A Framework for Modular VST-based NIMEs Using EDA and Dependency Injection

Patrick Palsbröker
FH Bielefeld University of Applied Sciences
Minden, Germany
ppalsbroeker@fh-bielefeld.de

Christine Steinmeier
FH Bielefeld University of Applied Sciences
Minden, Germany
csteinmeier@fh-bielefeld.de

Dominic Becking
FH Bielefeld University of Applied Sciences
Minden, Germany
dbecking@fh-bielefeld.de

ABSTRACT

In order to facilitate access to playing music spontaneously, the prototype of an instrument which allows a more natural learning approach was developed as part of the research project Drum-Dance-Music-Machine. The result was a modular system consisting of several VST plug-ins, which on the one hand provides a drum interface to create sounds and tones and on the other hand generates or manipulates music through dance movement, in order to simplify the understanding of more abstract characteristics of music. This paper describes the development of a new software concept for the prototype, which since then has been further developed and evaluated several times. This will improve the maintainability and extensibility of the system and eliminate design weaknesses. To do so, the existing system first will be analyzed and requirements for a new framework, which is based on the concepts of event driven architecture and dependency injection, will be defined. The components are then transferred to the new system and their performance is assessed. The approach chosen in this case study and the lessons learned are intended to provide a viable solution for solving similar problems in the development of modular VST-based NIMEs.

Author Keywords

software engineering for NIMEs, dependency injection, VST, event-driven-architecture

CCS Concepts

•**Applied computing** → **Sound and music computing;** •**Software and its engineering** → **Publish-subscribe / event-based architectures;** *Abstraction, modeling and modularity;*

1. INTRODUCTION

In 2015, as part of the interdisciplinary research project "Drum-Dance-Music-Machine"(DDMM), a NIME was conceptualized and developed which aims to facilitate easier access to music-making for preschool children.[5] We assumed that with the help of an instrument with a simple interface, like a drum, but which produces a melodic sound, children could be provided with an easy and intuitive way to produce consonance. Central to the modular system we created to solve this is a dynamic composition, which can be influenced by two drum pads, a Kinect camera and/or a Wii Balanceboard (as an alternative to drums) as input media. The data from these components is then processed on a PC using different Virtual Studio Technology (VST) plug-ins (see figure 1). Since then, the prototype has been further developed and evaluated several times.[26, 27] In this process some design weaknesses within the structure of the VST plug-ins were revealed, for example the communication between plug-ins, which is only possible to a limited extent. Furthermore, the planned enhancement of DDMM is very complicated due to the difficulty of debugging and measuring the runtime of each individual plug-in. We assume that other developers may encounter similar problems when creating modular VST-based NIMEs.

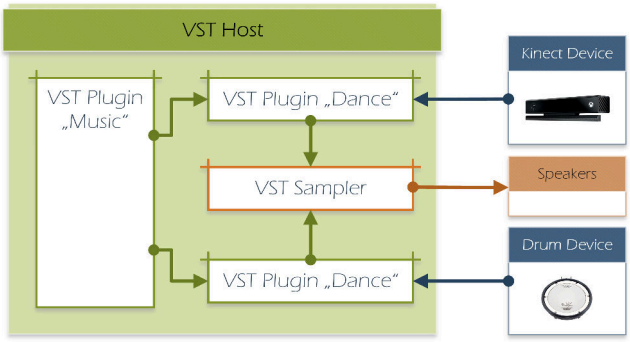

Figure 1: DDMMs VST structure

2. GOALS

These problems can be solved by converting existing plug-ins into an independent system of only one VST plug-in with a new framework, which provides a common basis for the existing plug-ins supporting them with all their functionalities. Thus, in this paper the basis for future extensions of DDMM, as well as the new development of VST-based NIMEs in general, will be defined and implemented in a first example. Tools for the internal plug-in communication and for communicating with external parts will also be added. The loaded plug-ins will furthermore be monitored and their runtime measured in order to find errors more quickly. In addition, a service monitoring system shall be implemented to illustrate the runtimes of the individual components graphically in order to make deviations in the execution time visible, which supports the optimization process. As an example, we will migrate our existing system DDMM into the new framework. After the system has been transferred, it will be re-evaluated in three iterations (functional tests, unit tests and performance tests) to find further problems and possible improvements for the future.

