Flow-Design and Event-Based Components: A New Approach in Designing Software Compared to Established Procedures

TOBIAS GEGNER
University of Applied Sciences Wuerzburg-Schweinfurt

Flow-Design is a new approach in developing software architectures. Flow-Design focuses on data processing function units and the communication among them. Event-Based Components represent a programming language independent methodology for implementing Flow-Design and can also be considered as a resulting software architecture. The Event-Based Components software architecture is structured by components that communicate via events. Therefore it is compared to the concept of service-oriented architecture and event-driven architecture which aim to create independent components as well. In conclusion Flow-Design and Event-Based Components provide a platform and programming language independent concept to develop software components. The resulting components are reusable, maintainable and testable. Although it can lead to a more complex coding.

Kategorie und Themenbeschreibung: Software Engineering

Object oriented programming, Software architecture, Software design, Software reusability, Software engineering.

1 INTRODUCTION

Software development has ever since strived for re-usability and maintainability. Manufacturing industry with standards across companies and national borders has reached a high level in reusing and composing parts built by different suppliers. Serving as a role model, manufacturing industry motivated computer scientists explore ways of mapping industry procedures to software development. Therefore in the past years different approaches arisen to develop software components of a high level of reusability. Within this evolution several approaches evolved using events to couple components and arrange communication among them. These event-based architectures are widely discussed and find appliance in distributed software systems as well as within sole software systems or components e.g. [Cugola et al., 2001, Eugster et al., 2003, Carzaniga et al., 1998, Fiadeiro and Lopes, 2006, Kowalewski et al., 2008].

Following the idea of using events to make software components more independent of their environment Ralf Westphal invented Flow-Design and Event-Based Components. By using design patterns and best practices he evolved the way of designing and implementing software. In the following this paper will introduce Event-Based Components to the reader. By examining a running example, the fundamental concepts of Flow-Design will be discussed, as well as differences to object-oriented analysis and design. Further the implementation of Flow-Design with Event-Based Components will be described. Subsequently a comparison of Flow-Design and Event-Based Components to established design processes and software architectures will be given. In the end a conclusion is given as well as a future outlook.

Author's address: Tobias Gegner, University of Applied Sciences Wuerzburg-Schweinfurt, Faculty of Computer Science and Business Information Systems, Sanderheinrichsleitenweg 20, 97074 Würzburg, Germany, www.fhws.de

2 INTRODUCING FLOW-DESIGN AND EVENT-BASED COMPONENTS

The concept of Event-Based Components (EBC) was mainly introduced by Ralf Westphal, who started to post articles in his German and English speaking blogs [Westphal, 2010b, Westphal, 2010d] in the year 2010. He also wrote articles about EBC in professional journals which will also be referenced in this paper. EBC is a new conceptual way to handle data communication between individual software components. It represents the code implementation of Flow-Design (FD), which is a different approach in designing software architectures. As the name Flow-Design already suggest it's about flows. More concrete, FD describes function units and data that flows among them to fulfill business logic. Ralf Westphal evolved FD and the Flow-Orientation on the basis of Flow-Based Programming and Event-Based Programming concepts which are discussed in computer science for quite a few years [Morrison, 2010, Faison, 2006]. His work was also influenced by Business Process Modeling [Freund and Rücker, 2012] and the Event-Driven Architecture [Bruns, Ralf and Dunkel, Jürgen, 2010]. Enriched with own thoughts he brought up a new approach of designing software: Flow-Design.

In this paper the terms component and flow will play a major role. Therefore based on Ralf Westphals usage of the terms flow and component, these two terms will be defined as follows:

Definition 1 (component). *A component is a modular part of a software. It encapsulates a set of related functions or a single functionality, optionally its data. Further it is replaceable within its environment. Interfaces describe required and provided services. Within this paper the terms component and function unit are used synonymously.*

Definition 2 (flow). *A flow represents communication among components of a software. It describes handing over control of data processing between components as well as hereby exchanged data. As an intellectual concept flows are depicted in diagrams.*

2.1 Running Example

To make the difference of the FD approach clearer a sample application is examined. A command line tool has to be developed, that reformats text from a given file. It reads text from a file, rearranges the line breaks to a given line length and writes the resulting text into a new file. To start the application from command line, three parameters have to be provided by the user. First parameter names the file that contains the text. Second parameter defines the maximum line length to which the text is reformatted. Third parameter names the target file into which the result hast to be written. The application must not change nor apply hyphenation in its first version, this is left to future versions. The application has to guarantee, that the maximum line length is not exceeded nor that fewer words than possible are arranged in one line. Punctuations have to be considered as part of the preceding word and therefore must not be changed in any way [Westphal, 2011d].

2.2 Flow-Design

Designing software with the object-oriented analysis and design (OOAD) approach concentrates on building units of data with associated operations. That way the problem is

analyzed for substantives which become objects in the code representation of the solution. Flow-Design turns this the other way round. The problem is analyzed for verbs which become operations (function units). FD concentrates on what should be done and then looks for the data that needs to float between the function units. Seen that way, FD is closer to the Input-Process-Output (IPO) approach than to OOAD. It centres around the processes that need to be done to solve a certain problem, representing the P in IPO. Therefore the data is represented by I and O in IPO. The customer has data (input) and wants it to be transformed (process) into another form or representation (output) [Westphal, 2010b, Westphal, 2011a].

