**Architecture document**

Automatically supplement missing data



|  |
| --- |
| **Date : 30-03-2016** |
| **Version : 0.1** |
| **Status : Concept** |
| **Document name : Architecture document v0\_1** |
| **Author : Tim Hermens** |

**Version**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date** | **Author(s)** | **Changes** | **Status** |
| 0.1 | 30-03-2016 | Tim Hermens | First concept. | Concept |

**Distribution**

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Recipient** |
|  |  |  |
|  |  |  |

**Approval**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date of approval** | **Name** | **Signature** |
|  |  |  |  |

**Glossary**

**<Word>**

<Explanation>**Table of contents**

1. Introduction 5

1.1 Purpose 5

1.2 Scope 5

1.3 References 5

1.4 Document overview 5

1.5 Stakeholder representation 5

2. Architecture background 6

2.1 Problem background 6

2.2 Solution background 6

3. Architectural representation 7

3.1 Abstract context overview 7

3.2 Abstract component overview 7

3.3 Detailed component overview 7

3.4 Detailed implementation overview 7

4. View points 8

4.1 Use case view 8

4.2 Logical view 8

4.3 Implementation view 8

4.4 Deployment view 8

# Introduction

## Purpose

This software architecture document provides a comprehensive architectural overview of the system that consists of the analysis tool product (to analyze algorithms with) and the algorithm product (to predict missing skills in profiles). It presents a number of different architectural diagrams to depict different aspects of the system. It's intended to capture and convey the significant architectural decisions which have been made on the system. Decisions are considered architectural decision when they involve improving or discarding quality attributes that are of importance for the system.

*< Explain the SAD’s overall purpose and scope. Explain the criteria for deciding which design decisions are architectural (and therefore documented in the SAD) and which are non-architectural (and therefore documented elsewhere).>*

## Scope

The scope of this software architecture document is to depict the architecture of the system that consists of the analysis tool and the algorithm.

## References

|  |  |
| --- | --- |
| **Reference code** | **Reference** |
|  |  |

## Document overview

In order to fully document all of the aspects of the architecture, the software architecture document contains the following sections:

* Section 2: describes the function and purpose of the system and which architecture is chosen out of a selection of most interesting architectures;
* Section 3: describes the composition of the system at four different levels, from high to low;
* Section 4: describes architectural views based on the 4+1 model of Kruchten of the system. It starts with the use case view, describing the functional requirements with a significant impact on the architecture. Followed by the logical view, describing the different layers of the system. Followed by the implementation view, describing the technical implementation of the layers of the system which were discussed in the logical view. And finally the deployment view, describing the mapping of the system onto the hardware and showing the system's distributed aspects.

## Stakeholder representation

TBD

*<Provide a list of the stakeholder roles considered in the development of the architecture described by this SAD. For each role, list the stakeholder concerns that can be addressed by the information in this SAD. A convenient way to represent this information is as a matrix, where the rows list stakeholder roles, the columns list concerns, and the cells indicate how serious the concern is to a stakeholder in that role. The following stakeholders shall be considered at a minimum:*

*• application software developers*

*• infrastructure software developers*

*• end users*

*• project segment teams 6 CMU/SEI-2005-TN-020*

*• application system engineers*

*• application and platform hardware engineers*

*• security engineers and certifiers*

*• safety engineers and certifiers*

*• communications engineers*

*• system-of-system engineers*

*• chief engineer/chief scientist*

*• lead system integrator (LSI) program management*

*• government program management (including those concerned with licensing)*

*• system integration and test engineers*

*• external test agencies*

*• operational system managers*

*• trainers*

*• maintainers*

*• other service representatives*

*• auditors (LSI internal, the Government Accounting Office, etc.)*

*• representatives of standardization activities*

# Architecture background

## Problem background

*<In this section, explain the constraints that significantly influenced the architecture. Structure the information in the following sections:>*

### System overview

The project system consists of two products: product B1 and product B2.

Product B1, the server-side algorithm functionality, serves as an extension for the current profile matching algorithm, which aims to improve the overall profile matching results by adding missing [KSC](#KSC) data to the profiles. This product will be integrated in the server's Analysis framework.

Product B2, the algorithm analysis tool, is a new self-contained product. This product should work without a internet connection and doesn't have any interconnection with other existing products or external databases.

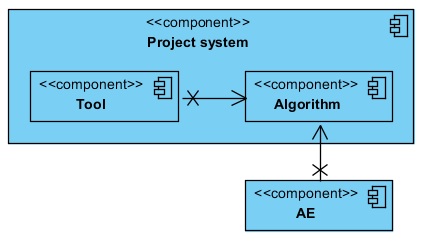


Figure 1 - Context view of the system

In Figure 1, you can see a context view of the system. The purpose of this context view is to provide an overall overview of other systems the project system communicates with. The tool and algorithm are part of the project system. The Analysis Environment (AE) is a system that's currently being used by the company 8vance to scrape and process profile data. The tool as well as the AE will use/communicate with the algorithm.

*<Describe the general function and purpose of the system or subsystem whose architecture is described in this SAD. If appropriate, include a context diagram showing the system or subsystem, and other systems or subsystems with which it communicates or interoperates.>*

### Goals and context

The goal of this software architecture is to ensure the longevity of the system by considering and integrating important quality attributes like scalability. This document can be used as a reference guideline when the system lacks performance and needs to be scaled horizontally.

The architecture serves as the foundation of the system. It defines a structured solution to meet all the technical and operational requirements, while optimizing the quality attributes. Further, it involves a set of significant decisions related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the system. The architecture is designed as such that it incorporates all the fundamental quality attributes. This ensures the foundation meets the non-functional requirements, meaning the architecture can never be the limiting factor of the system.

