# SE-155
# DBSA - A Device-Based Software Architecture for Data Mining

Janne Kätevä, Perttu Laurinen, Taneli Rautio, Jaakko Suutala, Lauri Tuovinen, Juha Röning
Intelligent Systems Group
Department of Electrical and Information Engineering
FI-90014 University of Oulu, Finland
firstname.surname@ee.oulu.fi

## ABSTRACT

In this paper a new architecture for a variety of data mining tasks is introduced. The Device-Based Software Architecture (DBSA) is a highly portable and generic data mining software framework where processing tasks are modeled as components linked together to form a data mining application. The name of the architecture comes from the analogy that each processing task in the framework can be thought of as a device. The framework handles all the devices in the same manner, regardless of whether they have a counterpart in the real world or whether they are just logical devices inside the framework. The DBSA offers many reusable devices, ready to be included in applications, and the application programmer can easily code new devices for the architecture. The framework is bundled with connections to several widely used external tools and languages, making prototyping new applications easy and fast. In the paper we compare DBSA to existing data mining frameworks, review its design and present a case study application implemented with the framework. The paper shows that the DBSA can act as a base for diverse data mining applications.

## Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

## Keywords

software frameworks, data mining

## 1. INTRODUCTION

It is widely accepted that software frameworks can significantly increase software quality and help reduce development efforts [6]. This is also the case with data mining software. However, frameworks are not widely used in the community; the focus of data mining research has been on algorithm and class library development, although recently the number of data mining frameworks has increased [14].

It is resource consuming to implement new software products from scratch. Software architectures help developers build increasingly complex and large systems [3]. Development is also sped up if there is a good framework available that implements well-tested and validated functionality common for the application domain. In the case of data mining, such functionality could be, for example, database connectivity or support for real-time data streams from sensors. Frameworks enhance productivity and guide application developers into using well-defined patterns and solutions [3]. This is especially important if the focus of the application developer has previously been mainly on algorithms.

The purpose of this paper is to present a general-purpose, component-based data mining software framework based on an analogy where processing tasks are thought of as devices. Each device has its own specific functionality, and the devices are designed to work independently with a prespecified input and output. A data mining application is built using the framework by defining the employed devices and their parameters along the connections between the device outputs and inputs with a control script written in an XML document. The devices are further arranged into layers based on the abstraction level of their functionality. The framework is called the Device-Based Software Architecture (DBSA). It is written in C++ and combines features from the pipes-and-filters and layered architectural styles, which are commonly seen in data mining software [9, 11, 2].

DBSA can be used in a variety of data mining tasks. One of the fundamental motives for developing it has been to make a hardware-independent and highly flexible framework for data stream mining. Another aspect taken into account from the beginning has been the portability of the framework to as wide a range of platforms as possible, be they mobile, desktop or server platforms. DBSA offers connections to external tools and a knowledge base, but these aspects are not fully in the scope of this paper. The concept of the database structure DBSA uses and the interface for external tools are introduced, but not discussed in detail.

There are currently some versatile data mining frameworks available [9, 11, 2]. Unfortunately, most of these are either commercial or restricted in platform support. Java is frequently used for framework implementation, in which case a Java interpreter is needed for each platform where the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*SAC'10* March 22-26, 2010, Sierre, Switzerland.
Copyright 2010 ACM 978-1-60558-638-0/10/03 ...$10.00.

framework is to be used. This excludes at least some mobile platforms such as Maemo, for which there is no Java Virtual Machine available. Compared to the frameworks cited above, DBSA has new functionalities and is designed to be an open-source framework with extensive platform support.

This paper is organized as follows: Section 2 describes the motivation for creating a new data mining framework. It also gives a short introduction to some existing data mining frameworks and libraries and discusses how DBSA relates to them. The design of the framework is reviewed in Section 3. Section 4 presents a case study, a gesture-controlled Pacman game with DBSA as the underlying framework. Section 5 concludes the paper.

## 2. MOTIVATION AND RELATED WORK

The development of the DBSA framework began with the need for an application where human gestures are monitored and recognized from a data stream collected with various sensors. With many different computers, operating systems, sensors and data storages to manage, our research team has struggled with applications that do not work together well, or work only with specific systems. There has been a pressing need for an architecture that can easily unify all of these hardware and software components within a single comprehensive framework.

In [1] the rationale for creating a multi-platform unified software architecture is justified: When an organization is producing multiple similar software products and reusing the same architecture in them, a software product line can be used to reduce implementation costs and time to market. A software product line is a set of software systems sharing a common architecture and a set of reused components, which together form a software platform to satisfy the specific needs of a particular domain.

