INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2025, VOL. 71, NO. 4, PP. 1–7
Manuscript received September 3, 2025; revised October 2025. doi: 10.24425/ijet.2025.155469

Data-driven dependency injection for embedded software, enhancing reusability with C++

Slawomir Niespodziany

***Abstract*—This article presents an architecture for engineering reusable embedded software using modern C++ principles and a custom–built dependency injection framework. It details the framework's design, specifically tailored for resource-constrained environments. The framework promotes modular and testable architecture. Its data–driven (via Json file) configuration defines component dependencies and determines their instantiation. The article demonstrates how such approach facilitates component decoupling and provides a viable path for developers to create scalable, portable, and high-quality embedded software, significantly reducing future development efforts.**

***Keywords*—data driven; dependency injection; embedded; reusability**

I. Introduction

WHEN it comes to non-recurring engineering costs of building an embedded system, the contribution of software development grows, compared to the cost of hardware design in recent years. The reason for that has two origins.

Firstly, semiconductor complexity grows accordingly to the Moore's law [1]. While several decades ago the hardware capabilities were limited and only basic software algorithms were be able to fit into memory and execute with the available processing power, now the progressing development has lead the industry to a point where hardware is no longer the bottleneck. Silicon manufacturers constantly put more features into their products. The chips scale down. The effort from the mechanical design perspective doesn't change much. PCB design is still a task which can be tackled by individuals or small teams. On the other hand side, the software for more complex systems is sometimes developed by hundreds of engineers. This happens to be a problem from various perspectives (e.g. management or budget), but at certain project scale (e.g. spaceships, self-driving cars) increasing the team size may not necessarily lead to increasing its capabilities [2]. With a little luck, AI based solutions may be a remedy in some time, but that doesn't seem to be yet.

Secondly, for a business case to be successful nowadays, the functionality of a device shall not be trivial. The unique intellectual properties embedded and tightly integrated into products are essential to win markets. No less important is the ability for the product to adapt to changing expectations. The embedded electronics - once shipped, remain untouched for the product's lifetime. On the other hand, the software is a subject to constant OTA (or non–OTA) updates [3], sometimes making its development a never ending story.

Slawomir Niespodziany is with Institute of Computer Science, Warsaw University of Technology, Poland (e-mail: slawomir.niespodziany@pw.edu.pl).

II. Reusable software

One solution to mitigate the mentioned problems is reusing once developed software. A prerequisite for this is implementing it in a way which allows for such reuse in the future. In long term this leads to enhancing scalability and maintainability [4], directly impacting the underlying business needs.

Over the years software industry has developed multiple methodologies and patterns which support reusability. They are, however, underestimated in embedded engineering [5]. It is a complex field in the sense of combining various sciences and problem domains (e.g. electronics, system engineering, signal processing, low-level development). For this reason, it happens to lag behind the industry in applying some of the modern practices. Recently [6], such include contenerization, dependency management, CI/CD pipelines, evolving build tools and incrementing language standards (e.g. C++14). However, since all of those are in reach, the only missing piece is an established practice, directing how to utilize them. It may seem difficult to provide a methodology applicable too any new project, as embedded software is usually tightly coupled with the hardware specific for each. However, the C++ language standard is a great starting point.

This paper describes an architecture proposal and a framework implementing it - *Diff* - **D**ependency **I**njection **F**ramework in its **F**irst variant. The proposed architecture structurizes process of designing embedded software, enabling it to be reused in the future. Its main design goal is to provide universal and testable software, portable between an arbitrary set of architectures and targeted for embedded systems (but with an option to be applied in other projects too). The fundamental building block is Dependency Injection pattern [7]. Because of the nature of embedded platforms, the implementation language is C++14, shall nowadays be supported by all maintained compilers (it is already over 10 years old). Compared to C, or even C++03 it offers a significant improvement in capabilities, which allow for implementing various automations, described further.

