses on acquisition accessibility, providing a cheap microcontroller that can also sample signals such as EEG, EMG, and ECG. An increasing number of signal processing applications have been built using SDKs on top of OpenBCI [2], allowing users to monitor signals in real time.

OpenViBE, an open-sourced software platform for BCIs, focuses on improving accessibility at the software level [10]. It uses a flexible, data-flow architecture that is specifically tuned for constructing Brain-Computer Interfaces. In addition, a GUI front-end allows users to build systems using direct-manipulation interactions. While these features impose requirements on underlying operating system and other software dependencies, OpenViBE's low barrier of use has successfully broadened the participation in building BCIs (with over 90 references in scholarly articles since 2014 alone). Finally, other physiological computing products more narrowly emphasize accessibility of a specific technology (OpenEEG [9]), domain (Libelium e-Health Sensor Platform [3]), or educational goal (SpikerBox [7]).

FlyLoop differentiates itself from existing solutions by maintaining its identity as a lightweight, minimal programming framework that focuses on **usability for the developer**. It attempts to hide issues that are difficult to implement in physiological computing systems, but leaves the choice of input sensors (which may or may not be from physiological sources), machine-learning libraries, and output dependencies entirely in the hands of the developer.

FLYLOOP CORE MODULES

FlyLoop is a Java micro framework that consists of four fundamental pieces that we identify as critical components of any physiological computing system:

- **Data sources:** sources of streaming data input
- **Filters:** manipulates or processes the data
- **Learners:** mappings from data to user state
- **Outputs:** outputs data from the system

In the following sections, we discuss the design behind each of these categories of components, beginning with the decision to allow them to function interchangeably.

Pusher-Receiver Design

The FlyLoop framework was constructed to employ a modular and flexible flow of data through the system. FlyLoop follows the roadmap of other data flow systems by implementing a communication interface that can be used between all modules - the *Pusher-Receiver Design*.

All modules involved in the system's data pipeline inherit from a `Receiver` class or implement a `Pusher` interface (using a Push design pattern). In either case, information is transferred either into or out of a module one data point at a time - the speed of which is controlled by a system-wide polling rate (discussed further in *Sensor Fusion*). To prevent limiting data to any particular format (int, float, string), data is passed as the generic class `Object` in Java. Pushers can output data to one-to-many Receivers. Similarly, Receivers can input data from one-to-many Pushers. Enforcing a consistent input/output definition enables modules to be used at any juncture of the data pipeline, and potentially with varying input sources.

Design flexibility is critical for advancing the state-of-the-art in physiological computing systems. Different physiological sensors may sample at different frequencies, and data

pipelines can vary widely. Constructing a communication protocol makes it easier to treat the data flow of the application as a fluid system that can be manipulated and experimented on. In the following sections, we explain how each core module of FlyLoop takes advantage of the Pusher-Receiver design.

DataSource

The `DataSource` component interfaces with any kind of streaming input. So far, we have motivated this work largely from the perspective of physiological sensors. However, systems that restrict input to specified hardware or even a particular modality are constraining. Intelligent systems operate best when they leverage all valuable information sources of information.

Figure 1. DataSources can represent any form of streaming input, outputting the raw data to one-to-many Receivers

As a result, DataSource makes no assumptions about incoming data and contains two core operations that must be completed the by a developer - `startCollection` and `getOutput`. Given the Pusher-Receiver design, getOutput returns a single datapoint from the sensor of data source. This allows disparate inputs to be treated in a similar manner, whether they are from EEG or mouse movements. In the *System Design* section, we discuss how data sources of various sampling rates work in concert in the FlyLoop Framework.

Filter

A `Filter` takes incoming data and modifies it. In sensor-based systems, data is often modified in a number of ways in order to transform it from raw data to meaningful information. This can take the form of noise-reducing filters, aggregating input from multiple sensors, or extracting high-level features for input to a model.

Figure 2. The Filter module modifies data in some way, receiving input from one-to-many Pushers, and outputting data to one-to-many Receivers.

While identifying best-practice signal processing techniques remains an open question for physiological signals, researchers continue to spend considerable effort experimenting with various combinations of algorithms. A design goal of FlyLoop is to **allow developers to rapidly swap in and out new filtering algorithms, or quickly define new sets of features to serve as input to a model**. As a result, any Filter can act as the receiver of any number of other modules, and act as the pusher to any number of modules. In Fig. 5, we show two example systems that demonstrate some of the flexibility that this affords.