*<Describe the goals and major contextual factors for the software architecture. Include a description of the role software architecture plays in the life cycle, relevant acquisition factors, the impact of the LSI model, the effects of incremental development, and the relationship to system engineering results and artifacts.>*

### Significant driving requirements

The software architecture is shaped by several quality attributes taken from the ISO 25010 model. This model considers the following quality attribute groups: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. The following table contains the important quality attributes for the system, sorted from most important to least important.

|  |  |  |
| --- | --- | --- |
| **Quality type** | **Description** | **Importance** |
| Performance | The algorithm has to be able to predict missing skills of one profile in 1 second or less. Eventually, the algorithm has to make predictions for millions of profiles and the scraping of profile data will be a continuous process, meaning the algorithm needs to be able to keep up. Additionally, users can provide profile data themselves and the algorithm should be able to predict the missing skills in an instant. | 5 |
| Performance | The system is able to predict missing skills of 1 million profiles in 24 hours. | 5 |
| Adaptability/  scalability | The algorithm must be adaptable in different or evolving software products that support Python 2.7. Additionally, the system must support hardware scalability, meaning multiple servers could be introduced that are used to make predictions with the algorithm to surpass the 1 million profiles per day prediction goal. | 5 |
| Interoperability | The algorithm and 8vance's Analysis environment as well as the algorithm and analysis tool must be able to exchange information with each other. | 5 |
| Availability | The algorithm must be available to be used to create predictions at all times. | 5 |
| Testability | The analysis tool will mainly be used to test the quality and performance of an algorithm as well as the performance of the product itself. Multiple test criteria can be established that should measure the quality and performance of an algorithm. The input and output data of an algorithm can also be tested on correctness. | 4 |
| Correctness | The algorithm must be able to calculate certainty scores for every prediction with great precision to identify whether or not predictions are correct. | 4 |
| Modifiability | Both products must be easily modifiable to improve the product quality. | 4 |
| Reusability | The algorithm must be usable in both 8vance's Analysis environment as well as in the analysis tool. | 4 |
| Installability | The analysis tool must be installable and thus usable on any operating system. | 4 |
| Modularity | Both products are composed of discrete components. When one component is changed, it should have a minimal impact on the other components. This allows for more effective changes to the system. | 4 |
| Recoverability | In the event of an interruption or a failure of the algorithm product, the system can recover the trained algorithm so that the algorithm product doesn't have to retrain the algorithm. In case of a failed connection to one server the algorithm product is running on, another server can be connected to if available. | 3 |
| Safety | In the analysis tool, whenever the user wants to select/specify a new algorithm, data source, dump target, trained algorithm, or start an analysis process, kill an analysis process or import an analysis result, the tool must query the user if he's certain he wants to continue to prevent the loss of previous data. | 2 |
| Safety | In the analysis tool, the user is able to kill an analysis process. When the user starts an analysis process, the user can't perform any actions in the application except from killing the analysis process. If there's a problem with the analysis resulting in the process to never end, that means the user could never perform any actions. By providing the user with the option to kill an analysis process, there's a safeguard to prevent this problem. | 2 |
| Learnability | The analysis tool must be easy to use. Apart from algorithm developers, users without any understanding of the inner workings of the algorithm and the tool should be able to use and understand the intended use of the tool. | 1 |

Table 1 - Important quality attributes of the system, sorted from most to least important.

When choosing the architecture of the system, it's important that the base architecture covers as many of these important quality attributes as possible. If any important quality attributes remain, other sub architectures or design patterns should be chosen to cover those as well.

*<Describe behavioral and quality attribute requirements (original or derived) that shaped the software architecture. Include any scenarios that express driving behavioral and quality attribute goals, such as those crafted during an ATAM evaluation [Clements 01].>*

## Solution background

*<In this section, provide a description of why the architecture is the way it is and why it is appropriate for satisfying the functional and quality attribute goals levied upon it. Structure the information in the following sections:>*

### Architectural approaches

As said before, it's important that the system's base architecture covers as many of the important quality attributes mentioned in Table 1 as possible. Any other important remaining quality attributes should be covered by choosing other sub architectures or design patterns.

The architectural designs or patterns that are discussed in this section were chosen based on the quality attributes they offer. The most interesting architectural designs or patterns that cover this system's demanded quality attributes are discussed in this section.

#### Pipe and filter

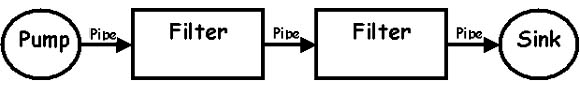


Figure 2 - Pipe and filter pattern

This architectural pattern lays emphasis on the incremental transformation of data by successive component. The connections between modules are data stream which is first-in/first-out buffer that can be stream of bytes, characters, or any other type of such kind. The main feature of this architecture is its concurrent and incremented execution.

##### Advantages

Pipe and filter has the following advantages:

* Provides concurrency and high throughput for excessive data processing.
* Provides reusability and simplifies system maintenance.
* Provides modifiability and low coupling between filters.
* Provides simplicity by offering clear divisions between any two filters connected by pipe.
* Provides flexibility by supporting both sequential and parallel execution.

##### Disadvantages

Pipe and filter has the following disadvantages:

* Not suitable for dynamic interactions.
* Overhead of data transformation between filters.
* Does not provide a way for filters to cooperatively interact to solve a problem.
* Difficult to configure this architecture dynamically.

