Welcome to everyone and thank you attending to the talk about the Attribute Driven Design method to tackle complexity in the heart of software architecture.

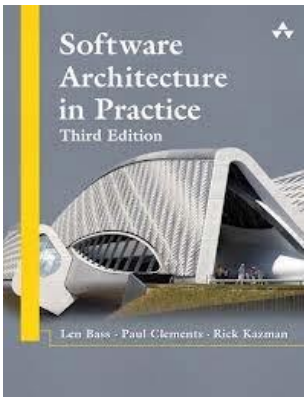
I'm gonna speak today about what the Attribute Driven Design is, how can it help software designers including architects to design great software architectures, what important inputs need to be available in order to apply the method successfully and a practical example of the application of the method in a software system.

Outline

- Software Architecture and importance.
- Important inputs in architectural design.
- Introduction to Attribute Driven Design 3.0.
- Design concepts and considerations.
- Application of ADD
- Conclusion
- Questions & Answers

Software Architecture definition

- The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.



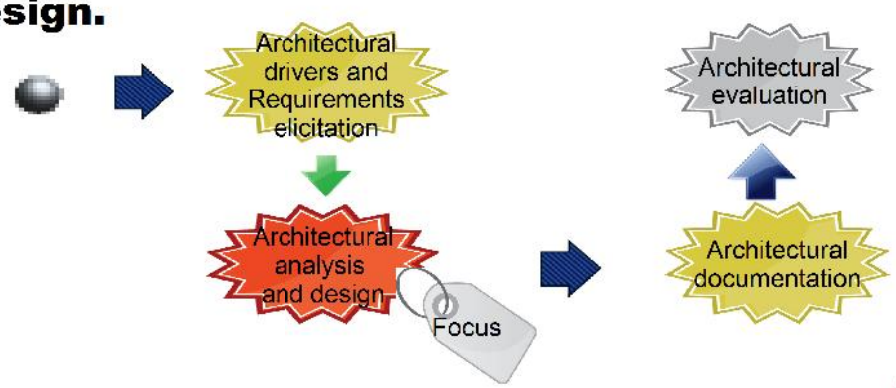
There are many definitions of software architecture and the one what I'll present today is this one:

I wanna stress three important elements named in this definition:
-1 A set of structures
-2 software elements and the relations among them and
-3 properties of the structures and also of the software elements.

I'll come back to this three elements of the definition when I get to explain the Attribute Driven Design steps method because the outputs of the method are exactly the elements of this definition.

The 4 phases in architectural design life cycle

- **In software architecture life-cycle there are 4 important phases.**
- **The focus of this talk is indeed Architectural design.**



In the life cycle of software architecture development we have 4 important phases to include.

And an interesting thing that I've personally found about this method is that it includes in the steps the most important tasks what we should do to flesh out architectures with predictable outcomes.

Importance of software architecture

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.

1. In this respect it's important to note that one successful architecture in, for example, an large enterprise company will be conpleted wrong for an avionics application.
2. If you create a system with no semantic coherence, high coupling reasoning about and apply changes will be harder and harder as the system evolves.
3. This is particulary important because even before a single line of code is written, we can predict system responses to some stimuli and correct the design in this phase which is the most cheapest moment to discover flaws in the system.
4. If you as an architect did not document or communicate your software architecture to your stakeholders, you have a bigger chance of having to loose your time because people will come again and again with questions about the design. This makes also not possible to conduct an architecture evaluation and is very riscky for all not trivial projects.

Importance of software architecture

5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
8. An architecture can be created as a transferable, reusable artifact that forms the basis of a product line in software product line development.

5. I think every one will agree with this, that if you don't make the right design decision or you just don't make them early enough, you can cause the whole project to be cancelled because changing something fundamental can be prohibitively expensive
6. When a good architecture is designed by limiting complexity, implementing new features is less complex.
7. By fleshing out a some preliminary architecture deliverables we can predict some costs and budget because the mayor components have been identified.
8. This is particularly true in the car and avionics industry where around 60 % and sometimes more of the software is reused based on the architecture.

Importance of software architecture

9. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
10. An architecture can be the foundation for training a new team member or a new hire.

9. I have personally seen this in projects where me as developer did not have to deal with infrastructural issues and external communications with other systems so that I could focus my attention on developing business logic and features.
10. I have personally come to work in a company and in order to understand the system I had to look at the code and a simple page diagram where according to them that was the "architecture of the system". And to understand the architecture faster, a proper documented architecture would have been better.

Important inputs in architectural design

- Business drivers or business goals: are the basics for creation of quality attribute requirements. This business drivers are the ones what should drive and shape the architecture.
- There are very mature methods like Quality Attribute Workshop (QAW) and the Utility Tree for eliciting, specifying, prioritizing, and validating quality attributes that takes as input the business drivers that motivate the creation of the system.
- Quality Attribute: a quality attribute is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders.
- Quality attributes scenario: a quality attribute scenario is a testable, falsifiable hypotheses about the quality attribute behaviour that binds a stimulus with an expected response.

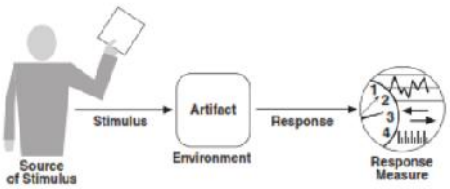


FIGURE 4.1 The parts of a quality attribute scenario

An important output of the Quality Attribute Workshop or the The Utility tree is a list of prioritized quality attributes. It is very important to get this quality attributes right because the critical choices that we make when we do architectural design determine, in large part, the ways that the system will or will not meet these driving quality attributes goals..

Important inputs in architectural design

- General quality attribute scenario for availability.

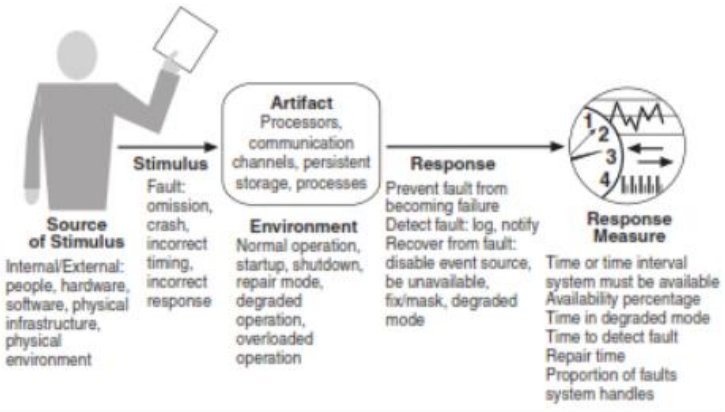


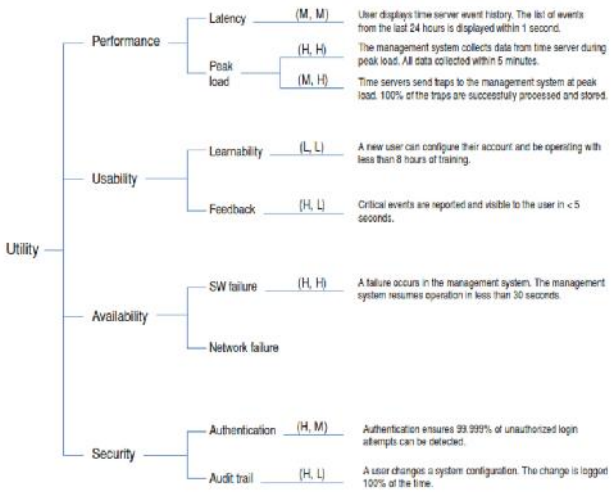
FIGURE 4.2 A general scenario for availability

As a concrete quality attributes scenario for availability we can take as example
“A failure occurs in the one of the components for calculating altitude in the flight control software. The system continues calculating altitude and producing accurate results with not problem and starts a recovery procedure for that component”.