2.2.1 How to Apply Flow-Design

Following the OOAD approach a possible solution to the running example (2.1) might come up with a couple of single objects generated from the substantives of the task: File ; Text ; Line. Then the discovered data objects are filled with methods that implement the business logic. The object-oriented solution would also be well documented with a UML 2.0 class diagram [Booch et al., 2006]. A simplified diagram of the OOAD solution can be seen in figure 1. The outcome of this approach is a static diagram which gives an overview of the data model but has no information about the work flow.

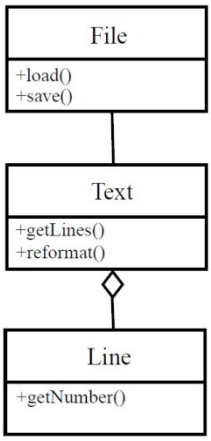

Figure 1. Simplified UML class diagram of the OOAD solution for the Running Example (2.1). Adapted from [Westphal, 2011d]

In contrary, solving Running Example (2.1) with FD would start by looking at the operations that need to be developed. Starting with a rougher first draft of the solution a Flow-Design diagram is denoted that shows the very basic operations represented as function units. In further cycles the solution would be more and more refined. By using FD a possible solution for the problem might identify the following basic function units: *read text from file*; *reformat text*; *write text to file*. Next development cycle the function unit reformat text might be refined to these function units: *fragment to words*; *aggregate to line*; *aggregate to text*. In figure 2 the Flow-Design diagram of the FD solution can be seen.

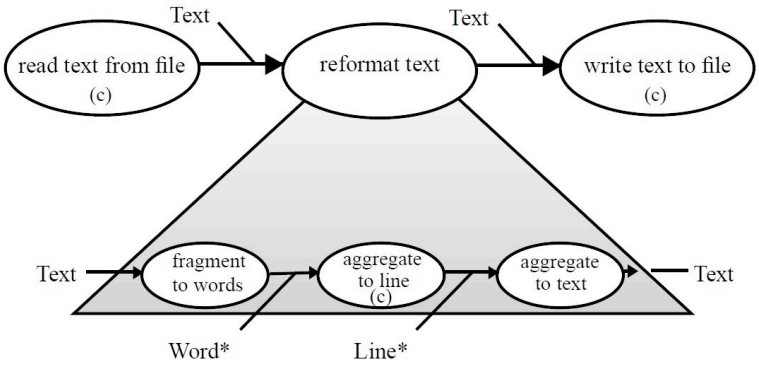

Figure 2: Flow-Design diagram of the FD solution for the Running Example (2.1). Asterisks imply multiplicity one-to-many. Adapted from [Westphal, 2011d]

2.2.2 Flow-Design Diagrams

As Flow-Design introduces a new way of analyzing and designing software and software architectures it also brings a notation with it to depict the design process which was stated by Ralf Westphal as well as FD itself [Westphal, 2010d]. The Running Example (2.1) and its suggested FD solution illustrate how FD follows the process of the solution naturally and depicts the data flow between the function units with FD diagrams. Function units are depicted as circles or ellipses with their name in it. Flows, connecting the function units, are called wires. Therefore connecting function units is called wiring and is denoted with an arrow from the output of a function unit to the input of another function unit. Data only occurs on flows and is denoted with its type and multiplicity. As the FD diagram shows the work flow of the solution one can see how the solution works on different levels of abstraction. It's also possible to variegate the level of detail by switching the layer of abstraction in the diagram to get a deeper view of the solutions details. This way the customer or a new team member can be inducted to the current state of the solution quickly without having to know details about the implementation.

2.2.3 How Flow-Design Structures and Connects Software Components

Going deeper in the level of abstraction, function units have to describe more concrete what they do. Applying this Top-Down approach, FD follows the concept of functional decomposition [Westphal, 2011d]. This way FD enables one to take a look at the solution on different levels of abstraction, giving more or less details about the functionality taking respect of what is needed in the current situation. Taking a closer look at this approach one will see, that in FD there are two different types of function units. FD builds up a hierarchical structure in form of a tree of function units starting at the root, which represents the whole application. Going down the tree there are nodes with further children and leaves at the end of the tree. At the leaves function units implement logic as well as data processing is provided. On the contrary nodes group leaves to form units of a higher level of abstraction. These function units arrange the data flow among the grouped leaves. As seen in the FD solution for the Running Example (2.1), the function unit

reformat text represents a node and decomposes to the three leaves fragment to words, aggregate to line and aggregate to text. This way nodes offer a more cohesive functionality for units on a higher level of abstraction. By distinguishing between nodes and leaves FD follows the concept of Separation of Concerns [Reade, 1989]. The two different kinds of function units in FD handle two different concerns: implementing functionality and aggregating functionality. However the two kinds of function units are depicted with the same symbol in FD diagrams. According to electronic engineering Ralf Westphal named the aggregating function units Boards and the aggregated function units, that implement the logic Parts [Westphal, 2010c]. In figure 3 an example is given of the tree structure with Boards and Parts assembling to the whole application.

