Towards an Architectural Framework for Agile Software Development

Abstract

—One of the ideas of agile software development is to respond to changes rather than following a plan. Constantly changing businesses result in changing requirements, to be handled in the development process. Therefore, it is essential that the underlying software architecture is capable of managing agile business processes. However, criticism on agile software development states that it fails to pay attention to architectural and design issues and therefore is bound to engender suboptimal design-decisions. In this paper we propose an architectural framework for agile software development, that by explicitly separating computational, coordinational, and communicational models offers a high degree of flexibility regarding architectural and design changes introduced by agile business processes. The framework strength is facilitated by combining the characteristics and properties of state-of-the-art middleware architectural styles captured in a simple API.

The benefit of our approach is a clear architectural design with minimized effects of changes the models have on each other, accompanied by an efficient realization of new business requirements.

I. INTRODUCTION

Business constantly changes. Therefore, software architectures should be able to manage agile business processes and need to have the ability to meet future changes and business needs. The field of agile software development [1], [2] (ASD) addresses exactly the challenges of an unpredictable, turbulent business and technology environment. Thus, the question is how to better handle architectural changes, while still achieving high quality.

For distributed systems it is essential to make use of a flexible and adaptable platform that can respond to new requirements in an efficient way. Consequently, the usage of appropriate architectural styles for the design of software systems is a challenge. A common approach towards creating flexible, dynamic business processes and agile applications is the service-oriented computing style (SOA) [3]. For instance, the Enterprise Service Bus (ESB) [4] promises to interconnect and route services in a loosely coupled manner for a clear separation of business logic and integration logic. However, an ESB routes service data from one application to another and usually does not keep the history of messages and service interaction, i.e. does not maintain a global state

Thus, the main question in software development regarding software architecture still remains [5]. On the one hand how many various numbers of eventualities have to be taken into consideration, and therefore how much time and effort should be invested into design and implementation of components with respect of a good architectural design to cover all these circumstances, which eventually at the end may be not used at all. On the other hand, no or hardly any planning ahead, and at the same time bearing the risk of redesigning the existing architectural design from the scratch, once it is not capable of handling the latest requirement. All in all, it means that architecture and business do not evolve in the same way and same ”speed” [6]. Problems regarding architectural and design issues in ASD have been discussed in several papers, like [7], [8], [9], [10], stating that ASD fails to pay attention to architectural and design issues and therefore is bound to engender suboptimal design-decisions.

In previous work [11], we argued that although, software systems are usually not built by means of a single architectural style, there is a tight coupling between the application and the used style. This implies adaptations of the application in case the middleware has to be altered due to changing business requirements.

Extending the ideas in [11], in this paper we propose the Architecture Framework for Agile processes (AFA)1, in which it is explicitly distinguished between computational logic, coordinational and communicational models. The three models are independent of each other and therefore AFA offers a high degree of flexibility regarding architectural and design changes introduced by agile business processes. Like in [11], our approach combines and includes the characteristics and properties of the major architectural styles found in distributed middleware (section IV) captured in a simple API. AFA can be seen as an abstraction layer between applications and architectural styles, and as such it provides loosely coupling between the applications and their way of coordinating each other, between applications and architectural styles, between the applications and the way they exchange information.

The benefits of the proposed AFA approach are a) the efficient realization of changing business requirements affecting the underlying architecture; and b) adaptations of the architecture transparent to the application resulting in less complex application logic since it can entirely focus on its business process.

The remainder of this paper is structured as the following: section II pictures the use case, section III defines research questions, section IV summarizes related work, section V describes the concept and the architecture, while section VI discusses the proposed concept. Finally section VII concludes the paper and proposes further work

II. SCENARIO

In this section, we introduce a fictive scenario, based on an insurance company and its agents in field services, which should demonstrate the need for a change in the architecture due to new business requirements. In section V-C we explain how AFA abstracts these transitions.

As a starting point let’s assume that agents visiting potential customers fill in insurance related forms at the customer. Due to technical and economical reasons the mobile agent needs a permanent connection to the main insurance server of the company, both physically via e.g., UMTS and logically to its services. However, the required permanent connection between the agents and the main server hinders the agents to work efficiently with their customers. The agents cannot be sure whether the transmission capabilities of the provider cover the area where the customer lives, leading to an unreliable customer information management. This brings in a new requirement demanding the agent capability of working offline as well, without being dependable on a permanent connection. However, this leads to a break in the architecture in the sense that data stored before on the main server only, has to be partially replicated to the agents’ mobile devices and persisted there. Therefore, both the server and the application need to manage their own data and need to have the capability of synchronizing data changes.

III. RESEARCH QUESTIONS

Inspired by agile software development and based on the limitations of traditional middleware technologies with respect to introduction of new business requirements and their effects, we wanted to investigate a) the advantages and limitations of the proposed approach with respect to changing business requirements, b) what are the advantages of decoupling the three models, and c) how to realize changing business requirements transparent to the participating clients.

IV. RELATED WORK

This section summarizes related work on architectural styles and agile software development.

A. Architectural Styles Distributed middleware are mostly based on either dataflow style, such as pipes-and-filters, on data-centered style, i.e. a repository, or on implicit invocations, like publish-subscribe or event-based [12].

1) Dataflow Architectural Style: Pipes-and-filters, representing the dataflow style, define independent components (filters) that can be connected with each other but which do not know about the existence of other filters [13]. The connections between filters determine the pipeline. Sharing data between filters is only possible by passing it from one filter to the next, even if it is not needed in an intermediary step. SOA [3] typically makes use of the pipes-and-filters style. Services can be implemented as filters and the way of routing messages determines the pipeline that represents the business logic. The ESB [4] is the major platform used in SOA offering the necessary functionality in order to make use of SOA. The ESB discards any service-relevant data after message delivery. Thus, it cannot offer a shared repository that clients can use in order to coordinate themselves.

2) Data-centered Architectural Style: The essence of data-centered styles is that multiple components have access to the same data store, and communicate through that data store. A shared repository does provide its clients with access to shared data. Databases are the typical representation of this data-centered architectural style. Active repositories tie together the shared repository with another architectural style, which are event-based systems [14]. An active repository is able to notify registered clients about changes [13]. A repository does not provide the means for specifying in which order its shared data needs to be processed by its clients. Thus, repositories cannot offer routing capabilities in order to determine the processing sequence among its clients. Thus, it is irrelevant for the usage in pipes-and-filters.

Another data-centered architectural style is the blackboard based one, in which the state of the information on the blackboard determines the order of execution. A representative of the style is e.g., the Linda coordination model by David Gelernter [15]. It describes the usage of a logically shared memory, called tuple space, by means of simple operations (out, in, rd, eval) as a communication mechanism for parallel and distributed processes. Un like Linda, AFA (section V) allows e.g., storing shared data in a customizable structured way. This facilitates the efficient implementation of coordination concerns among middleware clients.

In [16] an extension to the pipes-and-filters style was proposed, where a shared repository is also supported. However, the hybrid framework does not offer the abstraction of the pipes-and-filters style but rather adds shared data to the pipeline.

3) Implicit Invocation Architectural Style: This style is characterized by calls that are invoked indirectly and implicitly as a response to a notification or an event. The group is represented by the publish/subscribe [17] and event-based [14] architectural styles.

B. Agile Software Development

Concepts for agile software development (ASD) have been created by experienced practitioners and can be seen as a reaction to e.g., plan-based methods, which attach value to” a rationalized, engineering-based approach” [18]. There, problems are seen to be fully specifiable and solvable with an optimal and predictable solution. By means of extensive planning and codified processes development can be made efficient and predictable. By contrast, ASD has been proposed as a solution to problems resulting from an unpredictable world. Several agile methods have evolved over time, like XP [19] or Scrum [20]. However, there is also skepticism [2] regarding ASD with respect to architecture design and implementation issues. One of them is that agile development is an excuse for developers to implement as they like, coding away without proper planning or design [7], [5] and consequently causing suboptimal design decisions [8], [9].

V. ARCHITECTURE

This section pictures the idea of AFA in detail. It gives a brief introduction of its components and explains how decoupling between the three models is achieved. Finally, it describes how the given change in the scenario (section II) can be realized with the proposed concept.

