EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

ENTERPRISE INTEGRATION PATTERNS

Author: Tibor Dulai

MAGYARORSZÁG KORMÁNYA

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

ENTERPRISE INTEGRATION PATTERNS

1. Introduction

Author: Tibor Dulai

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

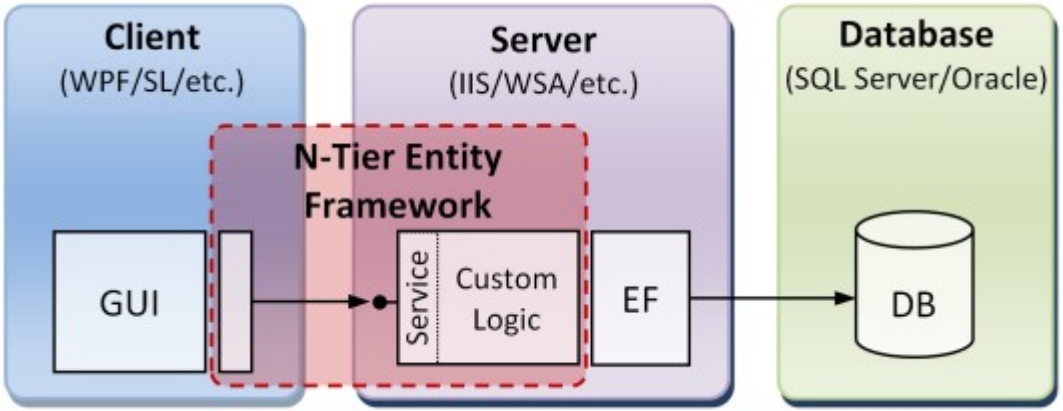
Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

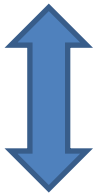
Enterprise integration

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



Single application with a distributed n-tier architecture



Integrated applications

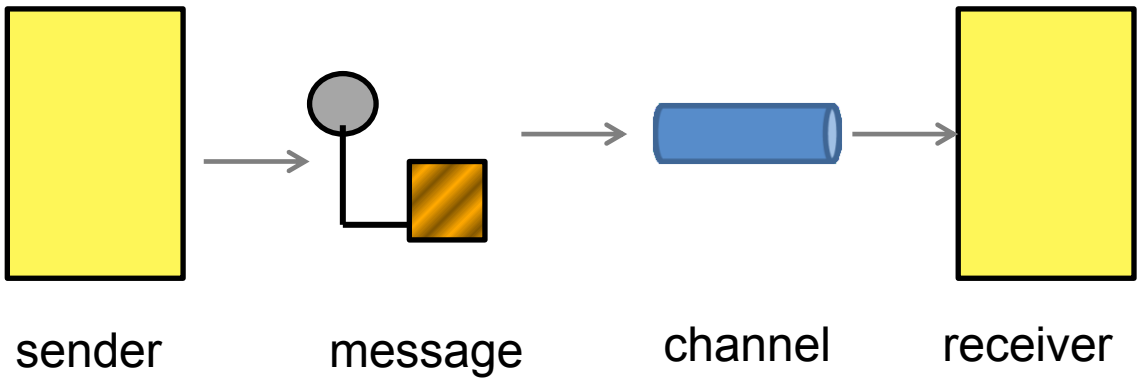
(„run by itself”, „loosely coupled”)

Messaging

This technique will be in the center of our interest.

We will get to know with the following concepts:

- message
- messaging system
- „send and forget” approach
- callback
- asynchronous calls



Be aware of …

Although, integration need is frequent (e.g. Calendar synchronization, development-market side), we have to keep in mind, that:

- Networks are **unreliable**: different segments, delays, interrupts

- Networks are **slow**: performance problems (vs. local method call)

- Applications are **different**: in programming language, in operation platforms, data formats. An interface is needed.

- **Changes**: it can cause an avalanche of changes. That's why an integration solution needs to minimize dependencies, for example by loose coupling.

Integration approaches

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

1. **File transfer**: agreement is needed on:
   - the name and location of the file
   - the format of the file
   - the timing of when it is written or read
   - who will detete it.

2. **Shared database**: applications share the same database located at a single place. There is ***no direct data transfer*** between the applications.

3. **Remote Procedure Invocation**: an application makes some of its functionality accessible to other applications. ***Real time*** and ***synchronous*** communication.

4. **Messaging**: Messages are published to a ***common message channel***. They are read ***later***. ***Asynchronous*** communication. Agreement is required on:
   - the channel
   - the format of the message.

Examples for synchronity

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Telephone call
synchronous

Voice mail
asynchronous

Basic concepts

Messaging: a technology that enables high-speed, asynchronous, program-to-program communication with reliable delivery of *message*s via channel (queue) between the ***sender (producer)*** and ***receiver(s) / (consumer(s)***).

Channel: behaves like a collection or array of messages, is shared across multiple computers and can be used concurrently by multiple applications.

Message: a data structure (string, byte array, record, object). It can ***represent data***, ***command*** or ***event***. It has a ***header*** (meta information for the messaging system) and a ***body*** (data for the receiver) as its parts.

Messaging system (or message-oriented middleware – MOM): a separate software system what provides messaging capabilities. It is needed, because networks are ***unreliable***, or the receiver is ***not ready to receive*** the sent message. Messaging system ***repeats*** trying to transmit the message ***till it succeeds***.

Comparison of messaging systems and database systems

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Messaging system

- manages messaging

- an administrator has to configure the system with the channels

- goal: transmit the messages in a reliable way

Database system

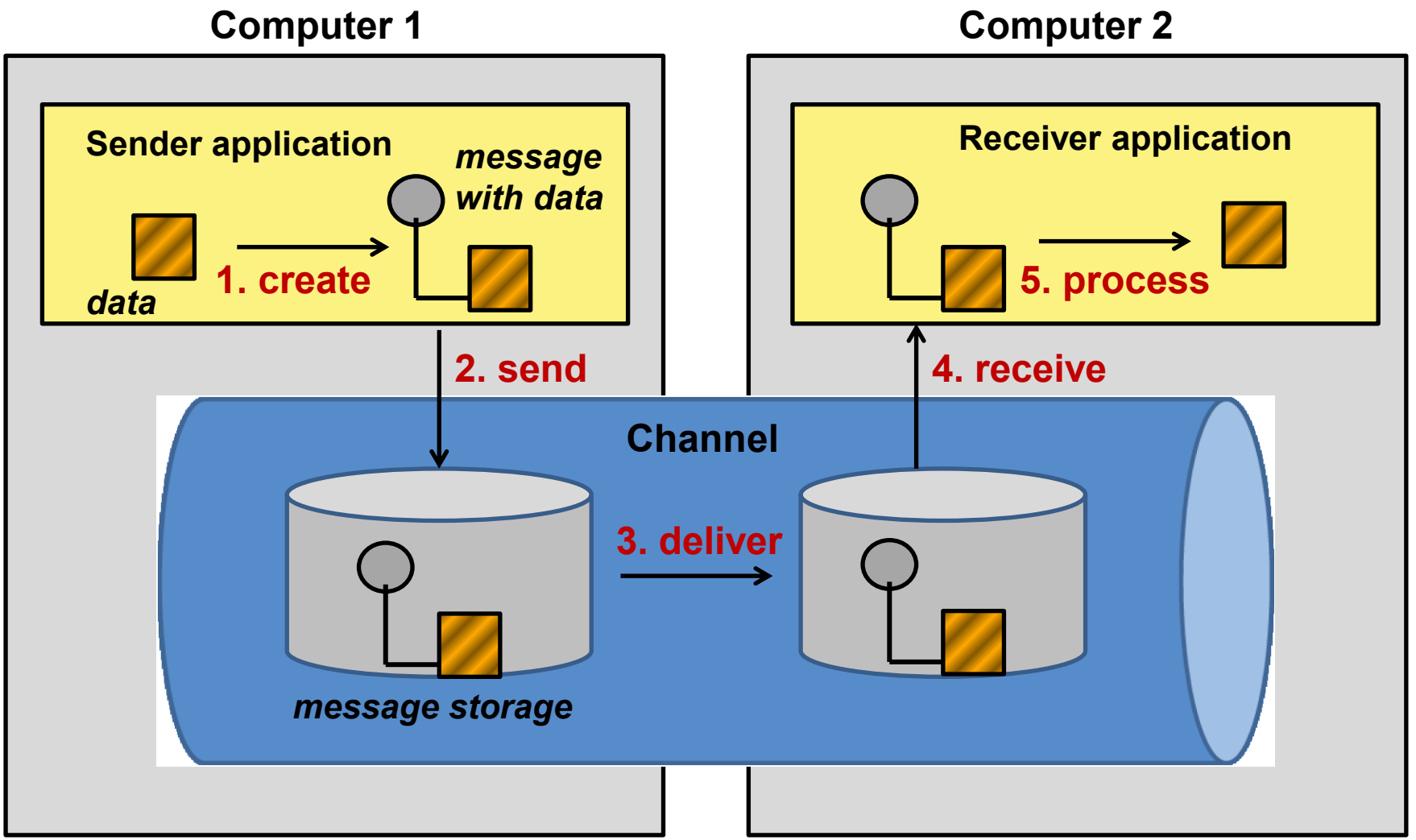
- manages data persistence

- an administrator has to populate the database schema for an application's data

- goal: each data record has to be safely persisted

Message transmission

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



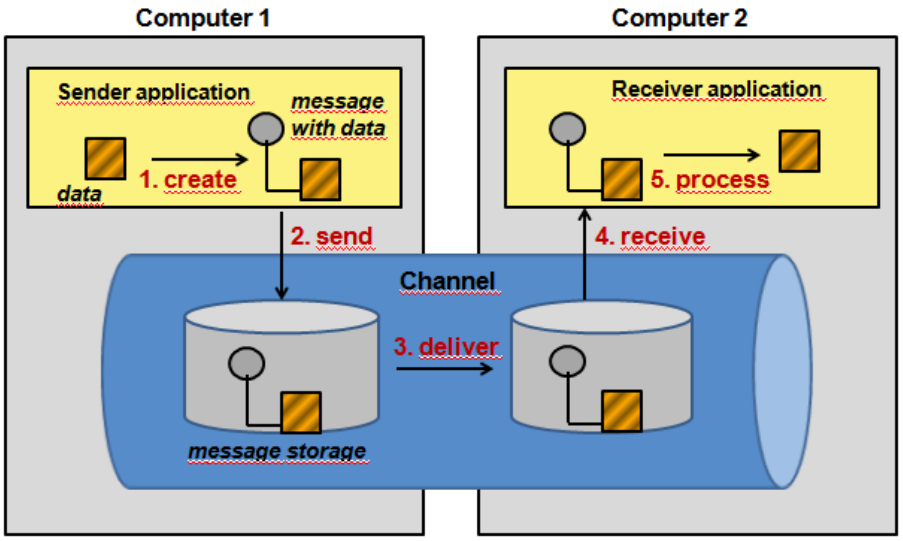
Transmission „style”

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

„Send and forget”: after sending go on to another work. The sender can be sure that the messaging system transmits the message to the receiver.

+

„Store and forward”: store on the sender site, deliver, then store on the reveiver’s site. Delivery is repeated until a successful action is reached.



Not only simple data-delivery happens to the receiver, because wrapping data into a message and give that to the messaging system have the following properties:

- assures **reliability, exactly one copy** on the receiver’s site

- delegates messaging task to the **messaging system**

Benefits of messaging I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Messaging is **more immediate** than *File Transfer*, **better encapsulated** than *Shared Database*, and **more reliable** than *Remote Procedure Invocation*

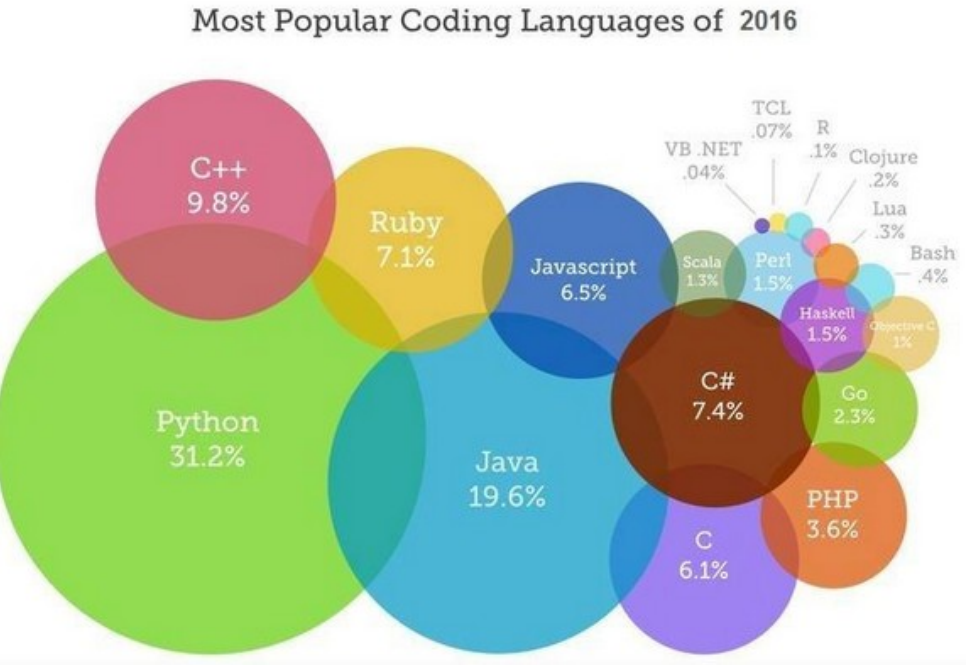
+

Remote communication: data serialization (e.g. byte stream) needed. If there is no remote communication then messaging is not needed (e.g. shared memory is enough).