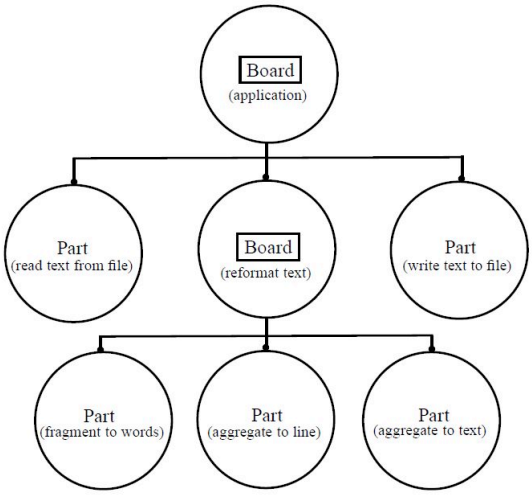

Figure 3: Tree showing the hierarchical structure of FD and two different types of function units. Adapted from [Westphal, 2010c]

By designing Boards and Parts with FD a high granularity can be accomplished and Parts can be build serving just one single task. This way dependency among function units is avoided as well as the Single Responsibility Principle [Martin, 2003] is fulfilled. Parts are independent from functionality of other Parts or Boards due to the fact, that function units only process incoming data and provide the processed data, following the IPO approach. There are no implicit dependencies in form of calls to other FD function units that would break the data flow. While Boards are implicit depended of the Parts they integrate. This implicit dependency is not critical because of the simplicity of Boards. Boards only concern about putting Parts together and should have no business logic implemented.

Beside implicit dependency it is possible that Parts are dependent of existing components or APIs (Application Programming Interface) that provide needed legacy functionality. This is a special form of dependency and should be explicit depicted in FD diagrams [Westphal, 2011d]. To integrate legacy components or APIs in FD and show the explicit dependency the API or components are wrapped into a separate function unit, serving as adapter for FD. This separate function unit takes the needed data for the API or component as input and delivers the result as output data, forming a data flow in FD style. This way legacy components or APIs can be fully integrated into the FD process.

2.2.4 Special Features of Flow-Design and Flow-Design Diagrams

Flow-Design diagrams offer similar ways than UML 2.0 diagrams [Booch et al., 2006] to depict characteristic features in the software design. Later these features will be translated into EBC and therefore executable object-oriented code.

Configurable: As well as in OOAD there is the possibility that function units have to be configured preliminary before they are able to fulfill their function. In Running Example (2.1) that would be read text from file which needs the source file name, aggregate to line which needs the maximum line length and write text to file which needs the target file name. In FD diagrams configurable function units are annotated with a '(C)', as seen in figure 2. The configuration of these function units has to be done before they process any data in the flow. Implementation of the configuration is left to EBC [Westphal, 2010d].

Entry point: To mark where the whole flow and therefore the data processing starts an entry point annotation is available. The function unit which is processed first when the software starts up is annotated with a '(E)'. This is likewise to UML 2.0 activity diagrams [Booch et al., 2006], where the start point of the process is also marked. The implementation details are left to EBC as well [Westphal, 2010d].

Singleton: Since function units are translated into object-oriented code they are multiple instantiated by default. So referring to function units with the same name in a flow means to refer different instances of the same function unit. In example referring to a function unit at the beginning and at the end of a flow would mean pointing to two different instances of the same function unit. This occurs because, as declared previously, function units are independent of their environment and therefore have no knowledge of existing instances of other function units. In case it is needed to reuse the same instance of a function unit at different points of the flow, the Singleton design pattern [Gamma, 1999] can be applied. This has to be denoted in the FD diagram by annotating the function unit with a '(S)' [Westphal, 2010d].

Split and Map: Function units are independent, provide their result to unknown consumers and are connected via flows by Boards. Though it is possible to connect a function unit not only with one but multiple other function units. This is achieved by wiring the output of a producing function unit (P) with the input of every consuming function unit (C). This is called a Split and can be depicted in FD diagrams by just denoting a wire from function unit P to every function unit C. In that case, the same data flows on every wire. Yet the flow is still synchronous and the different branches are not processed in any specific way. Matters of concurrency are left to EBC and its implementation. Letting only Parts of the original data flow pass to connected function units is called a Map. This can be depicted explicitly or implicitly in FD diagrams. Explicit a box is painted into the branch of the flow where the Map is installed. Implicit the data type of the affected branch differs from the original flow. Maps may be applied when a consuming function unit handles only a subset of the flows data at its input [Westphal, 2010d].