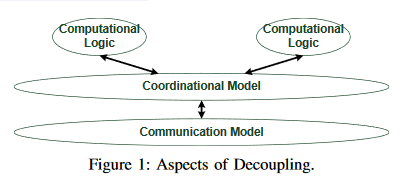
A. The Architectural Framework for Agile Processes

The main component of the AFA architecture are Internet addressable containers [21] which is a collection of entries accessible via a basic simple API. The container’s interface provides an API for reading, taking, and writing entries, but extends the original Linda API with the methods destroy, shift and notify. Destroy removes an entry from the container, while shift writes an entry after it has removed one. Another important component is the so called coordinator [21] which are programmable parts of the container being responsible for managing their view on the entries in the container. The aim of a coordinator is to represent a coordination model and to structure and organize the entries in the container for efficient access [22]. The difference to Linda is that a container may be bounded to a maximum number of entries, and allows the usage of so called coordinators with each having its specific and optimized view on the stored entries. In contrast to databases, AFA offers blocking operations known from the Linda model, thus allowing queries for future data states. Furthermore, databases need a static data model of the entries they have to store, while containers allow the usage of several different coordinators at the same time, enabling efficient dynamic data models, and thus being schema-free.

The last main component of AFA are the so called aspects [22], which represent additional computational logic and are executed on the peer where the container is located. Aspects are triggered by operations on a specific container, rather on the according impact. Aspects can be located before or after the execution of an operation and added and removed at any time during runtime. A detailed explanation of how aspects work and the interrelation between aspects and containers is given in [22].

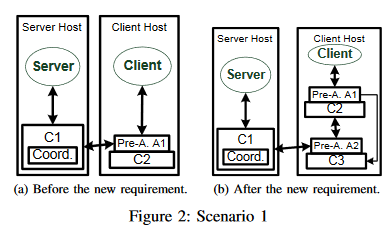
B. Supported ways of Decoupling in AFA

In [11] we have described how different architectural styles can be combined to manage architecture limiters and breaker. The concept identified several layers, each responsible for different tasks. By means of combining the different responsibilities of each layer a specific architectural style with specific configuration has been created. However, those layers can be explicitly clustered and categorized (figure 1) resulting in models distinguished by their capabilities for managing computational, coordinational, or communicational requirements.



As stated in [15] the computation model is used to express the computational requirements of an algorithm, while the coordination model is used to express the coordinational requirements. On the one hand, this decoupling allows to change the application without having an effect on the way it coordinates itself with other applications. On the other hand, a fifo style of coordination - comparable to message-based communication - can be switched to e.g., lifo style of coordination, or more complex coordination models [21]. This is done by replacing the existing coordinator in the container. In traditional sense, the computational and coordination models are combined, since a lot of the systems rely on the pipes-and-filters or call-and-return architectural styles. This implies that the application itself also has to contain and implement the complexities coming along with the used coordination model.

However, so far it has only be defined how and which applications coordinate each other. It has not been specified whether the applications run in the same process, on the same machine, or distributed on the Internet. Furthermore, it has not been specified how the necessary information needed for coordination is exchanged between the participating application of the coordination model.



This part is done by the communication model. To be correct, this model needs to be divided into a dissemination/distribution model and the real communication model. Since containers store entries, the dissemination model specifies where the container and its data have to be stored physically, whether there are multiple replica of it and how they are kept consistent [22]. The communication model describes how data from one container is transmitted to the other. The model may contain lower level protocols like udp, tcp, or higher level ones like P2P protocols.

A big contribution to this picture is done by the aspects. As mentioned aspects may contain any computational logic, and may return several different values [22] once that logic has been executed. In combination this adds extra value to the framework. As an example, a container may host a pre-aspect that does nothing else, but rerouting incoming operations to other services and then skipping the operation. This implies that the container will never be filled with data. However, the application has not been changed, it still uses the same coordination model, but data is not retrieved from or written into the container. As an example the data may be retrieved from and written to a database via a WebService call.

C. Handling Agile Business Requirements with AFA

This section intends to explain the idea of AFA by means of the scenarios described in section II. The new business requirement in the first scenario is to allow the agents to work offline. Figure 2a shows the used components in an AFA environment before the new requirement. The figure shows that both, server and client part, have a container running on their machines. However, only the server part keeps the data accessed by the clients. Clients access the data by accessing the local container C2. However C2 does not contain any data, since pre-aspect A1 intercepts the operation and reroutes it to the server.

The new requirements state that some data needs to be stored at the client as well. A very simple solution to that problem is shown in figure 2b. Instead of forwarding operations to container C1, pre-aspect A1 now makes a copy of it and executes the operation on container C3 as well. This container is needed to keep track of changes while the client is offline in order to be capable of executing a synchronization strategy once the client is online again. Synchronization is done by pre-aspect A2, that checks connectivity status and updates container C2, once the synchronization process has started.

VI. DISCUSSION

In this paper, we propose the concept of an Architecture Framework for Agile processes (AFA) in order to allow the realization of new business requirements. The advantages of the proposed approach with respect to changing business requirements are that the applications do not have to consider a) the underlying architectural style, b) the used coordination model, c) the used communication technology, or d) the way how data is disseminated. However, an abstraction technology placed between middleware and client application causes an additional overhead which affects overall performance. Nevertheless, the proposed concept allows decreasing development and migration time by reducing the effort needed to adapt the current system to new business requirements and therefore it saves costs while improving adaptability and re-usability.

Summing up, we take a look at the quality attributes defined in [23].

Evaluations have to be performed, but we think that the overall performance decreases due to the additional layers in AFA.

Security is supported transparently to client applications by adding security-relevant aspects to the AFA middleware.

As shown in [22] containers in AFA can be transparently replicated in order to improve availability and fault tolerance.

With respect to usability, AFA offers a generic interface. Thus, components always act upon the same interface, which improves modifiability, modularization and encapsulation. By abstracting the underlying architectural style AFA can be ported to many different middleware technologies, such as inter-process communication (e.g. RMI), SOA, etc.

A specific constellation of containers, their coordinators, and installed aspects can be seen as a pattern for a specific problem. In this sense AFA facilitates re-usability.

Since changes are restricted to one of the three models, other components can be tested in the same manner which improves testability.

VII. CONCLUSION AND FUTURE WORK

In this paper, we described the concept of the Architecture Framework for Agile processes (AFA) as an abstraction framework in order to allow the efficient realization of new business requirements with minimal effects on other components in the architecture.

Based on the components AFA provides, and the explicit separation between computational, coordinational, and communicational model a high degree of flexibility has been achieved with minimal effects on each other in case one of the models has to be adapted due to new business requirements.

The benefits of the approach are a rigid decoupling between the models and architectural styles allowing changes in the architecture with minimal effect on other components, resulting in less testing and therefore minimized time-to-market. Since new requirements can be mapped to one of the models, the time needed to adapt the system reduces, and therefore saves costs.

Further work will include a benchmarking of the framework, i.e. to what extent does the additional abstraction layer decrease computational performance. A more comprehensive evaluation with respect to testing and development time is intended. The latter case will also investigate the influence of software development experience of software engineers on software development with AFA support.

Traducian Español

Hacia un marco arquitectónico para el desarrollo ágil de software

Resumen

—Una de las ideas del desarrollo ágil de software. Es responder a los cambios en lugar de seguir un plan. Los negocios en constante cambio dan como resultado requisitos cambiantes, que se manejarán en el proceso de desarrollo. Por lo tanto, es esencial que la arquitectura de software subyacente sea ​​capaz de administrar procesos de negocios ágiles. Sin embargo, las críticas sobre el desarrollo ágil de software afirman que no presta atención a los problemas de arquitectura y diseño y, por lo tanto, está obligado a generar decisiones de diseño subóptimas. En este trabajo proponemos un marco arquitectónico para El desarrollo ágil de software, que al separar explícitamente los modelos computacionales, de coordinación y de comunicación, ofrece un alto grado de flexibilidad con respecto a los cambios de arquitectura y diseño introducidos por los procesos de negocios ágiles. La fuerza del marco se facilita al combinar las características y propiedades de los estilos arquitectónicos de middleware de vanguardia capturados en una API simple.

