Software Development

Group ARRO Lyngby

16/09/2024

Systems Integration

OLA 2

Name	Mail
Andreas Fritzbøger	cph-af167@cphbusiness.dk
Owais Dashti	cph-od42@cphbusiness.dk
Rasmus Taul	cph-rt91@cphbusiness.dk
Rabee Fawzi Abla	cph-ra157@cphbusiness.dk

cphbusiness

COPENHAGEN BUSINESS ACADEMY

Contents

1. GitHub Repo

GitHub Repo

2. Enterprise and Solution Architecture

What are the main differences between Enterprise and Solution Architecture. Describe the different roles that these play in designing large scale Systems?

2.1. The difference between Enterprise and Solution Architecture

Enterprise Architecture is for enterprise architects, who try to create a sense of how the organization is structured.

TOGAF (The Open Group Architecture Framework), a framework for Enterprise Architecture, has four levels for examining Enterprise Architectures. The first three levels are:

- **Strategic Architecture**. The highest level of them all is where the enterprise architects work with the CEO and CIO of the organization and try to understand the long-term strategy of the organization.
- **Segment Architecture**, or known as Subordinate Architecture. This level works at portfolio, program, and domain levels. This could be one of the departments in the organization, e.g. the HR, Finance, or Manufacturing department.
- **Capability Architecture**. This is project level. The business architects focus on a specific project within a domain of the organization.

The three levels cover the structure of the organization, looking at how it is built. This is seen as the Enterprise Architecture. The higher the level, the more the organization is covered. The lower the level, the more specific the architects can analyze and interpret the organization from its Segments and Capabilities.

However, the levels only covers, what the organization's capabilities are. How to resolve the capabilities, is done at the fourth and final level of TOGAF, called Solution Architecture. The team in Solution Architecture looks at the capabilities, defined by the others teams in the higher levels of TOGAF, and designs and delivers solutions to the capabilities.

Enterprise Architecture is for creating overview of the organization structure.
Solution Architecture is for delivering solutions to improve the organization structure.

2.2. The roles in Enterprise and Solution Architecture

The people defining an organization's capabilities, and delivering solutions to the capabilities, are called Enterprise architects. However, the Enterprise architect role is a term for everyone working within Enterprise Architecture. Each Enterprise architect can be an architect for a specific architecture. Those being the Business, Application, Data, or Technology Architecture, which are parts of the Enterprise Architecture of an organization.

The specific roles are:

- **Business architects**. Specialists study and interpret the business, process, and organizational structure and they are placed in the larger organization.
- **Application architects**. Those, who try to understand, how an application supports the business. The specialists also analyze, what data does it manage, and how does it interact with other applications.
- **Data architects**. They look at the Data Architecture, and analyze how the data flow within the organization and enable the business.

- **Technology architects**. The specialists, who examines how the technology platform supports the Application, Data and Business Architecture.

The roles are expected to have a great proficiency in their specific field of the Enterprise Architecture. A data architect is expected to be excellent at analyzing the Data Architecture, but not required to be an expert in understanding the Business Architecture.

Everyone has their own role to play, within the larger Enterprise architecture, with their own set of specific skills. Everyone is an enterprise architect, but specializes in specific areas in the organization.

3. Teams in Modern Architecture (Stefan Tilkov)

In his video called Practical Architecture what does Stefan Tilkov say about the role of teams in modern architecture. He refers to a book Team Topologies – what "team topologies" does the book describe? Do you agree from your own experience that the team is a core part of a successful project?

In his video, *Practical Architecture*, Stefan Tilkov emphasizes that team structure is a key element in modern software architecture, particularly for larger projects. Architects must consciously design team setups that encourage autonomy and independence. By aligning the organizational structure with the size and context of the project, the architecture can better support team needs.

For smaller, single-team projects, a dedicated architect may not be necessary. However, in larger multi-team projects, the presence of a defined architectural role or team becomes crucial. Effective communication and collaboration mechanisms, like regular architecture meetings or architecture roles within teams, are essential for maintaining alignment and ensuring smooth project execution.