NIME'18, June 3-6, 2018, Blacksburg, Virginia, USA.

3. RELATED WORK

With its combination of different interfaces, DDMM can best be described as a sensor network. It has been proven that sensor networks benefit from event-driven architecture (EDA).[6, P. 39] Tests have shown that EDAs are capable of much smaller event intervals than approaches using polling without any dropped events.[11] There are also some projects, utilizing a Space-Based Architecture (SBA), which combines paradigms of both, event-driven and service-oriented architectures, in order to handle the coordination of distributed applications and autonomous components while keeping the dependencies to a minimum. One of these projects used the XVSM middleware to build an event driven extension, which handles asynchronous jobs, expressive triggering conditions and time-based constraints.[7]

EDA is also used in a variety of other research fields like smart home and smart city[12, 8], machine learning[24], distributed automation systems[17] and mobile computing[1]. Furthermore it is used in several IoT standards (compare listed standards in [2]) and in the design of context aware systems (CAS)[4]. Even though there are no cases in research known to us where EDA is used in the context of VST-plug-ins, the positive results in other fields are a good indicator for successfully establishing the foundation of the intended framework on the paradigms of EDA. One of the key aspects of DDMM is its modularity.[5] To keep this independence while creating the new framework, it might be reasonable to make use of the dependency injection (DI) pattern, since it has proven to effect a variety of quality attributes of a software system, like in particular the extensibility. Thus it would be easy to add more interfaces to DDMM and at the same time provide testability by supporting mock objects in unit testing[25] and re-usability by breaking transitive dependencies[28].

In research, dependency injection is mostly found in web applications and Java development. In the context of web development there were attempts to integrate DI into different frameworks like the one proposed by Nozawa, Hotta and Hagiwara in order to develop mobile websites.[16] In June 2012 with the release of AngularJS[1] DI became key component in one of the most popular front-end frameworks which is relevant to this day.[20] Within the Java environment there are a couple of frameworks providing DI functionality like Spring[2] or PicoContainer[3]. Some projects tried to build upon, and extend these frameworks[10] or developed new ways of using DI, for example in combination with the adapter pattern[21]. Other fields of research, where DI is used, are game engines[18] or even service-oriented applications (SOA)[15]. SOAs are very close to EDAs and there were already successful attempts to combine them.[13] Thus, a combination of EDA and dependency injection might solve the issues mentioned before.

4. THE FRAMEWORK

The existing concept of DDMM is based on the JUCE framework[4]. This does not only simplify the development of VST plug-ins, but also offers a variety of useful methods for handling MIDI data, the file system and the development of graphical user interfaces. The new framework will therefore also be built on the basis of JUCE. The fact that these JUCE features will not be changed should facilitate the porting of the existing plug-ins. At the same time, however, JUCE dictates a structure on which the new VST framework must be based. Thus, the main class of the framework has to inherit from *AudioProcessor*, which cyclically invokes the *processBlock* method.[22]

[1]https://angularjs.org/
[2]https://spring.io/
[3]http://picocontainer.com/
[4]https://juce.com/discover

4.1 Requirements

After an extensive analysis of the initial system, some weaknesses have come to light. In order to avoid well-known errors and to improve performance issues, existing problems should be taken into account when developing the framework and restructuring DDMM. These issues will now be examined in detail and requirements for the new framework will be formulated:

4.1.1 Plug-in communication

One of the key aspects when creating a modular VST-based NIME is the communication between different plug-ins. The VST-protocol provides a unidirectional connection via a MIDI-buffer that is handled by the VST-Host, but no way to respond back to the plug-in that sent this MIDI-stream. In our use case the plug-in "Music" generates a composition and sends it to other plug-ins which each handle additional data from specific connected interfaces. Based on the comparison between the interface data and the received composition, the "Music" plug-in changes the composition dynamically(see figure 1). The prototype solved this issue with inter-process communication using pipes for the benefit of modularity, but this caused a delay in the processing of inputs, e. g. when changing the key, which in a time-critical topic like music can lead to a non-intuitive behavior of the program and frustration with the users. Furthermore, with such a limited form of communication, the cooperation of several plug-ins is difficult to realize and could lead to problems with subsequent development.