Join: While it is possible to create function units with more than one input it may be needed to combine different branches of input flows into one. Especially when all data from the input flows is needed to proceed, a Join should be installed. A Join takes multiple input flows and handles over a tuple of the flow data to its function unit. Flows are considered as separate standard Parts. They are depicted as a vertical line in front of the function units input [Westphal, 2010d].

2.3 Event-Based Components

Event-Based Components are the implementation of Flow-Design architectures and aim at the goal of creating composable function units or components. Designing an application with the FD

approach leads to Boards and Parts, independent function units. These function units work without knowledge of their environment and are therefore independent. In FD diagrams function units are connected via data flows which represent the communication between different Boards and Parts. At that point the function units are composable, at least in terms of FD diagrams. EBC describe how to translate FD in object-oriented code, so that the implemented function units and components are composable and independent as well.

2.3.1 How to Apply Event-Based Components

The basic rules of translating Flow-Design diagrams into runnable object-oriented code are:

- Every function unit, whether Board or Part, becomes a class.
- Every input flow becomes an event-handler method of a class.
- Every output flow becomes an event-sender method of a class.

Following these rules, translating FD diagrams into code is pretty much straight forward. Though the connections between independent Parts have to be created. As well as the assembling of Parts to Boards needs to be solved. To achieve this and preserve the flexibility of FD, best practices and design patterns are applied. For readability of coding, it is helpful to prefix methods for input and output pins accordingly with ‘in ’ and ‘out ’ [Westphal, 2010c].

2.3.2 Issues of Component-Based Software Engineering

If we take a look at Component-Based Software Engineering [Heineman and Councill, 2001] we see that composing different components can be achieved through extracting interfaces from components. The extracted interfaces represent the service contract for the implementing component. Therefore the client components are independent from a concrete implementation of the service interface. Breaking with static dependencies from interface implementations at compile time can be accomplished by making use of the Dependency Injection design pattern [Fowler, 2004]. Thus client components can be bound dynamically at runtime to a concrete service interface implementation. Hereby the client component gets more independent of its environment because it neither has to have knowledge of the implementation of the service interface nor it has to be able to instantiate a concrete implementation of the service interface. Yet a client component is still dependent of the service interface it makes use of. Since interfaces often define more than only one single method and may change over time the Single Responsibility Principle is broken. A client component would not only have to change if the signature of a used method of the interface changes but also if the interface gets decomposed into smaller distinct interfaces. Here the topological dependency of components from their environment shows up. Ralf Westphal named these components Injection-Based Components (IBC) due to the fact, that IBC are injected as whole component. This already points to where EBC are going. EBC resolve topological dependencies and leave components with functional dependencies only, which are uncritical and integral part of client-server relationships [Westphal, 2010e, Westphal, 2010a].

2.3.3 Connecting Function Units with Event-Based Components

Event-Based Components connect different components by using events and event-handlers. A client component that needs a certain service fires an event. A subscribed event-handler of a

service component receives the service request via the fired event and processes the transmitted data. When the service component finished processing the data it fires an event itself and publishes the result that way. That is the basic procedure that maps data flow from FD to EBC. Depending on the used programming language for implementation, different techniques can be used for realizing the events and event-handling. As for example in C# event-handling is an integral part of the programming language and the .Net Framework[11] . Therefore delegates can easily be used to implement events. Whereas the Java Runtime[12] does not support events by default. Here an event-handler has to be implemented manually, for example by implementing the Observer design pattern [Gamma, 1999].

To label the events and create a common code representation for flows, a generic event interface is being created. Both, client and service, use this common event interface to typecast their event data. Depending on the used programming language, solely one interface is needed for the event (e.g. C#) or an interface for event-handler and event-sender is needed to accomplish the same functionality. Except the higher effort in implementation there is no disadvantage for the latter approach and therefore it is not distinct in this paper. For example an event-handler interface in Java would look like:

```
interface IEventHandler<T>{ handle(T event) }
```

That way type safety and flexibility is provided either. Alternatively, if type safety is dispensable or the programming language of choice does not support generics, all events can implement a simple marker interface without a generic type. Through this shared event interface the communication between all components is arranged. Clients send events of a certain type and services provide event-handler methods for that certain event type. With this event based communication between components a client neither gets dependent of a concrete service implementation nor of an aggregated service interface. The client just gets connected to a single service method respectively its interface. Although it looks like the client gets dependent of a certain event or event-handling method, in fact it is the Board which is dependent of the Parts it connects. As mentioned previously, that kind of dependency is nothing to bother about. That way the client component gets topologically independent of the service and the service interface. Standard connectors, likely to pins of electronic parts in electronic engineering, are established through this approach. Due to the similarity to electronic engineering parts, event-sender and event-handler methods are called input pins and output pins. Thus Parts provide certain channels for in and outgoing messages in form of input and output pins. Boards aggregate Parts and other Boards to new units of a higher level of abstraction. Wiring the input and output pins of the included Boards and Parts together to enable the flow means to register event-handler at the corresponding event-sender. Externally the aggregating Board presents input and output pins which are connected to the internally starting and ending point of the flow respectively their pins. Taking a look at Running Example (2.1) the Board reformat text provides an input pin for data of type Text and an output pin of the same data type. Diving into deeper, it is shown that the Boards input pin is connected to the input pin of the Part fragment to words. Connecting input pins of a Board to input pins of an included function unit is accomplished by delegating the incoming event and its parameters to the according event-handler method of the included Board or Part which is in charge of the event-handling. The output pin of fragment to words is wired to aggregate to line which is further wired to aggregate to text. The output pin of the Part aggregate to text is wired to the output pin of the Board by passing through the Boards parameter to the Part. At the beginning and at the end of the internal flow of reformat text is the data type Text. Within the flow itself there are different data types flowing between the internally wired Parts. It is essentially important, to go

[11] Based on .Net Framework 4 published on 21st of February 2011.