Besides DBSA, some data mining frameworks already exist. Three of those are introduced here: Smart Archive, RapidMiner and KNIME. These are compared to DBSA because of their resemblance to it and because they are (or are intended to be in the future) freely available to download and use. In addition to these three we bring up Weka, a library of data mining algorithms with a software suite built around it that gives it some framework-like qualities.

**Smart Archive** (SA) is a domain-specific but application-independent framework for data mining, developed at the University of Oulu. DBSA and SA are both derived from the same reference architecture introduced by Laurinen et al. [9]. Like DBSA, SA models applications as independent components connected by data flows, using pipes-and-filters architectural patterns to implement the data flows. SA is implemented in Java, but there is a C++ variant called Mobile Smart Archive (MSA) [12], which focuses on mobile Linux platforms. SA can be seen as a predecessor of DBSA; SA has not been released to the general public, but it has been used to implement industrial applications. [14]

**RapidMiner**, formerly YALE (Yet Another Learning Environment), is a widely-used [8] free open-source environment for KDD and machine learning developed at the University of Dortmund. It provides a variety of methods that enable the fast prototyping of new applications. RapidMiner also provides functionality for process evaluation and optimization. A visual programming paradigm is used to ease the design of new schemes. The visual representation is stored internally as XML to enable automated applications after the prototyping phase. RapidMiner is implemented in Java. [10, 11]

**KNIME** (Konstanz Information Miner) was developed at the University of Konstanz. It is a modular data exploration platform where the user can visually create data flows, selectively execute some or all analysis steps, and later investigate the results through interactive views. KNIME integrates analysis modules of the Weka data mining library, and additional plugins allow R scripts to be run. KNIME is based on the Eclipse platform, [5] and it is easily extensible through a modular API. KNIME is implemented in Java and it is dual-licensed, so there is a non-profit, open-source license available. [2]

**Weka** (Waikato Environment for Knowledge Analysis) is a popular suite of machine learning software. It was developed at the University of Waikato. It is widely used in other data mining software solutions, including RapidMiner and KNIME. Weka contains a collection of visualization tools and algorithms for data analysis and predictive modeling. This functionality can be accessed via graphical user interfaces. Weka supports several standard data mining tasks. It is not capable of multi-relational data mining, but there is a separate piece of software for converting a collection of linked database tables into a single table that is suitable for processing. Weka is free software and written in Java. [15]

Compared to the other frameworks, the component-level abstraction of DBSA, in which everything is a device, makes it more generic in the sense that the application developer handles every component in the same way. From the point of view of the framework, real-world devices (e.g. inertial sensors, cameras) are the same as logical devices (e.g. database tables, classification algorithms). DBSA can be a base for many different data mining applications, but its main focus is on data-stream mining from multiple sources.

On the other hand, DBSA also concentrates on the efficiency of data processing. As a result, the format of the data piped through the architecture is application-dependent and must be defined by the application programmer. However, the architecture comes ready-fitted with several reusable data types, and the application programmer is free to build a generic data type if one is needed.

The main requirements of DBSA are shared by most software frameworks. The architecture should be easy to use, platform-independent and efficient. It also should be flexible: adding new features and support for new data sources and sinks should be straightforward. The possibility of using external tools, such as Matlab, provides application developers with an easy way to rapidly prototype systems. The algorithms of an application can be first implemented using external tools, and then, in a later development phase, given final implementations in the target language of the application.

The typical users of DBSA could be academic researchers with varying backgrounds in software development. Most likely their focus will have been on algorithm development, not software design. External tools such as Matlab and R are likely to be more familiar to them than the target programming language. In such situations DBSA offers an easy way to develop algorithms while at the same time contributing to the final application. Each algorithm developed using an external tool instantly becomes part of a working prototype that can be turned into the final application by substituting target-language implementations of the same algorithms for