If we have to design for achieving this kind of quality attribute scenario it will be something like triple redundancy with independent components calculating altitude and a voter component that takes and analyses the results of all of them to decide which is trustable and a monitor that constantly verifies the health of every component.

Important inputs in architectural design

- **Utility tree example.**



Utility Tree

If no stakeholders are readily available to consult, we still need to decide what to do and how to prioritize the many challenges facing the system.

One way to organize your thoughts is to create a Utility Tree. We have H for High, M for medium and L for low.

The Utility Tree helps us to articulate the quality attribute goals in detail, and then to prioritize them.

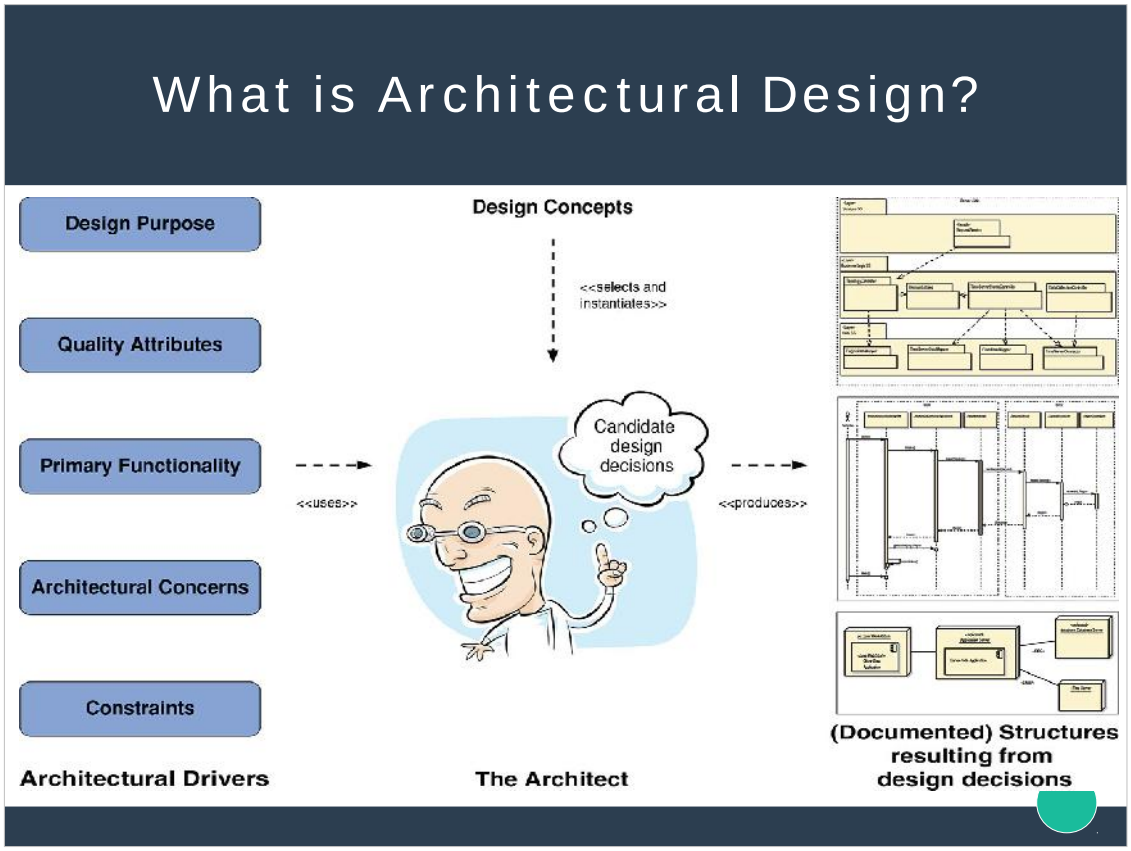
The first dimension in for example (H,H) is the business importance which is ranked by the customer and the second is the technical Risk or difficulty according to the architect.

Then when starting design, we as architects should first focus with the ones that have (H,H) and then (H,M) and (M,H)..

Important inputs in architectural design

- Primary functionality
  - Defined a functionality that is critical to achieve the business goals that motivate the development of the system
  - Approximately 10% of the the use cases or user stories are likely to be primary.
- In architectural design allocation of functionality to elements is what matters, rather than functionality per se, because the way the system is structured does not normally influence functionality.
  - We can have a single enormous module and externally the system will look and feel the same if we consider only “functionality” in architecture design (And I’ve seen this done in a few companies).

Important about functionality is to think about how will it be allocated to elements (usually modules when we create module structures) to promote modifiability or reusability.



Design, like architectural design, is not different than design in other fields in general. It's the process of making design decisions, working with available skills, constraints and materials to achieve particular requirements.

When designing a software architecture, we typically make design decisions involving translation of business requirements into architectural drivers, designing around the given constraints that also become drivers and finally translating those architectural drivers into structures that describe the architecture.

This structures allows us to reason about and evaluate the properties of the system (that is, the desire qualities that we want it to have) even before the fist line of code is been written. The documented structures are related to each other in order to realize the primary functionality and the quality atributes.

Note that we see here the elements of the deffinition of softare architecture from the beginning of the presentation.

Introduction to ADD 3.0

- ADD is a well established and mature software architecture design method that provide step by step guidance in how to architect software systems in a series of iterations.
- It focuses on the achievement of quality attributes requirements through the selection and instantiation of different types of structures. It encapsulates a roadmap that contains the best practices applied in the most competent architecture organizations and it packages the best methods observed in other architecture design methods.
- The version 2.0 was introduced 2006, by some important members from the Software Engineering Institute (SEI/CMU), in a Technical Report.
- Later in 2013, it was improved by H. Cervantes, P. Velasco and Rick Kazman, and was documented after more and more scientific research. In 2016 it appeared ADD 3.0 with more improvements in a book called Designing Software Architectures, that addresses design specifically
- In short, the authors of the books Software Architecture in Practice (1st, 2nd and 3rd Edition), Documenting and Evaluating Software Architectures and Designing Software Architectures, A practical Approach from the Software Engineering Institute series.

The version 2.0 was first introduced by very influential computer science scientists (R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord and B. Wood) in 2006, from Software Engineering Institute (SEI/CMU), Technical Report.

Later in 2013, it was improved by H. Cervantes, P. Velasco and Rick Kazman, and was documented in a book with additional improvements (Version 3.0) and examples in Designing Software Architectures (described under).