##### Quality attributes coverage

This architecture design ensures the *performance* quality attribute. The pipe and filter pattern makes use of parallelism which provides concurrency and high throughput, resulting in great performance. However, if every filter uses a different data structure which requires a continuous transformation of data, it could negatively impact the performance. (http://portal.ou.nl/documents/114964/2986739/IM0203\_03.pdf, page 53)

This architecture design ensures the *adaptability* and *modifiability* quality attributes. Filters can be added and replaced easily. This is because of the standard interface that's required for the filters to use. (http://portal.ou.nl/documents/114964/2986739/IM0203\_03.pdf, page 43)

This architecture design ensures the *reusability* quality attribute. It's possible to build different pipelines by recombining a given set of filters. Filters are stand-alone and can be treated as black boxes. This isolation of functionality ensures modifiability and reusability. (http://portal.ou.nl/documents/114964/2986739/IM0203\_03.pdf, page 43; http://www4.desales.edu/~dlm1/it533/class03/pipe.html)

This architecture design ensures the *modularity* quality attribute to a certain point. Filters do not know anything about other filters. To increase modularity, object-oriented frameworks can be developed that allow stages to be represented by objects or procedures that can easily be used by the pipeline. Such frameworks aren't difficult to construct using standard OOP techniques. (http://www.informit.com/articles/article.aspx?p=366887&seqNum=8)

This architecture design ensures the *testability* quality attribute. Analysis/testing of the pipe and filter system is easy, because it's a simple composition of the behaviours of the filters involved. When the input is called x, the behaviour of the first filter is described by function *g*, and the behaviour of the second filter is described by function *f*, the result of the pipeline can be described as: *f(g(x))*. Because of this composition, it's possible to analyze/test throughput as well (determined by slowest filter). http://portal.ou.nl/documents/114964/2986739/IM0203\_03.pdf, page 43)

##### Analysis

This architecture design is particularly useful for pre- and post-processing the profile data, and executing the algorithm (predictions). These steps can be seen as separate filters which are linked together and executed in a particular order. However, having the flexibility to switch the filters isn't particularly useful for this system because there simply won't be a need to switch them. The pipeline always have to start with pre-processing the data and stop with post-processing the data. The filters in between call the algorithm functionality to predict the missing data. This order of filters won't ever change, so the strengths of this architecture design like modifiability and reusability don't count. This means the strength of this design currently lies in the performance, testability, and to a certain point, modularity.

#### Layered architecture

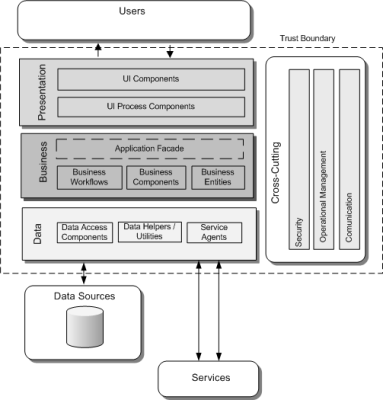


Figure - Layered architecture

Layered architecture focuses on the grouping of related functionality within an application into distinct layers that are stacked vertically on top of each other. Functionality within each layer is related by a common role or responsibility. Communication between layers is explicit and loosely coupled.

The layers of an application may reside on the same physical computer (the same tier) or may be distributed over separate computers (n-tier), and the components in each layer communicate with components in other layers through well-defined interfaces. For example, a typical layered architecture consists of a presentation layer (functionality related to the UI), a business layer (business rules processing), and a data layer (functionality related to data access, often almost entirely implemented using high-level data access frameworks).

##### Advantages

The layered architecture has the following advantages:

* Design based on incremental levels of abstraction.
* Provides enhancement independence as changes to the function of one layer affects at most two other layers.
* Separation of the standard interface and its implementation.
* Implemented by using component-based technology which makes the system much easier to allow for plug-and-play of new components.
* Each layer can be an abstract machine deployed independently which support portability.
* Easy to decompose the system based on the definition of the tasks in a top-down refinement manner.
* Different implementations (with identical interfaces) of the same layer can be used interchangeably.

##### Disadvantages

The layered architecture has the following disadvantages:

* Lower runtime performance since a client’s request or a response to client must go through potentially several layers.
* There are also performance concerns on overhead on the data marshalling and buffering by each layer.
* Opening of interlayer communication may cause deadlocks and “bridging” may cause tight coupling.
* Exceptions and error handling is an issue in the layered architecture, since faults in one layer must spread upwards to all calling layers.

##### Quality attributes coverage

This architecture design ensures the *reusability* quality attribute. For example, future presentation layers can reuse the business and data access layers because the lower layers have no dependencies on the higher layers.

This architecture design ensures the *manageability* quality attribute. The separation of core concerns helps to identify dependencies, and organizes the code into manageable sections.

This architecture design ensures the *performance* and *adaptability* quality attributes. Distributing the layers over multiple physical tiers can improve the scalability and performance.

This architecture design ensures the *testability* quality attribute. Having well-defined layer interfaces increases the testability, as well as the ability to switch between different implementations of the layer interfaces. Separated Presentation patterns can be considered to improve the testability by building mock objects that mimic the behaviour of concrete objects such as the model, controller or view.

This architecture design ensures the *modularity* quality attribute. The layered architecture is implemented by using component-based technology which makes the system much easier to allow for plug-and-play of new components.

##### Analysis

The system involves clear divisions between core services, critical services and user interface services. Performance can be a concern if there're a lot of layers information must go through, but that won't be the case for this system.

So the strength of this design lies in the reusability, manageability, performance, adaptability, testability and modularity.

#### Blackboard

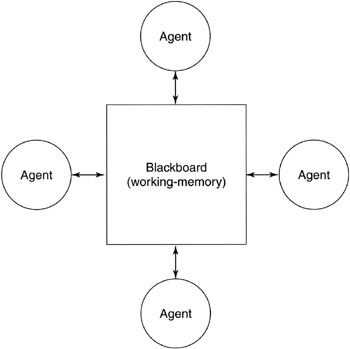


Figure 4 - Blackboard architecture

In the blackboard architecture, the data store is active and its clients are passive. Therefore the logical flow is determined by the current data status in data store. It has a blackboard component, acting as a central data repository, and an internal representation is built and acted upon by different computational elements.

Further, a number of components that act independently on the common data structure are stored in the blackboard. In this style, the components interact only through the blackboard. The data-store alerts the clients whenever there is a data-store changes. The current state of the solution is stored in the blackboard and processing is triggered by the state of the blackboard.