Tilkov references *Team Topologies*, a book that stresses the importance of long-lived, autonomous teams in software development. The book suggests that the structure and collaboration between teams are critical to project success, advocating for well-defined, independent teams that can operate with minimal external dependencies.

From our experiences, the success of a project is heavily dependent on how well the teams are structured and empowered to make decisions autonomously. Clear communication, working iteratively, well-defined responsibilities, and minimal dependencies betweens teams typically enhance the efficiency and quality in the software development that we are a part of.

4. Some things are best done centrally (Stefan Tilkov)

Also in this video Stefan Tilkov explains why some things are best done centrally (for example at 23 min 30 seconds). Why do you think he is saying this? What does he claim are the benefits?

In *Practical Architecture*, Stefan Tilkov explains that while autonomy is important, some decisions must be made centrally to ensure effective collaboration and avoid chaos. Centralizing key decisions such as team structures, recurring team responsibilities, or interfaces between different parts of the system, enables teams to work autonomously without getting stuck by constant negotiations and individual preferences.

He suggests that centralized decisions, such as defining interfaces or standards, should remain stable for some period, e.g. for a few weeks. This stability allows teams to gain experience and

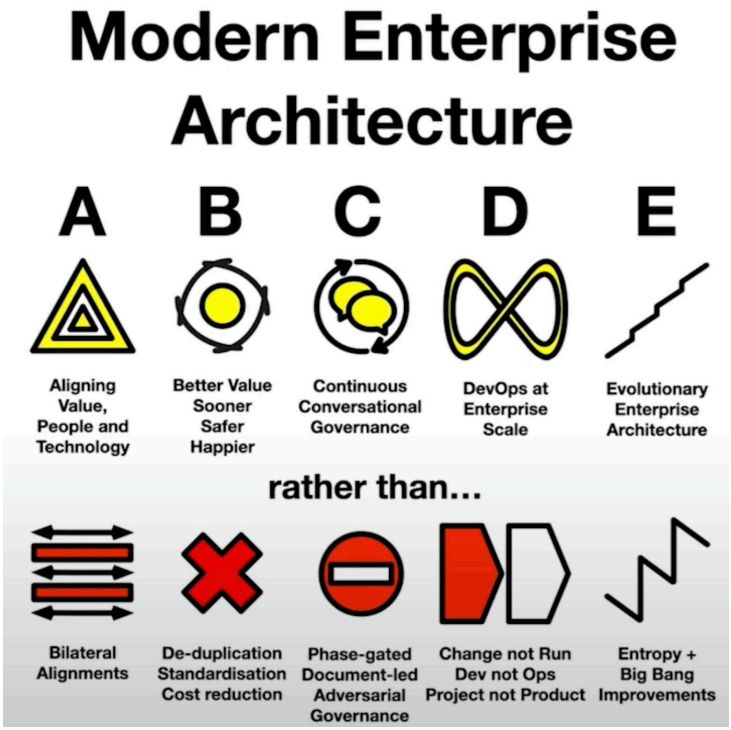
gather feedback before revisiting and modifying decisions. By centralizing these key aspects, the overall architecture becomes more cohesive and easier to manage.

Centralizing decisions helps prevent fragmentation and ensures that teams can operate more independently once those decisions are in place. It also reduces the cognitive load on individual teams by eliminating the need to constantly interact with other teams about recurring issues. This approach strikes a balance between autonomy and the need for broad consistency, promoting a smoother development process. Moreover, certain non-technical aspects, such as security, compliance, and privacy, should be centrally enforced across the organization. By using education, assessments, and training, architects can ensure these concerns are addressed in a consistent way, avoiding potential risks or inconsistencies.

5. The "ABCDE" framework

In the video Architecting for Outcomes Simon Rohrer describes old fashioned and modern enterprise architecture. He talks about the A B C D E of modern architecture to compare modern and legacy ways of working. Do you find his arguments persuasive and if so why?