4.1.2 Consistent and measurable components

Because of JUCE all synchronous tasks are processed by the *processBlock* method, but in a modular NIME the further structure is left to each plug-in. With *DeviceInputPluginProcessor* the input plug-ins in our use case already have a common approach for matching inputs with the metronome, but the architecture of plug-in "Music" is completely different. A precise specification of what has to be done when and in which area, which is generally admitted and consistent for all plug-ins, would not only benefit the overall overview- and orientation time, but would also be helpful in the development of standardized performance measurements and unit tests.

4.1.3 Buffers and thread safety

When dealing with data from sources outside the VST-host, buffers usually are each filled by a separate thread and read by the main thread of the VST-host. This way thread-related race conditions cannot be completely excluded. For this reason, it is necessary to protect the buffer with a mutex to prevent possible problems.

4.2 Concept

In order to offer a more uniform structure and thus enable the unproblematic migration of old plug-ins as well as the new development of further plug-ins, a tripartite workflow will be implemented (see figure 2). It is based on a modified version of EDA and represents the three elements of the EDA processing model (event-object, -source and -sink)[6, p.51]. The event-objects are represented by the interface *Event*, which will replace all existing buffers. The interface *Condition* is designed for the event-source, which determines whether an action must be executed based on the existing instances of *Event*. The interface *Trigger* is developed to

map the event-sink, which will execute a particular task as soon as it is called by a *Condition*. Concrete event-buffers and triggers will be implemented as singletons in order to allow access to the same instance in each module.

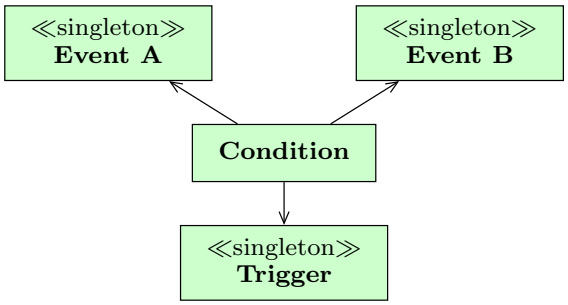

Figure 2: EDA concept with event, condition and trigger

4.2.1 Modules

In order to improve communication between different plug-ins, the existing structure will be abolished and thus the previously independent VST plug-ins are integrated to the new interface *Module*. This interface will contain some methods of the *AudioProcessor* class and additional methods are provided to register the components implemented by the individual modules within the context of the main system. In this way, all existing VST plug-ins merge into one. As a result, the system loses modularity, but individual functionalities and properties of the system could be switched off, for example by VST parameters from the graphical user interface. This could be implemented with checkboxes allowing the deactivation of single guitar strings, for example, or the deactivation of a whole guitar interface as part of an fictitious NIME with several string-based modules. In the case of DDMM it can be used to deactivate the drum module, if it is not needed or unwanted. So as long as this checkbox is not marked, the module could ignore all incoming strokes on the drum or even terminate the connection to the drumpad.

4.2.2 Events

With the transition to the new framework and the fusion of memory space, it will be possible to make all buffers visible outside of their respective module. Thus, each buffer can be used by all modules. To do so, all buffers are transferred to the newly created interface *Event* and each concrete event will be implemented as a singleton enabling each module to access a shared set of event objects. All entries in the event buffer should bear the timestamp of their creation and a lifespan. This enables the system to automatically iterate over all events and search for outdated entries, which can then be automatically removed. The origin of the data used to fill the event buffers may vary significantly. For example, data sources within the VST-host transmit their data via MIDI-buffer and write into the buffer when the *processBlock* method is called, but some other interfaces, like any sort of external inputs, for example a drum machine, require a separate thread to retrieve the data. In DDMM we have a similar situation with the music plug-in on the one hand, generating its composition inside the *processBlock* method, and the plug-in ”Balanceboard” on the other hand, which receives data asynchronously. For this reason each module is designed to choose how and when it supplies its events with data. If possible, however, a validity check should take place before the event buffer is filled, so that later on it can be assumed, that each record represents a correct and reliable event.