A major difference with traditional database systems is that the invocation of computational elements in a blackboard architecture is triggered by the current state of the blackboard, and not by external inputs.

The blackboard model uses so-called Knowledge Sources (KS) to solve parts of a problem and aggregate partial results. Interaction among knowledge sources takes place exclusively through the blackboard.

##### Advantages

The blackboard architecture has the following advantages:

* Blackboard Model provides concurrency that allows all knowledge sources to work in parallel as they are independent of each other.
* Its scalability feature facilitates easy steps to add or update knowledge source.
* It supports experimentation for hypotheses and reusability of knowledge source agents.

##### Disadvantages

The blackboard architecture has the following disadvantages:

* The structural change of blackboard may have a significant impact on all of its agents, as close dependency exists between blackboard and knowledge source.
* Blackboard model is expected to produce approximate solution; however, sometimes, it becomes difficult to decide when to terminate the reasoning.
* This model suffers some problems in synchronization of multiple agents, therefore, it faces challenge in designing and testing of the system.

##### Quality attribute coverage

This architecture design ensures the *scalability* quality attribute. Knowledge sources can work in parallel as they're independent of each other. More knowledge sources can be added or updated easily.

This architecture design ensures the *reusability* quality attribute. Knowledge sources can be reused as they do not have direct communication and thus dependencies with each other.

##### Analysis

This architectural design could be used to execute the algorithm functionality to create the predictions. However, the blackboard design is mostly effective when the knowledge sources are truly independent of each other, which isn't the case for this system. At least one knowledge source has to do the pre-processing part and the other knowledge sources that create predictions or post-process the data need to have this pre-processed data. So there's already a dependency here between those knowledge sources. This also means the architecture design loses scalability and performance as not every knowledge source can work in parallel.

Performance is not a strength of this architecture design because of the mentioned problem with not being able to run all the knowledge sources in parallel. Knowledge sources that require pre-processed data need to wait for the knowledge sources that deliver that data. And knowledge sources that want to post-process the data need to wait for the knowledge sources that deliver the predicted data. The knowledge sources that deliver the predicted data can be run in parallel, but that also means that the blackboard needs to synchronize the delivery of this data and the execution of the post-processing knowledge sources. This means the blackboard would be complex to build and manage, and can become the bottleneck of the system because of its complexity and responsibilities.

So for this system, the main strength of this architectural design lies only in its reusability.

#### MVC

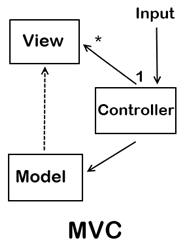


Figure - MVC design pattern

Model-View-Controller (MVC) decomposes a given software application into three interconnected parts that help in separating the internal representations of information from the information presented to or accepted from the user. The Model encapsulates the underlying data and business logic. The Controller responds to user actions and direct the application flow. The View formats and presents the data from model to user.

##### Advantages

MVC has the following advantages:

* Multiple views synchronized with same data model.
* Easy to plug-in new or replace interface views.

##### Disadvantages

MVC has the following disadvantages:

* Multiple pairs of controllers and views based on the same data model make any data model change expensive.
* The division between the View and the Controller is not clear in some cases.

##### Quality attribute coverage

This design pattern ensures the *reusability* and *maintainability* quality attributes. MVC separates the different aspects of a piece of software. This separation promotes code reusability and a more structured application architecture, improving the maintainability.

This design pattern ensures the *testability* and *modifiability* quality attributes. A small amount of well-defined responsibilities of the separated aspects of the software makes it easier to test and modify the code.

##### Analysis

The MVC design pattern can be useful for both the tool as well as the algorithm. The three different layers M, V and C lend themselves perfectly for the algorithm. Both the AE and tool want to execute the algorithm and expect similar input and different output interfaces. The View layer of MVC can be used to define this interface (one View for the AE, and one for the tool) and do the post-processing of the data. The Controller layer of MVC can be used to call the data from the Model in the right order to do the pre-processing, predictions and post-processing. The Model layer of MVC can be used to pre- and post-process the data and execute the algorithm's functionality like creating predictions.

Normally, there's one Controller per View in MVC. You actually want to have a general Controller that handles the post-processed data conversion for both Views. Additionally, it could be possible that multiple algorithms are used to create the predictions. This means the Model can consist of multiple algorithms.

Lastly, the input should be coming from the View instead of the Controller.

MVC can also be useful to decouple the UI from the rest of the system. This will improve the maintainability of the system.

#### Message Broker

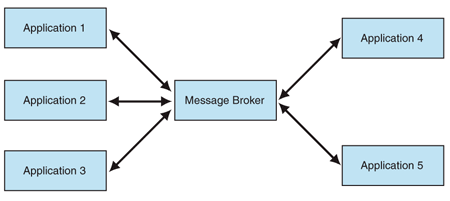


Figure 6 - Message broker pattern

A message broker is a physical component that handles the communication between applications. Applications communicate only with the message broker. The message broker looks up the registered applications and then passes the message to them.

The message broker can expose different interfaces to the collaborating applications, meaning it doesn't enforce a common interface.

Prior to using the message broker, applications must be registered so that the message broker can dispatch requests and responses to them.

##### Advantages

The message broker has the following advantages:

* Reduced coupling. The message broker decouples the senders and the receivers. Senders communicate only with the message broker.
* Improved integrability. The applications that communicate with the message broker do not need to have the same interface.
* Improved modifiability. The message broker shields the components of the integration solution from changes in individual applications.
* Improved security. Communication between applications involves only the sender, the broker, and the receivers. Other applications do not receive the messages that these three exchange. With the broker acting as the intermediary between the communicating parties, information needn't be encrypted.
* Improved testability. The message broker can easily be mocked. Mocking facilitates the testing of individual applications as well as of the interaction between them.