At first glance, it can appear that Filters may be difficult to implement given that the framework pushes a single data point at a time through the system. However, in keeping with our design goals, FlyLoop focuses on **usability at module creation** as well. We provide support functions - `getDatapoints` - that allows the programmer to ask for a window of data of any size, for example. Rather than considering data one-point-at-a-time, the system will wait for the necessary interval of data and return it to the Filter.

These design decisions allow developers to focus their attention on the signal processing algorithms themselves, instead of spending considerable effort on timing issues. Below, we show an example of a Filter that computes the moving average of an incoming signal. Note that the input and output of data is largely hidden from the developer, and that the logic of the module is almost completely devoted to the algorithm.

```
/** Moving avg. low-pass filter with given interval size*/
public class AvgFilter extends Filter {
  public AvgFilter(int interval) {
    super(interval);
  }

  public Object filterData() {
    // Returns the last interval data points
    Queue<Double> dataPoints = getDataPoints(interval);
    // Compute the average
    double total = 0;
    for (Double d : dataPoints) {
      total += d;
    }
    return new Double(total/dataPoints.size());
  }
}
```

Listing 1. A simple example of building a Filter module

Learner

The `Learner` class, which accepts input from any Filter or DataSource, has two primary responsibilities: 1) to construct a model based on incoming training data, and 2) to output real-time classifications based on new data. Working in concert with a `Calibrator` (which we describe in detail in *System Design*), the Learner defines a mapping from incoming data to meaningful output. While this is often done using statistical techniques or known machine-learning algorithms (for example, we provide a Learner that interfaces with Weka, a widely used machine-learning library), the exact definition is to be defined by the developer, and Learners fundamentally consist of a single `learn` method. In addition, we provide mechanisms to accept the output or input of pre-existing models, allowing the system to immediately begin its real-time classification phase.

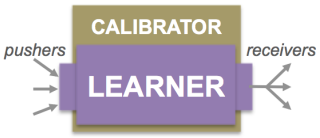

Figure 3. The Learner module, which interacts with a Calibrator, builds a model and classifies data in real time. It can take input from one-to-many Pushers and output predictions to one to many Receivers

Like all Receivers, Learners have access to the `getDataPoints` method that simplifies the collection of data from incoming modules, allowing developers to quickly define their own feature sets.

Output

Data can be pushed from any module to an `Output` at any juncture of the data pipeline. Because the module treats input from any Pusher in a similar manner, Output can produce a mixture of DataSources, Learners, or Filters. For example, it can be used as an interface to send model classifications over the network, or to construct logs that preserve the history of the system. The generic design affords flexibility within the system as well. Developers can build Output modules that interface with existing data visualization tools and monitor the signal in real-time at various states of the pipeline.

Figure 4. The output module can take input from one-to-many Pushers

SYSTEM DESIGN: HIDING COMPLEXITY

Simple Training: Calibrator

In many PCS, data flows through the system in two distinct stages: *training (or calibration)* and *testing*. In the training phase, signals are recorded during known user states to create training examples for a model. For example, a users may be asked to perform complicated arithmetic to capture responses during high workload. Then in the testing phase, after the model is constructed, predictions of user state are generated in real time from new sensor data. While the flow of data between these two stages is similar, differentiating them within the system can be a challenge. FlyLoop hides the engineering behind this two-step process from the programmer by using a `Calibrator` module.

The Calibrator module uses a small set of built-in functions to communicate the labels of training data to a Learner, as well as whether it should be calibrating or making predictions in real-time. Each time a calibration task changes the desired user state, that information is immediately communicated to the Learner to associate the true label with incoming data.

We provide this minimal specification so that a developer can use a diverse set of training tasks in the system without significantly refactoring the code. We hope that this will encourage developers to build libraries of potential training tasks that could be used by a variety of systems and users. Below, we provide a simple example of a Calibrator that is designed to differentiate between low workload and high workload.

```
/** A workload calibrator */
public class WLTrainer extends Calibrator {
  // The user states we are trying to predict
  private String[] STATES = {"low", "high"};
  private NUM_TRIALS = 5;

  public WLTrainer extends Calibrator(Learner learner) {
    states(STATES, learner);
  }

  /** A naive calibration task which cycles between
  training trials of low workload and high workload */
  public void calibration() {
    setCalibrating(true);
    for (int i = 0; i < NUM_TRIALS; i++) {
      this.setCurrentState("low");
      // Insert call to low workload task here
      this.setCurrentState("high");
      // Insert call to high workload task here
    }
    setCalibrating(false);
  }
}
```