Benefits of messaging II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Platform/language integration: a demilitarized zone of middleware is required to negotiate between the applications (e.g. by the lowest common denominator, such as flat data files with obscure formats). The messaging system is an ***universal translator***.

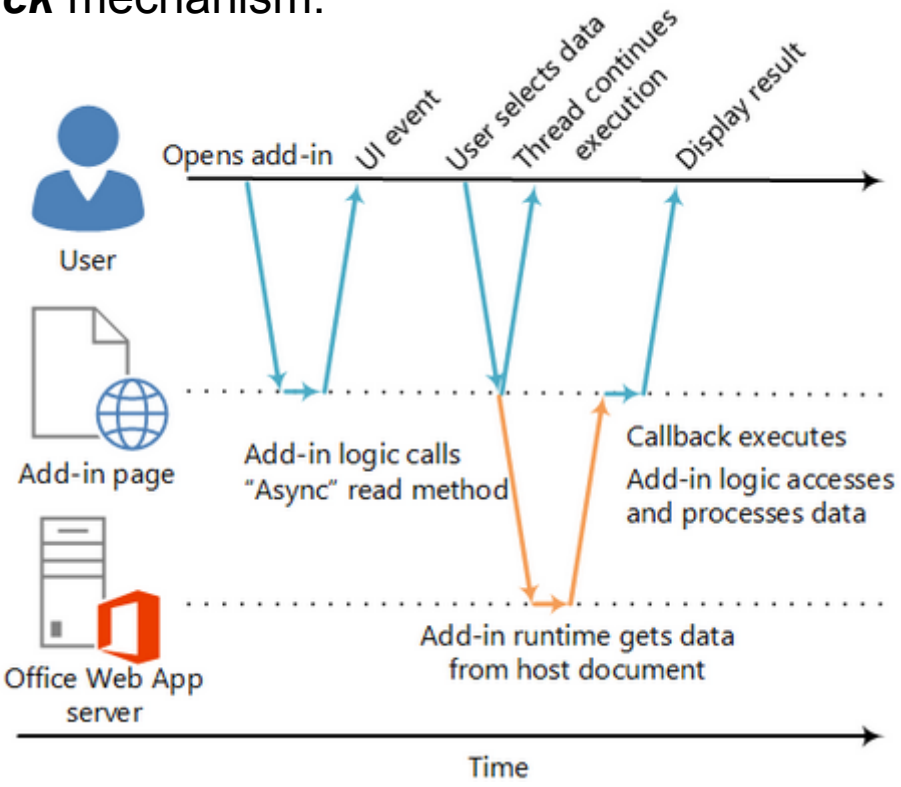


http://blog.codeeval.com/codeevalblog/2016/2/2/most-popular-coding-languages-of-2016

Benefits of messaging III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Asynchronous communication: „***send and forget***” approach is enabled. The sender has to wait for the message only to be stored successfully in the channel. Acknowledgement or notification by the result needs another message and a ***callback*** mechanism.



https://docs.microsoft.com/en-us/office/dev/add-ins/develop/asynchronous-programming-in-office-add-ins

Benefits of messaging IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Efficient timing of message sending: the sender can ***batch requests*** to the receiver. Both applications can run at ***maximum throughput***. Otherwise, in case of synchronous communication, the sender should wait for the receiver to finish the processing before sending a new message.

Benefits of messaging V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Throttling: in case of too many simultaneous requests, the ***receiver is able to control the rate*** of consuming the requests (***without blocking*** the sender – in contrast to e.g. remote procedure call).

Reliable communication: because of „***store and forward***" approach. The message is stored on both the sender's and the receiver's computer (memory, disk, …). The problem in simple forwarding is the unreliability of the networks or the receiver is not ready. The messaging system ***retries*** sending the messages until it succeeds.

Benefits of messaging VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Disconnected operation: messaging is ideal for applications which has to operate often disconnected and synchronize when the network is available.

Thread management: By applying asynchronous communication, there is ***no need for blocking*** the sender applications. Instead of that ***callbacks*** are used. The only thread that blocks is the small, known number of ***listeners*** waiting for replies. ***After crash*** they can ***be re-established easily***.

Benefits of messaging VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Mediation: for an application, a messaging system seems like a ***directory of other applications or services*** which can be integrated. After disconnection ***reconnection*** needed only with the messaging system. ***Redundant resorces*** can be applied (it may have an effect on availability, fault-tolerance, balance load, and QoS).

These benefits may influence both the development and also the strategic decisions of an enterprise.

Asynchronous messaging - Challenges I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Asynchronous approach is very useful in integration, but also has some difficulties:

Complexity: to develop an asynchronous system is more difficult. ***Event-driven programming model*** has to be used with several ***event handlers***, instead of simply calling methods from each other. Debugging is also a harder task.

Synchronous approach	Asynchronous approach
Simple method call	Request message Request channel Response message Response channel Correlation identifier Invalid message queue

Asynchronous messaging - Challenges II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Sequence of messages: the time when a message is delivered is not guaranteed. I means that the ***sequence of messages can be modified*** during delivery.

Need for synchron: some applications prefer or need synchronous approach (e.g. selecting a product in the web shop we want to see its price right away). In some cases messaging system ***has to bridge the gap*** between the synchronous and asynchronous approaches.

Asynchronous messaging - Challenges III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Performance: it is not the best choice to transport lots of small messages because of the message ***overheads***.
E.g. if syncronization is needed between two systems, its two main steps:

- replication of data (ETL – export, transform, and load tools perform better than messaging systems)
- keep the systems in sync (messaging is a good choice for that)

Limited platform support: many messaging systems are ***not available*** on all platforms. (E.g. FTP is a more common solution.)

Asynchronous messaging - Challenges IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

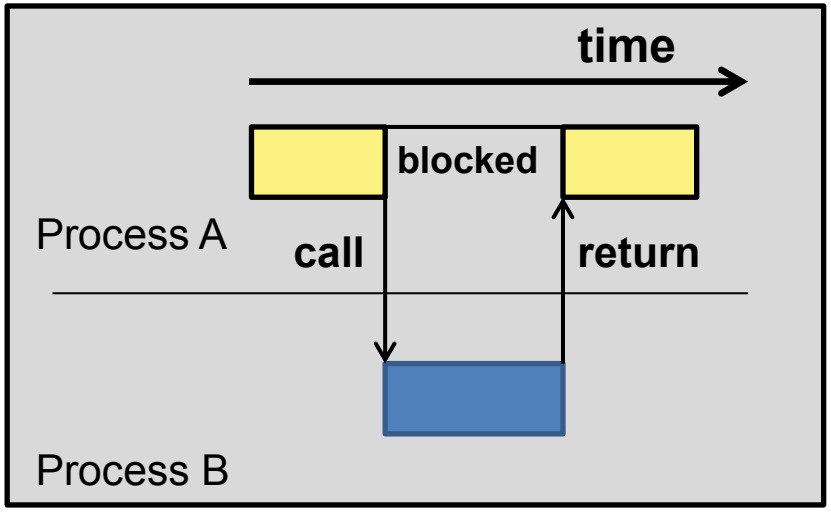
Vendor lock-in: messaging system implementations often rely on ***proprietary protocols***. Even JMS does not control the physical implementation. It results that messaging systems are often ***not compatible*** with each other.

It results in a strange challenge:
we have to integrate multiple integration solutions.

Synchronous vs. Asynchronous communication

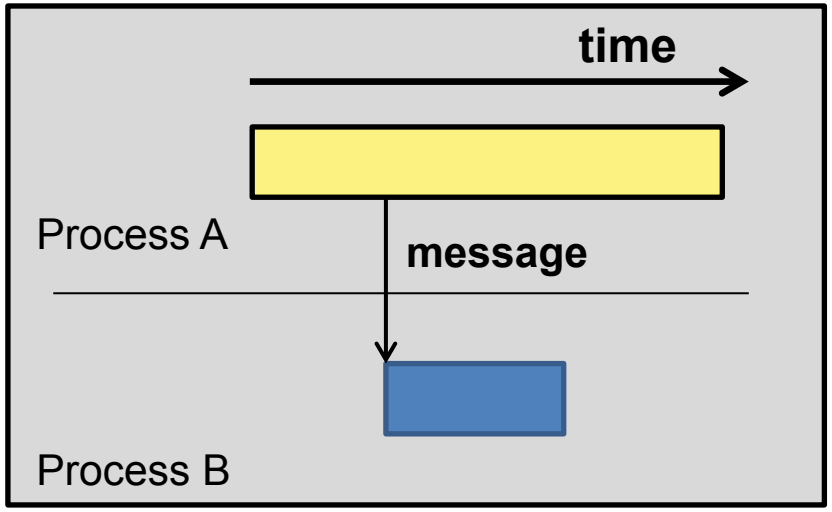
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Synchronous call



- the calling process is **halted** (even in case of RPC, when Process B may run on another computer in a different process)

Asynchronous message



- delivery **retried** until it succeeds
- „**send and forget**” approach

Properties of asynchronous communication

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Multiple threads: ***concurrent*** run of sub-processes that ***improves performance*** but it makes ***debugging more difficult***. It makes possible for some sub-processes to progress while others may wait for external results.

Callback: it is needed ***if*** a caller intends to be ***notificated by the result*** of the call. It ***improves performance***, but the caller has to be able to:

- ***remember*** (based on the result) which call the result belongs to
- ***process the result*** even the caller is in the middle of other task

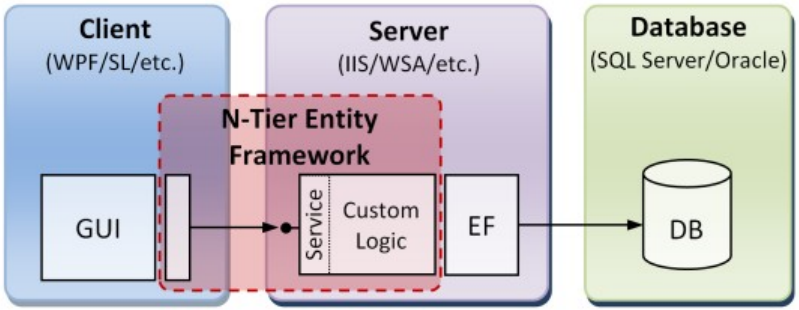
Sub-processes can be executed in any order: it has also positive effect on the ***performance*** (one can progress even if the other has to wait for an external result), but we have to ensure that:

- sub processes can run ***independently in any order***
- the caller ***remembers*** (based on the result) which call the result belongs to

Distributed applications vs. Integration

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

An n-tier architecture



- parts are ***tightly coupled***, they are ***dependent*** directly on each other, so one tier can not function without the orthers

- communication is ***synchronous***

- its users mainly accept only ***rapid system response***

⬇

application distribution

Integrated applications

- ***independent, loosely coupled*** applications, each of them can run by itself

- each application deals with a ***specific set of functionality*** and ***delegetes*** them to other apps

- ***asynchronous*** communication, ***concurrency***, ***broader time horisons***

Commercial messaging systems

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Benefits of asynchronous messaging → **significant market**

Groups of messaging vendors' products:

I. Operating systems or DB platforms

e.g. **Microsoft Message Queuing (MSMQ)**
/from Windows 2000, Windows XP, …/
(accessible through APIs: System.Messaging namespace, COM components, part of .NET)

Oracle AQ

II. Application servers

e.g. **Java Messaging Service (JMS)**
/from J2EE v1.2 specification; since then all J2EE app. Servers (e.g. IBM WebSphere, BEA WebLogic, too) provide an implementation of JMS spec.; SUN delivers a reference implementation with the J2EE JDK/

III. Enterprise Application Integration (EAI) suites

Usually include JMS or other client APIs.

- **IBM WebSphere MQ**
- **Microsoft BizTalk**
- **TIBCO**
- **WebMethods**
- **seeBeyond**
- **Vitria,** etc.

IV. Web Services toolkits

Very popular segment. Active work on standardizing reliable message delivery over web services, e.g.:

- **WS-Reliability**
- **WS-ReliableMessaging**
- **ebMS**

Vendors' terminology

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Enterprise Integration Patterns	Message Channel	Point-to-Point Channel	Publish-Subscribe Channel	Message	Message Endpoint
Java Messaging Service (JMS)	Destination	Queue	Topic	Message	Message Producer, Message Consumer
Microsoft MSMQ	Message Queue	Message Queue		Message	
WebSphere MQ	Queue	Queue		Message	
TIBCO	Topic	Distributed Queue	Subject	Message	Publisher, Subscriber
WebMethods				Document	Publisher, Subscriber
SeeBeyond	Intelligent Queue	Intelligent Queue	Intelligent Queue	Event	Publisher, Subscriber
Vitria	Channel	Channel	Pub./Sub. Channel	Event	Publisher, Subscriber

Pattern form

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Pattern = decision + consideration

Pattern language = a web of ***related patterns***, guiding through a ***decision process***. Perfect tool for documenting an expert's knowledge.
It teaches how to solve a limitless variety of problems within a bounded problem space.

A pattern ***does not only offer a solution for a problem***, but also tells why to apply that, what other possibilities are and what kind of constraints they have.

Patterns are ***prescriptive***: they also describe what to do to solve a problem (not only: how)

There are several pattern forms. We use ***Alexandrian form*** *(named after Christopher Alexander)*, because in this case it is easy to identify important sections at a glance, ***visually***.

The elements of the applied pattern structure I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Name: ***ID*** of the pattern + ***describes what the pattern does*** (well applicable in a conversation between designers)

Icon: the representation of the pattern in the ***visual language***. Well applicable in ***diagram***s. With their help the ***composability of patterns*** can easily be expressed.

Context: the „***stage of the problem***”: what you have worked on that made you face with the problem. It often refers to other patterns you may have already applied.

Problem: ***the difficulty*** you are facing ***in a form of a question***. ***One bold, and indented sentence***, which is a perfect basis to decide whether the pattern is relevant or not.