##### Disadvantages

The message broker has the following disadvantages:

* Increased complexity. The message broker must communicate with all the parties involved, meaning many interfaces and protocols could have to be supported. A multithreaded message broker makes it harder to handle exceptions.
* Increased maintenance effort. Broker-based integration requires that the integration solution register the applications with the broker.
* Reduced availability. A single broker means a single point of failure. A second broker could solve this problem. However, a secondary message broker adds the issues that are associated with synchronizing the states between the primary message broker and the secondary message broker.
* Reduced performance. With the message broker acting as the intermediary between the communicating parties, overhead is created.

##### Quality attribute coverage

This design pattern ensures the *scalability* and *adaptability* quality attributes. New applications can easily be added to the system by providing new interfaces and registering the applications to the broker.

This design pattern ensures the *modifiability* quality attribute. Changes to the broker and/or components of the integration solution have no effect on each other as long as there're no changes to the interface.

The design pattern ensures the *testability* quality attribute. The broker can be mocked, allowing the testing of individual applications as well as of the interaction between them.

##### Analysis

This design pattern offers a solution to enable the communication between components of different applications. These applications can be located on different physical environments. The algorithm will most likely be distributed on different physical environments to improve the performance of the system in the foreseeable future. This design pattern would be useful right then.

The main disadvantage of the message broker is the lacking performance. Overhead incurs because all communication need to go through the broker and no direct communication between sender and receiver is available.

However, there're variants of the message broker. The performance-optimized message broker called *Client-Dispatcher-Server* trades the ease of integration for performance. This broker looks up the receiver and then connects it to the sender, thus allowing the sender and the receiver to communicate directly. The direct connection eliminates the performance penalty that is associated with an intermediary between the communicating parties. However, this performance optimization only works if the sender and the receiver have the same interface.

#### Improving performance and availability

As performance is one of the most important quality attributes, it's desired to improve it as much as possible. Eventually, there should be a scalable infrastructure tier that accounts for changes in server load while maintaining an acceptable level of performance.

Individual servers have a maximum amount of load capacity for any given application. If the server load increases beyond the limitation of the server, the application will either fall below performance expectations or, in the worst case, become unavailable.

Individual servers have maximum physical performance limitations (e.g. the amount of memory, the number of processors). For example, if the server is capable of housing only four processors, you cannot add a fifth processor to enhance performance.

##### Load-balanced cluster

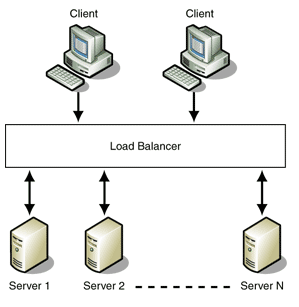


Figure - Load-balanced cluster

Install and use the algorithm library on multiple servers that are configured to share the workload. This type of configuration is a load-balanced cluster. Load balancing scales the performance of server-based programs by distributing client requests across multiple servers.

Load balancers use different algorithms to control traffic. The goal of these algorithms is to intelligently distribute load and/or maximize the utilization of all servers within the cluster. The most notable algorithms are as follows:

* Round-robin. The load is distributed equally to each server. Round-robin is suitable when the servers in the cluster have equal processing capabilities.
* Weighted round-robin. A performance weight is assigned to each server, meaning some server have a higher performance than others. This weight is taken into account when distributing the load to each server.
* Load-based. Requests are sent to the server with the currently lowest load.

The round-robin algorithm could be used when a lot of predictions need to be made. The load-based algorithm could be used when only a few predictions need to be made or when the algorithm needs to be (re)trained.

When the load balancer receives a request, one of the servers in the group processes the request. Every server is capable of handling the request independently. If any server is unavailable, other servers can still process requests without being affected. Thus, the overall availability and performance of the service is much higher than if a single server were serving all the requests. However, using a single physical load balancer or a single network switch in front of a set of software load-balanced servers introduces another single point failure.

##### Asymmetric server clustering

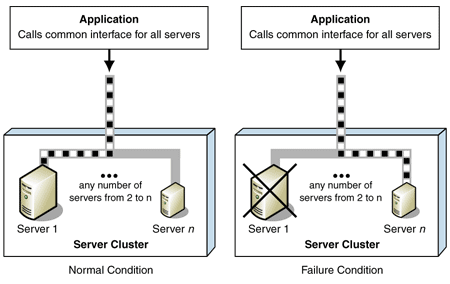


Figure 8 - Asymmetric server clustering

The previously mentioned *Load-balanced cluster* design pattern is also known as the *Symmetric server clustering* pattern.

In asymmetric server clustering, a standby server exists only to take over for another server in the event of a failure. If one of the servers in a cluster becomes unavailable, another node takes over the function of the failed server.

The standby server performs no other useful work and is either as capable as or less capable than a primary server. A less capable, less expensive standby server is often used when primary servers are configured for high availability. One common type of asymmetric clustering is known as a *failover clustering*.

##### Failover clustering

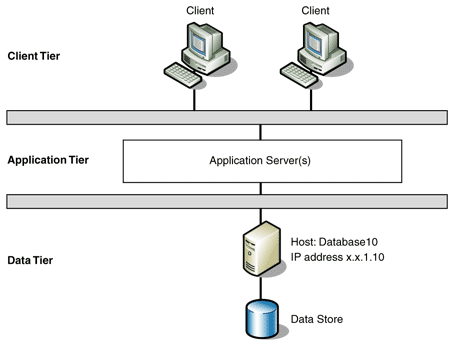


Figure - Failover clustering

The process of one server taking over for a failed server is commonly known as failover. A failover cluster is a set of servers that are configured in such a way that another server automatically takes over for a failed server and continues processing. Each server in the cluster has at least one other server in the cluster identified as its standby server.

The following mechanisms can be used to indicate a system failure:

* Push heartbeats. The active server sends specified signals to the standby server at a regular interval. If the standby server doesn't receive a signal over a certain time interval, it determines that the active server failed and takes the active role.
* Pull heartbeats. The standby server sends requests to the active server at regular intervals. If the active server doesn't respond to a pre-determined amount of requests, the standby server takes the active role.