Simon Rohrer's argues that modern enterprise architecture are compelling, because they contrast outdated approaches with the needs of today's dynamic businesses. He criticizes traditional, siloed structures and rigid, phase-gated processes as ineffective in a rapidly changing environment. Instead, he advocates for continuous governance, aligning value, people, and technology, and embracing evolutionary design. The "ABCDE" framework emphasizes delivering measurable, balanced outcomes like "Better value, Sooner, Safer, Happier," providing a practical and adaptable approach to modernizing enterprise architecture that is more responsive and scalable.



6. Continuous Conversation

In the same video he explains the concept of the "Continuous Conversation". In what ways does he say it benefits Saxo Bank? How does he connect the Continuous Conversation to the Devops Infinity Loop.

In the video, Simon Rohrer explains that the "Continuous Conversation" at Saxo Bank involves ongoing communication between architects and teams to ensure that decisions are made in real-time, rather than through formal phase-gated processes. This approach supports agility by allowing teams to quickly adapt, reduce complexity, and focus on delivering value. Rohrer connects this idea to the DevOps Infinity Loop by emphasizing that continuous feedback and collaboration mirror the DevOps model of constant integration and delivery, making architecture an evolving process aligned with business needs.

7. Integration Patterns for messaging

In the course of Systems Integration we will focus on building integrated Enterprise scale applications using messaging and there are a number of core Integration Patterns for messaging. We will describe 5 integration patterns for messaging and provide a use case for them + simple code examples.

7.1. Pipes and Filters Pattern

The Pipes and Filters pattern breaks a complex process into smaller, reusable steps (filters) connected by communication channels (pipes). Each filter performs a transformation on the data, and the pipes handle the message flow between filters.

- Use Case: In an e-commerce application, the order fulfillment process can be split into multiple filters: one for payment validation, another for inventory checks, and a final one for shipping. Messages (orders) pass through each stage, getting processed at every filter.

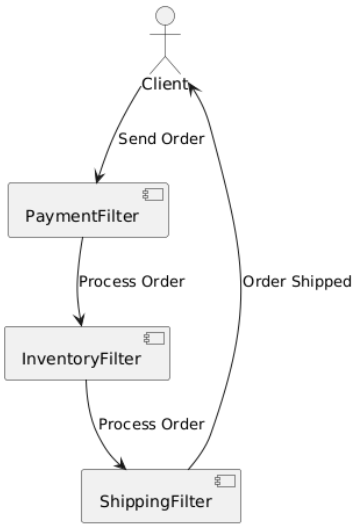


Figure 2: A series of filters process an order sequentially, with each filter performing a specific task before passing the result to the next filter.

Example in code:

```
class Filter {
  process(data) {
```

```
    return data;
  }
}

// Our first filter
class PaymentFilter extends Filter {
  process(order) {
    if (!order.paymentValid) {
      throw new Error("Payment failed!");
    }
    console.log("Payment processed.");
    return order;
  }
}

// Our second filter
class InventoryFilter extends Filter {
  process(order) {
    if (order.items.length === 0) {
      throw new Error("No items in inventory.");
    }
    console.log("Inventory check passed.");
    return order;
  }
}

// Our third filter
class ShippingFilter extends Filter {
  process(order) {
    console.log("Order shipped.");
    return order;
  }
}

// Our pipeline, class initialization, order
const orderPipeline =
[new PaymentFilter(), new InventoryFilter(), new ShippingFilter()];

let order = { paymentValid: true, items: ['item1', 'item2'] };

// applying each filter in the orderPipeline sequentially to the order object
orderPipeline.forEach(filter => order = filter.process(order));
```

7.2. Request-Reply Pattern

The Request-Reply pattern involves a sender (requester) that sends a request message to a recipient (replier) and waits for a reply message. This pattern is commonly used in synchronous communication but can also be used asynchronously with messaging systems.

- Use Case: A financial services application may use Request-Reply to check a customer's credit score. The service sends a request to the credit agency and waits for a response with the score.