In short, the authors of very influential book in the industry Software Architecture in Practice (1st, 2nd and 3rd Edition), Documenting and Evaluating Software Architectures and Designing Software Architectures, from the Software Engineering Institute series.

Attribute Driven Design 3.0

- Design purpose:
  - Project proposal (pre-sales process as called in the consulting world)
  - Exploratory prototype
  - During development
- Constraints
  - Mandated technologies,
  - other systems with which your system needs to interoperate
  - Laws that the system need to be compliant with
  - Abilities and availability of the developers
  - Not negotiable deadlines
  - Backward compatibility with older versions of systems ...
- Architectural Concerns
  - General concerns
  - Specific concerns
  - Internal requirements
  - Issues

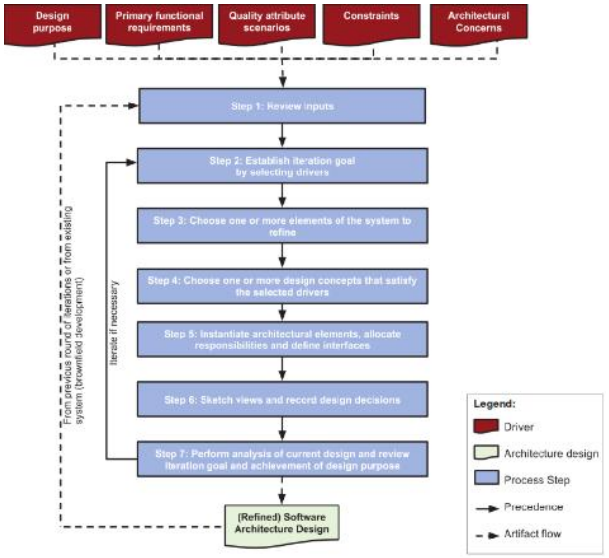


FIGURE 3.1 Steps and artifacts of ADD version 3.0

I'll now speak about the Design Purpose, Constraints and Architectural concerns which are the missing elements of the ADD imputs.

Design Purpose

We need to be clear about the purpose of the design that we want to achieve. We need to know the answers or find them for questions like:

-is this design for a project proposal? Because if it is, the purpose will be to understand and break down the architecture in sufficient detail that the units of work are understood so that we can estimate it. In that case the architecture will not be very detailed.

-Is an explanatory prototype? If yes, the purpose will be to explore the domain, to explore new technology or to show something to the customer to get rapid feedback, or to assess some quality attribute such as performance, scalability, availability or failover).

Constraints

Constraints are design decisions where we as architects have no or very little control about and have most of the time no other option than to design around them.

Architectural concerns

Under General concerns we have the "broad issues" that one need to address when designing an architecture. Examples are Establishing an overall system structure, allocation of functionality to modules, allocation to modules to for example UI and Back end teams, supporting delivery, deployment, updates.

We also have Specific concerns that are more detailed issues such as exception management, configuration, logging, dependency management, security, caching.

Internal requirements:

Like continuous integration and deployment.

Issues which are the results of analysis activities after architecture evaluation.

Attribute Driven Design 3.0

- **Step 1:**

  we need to review the inputs and make sure that the other drivers needed to the design activity are available. Particular attention need to be given to the quality attribute scenarios that need to be prioritized by the most important stakeholders.

  If this is not the first round or this is not a greenfield development, we need to take as additional input the existing architectural design. If there is not documentation of the brownfield system, it needs to be produced or at least the most important components need to be well understood.

- **Step 2:**

  - **here we select the drivers that we want to achieve as first. If this is a greenfield system in a mature domain (enterprise applications), an iteration goal should be to produce an overall system structure by choosing an reference architecture (see next slide for an example).**

- **Step 3:**

  - **In order to satisfy drivers we need to produce architectural structures. Here, we select the elements that are involved in the satisfaction of the specific selected drivers from step 2. We then need to refine this element in the next steps until we are convinced that the selected driver is**

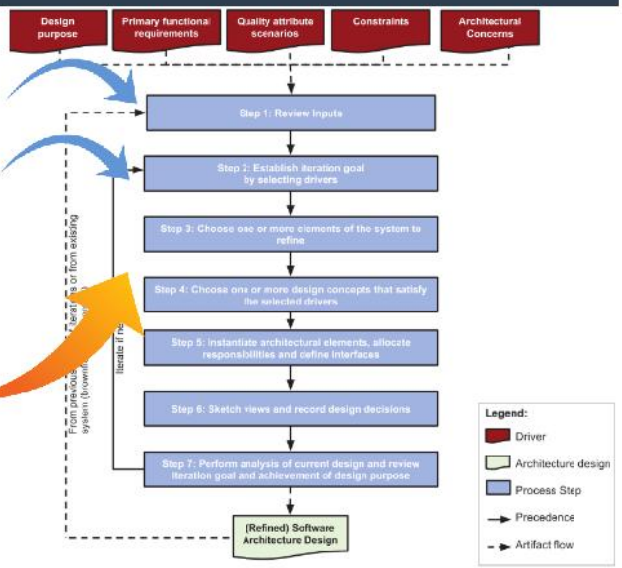


FIGURE 3.1 Steps and artifacts of ADD version 3.0

Design concepts and considerations

- Reference architecture example:

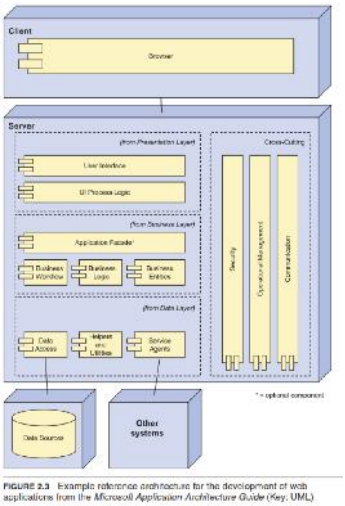


FIGURE 2.3 Example reference architecture for the development of web applications from the *Microsoft Application Architecture Guide* (Key: UML)

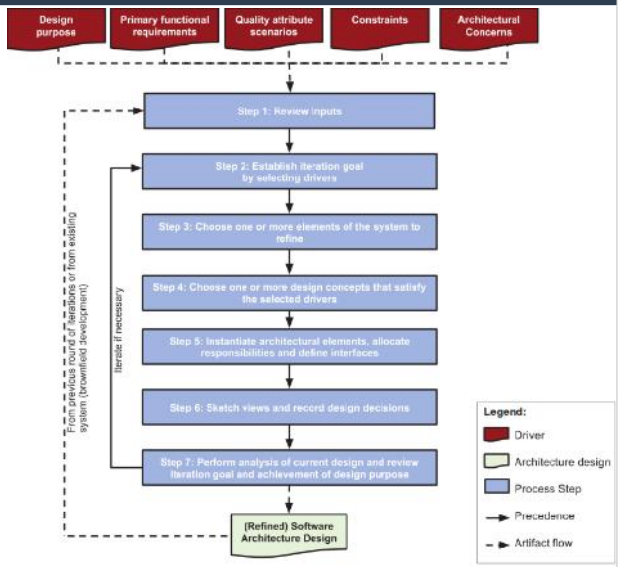
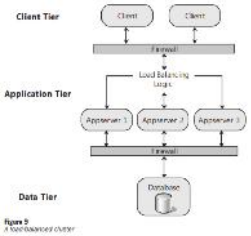


FIGURE 3.1 Steps and artifacts of ADD version 3.0

Here we have one example of one of the categories of design concepts. This is a reference architecture for the deployment of web applications from the Microsoft Application Architecture Guide. In step 4 of the method I'll come back to design concepts.