Heartbeats are sent over dedicated communication channels so that no network problems occur which could cause failovers.

###### Synchronizing

Before the standby server can start processing transactions, it must synchronize its state with the state of the failed server. There are three different approaches to synchronization:

* Transaction log. The active server maintains a log of all changes to its state. Periodically, a synchronization utility processes this log to update the standby server's state to match the state of the active server. When the active server fails, the standby server must use the synchronization utility to process any additions to the transaction log since the last update. After the state is synchronized, the standby server becomes the active server and begins processing.
* Hot standby. Updates to the internal state of the active server are immediately copied to the standby server. The standby server can immediately become the active server and start processing transactions.
* Shared storage. Both servers maintain their state on a shared storage source. The standby server can immediately become the active server because no state synchronization is required.

Since the performance is one of the most important quality attributes, the latter approach is the most interesting one as it doesn't require continuous data synchronization between the servers.

*<Provide a rationale for the major design decisions embodied by the software architecture. Describe any design approaches applied to the software architecture— including the use of architectural styles or design patterns—when the scope of those approaches transcends any single architectural view. Explain why those approaches were chosen and specifically why they were chosen over other seriously considered approaches. Describe any relevant issues dealing with commercial off-the-shelf (COTS) or government off-the-shelf (GOTS) components, including any associated trade spaces.>*

### Analysis results

A variety of architectural styles and design patterns have been discussed in the previous section. Let's have a look at how well every architectural style and design pattern support the important architectural-related quality attributes.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Architectural style or design pattern** | **Performance** | **Adaptability/scalability** | **Testability** | **Modifiability** | **Reusability** | **Modularity** | **Availability** |
| Pipe and filter |  |  |  |  |  |  |  |
| Layered (n-tier) |  |  |  |  |  |  |  |
| Blackboard |  |  |  |  |  |  |  |
| MVC |  |  |  |  |  |  |  |
| Message broker |  |  |  |  |  |  |  |
| Client-Dispatcher-Server |  |  |  |  |  |  |  |
| Load-balanced cluster |  |  |  |  |  |  |  |
| Asymmetric server clustering |  |  |  |  |  |  |  |
| Failover clustering |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Great |  | Good |  | Average |  | Bad |  | Terrible |

Table 2 - Degree to which a architectural style or design pattern supports the important architectural-related quality attributes

The first four architectural styles are the styles that describe the foundation of the system. All quality attributes are of importance here. The next two styles describe a possibility for the communication between the distributed components. The performance quality attribute is of importance here. The last three styles can be used for the deployment of the system, which describe how the servers operate. Performance and availability are the most important quality attributes here.

The Layered N-tier architectural style combined with the Load-balanced cluster style seems to be the best to use for the foundation of this system. This combination ensures high performance and availability, which are two of the most important quality attributes. MVC could be used for both the tool (to decouple the UI from the code) and the algorithm library (read the [MVC section](#MVC) how it would help).

Lastly, the Client-Dispatcher-Server design pattern can be used to enable the communication between external components. The algorithm prediction part of the system is likely to be distributed over several servers. This pattern creates the possibility to communicate with these servers over a message oriented middleware.

*<Describe the results of any quantitative or qualitative analyses that have proven the software architecture is fit for purpose. If an architecture evaluation has been performed using the ATAM or a comparable method, include the analysis sections of the final evaluation report. Refer to the results of any other relevant trade studies, quantitative modeling, or other analysis results.>*

### Requirements coverage

All original requirements (specified in the requirements document) can be implemented with the chosen software architecture.

There're some requirements that are derived from the chosen architecture:

|  |  |
| --- | --- |
| **ID** | **Requirement description** |
|
|  | The system can distribute algorithm prediction requests over a server cluster. |
|  | The system can measure the load of every server in the server cluster. |
|  | The system can forward an algorithm train request to the server with the lowest load. |
|  | The system can send heartbeat messages to the servers in the server cluster to determine which servers are operational and which are not. |
|  | In the event of a server failure, the system can forward the requests that were sent to the failed server to another server. |
|  | The system logs every request that is sent to a server. |
|  | The system can indicate if a request to a server has been processed successfully. |

Table - Derived requirements

*<Describe the requirements (original or derived) addressed by the software architecture. Include those requirements or constraints that are derived from higher level SADs.>*

### Summary of changes in current version

No changes after the original release have been made yet as the release version has yet to be finished.

*<For versions of the SAD after the original release, summarize the actions; the decisions and decision drivers; the requirements changes and analysis and trade study results that became decision drivers; and explain how these decisions caused the architecture to evolve or change.>*

# View points

## Functional viewpoint

### Abstract context overview

In Figure 1 in section 2.1.1, the abstract context overview of the system has already been shown and discussed. Let's have a brief look at an abstract component overview variant of the system.

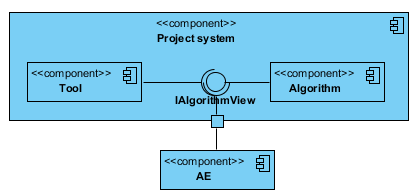


Figure 10 - Abstract component overview of the system

As can be seen in Figure 10, the system consists of two major components. The first one being an analysis Tool and the second one being the Algorithm library.

#### Tool

The analysis Tool will primarily be used to analyze the prediction results of algorithms and retrain them if deemed necessary.

#### Algorithm

The Algorithm library offers the functionality to create predictions with an algorithm and (re)train it.

##### IAlgorithmView

The Algorithm library implements this interface that exposes its operations (such as creating predictions and (re)training an algorithm) that can be called by other components, such as the Tool and AE.

#### AE

The Analysis Environment (AE) is a system that's currently being used by the company 8vance to scrape and process profile data.