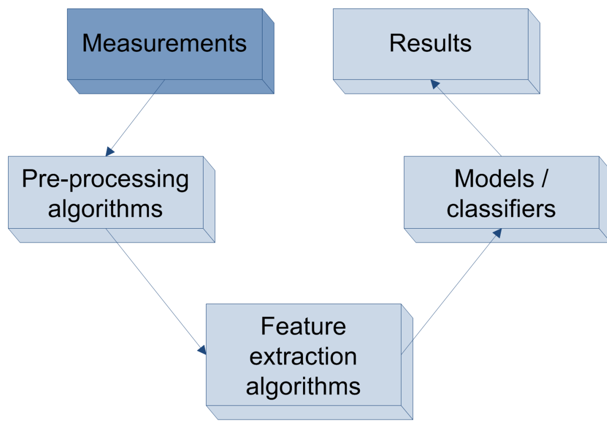


**Figure 1: The reference architecture**

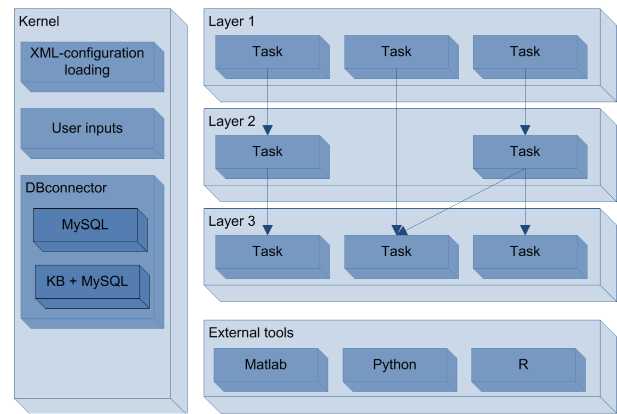


**Figure 2: The DBSA system architecture**

the external tools used in the prototyping phase.

DBSA has been already used in various data mining tasks, including machine vision and gesture recognition applications. In the future, it will also be used in other projects. One machine vision example can be seen in the Section 4. The gesture recognition application uses accelerometer sensor data to recognize user gestures that can be used for example to control a graphical user interface without traditional input devices.

Future plans include model selection applications where one external tool is used to create the model and another tool for testing it. Another application will use DBSA to distribute processing to multiple computers. This is especially useful in mobile platforms where the heavy calculations can be made in a remote server. This is possible by using network devices, which will take care of the communication between the DBSA instances in mobile and server platforms.

# 3. DESIGN

This section describes the architectural style and system components of the DBSA framework. It covers the major design decisions of DBSA, the rationale behind them and the way they affect the functional and quality properties of DBSA and distinguish it from other data mining frameworks. For an application developer it is not necessary to have such extensive knowledge of the inner mechanics of DBSA; basic framework functionality and device and configuration creation can be understood without it.

## 3.1 Description of the Framework

A reference architecture is an abstract model of the interoperability of the components common to the applications belonging to a given architectural family. The reference architecture common to data mining applications is shown in Figure 1 [9]. This architecture can be found in data mining applications implemented with both SA and DBSA, even though the actual system architectures of these two frameworks differ considerably.

As stated before, the architecture organizes processing tasks based on an analogy where each task is represented by a device. A device may be a real-world device, such as a temperature sensor, or a logical device that implements e.g. a data mining algorithm and has no physical counterpart. In other words, a logical device is a device that exists only as a software component within the framework, performing a processing task. The devices are further arranged into layers based on the abstraction of their function: the tasks of devices in the same layer are somehow related or have the same level of abstraction. However, the use of these layers is not required, although it is advisable.

Alongside the layered architecture resides the kernel. Its components take care of constructing devices and device configurations during runtime. Kernel components also take care of device and database connection management. Figure 2 illustrates the system architecture.

As an example, let us consider a weather monitoring station with the tasks of measuring and monitoring temperature and wind conditions. There is a sensor that can measure the temperature and send the result to the server for monitoring purposes, and another sensor for wind speed. This arrangement gives us four logical tasks: temperature measuring, temperature monitoring, wind speed measuring and wind speed monitoring. Furthermore, we have two abstraction layers: a real-world sensor layer and a monitoring layer. Figure 3 depicts this configuration. Sensor layer devices are connected to their monitoring layer counterparts; the arrows in the diagram depict the data flow from one device to another. Devices and device layers are drawn as boxes.

A device is not, in fact, just a black box with no visible internal structure. Inside each device is a processor that transforms the input acquired from the previous layer of devices into the information or knowledge it provides for other devices. The processor, in turn, has at least one subcomponent, the recognizer, which is in charge of the actual processing of the received data. The processor may also contain a progress tracker to keep track of the current state of the work of the recognizer. Also, each device has buffers that are used to store and transfer the produced data or knowledge. The device components are illustrated in Figure 4.