III. Componentization

Well designed reusable application is composed of multiple independent parts [8]. Each can be either used to assemble another application, or replaced for its updated version (e.g. to fix bugs) without the impact on other parts. In this sense such parts can be viewed as building blocks. In the context of this paper they are referred to as *components*. Each component provides certain, well-defined functionality.

Components in an application interact with each other. One way to achieve this is to make one component depend on another. As a result the first knows the interface of the second. However, this can easily result in leakage of the internal details of the dependency into its interface, resulting in tight coupling of both components.

A better approach provides the interface as a separate entity, requiring both components depend on that interface [9]. In the proposed architecture it is a separate component - an *interface component*. It only consists of the definition of the interface, no implementation. In most cases it will fit into a single header file, thus it is easy to maintain and is expected to change rarely. Nevertheless, it is an important element, as both neighboring components depend on it (Figure 1). One component will implement that interface and the other will use it.

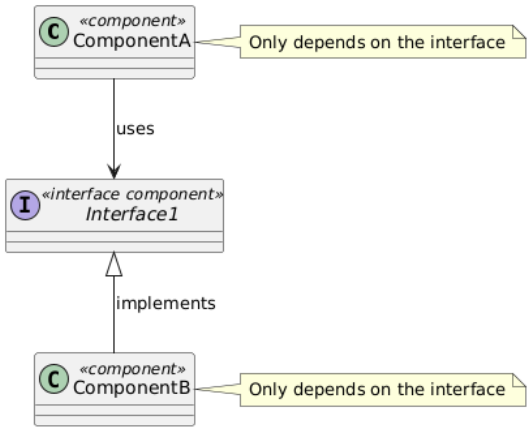

Fig. 1. A thin, independent interface provides a way for both components to interact with each other. *ComponentA* uses interface implemented by *ComponentB*. Both components are dependent on that interface, but not on each other. This results in a decoupled set of building blocks for the application.

The components consisting of implementations of various features are simply referred to as *components*.

A. Interface component

As interface component consists of no implementation, it fits into one or several header files and can be stored in form of a header–only library. It is provided as an abstract C++ class, consisting of pure–virtual member functions. This isolates the interface. It allows other components to inherit from it and implement the required functionality.

The advantage of such approach comes primarily from the design order. Firstly, an interface is defined and released. Only then a component inherits from it and provides the implementation. This prevents from information leakage from the inside of a component towards the inter–component boundary.

An interface obviously needs to be thoughtfully designed, as it is expected to account for all the information, which may be passed through it in the future use–cases. It is a subject for updates and versioning. New interface versions may be released in time and they may even form an inheritance hierarchy to maintain backward compatibility. It must be kept in mind, however, that using an updated interface version will result in the need of updating all the implementations depending on it. While it may seem to be a problem, it shall rather be considered as a form of controlled damage. Usually, when software is developed without this kind of structure, updates which drag multiple other updates along occur in random points. With the proposed architecture this area is known upfront and thus can be kept minimal to reduce the chance for an update to be necessary.

B. Component

Regular component can be accessed and communicated with through the interfaces it inherits end exposes. It consists of algorithms and data structures and provides implementations for the abstract functions. A component is implemented in form of a C++ class and may utilize other classes and external libraries, depending on the domain of the problem it solves. It may inherit from multiple interfaces. This allows for those interfaces to be fine grained and independent from each other.

A component only depends on its related interface components. Because no other components depend on it, it can be easily replaced by its equivalent (e.g. an updated version), without breaking any dependency relations.

1) Implementation: For an implementation of a component there are no restrictions regarding its internals. From the framework perspective it has to be provided as a C++ class and inherit a base class template *diff::Component*. This base class serves several purposes. Firstly, it stores ID of a component used by the framework to identify it and the dependencies it exposes. Secondly, it provides the framework with information about the interfaces through which a it shall be exposed as a dependency. This provides an important distinction from the inheritance hierarchy, which simply results from the implementation needs and should not be considered by the framework. Lastly, the base class provides a member function used for per instance configuration, using user–provided set of parameters.