Design concepts and considerations

- **Load-balanced cluster**:from the Microsoft Application Architecture Guide book:



  - Single point of failure if not well designed

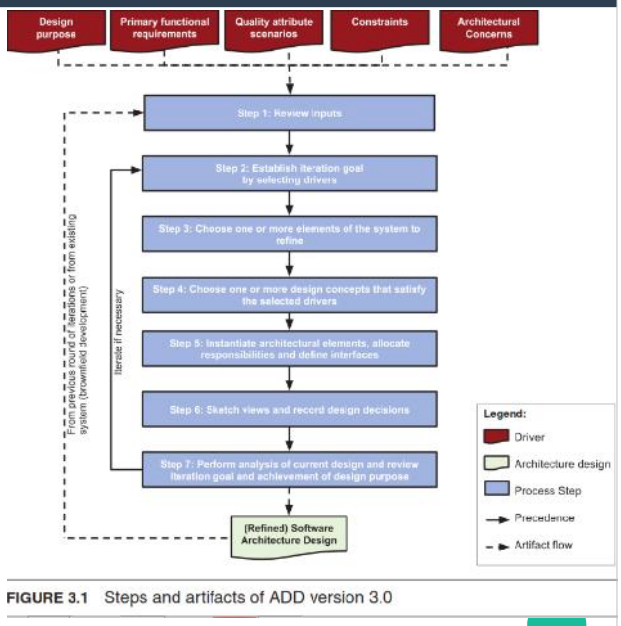


FIGURE 3.1 Steps and artifacts of ADD version 3.0

Here we have one example of a load balanced cluster deployment pattern. If you apply this deployment pattern, make sure that you design the load balancer with fail-over mechanism because if not, the load balancer can be itself a single point of failure like that single bike. Here you can add an architectural concern task to you backlog to address this weakness of the pattern.

Design concepts and considerations

- Failover cluster from the Microsoft Application Architecture Guide book:

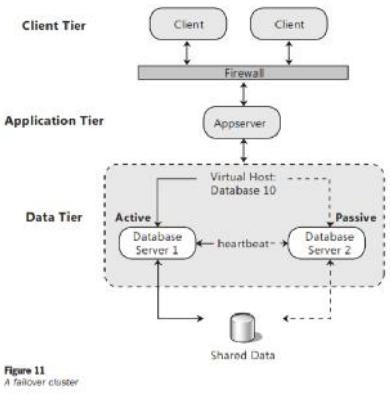


Figure 11
A failover cluster

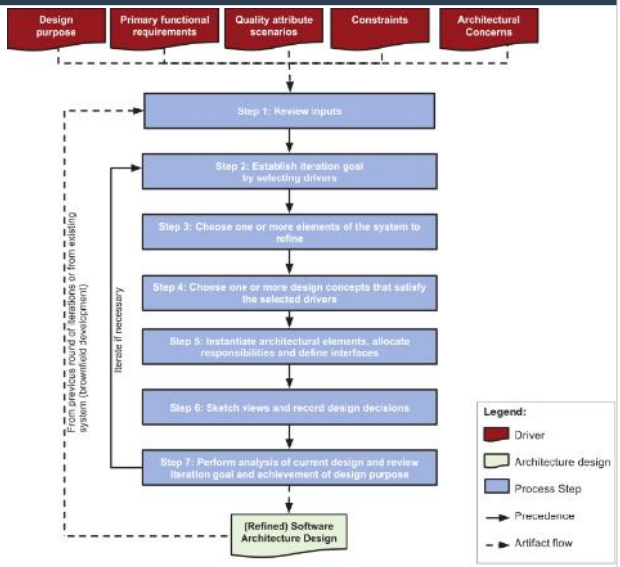


FIGURE 3.1 Steps and artifacts of ADD version 3.0

You can see in this pattern that a bunch of patterns are employed in just one package.

-You can see the layers pattern with three layers. You can see the architectural tactic active redundancy (hot spare) and passive (worm spare). You can also see the heartbeat availability tactic employed.

Design concepts and considerations

- Availability tactic from the Software Architecture in practice 3rd Edition:

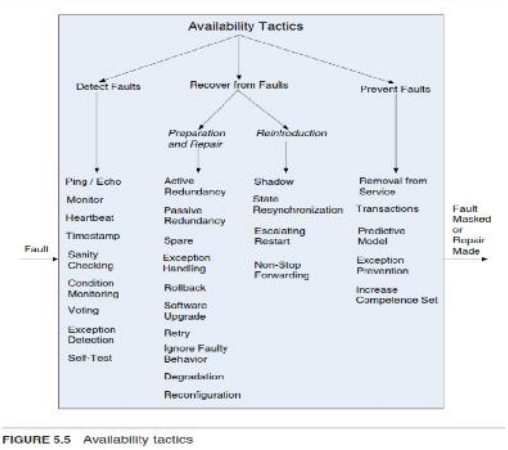


FIGURE 5.5 Availability tactics

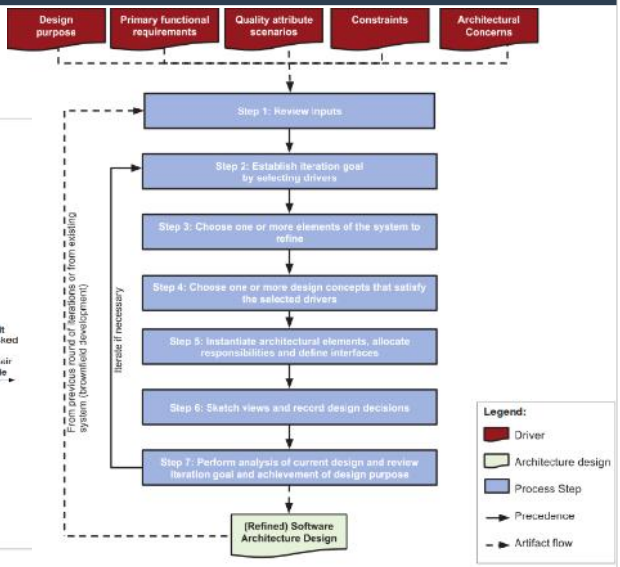


FIGURE 3.1 Steps and artifacts of ADD version 3.0

Here is an example of an availability tactic palace produced with the help of a Chief Architect of Boeing Corporation. This is almost the entire palace of availability tactics that we as architects can use to achieve demanding quality attribute requirements.

To detect faults we can use ping / Echo, where one component assumes that the other component is ok if it gets the echo.

We can also use the monitor in combination with the Circuit Breaker to allow a fault detection mechanism and a recovery procedure or an auto-scaling policy.