The main subcomponent of the processor is called the recognizer because often its task is to recognize *actions* or *events* from the incoming data stream. An action is a low-level pattern in the data, whereas an event is understood as a higher-level pattern that is usually composed of actions recognized by lower-level devices. For example, raising a hand could be an action and lowering it another one. If these actions are recognized by a device, another device could recognize a greeting event by comparing the action

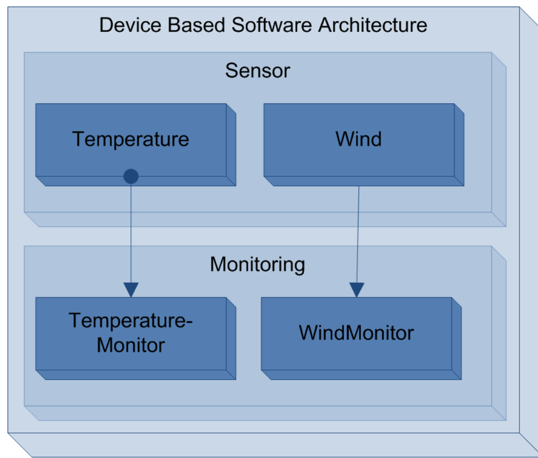


**Figure 3: Example configuration**

stream to a predefined sequence, the *reference event*. Similarly, there may be reference actions in the first device, compared against incoming data to find hand movements.

The subcomponents of the processor communicate with each other. The output of the recognizer is an estimate of the most recent action or event and can be given for ongoing or finished actions. The progress tracker follows the progress of the ongoing action identified by the recognizer, comparing it to the reference action. The progress of the action can also be used by the recognizer when recognizing the action. The output of the device is stored in a buffer. A device may have multiple buffers, each containing a different type of information or knowledge. When the data items reach the buffer, they are kept in a queue until another device reads them from the buffer.

The devices consist of the subcomponents described above because of the device analogy. Most data mining frameworks have a pipes-and-filters architecture where the filters are chiefly interfaces; they may in some cases have a complex structure, but most filters are simple components with very little state or no state at all [4]. In contrast, the devices in DBSA carry a large quantity of state information with them. The subcomponents guide the application developer to organize the solution in a more structured manner than simple filters do.

The layered architecture of DBSA can be clearly seen in Figures 2 and 3. With this architecture DBSA combines features adopted from the interpreter architecture. During execution the user can give commands to the architecture to control devices or fetch information from them using the interpreter. The architecture of device configurations is layered, although in some cases the configuration may be closer to pipes-and-filters architecture or a hybrid of the two styles.

The devices have to be created by the user before they can be used in configurations. There are two ways of adding new devices to the framework. One way is to add a device as a plugin, in which case it will be loaded during runtime if needed. The other option is to make the device a part of the architecture, in which case it will be compiled with the architecture. The components that are part of the architecture are called kernel devices.

Once the devices have been created, the device layers are defined using the created devices. Layers are used to group

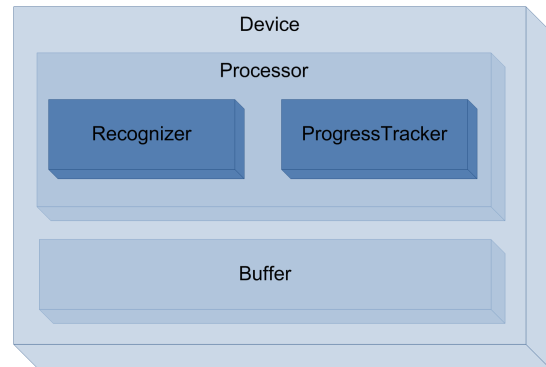


**Figure 4: The device components**

related devices in execution. The main function of layers is to organize the employed devices and offer easier managing of device groups for the kernel. During execution, each device is executed in separate threads to increase hardware resource usage and performance. Also, the thread managing is done in these layers. The layer is responsible for the starting, pausing and stopping of device threads. Devices and layers are the most visible components of the framework to the application user.

In addition to devices and layers, DBSA has several utility components that are not part of the core architecture of DBSA but are included in it to ease the work of application developers. For example, there are components for database connection management and data structures for storing various kinds of information. Such components are not necessarily visible to the user, nor should they be; from the perspective of usability it is better to hide the implementation details of these functionalities from the user.

## 3.2 Architectural Patterns Used

An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used. These constraints define a set of architectures that fulfills them. The architectural patterns implemented by a system affect the quality attributes of the system. [1]

The architectural patterns that can be seen in the DBSA framework are *pipes-and-filters architecture*, *layered architecture* and *interpreter architecture*. These patterns are used in the framework to help it fulfill its functional requirements and desired quality attributes. In the following paragraphs we discuss these architectural patterns and how DBSA implements them.