4.2.3 Triggers

Triggers are the executing components of the framework. All externally visible changes, such as writing to the audio- or MIDI-buffer, should occur within an instance of *Trigger*. Since triggers are implemented as singleton, they can be addressed by all instances of *Condition*(see next section) without having to fear unforeseen mutual effects. The execution time is divided into two parts: Since triggers can be addressed by several conditions or other conditions can have an indirect influence on the output of the trigger, it can be useful to save tasks and only execute them once all conditions have been processed. For this purpose, all triggers should implement the *fire* method, which will be called centrally in the *processBlock* method after all conditions have been checked. For example, different triggers may want to change important parameters for the current block. To include these eventualities, it would be useful to save all parameters (eg. note to be played) in an internal buffer first and when calling up the *fire* method, after all conditions have been executed, write the final changes to the MIDI-buffer.

4.2.4 Conditions

Conditions are check routines that determine the need for action for a trigger based on the contents of the event buffers. To do this, concrete conditions have to implement a newly developed interface *Condition* that prescribes a method, which will be executed centrally at a fixed point in time. The execution of conditions should always take place after the events have been cleared and filled, so that conditions will only receive up-to-date records.

Each condition accesses any number of events and at least one trigger. As the EDA processing model demands, event-sources are just responsible for detecting the need to fire an event-object without knowing how it is processed[6, P.51]. For this reason no external changes to variables will be made within conditions. This separation of powers is supposed to prevent immobility and viscosity of the system if strictly followed. Although an ”easier” implementation would be possible, this would weaken the structure of the entire system[14]. In addition, conditions can be switched on and off as required without affecting other conditions due to the low binding to other components.

4.3 Processing

The concept is based on the processing model of an EDA, but due to the use of the VST protocol, the processing of the current audio- and MIDI-buffer must be completed within each block. For this reason a software pipeline is implemented around the processing model, controlling the scheduling of the individual components. The process starts with the initialization of the framework by JUCE when the plug-in library is loaded into a DAW. At this point, all modules defined in the framework are activated and requested to register components such as VST parameters, events, conditions and triggers. After the initialization of the plug-in is completed, the system expects the first call of the *processBlock* method. Each time JUCE calls this procedure, the buffers for MIDI and audio will be processed. Due to the frameworks structure, this cyclically called operation is divided into individual pipeline segments in which the individual tasks are executed (see figure 3).

The first step is to execute the cleanup methods of all events, where all obsolete entries, that were created either in an earlier run or at any time by a separate thread, are removed from the event buffers. The second step is to execute the *processBlock* method of the individual modules. This is called exactly like the *processBlock* method of the plug-in

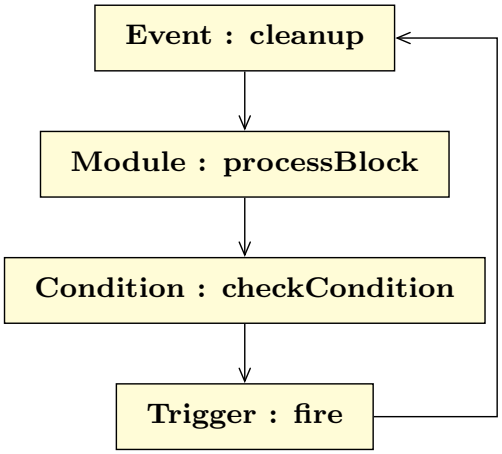

Figure 3: Software pipeline concept

with the current audio- and MIDI-buffer of the ongoing block. Within the methods, the modules can fill their events or modify parameters that have been changed by the user with the help of the GUI. After the events have been cleaned up and filled with current data, each module has the opportunity to send important parameters to the conditions. Then the *checkCondition* method of each condition will be executed in order to evaluate whether actuating of triggers is required.