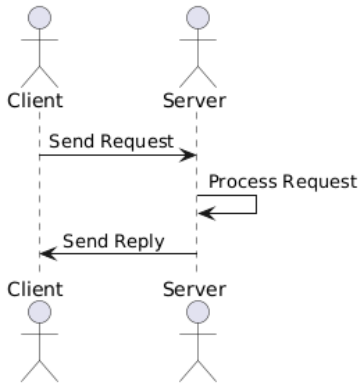


Figure 3: A client sends a request to a server, and the server processes it before sending back a reply.

Example in code:

```
const sendMessage = (message) => {
  return new Promise((resolve) => {
    setTimeout(() => {
      const response = { creditScore: 750 }; // Mocked reply
      resolve(response);
    }, 1000);
  });
};

const requestCreditScore = async (customerId) => {
  console.log("Requesting credit score...");
  const reply = await sendMessage({ customerId });
  console.log("Received credit score:", reply.creditScore);
};

requestCreditScore(12345);
```

7.3. Publish-Subscribe Pattern

In the Publish-Subscribe (pub-sub) pattern, messages are published to a message broker (Exchange), and multiple consumers (subscribers) receive the message. Subscribers can be added or removed dynamically without affecting the publisher.

- Use Case: A news application sends notifications of breaking news to all of its subscribers. Multiple users are subscribed to the same news feed, and when the news is published, all subscribers receive the notification.

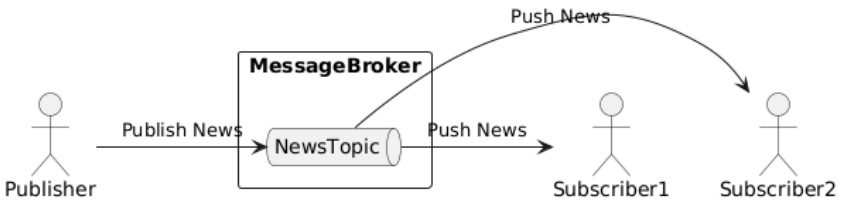


Figure 4: A publisher broadcasts a message to a topic in a message broker, which then pushes the message to all subscribed consumers.

Example in code:

```
class NewsPublisher {
  constructor() {
```

```
    this.subscribers = [];
  }

  subscribe(subscriber) {
    this.subscribers.push(subscriber);
  }

  publish(news) {
    this.subscribers.forEach(sub => sub.receive(news));
  }
}

class Subscriber {
  receive(news) {
    console.log("Received news:", news);
  }
}

const newsPublisher = new NewsPublisher();
const user1 = new Subscriber();
const user2 = new Subscriber();

newsPublisher.subscribe(user1);
newsPublisher.subscribe(user2);

newsPublisher.publish("Breaking News: New Event Announced!");
```

7.4. Message-Broker Pattern

The Message Broker Pattern is used to mediate communication between different applications by sending and receiving messages through a central component called a message broker. The broker acts as an intermediary, which means that the publisher and consumer don't need to communicate directly or even know about each other. The broker handles routing messages, making sure the correct consumer gets the correct message.

- Use Case: In an e-commerce system, when a customer places an order: then the publisher (Order Service) sends an order message to the message broker. The message broker routes the message to the consumer (Shipping Service), which processes the order and prepares the shipment.

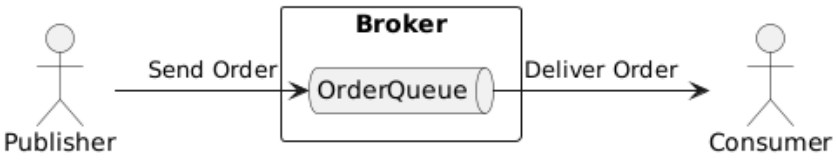


Figure 5: A publisher sends a message to a broker-managed queue, which routes the message to the appropriate consumer.