The elements of the applied pattern structure II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Forces: the ***constraints*** that make the problem difficult to solve. Often consider ***other solutions*** which seem ***promising but don't work***.

Solution: a ***template*** that describes ***what to do to solve the problem***. It is well applicable for the variety of circumstances represented by the problem. It is ***one bold, and indented sentence.***

Sketch: an ***illustration of the solution.***

Results: the ***details*** about how to apply the ***solution*** and how it resolves the forces. It may contain ***new challenges*** which are resulted by the applying of the pattern.

Next: ***other patterns to be considered after applying the current pattern***. This relation between patterns ***leads to*** the ***pattern language*** (instead of a simple pattern calatog). (since the applying of a pattern leads to problems that necessitate other patterns to solve them)

The elements of the applied pattern structure III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Sidebars: more ***detailed technical issues or variations*** of the pattern. This part of the pattern is placed ***visually apart*** from the other sections.

Examples: one or more examples of the pattern's application. They may differ in their elaboration, they ***can be skipped*** without loosing any infomative part of the pattern.

Patterns are for teaching how to solve a new problem not described in the tutorials.

Diagram notation

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

An ***integration solution*** usually consists of many different pieces: endpoints, channels, messages, routers, etc.
However, ***there is not a common, complete notation*** for that.

Maybe UML is the closest one:
- it has class diagram, interaction diagram, but
- it does not deal with messaging solution.

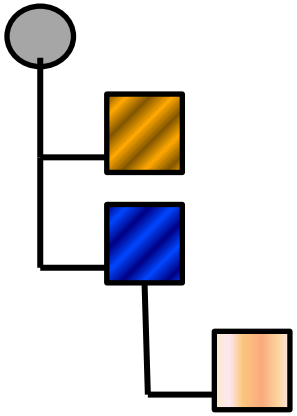
UMLEAI (UML profile for Enterprise Application Integration) provides a tool to describe message flows. It is
- very ***useful for*** basis of ***code generation***, but
- ***does not cover all the patterns*** of the pattern language
- ***sketch-style notation is needed*** instead of precise visual specification

That's why a new notation was introduced, whose elements are as follows …

Diagram notation - Message

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message:



The message is sent to a component over a channel. The word „component” is used very loosely.

The notation of a message is a ***small tree with a round root and nested, square elements*** (shaded or colored to highlight their usage in a particular pattern).

Highlighting the ***structure of a message,*** we are able to ***follow its modification*** (add, re-arrange or remove fields) in a visual way.

Diagram notation - Channel, Component

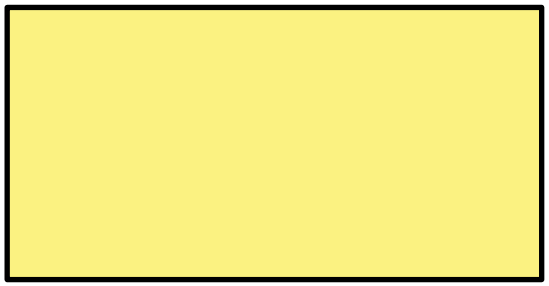
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Channel:

or

3D channel notation is used when we intend to highlight the channel itself.

Component:



For describing the application design (e.g. written in C# or Java) UML class and sequence diagrams are often used.

About the examples

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

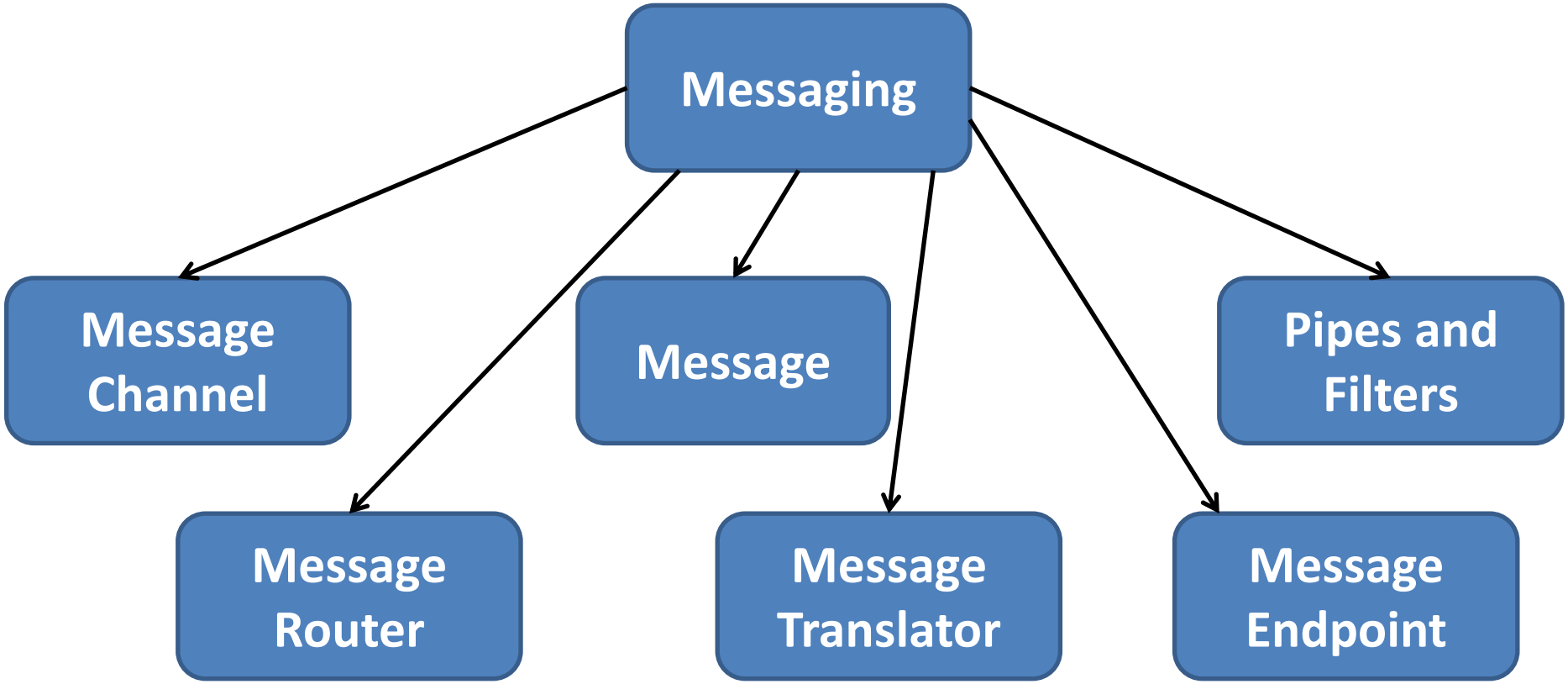
Because of ***focusing on readability***, only the main parts of sample codes will be presented. They are lack of error checking and thoughtful programming presentation.

The Java examples are based on **JMS 1.1** specification (and **J2EE 1.4** specification).

The Microsoft .NET examples are written in **C#** and are based on the **1.1** version of **.NET** Framework.

Root patterns

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



Main interests of user groups

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The most important root patterns for the following experts are:

- **System administrators**: Messaging Channels and System Management

- **Application developers**: Messaging Endpoint and Message

- **System integrators:** Message Routing, Pipes and Filters and Message Translator

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

THANK YOU FOR YOUR ATTENTION!

Reference:

Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns – Designing, Building and Deploying Messaging Solutions, Addison Wesley, 2003, ISBN 0321200683

www.enterpriseintegrationpatterns.com

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

ENTERPRISE INTEGRATION PATTERNS

2-3. Solving Integration Problems with Patterns

Author: Tibor Dulai

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

Need for integration I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Enterprises have **100s-1000s of applications**
(3rd party systems, web sites, SAPs, departmental solutions)

Its reasons:
- **Hard to create a single comprehensive** business application (attempts: ERP vendors like SAP, Oracle, PeopleSoft)
- Enterprises like the **flexibility for selection** of systems + **individual needs**

↓

Vendors produce focused application with **specialized** core function. **New functionalities** often have to be added.

↓

Integration is needed

Need for integration II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Defining a clear **functional separation** between systems is **hard**.

e.g.
customer disputing a bill → ? → customer care / billing function

A single business transaction (from the point of view of the customer) require the **coordination of many systems**.

An example: ordering an airplane ticket:

- validation of the customer ID
- verification of the customer credit stand
- checking ticket availability
- fulfilling the order
- providing the ticket
- computing sales tax
- sending a bill

Integration: supports **common business processes** and **data sharing** across applications. It needs to provide **efficient, reliable and secure data exchange** between multiple enterprise applications.

Integration challenges I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Multiple applications
running on
multiple platforms
in
different locations.

- Requires a **shift in corporate politics:** Conway's law: "Organizations which design systems are constrained to produce designs which are **copies of the communication structures** of these organizations."
IT groups were assigned to functional areas. Now communication is needed among multiple computer systems, among business units and IT departments.

Integration challenges II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- **Critical functions:** after the integration of critical business functions, **their proper functioning becomes vital** (failure may cause lost orders, misrouted payments, diasppointed customers)

- **Limited control** over „legacy” systems and packaged applications. Sometimes it would be easier to implement part of the solution, but technically it is not allowed.

- Only **few standards**: although, there are useful standards in XML, XSL and Web services, there is a **lack of interoperability between „standard-compliant” products** (e.g. CORBA, what offered a sophisticated technical solution for integration).

Integration challenges III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Existing standards address **only a fraction of the** integration **challenges**: Binding all data exchanges to XML would result in a **too rigid system**. Moreover, a common presentation **does not imply common semantics** (e.g. the concept of "account"). Resolving semantic differences is a difficult and time-consuming task

- **Operating and maintaining** integrated systems is very hard. **Deployment, monitoring, and trouble-shooting** of these systems are complex tasks.

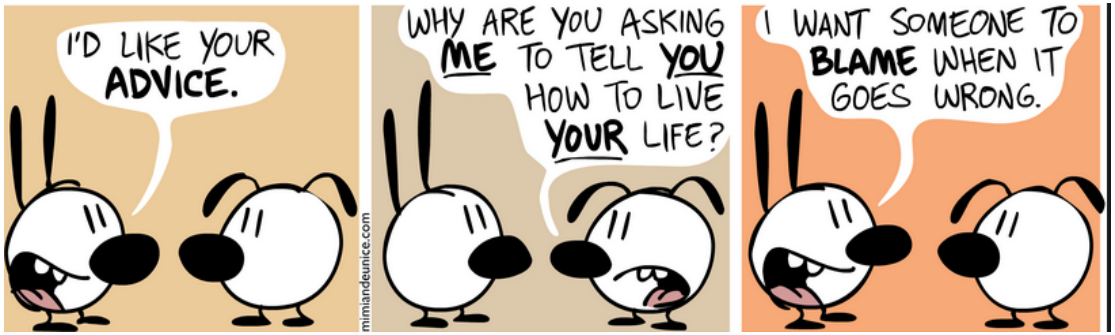
What is behind of integration patterns

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Integration is a **wide-scale and difficult** problem.
People who can deal with it easily usually have **faced and solved enough** integration problems.

If they face a new problem, they
- can **compare** it to a prior, already solved problem
- they **learned the pattern** of the problem and its solution by trial-and-error or from experienced people.

Patterns are not copy-paste code samples but **advices for solving often recurring problems.**



Integration project types

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

We can integrate several things (computer systems, people, enterprises, etc.)

The 6 main integration project types are:

- **Information portals**
- **Data replication**
- **Shared Business Functions**
- **Service-Oriented Architectures**
- **Distributed Business Processes**
- **Business-to-Business Integration**

These pure types are **often combined**: e.g. in distributed business processes data replication often takes place.

We characterize them in more details, one-by-one.

Integration project types
Information portal

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

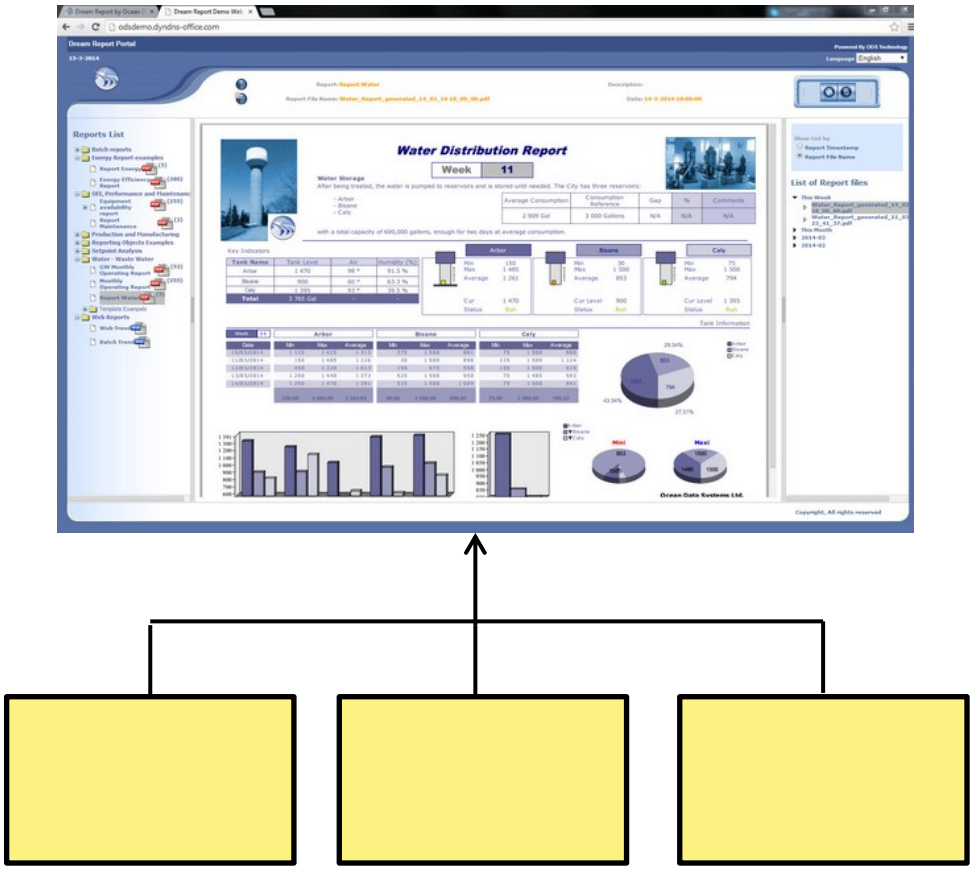
It aggregates information from **multiple sources** into a single display.