Architectural tactics the the building blokes from which design patterns are created. Tactics are like atoms and patterns are molecules as way of analysis

Design concepts and considerations

- Important design concepts are the externally develop components such as frameworks. This, as deployment patterns instantiate pattern, tactics.

Problem	Framework	Use
OO – Relational Mapping	Hibernate	XML, annotations
Local user interface	Swing	Inheritance
Component connection	Spring	XML, Annotations
Unit testing	JUnit	Inheritance, Annotations
Web UI	JSF	XML

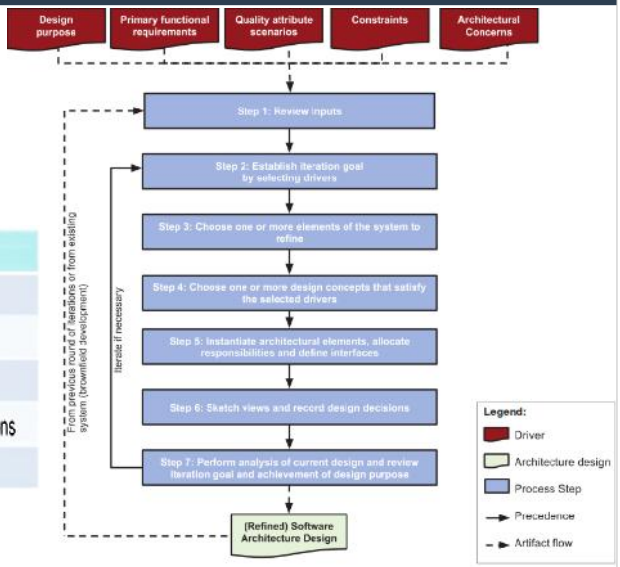


FIGURE 3.1 Steps and artifacts of ADD version 3.0

If you need to address OO-Relational mapping, you can use ...

It's important to note that with this method, design stops from being random, and becomes rational, directed and intentional. We should not be designing solutions in an adhoc way, but instead, use a method that architects use in the most competent architectural organisations.

Some of the collected experience of hundreds of architects dictate:

-to achieve high modifiability, design with good modularity, which means use semantic coherence and low coupling at method level, class level, subsystem and systems level.

-to achieve high availability, avoid having any single point of failure with proper redundancy and failover mechanisms. Take as example the Simian Army of Netflies and the Caos monkey how kill processes randomly to allow automatic recovery of instance by other processes.

-to achieve scalability, do not hard-code limits for critical resources (so build variation points into the architecture and flexibility like with properties files). Take as example the Simian Army of Netflies and the Conformity monkey that kill instances that doe not belong to an auto-scaling group to ensure that auto-scaling mechanism works as expected.

-to achieve security, limit the access points to critical resources (use for example an intermediary and enforce the use of it by components)

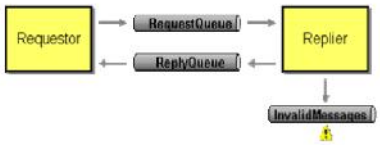
-to allow testability, externalize state so that the state can be changed easily for testing.

Design concepts and considerations

- Enterprise Integration Patterns for messaging from the Enterprise Integration Patterns book.

JMS Request/Reply Example

This is a simple example of how to use messaging, implemented in JMS [JMS]. It shows how to implement *Request-Reply*, where a requestor application sends a request, a replier application receives the request and returns a reply, and the requestor receives the reply. It also shows how an invalid message will be rerouted to a special channel.



Components of the Request/Reply Example

This example was developed using JMS 1.1 and run using the J2EE 1.4 reference implementation.

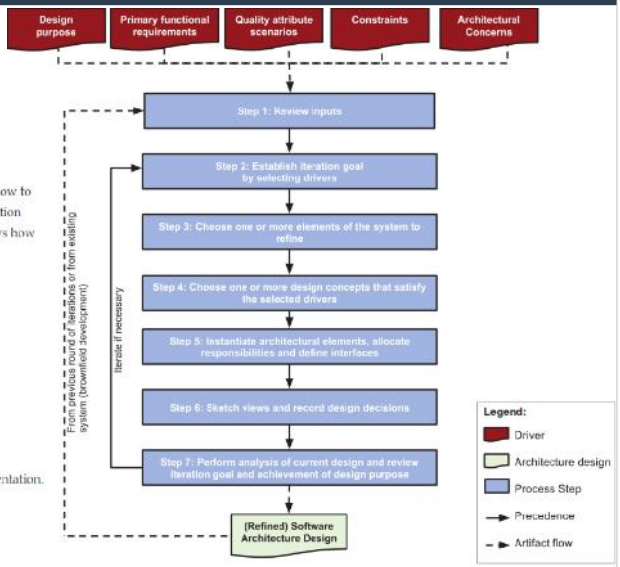


FIGURE 3.1 Steps and artifacts of ADD version 3.0

In this integration pattern we can see the request reply pattern as design concept that we can choose if we need to decouple the request from the response asynchronously with high throughput requirements.

Application of ADD

- **Step 4:**

  This is probably the most difficult step in the design process.

  It's difficult because we need to identify potential design concepts generate candidates and choose the best suited for the satisfaction of the drivers.

- **Design concepts: The building blocks for creating architectural structures.**
  - **Here is where we need to use our engineering capabilities. Design concepts categories:**
    - **Reference architectures (such as the one for web application in previews slide)**
    - **Architectural design patterns (such as pipes and filters, broker, and other architectural styles) or the GANG of 4, GRASP or Half-Sync-Half-Async and hundreds of others**
    - **Deployment patterns**
    - **Architectural Tactics**
    - **Externally develop components**

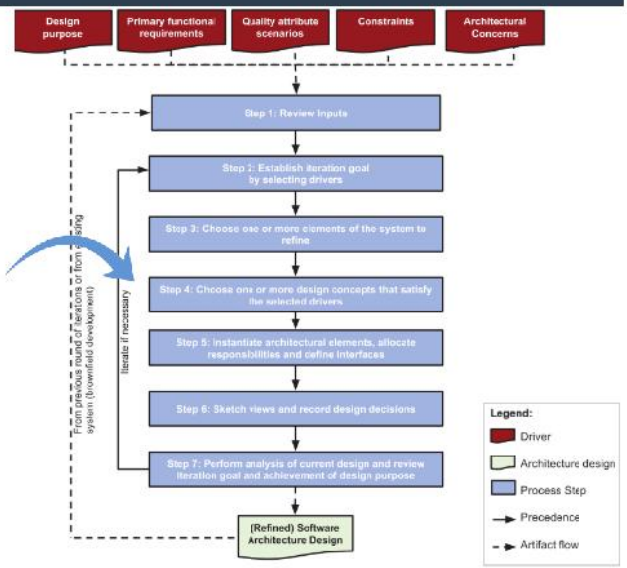


FIGURE 3.1 Steps and artifacts of ADD version 3.0

We have all this tools as architects available and they have been developed over the course of years by other architects. This is a catalogue of what we called **design concepts** that we can use to achieve a particular goal in hour systems.

Here in step 4 starts the real architectural design.