In the last step, after the existing conditions have been checked, all registered triggers get the possibility to make final changes to the buffers of the block or other parameters and then the fire methods will be called. Once all of the above steps have been executed, the processing of the current block is finished. Due to the fact that all tasks running in the main thread are executed in the *processBlock* method of the plug-in, an exception handling and a time measurement can be integrated into this cycle. Thus, by means of systematic centralization, errors occurring at runtime can be identified more quickly and at the same time the components with the highest optimization potential can be determined without great effort.

4.4 Implementation

The concept of Dependency Injection is not implemented in the standard library of C++ [3, 9], so additional libraries have to be used. For this purpose a selection of frameworks such as PocoCapsule[5], Hypodermic[6], Wallaroo[7] and Infector++[8] is available. However, these frameworks have either not been maintained for years, do not meet the system requirements, or have a very limited feature list. After reviewing all frameworks, Google Fruit[19] turned out to be the best solution for the application.

Fruit uses meta-programming and some special C++11 features to detect injection problems and performs most checks already at compile time. This makes it particularly efficient at runtime and ensures low overhead. However, JUCE uses the Microsoft MSVC compiler, which did not provide the required features for a long time, but since the release of Visual C++ 2017 in March 2017 all required features are available. Official support for MSVC is currently experimental, but initial tests have shown that the system is reliable and therefore Fruit could be used in this work. The framework makes it possible to break down the source code into components. These have to implement an interface into which the functionalities of the components are then injected. Components can use or be based on other components and form new components in combination. The major advantage of this pattern is the loose coupling between the components, which keeps the system easily expandable[19]. In our framework for each of the three components *Event*, *Condition* and *Trigger* appropriate interfaces were defined at first. They have to supply a constructor marked with the Fruit *INJECT* macro, which is used as a reference to detect errors with injection-candidates at compile time. Now different implementations can be injected into the interface. Each implementation has to provide a method, supplying the Fruit injector with a matching *Component* object using the *bind* function like for example the *PlayMetronomeEvent* implementation of the interface *Event* (see Listing 1).

To realize singleton *Events* and *Triggers* every implementation of these interfaces is given a static instance, which is bound using the *bindInstance* method (see Listing 1, line 10). This way every time a specific *Event* or *Trigger* is injected, it will always be a reference to a globally unique instance. During runtime *Conditions* have to look for specific *Events* and fire connected *Triggers*. To provide them with these components, they are defined as requirements. This way Fruit injects them into the constructor of the *Condition* as it is created. Fruit can not distinguish between multiple implementations of the same interface. However, this can be solved by using the "Annotated" macro (see Listing 1, line 6 and 8) to specify which specific implementation should be used.

```
1  static PlayMetrEvent pmeinstance;
2
3  const Component <Annotated<MetrEvent, Event>>&
4    getPlayMetrComponent() {
5      static const Component
6        <Annotated<MetrEvent, Event>> comp;
7      comp = createComponent()
8        .bind<Annotated<MetrEvent, Event>,
9          PlayMetrEvent>()
10       .bindInstance(pmeinstance);
11     return comp;
12 }
```

Listing 1: Function to generate a Fruit component

To migrate existing VST plug-ins into the new framework, a new module has to be created for each of them. The functionality needs to be shifted into the new component design and the components for each module have to be provided by the three central methods *registerEvents*, *registerConditions* and *registerTrigger*, where they are initialized and provided to the main system. Components are managed by the *AudioProcessor* class, created by JUCE. This class can be considered the main class of the plug-in and is responsible for running the main-loop we defined in 4.3, which is executed blockwise by the DAW. Since every component of the plug-in is executed in one central method, this method will be used to implement analysis tools like performance tests (see section 5.3). To provide a simple control interface for the plug-in, *AudioProcessor* can also implement a new GUI using the tools provided by JUCE. In case of DDMM, it automatically integrates the existing GUIs provided by the former plug-ins. This way the whole system is very modular and every module can be removed, changed or replaced with no effect on the functionality of the GUI.