The data flow between devices in DBSA can be seen as a classic example of the asynchronous pipes-and-filters architecture. Each device has an output buffer where the device stores its output. When the device has processed its data, the next device in the device configuration handles it as its input. In effect this means that the output buffer of a device is also the input buffer of the next device. The main advantage is that the processing can be done in steps. Also, it makes the system highly configurable because filters are dependent only on the type of their input data, thus they can be combined in various different combinations.

The layering of the architecture of DBSA is based on the abstraction level of the devices. In other words, each layer consists of devices that are similar in terms of abstraction.

Because the layered pattern organizes computational tasks based on the level of abstraction, the computational relations are not usually taken into account. This may have a negative impact on performance. However, using parallel processes may increase the performance if it is done correctly. The approach where a thread of its own is assigned for each layer usually does not increase performance. Better results may be achieved by executing each device on separate threads [4]. A simple example of layered architecture was seen in Figure 3, and a more intricate example is given in Section 4. Devices often communicate primarily with devices residing in layers immediately next to their own layer, but the layer architecture is flexible: a device may be connected to another device in any layer, bypassing any intermediate layers.

In addition to data read or generated by devices, DBSA also accepts functional instructions as input. The framework has a command-interpreter component that reads the instructions and translates them into control signals to the framework. The interpreter provides predefined commands for controlling applications made with DBSA; for example, there are commands for starting, stopping and pausing an application.

The main reasons why these architectural patterns are used in DBSA are configurability, flexibility and performance. The combination of pipes-and-filters and layered architecture together provides excellent configurability and flexibility for the system. The performance is increased by effective multi-threading. The filters are excellent units for concurrency because the interface for the filters is very narrow, which boosts performance since there is a reduced number of synchronization points.

# 4. CASE STUDY: GESTURE-CONTROLLED PACMAN GAME

The following example provides a versatile demonstration of the capabilities of DBSA. Besides the framework, the example uses two external tools, Matlab and Python. The example is gesture-controlled Pacman game. This game was chosen for demonstration purposes because it is familiar to most people and can be used to introduce DBSA capabilities in an interesting manner. A camera feed is used as input to a simple machine vision algorithm and also as a visualization for the user, but the main visualization is the playable Pacman game. The goal of the game is to control the Pacman character by tilting an accelerometer up, down, right or left. The game is started by the machine vision algorithm when it recognizes a Pacman image in one of the video feed frames.

The example involves data mining from two data streams. One of these is the accelerometer data, which is collected and analyzed in real-time. The other data stream is the camera feed. The frames from the camera are analyzed based on the colors in the image. After interesting colors have been identified, features are calculated from these areas to find out if a Pacman image is in the frame. If a correctly shaped image is found, the game starts if it is not already running. The code of the Pacman game, written in Python, comes from an open-source project [13], but all the other components of the application were developed in-house.

Figure 5 shows a screenshot of the example. The actual Pacman game is in the right-hand window. The video feed window is illustrating a recognized Pacman shape for the user. The accelerometer data is visualized in the bottom left window. It views all 3 axes of the accelerometer data. The tilting of the sensor can be clearly seen in this window.

## 4.1 Case Configuration

The example application consists of three logical layers. These are the *sensor*, *tracking* and *visualization* layers. The sensor layer collects the input data of the application. The `Shake` device is used to collect data packets from a wireless accelerometer sensor. The other device in the layer, `VideoStream`, reads camera frames at a specified rate.

In the tracking layer there are also two devices. `MovementTracker` calculates the inclination angles of the sensor from the accelerometer data. If the sensor is tilted in some direction - left, right, up or down - this device generates a data item containing this tilting information. `VideoTracker` reads the frames from the `VideoStream` device and uses Matlab to analyze the image. The device gets the number of Pacman shapes found and their bounding boxes as feedback from Matlab. Information about the Pacmen found is placed in one of the device's two buffers. The other buffer contains the camera frames with the bounding boxes drawn on top of the original image.

The visualization layer is the only layer visible to the user and has three devices. The `Pacman` device executes the Pacman game written in Python and controls the flow of incoming information about sensor tilting and Pacman shapes found. The `SignalComparer` device is used to visualize the raw accelerometer data in 3 dimensions. Finally, the `Window` device visualizes the camera feed and tracking information for the user in its own window.

Figure 6 illustrates the configuration of the example application. The connections between devices are still fairly simple in this example; they do not give a full view of the possibilities of DBSA configurations. For instance, DBSA also allows device loops to be constructed, a feature not used in the example.