To start the selection process we need to look at the **design purpose** of the iteration (so the selected driver to satisfy) and take into account the **catalogued constraints** and **architectural concerns**. Note that this step of the method is helpful because this narrows down the amount of options that we can use to achieve he design purpose.

We need to combine most of the times multiple design concepts to achieve a particular goal and that is the reason why we should be skillful in at least the next areas:

- about how to apply design principles like **advanced object oriented** skills, **high coherence such as semantic and functional coherence**, **low coupling**,

-have knowledge about **design patterns like the Gang of 4**

- deployment patterns, **integration patterns and messaging.**

- General Responsibility Software Assignments Patterns like the **GRASP patterns** introduced bij Craig Larman in his influential book called Applying UML and Patterns,

-knowledgeble about **quality attributes and how to achieve them usign architectural tactics**, **reference architectures and frameworks or the so called externally developed components, such as Hibernate, JPA, or other technologies**.

Application of ADD

- **Step 5:**
- **In step 5 is we do the Instantiation of the selected elements from step 4 involving making more design decisions having in mind the categorised constraints and architectural concerns.**
- **Instantiation means that we take the selected design concepts and tailor it to the problem at hand. This means adapting the design concept to our needs.**
- **If you selected the reference architecture for deployment of web applications, you need to make design decisions about how many layers your design will have as well as how are you going to deal with for example security and the cross-cutting concerns that the reference architecture has. This can be in another iteration if you decide it so.**
- **In other cases instantiation simple means configuration and even a Prove of Concept (POC) prototype can also be instantiation.**
- **For allocating responsibilities and defining interfaces we should use dynamic analysis like, activity diagrams, sequence diagrams, and communication diagrams. Even better to use the more sophisticated and rigorous Responsibility Driven Design methods as explained in the GRASP Patterns (the General Responsibility Software Assignment Patterns) from Craig Larmax's bible book. Then you have allocated your functionality with high degree of semantic coherence, low coupling and all the good qualities of an excellent design.**
- **We finish this step when the elements are connected via the interfaces. Here we have produced the relationships of the definition of Software Architecture. Don't document all the interfaces precisely as this is not necessary. Only the ones that address the selected drivers because the other ones required a significant amount of time and have no mayor**

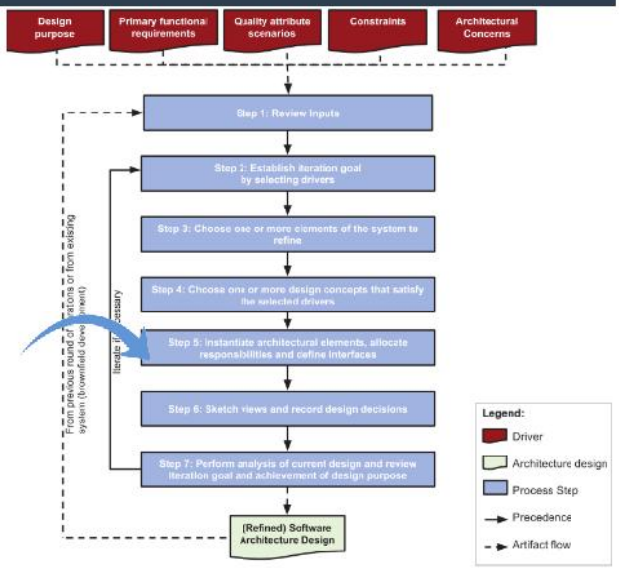


FIGURE 3.1 Steps and artifacts of ADD version 3.0

Application of ADD

- **Step 6 part 1: Sketch Views**
- **Here is where we record the architecture work done in previews steps 4 and 5. You don't have use necessarily a formal language like UML for sketching the design decisions all the time.**
- **Make sure that you are consistent in your diagrams and make sure that you have added a legend explaining what the boxes, arrows, lines and used symbols mean. This is an example of the reference architecture for big Data Domain using the Lambda Architecture. This is what the architect used as starting point in steps 4 and 5 for some big data demanding quality attributes.**

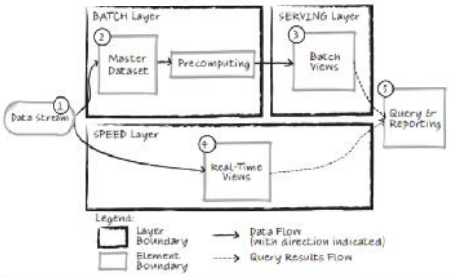


FIGURE 5.2 Lambda Architecture

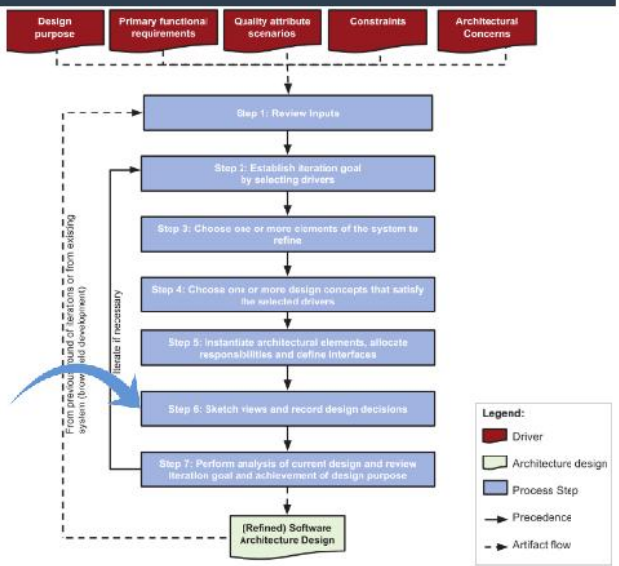


FIGURE 3.1 Steps and artifacts of ADD version 3.0

Application of ADD

- **Step 6 part 2: Record design decisions**

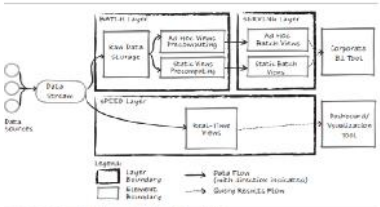


FIGURE 5.4 Instantiation of the Lambda architecture

Element	Responsibility
Data Sources	Web servers that generate logs and system metrics (e.g., Apache access and error log, Linux syslog).
Data Stream	This element collects data from all data sources in real-time and dispatches it to both the Batch Layer and the Speed Layer for processing.
Batch Layer	This layer is responsible for storing raw data and precomputing the batch views to be stored in the Serving Layer.
Serving Layer	This layer exposes the batch views in a data store (with no random writes, but batch updates and random reads), so that they can be queried with low latency.
Speed Layer	This layer processes and provides access to recent data, which is not available yet in the serving layer due to the high latency of batch processing, through a set of real-time views.
Raw Data Storage	This element is a part of the batch layer and is responsible for storing raw data (immutable, append only) for a specified period of time (QA-7).
Ad Hoc Views Precomputing	This element is a part of the Batch Layer and is responsible for precomputing the Ad Hoc Batch Views. The precomputing represents batch operations over raw data that transform it to a state suitable for fast human-time querying.
Static Views Precomputing	This element is a part of the Batch Layer and is responsible for precomputing the Static Batch Views. The precomputing represents batch operations over raw data that transform it to a state suitable for fast human-time querying.