5. TEST ENVIRONMENT AND RESULTS

The evaluation of the system was divided into three main phases: First, the system was checked and evaluated from the user's point of view. As there have not been done any

[5]https://github.com/skyrpex/pococapsule
[6]https://github.com/ybainier/Hypodermic
[7]http://wallaroolib.sourceforge.net/
[8]https://sourceforge.net/projects/infectorpp/

functional changes to the existing system and since it has already been tested in practice, these checks will be limited to laboratory conditions. In the second phase, we wrote unit tests for the individual classes, in order to support the observations from phase one with measurable results and to locate possible sources of error. This should ensure that the test environment is easily extensible for future developments of DDMM. Since audio programming is highly time-critical, a centrally managed performance test was used in the third and final phase to analyze the new system and search for components that need to be optimized.

5.1 Functional Tests

To carry out the function test, test cases were first developed and entered into a protocol template. These test cases were then processed sequentially, the actual results were compared with the expectations and any observations recorded. In four of these tests errors occurred, which could be traced back to two errors in the program code. The first problem was found, when changes have been made to the sample rate or the size of the audio buffer while the plug-in was running. When changing a sound parameter in the DAW, JUCE returned a *DanglingPointerException* in a *MidiFile* object. Secondly, an error occured when attempting to save or restore the current state of the plug-in. When saving the state, the *getStateInformation* method is executed in the audio processor. In this, an XML element is created for each module and named with the module name. This resulted in an *isValidXMLName Exception*. The reason for this is the fact that there was a blank in one of the module names.

5.2 Unit Tests

A variety of libraries are available for unit tests under C++, such as CPPUnit3[9], CUTE4[10] or Boost5[11]. However, JUCE itself provides unit test classes, which are equivalent to most alternatives in terms of functionality. Since the existing project is managed by Projucer and it does not support the creation of additional build targets, a new test project needs to be created. In the test project the library, which is generated when compiling the plug-in, as well as the "fruit.lib" are then submitted to the linker as an external dependency. With this workaround the unit tests can always be compiled with the latest version of the VST plug-in. For the execution of the tests a *UnitTestRunner* object has additionally been implemented. This must be instantiated in the main method of the application and then executes all unit tests registered in *UnitTest::getAllTests*[23].

5.3 Performance Tests

The *PerformanceTest* class was implemented to perform the performance tests. It provides a number of methods to define individual measurement points and then uses the JUCE *FileOutputStream* to export the measurement results to a CSV file. The largest measurement unit is determined by the methods *startNewMesurement* and *endMesurement*. These represent a line of the CSV and are executed at the beginning and end of the "processBlock" method. First, the total execution time of all components should be compared with each other. In our use case it was found that the *onProcessBlock* method of the Dynamic Composition component takes 61% of the execution time of DDMM identifying this component as a key issue in performance optimization. The two components *HitDrumOnMetronomCondition* and *PlayMidiNoteTrigger* also require a substantial share of execution time. However, these components were expected to be subject to high stress and 8% of the execution time in each case means that their priority for optimization is rather low. When considering the execution time in each cycle (see figure 4) it is obvious that the execution time of Dynamic Composition (displayed in dark blue) is not only continuously high, but also subject to strong and irregular jitter. All other components have a relatively small amplitude with regular peaks. It is also noteworthy that the drum module has a very short execution time, which is due to the fact that it outsources its work to a separate thread. This approach could also be used for other modules.