Based on their complexity it has three types:

- Simple: usually divides the screen into multiple **zones** (their sources usually differ).

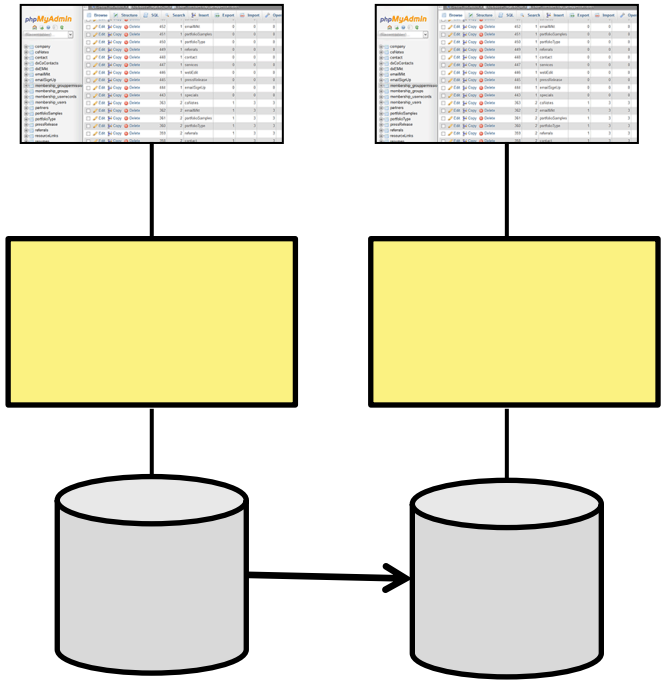
- More complex: there is a **limited interaction** between its zones (e.g. it picks up an item from zone I which causes to refresh the detailed information in zone II.

- Much more complex: blur the line between a portal and an **integrated application**.



Integration project types
Data replication

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



It is a possible solution if many business systems **needs the same data** (e.g. customer's contact is needed for billing, shipping, customer relations, etc.)

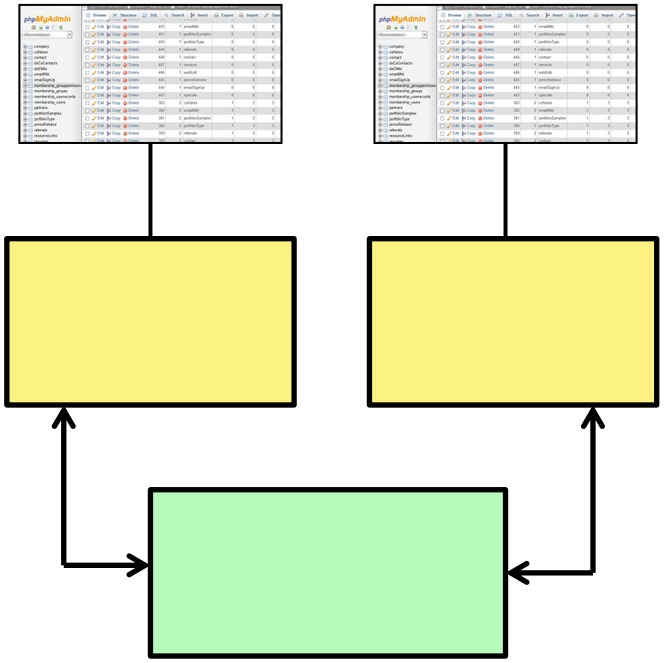
Because of the systems' own **local database**, data replication is needed for updating all the systems.

Different ways of data replication:

- build **replication functions into the database**
- export data into **files** and re-import them into the other systems
- use **message-oriented middleware** to transport data records

Integration project types
Shared Business Function

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



In business applications some functions appear **more than once** (e.g. verification of address vs. postal code).

It is more efficient if we **implement once** these functions as shared business functions and **make them available** as services to other systems.

Has the same needs as data replication.

E.g. the cholice between the two approaches: do we **always request** the customer data by a function when it is needed or we always **store a redundant copy**?
The answer can be influenced by the **frequency of change** and the **amount of control** we have over the systems.

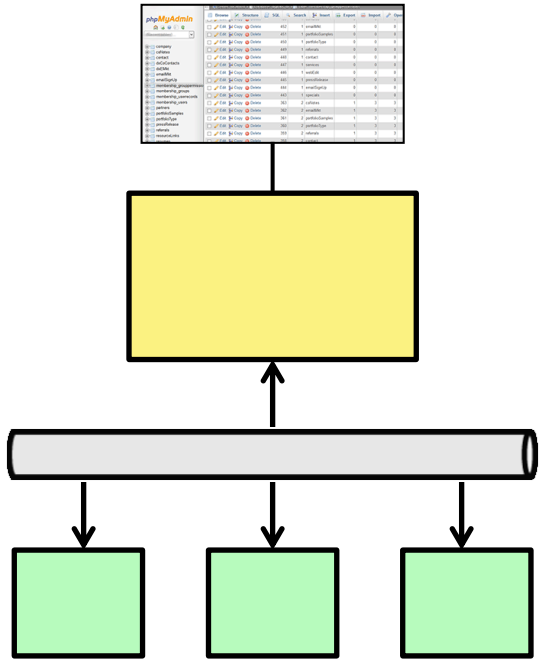
Integration project types
Service-Oriented Architecture

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Its basic elements are the **shared business functions** (services)

They are organized into Service-Oriented Architecture (SOA), if:

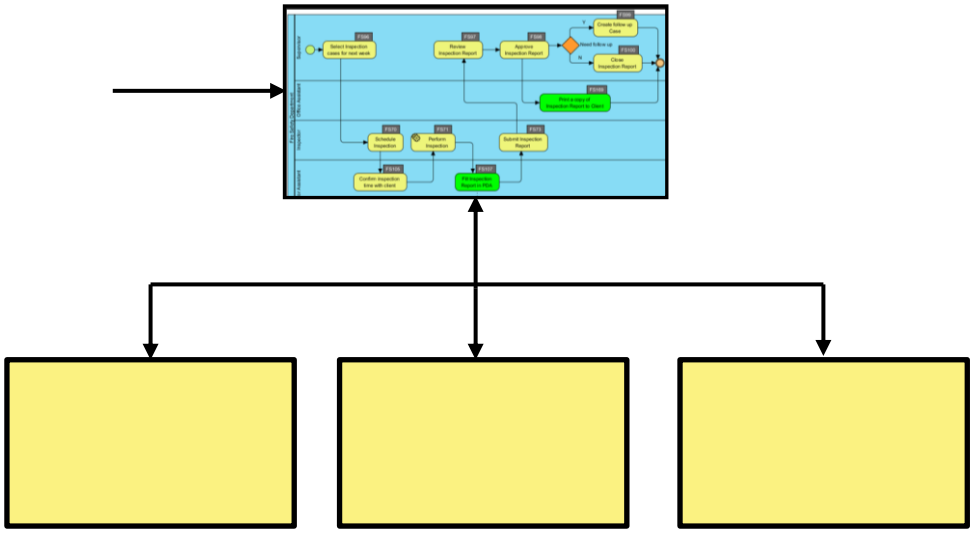
- a **service directory** exists **(a centralized list of all available services)**
- each service describes its **interface** **(communication contract)**



The key features of SOA are **service discovery** and **negotiation**.
They blur the line between integration and distibuted applications.
Service calls are easy and consistent. That's why new applications can be developed by calling the remote services of other applications (integration).

Integration project types
Ditributed Business Process

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



In this case all the relevant functions are implemented in **standalone applications** and there is a **business process management component** which **coordinates** the applications and manages the execution of the business function across the multiple applications.

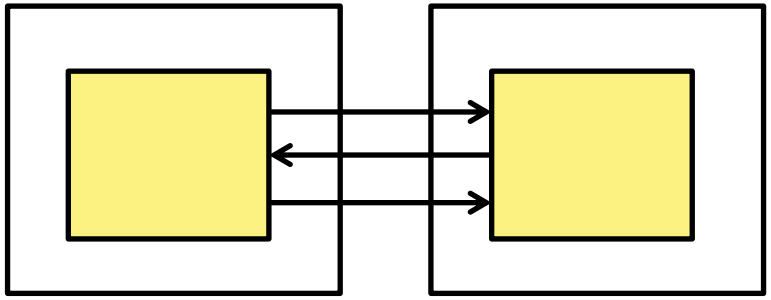
The business functions of the applications can be handled like services, and they can be organized into a SOA. So, the line between SOA and distributed business functions can blur.

Integration project types
Business-to-Business Integration

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Till now we have dealt with business processes **inside one enterprise.**

However, business functions should be available by **outer business partners** in many cases. (e.g. a customer intends to know the price of a product and it triggers the enterprise to get knowledge from its supplier about the product)



These cases require **integration between the business partners**.

This integration-type requires new issues to be solved:

- **transport protocols**
- **security**
- **standardized data formats**

The role of assumptions in coupling

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Loose coupling becomes popular thanks to **Web services architectures.**

In loose coupling the set of **assumptions** about the communicationg parties (components, applications, users, services, programs) **are reduced.**

In case of **more assumptions**, the communication is **more efficient**, but **less tolerant to changes and to interruptions** (tightly coupling).

For example, the assumptions of a local method invocation are:

- same process/machine
- same languages
- exact number and types of parameters
- immediate call
- synchronous communication

If we apply the approach of local method call in RPC

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

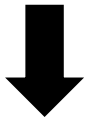
For **remote communication** (e.g. RPC/RMI in CORBA, Microsoft DCOM, .NET Remoting, Java RMI, RPC-style Web services), many integration approaches **follow the semantics of local method call**.

Its **reason** is that:

- developers are **familiar** with this syncronous solutions
- we got **more time to decide** about a component to be **local or remote**.

But: several **assumptions** of local method call **are invalid** in remote communication. The problems:

- due to different **programming language**
- remote call is **slower** than a local one
- **synchronous** call (wait)
- possible **network problems** (availability)
- **security** (authentication, evesdropping)
- possible change of remote method **signature**



Hard to maintain and poorly scalable solutions

A small integration example

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Bank transfer

Front end: web application

Back end: the financial system (which makes the transfer)

For the simplicity we assume the necessary data for the transfer are:

- **name**
- **amount** of money

We analyze **two possible integration solutions** between the front end and the back end.

We start with a popular but **tightly coupled** solution and releasing the **assumptions** one-by-one we get to a **loosely coupled** solution.

The „bank transfer” example I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

One evident solution of the example is to apply **TCP/IP.**

Several **positive properties**:
-TCP/IP is very **popular, wide supported** (every important programming language and opertaion system has TCP/IP stack)
- **easy** connection **independent**ly of the operating system and programming language

C# example:

```
String hostName = "www.example.com";
int port = 80;
IPHostEntry hostInfo = Dns.GetHostByName(hostName);
IPAddress address = hostInfo.AddressList[0];
IPEndPoint endpoint = new IPEndPoint(address, port);
Socket socket = new Socket(address.AddressFamily,
SocketType.Stream, ProtocolType.Tcp);
socket.Connect(endpoint);
byte[] amount = BitConverter.GetBytes(1000);
byte[] name = Encoding.ASCII.GetBytes("Tibi");
int bytesSent = socket.Send(amount);
bytesSent += socket.Send(name);
socket.Close();
```

The „bank transfer” example II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

But the **TCP/IP**-based integration is **tightly coupled** solution and may cause some problems because of its assumptions:

Assumption I: Platform independence – internal data representation format:

Platform independence works **only for simple messages**. We used **byte streams** in our code sample (BitConverter class converts data into byte arrays using the memory representation of the original data).

But:

I. An **integer** can be stored on **32 bits** (for example .NET) or on **64 bits**. (a 64 bit system would interpret the transferred name also as the part of the amount)

II. 232 3 0 0 is:

in case of **big endian format**:

$232*2^{24} + 3*2^{16} =$

3892510720

in case of **little endian format** (PCs):

$232 + 3*2^{8} =$

1000

The „bank transfer” example III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Assumption II: Known server location

The sample code applied in a **hard coded machine address + port**. The code should be **independent** of the address of the remote application (it may change, e.g. because of machine failure or load-balancing)

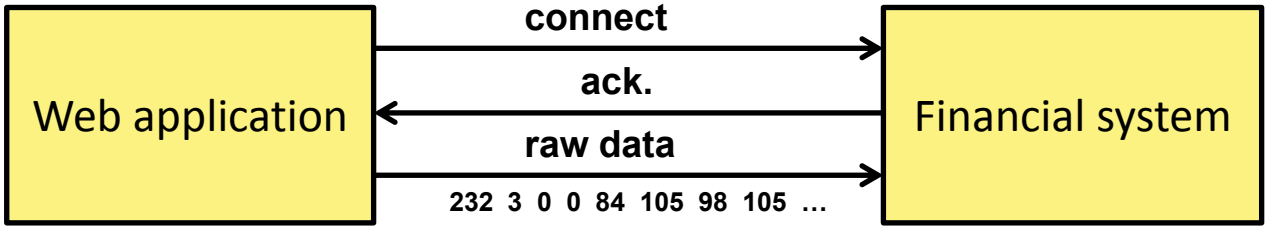
Assumption III: Time-related assumptions

TCP is a **connection-oriented** protocol. Before data transfer an established connection is needed. Both endpoints and also the network have to be **available** during the whole communication.