*<High level overview of the communication between the systems.>*

### Concrete component overview

#### Tool component

In Figure 11, a concrete component overview of the Tool component can be seen. As can be seen, it's a layered architecture consisting of three layers.

##### Presentation layer

The Presentation layer basically contains a collection of GUIs the user interacts with. The user can trigger all kind of events by interacting with elements on the GUI. For instance, the user can click a button that triggers an event which will allow him to select an algorithm he wants to analyze.

The Presentation layer contains the MVVM design pattern. The MVVM pattern consists of three components: the View, View-Model and Model. In this case, the View represents a collection of GUIs the user interact with. The View-Model contains the logic to handle events of the View and update the View with data from the Model. The Model contains all the information that's relevant for the View in question. The Model can also update the View if data in the Model is changed.

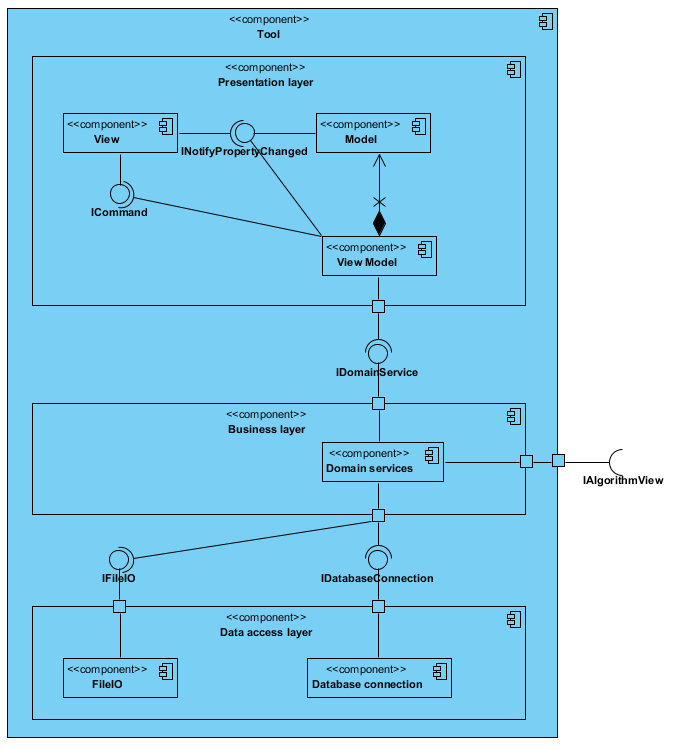


Figure 11 - Concrete component overview for the tool

###### INotifyPropertyChanged

The INotifyPropertyChanged interface is used to update the elements on the GUI of the View with the correct data.

###### ICommand

The ICommand interface is used to handle events in the View Model which were triggered when a user interacts with an element on the GUI of the View.

##### Business layer

The Business layer contains the core functionality of the system and encapsulates the relevant business logic. This layer will contain the functionality to select algorithms, execute algorithms, train algorithms; in other words: all the user functionality. This functionality is implemented in the Domain services component.

###### IDomainService

The IDomainService interface exposes the Domain services' operations that can be used by another component. These operations are called by the Presentation layer's View Model component.

###### IAlgorithmView

The IAlgorithmView interface consists of operations that can be called in the Algorithm library component (such as training an algorithm and create predictions). The Domain services component can call these operations.

##### Data access layer

The Data access layer contains the logic required to access the underlying data stores, such as databases and (configuration) files. It offers a common data access functionality in order to make the application easier to configure and maintain.

###### IFileIO

The IFileIO interface consists of operations that can be called by the underlying Business layer's Domain services component. These operations allow for reading and writing data from/to a (configuration) file.

###### IDatabaseConnection

The IDatabaseConnection interface consists of operations that can be called by the underlying Business layer's Domain services component. These operations allow for reading and writing data from/to a database.

#### Algorithm component

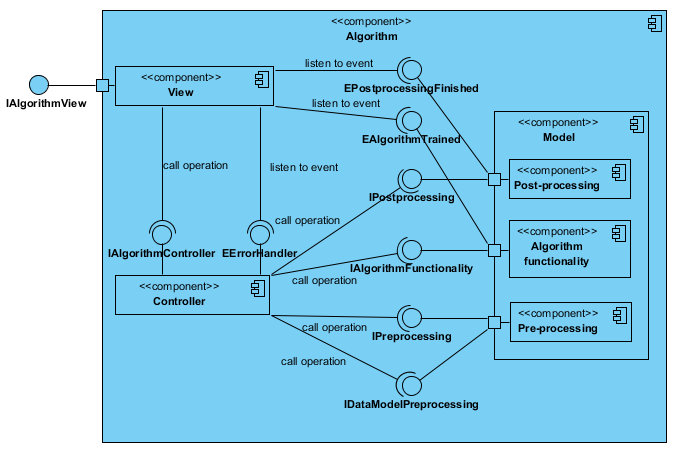


Figure 12 - Concrete component overview for the algorithm library

Figure 12 shows the concrete component overview for the Algorithm library component. The MVC design pattern is implemented in this component.

##### View

The View is used as an interface for the input and output data for the algorithm. The View contains operations that can be called in the Controller and passes the input data it receives to the Controller.

###### IAlgorithmView

The IAlgorithmView interface consists of operations that can be called in the View component (such as training an algorithm and create predictions). These operations expect input profile data or algorithm parameters. There're also an error handler, post-processing finished, and algorithm trained events specified the View needs to listen to.

##### Controller

The Controller is used to call operations in the Model to pre-process the input data received from the View, call algorithm-specific operations with the pre-processed data, and post-process the result of the algorithm-specific operations. So basically, the Controller creates a sort of pipeline that processes all the required steps to get the result the View is asking for.

###### IAlgorithmController

This interface consists of operations that can be called in the Controller component. This interface is used by the View component.