El beneficio de nuestro enfoque es un diseño arquitectónico claro con los efectos minimizados de los cambios que los modelos tienen entre sí, acompañados por una realización eficiente de los nuevos requisitos comerciales.

I. INTRODUCCIÓN

El negocio cambia constantemente. Por lo tanto, las arquitecturas de software deben ser capaces de administrar procesos de negocios ágiles y deben tener la capacidad de satisfacer cambios futuros y necesidades comerciales. El campo del desarrollo de software ágil [1], [2] (ASD) aborda exactamente los desafíos de un entorno empresarial y tecnológico impredecible y turbulento. Por lo tanto, la pregunta es cómo manejar mejor los cambios arquitectónicos, mientras se logra una alta calidad.

Para los sistemas distribuidos es esencial hacer uso de una plataforma flexible y adaptable que pueda responder a los nuevos requisitos de una manera eficiente. En consecuencia, el uso de estilos arquitectónicos apropiados para el diseño de sistemas de software es un desafío. Un enfoque común hacia la creación de procesos de negocios flexibles y dinámicos y aplicaciones ágiles es el estilo informático orientado a servicios (SOA) [3]. Por ejemplo, el Enterprise Service Bus (ESB) [4] se compromete a interconectar y enrutar los servicios de manera flexible para una clara separación de la lógica empresarial y la lógica de integración. Sin embargo, un ESB enruta los datos del servicio de una aplicación a otra y generalmente no guarda el historial de mensajes y la interacción del servicio, es decir, no mantiene un estado global.

Por lo tanto, la pregunta principal en el desarrollo de software con respecto a la arquitectura de software aún permanece [5]. Por un lado, cuántas cantidades diferentes de eventualidades deben tomarse en consideración y, por lo tanto, cuánto tiempo y esfuerzo se debe invertir en el diseño e implementación de componentes con respecto a un buen diseño arquitectónico para cubrir todas estas circunstancias, que eventualmente El final no puede ser usado en absoluto. Por otro lado, no hay planificación previa o casi ninguna, y, al mismo tiempo, corre el riesgo de rediseñar el diseño arquitectónico existente desde cero, una vez que no es capaz de manejar el último requisito. En definitiva, significa que la arquitectura y los negocios no evolucionan de la misma manera y la misma "velocidad " [6]. Los problemas relacionados con los problemas de arquitectura y diseño en ASD se han discutido en varios artículos, como [7], [8], [9], [10], y afirman que ASD no presta atención a los problemas de arquitectura y diseño y, por lo tanto, está obligado a generar Decisiones de diseño subóptimas.

En el trabajo anterior [11], argumentamos que aunque los sistemas de software generalmente no se construyen por medio de un solo estilo arquitectónico, existe un acoplamiento estrecho entre la aplicación y el estilo utilizado. Esto implica adaptaciones de la aplicación en caso de que el middleware tiene que ser alterado debido a la evolución de las necesidades de negocio.

Ampliando las ideas en [11], en este documento proponemos el Marco de Arquitectura para Procesos Agile (AFA) 1, en el cual se distingue explícitamente entre lógica computacional, modelos de coordinación y comunicación. Los tres modelos son independientes entre sí y, por lo tanto, AFA ofrece un alto grado de flexibilidad con respecto a los cambios de arquitectura y diseño introducidos por los procesos de negocios ágiles. Al igual que en [11], nuestro enfoque combina e incluye las características y propiedades de los principales estilos arquitectónicos que se encuentran en el middleware distribuido (sección IV) capturados en una API simple. AFA puede verse como una capa de abstracción entre aplicaciones y estilos arquitectónicos, y como tal, proporciona un acoplamiento flexible entre las aplicaciones y su forma de coordinarse entre las aplicaciones y los estilos arquitectónicos, entre las aplicaciones y la forma en que intercambian información.

Los beneficios del enfoque AFA propuesto son a) la realización eficiente de los requisitos comerciales cambiantes que afectan la arquitectura subyacente; y b) adaptaciones de la arquitectura transparente para la aplicación, lo que resulta en una lógica de aplicación menos compleja, ya que puede centrarse completamente en su proceso de negocios.

El resto de este documento está estructurado de la siguiente manera: la sección II describe el caso de uso, la sección III define las preguntas de investigación, la sección IV resume el trabajo relacionado, la sección V describe el concepto y la arquitectura, mientras que la sección VI analiza el concepto propuesto. Finalmente la sección VII concluye el documento y propone trabajo adicional.

II. ESCENARIO

En esta sección, presentamos un escenario ficticio, basado en una compañía de seguros y sus agentes en servicios de campo, que debe demostrar la necesidad de un cambio en la arquitectura debido a los nuevos requisitos comerciales. En la sección VC explicamos cómo AFA resume estas transiciones.

Como punto de partida, supongamos que los agentes que visitan a los clientes potenciales completan los formularios de seguros relacionados con el cliente. Debido a razones técnicas y económicas, el agente móvil necesita una conexión permanente con el servidor de seguros principal de la empresa, tanto físicamente a través de UMTS como lógicamente a sus servicios. Sin embargo, la conexión permanente requerida entre los agentes y el servidor principal impide que los agentes trabajen de manera eficiente con sus clientes. Los agentes no pueden estar seguros de si las capacidades de transmisión del proveedor cubren el área donde vive el cliente, lo que lleva a una gestión de información del cliente poco confiable. Esto trae un nuevo requisito que exige la capacidad del agente para trabajar también fuera de línea, sin ser confiable en una conexión permanente. Sin embargo, esto conduce a una ruptura en la arquitectura en el sentido de que los datos almacenados antes en el servidor principal solamente, tiene que ser parcialmente replicado a los dispositivos móviles de los agentes y persistieron allí. Por lo tanto, tanto el servidor como la aplicación necesitan administrar sus propios datos y deben tener la capacidad de sincronizar los cambios de datos.

III. PREGUNTAS DE INVESTIGACIÓN

Inspirados por el desarrollo ágil de software y en base a las limitaciones de las tecnologías de middleware tradicionales con respecto a la introducción de nuevos requisitos comerciales y sus efectos, quisimos investigar

a) las ventajas y limitaciones del enfoque propuesto con respecto a los requisitos comerciales cambiantes,

b) qué son las ventajas de desacoplar los tres modelos, y

c) cómo realizar cambios en los requisitos comerciales de forma transparente para los clientes participantes.

IV. TRABAJO RELACIONADO

Esta sección resume el trabajo relacionado con los estilos arquitectónicos y el desarrollo ágil de software.

1. Estilos arquitectónicos

El middleware distribuido se basa principalmente en un estilo de flujo de datos, como tuberías y filtros, en un estilo centrado en los datos, es decir, en un repositorio, o en invocaciones implícitas, como publicación-suscripción o basado en eventos [12].

1) Estilo arquitectónico del flujo de datos: los tubos y filtros, que representan el estilo del flujo de datos, definen componentes independientes (filtros) que pueden conectarse entre sí pero que no conocen la existencia de otros filtros [13]. Las conexiones entre los filtros determinan la tubería. Compartir datos entre filtros solo es posible pasando de un filtro a otro, incluso si no es necesario en un paso intermedio. SOA [3] normalmente utiliza el estilo de tuberías y filtros. Los servicios pueden implementarse como filtros y la forma de enrutar los mensajes determina la canalización que representa la lógica empresarial. El ESB [4] es la principal plataforma utilizada en SOA que ofrece la funcionalidad necesaria para hacer uso de SOA. El ESB descarta cualquier dato relevante para el servicio después de la entrega del mensaje. Por lo tanto, no puede ofrecer un repositorio compartido que los clientes puedan usar para coordinarse.