The „bank transfer” example IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Assumption IV: procedure-signature (number and types of parameters)



Data transfer:

- first 4 bytes represent amount
- character sequence for the name

For inserting **additional parameters** (e.g. the currency) both endpoints' code have to be modified.

These assumptions cause the solution to be tightly coupled!

In the following session we will eliminate these constraints/dependencies one-by-one.

The „bank transfer” example V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

I: Internal data representation

We need a really platform-independent, **self-describing, standard *data format*** (like XML).

II: Location (address) of the component

There should be an intermediate, **addressable „*channel*”** where we send to and recieve from the data. It means a „**logical address**”.

III: Time-related dependencies

The channel has to be implemented **not using a connection-oriented protocol** (to avoid the need for **availability** of the components at the same time). It has to have the capability of **quing up the sent data**. It requires **data-wrapping into self-contained *messages*** (so the channel knows how much data to buffer/deliver).

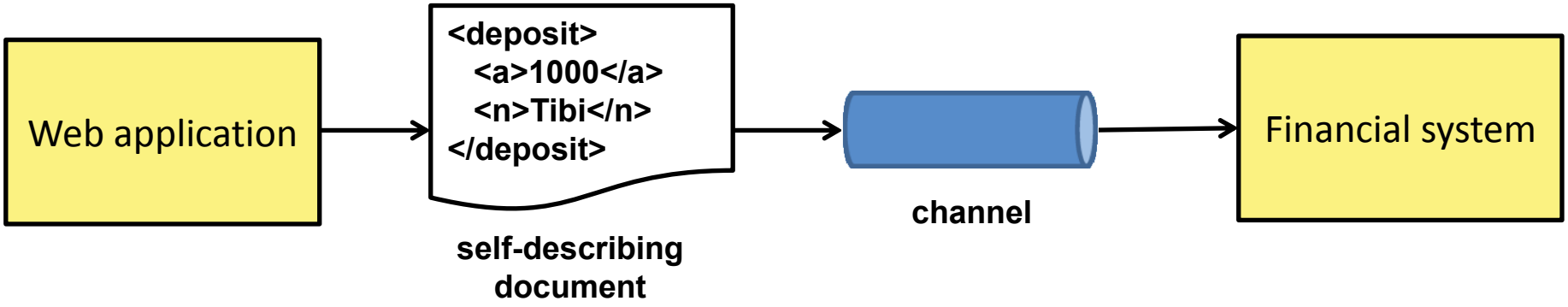
The „bank transfer” example VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

IV: Procedure-signature

We need to allow **data format *transformation* inside the channel**. It solves if one of the endpoints changed. Moreover, this solution also supports cases where **multiple sources** send data to the same channel.

The result: a message-oriented middleware solution



It is a more **tolerant to change, flexible and scalable**, ***loosely coupled*** solution.
Its drawbacks are: more **complex** to design, to build and to debug.

How to build a message-oriented middleware? I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Middleware: things that sit between applications ensuring that the data is transported (usually through a network) and is understood by all endpoints.

The **communication channel** (what transports data between endpoints) can be, for example:

- a series of TCP/IP connections
- shared database
- shared file
- a DVD, a CD, a pendrive or a floppy disk, …

The snippet of data is placed inside this channel (small or large) that has an agreed-upon meaning for all the integrated applications. This piece of data is the **message**.

At this point, messages can be sent across channels.

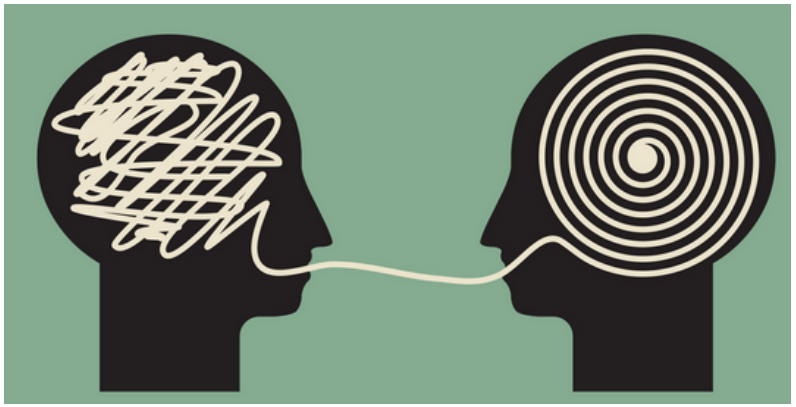
How to build a message-oriented middleware? II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Middleware needs to solve **internal data format conversion** between the integrated applications (e.g. one application stores forname and surname separated while others do not; one application lets to store more than one address for the same customer while others do not…). We usually have only **limited control** over the internal data format of the applications.

Translation solves this problem.

At this point, data can be sent from one system to another and we can accommodate differences in data format.



How to build a message-oriented middleware? III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

In several cases there are **more than one system** to integrate. It involves that the sender has to specify the target system(s).
Let's imagine, that a user changes his/her address in one system. Then this system **should inform all the other systems** which have a local copy of the address. (All components should know about the others – even about the newcomers.)

Middleware could take care of this problem and **send the messages to all the correct places.** ***Routing*** **component** has this responsibility (like a message broker).

Now we can send data from one system to the appropriate target system(s) accommodating differences in data format.

How to build a message-oriented middleware? IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Dealing with multiple applications /often on different platforms in different locations/, data format, channels, transformations and routing can quickly become **complex**. A ***system management*** function could inform us about what is going on inside the system. Its main tasks are:

- monitoring data flow
- checking and ensuring availability of the system components
- error reporting

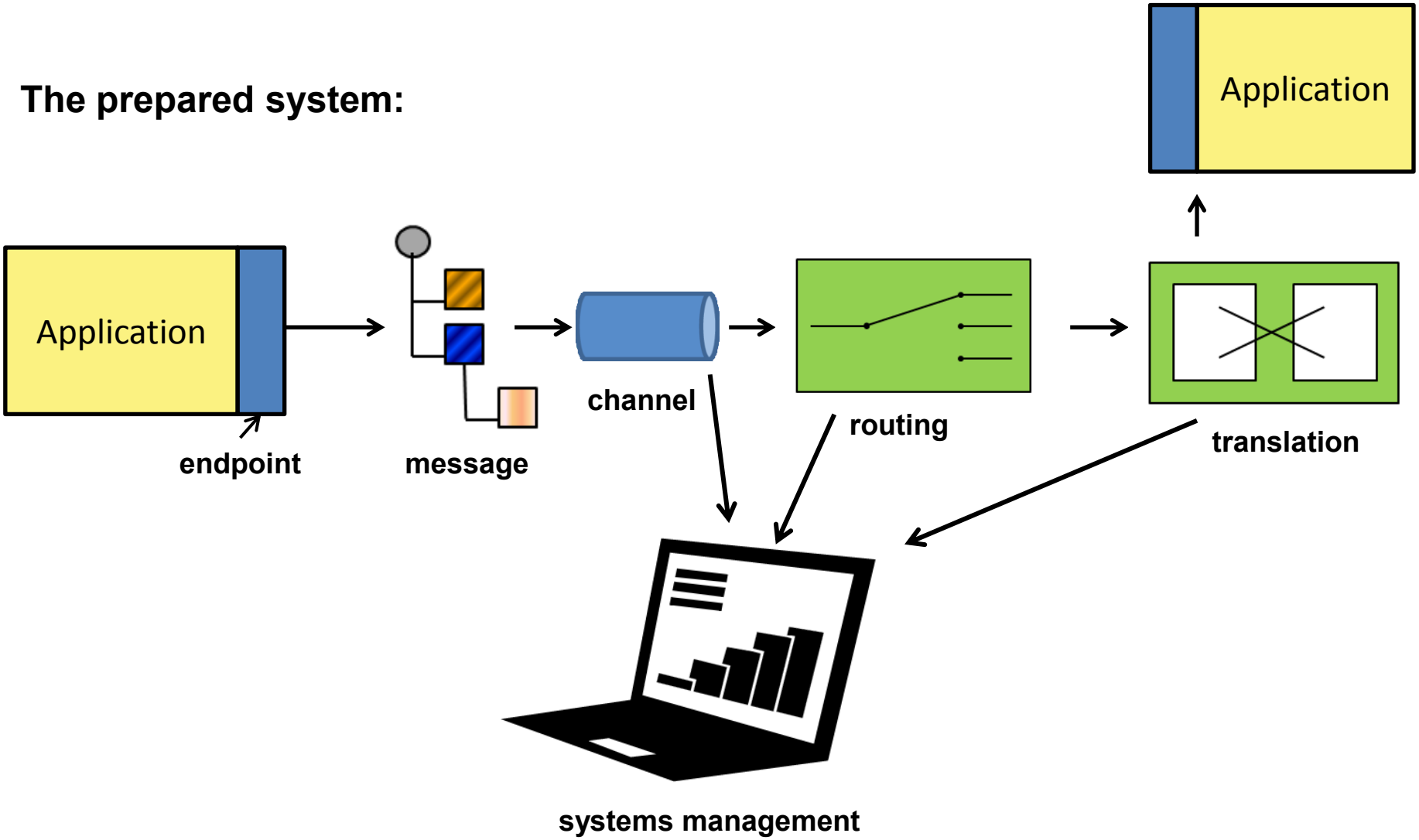
Now data can be sent between the appropriate components accommodating differences in data format, and the performance can be monitored.

Another problem a middleware has to deal with is, that some applications are **not prepared** to participate in an integration solution, is not able **to send data to the channel directly**. Into these application a ***message endpoint*** (a piece of code or a ***Channel Adapter***) has to be implemented to connect the application to the integration solution.

How to build a message-oriented middleware? V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

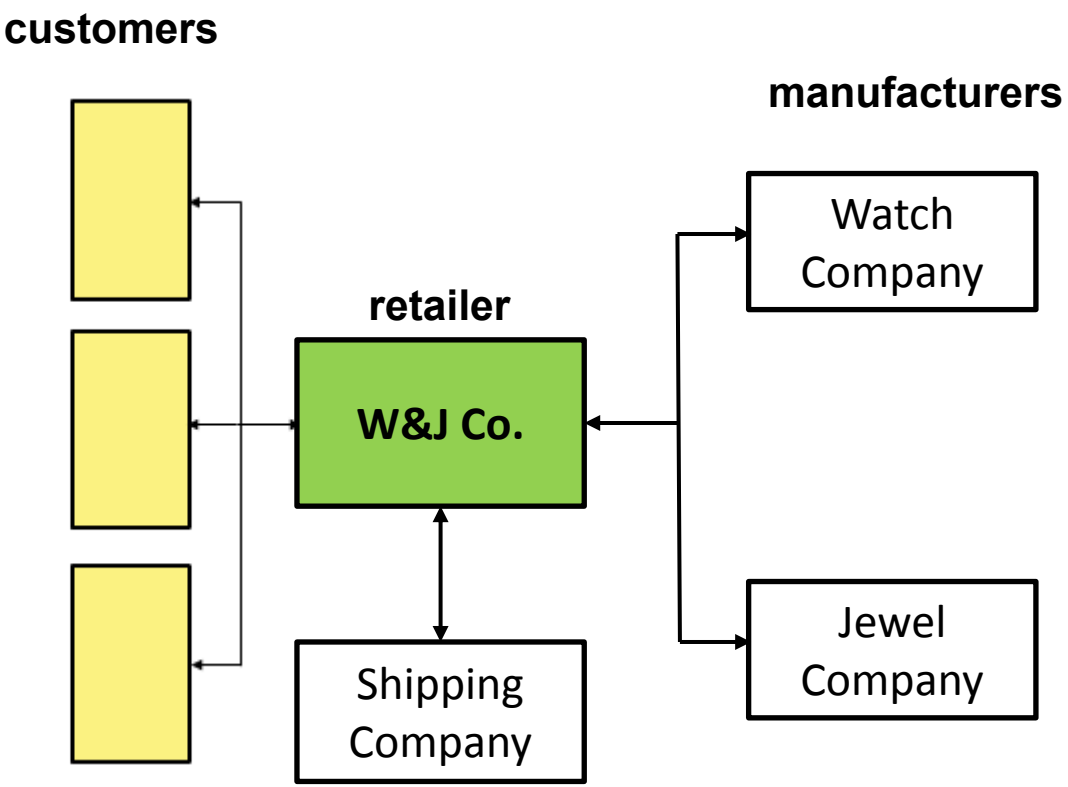
The prepared system:



A complex example for integration

The company to integrate (W&J Co.)