## 4.2 Implementation of the Example

In addition to the DBSA devices described above, the example also involves two physical devices. Shake SK6 [7] is a versatile wireless sensor box that contains, among other things, a 3-dimensional accelerometer sensor. This device is used as the real-world counterpart of the Shake device in the architecture. A regular webcam is used to produce the camera feed.

The construction of this example began by first planning the device configuration shown in Figure 6. The next step was to design a new data type to store information about the orientation of the accelerometer sensor. This data type was used to transfer information between the `MovementTracker` and `Pacman` devices. Since the Pacman character can move only in four directions (up, down, left and right), the three-axis acceleration information from the `Shake` device is interpreted in `MovementTracker` as movements in these directions. The data type for camera frames is slightly more difficult, but one implementation for it is built into the framework.

Once the required data types are available, the devices must be implemented. The `Pacman` device is the most interesting device in this configuration. The basic idea of this device is to read available data items from its input queues

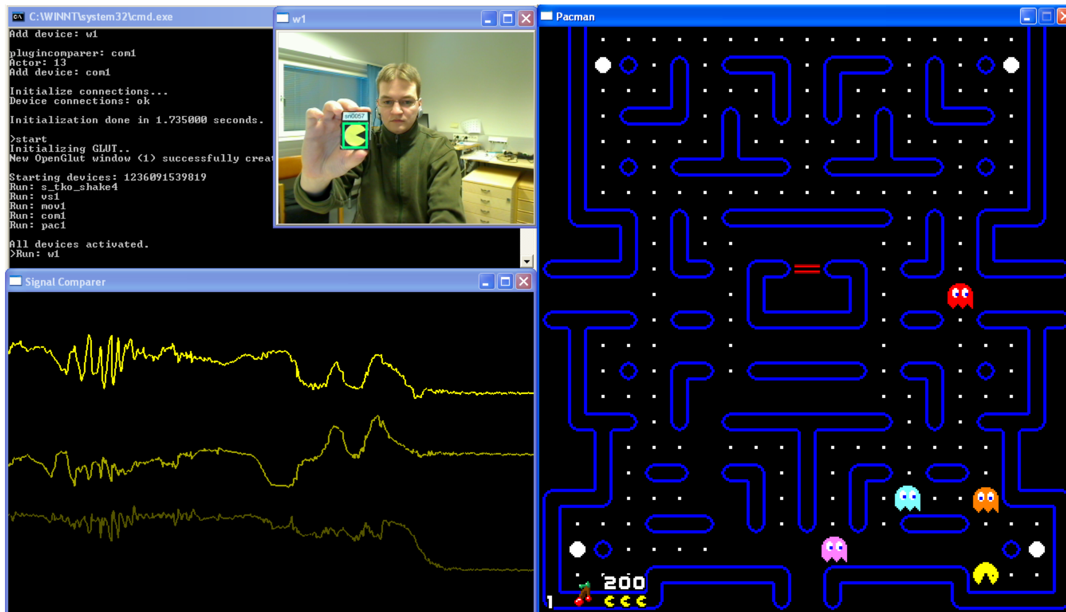


**Figure 5: Screenshot of the Pacman example.**

and then feed the received information to the Python interpreter, which is running the actual game. The data type for movement contains two values, one for the x axis and one for the y axis. The first thing to do is to check which of the axes is dominant and then find out whether the value is negative (corresponding to left or down) or positive (right or up). Based on these checks the `Pacman` device sends commands to the Pacman game running on Python to change the direction of the Pacman character. The Matlab and R interfaces can be used in applications in a similar manner.

When all the devices have been coded, they must be connected using the XML configuration file. The full XML file is not described here, but some of the most interesting points are discussed. The Pacman device is described in XML in the following listing. The description is simple because the device has only one buffer and it does not need any additional parameters. The *actor* tag tells the id of the person who is doing the data mining with the application. The *buffer_size* tag value 0 means that the buffer of this device is dynamic, so it reserves space only for new items that have not been read yet.

```
<device model="pacman" name="pac1">
        <actor>13</actor>
        <buffers>
                <buffer type="ALL">
                        <buffer_size>
                                0
                        </buffer_size>
                </buffer>
        </buffers>
</device>
```