2) Estilo arquitectónico centrado en los datos: la esencia de los estilos centrados en los datos es que múltiples componentes tienen acceso al mismo almacén de datos y se comunican a través de ese almacén de datos. Un repositorio compartido proporciona a sus clientes acceso a datos compartidos. Las bases de datos son la representación típica de este estilo arquitectónico centrado en los datos. Los repositorios activos unen el repositorio compartido con otro estilo arquitectónico, que son sistemas basados ​​en eventos [14]. Un repositorio activo puede notificar a los clientes registrados sobre los cambios [13]. Un repositorio no proporciona los medios para especificar en qué orden los clientes deben procesar sus datos compartidos. Por lo tanto, los repositorios no pueden ofrecer capacidades de enrutamiento para determinar la secuencia de procesamiento entre sus clientes. Por lo tanto, es irrelevante para el uso en tuberías y filtros.

Otro estilo arquitectónico centrado en los datos es el basado en la pizarra, en el que el estado de la información en la pizarra determina el orden de ejecución. Un representante del estilo es, por ejemplo, el modelo de coordinación de Linda por David Gelernter [15]. Describe el uso de una memoria compartida de forma lógica, llamada espacio de tupla, mediante operaciones simples (out, in, rd, eval ) como un mecanismo de comunicación para procesos paralelos y distribuidos. Naciones Unidas como Linda, AFA (sección V) permite, por ejemplo, almacenar datos compartidos de una manera estructurada personalizable. Esto facilita la implementación eficiente de las preocupaciones de coordinación entre los clientes de middleware.

En [16] se propuso una extensión al estilo de tuberías y filtros, donde también se admite un repositorio compartido. Sin embargo, el marco híbrido no ofrece la abstracción del estilo de tuberías y filtros, sino que agrega datos compartidos a la tubería.

3) Estilo arquitectónico de invocación implícita: este estilo se caracteriza por llamadas que se invocan de manera indirecta e implícita como respuesta a una notificación o evento. El grupo está representado por los estilos arquitectónicos de publicación / suscripción [17] y basados ​​en eventos [14].

B. Desarrollo de software ágil

Los conceptos para el desarrollo ágil de software (ASD, por sus siglas en inglés) han sido creados por profesionales con experiencia y pueden verse como una reacción a, por ejemplo, métodos basados ​​en planes, que agregan valor a " un enfoque racionalizado, basado en la ingeniería” [18]. Allí, los problemas parecen ser totalmente especificables y solucionables con una solución óptima y predecible. Por medio de una extensa planificación y el desarrollo de procesos codificados se puede hacer eficiente y predecible. Por el contrario, el ASD se ha propuesto como una solución a los problemas resultantes de un mundo impredecible. Varios métodos ágiles han evolucionado con el tiempo, como XP [19] o Scrum [20]. Sin embargo, también hay escepticismo [2] con respecto a ASD con respecto al diseño de la arquitectura y los problemas de implementación. Uno de ellos es que el desarrollo ágil es una excusa para que los desarrolladores implementen lo que quieran, codifiquen sin la planificación o el diseño adecuados [7], [5] y, en consecuencia, causen un diseño subóptimo. decisiones [8], [9].

V. LA ARQUITECTURA

Esta sección muestra la idea de AFA en detalle. Da una breve introducción de sus componentes y explica cómo se logra el desacoplamiento entre los tres modelos. Finalmente, describe cómo el cambio dado en el escenario (sección II) se puede realizar con el concepto propuesto.

A. El marco arquitectónico para procesos ágiles

El componente principal de la arquitectura AFA son los contenedores a los que se puede acceder por Internet [21], que es una colección de entradas accesibles a través de una API simple básica. La interfaz del contenedor proporciona una API para leer, tomar y escribir entradas, pero amplía la API original de Linda con los métodos de destruir, cambiar y notificar. Destroy elimina una entrada del contenedor, mientras Shift escribe una entrada después de que haya eliminado una entrada. Otro componente importante es el llamado coordinador [21], que son partes programables del contenedor que son responsables de administrar su vista en las entradas en el contenedor. El objetivo de un coordinador es representar un modelo de coordinación y estructurar y organizar las entradas en el contenedor para un acceso eficiente [22]. La diferencia con Linda es que un contenedor puede estar limitado a un número máximo de entradas, y permite el uso de los llamados coordinadores, cada uno con su vista específica y optimizada en las entradas almacenadas. A diferencia de las bases de datos, AFA ofrece operaciones de bloqueo conocidas del modelo de Linda, lo que permite realizar consultas para futuros estados de datos. Además, las bases de datos necesitan un modelo de datos estáticos de las entradas que tienen que almacenar, mientras que los contenedores permiten el uso de varios coordinadores diferentes al mismo tiempo, lo que permite modelos de datos dinámicos eficientes y, por lo tanto, están libres de esquemas.

El último componente principal de AFA son los llamados aspectos [22], que representan una lógica computacional adicional y se ejecutan en el par donde se encuentra el contenedor. Los aspectos son desencadenados por operaciones en un contenedor específico, en lugar del impacto correspondiente. Los aspectos se pueden ubicar antes o después de la ejecución de una operación y se pueden agregar y eliminar en cualquier momento durante el tiempo de ejecución. En [22] se ofrece una explicación detallada de cómo funcionan los aspectos y la interrelación entre los aspectos y los contenedores.

B. Formas soportadas de desacoplamiento en AFA

En [11] hemos descrito cómo se pueden combinar diferentes estilos arquitectónicos para administrar los limitadores y el interruptor de la arquitectura. El concepto identificó varias capas, cada una responsable de diferentes tareas. Mediante la combinación de las diferentes responsabilidades de cada capa, se ha creado un estilo arquitectónico específico con una configuración específica. Sin embargo, esas capas se pueden agrupar y categorizar explícitamente (figura 1), lo que da como resultado modelos que se distinguen por sus capacidades para gestionar los requisitos informáticos, de coordinación o de comunicación.

Como se indica en [15], el modelo de cálculo se usa para expresar los requisitos computacionales de un algoritmo, mientras que el modelo de coordinación se usa para expresar los requisitos de coordinación. Por un lado, este desacoplamiento permite cambiar la aplicación sin tener un efecto en la forma en que se coordina con otras aplicaciones. Por otro lado, un estilo fifo de coordinación - comparable a la comunicación basada en mensajes - se puede conectar a, por ejemplo, LIFO estilo de coordinación, o modelos de coordinación más complejos [21]. Esto se hace reemplazando el coordinador existente en el contenedor. En el sentido tradicional, los modelos computacionales y de coordinación se combinan, ya que muchos de los sistemas se basan en los estilos arquitectónicos de tuberías y filtros o de devolución y devolución. Esto implica que la aplicación en sí misma también debe contener e implementar las complejidades que acompañan al modelo de coordinación utilizado.

Sin embargo, hasta el momento sólo ha se define cómo y qué aplicaciones se coordinan entre sí. No se ha especificado si las aplicaciones se ejecutan en el mismo proceso, en la misma máquina o distribuidas en Internet. Además, no se ha especificado cómo se intercambia la información necesaria para la coordinación entre la aplicación participante del modelo de coordinación.

Esta parte está hecha por el modelo de comunicación. Para ser correcto, este modelo debe dividirse en un modelo de difusión / distribución y el modelo de comunicación real. Desde entradas almacenar contenedores, el modelo de difusión especifica en el que el recipiente y sus datos tienen que ser almacenados físicamente, si existen réplicas múltiples de la misma y la forma en que se mantienen consistentes [22]. El modelo de comunicación describe cómo los datos de un contenedor se transmiten al otro. El modelo puede contener protocolos de nivel inferior como udp , tcp o protocolos de nivel superior como los protocolos P2P.

Una gran contribución a esta imagen se hace por los aspectos. Como se mencionó, los aspectos pueden contener cualquier lógica computacional y pueden devolver varios valores diferentes [22] una vez que se haya ejecutado esa lógica. En combinación, esto agrega valor extra al marco. Como ejemplo, un contenedor puede albergar un aspecto previo que no hace nada más, pero desviando las operaciones entrantes a otros servicios y luego omitiendo la operación. Esto implica que el contenedor nunca se llenará de datos. Sin embargo, la aplicación no se ha cambiado, todavía utiliza el mismo modelo de coordinación, pero los datos no se recuperan ni se escriben en el contenedor. Como ejemplo, los datos pueden recuperarse y escribirse en una base de datos a través de una llamada de servicio web.