[12] Based on Java SE 7 published on 28th of July 2011.

strict with the FD design specifications while implementing EBC. This means, that Boards must not implement any business logic but wiring included Parts and Boards. Otherwise design and implementation drift apart and hidden dependencies may occur. EBC are abstracted through interfaces as well and the previously mentioned Dependency Injection design pattern is used to hand over function units and to aggregate Parts to Boards. Due to the abstraction via interfaces Boards are free of dependencies of concrete implementations of Boards or Parts.

2.3.4 Special Features Implemented with Event-Based Components

As mentioned in 2.2.4, there are several characteristics in software design with FD. The following will depict how to implement these features with common object-oriented techniques. Moreover further concepts and their implementation with EBC are explained. Concrete details how each feature has to be exactly implemented with EBC is depending on the programming language of choice. Therefore this is left up to the implementer.

Entry point: The flow starts, where the entry point in the FD diagram was denoted. Implementing the entry point with EBC is solved with a marker interface which defines the start up method for the flow. Depending on the chosen programming language, the start up method for the application varies. However it is common for start up methods to take a collection of string arguments. Therefore the entry point interface in EBC defines a method named ‘run’ with a parameter typed as string collection as well. For example an entry point interface in Java would look like:

```
interface IEntrypoint{ void run(string[] args)}
```

At application start up the ‘run’ method of the entry point has to be called and the start arguments have to be passed to the method. Acting as the start of the flow, the entry point interface has to be implemented by the Board which aggregates the function units to the whole application. This is at root of the function unit hierarchy tree [Westphal, 2010d].

Configurable: Configurable function units need to be configured at start up at. Yet at least before the take part of the flow. Similar to the entry point, that takes start up arguments, configurable Parts implement a marker interface. This interface defines a method ‘configure’ with a parameter typed as string collection like the entry point interface does. Thereby the interface implementing Part receives the needed parameters to configure itself. In Running Example (2.1) it is required that three parameters are provided at start of the application to define the reformatting details of the text. By implementing the FD diagram, seen in figure 2, with EBC the entry point would get the three parameters at start up and assign them to the according configurable Parts [Westphal, 2010d].

Singleton: If a function unit is only to be instantiated once, the Singleton design pattern [Gamma, 1999] has to be applied. Implementing a singleton of a FD diagram has to be accomplished with instruments of the chosen programming language. Ensuring that only one instance can be created during the program run is mostly achieved through a private class constructor. Only the class itself is able to create a new instance. Though a public static method has to be provided that creates one single instance of the class and provides it to the outside. That way it is guaranteed that, within the same or different flows, every time a singleton function unit is wired it is always the same instance of the function unit [Westphal, 2010d].

Split and Map: Connecting more than one input pin to one output pin is called a split. Data doesn't flow parallel on all branches, but sequential in undefined order. This can be achieved by wiring multiple function units to the output pin. Implementing with EBC that means assigning multiple event-handler to the event-sender. If not all of the data has to flow on all branches of the split flow, a map, that filters data, is needed. To implement a map with EBC a separate class has to be constructed serving as a standard Parts. It takes in the original data and outputs particular Parts of that data filtered by certain criteria. When needed a map is put between two function units while wiring them. So the data flowing from one function unit to the other gets filtered by the map. Summing up, a map helps to connect two function units where the receiving function unit is only able to process a subset of the output data. Therefore a map can also be seen as an adapter [Westphal, 2010d].

Join: A join depicted in a FD diagram aggregates two output pins to one input pin. To implement a join with EBC a separate standard Part, similar to a map, needs to be created. This extra class provides two input pins and one output pin. The provided output pin can either have two parameters or one tuple sticking the two parameters together. By default a join generates an output value whenever a new input is received. Nevertheless for the first output the join class needs all inputs. It is also possible to create joins of different behaviors. Therefore a join can reset all inputs after every output or only produce an output if input on a particular pin is received. Coding details are left to the implementation with the chosen programming language. Installing a join is similar to installing a map. When needed the join is put between the participating function units. The output pins to join are wired with the input pins of the join Part. Then the output pin of the join is wired to the input pin of the receiving function unit [Westphal, 2010d].