The Pacman device consumes data queues from two devices: `MovementTracker` and `VideoTracker`. The following listing illustrates this configuration. **Pac1** is the `Pacman` device, which consumes movement data from the queue named "ALL", which comes from the `MovementTracker` device named **mov1**. The `Pacman` device also consumes data from the queue named "PAGGI", which originates from the `VideoTracker` device named **vt1**. This data is used to start the game when a Pacman image is found in the video stream.

```
<connections>
        <consumer name="pac1">
                <producer name="mov1">
                        <queue type="ALL"/>
                </producer>
                <producer name="vt1">
                        <queue type="PAGGI"/>
                </producer>
        </consumer>
</connections>
```

The Matlab code for image recognition was written for this demonstration application only and has not been tested in all conditions, making it sometimes error prone. As already stated, the Pacman game written in Python is an open-source project available on the Internet. However, all the other components such as `Shake` and the visualization devices were made as reusable as possible and have been thoroughly tested, so they can be reliably used in other data mining applications as well.

## 4.3 Evaluation of the Example Implementation

The implementation of the example illustrates various capabilities of DBSA in data mining. It does not use all of the functionality and configurability of DBSA, which limits its usefulness for extensive testing purposes. However, the application serves well as a proof of concept for what can be done, as well as for testing a relatively simple configuration and some of the interfaces to external tools.

Building the demonstration application was straightforward thanks to the "everything is a device" abstraction, which allowed each distinct task to be implemented independently from the others. The framework has built-in data types for Shake accelerometer values and camera frames, so the only new data type that had to be implemented was the one for Pacman movement (as stated earlier, the `Move-`

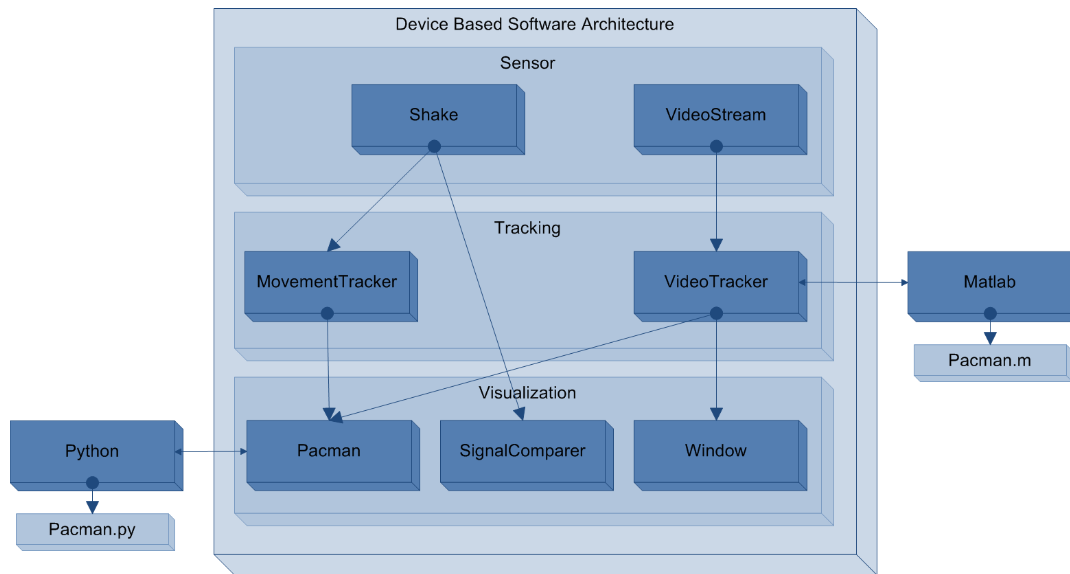


**Figure 6: Device configuration of the Pacman example.**

`mentTracker` device interprets acceleration values to control commands for Pacman to go up, down, left or right). The creation of the prototype was very rapid, since most of the devices used had already been implemented for other applications. The only DBSA device made specifically for this example was the Pacman device. The only other thing from the DBSA point of view was creating the application structure in XML.

The majority of programmer time was spent working with external code. The Pacman game had to be modified to accept control commands from the DBSA application instead of the keyboard, but more importantly, an image recognition algorithm had to be written in Matlab because there was no suitable algorithm available. Besides demonstrating the use of external applications with DBSA, implementing the algorithm can be viewed as an example of rapid prototyping. Matlab allowed different detection algorithms to be tried out quickly, and when working this way, the algorithm developer can be someone with no C++ programming experience. Of course, once a good algorithm has been found in Matlab, it is possible to implement it in C++ or C if it is necessary for the application to have no external connections (for example if the application is used in embedded or mobile systems).