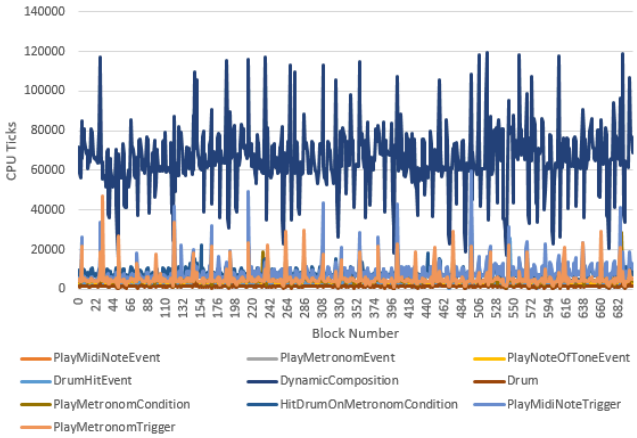

Figure 4: Execution Times

6. CONCLUSION AND FUTURE WORK

In summary, it can be said that the goal of this work, the development of a framework for evaluating and optimizing VST plug-ins, can be considered fulfilled. We integrated the concepts of EDA and dependency injection into VST plug-ins to address problems in the development of modular NIMEs based on defined objectives and the requirements formulated in the analysis of an existing system. The implementation of the proof of concept has shown that the overall concept is valid and the porting of the plug-ins into the new structure ran smoothly and quickly. This speaks for a good extensibility of the system. During the evaluation of the system, functional tests showed that the occurring errors were easy to locate and performance tests showed which components had the highest optimization potential.

As for our use case there are numerous possibilities for the future development of DDMM in the remaining term of the project and beyond. First of all, the other plug-ins (Kinect and Balanceboard) should be transferred to the new framework. Furthermore it would make sense to optimize the performance of the existing modules, by implementing the possibilities already mentioned. In addition, new modules with different input options or other concepts of interaction could be developed. For example, a free play mode without a clocked metronome would be an option. Finally, it could also be useful to develop a scripting language that generates new conditions with the help of simple controls in order to be able to implement new concepts in DDMM without relying on extensive programming skills.

7. REFERENCES

[1] Applying event-driven architecture to mobile computing. In *2013 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 58–63, Piscataway, NJ, 2013. IEEE.

[9] http://cppunit.sourceforge.net/doc/cvs/index.html
[10] http://cute-test.com
[11] http://www.boost.org

[2] Open connectivity foundation. https://openconnectivity.org/developer/specifications/upnp-resources/upnp, 2018. Accessed: 2018-01-10.

[3] Standard c++ library reference. http://www.cplusplus.com/reference/, 2018. Accessed: 2018-01-11.

[4] Z. Babaei, A. M. Rahmani, and A. Rezaei. Real-time reusable event-driven architecture for context aware systems. In *2016 24th Iranian Conference on Electrical Engineering (ICEE)*, pages 294–299, Piscataway, NJ, 2016. IEEE.

[5] D. Becking, C. Steinmeier, and P. Kroos. Drum-dance-music-machine: Construction of a technical toolset for low-threshold access to collaborative musical performance. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 112–117, Brisbane, Australia, 2016.

[6] R. Bruns and J. Dunkel. *Event-Driven Architecture : Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse.* Springer, 2010.

[7] S. CraB, E. Kuhn, V. Sesum-Cavic, and H. Watzke. An open event-driven architecture for reactive programming and lifecycle management in space-based middleware. In M. Felderer, H. Holmström Olsson, and A. Skavhaug, editors, *43rd Euromicro Conference on Software Engineering and Advanced Applications*, pages 189–193, Piscataway, NJ, 2017. IEEE.

[8] L. Filipponi, A. Vitaletti, G. Landi, V. Memeo, G. Laura, and P. Pucci. Smart city: An event driven architecture for monitoring public spaces with heterogeneous sensors. In J. Lloret Mauri, editor, *Fourth International Conference on Sensor Technologies and Applications (SENSORCOMM), 2010*, pages 281–286, Piscataway, NJ, 2010. IEEE.

[9] Information technology – programming languages – c++. Standard, ISO/IEC JTC 1/SC 22 Programming languages, their environments and system software interfaces, Dec. 2014.

[10] K. Jezek, L. Holy, and P. Brada. Dependency injection refined by extra-functional properties. In M. Erwig, editor, *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 255–256, Piscataway, NJ, 2012. IEEE.