Concurrency: Implementing concurrency with EBC can be achieved by using splits and joins. With splits and joins flows can be segmented into multiple branches and brought together as well. As mentioned previously a split is not parallel by default. Therefore available concurrency techniques of the chosen programming language, like threading, threadpools and semaphores, have to be used. This way split flows can be processed in separate threads. With joins the parallelized flows can be back to one thread. In Running Example (2.1) the reformatted text could not only be written into a file but also be printed on the command line as well as directly sent to a printer. Or certain data of the flow could be logged with a logging tool. These further steps could be processed parallel to the original flow and therefore not disturb a user's work flow. As with all parallel processing, concurrency in EBC demands attention during design and implementation to avoid race conditions and dead locks [Westphal, 2010d].

Explicit dependencies: As explicit dependencies of legacy components or APIs can appear in FD, this has also to be handled during implementation with EBC. To make dependencies explicit a generic marker interface is used. This interface defines one method to inject the class of which the implementing function unit is dependent of. The method named 'inject' has one parameter: the required class. Through this interface the dependency is visible in the coding. The dependency is handled at one certain point in the coding. This makes the implementation more maintainable and the coding more readable. By using a method to inject the required class it is independent of object creation. Therefore dependency injection is possible at any time during start-up of the application [Westphal, 2010d].

Exception handling: As most object-oriented programming languages share the principle of exceptions this is also an issue in EBC. Exceptions which can or should not be handled within a function unit are published. Often it is desirable to have a single part in the software where exceptions are handled. Therefore exceptions in function units are published via an extra output pin. Depending on the programming language of choice this output pin can be parameterized with an universal exception type or left untyped.

2.3.5 Tooling

EBC are growing in popularity. Resulting in a community that discusses Flow-Design and Event-Based Components on internet forums. Basic Concepts are discussed as well as possibilities to use tools for EBC development. Meanwhile miscellaneous tools for EBC have been developed to facilitate designing applications with FD and developing with EBC. These tools intend to make routine activities easier for developers or even release them. A graphical presentation of FD should be enabled. And the resulting code from EBC should become unified and therefore gain quality and readability [Westphal, 2011b].

To counter the probably most confusing part of EBC coding, the wiring of components, Ralf Westphal and Stefan Lieser introduced a Domain Specific Language (DSL) for EBC. With the help of this XML based DSL, which is therefore named ebc.xml, wiring Boards and Parts should become more easy. Through ebc.xml aggregating Boards and Parts to lager, more abstract function units is separated from coding. Though it is also separated from business logic which is found within Parts. By using the XML-File ebc.xml no wiring is left to the code but all defined in well-structured and defined XML files. Following the idea of Separation of Concerns [Fowler, 2003] ebc.xml separates implementations details and business logic from structuring components. Being independent of a certain platform or programming language is another advantage of ebc.xml. Finally the code according to the target programming language is generated with an ebc.xml -compiler. The resulting code aggregates Parts to Boards according to the XML file [Westphal, 2011c].

The basis for a common visualizing and wiring of Boards and Parts was built through ebc.xml. Therefore besides the ebc.xml -compiler an ebc.xml -visualizer exists too. This visualizer is able to generate a graphical representation of the wiring defined with ebc.xml files. Hence besides unifying EBC code a unified graphical presentation of EBC is provided with a tool. A better understanding of EBC is also accomplished as well as a clear presentation of wiring and therefore the flow between function units [Westphal, 2011c].

From the EBC developer community an ebc.xml -designer evolved. Mike Bild developed an ebc.xml -designer that enables graphical development of FD and EBC. Boards and Parts are standard units which can be named and combined on the work space. Wires and input respectively output pins are used to connect components graphically. Data flowing on the wires between components can be typed. Eventually the complete design is translated into ebc.xml DSL and is further processed by the ebc.xml -compiler [Westphal, 2011c].

Under the project ‘Event-Based Components Tooling’ [Westphal and Lieser, 2011] Ralf Westphal and Stefan Lieser started an open source project. The projects purpose is to provide tools and standard components for EBC like joins and splits. These pre-assembled components lighten and fasten developing software with EBC. Using standard components the resulting code gets more robust gains readability. These EBC tools are available as compiled programming libraries. Also the source code is available through the projects website. At this point, EBC tools are only available for the programming language C#.

3 CONSIDERATION OF FLOW-DESIGN AND EVENT-BASED COMPONENTS IN CONTEXT OF SOFTWARE ARCHITECTURES

Flow-Design introduced a new way of engineering software architectures while Event-Based Components specified the rules of implementation. Next, established software architectures are considered in contrast. In which way FD and EBC differ from event-driven architecture, service-oriented architecture and object-oriented analysis and design is discussed in the following.