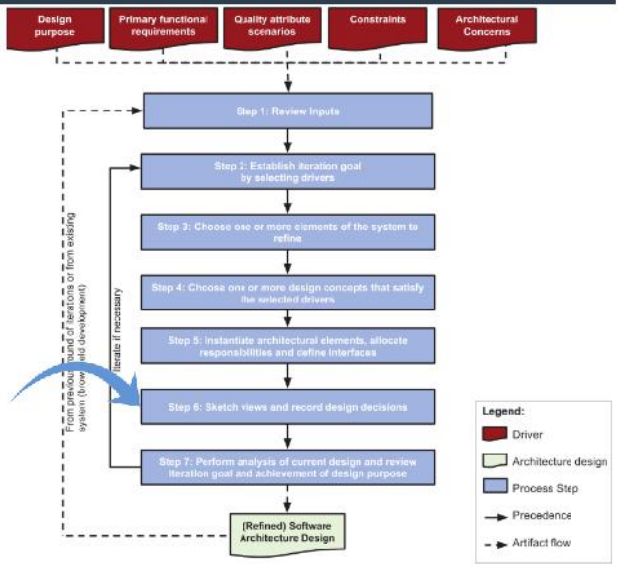


FIGURE 3.1 Steps and artifacts of ADD version 3.0

Application of ADD

- **Step 6 part 2: Record design decisions**

5.3.3.5 Step 6: Sketch Views and Record Design Decisions

Figure 5.6 illustrates the result of the instantiation decisions. The responsibilities of the elements shown in the diagram were discussed in step 6 of Iteration 1. The following table summarizes the technology families and candidate specific technologies selected for these elements:

Element	Technology Family	Candidate Technology
Data Stream	Data Collector	Apache Flume
Raw Data Storage	Distributed File System	HDFS
Ad Hoc Views Precomputing	Data Processing Framework	Apache Hive
Static Views Precomputing	Data Processing Framework	Apache Hive
Ad Hoc Batch Views	Interactive Query Engine	Impala
Static Batch Views	Interactive Query Engine	Impala
Real-Time Views	Distributed Search Engine	Elasticsearch
Dashboard/ Visualization Tool	Interactive Dashboard	Kibana

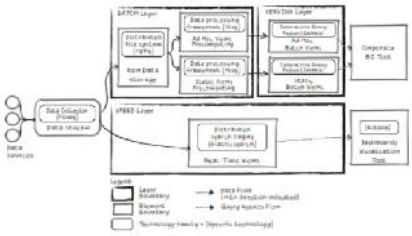


FIGURE 5.6 Iteration 2 instantiation design decisions

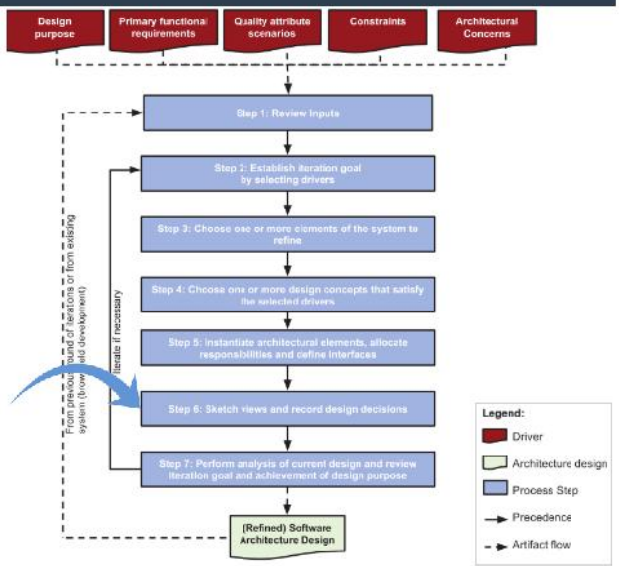


FIGURE 3.1 Steps and artifacts of ADD version 3.0

Application of ADD

- **Step 6 part 2: Record design decisions**

The BI Platform family group and related technologies are not considered further in this design exercise because the corporate BI tool is external to the target system.

Design Decisions and Location	Rationale and Assumptions
Select the Data Collector family for the Data Stream element	Data Collector is a technology family (and an architectural pattern) that collects, aggregates, and transfers log data for later use. Usually Data Collector implementations offer out-of-the-box plug-ins for integrating with popular event sources and destinations. The destinations are the Raw Data Storage and Real-Time Views elements, which will also be addressed in this iteration.
	Alternative — **Reason for Discarding**
	ETL Engine — The main purpose of ETL engines is to perform batch transformations, rather than per-event operations. This means that real-time performance and scalability criteria (QA-1, QA-2) will be extremely difficult to meet (if it is possible to meet them at all).
	Distributed Message Broker — Although this technology family can be solely used to implement the Data Stream element, it provides less support for extensibility (QA-9) and, therefore, is better suited as a complement to the data collector. This can be achieved, for example, using Flavka—a combination of Apache Flume (Data Collector) and Apache Kafka (Distributed Message Broker).
Select the Distributed File System family for the Raw Data Storage element	According to the Lambda architecture principles, the Raw Data Storage element must be immutable. Thus new data should not modify existing data, but just be appended to the dataset. Data will be read in batch operations for transforming raw data to Batch Views. For these purposes, we can confidently choose a Distributed File System.
	Alternative — **Reason for Discarding**
	NoSQL Database — Although NoSQL databases (especially column-family and document-oriented) can be used for storing raw data, such as logs, this will cause unnecessary overhead in resource consumption (mostly memory consumption because of caching mechanisms) and maintainability (because of the need of configuring and evolving a schema).
	Analytic RDBMS — All relational databases including analytic capabilities are based on the relational model, forming tables and rows. This works very well for executing complex queries, but this option is awkward (and expensive) for storing semistructured logs in their raw format.

(continues)

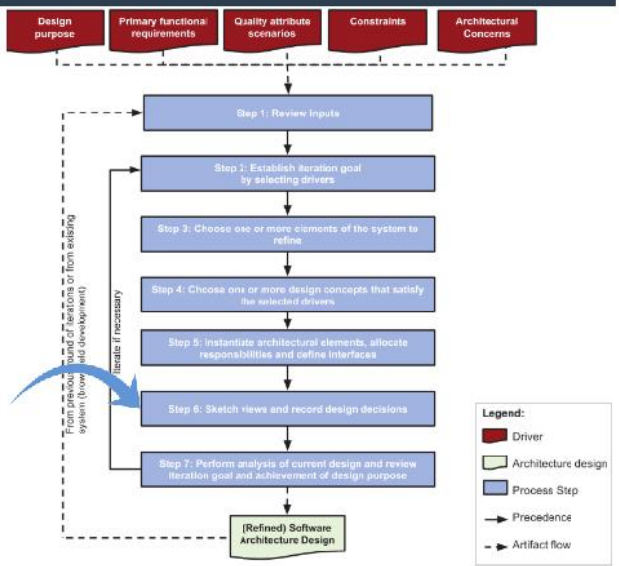


FIGURE 3.1 Steps and artifacts of ADD version 3.0

Application of ADD

- **Step 6 part 2: Record design decisions**

Design Decisions and Location	Rationale and Assumptions
Select Interactive Query Engine family for both the Static and Ad Hoc Batch Views elements	As we stated in the previous iteration, the Batch Views element is refined into two elements, the Static and Ad Hoc Batch Views, to support two use cases: the generation of static reports (UC-3, 6) and the support for ad hoc querying (UC-4). The main design decision is to use the same technology family for both Static and Ad Hoc Batch Views—namely, the Interactive Query Engine. These engines allow analytic database capabilities over data stored in a Distributed File System (thus this technology family is also selected implicitly). If we select a technology that is fast enough, it can be used for both elements. The benefit of using a single technology family is that we do not need to have separate storage technologies for reporting and querying data.