C. Manejo de requisitos comerciales ágiles con AFA

Esta sección pretende explicar la idea de AFA a través de los escenarios descritos en la sección II. El nuevo requisito comercial en el primer escenario es permitir que los agentes trabajen sin conexión. La Figura 2a muestra los componentes utilizados en un entorno AFA antes del nuevo requisito. La figura muestra que tanto el servidor como la parte del cliente tienen un contenedor ejecutándose en sus máquinas. Sin embargo, solo la parte del servidor mantiene los datos a los que acceden los clientes. Los clientes acceden a los datos accediendo al contenedor local C2. Sin embargo, C2 no contiene ningún dato, ya que el aspecto previo A1 intercepta la operación y la redirige al servidor.

Los nuevos requisitos establecen que algunos datos también deben almacenarse en el cliente. Una solución muy simple para ese problema se muestra en la figura 2b. En lugar de reenviar las operaciones al contenedor C1, el aspecto previo A1 ahora hace una copia del mismo y ejecuta la operación en el contenedor C3 también. Este contenedor es necesario para realizar un seguimiento de los cambios mientras el cliente está fuera de línea para poder ejecutar una estrategia de sincronización una vez que el cliente vuelva a estar en línea. La sincronización se realiza mediante el aspecto previo A2, que comprueba el estado de conectividad y actualiza el contenedor C2, una vez que el proceso de sincronización ha comenzado.

VI. DISCUSIÓN

En este documento, proponemos el concepto de un marco de arquitectura para procesos ágiles (AFA) con el fin de permitir la realización de nuevos requisitos comerciales. Las ventajas del enfoque propuesto con respecto a los requisitos comerciales cambiantes son que las aplicaciones no tienen que considerar a) el estilo arquitectónico subyacente, b) el modelo de coordinación utilizado, c) la tecnología de comunicación utilizada, o d) la forma en que los datos son diseminado. Sin embargo, una tecnología de abstracción colocada entre el middleware y la aplicación cliente genera una sobrecarga adicional que afecta el rendimiento general. Sin embargo, el concepto propuesto permite disminuir el tiempo de desarrollo y migración al reducir el esfuerzo necesario para adaptar el sistema actual a los nuevos requisitos comerciales y, por lo tanto, ahorra costos al tiempo que mejora la adaptabilidad y la reutilización.

En resumen, echamos un vistazo a los atributos de calidad definidos en [23].

Las evaluaciones deben llevarse a cabo, pero creemos que el rendimiento global disminuye debido a las capas adicionales en AFA.

Seguridad es compatible de forma transparente con las aplicaciones cliente agregando aspectos relevantes para la seguridad al middleware AFA.

Como se muestra en [22], los contenedores en AFA se pueden replicar de manera transparente para mejorar la disponibilidad y la tolerancia a fallas.

Con respecto a la usabilidad, AFA ofrece una interfaz genérica. Por lo tanto, los componentes siempre actúan sobre la misma interfaz, lo que mejora la modificabilidad, la modularización y la encapsulación. Al abstraer el estilo arquitectónico subyacente, AFA se puede portar a muchas tecnologías de middleware diferentes, como la comunicación entre procesos (por ejemplo, RMI), SOA, etc.

Una constelación específica de contenedores, sus coordinadores y aspectos instalados pueden verse como un patrón para un problema específico. En este sentido, la AFA facilita la reutilización.

Dado que los cambios están restringidos a uno de los tres modelos, otros componentes pueden probarse de la misma manera que mejora la capacidad de prueba.

VII. CONCLUSIÓN Y TRABAJO FUTURO

En este documento, describimos el concepto del Marco de Arquitectura para Procesos Agile (AFA) como un marco de abstracción para permitir la realización eficiente de nuevos requisitos de negocios con efectos mínimos en otros componentes de la arquitectura.

Sobre la base de los componentes que proporciona AFA, y la separación explícita entre los modelos computacional, de coordinación y comunicacional, se ha logrado un alto grado de flexibilidad con efectos mínimos entre sí en caso de que uno de los modelos deba adaptarse debido a los nuevos requisitos comerciales.

Los beneficios de este enfoque son un desacoplamiento rígido entre los modelos y los estilos arquitectónicos que permiten cambios en la arquitectura con un efecto mínimo en otros componentes, lo que resulta en menos pruebas y, por lo tanto, minimiza el tiempo de comercialización. Dado que los nuevos requisitos se pueden asignar a uno de los modelos, el tiempo necesario para adaptar el sistema se reduce y, por lo tanto, ahorra costos.

El trabajo adicional incluirá una evaluación comparativa del marco, es decir, en qué medida la capa de abstracción adicional disminuye el rendimiento computacional. Se pretende una evaluación más completa con respecto a las pruebas y el tiempo de desarrollo. El último caso también investigará la influencia de la experiencia de desarrollo de software de los ingenieros de software en el desarrollo de software con soporte de AFA.

**Architecture and Agility: Married, Divorced, or Just Good Friends?**

¿El desarrollo ágil necesita arquitectura? ¿La arquitectura necesita un desarrollo ágil? ¿Es posible responder a estas preguntas sin un debate polarizador caracterizado más por caricatura y puntos de vista culturales arraigados que por definiciones claras y razonamiento abierto, un debate que se asemeja más a dos monólogos que se transmiten uno al otro que un diálogo? Quizás reformular la pregunta en términos más generales ofrezca un mejor lugar para comenzar: en lugar de enfocarnos específicamente en enfoques ágiles, deberíamos considerar los procesos de desarrollo de manera más amplia. Y, en lugar de plantear una suposición como una pregunta, deberíamos considerar una pregunta más abierta y neutral: ¿Cuál es la relación entre arquitectura y proceso?

Arquitectura y proceso

La arquitectura se define comúnmente como “la descomposición estructural de un sistema, incluida su descomposición en partes, su conectividad, los mecanismos de interacción y los principios rectores que informan el diseño del sistema”. 1

Aunque no es incorrecta desde una perspectiva técnica, esta definición está abierta a un amplio espectro de interpretación. Puede significar cualquier cosa, desde un boceto de diseño de alto nivel con poca relación con la tecnología, el código o el sistema real que se está construyendo 2 hasta un diseño grande y rígido con muchos detalles de clase y nivel de código. En la práctica, ninguna de las vistas ofrece una guía suficiente para el proceso real de desarrollo, una contribución necesaria a la arquitectura "buena". La primera forma es demasiado abstracta para dar una orientación concreta; El segundo es demasiado prematuro, decisivo antes de que se conozcan los detalles relevantes. Por lo tanto, no es sorprendente que algunos en la comunidad ágil sostengan que la arquitectura no es una preocupación central en el desarrollo de software práctico.

Grady Booch3 y Martin Fowler, en contraste, ofrecen definiciones de arquitectura de software orientadas al valor. Ambos definen la arquitectura en términos de decisiones significativas que son costosas y difíciles de cambiar. Paul Dyson y Andy Longshaw extienden la consideración estructural de la arquitectura con una razón: el “por qué”, que guía las decisiones de diseño. 5 Estas definiciones nos ayudan a ver la arquitectura como algo que responde a un conjunto de necesidades, como requisitos funcionales, características operativas, y habitabilidad del desarrollador.

En la práctica, la expresión de una arquitectura de software puede incluir una sección transversal de las decisiones, desde un bosquejo escrito a mano hasta detalles críticos, algunos de los cuales se reconocen de manera adecuada y explícita como decisiones, algunos de los cuales son suposiciones o hechos, y otros son decisiones que se hacen involuntariamente y se reconocen como significativos solo en retrospectiva. En consecuencia, la arquitectura se convierte en un servicio para guiar y desarrollar un sistema de software hacia el cumplimiento de un conjunto de objetivos comerciales y técnicos, 6 expresados ​​en la forma que mejor ayude a la comunicación y el intercambio. No es un artefacto técnico en sí mismo, expresado como una descripción puramente formal.