A component may expose dependencies in two ways - directly and from the side. The component object may be a dependency itself. In such case it implements a specific interface and is then injected as it. For a more fine grained functionality, there may be a need to expose multiple smaller dependencies. For this reason, a set of internal objects, which implement certain interface can be exposed from the side of the component. In such case the ID of each such dependency is composed of the parent component ID and a suffix provided individually for each dependency. In such case the parent component, which owns its side dependencies, implements a member function which fills up a map consisting of suffixes and references to the internally owned dependency objects.

The first parameter of the base class template is the component type itself. The framework uses CRTP [10] (Curiously Recurring Template Pattern) to automate several tasks and offload them from the user. Then a wrapper type *diff::as* can be used to make the component inherit a selected interface. This also makes the framework notified about the implemented dependency type and will result in exposing it as a dependency of the interface type. A wrapper type *diff::side* can similarly be used to expose multiple side dependencies, but it also requires a corresponding member function to be implemented. Examples are shown in Figure 2 and Figure 3.

```
using diff;
class MyComponent : public Component<
                            MyComponent,
                            as<InterfaceA>> {
public:
 // Implementation of InterfaceA

 ...
};
```

Fig. 2. Component *MyComponent* inherits from a base class template *diff::Component*, parametrized with the component type *MyComponent* itsself (CRTP) and a wrapper class *diff::as* for *InterfaceA*, further implemented by the component itsself. This enables framework to expose the component as a dependency of type *InterfaceA*. Its identifier is the same as the component ID.

```
using std;
using diff;

class MyComponent : public Component<
                            MyComponent,
                            as<InterfaceA>,
                            side<InterfaceB>> {
 ...

protected:
 class MyInternal : public InterfaceB { ... };

 MyInternal a, b;

 void side(map<string,
               reference_wrapper<InterfaceB>>
             &sideDeps) {
  sideDeps.emplace("a", a);
  sideDeps.emplace("b", b);
 }

};
```

Fig. 3. By extending the example in Figure 2, base class template is parametrized with wrapper type *diff::side* for *InterfaceB*. This also requires providing member function *side* and enables the framework to expose objects returned by this function. In this case, two objects - *a* and *b* - are exposed as *InterfaceB* under the identifiers composed with the parent object ID and given suffixes.

Dependencies are passed to a component via its constructor. The framework automatically determines the number and types of dependencies to be injected for a component. Depending on the user provided dependency identifiers proper dependencies are selected for injection. Figure 4 shows an exemplary implementation.

```
#include <InterfaceA.h>
#include <InterfaceB.h>

using diff;

class MyComponent :
            public Component<MyComponent> {
public:
 MyComponent(InterfaceA &a, InterfaceB &b);

 ...
};
```

Fig. 4. Consuming dependencies is as simple as accepting them in component constructor. The injection is handled by the framework as described in further sections.

C. Module

Component is a logical entity, but it has to be released in a physical form. For an interface component this form is a header-only library, usually consisting of a single header file. It does not require building, thus it is natively platform independent. Regular components are released in form of statically linked libraries. Less often, depending on the requirements and possibilities, they may also be dynamically linked. Storing components in form of C++ libraries allows them to be processed, managed, versioned and distributed with use of tools developed for that purpose and used across the industry (e.g. CMake, Conan2).
Another benefit of keeping all the components in separate libraries is the fact that they can be either linked with the final binary or not. This trivial integration method enables automated composition of multiple varaints of an application and can be automatically realized by CI/CD pipelines [11].