[11] R. Kannan, J.-H. Jo, and H. S. Go. Improved event driven architecture for tizen sensor framework. In T. Catarci, editor, *1st International Conference on Mobile Software Engineering and Systems (MOBILESoft 2014)*, pages 75–78, New York, NY, 2014. Association for Computing Machinery, Inc.

[12] T. Kato, N. Ishikawa, and N. Yoshida. Distributed autonomous control of home appliances based on event driven architecture. In *2017 IEEE 6th Global Conference on Consumer Electronics (GCCE)*, pages 1–2. IEEE, 2017.

[13] Z. Laliwala and S. Chaudhary. Event-driven service-oriented architecture. In V. C. S. Lee, editor, *International Conference on Service Systems and Service Management, 2008*, pages 1–6, Piscataway, NJ, 2008. IEEE Service Center.

[14] R. C. Martin. Design principles and design patterns. *Object Mentor*, 1(34), 2000.

[15] C. Mateos, M. Crasso, A. Zunino, and M. Campo. Separation of concerns in service-oriented applications based on pervasive design patterns. In S. Y. Shin, editor, *Proceedings of the 2010 ACM Symposium on Applied Computing*, page 849, New York, NY, 2010. ACM.

[16] T. Nozawa, H. Hotta, and M. Hagiwara. A development framework for mobile user-interfaces based on html centric dependency injection. In *Proceedings of the 2008 IEEEWICACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 02*, pages 186–189, Washington, DC, 2008. IEEE Computer Society.

[17] C. Pang, J. Yan, and V. Vyatkin. Time-complemented event-driven architecture for distributed automation systems. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(8):1165–1177, 2015.

[18] E. B. Passos, J. W. S. Sousa, E. W. G. Clua, A. Montenegro, and L. Murta. Smart composition of game objects using dependency injection. *Computers in Entertainment*, 7(4):1, 2009.

[19] M. Poletti. google/fruit. https://github.com/google/fruit, 2018. Accessed: 2018-01-11.

[20] M. Ramos, R. Terra, and M. T. Valente. Angularjs performance: A survey study. *IEEE Software*, page 1, 2017.

[21] A. Roemers, K. Hatun, and C. Bockisch. An adapter-aware, non-intrusive dependency injection framework for java. In M. Plümicke and W. Binder, editors, *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'13)*, ICPS, page 57, New York, New York, 2013. Association for Computing Machinery.

[22] ROLI Ltd. Audioprocessor class reference. https://juce.com/doc/classAudioProcessor, 2018. Accessed: 2018-01-11.

[23] ROLI Ltd. Unittestrunner class reference. https://juce.com/doc/classUnitTestRunner, 2018. Accessed: 2018-01-23.

[24] A. Roy, S. Venkataramani, N. Gala, S. Sen, K. Veezhinathan, and A. Raghunathan. A programmable event-driven architecture for evaluating spiking neural networks. In *ISLPED 2017*, pages 1–6, Piscataway, NJ, 2017. IEEE.

[25] F. Solms and L. Marshall. Contract-based mocking for services-oriented development. In F. F. Blauw, editor, *SAICSIT 2016*, ACM international conference proceedings series, pages 1–8, New York, NY, USA, 2016. The Association for Computing Machinery, Inc.

[26] C. Steinmeier and D. Becking. Toddlers testing ddmm: Evaluation results and ideas towards creating better learning environments for small children. In *25th International Conference on Computers in Education (ICCE)*, 2017.

[27] C. Steinmeier and D. Becking. Visual feedback for ddmm: A simple approach for connecting and synchronizing unity animations with events from vst plugins. In *43rd International Computer Music Conference (ICMC)*, Shanghai, China, 2017.

[28] H. Y. Yang, E. Tempero, and H. Melton. An empirical study into use of dependency injection in java. In F. K. Hussain, editor, *19th Australian Conference on Software Engineering, 2008*, pages 239–247, Los Alamitos, Calif., 2008. IEEE Computer Soc.