El proceso puede considerarse como la respuesta a un conjunto de preguntas: ¿Quién está haciendo qué, cuándo, cómo, por qué y para quién? Por lo tanto, todos los proyectos de desarrollo de software tienen un proceso, incluso si no es explícito: independientemente de las respuestas, estas preguntas pueden considerarse como el proceso, ya sea flexible o formal, ya sea que proporcione a un equipo control activo sobre el desarrollo o no.

Sin embargo, podemos ver que la forma en que el personal, el presupuesto y la planificación de un proyecto son decisiones que influirán significativamente en la elección y la viabilidad de cualquier arquitectura. Esto se aplica a muchas cosas, desde la función de forzamiento que aplica la ley de Conway7 a cómo se divide el sistema, independientemente de su visión original, a las cuestiones de elección de tecnología y habilidades, alcance previsto y modelo de lanzamiento y, en última instancia, al sistema real. diseño.

La interdependencia entre estas muchas influencias, junto con una tendencia a cambiar a lo largo del tiempo, exige procesos que establezcan cadenas de decisión claras para los aspectos relevantes de la arquitectura y ciclos de retroalimentación directos y significativos para ajustar cualquier decisión a la luz de la nueva información, respondiendo a las consecuencias inevitables de Descubrimiento e invención. Un proceso debe ser compatible con un equipo de proyecto, no al revés: ¡los equipos de proyecto no reciben pago por cumplir un proceso sino por entregar software en ejecución! Los procesos ágiles apuntan exactamente a este objetivo.

Qué significa la arquitectura para el desarrollo ágil

En contraste con algunas discusiones que observamos en la comunidad de software, el desarrollo ágil no se trata de la adopción por parte del culto de la carga de Scrum o cualquier otro proceso, conjunto de herramientas o metodología, aunque ciertamente observamos esto y lo consideramos un problema. La esencia de la agilidad es la capacidad de respuesta, el aprendizaje y la suficiencia. La agilidad se refleja en la sostenibilidad y la calidad del software y su desarrollo; por definición, el desarrollo insostenible y de mala calidad contradice y reduce la agilidad. El Manifiesto para el desarrollo de software ágil señala que "la atención continua a la excelencia técnica y al buen diseño mejora la agilidad", ofreciendo a la arquitectura un papel claro dentro de ágil, aunque no se expresa a través del gran diseño inicial.

Engineering Values: From Architecture Games to Agile Requirements

“The values to which people cling most stubbornly under inappropriate conditions are those values that were previously the source of their greatest triumphs.” —Jared Diamond, Collapse: How Societies Choose to Fail or Succeed

Personal Principles and values are important. Articulating the personal qualities that we prize gives us a way to set goals for ourselves and reflect on whether we’re living up to our own standards—or whether we’re so caught up in the day-to-day grind that we make a thousand little compromises and never notice.

Similarly, professional principles are touchstones that let us aim for rigor in our work rather than spending our efforts on a thousand different trade-offs and kludges. Perhaps such engineering principles are even more important in the early phases of any software life cycle, when the freedom to interpret the problem and design solutions can be almost unbounded.

And yet—when taken to extremes or applied blindly, any principle can produce the very outcomes that it was meant to avoid, stifling creativity and leading to bad decisions. Philippe Kruchten brought this point home to me afresh during his recent keynote and the personal interview that followed by enunciating some important values and their potential misuses in software requirements and architecture.

Kruchten’s work has often focused on software architecture, and in his consulting work, he has observed a large number of teams and their dynamics in operation. When I caught up with him at the Better Software East conference, he had just given a key-note summarizing some of the negative behavioral patterns that he has often seen in the field (he referred to them as “games architects play”). I was struck by how many of these initially sounded like positive engineering principles taken to extremes, which seems to be a cautionary tale regarding the potential for rules to get embedded out of context.

For Kruchten, one of the root causes of such games is that, like other humans, software professionals can be misled into thinking of ourselves as more rational than we are. Cognitive biases come into play during software development just as in other activities. For example, too often we fall rapidly into confirmation bias—finding valid-sounding reasons to support the things that we already believe. He also cautioned that we shouldn’t put the onus for such behavior just on architects: software development is one large continuum of decisions, and the same types of biases can come into play in each. Looking earlier in the life cycle, we would be kidding ourselves to think that users and requirements analysts are perfectly rational, and it’s just the “software guys” who misinterpret things and go off in strange directions.

The Pilot Project

Let’s begin with a game Kruchten identified that’s close to my own heart as an empiricist: the pilot project. There’s a solid principle at the heart of this one —namely, a desire to be intentional and cautious about new tools and technologies. A key component of making software engineering an evidence-driven field rather than one at the whim of fads is trying novel approaches in small-scale but realistic settings to see if the claimed benefits can really be obtained in one’s context (and at what price). Surprising as it sounds, I’ve seen organizations go off the rails by mandating widespread adoption of new technologies that turn out to be more complicated in practice than initially realized.

However, the “game” begins when organizations march into pilots with a preconceived idea of the outcome they want to see and eliminate anything that would be an obstacle to getting that predefined outcome—even if those obstacles are, you know, actually related to the way they normally do business. So how do we monitor ourselves and understand whether confirmation bias is actually coming into play? One heuristic is to count how often we hear, “But we’re not doing [something] on this project because this is just a pilot.” Has “just a pilot” become the mantra that we need to chant when removing anything that could possibly get an outcome other than our preferred one? It’s also useful to consider whether we’ve hand-picked a team that’s exceptionally good and might have obtained a successful outcome regardless of the technology or process they apply.

Blink

Another example Kruchten gave is “blink”—in which we tell ourselves that we have sufficient experience and know-how and can make decisions quickly, based on experience and intuition. Given that architects typically face a wide range of potential solutions for any system, being able to quickly make assessments about good strategies is important. After all, one recent best-seller has argued that such intuition is an important attribute of brilliant decision makers (M. Gladwell, Blink: The Power of Thinking without Thinking, Back Bay, 2007).

When done by architects who truly have the experience, such quick and intuitive assessments can be valuable in focusing efforts on the path with the most likely chance of success. The difficulty comes when cognitive biases come into play and lead to this ideal being applied in circumstances in which it has no business.

How do we recognize when we’re stumbling into the negative “blink” game? Perhaps by paying attention to how much the architects are really listening to the requirements folks. Do requirements analysts get to articulate what they’re looking to do with the system before they start hearing about pulldown menus, Apache servers, and other detailed solution elements?

analysis Paralysis

Some of Kruchten’s games spring not from a lack of rationality but from being too rational. For example, the opposite of “blink” is over-focusing on analysis to the detriment of actually getting things done. As with the other games, it’s hard to argue with the desire to do a thorough and rigorous analysis.

But too often, we might simply be afraid of making decisions, so continued analysis becomes a convenient decision avoidance strategy. For architects, this game can also take the form of a million questions and requests for clarification to the requirements team. While it can always be healthy to challenge requirements, extreme forms of this effectively push off any responsibility for engineering decisions to other stakeholders. In either case, while you’ve been trying to know everything about everything, you’re potentially blocking progress on other fronts as well.

Kruchten’s response to how to mitigate this tendency is to come to terms with the fact that we can’t make perfect decisions. On almost any development project, we have to make progress, even if it’s imperfect. Rather than aiming for perfection, it’s often healthier instead to aim to have some kind of traceback that lets the team recover if it reaches a bad state.

Valores de ingeniería: de los juegos de arquitectura a los requisitos ágiles

"Los valores a los que las personas se aferran más obstinadamente en condiciones inapropiadas son aquellos valores que antes eran la fuente de sus mayores triunfos". —Jared Diamond, Colapso: cómo las sociedades eligen fracasar o tener éxito

Los principios y valores personales son importantes. Articular las cualidades personales que valoramos nos permite establecer metas para nosotros mismos y reflexionar sobre si estamos cumpliendo con nuestros propios estándares, o si estamos tan atrapados en la rutina diaria que hacemos una Mil pequeños compromisos y nunca se dan cuenta.