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



Required functions:

- Take orders (via web, phone or fax)
- Process orders (verify inventory, customer status, shipping, invoice)
- Check status
- Change address (via Web)
- New catalog (periodically)
- Announcements (selective)
- Testing and monitoring

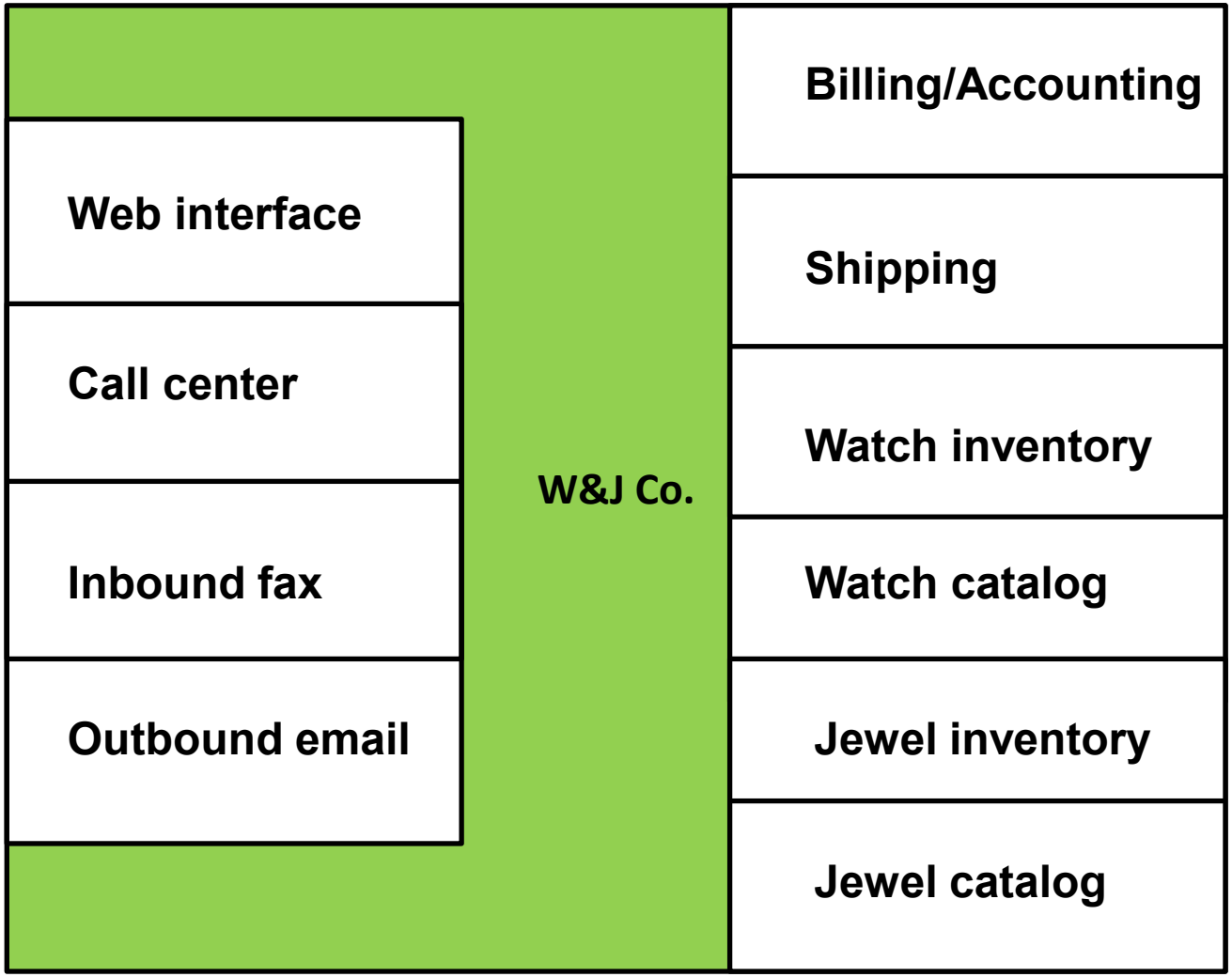
Details of the company

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- **4 channels** for interacting with customers

- a shipping system is for computing **shipping charges** and for interacting with **shipping** companies

- **2 inventory and catalogue systems** (because of historic reasons)
/so, the **existing systems have to be integrated**!/



1. function: Taking orders I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

We suppose that call center system is a packaged application, Web site is a custom J2EE application, and the fax system requires manual data entry into a Microsoft Access application.

Because of the 3 different inbound channels, first, we should unify the data format (this way the orders can be handled later independently of their origin)**.**

Placing an order is an **asynchronous process that connects many systems**, so we decided to implement a **message-oriented middleware** solution.

1. function: Taking orders II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

I. Via **call center**: it is a packaged application ⟶ channel adapter needs

A *Channel Adapter* can be attached to an application and publish messages to the channel when an event occurs in the application.
/e.g. a database adapter may add triggers to specific tables so that every time the application inserts a row of data a message is sent to the channel/
It works also in the opposite direction: consumes messages off the channel and triggers an action inside the application.

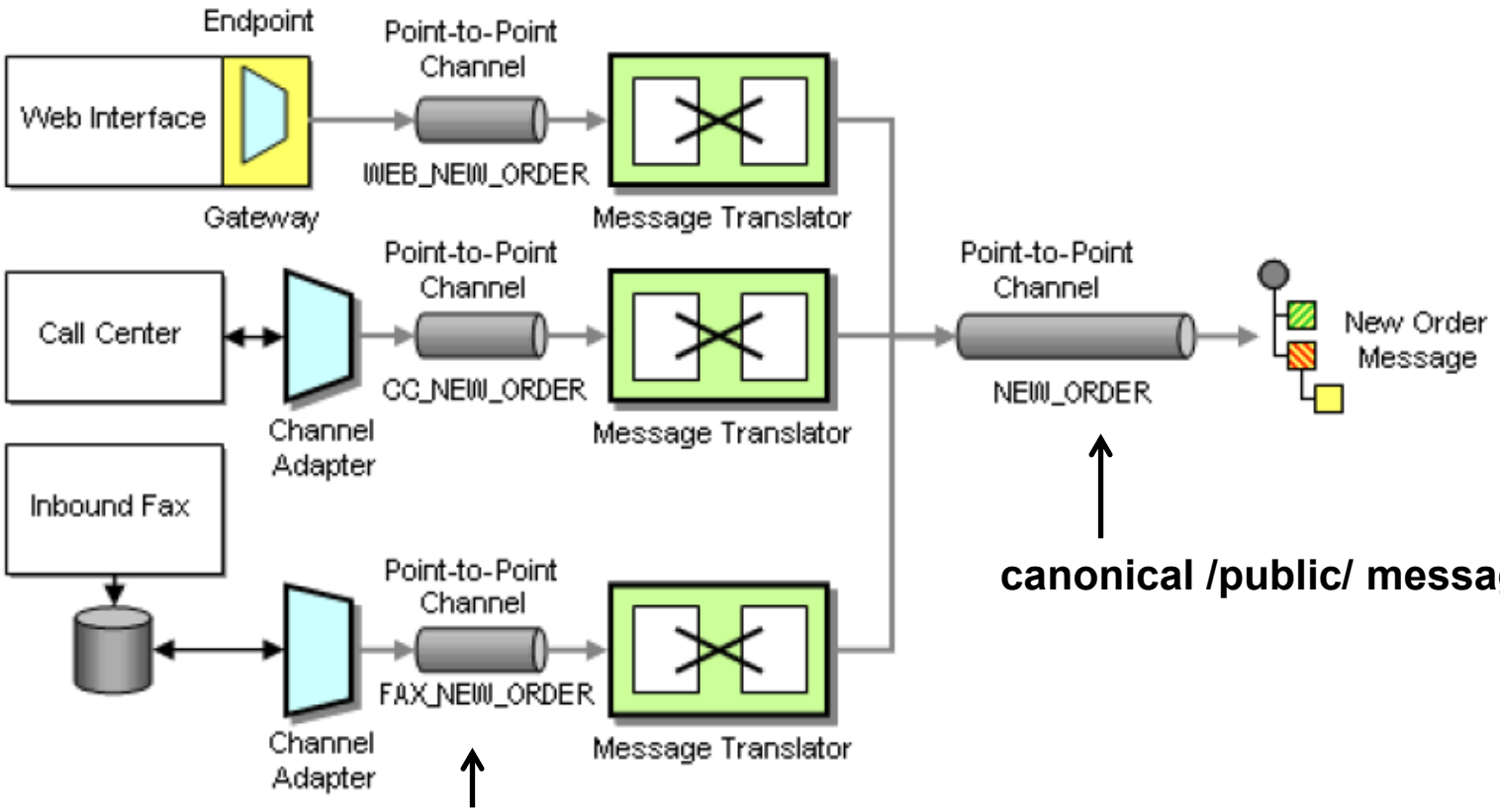
II. Via **inbound fax**: the same way: the application database is connected by a channel adapter

III. Via **Web application**: it is custom-built ⟶ the ***endpoint*** code can be implemented into the application. A ***Messaging Gateway*** is applied to separate the application code from the messaging system's code.

1. function: Taking orders III.

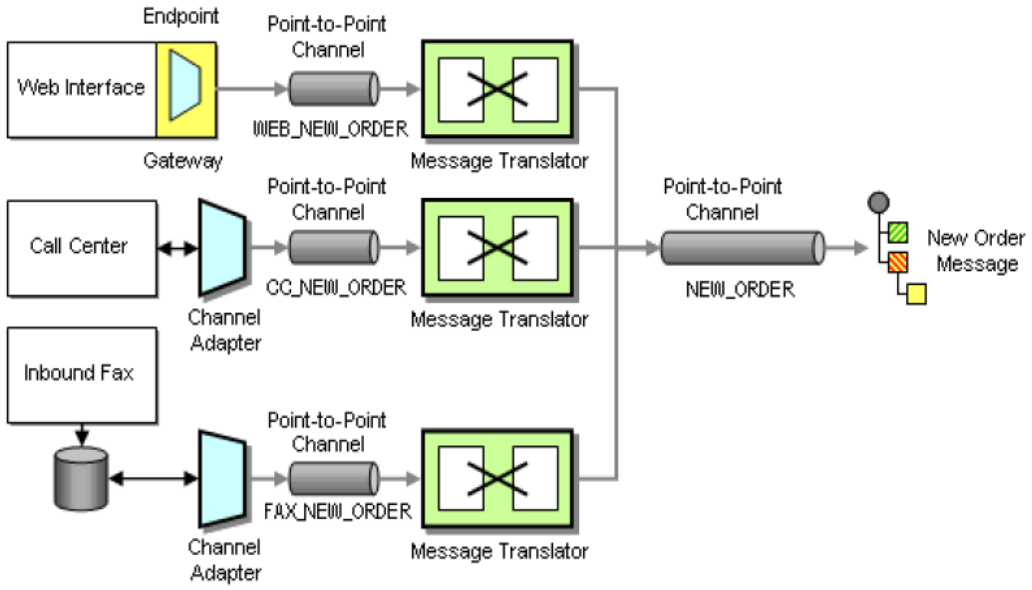
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Handle of the three different channels to unify orders



1. function: Taking orders IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



If any of the input applications **changes**, only the appropriate **message translator** has to be changed.

Point-to-point channels ensures that each order message is **consumed only once**.

Channel names reflect the data type they transmit. The NEW_ORDER channel is a so-called ***Datatype Channel,*** because it carries messages of only one type.

The NEW_ORDER message is a **Document Message**, because it carries document instead of any instruction to the receiver(s).

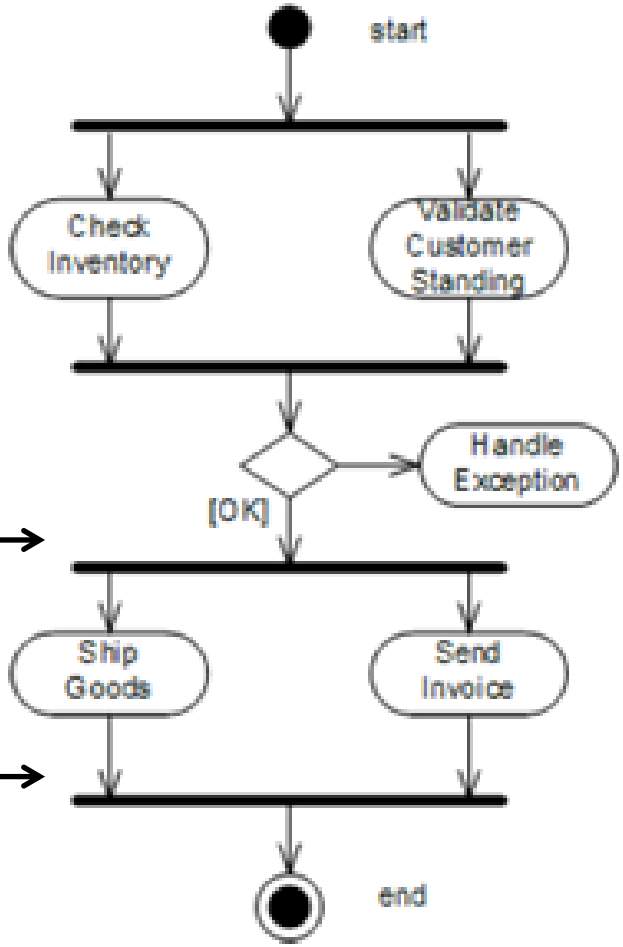
2. function: Processing orders I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The main steps are:

- verifying the customer's credit standing
- verifying the inventory
- if both verifications are ok, then billing and shipping

Its UML activity diagram



Fork (actions start simultaneously)