D. Performance considerations

The presented architecture is based on Dependency Injection design pattern which decouples components from each other. This comes at a cost of referencing dependencies through their polymorphic interfaces, not directly the objects. Because of this indirection, there is a slight overhead on performance. The compiler firstly accesses the virtual functions table and then calls the actual implementation [12]. This additional memory access may impact the performance of very specific applications, composed of multiple components, accessed very often (e.g. a component per each image pixel, or per data sample). For such high–performance applications a simple workaround could be to use the polymorphic interface to hand over memory pointers and then operate directly on buffers. In most cases, however, this effect is negligible. When considering the benefits brought by component decoupling, the resulting trade–off is desired.

IV. Component management

The previous section focused on how to implement individual components to make a decoupled design. This section describes mechanisms used internally by the framework to allow for data–driven management, based on user–provided information.

A. Component factory

A component of a given class may have multiple instances. Each instance is constructed during application startup and has its own configuration. This is handled by the framework, based on data loaded from the user. Startup time instantiation allows for the application to be reconfigured without rebuilding. For this purpose each component has its own factory and each factory is referenced by the initializing code when a component requires constructing.
Framework provides a template class *Factory*, which inherits from an *AbstractFactory*. This allows the initialization code to be independent from concrete component classes. All factories available within an application are accessed through a *FactoryRegistry*, which is a publicly accessible singleton class, aggregating them. Factory of each component type requires to be instantiated on its own and registered within the FactoryRegistry to be further available. This actions are automated and realized by the RAII-like *FactoryRegisterer*.

From the users point of view, to allow the framework to construct components of a given type, a FactoryRegisterer for that type has to be instantiated as a static variable in its own module. When the module is linked with the application binary, FactoryRegisterer performs construction of the underlying factory and registers it within FactoryRegistry. This happens without user intervention. From there, the framework engine is able to construct enu number of components of the discussed type. Figure 5 presents all the code required for that purpose from the implementers perspective. Figure 6 depicts architectural relationship between the described classes.

```
using diff;

class MyComponent :
            public Component<MyComponent> {
 ...
};

FactoryRegisterer<MyComponent> g_registerer;
```

Fig. 5. *FactoryRegisterer* (a RAII–type class) constructs a factory for the given type of components and registers its within a global *FactoryRegistry*. This happens transparently when the module is linked with the application binary.

B. Dependency registry

When a component implements interfaces meant for being injected as a dependencies, its reference is stored in a *DependencyRegistry*. It is a non–owning container, which holds references to all dependencies in the application. Each entry stores a reference to the target, its identifier and knows its (type

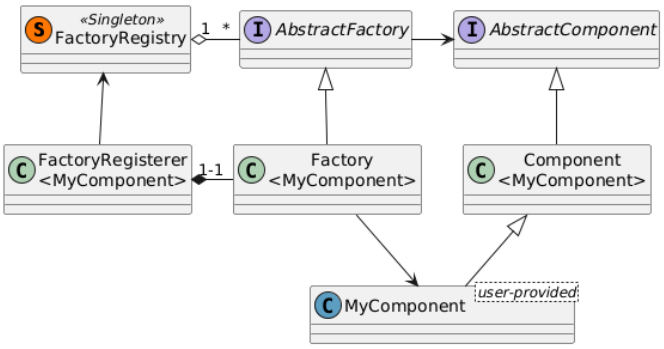

Fig. 6. User component derives from the *Component* template, which inherits from *AbstractComponent* interface. A factory class for the new component is instantiated by the registerer object and registered in the global registry. When constructing a component instance, the factory is accessed through the registry (via its abstract interface) and used to construct an instance (owned by the framework through its abstract interface).

of the interface). When a new component is constructed, its factory searches for matching dependencies in this registry and injects them into the objects constructor.

C. Component configuration

Each component instance may be individually configured. This configuration is loaded by the initialization code and used to customize each instance when constructed. Internally the configuration for each object is represented as a set of *ConfigEntry* objects. Each ConfigEntry is a template parametrized by data type stored by it. It has an abstract base class *AbstractConfigEntry*, which allows for aggregation of multiple entries together. Entries are kept in a *Config* object per component instance. Each entry is characterized by a key of type *std::string*.