De manera similar, los principios profesionales son una piedra de toque que nos permite apuntar al rigor en nuestro trabajo en lugar de gastar nuestros esfuerzos en miles de compromisos y confusiones diferentes. Tal vez tales principios de ingeniería sean aún más importantes en las fases iniciales de cualquier ciclo de vida del software, cuando la libertad para interpretar el problema y las soluciones de diseño casi no tienen límites.

Y, sin embargo, cuando se lleva al extremo o se aplica a ciegas, cualquier principio puede producir los mismos resultados que debía evitar, ahogando la creatividad y conduciendo a malas decisiones. Philippe Kruchten me recordó este punto durante su reciente presentación y la entrevista personal que siguió al enunciar algunos valores importantes y sus posibles abusos en los requisitos de software y la arquitectura.

El trabajo de Kruchten se ha centrado a menudo en la arquitectura del software, y en su trabajo de consultoría, ha observado una gran cantidad de equipos y su dinámica en funcionamiento. Cuando lo alcancé en la conferencia Better Software East, él acababa de dar una nota clave que resume algunos de los patrones de comportamiento negativos que a menudo ha visto en el campo (se refirió a ellos como "juegos que juegan los arquitectos"). Me sorprendió ver cuántos de estos sonaban inicialmente como principios de ingeniería positivos llevados a extremos, lo que parece ser una historia de advertencia sobre el potencial de las reglas para integrarse fuera de contexto.

Para Kruchten, una de las causas fundamentales de estos juegos es que, al igual que otros humanos, los profesionales del software pueden confundirse y pensar que somos más racionales que nosotros. Los sesgos cognitivos entran en juego durante el desarrollo del software al igual que en otras actividades. Por ejemplo, con demasiada frecuencia caemos rápidamente en el sesgo de confirmación, encontrando razones válidas para apoyar las cosas que ya creemos. También advirtió que no debemos poner la responsabilidad de tal comportamiento solo en los arquitectos: el desarrollo de software es un gran continuo de decisiones, y los mismos tipos de sesgos pueden entrar en juego en cada uno. Mirando antes en el ciclo de vida, nos engañaríamos a nosotros mismos para pensar que los usuarios y los analistas de requisitos son perfectamente racionales, y son solo los "tipos de software" los que malinterpretan las cosas y salen en direcciones extrañas.

El proyecto piloto

Comencemos con un juego que Kruchten identificó como un empirista cercano a mi corazón: el proyecto piloto. Hay un principio sólido en el corazón de este, a saber, el deseo de ser intencional y cauteloso con las nuevas herramientas y tecnologías. Un componente clave para hacer que la ingeniería de software sea un campo impulsado por la evidencia en lugar de uno al capricho de las modas, es probar enfoques novedosos en entornos a pequeña escala pero realistas para ver si los beneficios reclamados realmente se pueden obtener en el contexto de uno (ya qué precio) . Por sorprendente que parezca, he visto cómo las organizaciones se salen de los rieles al exigir la adopción generalizada de nuevas tecnologías que resultan más complicadas en la práctica de lo que se pensaba inicialmente.

Sin embargo, el "juego" comienza cuando las organizaciones entran en pilotos con una idea preconcebida del resultado que quieren ver y eliminan cualquier cosa que pudiera ser un obstáculo para obtener ese resultado predefinido, incluso si esos obstáculos están realmente relacionados con el forma en que normalmente hacen negocios. Entonces, ¿cómo nos vigilamos a nosotros mismos y entendemos si el sesgo de confirmación está realmente en juego? Una heurística es contar la frecuencia con la que escuchamos, "Pero no estamos haciendo [algo] en este proyecto porque esto es solo un piloto". Se ha convertido "simplemente un piloto" en el mantra que debemos cantar al eliminar cualquier cosa que pueda Posiblemente obtenga un resultado que no sea nuestro preferido? También es útil considerar si hemos elegido a un equipo que es excepcionalmente bueno y podría haber obtenido un resultado exitoso independientemente de la tecnología o el proceso que apliquen.

Parpadeo

Otro ejemplo que dio Kruchten es el "parpadeo", en el cual nos decimos que tenemos experiencia y conocimientos suficientes y que podemos tomar decisiones rápidamente, basadas en la experiencia y la intuición. Dado que los arquitectos generalmente enfrentan una amplia gama de posibles soluciones para cualquier sistema, es importante poder realizar evaluaciones rápidamente sobre buenas estrategias. Después de todo, un éxito de ventas reciente ha argumentado que tal intuición es un atributo importante de los tomadores de decisiones brillantes (M. Gladwell, Blink: El poder de pensar sin pensar, Back Bay, 2007).

Cuando son realizados por arquitectos que verdaderamente tienen la experiencia, estas evaluaciones rápidas e intuitivas pueden ser valiosas para enfocar los esfuerzos en el camino con la mayor probabilidad de éxito. La dificultad viene cuando los sesgos cognitivos entran en juego y conducen a que este ideal se aplique en circunstancias en las que no tiene negocio.

¿Cómo reconocemos cuando estamos tropezando con el juego negativo de "parpadeo"? Tal vez prestando atención a lo mucho que los arquitectos realmente están escuchando a los requisitos de la gente. ¿Los analistas de requisitos pueden articular lo que buscan hacer con el sistema antes de comenzar a escuchar sobre los menús desplegables, los servidores de Apache y otros elementos de solución detallados?

análisis parálisis

Algunos de los juegos de Kruchten no provienen de una falta de racionalidad sino de ser demasiado racionales. Por ejemplo, lo opuesto a "parpadear" es centrarse demasiado en el análisis en detrimento de realmente hacer las cosas. Al igual que con los otros juegos, es difícil discutir con el deseo de hacer un análisis exhaustivo y riguroso.

Pero con demasiada frecuencia, es posible que simplemente tengamos miedo de tomar decisiones, por lo que el análisis continuo se convierte en una estrategia conveniente para evitar las decisiones. Para los arquitectos, este juego también puede tomar la forma de un millón de preguntas y solicitudes de aclaración al equipo de requisitos. Si bien siempre puede ser saludable desafiar los requisitos, las formas extremas de esto eliminan de manera efectiva la responsabilidad de las decisiones de ingeniería a otras partes interesadas. En cualquier caso, mientras intentas saber todo sobre todo, también estás bloqueando el progreso en otros frentes.

La respuesta de Kruchten a cómo mitigar esta tendencia es llegar a un acuerdo con el hecho de que no podemos tomar decisiones perfectas. En casi cualquier proyecto de desarrollo, tenemos que avanzar, incluso si es imperfecto. En lugar de apuntar a la perfección, a menudo es más saludable buscar un tipo de rastreo que permita que el equipo se recupere si llega a un estado malo.

# Peaceful Coexistence: Agile Developer Perspectives on Software Architecture

The Agile Manifesto will be 10 years old in 2011. Agile methods are widespread in the software industry today, and a rapprochement between the agile and architecture-centric development communities has emerged. Nevertheless, some tension persists between these communities. To help separate facts from myths about the potential coexistence of agile development and software architecture, we conducted an exploratory study of experienced practitioners at a large company—specifically, the IBM Software Lab in Rome. By identifying and understanding the advantages and problems these professionals perceived, we aimed to facilitate the practical integration of these development approaches and illuminate future research directions. The participants had 18 years of developer experience on average, and most had already adopted agile approaches.

Although qualitative data is generally more suitable than quantitative data in exploratory studies, we developed a survey to capture quantitative data by first conducting focus groups in the lab and then synthesizing the comments into a survey that would capture quantitative data. We pilottested the initial survey and administered the final questions to 72 professional IBM developers.

Relevance of Software Architecture Uses

We wanted to know whether agile developers considered the use of software architecture relevant to their work, so our first question was, “In the context of agile development, how relevant is each of the following uses of software architecture?” Table 1 shows the list of uses, which we adopted from the ISO/IEC WD4 42010 (IEEE P42010/D6) standard for systems and software engineering architecture descriptions. The participating developers ranked the relevance level of each use from 0 (no relevance) to 3 (extremely relevant).