Alternative	Reason for Discarding
NoSQL Database	The Static Batch Views element can be implemented with the Materialized View pattern, by storing data in a form that is ready for querying and displaying in a reporting system (a corporate BI tool). The NoSQL Database family is often used for this purpose because it provides good scalability and, being open source, satisfies QA-8 (approximately 90 TB of aggregated data) and CON-1 (open source license). However, NoSQL databases are not good options to use as data warehouses for ad hoc queries because they were not designed for analytic purposes. Although they can be used for this purpose, this application will result in significant performance penalties. This alternative is therefore discarded as it can be used only for the Static Batch Views, but is ineffective for Ad Hoc Batch Views
Analytic RDBMS	Ad hoc queries can be any queries that are supported by a SQL-like interface. The query result must be returned within "human" time (QA-5). The described scenario is exactly what a data warehouse is used for. This pattern is usually implemented with Analytic RDBMS technologies following the Kimball or Inmon design approaches. At the same time, it will be quite costly to satisfy the scalability requirement of having approximately 90 TB of aggregated data. The cost per terabyte in MPP analytic databases is significantly higher (up to 30 times) than the same amount of data in a NoSQL database or a distributed file system (such as Hadoop).

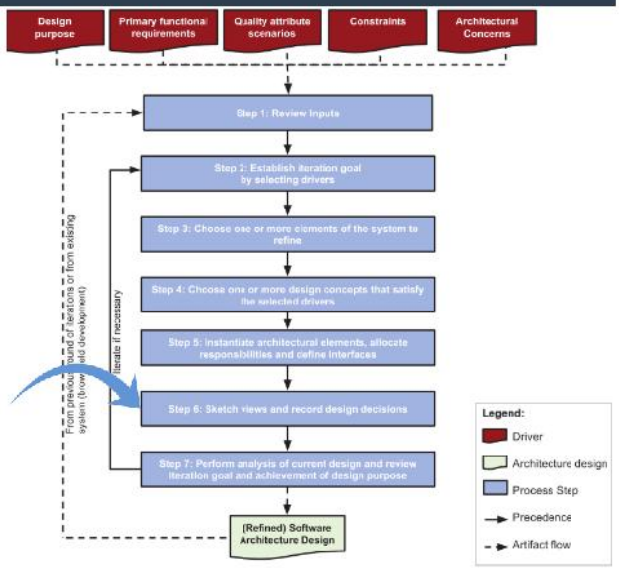


FIGURE 3.1 Steps and artifacts of ADD version 3.0

Application of ADD

- **Step 7: In**
- **In this step we review that the selected drivers have been addressed.**
- **If for example we had a quality attribute scenario to achieve and we had it selected for this iteration stating: "One the user presses the pay button, the system should complete the order within 5 seconds at peak load", and you also had decided that you will have 3 layers in your system in this iteration or in a previews one, you have to have assigned time budgets to each of those layers to make sure that they don't take longer than 5 seconds to process the order. That can be 1 second max for the presentation layer, 1 second max for the service layer and 3 seconds max for the integration layer.**
- **You should also have addressed the "peak load" requirement meaning that you either have resolved the processing of peak load requirement with for example some queuing mechanism or you have introduced a new architectural concern to address it in a later iteration.**
- **In that case the driver should be moved in this step to the partially address column (for example in a Kanban board) and the design decision should have been documented.**
- **Note the power of this method. It allow you to defer decisions to subsequent iterations so that you can focus in resolving the particular driver that you've selected. You finish this step when you have verified that you didn't forget anything.**
- **You continue with iterations and refinements by selecting more drivers and eventually introduced architectural concerns until the prioritized architectural driver have been addressed.**

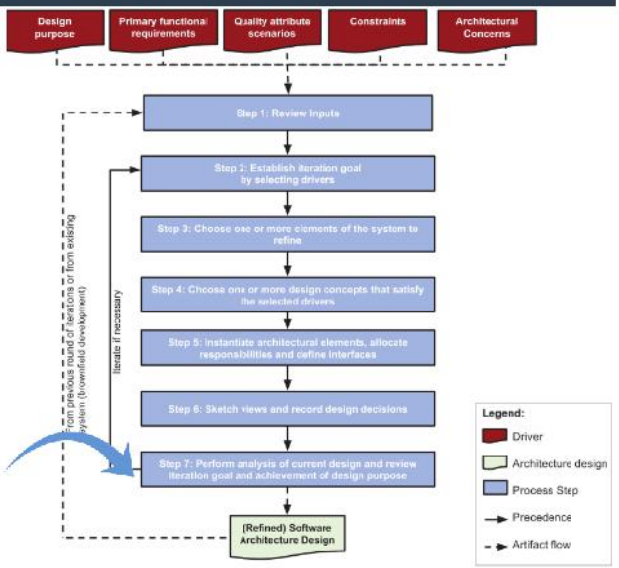


FIGURE 3.1 Steps and artifacts of ADD version 3.0

Conclusion

- In this presentation we've learnt what Software Architecture is, how important it is in an organization, what the Attribute Driven Design (ADD) method is and how it operates in practice. We have learnt that the outputs of the method are the elements, relations and properties of what is defined by the SEI as Software Architecture. We have learnt what quality attributes are, and have seen examples of well articulated Quality Attribute Requirements. We have learnt about the important inputs for the method, and how to process design in iterations. We have learnt the different available categories of design concepts such as Reference Architectures, Design (Architectural) pattern, Architectural Tactics, Externally developed components, examples of constraints and architectural concerns. We have seen that documentation and analysis are integral parts of the steps of the method and also how to record design decisions.

Questions & Answers

- Thank you and feel free to send me any question regarding the presented material in this session:

Luis Manuel Muegues Acosta

Software Architect at Ryanair

topcerebro@gmail.com

muegues@ryanair.com

References and further reading

- Software Architecture in Practice 3rd Edition
- Microsoft Application Architecture Guide 2nd Edition
- Designing Software Architectures
- Applying UML and Patterns Third Edition
- Enterprise Integration Patterns, Designing, Building, and Deploying Messaging Solutions
- Design Patterns: Elements of Reusable Object-Oriented Software.
- Pattern Oriented Software Architectures Volum 1,2,3 and 4.