Internal component code accesses its configuration with a dedicated member function inherited with the Component class:

```
template<typename T>
const T& config<T>(const std::string& key)
```

where *T* is the expected data type and *key* identifies a particular entry. Entries may be of type *std::string*, or an integers of any size. For the numeric entries the framework takes care of range checking and conversion operations. Figure 7 shows examplary usage and Figure 8 shows the described type hierarchy.

V. Data driven component instantiation

Component implementations are provided to an application through modules. Linking the corresponding library makes a component class available, but no instances have yet been created. The framework provides several mechanisms, intended to be executed at the startup phase to dynamically make use of the linked resources and compose a hierarchy of component instances. It is determined at this point which dependencies are injected into which components. All this happens according to a *Topology* object, consisting of the required data. This object

```
using diff;

class MyComponent :
            public Component<MyComponent> {
public:
 MyComponent() :
       buffer_(config<int>("bufSize"), 'x') {
      ...
    }

    void read(char *pBuffer) {
        std::memcpy(pBuffer,
                    &buffer_[0],
                    config<int>("chunkSize"));
    }

private:
    std::vector<char> buffer_;
};
```

Fig. 7. Component code accesses the user–config with config() member function inherited with the Component class. It can be invoked for a given type and key and returnd the corresponding value. Framework takes care of checking existence of the entry, its type and possible conversions.

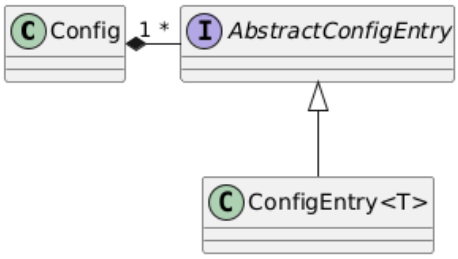

Fig. 8. Component config has flat structure. *ConfigEntry¡T¿* objects hold values, while being aggregated by a *Config* object. This is possible thanks to sharing a common abstract interface.

can be prepared in two ways - loaded from a Json file, or built with a sequence of calls to a *TopologyBuilder*.

A. Json defined topology

Topology loaded from a file is the most convenient way of working with an application, which requires dynamic reconfiguration, debugging, or a simple way of adding and removing features on the go. At the application startup the file is loaded and translated into a Topology object, which is then used to compose object hierarchy step-by-step.

The information stored in the file consists of a set of entries. Each entry represents a single component instance to be constructed. The information required by each consists of the type of object to be created and an ID to be assigned to it. The identifier can be further used to reference the instance when injecting it as a dependency for another objects. Additionally, a component with dependencies requires list of which objects shall be injected as those dependencies. Depending on the internal implementation, configuration is also be provided here. Entries in the file are processed sequentially, thus any instances used as dependencies shall occur prior the components depending on them. Figure 9 presents an exemplary Json file.

Figure 10 shows the resulting component topology. Figure 11 presents the component hierarchy used in the example.

```
[
    {
        "type" : "ThrottlePedal",
        "id" : "throttle_0",
        "config" : {
            "sensitivity" : "high"
        }
    },
    {
        "type" : "DieselEngine",
        "id" : "engine_0"
    },
    {
        "type" : "CruiseControl",
        "id" : "cruiseControl_0",
        "config" : {
            "maxSpeedKmph" : 140
        },
        "dependencies" : [
            "engine_0",
            "throttle_0"
        ]
    }
]
```

Fig. 9. This exemplary Json topology represents a cruise control system for a vehicle. The *CruiseControl* component has two dependencies - a **throttle** and an **engine**. These dependencies are satisfied by *ThrottlePedal* and *DieselEngine* components, which are injected according to the given identifiers. However, if the design consisted of complementary components *ThrottleLever* and *ElectricMotor*, they could be easily substituted, allowing to reuse *CruiseControl* implementation.