Listing 2. A simple example that demonstrates the structure of the Calibrator module

Defining Data Flow

After modules are built, the flow of data between them must be defined. Each module may accept one-to-many sources of data from one-to-many other modules. We share a simple example to demonstrate the design of the Receiver class makes these relationships straightforward:

```
// Sensor sends data through filter
brainSensor.setReceivers(mvgAvgFilter);
// Filter sends data to model
mvgAvgFilter.setReceivers(SVMlearner);
// Model sends data to the console
learner.setReceivers(consoleOutput);
```

Listing 3. Defining the data pipeline in FlyLoop

While the method appears simplistic, identifying the flow of data through a system is often a nontrivial task in physiological computing systems. Constraining the major design decisions of the system to simple statements increases the readability of the code and clarifies the biocybernetic loop to other users. Since each module is capable of receiving information from one-to-many modules, readability is not compromised even as the system scales to a more complex example, as shown in Figure 5.

Sensor Fusion

Combining data from multiple input sources is an additional challenge for building PCS. Sensors sample at different frequencies, and those frequencies can vary in their reliability. In addition, data may be integrated at various points in a pipeline. For example, fusion may occur directly after receiving data from the sensors or just before the model, in a Learner. As a result, designing for any single point of fusion limits the flexibility of the system.

We attempt to hide this complexity from the programmer by using a system-wide polling rate that is set to match the highest frequency of data input. At this frequency, FlyLoop triggers the push function of each module, transferring one data point to the next module in the data stream. The system provides two default options for sensors that sample at a slower rate: 1) data is repeated until a new point enters the stream, or 2) null values are pushed when there is no new data at a module. While these options can be overridden by the developer, its default behavior allows developers hide the complexity of sensor fusion and focus on other aspects of the system.

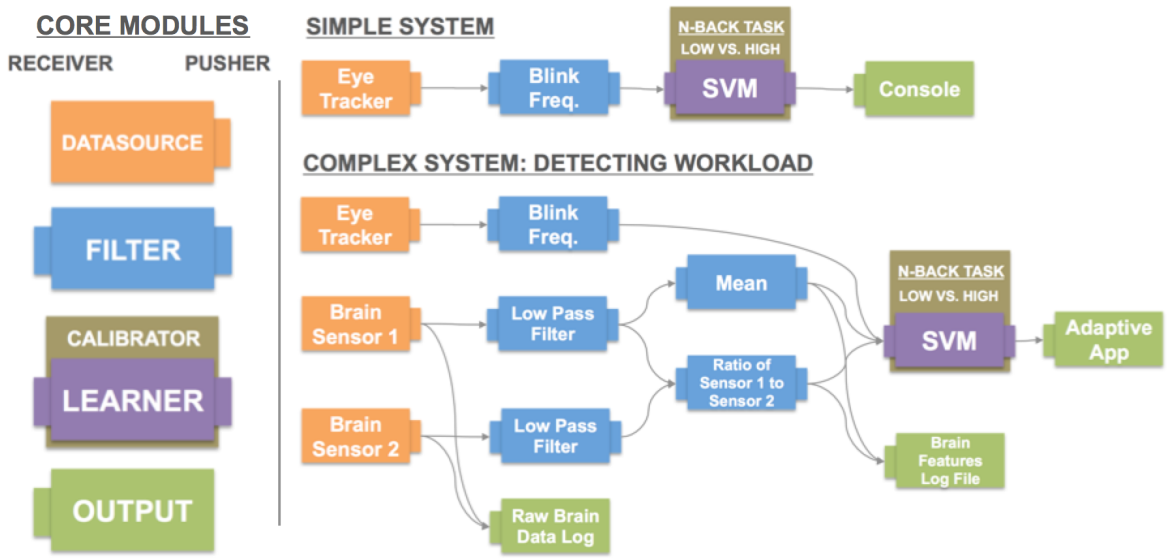

Figure 5. An example of two potential data pipelines in FlyLoop. The first simply uses blink frequency to predict workload. The second uses a combination of blink frequency as well as high-level features from two brain sensors. Note that output can be inserted at any point in the pipeline and that filters can receive input from one or more sources. The framework is designed to be as flexible as possible to the developers

EXAMPLE APPLICATION: ADAPTING TO WORKLOAD

To give a clearer understanding of FlyLoop's code structure, we walk through the 'complex system' example (Fig. 5) - an application that responds to user workload. For the sake of brevity, we choose to highlight critical portions of the code.