7.5. Scatter-Gather Pattern

In the Scather-Gather pattern, a request is sent to multiple recipients simultaneously (scatter), and their responses are gathered back into a single response. It is useful for scenarios requiring multiple services to process a request.

- Use Case: A travel booking system, e.g. Momondo, may send a request to several airline APIs to find the best flight price. The system gather responses from all airlines and returns the best option to the user.

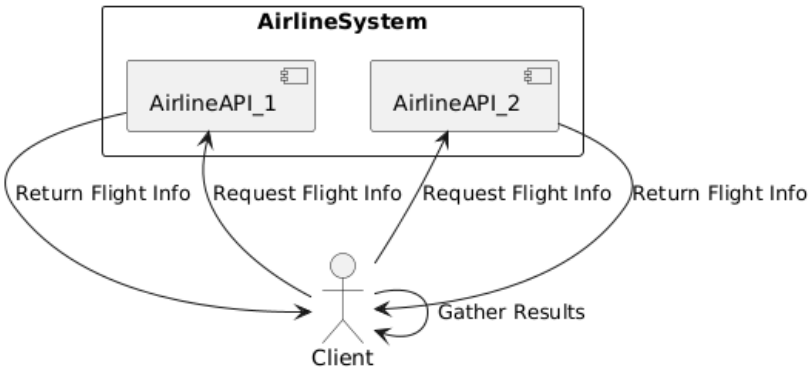


Figure 6: A client sends requests to multiple external systems (airline APIs) and gathers their responses to combine and process the final result.

Example in code:

```
const scatterRequest = (apis) => {
  return Promise.all(apis.map(api => fetch(api)));
};

const gatherResponses = async () => {
    const apis = ['https://api.airline1.com/flights', 'https://api.airline2.com/
flights'];
  const responses = await scatterRequest(apis);
  const bestFlight = responses.reduce((best, current) => current.price < best.price ?
current : best);
  console.log('Best flight:', bestFlight);
};

gatherResponses();
```

7.5.1. References

Enterprise Integration Patterns - Messaging Patterns Overview

8. Enterprise Integration Patterns 2

In Gregor Hohpe's talk Enterprise Integration Patterns 2 he describes the idea of conversations in a messaging architecture (from 18 minutes). He explains how a 'messaging' and 'conversation' architecture are different. What differences are there and what challenges does he describe for conversation based solutions (for example what is the difference between pub-sub and subscribe-notify).

8.1. Messaging Architecture:

- Asynchronous: Messages work asynchronously. That means the systems can operate independently from different systems.
- Decoupling: That means we reduce the dependencies between different systems.
- Focus on the message and follow the message.
- One-Way Messaging: In one-way messaging, a message is sent from a sender to a receiver without expecting a response. That means the sender the sender does not want to or need to know if the message was received.

- It involves queues, topics, channels, and more complex messaging models like request-response, publish-subscribe and more.
- Deals with transport: Pipes and Filters: Messages pass through filters connected by pipes. Each filter performs a function that, transforms the message as it passes through.
- Stateless: No memory of past interactions. Each request is independent.

8.2. Conversation architecture

- Focus on participants: All participants must agree on rules or guidelines to manage conversations.
- Two / multi-way: Conversation state and Endpoint state.
- Stateful: Remembers past interactions to improve future requests.
- Deals with resources: Creating an efficient interaction.

8.3. Conservation design

- Participant Roles: Between sender and receives.
- Message Types: Request, response, notification, and command.
- Protocol: The sequence for exchanging messages between participants. It ensures that messages are sent and received in a logical and expected order. ¨

8.4. Protocol covers:

- Message Ordering: Which messages should be sent and received?
- Error Handling: How to handle errors or unexpected messages.
- Acknowledgments: Ensures that messages are acknowledged to confirm receipt.

8.5. Challenges for conversation solutions

- Completeness.
- Deadlocks.
- Error states.
- Map from conversation state to participant's state.
- Difficult to represent state space.

8.6. Pub-Sub (Publish-Subscribe):

Pub-Sub works when a school needs to send a newsletter about sports or food so the student only wants sports topics. They will receive it.