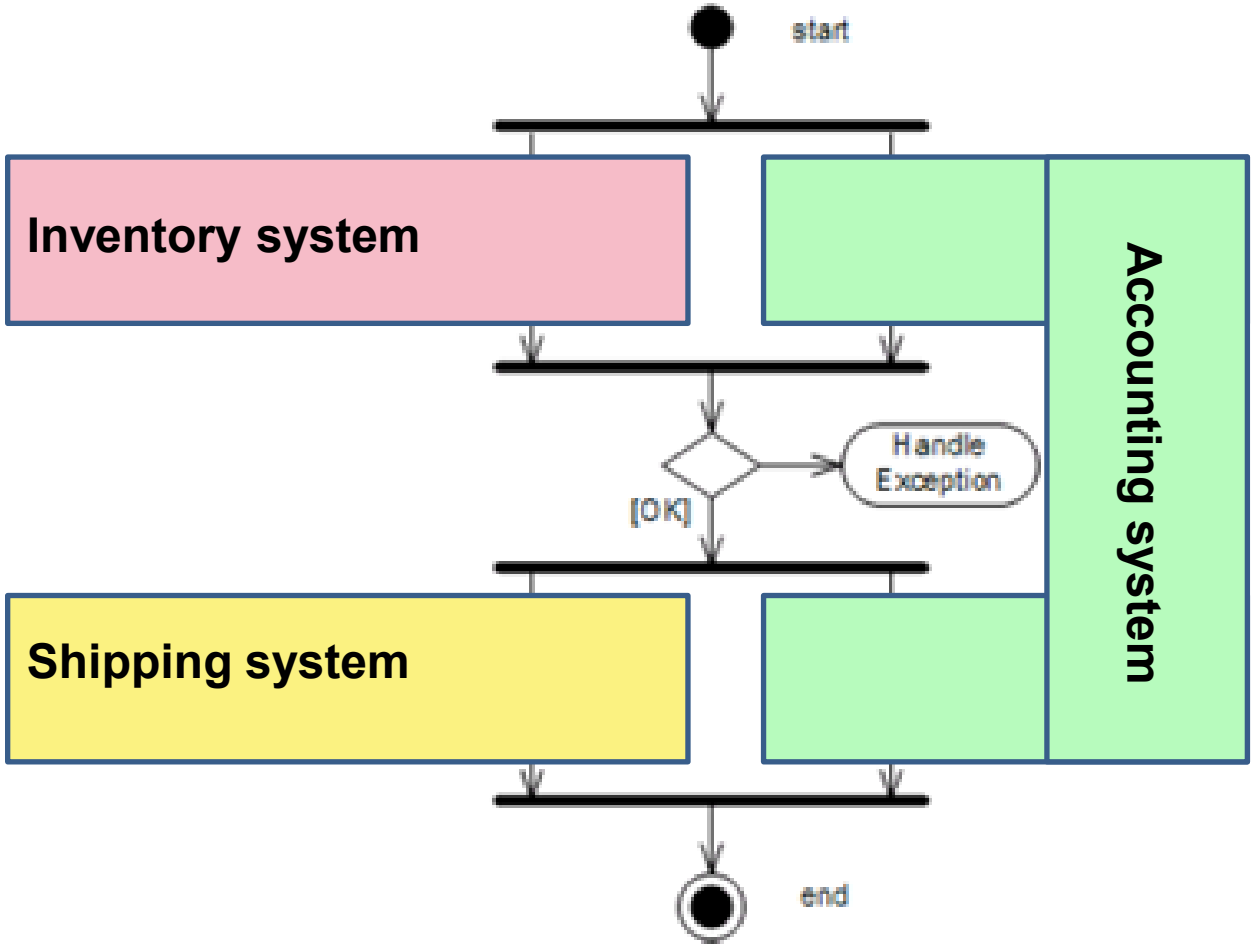
join

/exceptional handling is not detailed/

2. function: Processing orders II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Actions of processing orders vs. the (IT) departments of the company

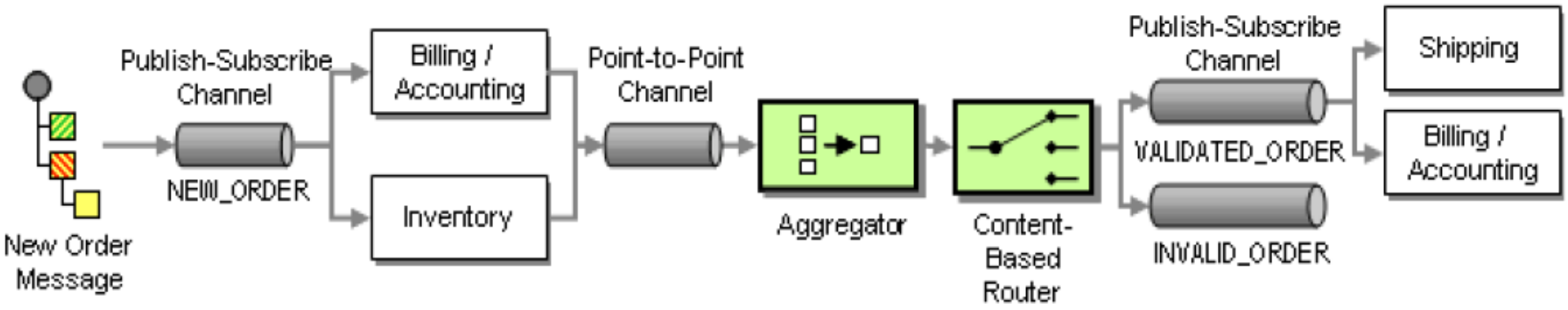


Distributed Business Process

2. function: Processing orders III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Integration design of processing orders



For „**fork**” we apply a ***Publish-Subscribe Channel***. It sends a message **to all active consumers**.

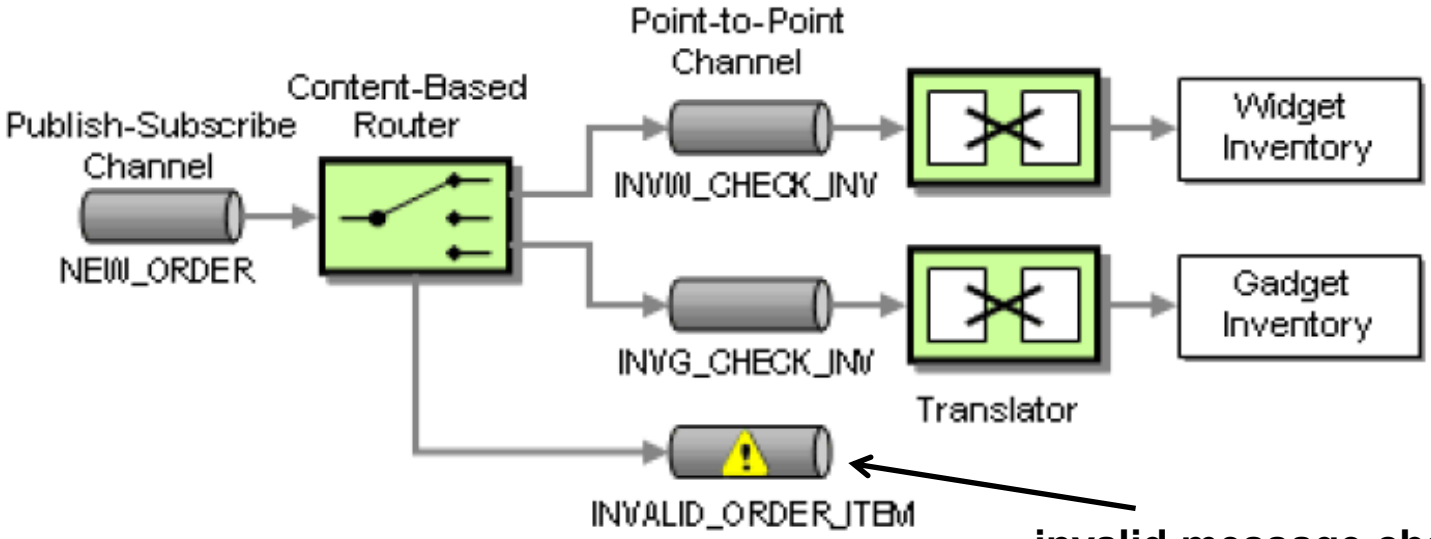
For „**join**” we apply an ***Aggregator***. It **receives multiple incoming messages and combines them into a single outgoing message**.

For a **branch** we apply ***Content-Based Router***. It **consumes a message and publishes it unmodified to a choice of other channels based on rules coded inside the router**.

2. function: Processing orders IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Inventory check has to be highlighted in more details, because of the two different inventories



The **meaning of a message** changes depending on which **channel** it occurs

Order messages are differentiated by their **item number starting letter**.

Both inventory systems use **different internal data formats**, that's why **message translators** are needed.

2. function: Processing orders V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

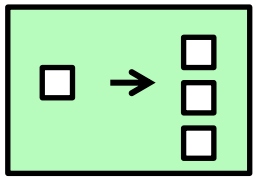
How to handle multiple items inside one order?

Question: **which inventory** system will check the inventory for the order?

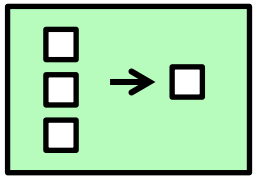
We could apply a **Publish-Subscribe Channel**. However, in that case how to handle invalid items?

For ensuring the central control of the content-based router and **the individual handling of orders** – like before – we involve a ***Splitter*** and an ***Aggregator*** into the solution.

Splitter breaks a single message into multiple individual messages.



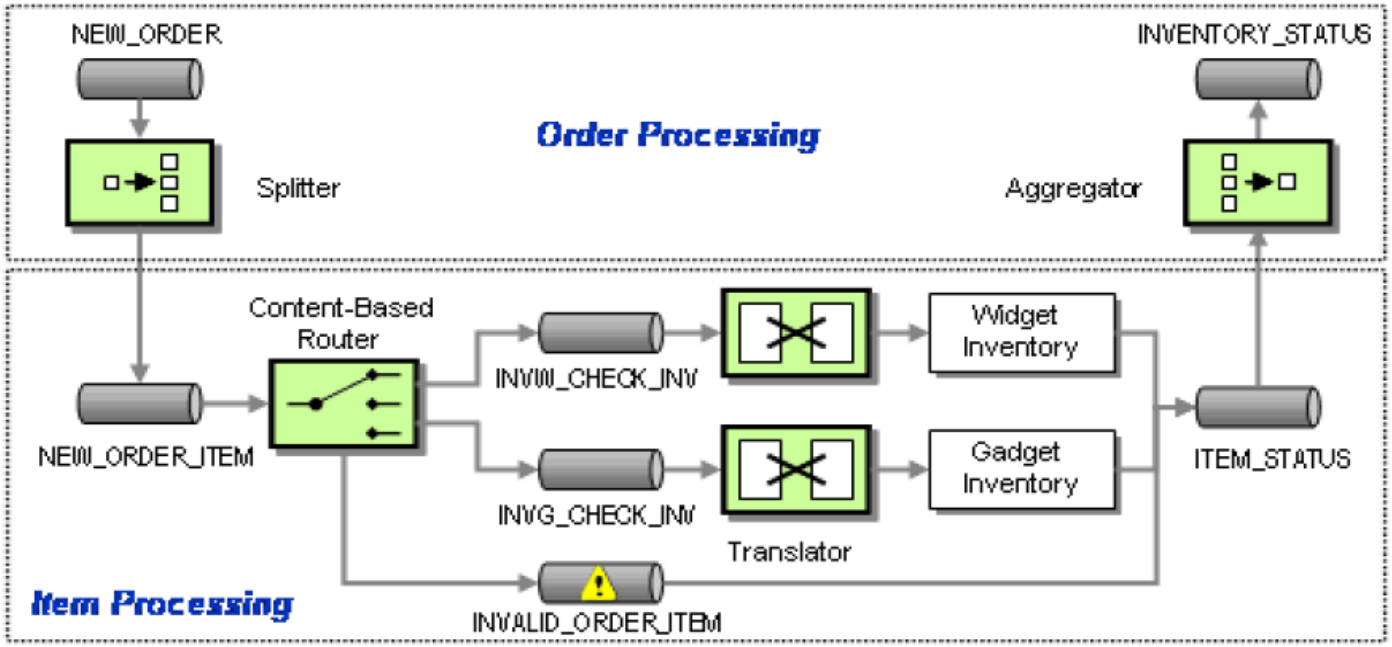
Aggregator combines multiple messages into a single message.



2. function: Processing orders VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Handle of multiple items in one order



For aggregation we have to know:

(1) Which messages belong together? (**correlation**)

(2) How do we know that all the messages have arrived? (**completeness condition**)

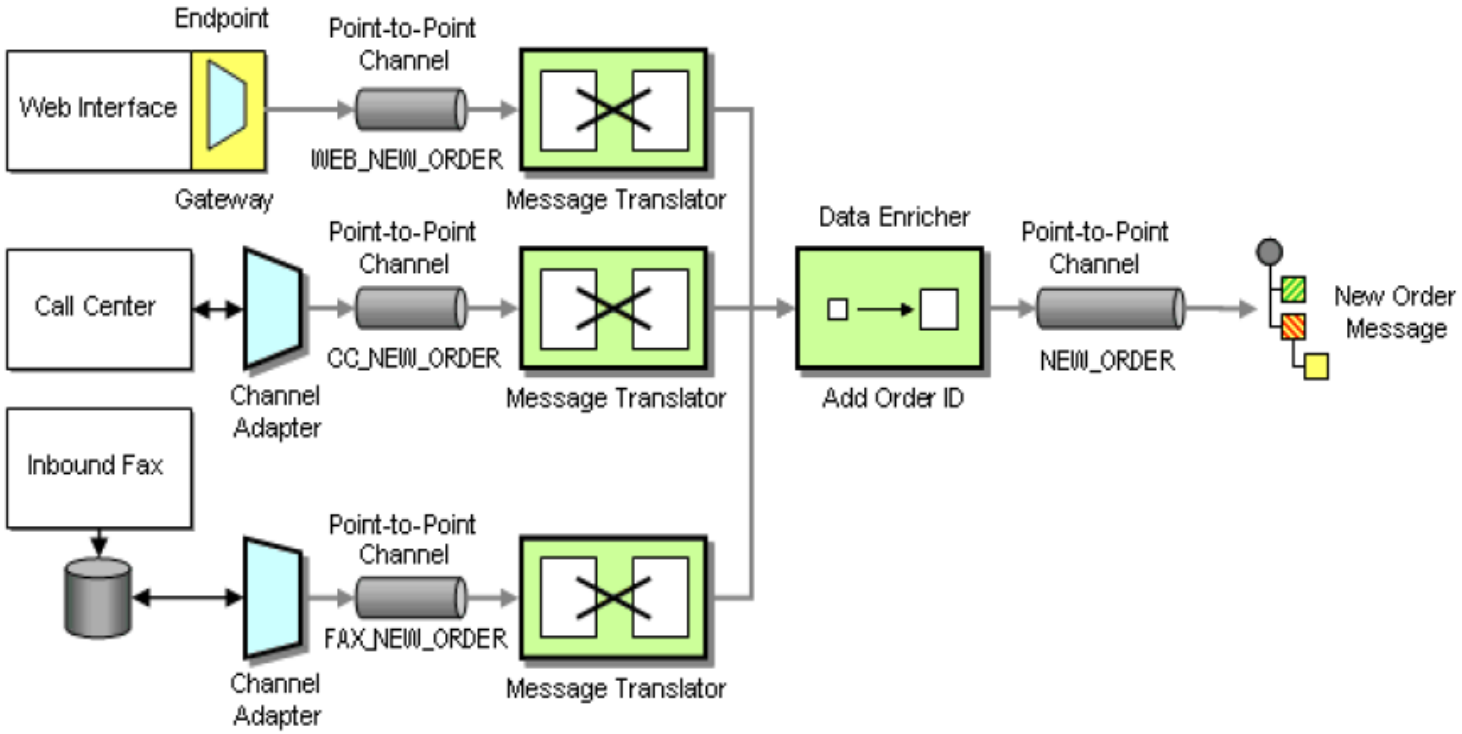
(3) How to combine the messages into the one resulted message? (**aggregation algorithm**)

2. function: Processing orders VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

(1) Correlation.