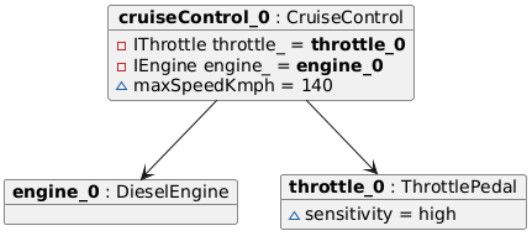

Fig. 10. Object topology resulting from the Json in Figure 9.

B. Programmatically defined topology

As much as Json file is convenient for working with, it may not be the best solution for deploying application for production with. Firstly, in production environment the application is not expected to require modifiable configuration. The resources may be limited, thus the usage of additional library for parsing Json documents may seem excessive. Also, if the system is really constrained (e.g. a bare–metal application) there may not even be a possibility to store and load files. Secondly, exposing configuration in such an explicit way may lead to security concerns. If the file was accessible for unauthorized access it could pose a risk of leaking the design details, breaking functionality, or even damaging the hardware.

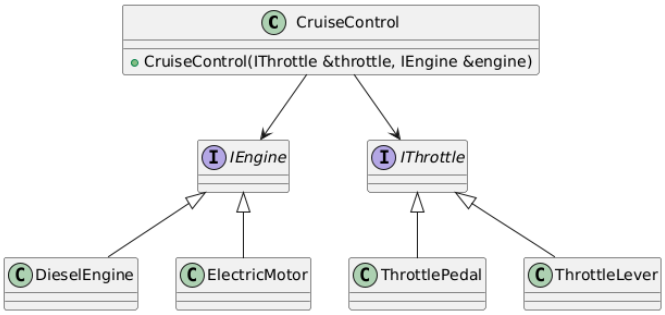

Fig. 11. Component hierarchy used in exemplary Json in Figure 9.

Anticipating for such scenarios, the framework offers a TopologyBuilder class, which can be used to construct Topology by executing a series of calls to its member functions. Figure 12 presents examplary code, constructing the same topology as the Json file presented in previous section. In fact, an arbitrary Json file can be automatically converted to the corresponding code with a dedicated tool. This saves development effort and allows for seamless integration with CI/CD.

```
using diff;

void load_topology(Topology &topology) {
    TopologyBuilder builder(topology);

    builder
        .component("ThrottlePedal",
                   "throttle_0")
        .config("sensitivity", "high");

    builder
        .component("DieselEngine"s,
                   "engine_0"s);

    builder
        .component("CruiseControl"s,
                   "cruiseControl_0"s)
        .config("maxSpeedKmph", 140)
        .dependency("engine_0"s)
        .dependency("throttle_0"s);
}
```

Fig. 12. The same topology as in Figure 9 defined programmatically.

VI. Exemplary applications

The described architecture has formed throughout several non–trivial, commercial projects. One key requirement was common to all of them - software reusability for further similar projects. In each case there was the first specific use–case, which could be satisfied by simply implementing a solution of moderate complexity. However, it was known upfront, that there will further use–cases - similar, but not identical - where simple configuration change would not be sufficient to fully cover the requirements. Moreover, it was required for the application to easily accommodate for functionalities yet to be implemented and not break, nor collide with, any existing features. Below are two examples where the proposed solution was proven successful.

A. Digital signal processing pipeline control

This automotive–grade application had to control a DSP pipeline composed of multiple DSP blocks. The pipeline was distributed across multiple processing units - general purpose CPUs and digital signal processors. Each customer required a slightly different topology of the processing chain and for each one the hardware executing the algorithms was different. This required two layers of configurability. Firstly, the set of DSP block, which the application communicated with had to be modifiable. Secondly, the communication method with subgroups of those blocks also had to be configurable.