We calculated the results shown in Table 1 by averaging the scores for each use. The uses are listed in descending order according to their relevance rank. The results show that 13 of 17 uses rank higher than the scoring midpoint of 1.5. In other words, only three of 17 software architecture uses are more irrelevant than relevant to agile practice. We can conclude that the participants considered software architecture relevant in the context of agile development.

When to Focus on Software Architecture

As Grady Booch said, “You don’t need architecture to build a dog kennel, but you’d better have some for a skyscraper.” Along this line, our second survey question was, “In the context of agile development, when should you focus on software architecture?” Optional answers were “always,” “never,” and “when the project is complex.” Because complexity is a broad term, we asked respondents who selected it to choose geographic distribution, number of requirements or lines of code, number of stakeholders, and “other” as the leading cause of complexity.

As Figure 1a shows, half the respondents selected project complexity as a reason to focus agile development on software architecture. Figure 1b reports the percentage results characterizing this complexity. In particular, practitioners perceived the number of requirements or lines of code as the leading indicator of project complexity that requires a focus on software architecture, followed closely by the number of stakeholders.

Agile Values and Architecture-Centric Principles

We wanted to characterize the relationships between agile values and architecture-centric principles. We distilled three main principles of architecture-centric methods:

■driven by nonfunctional requirements,

■requiring an upfront investment, and

■forcing software architecture compliance.

Participants characterized the relationships among all the combinations of these principles with the four values of the Agile Manifesto (http://agilemanifesto.org).

For the 12 combinations of agile values and architecture-centric principles, Figure 2 shows the distribution of relationships perceived as the most supportive, the most contrastive, and the overall average. According to Figure 2, the principles of architecture-centric method are, on the average, supportive (rather than contrastive or neutral) to agile values.

Other Results

Further results showed that a large majority of agile developers saw a need for new methods and special training to integrate architectural practices such as software architecture analysis, design, review—into agile approaches. Because most of them indicated a supportive relationship between the approaches in terms of values and principles, we assert that the main problem in combining agile and architecture-centric methods resides not in theoretical issues but in practical matters of adoption.

Our results also showed that agile developers significantly agreed on the value of architectural design patterns for integrating architectural practice into agile methods.

Finally, nonagile developers appeared to be pessimistic compared to agile developers. In particular, the former overestimated the contrasts in agile and architectural approaches.

Our results show that agile developers perceived software architectures as important and supportive to agile values, as opposed to neutral or contrastive. This kind of positive perception bodes well for future efforts to integrate agile and architecture practices.

# Coexistencia pacífica: Perspectivas del desarrollador ágil sobre arquitectura de software

El Manifiesto Agile tendrá 10 años en 2011. Los métodos ágiles están muy extendidos en la industria del software hoy en día, y ha surgido un acercamiento entre las comunidades de desarrollo ágil y centradas en la arquitectura. Sin embargo, persiste cierta tensión entre estas comunidades. Para ayudar a separar los hechos de los mitos sobre la posible coexistencia del desarrollo ágil y la arquitectura de software, realizamos un estudio exploratorio de profesionales experimentados en una gran empresa, específicamente, el IBM Software Lab en Roma. Al identificar y comprender las ventajas y los problemas que percibían estos profesionales, nuestro objetivo era facilitar la integración práctica de estos enfoques de desarrollo e iluminar las futuras direcciones de investigación. Los participantes tenían 18 años de experiencia como desarrollador en promedio, y la mayoría ya había adoptado enfoques ágiles.

Si bien los datos cualitativos son generalmente más adecuados que los datos cuantitativos en los estudios exploratorios, desarrollamos una encuesta para capturar datos cuantitativos mediante la realización de grupos focales en el laboratorio y luego sintetizando los comentarios en una encuesta que capturaría datos cuantitativos. Probamos la encuesta inicial y administramos las preguntas finales a 72 desarrolladores profesionales de IBM.

Relevancia de los usos de la arquitectura de software

Queríamos saber si los desarrolladores ágiles consideraban el uso de la arquitectura del software relevante para su trabajo, por lo que nuestra primera pregunta fue: "En el contexto del desarrollo ágil, ¿qué tan relevante es cada uno de los siguientes usos de la arquitectura del software?" La Tabla 1 muestra la lista de usos, que adoptamos del estándar ISO / IEC WD4 42010 (IEEE P42010 / D6) para descripciones de arquitectura de ingeniería de sistemas y software. Los desarrolladores participantes calificaron el nivel de relevancia de cada uso de 0 (no relevante) a 3 (extremadamente relevante).

Calculamos los resultados mostrados en la Tabla 1 al promediar los puntajes para cada uso. Los usos se enumeran en orden descendente según su rango de relevancia. Los resultados muestran que 13 de 17 usos tienen un rango superior al punto medio de puntuación de 1.5. En otras palabras, solo tres de los 17 usos de la arquitectura de software son más irrelevantes que relevantes para la práctica ágil. Podemos concluir que los participantes consideraron la arquitectura de software relevante en el contexto del desarrollo ágil.

Cuándo centrarse en la arquitectura de software

Como dijo Grady Booch, “No necesitas arquitectura para construir una perrera, pero es mejor que tengas algo para un rascacielos”. En esta línea, nuestra segunda pregunta de la encuesta fue: “En el contexto del desarrollo ágil, ¿cuándo debería? ¿Se enfoca en la arquitectura del software? "Las respuestas opcionales fueron" siempre "," nunca "y" cuando el proyecto es complejo ". Debido a que la complejidad es un término amplio, les pedimos a los encuestados que lo seleccionaron que eligieran la distribución geográfica, el número de requisitos o las líneas. de código, número de partes interesadas y "otros" como la causa principal de complejidad.

Como muestra la Figura 1a, la mitad de los encuestados seleccionó la complejidad del proyecto como una razón para enfocar el desarrollo ágil en la arquitectura del software. La Figura 1b reporta el porcentaje de resultados que caracterizan esta complejidad. En particular, los profesionales percibieron el número de requisitos o líneas de código como el indicador principal de la complejidad del proyecto que requiere un enfoque en la arquitectura del software, seguido de cerca por el número de partes interesadas.

Valores ágiles y principios centrados en la arquitectura

Queríamos caracterizar las relaciones entre los valores ágiles y los principios centrados en la arquitectura. Destilamos tres principios fundamentales de los métodos centrados en la arquitectura:

■ impulsado por requisitos no funcionales,

■ exigir una inversión inicial, y

■ forzando el cumplimiento de la arquitectura del software.

Los participantes caracterizaron las relaciones entre todas las combinaciones de estos principios con los cuatro valores del Manifiesto Ágil (http://agilemanifesto.org).

Para las 12 combinaciones de valores ágiles y principios centrados en la arquitectura, la Figura 2 muestra la distribución de las relaciones percibidas como las más favorables, las más contrastantes y el promedio general. De acuerdo con la Figura 2, los principios del método centrado en la arquitectura son, en promedio, favorables (en lugar de contrastivos o neutros) a los valores ágiles.

Otros resultados

Los resultados adicionales mostraron que la gran mayoría de los desarrolladores ágiles vieron la necesidad de nuevos métodos y capacitación especial para integrar las prácticas arquitectónicas, como el análisis de la arquitectura del software, el diseño y la revisión, en enfoques ágiles. Debido a que la mayoría de ellos indicaron una relación de apoyo entre los enfoques en términos de valores y principios, afirmamos que el problema principal en la combinación de métodos ágiles y centrados en la arquitectura no reside en cuestiones teóricas sino en cuestiones prácticas de adopción.

Nuestros resultados también mostraron que los desarrolladores ágiles acordaron significativamente el valor de los patrones de diseño arquitectónico para integrar la práctica arquitectónica en métodos ágiles.

Finalmente, los desarrolladores no ágiles parecían ser pesimistas en comparación con los desarrolladores ágiles. En particular, los primeros sobreestimaron los contrastes en los enfoques ágiles y arquitectónicos.

Nuestros resultados muestran que los desarrolladores ágiles percibían las arquitecturas de software como importantes y de apoyo para los valores ágiles, en lugar de neutros o contrastivos. Este tipo de percepción positiva es un buen augurio para futuros esfuerzos por integrar prácticas ágiles y de arquitectura.