- Publishers and subscribers are decoupled, which means they do not need to know about each other.
- Broadcasting: Messages are sent to all subscribers who have shown interest in a topic.
- Scalability: We can add new subscribers without impacting the publisher.
- Asynchronous: Messages are sent and received without waiting for an immediate response.
- Uses Pipes, queues, and Filters.

8.7. Subscribe-Notify.

- Subscriber and provider.
- Direct Notification: Subscribers are notified by the provider when an event happens.
- Tight Coupling: There is a tight coupling between the publisher and provider.
- Immediate Updates: Real-time applications.

9. The History and Role of Patterns

Also in Gregor Hohpe's video he explains the history and role of the idea of Patterns and Pattern Languages. How would you summarise what he explains – what do you see as the core requirements for patterns to be useful (see about 45 minutes in).

A concise summary of his explanation.

- The names of patterns are crucial (form a language).
- Patterns must have a stitch.
- Patterns have forces in Design decisions and teach (why and how).
- Patterns have the solution.
- Patterns must be recurrence (relevant)
- Patterns must be human-to-human communication. It is about design, not code.
- The value of the pattern in competition.
- The pattern has a sequence, they are ordered from the beginning with the largest.

10. The strangler pattern

In Simon Rohrer's article on modern enterprise architecture (link - also on Moodle in the folder for slides for this week) he goes in to more detail about the A B C D E of modern Enterprise Architecture he discusses the 'strangler' pattern for moving away from legacy systems – why is this seen as a complex problem and what does the strangler pattern do to help?

The "strangler pattern," as discussed by Simon Rohrer, helps manage the complex problem of migrating away from legacy systems. Legacy systems are deeply embedded in organizations, often highly interconnected, making them difficult to modernize or replace outright.

The strangler pattern addresses this by allowing gradual replacement: new functionality is developed and integrated while the legacy system continues to operate. Over time, parts of the legacy system are replaced or deprecated, "strangling" the old system as it transitions to a more modern architecture without causing major disruptions. This approach helps manage complexity and risk.

11. Core diagrams for Solution and Enterprise Architecture

In Jesper Lowgren's video Solution vs Enterprise Architecture Tutorial he describes three core diagrams that can be used to describe an architecture. What are they and what role do the play? Include examples.

Solution and Enterprise Architecture uses diagrams coming from Technology Architecture, and Business Architecture. Each of these have three diagrams.

The diagrams for Technology Architecture are:

- **Directives**. IT focused rules and compliance, which enables reuse through technical consistency.
- **Landscapes**. IT focused structures, which provides reference architectures for planning.
- **Designs**. IT focused requirements, which produces a blueprint for development or implementation.

They help architects organize IT.

The diagrams for Business Architecture are:

- **Principles**. Business focused rules, which enables reuse through business and technology consistency.
- **Visions**. Business focused structures, which aligns IT investments with business outcomes.
- **Requirements**. Business focused requirements, which form input into the design stage.

They help business leaders manage IT.

Solution Architecture and Enterprise Architecture, each make use of three digrams shared from both the Technology and Business Architectures.

The diagrams Solution Architecture use with examples:

- **Requirements (Business)** - The blueprints can be Use Cases and conceptual architecture, which has enough detail for a system to be developed, implemented, or integrated.
- **Landscapes (Technology)** - Showing how applications and databases integrate and interact in the organization.
- **Designs (Technology)** - Detailed view of interactions between people, processes, data, application, and technical infrastructure. It should at minimum show an application, data, and infrastructure architecture, and also a reference architecture, if it exists.

The diagrams Enterprise Architecture use with examples:

- **Visions (Business)** - A business capability that describes, what a business does, and not how a business does it, nor where it does it.

- **Principles (Business)** - Business principles, business capability models, and capability uplift roadmaps. They are used to drive application, data, and technology architectures.

- **Directives (Technology)** - Patterns, which are specific and uses technical language to tell, how projects and code should be developed.