B. Maritime hybrid propulsion integration

Another real world example where this concept was exercised is an application integrating multiple components of a hybrid maritime propulsion system. Such a system consists of a full–sized diesel engine, energy bank, electric motors with converters and additional appliances - pumps, sensors, protection devices, navigation equipment and control stand (possibly multiple). Most of such devices in such system are interconnected with multiple CAN buses, each possibly operating a different higher level protocol (e.g. CanOpen, J1939).
The set of devices used in in the project was relatively stable, with an option to add new device models from time to time. However, individual requirements of various customers resulted in different hardware configurations. A typical use–case was that an appliance (e.g. a pump or a sensor) could be plugged into different CAN bus, depending on its physical location and the routing of each bus. With the help of a Json file, such a requirement could be adapted for on the fly.

VII. Conclusions and further work

The proposed architecture, at first theoretical, later verified in real–world projects has a great potential to structurize software development in embedded engineering. It provides a guidance on how to implement reusable software modules and integrates the development process with modern methodologies (unit testing, CI/CD).

Further work on the project will consist of multiple additional features. The proposed framework already provides the capability for integration with modern tools, used throughout the industry, however, out-of-the-box integration with such would be another step forward. Another critical requirement is the ability to smoothly integrate with users CI/CD pipeline. For this reason some additional adjustments need to be implemented. Such include scripts supporting automatic recognition of modules required for linking and binary creation. A GUI tool can also be provided for topology creation (replacing writing the Json file manually).
One of the critical aspects of using any tool for embedded systems is their ability to not use dynamic memory allocation.

This is a crucial requirement of many safety sensitive applications. For this reason a set of custom allocators needs to be implemented to replace the ones provided by the standard library. Another aspect worth taking care of is configurability. While simple configuration may seem enough for basic use–cases, real world scenarios may require more complex, hierarchical component configuration.

Several of the mentioned features are already being worked on, others are planned. The proposed concept shows potential for improving embedded software development in multiple aspects.

REFERENCES

[1] C. A. Mack, "Fifty years of moore's law," *IEEE Transactions on semiconductor manufacturing*, vol. 24, no. 2, pp. 202–207, 2011. [Online]. Available: https://doi.org/10.1109/tsm.2010.2096437

[2] P. C. Pendharkar and J. A. Rodger, "The relationship between software development team size and software development cost," *Communications of the ACM*, vol. 52, no. 1, pp. 141–144, 2009.

[3] B. B. Brown, "Over-the-air (ota) updates in embedded microcontroller applications: Design trade-offs and lessons learned," *Analog Dialogue Technical Journal*, vol. 52, pp. 52–11, 2018.

[4] E. Razina and D. S. Janzen, "Effects of dependency injection on maintainability," in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, 2007, p. 7.

[5] P. Koopman, "Embedded system design issues (the rest of the story)," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*. IEEE, 1996, pp. 310–317. [Online]. Available: https://doi.org/10.1109/iccd.1996.563572

[6] D. Ajiga, P. A. Okeleke, S. O. Folorunsho, and C. Ezeigweneme, "Methodologies for developing scalable software frameworks that support growing business needs," *Int. J. Manag. Entrep. Res*, vol. 6, no. 8, pp. 2661–2683, 2024.

[7] D. Prasanna, *Dependency injection: design patterns using spring and guice*. Simon and Schuster, 2009.

[8] J. Helander and A. Forin, "Mmlite: A highly componentized system architecture," in *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, 1998, pp. 96–103. [Online]. Available: https://doi.org/10.1145/319195.319210

[9] K.-K. Lau, "Software component models," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 1081–1082. [Online]. Available: https://doi.org/10.1145/1134285.1134516

[10] K. Laemmermann, "C++ the design and evolution of c++," 2012.

[11] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, "Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 471–482. [Online]. Available: https://doi.org/10.1109/icsme52107.2021.00048

[12] K. Driesen and U. Hölzle, "The direct cost of virtual function calls in c++," in *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1996, pp. 306–323. [Online]. Available: https://doi.org/10.1145/236337.236369