First, we define each of the modules we want to use:

```
// Define the states we want to detect
String [] states = new String [] {"Low", "High"};
// Sensor initialization - each inherits from DataSource
EyeTracker eyeSensor = new EyeTracker();
BrainSensor brainSensor1 = new BrainSensor();
// Filter initialization - each inherits from filter
MovingAvgFilter mvgAvg = new MovingAverageFilter(200);
LowPassFilter lowPass1 = new LowPassFilter();
// Initialize an SVM using the states we want to detect
Learner svmLearner = new WekaML(states, new SMO());
// Initialize Outputs - each inherits from Output
DataLog rawData = new DataLog('rawData.csv');
DataLog featuresFile = new DataLog('brainFeatures.csv');
MyApp myApp = new MyApp();
```

Next, we define the flow of data in the system. We focus on the data pipeline specifically from the brain sensors:

```
// Define the flow of data from the sensors
brainSensor1.setReceivers(new Receivers[] {lowPass1,
     rawData});
brainSensor2.setReceivers(new Receivers[]{lowPass2,
     rawData});
// Define the flow of data from the filters
lowPass1.setReceivers(new Receivers[]{ratioFilter,
     meanFilter});
lowPass2.setReceivers(ratioFilter);
// Define the flow of data from the feature generators
meanFilter.setReceivers(new Receivers[]{featuresFile,
     svmLearner});
ratioFilter.setReceivers(new Receivers[]{featureFile,
     svmLearner});
// Define where SVM classifications are sent
svmLearner.setReceivers(myApp);
```

In this particular application, we decide that the n-back task (a well-validated task from the psychology literature) is a suitable training task for high and low workload. So finally, we define the Calibrator and start the system:

```
// Create the training task
NBackTraining nbackTask = new NBackTraining(states,
     svmLearner);
// Define a system-wide polling rate
final static int POLLING_RATE = 200;
// Start the system
FlyLoop loop = new FlyLoop(nbackTask, sensors,
     POLLING_RATE);
loop.go();
```

Using this code, we can build a system that predicts workload in real time to pass onto an adaptive application.

DISCUSSION

In this paper, we discussed the design of the FlyLoop framework, emphasizing flexibility and ease-of-use. In this section, we will expand on the implications of its design.

Designing for Replication, Accessibility

Despite the emphasis on reproducibility in the HCI community, replicating PCS experiments that rely heavily on complex systems is challenging. With this in mind, we designed FlyLoop to maximize system transparency and code clarity.

FlyLoop's **modular design promotes reuse**, as the coupling between modules in the framework is severely constrained by the Pusher-Receiver model. Thus it enables interoperability between classes/modules from different developers. In addition, because modules can be inserted midstream to an existing framework, it **encourages extensions of existing physiological computing systems**, minimizing the overhead placed on developers to build their own systems.

Second, the data flow definition promotes **system transparency** by clearly defining each transition. Researchers reviewing a piece of code no longer need to sift through multiple objects or parse complicated looping schemes to obtain a high-level view of the system. This follows closely to Ben Shneiderman's well known information-seeking mantra of 'overview, zoom and filter, details on demand' [11].

Increasing Accessibility Beyond the Developer

While the FlyLoop framework simplifies front-facing code, code of any kind can be intimidating to those without experience in a language. However, as physiological sensors become more ubiquitous, constraining participation in any capacity is undesirable. FlyLoop's modular design opens doors for future extensions to broaden accessibility even beyond the developer.

Currently, we are developing **a simple, easy-to-understand language** that sits on top of FlyLoop in the form of configuration files. This would allow users to select components, set parameters, and define data-flows without any interaction with code. Configuration files could be compiled to Java code that conforms to the definitions in our framework.

Moving forward, systems that leverage data-flows in other domains have successfully used visual programming languages to increase participation. For example, PureData, a tool often used in the music community, translates its modular design into a visual interface that looks similar to the conceptual design we presented in Fig. 5. It has been suggested that this visual language has increased the public outreach in speech processing, among other sound-based analysis [8]. Given the modular construction of FlyLoop, a similar visual language could be designed on top of the existing framework.