3.1 Separating Design and Implementation

Flow-Design is an approach to design software architectures. Therefore it is compared to object-oriented analysis and design. By use of FD approach new architectures respectively possibilities to design architectures arise. Diagrams, like UML for OOAD, are provided to support and document the design stage. Where OOAD puts objects and therefore data into foreground, FD focus on function units which process data. The exchange of input and output data between components is called flow. OOAD maps reality into software by using objects. In doing so, objects are units of data and operations. They encapsulate knowledge and therefore data. Operations represent a particular behavior. Though a software system is built of and described by cooperating objects. The relationships between these objects causes a dependency among them. When changing or maintaining an object-oriented programming (OOP) implementation, this dependency comes to relevance. If the signature of an objects method changes, changes may have to be done in many parts of the software to react on the change. Dependent objects have to be adjusted to regain the systems operability. Implementations of complex software systems result in vast amount of objects. Relations and dependencies between them is hard to see in coding an often not well documented in diagrams. Therefore it is often unclear how much a maintenance or change will affect the software system. Documentation is often not up to date with coding and is itself of high complexity. A OOAD designed software can be described with 13 UML 2.0 diagrams.

Component-based software engineering (CBES) focuses on the idea of reusability. Based on OOAD, CBES targets on creating bigger, more abstract elements. These components are meant to be reused without knowing any details of their implementation. By a high grade of reusability and therefore a higher number of users, components are thought to be more robust. Over time it is supposed to create a portfolio of standard components which can be reassembled and reused in following software projects. These components have fixed contracts, represented through interfaces describing exactly the functional range. Distributed components create the basis for service-oriented and event-driven architectures.

EBC is a methodology for implementing FD as well as an architecture similar to event-driven architecture (EDA) and service-oriented architecture (SOA). It is independent of a particular programming language or platform. Although it builds up rules and specifications for implementation. OOP is the programming paradigm used for implementing EBC, because events are part of many OOP languages, but at least can be implemented through design patterns. Though, unlike EDA and SOA exchanging messages on system level between independent software, EBC is by default an architecture limited to the boundaries of a software.

3.2 Similarities and Differences to Established Software Architectures

SOA builds up on the concept of CBES. CBES manages components via a repository to be reused during software development. In contrary SOA installs components as independent services. Instead of administrating components with a repository, the service components are managed by a service broker within a service catalogue. Service provider publish their provided services to the service broker and service consumer place their service request to the service broker. Message exchange is managed by a service-mediation system, also called enterprise service bus. Communication is synchronously, bidirectional and message oriented. SOA enables communication across independent software as well as using existing services during software development [Masak, 2007, Reussner and Hasselbring, 2009].

EDA uses events to exchanges messages. EDA is not about administrating components or services, but rather to manage the event flows. Events arise from state changes and are triggered by a source to be processed by a sink. Source and sink are independent from each other.

Components, software systems but also hardware like sensors can be source of an event. Via an event channel the events reach the middleware that distributes events to registered sinks. Event receivers, the sinks, have to register themselves with the middleware for certain events. Communication in EDA is asynchronously, unidirectional and event oriented [Bruns, Ralf and Dunkel, Jürgen, 2010].

EBC has commonalities with SOA and EDA. Similar to SOA, EBC builds up components to fulfill a certain service. This service is defined by a contract represented through an interface. Hence EBC is not about building service components providing several services like SOA. EBC creates components to provide one certain, cohesive functionality. This has the effect of loose coupling and minor dependencies. Here lies the analogy to EDA. Where EDA operates across system boundaries, EBC is limited to a single software system. Therefore in contrary to EDA, EBC doesn't need a middleware to transport events from source to sink. EBC solves this via components called Boards. These aggregate the smallest function units, Parts, to Boards. Wiring source and sink, called output and input pins, is arranged by Boards enabling the events to flow between function units. Similar to EDA the communication is unidirectional and event oriented. Although communication is by default synchronously the flow can be made asynchronously and concurrency with instruments of the programming language chosen for implementation [Bruns, Ralf and Dunkel, Jürgen, 2010, Masak, 2007, Reussner and Hasselbring, 2009].

4 CONCLUSION

Flow-Design changes the view in designing software. In contrary to OOAD, FD concentrates on processes respectively function units. It solves dependency and coupling issues by generating a design of separate components which are connected via flows and therefore are independent of each other. Independent components build with FD are more easily to test because they don't make calls to other components. Instead they take input and deliver output. Though no mock-ups are needed for testing. The function units can be tested separately without influence of other components. With Event-Based Components it is possible to translate the design into code and keep design and implementation aligned. Teams and developers can implement the design self-responsible. The internal logic of the function units is not part of the design, this is left to the implementers. FD defines the flow and the contract between components, not the internal structure. The communication between components is accomplished through events. Therefore it is unidirectional, the message flow takes only effect in one direction. Common client-server practices in which the client calls for a service and gets the response are not part of FD and EBC. Therefore this new approach requires software designers and developers to adapt to a new thinking. Aggregating Parts to Boards by wiring input and output pins, means creating components of a higher abstract level. The more function units are aggregated, the more complex the wiring gets. Hence the resulting coding of EBC can be complex and unclear. But tools for designing and the implementation for EBC are already developed. Although C# is the only tool supported programming language at the present time and Ralf Westphals coding examples are all presented in C#.