###### EErrorHandler

The EErrorHandler is an event that's used by the Controller to push error messages to the View. The View listens to this event so it can send the error messages to the Tool or AE component.

##### Model

The Model contains all of the following data-related operations:

* Pre-processing. The Model pre-processes the data by comparing the input data to external data models to find similar data. This similar data is translated to one common interpretation. After this translation, some data may need to be converted to another format to make it usable for the algorithm.
* Algorithm-related functionality. The Model can call the algorithm-related functionality such as making predictions and training the algorithm.
* Post-processing. The Model post-processes the data by converting the necessary data back to the original format.

Depending on the request of the Controller, the Model forwards a response to the View after executing specific algorithm-related functionality (like retraining) or post-processing the data.

###### EPostprocessingFinished

This event is used by the Model's Post-processing component to push the post-processed data to the View. The View listens to this event so it can send the post-processed data to the Tool or AE component.

###### EAlgorithmTrained

This event is used by the Model's Algorithm functionality component to push a notification message to the View that the algorithm has been (re)trained. The View listens to this event so it can send this notification message to the Tool or AE component.

###### IPostprocessing

This interface consists of operations that can be called in the Model's Post-processing component. This interface is used by the Controller component.

###### IAlgorithmFunctionality

This interface consists of operations that can be called in the Model's Algorithm functionality component. This interface is used by the Controller component.

###### IPreprocessing

This interface consists of operations that are relevant to converting data to another format that can be called in the Model's Pre-processing component. This interface is used by the Controller component.

###### IDataModelPreprocessing

This interface consists of operations that are relevant to translating data via external data models that can be called in the Model's Pre-processing component. This interface is used by the Controller component.

*<High level overview of the important components in the systems.>*

*<Views belonging to the functional viewpoint describe the system’s run time functional elements and their responsibilities, interfaces, and primary interactions. The functional view of a system defines the architectural elements that deliver the system’s functionality. These views document the system’s functional structureincluding the key functional elements, their responsibilities, the interfaces they expose, and the interactions between them. Diagrams used in these views model elements, connectors, interfaces, responsibilities and interactions, for instance using UML component diagrams.>*

## Information viewpoint

*<Views belonging to the information viewpoint describe the way that the architecture stores, manipulates, manages, and distributes information. This viewpoint concerns both the information structure and the information flow. Diagrams for these views are for instance DFD’s (Data Flow Diagrams), UML Class diagrams, ER (entity-relation) diagrams, and so on.>*

## Concurrency viewpoint

*<Views belonging to the concurrency viewpoint describe the concurrency structure of the system, mapping functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently, and shows how this is coordinated and controlled. This viewpoint is concerned with the system’s concurrency and state-related structure and constraints. Diagrams which can be used for these views are for instance UML state diagrams or UML component diagrams.>*

## Development viewpoint

*<Views belonging to the development viewpoint describe the architecture that supports the software development process. Aspects are module organization, standardization of design and testing, instrumentation, code structure, dependencies, configuration management. Diagrams for these views are for instance UML component diagrams with packages.>*

## Deployment viewpoint

*<Views belong to the deployment viewpoint describe the environment into which the system will be deployed, including the dependencies the system has on its run time environment. Aspects in this viewpoint are specifications of required hardware, of required software, or network requirements. Diagrams for these views are UML deployment diagrams.>*

## Use case view

<Refer to use cases that are important for creating the architecture. The use cases must have central functionalities, contain many architectural elements, or specify a delicate point of the architecture.>

|  |  |
| --- | --- |
| **UC Code + Name** | **Architectural relevance** |
|  |  |

## Logical view

<Beschrijf hier de architectureel significante logische opbouw van het systeem. Denk hierbij aan de decompositie in lagen en deelsystemen. Beschrijf ook de manier waarop Use Cases, rekening houdend met deze logische decompositie, technisch worden vertaald naar Use Case Realizations.

*Beschrijf de onderkende lagen (layers) en hun verantwoordelijkheid binnen het systeem. Ga hierbij bijvoorbeeld uit van het 4-lagen-model:*

* *Presentatie*
* *Service*
* *Domein*
* *Data*

*Geef van iedere laag duidelijk aan wat zijn verantwoordelijkheid binnen het systeem is en hoe op logisch niveau de communicatie met andere lagen plaats zal vinden. (De technische invulling van de lagen wordt beschreven in de Implementation View)*>

## Implementation view

<*Deze sectie beschrijft de technische invulling van de logical view.>*

### Package structure

*<Geef hier een overzicht van de packagestructuur in de ontwikkelomgeving. Probeer bij de package structuur de logische lagen scheiding terug te laten komen in de naamgeving van de packages, zoals in het voorbeeld hieronder:*

* *nl.bedrijf.afdeling.project.subsysteem.ui*
* *nl.bedrijf.afdeling.project.subsysteem.service*
* *nl.bedrijf.afdeling.project.subsysteem.domain*
* *nl.bedrijf.afdeling.project.subsysteem.data*

*Illustreer de samenhang van de package structuur dmv een package-diagram.>*

### Layer structure

*<Beschrijf hier de technische invulling van de in de Logical View onderscheiden lagen. Benoem ook de regels voor een component om in een bepaalde laag opgenomen te worden.>*

### (Re)use of components and frameworks

*<Beschrijf hier de bij de bouw te (her)gebruiken componenten en frameworks (intern en van derden). Dit voor zover ze niet bij de invulling van de lagenstructuur zijn behandeld. Indien er bij de eisen bepaalde frameworks zijn genoemd, dienen deze hier terug te komen.>*

## Deployment view

<Beschrijf hier de fysieke netwerk(hardware) configuraties waarop de software gaat draaien. Beschrijf minimaal de configuraties van de verschillende fysieke nodes (computers, CPUs), de interactie tussen (deel)systemen en de connecties tussen deze nodes (bus, LAN, point-to-point, messaging, SOAP, http, https). Maak gebruik van een deployment-diagram.>