Overall, the results from the example are encouraging. Tests with the application clearly show that DBSA is able to mine multiple data streams simultaneously and in real time. Without efficient multi-threading of the devices this might not be possible; even if it were possible, a single-threaded implementation that achieves comparable performance would be very difficult to accomplish. The main demonstration platform, an older laptop PC (MS Windows XP SP3, Pentium M 2.0Ghz, 1.5GB RAM, ATI MOBILITY RADEON X600), was entirely sufficient for the application to run smoothly.

A final point to note is that compared to the development effort required by the case study, building such applications without the support of an appropriate framework would be difficult and time consuming. The implementation of the example, including testing, took approximately 20 hours of developer time, not including the time spent on Matlab and Python code. All the devices in the configuration could be implemented separately, making unit testing much easier, since each device could be examined in isolation from all others.

## 5. CONCLUSIONS

This paper introduced the Device-Based Software Architecture, a multi-platform framework for creating data mining applications quickly and effortlessly. The architecture is so named because of the analogy that each processing task in the framework can be thought of as a device. To the architecture it does not matter whether the device has a real-world counterpart or whether it is, for example, just an algorithm performing a particular data mining task. Device configurations are stored in external XML files, which are used by the framework kernel during runtime to dynamically build device hierarchies. This allows developers to test new application configurations effortlessly by changing a few lines of XML code. The framework can be easily extended by adding new devices to it as plugins.

The DBSA framework was compared to some other freely distributed data mining application frameworks and its architecture was briefly discussed. Then an example application was created to demonstrate the versatility, efficiency and rapidness of the framework in data mining application development. Two real-world devices were used: an accelerometer and a webcam. The completed application was a Pacman game controlled by tilting an accelerometer and started by recognizing a Pacman shape from a video frame. The example application had connections to two external tools, Matlab and Python. External tool connectors can be used effectively when prototyping applications, allowing developers with limited software engineering expertise, such as researchers, to work with familiar tools.

## 6. ACKNOWLEDGMENTS

The authors would like to thank Infotech Oulu for support. Lauri Tuovinen would also like to thank GETA (The Graduate School in Electronics, Telecommunications and Automation) and Tauno Tönning Foundation for financial support.


## 7. REFERENCES


[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, December 1997.

[2] M. R. Berthold, N. Cebron, F. Dill, G. D. Fatta, T. R. Gabriel, F. Georg, T. Meinl, P. Ohl, C. Sieb, and B. Wiswedel. Knime: The Konstanz Information Miner. In *Proceedings of the 4th Annual Industrial Simulation Conference*, pages 58–61, 2006.

[3] F. Berzal, I. Blanco, J.-C. Cubero, and N. Marin. Component-based data mining frameworks. *Commun. ACM*, 45(12):97–100, 2002.

[4] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[5] Eclipse Foundation. Eclipse Project. http://www.eclipse.org.

[6] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, 1997.

[7] S. Hughes and S. O'Modhrain. SHAKE - Sensor Hardware Accessory for Kinesthetic Expression. In *Proceedings of 3rd International Conference on Enactive Interfaces*, pages 155–156, 2006.

[8] KDnuggets.com. Poll: Data mining software (2008). http://www.kdnuggets.com/polls/2008/data-mining-software-tools-used.htm.

[9] P. Laurinen, L. Tuovinen, and J. Röning. Smart Archive: A Component-based Data Mining Application Framework. In *Proceedings of the 5th International Conference on Intelligent Systems Design and Applications*, pages 20–26, Wroclaw, Poland, 2005. IEEE Computer Society Press.

[10] I. Mierswa, M. Wurst, R. Klingenberg, M. Scholz, and T. Euler. YALE: Rapid Prototyping for Complex Data Mining Tasks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 935–940, 2006.

[11] Rapid-I. YALE Becomes RapidMiner. http://rapid-i.com/content/view/64/74/lang,en/.

[12] T. Rautio, P. Laurinen, and J. Röning. Component-based framework for mobile data mining with support for real-time sensors. In *The Proceedings of International Conference on Agents and Artificial Intelligence*, pages 208–213, 2009.

[13] D. Reilly. Open source Pacman game made with Python. http://pinproject.com/pacman/pacman.htm.

[14] L. Tuovinen, P. Laurinen, I. Juutilainen, and J. Röning. Data Mining Applications for Diverse Industrial Application Domains with Smart Archive. In *Proceedings of the IASTED International Conference on Software Engineering (SE 2008)*, pages 56–61, 2008.

[15] Weka. Weka 3: Data mining software in java, accessed 14.1.2009. URL: http://www.cs.waikato.ac.nz/ml/weka/.