Ralf Westphal keeps on working on FD and EBC as well as he spreads his new way of thinking in software development. Tending towards automated creation of EBC out of FD models Ralf Westphal introduced the Flow Execution Engine [Westphal, 2010d] which generates Boards and Parts from a design. Afterwards the Parts need to be filled with logic. This relieves developers from wiring Parts to Boards. Hereby he also developed a runtime which is able to execute flows. Although up to now limited to German speaking areas, supported by a growing community FD and EBC gain popularity and attention.

REFERENZEN

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 2006. Das UML-Benutzerhandbuch: Aktuell zur Version 2.0. *Addison-Wesley*, München [u.a.].

BRUNS, RALF AND DUNKEL, JÜRGEN 2010. Event-Driven Architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse. *Springer-Verlag*, Berlin and Heidelberg.

CARZANIGA, A., DI NITTO, E., ROSENBLUM, D. S., AND WOLF, A. L.1998. Issues in Supporting Event-Based Architectural Styles. *Proceedings of the third international workshop on Softwarearchitecture ISAW 98*, pages 17–20. Available: http://portal.acm.org/citation.cfm?doid=288408.288413

CUGOLA, G., DI NITTO, E., AND FUGGETTA, A. 2001. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850.

EUGSTER, P. T., FELBER, P. A., GUERRAOUI, R., AND KERMARREC, A.-M. 2003. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131.

FAISON, E. W. 2006. Event-Based Programming: Taking events to the limit. *Apress and Springer*, Berkeley and Calif. and New York.

FIADEIRO, J. L. AND LOPES, A. 2006. A formal approach to event-based architectures. In Baresi, L. and Heckel, R., editors, *Fundamental Approaches to Software Engineering*, pages 18–32. *Springer-Verlag*, Berlin and Heidelberg.

FOWLER, M. 2003. Patterns of enterprise application architecture. *Addison-Wesley*, Boston.

FOWLER, M. 2004. Inversion of Control Containers and the Dependency Injection pattern. *Available: http://www.martinfowler.com/articles/injection.html*

FREUND, J. AND RÜCKER, B. 2012. Praxishandbuch BPMN 2.0. *Hanser, Carl*, München, 3 edition.

GAMMA, E. 1999. Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*, Reading and Mass, 17 edition.

HEINEMAN, G. T. AND COUNCILL, W. T. 2001. Component-based software engineering: Putting the pieces together. *Addison-Wesley*, Boston.

KOWALEWSKI, B., BUBAK, M., AND BALI´S, B. 2008. An Event-Based Approach to Reducing Coupling in Large-Scale Applications. In Bubak, M., van Albada, G., Dongarra, J., and Sloot, P., editors, *Computational Science – ICCS 2008, volume 5103 of Lecture Notes in Computer Science*, pages 358–367. *Springer-Verlag*, Berlin and Heidelberg.

MARTIN, R. C. 2003. Agile software development: Principles, patterns, and practices. *Prentice Hall*, Upper Saddle River and N.J.

MASAK, D. 2007. SOA? Serviceorientierung in Business und Software. *Springer-Verlag*, Berlin and Heidelberg.

MORRISON, J. P. 2010. Flow-Based Programming: A new approach to application development. *J.P. Morrison Enterprises*, Unionville and Ont., 2 edition. *Available: http://www.worldcat.org/oclc/694201092*

READE, C. 1989. Elements of functional programming. *Addison-Wesley*, Wokingham and England.

REUSSNER, R. AND HASSELBRING, W. 2009. Handbuch der Software-Architektur. *Dpunkt-Verl.*, Heidelberg, 2 edition.

WESTPHAL, R. 2010A. Nicht nur außen schön: Event-Based Components. *dotnetpro*, (8.2010):126–133.

WESTPHAL, R. 2010B. One Man Think Tank Gedanken. *Available: http://ralfw.blogspot.de/search/label/Event-based%20Components*

WESTPHAL, R. 2010C. Stecker mit System: Event-Based Components. *dotnetpro*, (7.2010):126–133.

WESTPHAL, R. 2010D. The Architect's Napkin: .NET Software Architecture on the Back of a Napkin. *Available: http://geekswithblogs.net/theArchitectsNapkin/category/11899.aspx*

WESTPHAL, R. 2010E. Zusammenstecken - funktioniert: Event-Based Components. *dotnetpro*, (6.2010):132–138.

WESTPHAL, R. 2011A. Lass es fließen: IEnumerable<T> richtig einsetzen. *dotnetpro*, (3.2011):128–134.

WESTPHAL, R. 2011B. Tools für Event-Based Components, Teil 1: Auf dem Weg zum Autorouter. *dotnetpro*, (1.2011):136–140.

WESTPHAL, R. 2011C. Tools für Event-Based Components, Teil 2: Das Navi für Signale. *dotnetpro*, (2.2011):136–140.

WESTPHAL, R. 2011D. Von Substantiven, Verben und EBCs: Abhängigkeiten in Flow Designs. *dotnetpro*, (4.2011):132–138.

WESTPHAL, R. AND LIESER, S. 2011. Event-Based Components Tooling. *Available: http://ebclang.codeplex.com*