LIMITATIONS

While the FlyLoop framework succeeds in hiding complexity from the programmer, its Pusher-Receiver design also has limitations. Since the same modules are used in both training and classification pipeline, Filters must use algorithms that can operate in online environments. While this design constraint is not dissimilar to one that many researchers face when constructing physiological computing systems, it may make manipulations that require a global view of the data challenging.

The online approach also limits the effectiveness of algorithms whose speed are dependent on matrix operations (which are common in languages such as MATLAB). While we did not encounter any slowdowns in informal tests using a commercial EEG device and eye-tracker, our design could impact systems that have a high volume and velocity of data.

Finally, we focused on the foundations of the framework itself instead of a rich feature set. As a result, the out-of-box capabilities of FlyLoop may initially feel underwhelming. However, it is our hope that the design will encourage Output modules that interface with other open data-analysis frameworks (such as OpenViBE), or Learner modules that interact with popular machine-learning libraries (we currently provide an interface with Weka). We believe that FlyLoop thrives as tool for rapid prototyping and experimentation.

CONCLUSION

In this paper, we presented the FlyLoop framework - a simple, lightweight Java framework for building systems that intelligently react to physiological input. Rather than developing front-end solutions for building physiological computing systems, FlyLoop acts as a minimalistic framework that emphasizing system flexibility and usability for developers. Maximizing code clarity at both the system and module level, we believe that systems built with FlyLoop will be easily replicable, and its modular design will encourage experimentation and innovation of adaptive mechanisms. In addition, the fundamental pieces of the framework lend itself to a wider range of inputs and outputs, functioning with any streaming input. While the field of physiological computing systems is still relatively young, we hope that with accessible frameworks such as FlyLoop, we can broaden the demographic of developers inventing the next generation of user interfaces.

REFERENCES

1. A. Alves, H. Silva, A. Lourenço, and A. Fred. 2013. BITalino: A Biosignal Acquisition System based on the Arduino. *Proc. of BIOSIGNALS 2013* (2013), 261–264.
2. P.J. Durka, R. Kuś, J. ygierewicz, M. Michalska, P. Milanowski, M. abcki, T. Spustek, D. Laszuk, A. Duszyk, and M. Kruszyski. 2012. User-centered design of brain-computer interfaces: OpenBCI.pl and BCI Appliance. *Bulletin of the Polish Academy of Sciences: Technical Sciences* 60, 3 (2012), 427–431. DOI: http://dx.doi.org/10.2478/v10175-012-0054-1
3. Libelium e Health Sensor Platform. 2015. (2015). http://www.libelium.com/130220224710/
4. S.H. Fairclough. 2009. Fundamentals of Physiological Computing. *Interacting with Computers* 21 (2009), 133–145. DOI: http://dx.doi.org/10.1016/j.intcom.2008.10.011
5. S.H. Fairclough. 2015. We Need To Talk About Clippy. (2015). http://physiologicalcomputing.org/2015/03/we-need-to-talk-about-clippy/
6. M. Magnaudet and S Chatty. 2014. What Should Adaptivity Mean to Interactive Software Programmers?. In *Proc. of EICS 2014*. 13–22. DOI: http://dx.doi.org/10.1145/2607023.2607028
7. T.C. Marzullo and G.J. Gage. 2012. The SpikerBox: A low cost, open-source bioamplifier for increasing public participation in neuroscience inquiry. *PLoS ONE* 7, 3 (2012). DOI: http://dx.doi.org/10.1371/journal.pone.0030837
8. R K Moore. 2014. On the use of the Pure Data programming language for teaching and public outreach in speech processing. In *Proc. of Interspeech 2014*. 1498–1499.
9. OpenEEG. 2015. (2015). http://openeeg.sourceforge.net/
10. Y. Renard, F. Lotte, G. Gibert, M. Congedo, E. Maby, V. Delannoy, O. Bertrand, and A. Lécuyer. 2010. OpenViBE: An Open-Source Software Platform to Design, Test, and Use BrainComputer Interfaces in Real and Virtual Environments. *Presence: Teleoperators and Virtual Environments* 19, 1 (2010), 35–53. DOI: http://dx.doi.org/10.1162/pres.19.1.35
11. B Schneiderman. 1996. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proc. of IEEE VIsual Languages 1996*. 336–343.