Since **one customer can have more than one order**, customer ID is not appropriate for determining which messages belong together. **Order ID** has to be introduced. It is inserted into the message by ***Content (Data) Enricher***, which can **add missing data items to an incoming message**.



2. function: Processing orders VIII.

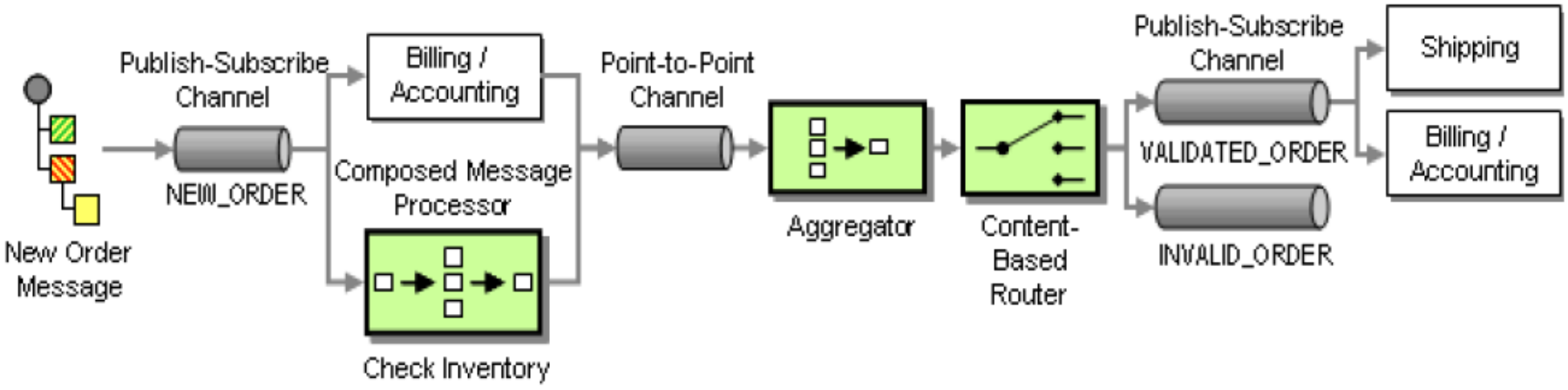
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

(2) Completeness

Aggregator can determine that all the messages have arrived if **all the messages are routed to the aggregator** (including invalid order messages) and a **„number of items" field** exists in the order message.

(3) Aggregation algorithm

Concatenation.



Splitter + Router + Aggregator = *Composed Message Processor*

3. function: Checking status I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

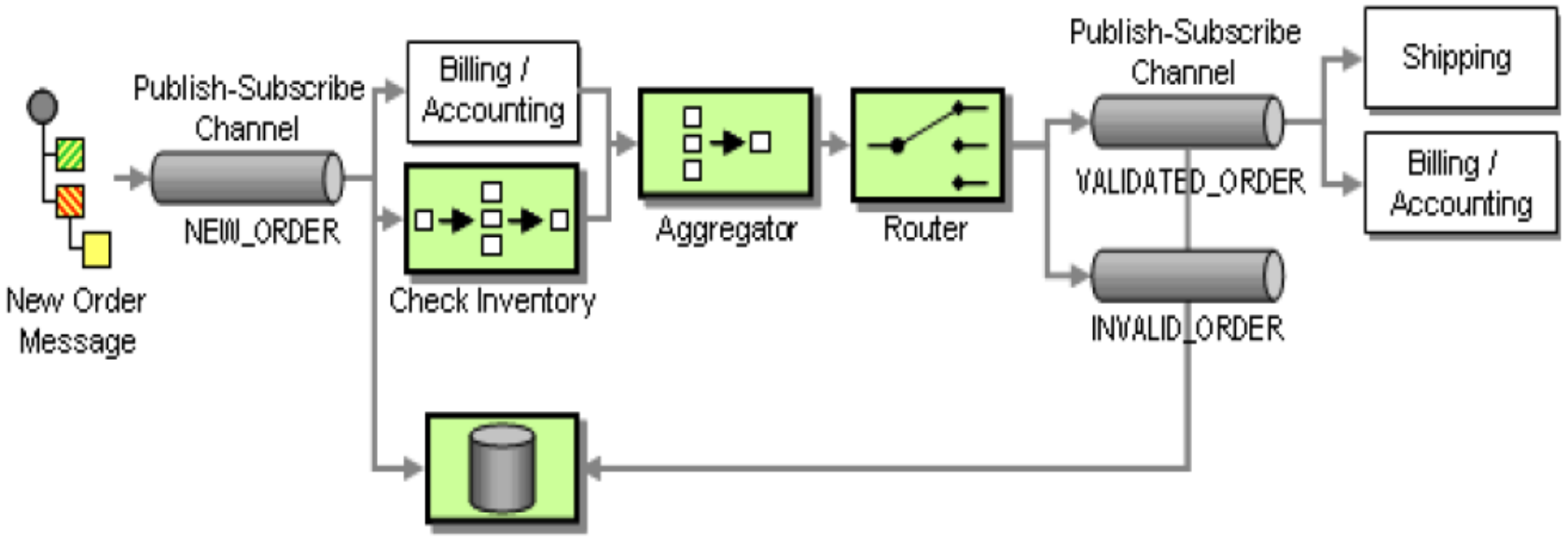
Between taking an order and shipping **long time can pass**: in some cases the inventory is out of the requested item, so it has to wait for the item to arrive and shipping can take place only after that.
Fortunately, **asynchronous messaging** ensures that each component can take part in the communication **at its pace**.

Of course, customers may be **curious of the state** of their order (even they want to ask only for the existing items immediately).
Since in integrated systems **messages travel through several components**, tracking is a hard task.
The last message related to the order is needed.

3. function: Checking status II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

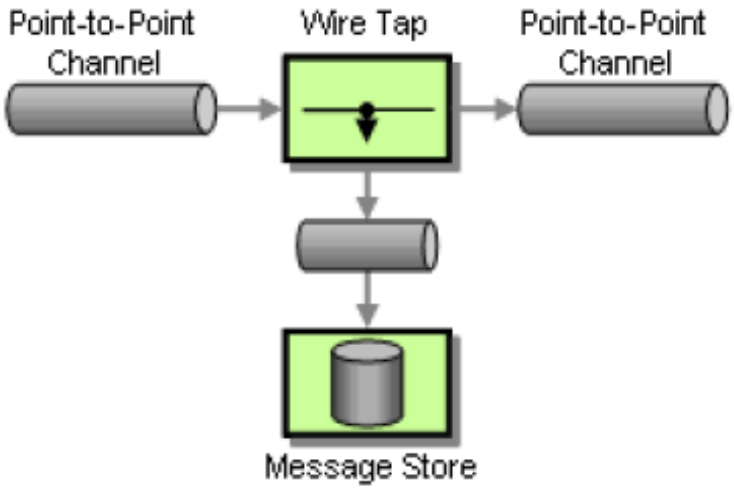
For tracking the message flow and order status, we can add a ***Message Store*** to a **Publish-Subscribe Channel** which transmits the **new and validated orders**. Then, **Message Store can be queried for the status**.



3. function: Checking status III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

If we use **Point-to-Point Channel**, the solution is not so simple and we have to use a ***Wire Tap***.
A Wire Tap **consumes a message off one channel and publishes it to two channels.**



Then, the second targeted channel can be used to send messages to the Message Store.

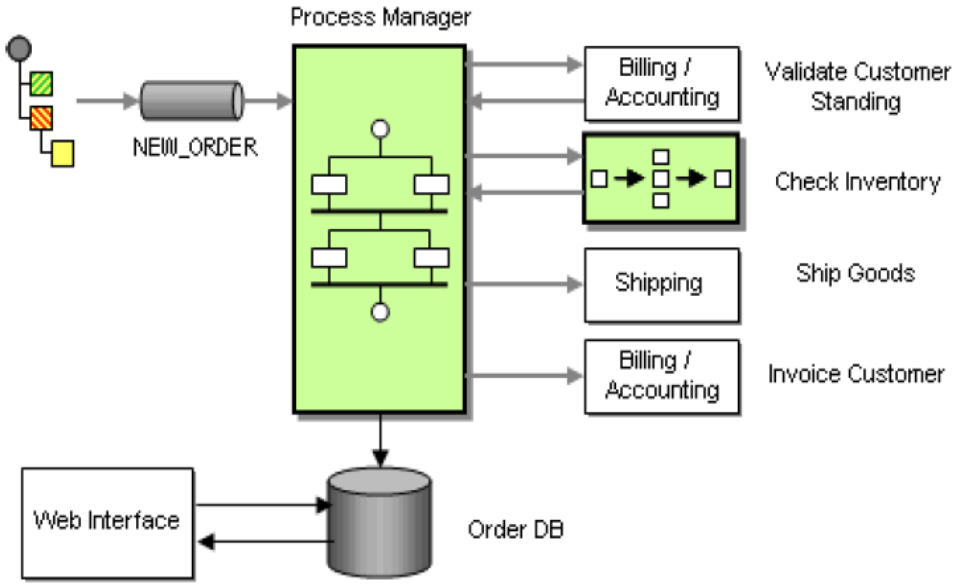
3. function: Checking status IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Another **advantage of** storing the order data in a **database:**
It is **not** necessary for **each message to carry all** the relevant data anymore (e.g. a **customer ID** is enough).
Using the database, all the subsequent **messages can refer** to the data stored in the message store (it is called ***Claim Check***).

Based on the status **information stored** in the message store we are able to **determine the next step** in the process (instead of connecting the components with fixed channels).

So, a **Message Store can be turned t** be a ***Process Manager***.
Process Manager has a central role: **manages the flow of messages through the system.**



3. function: Checking status V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The main functions of a **Process Manager**:
- **stores data**
- **keeps track** of the process and **determines the next step** in the process.

Applying a Process Manager **reuse increases**, **rapid changes and maintenance are allowed** and the whole architecture is turned into **Shared Services**.

Process Manager uses a **persistent store** (usually database or files).

The **Web interface**, which the customers use for **checking status** (it is a **synchronous process**) could be **connected** with **either the process manager or directly to the store**. In our solution Web interface is a custom application, so we connect that to the database directly (this **Shared Database** is the **easiest** way providing **up-to-date** data, however, realizes **tightly coupling**).

3. function: Checking status VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

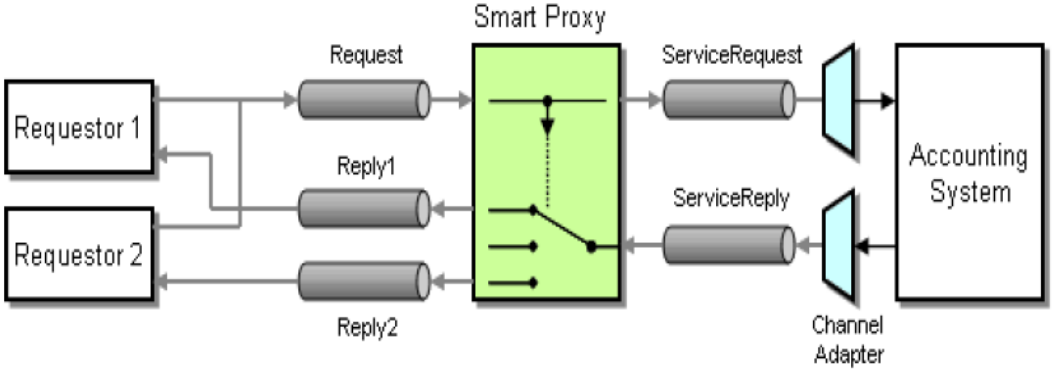
Now, all the „services” can be invoked from any other component.

For turning it into a **Service-Oriented Architecture** we need**:**

- a **lookup from a service registry (service discovery)**
- **interface contract** for each service (what describes their functions)
- **return address support** for each request-reply service (caller should be able to define a special reply channel)

For ensuring the Return Address in case of systems which were not developed caring return address: we use ***Smart Proxy.***

Smart Proxy **can store information from the request and uses it to process** (e.g. route) **the reply message.** It is also useful in **monitoring the QoS** (e.g. response time) of the external service.



4. function: Change address I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Customers are allowed to change their addresses via Web interface. One customer may have more than one address (e.g. billing and shipping address).

The two basic approaches:

a. including address data **into each NEW_ORDER message**

b. replicating address data to other systems

4. function: Change address II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

a. (address data in each NEW_ORDER message):

Advantage: **an existing channel can be used** to transport the additional data

Disadvantage: **big amount of transmitted data** (address change is not so frequent)

Since billing and shipping systems are **packaged applications**, they may **not** be **able to handle** order messages completed with extra address information; they have their **own local database**. For updating them, we have to:

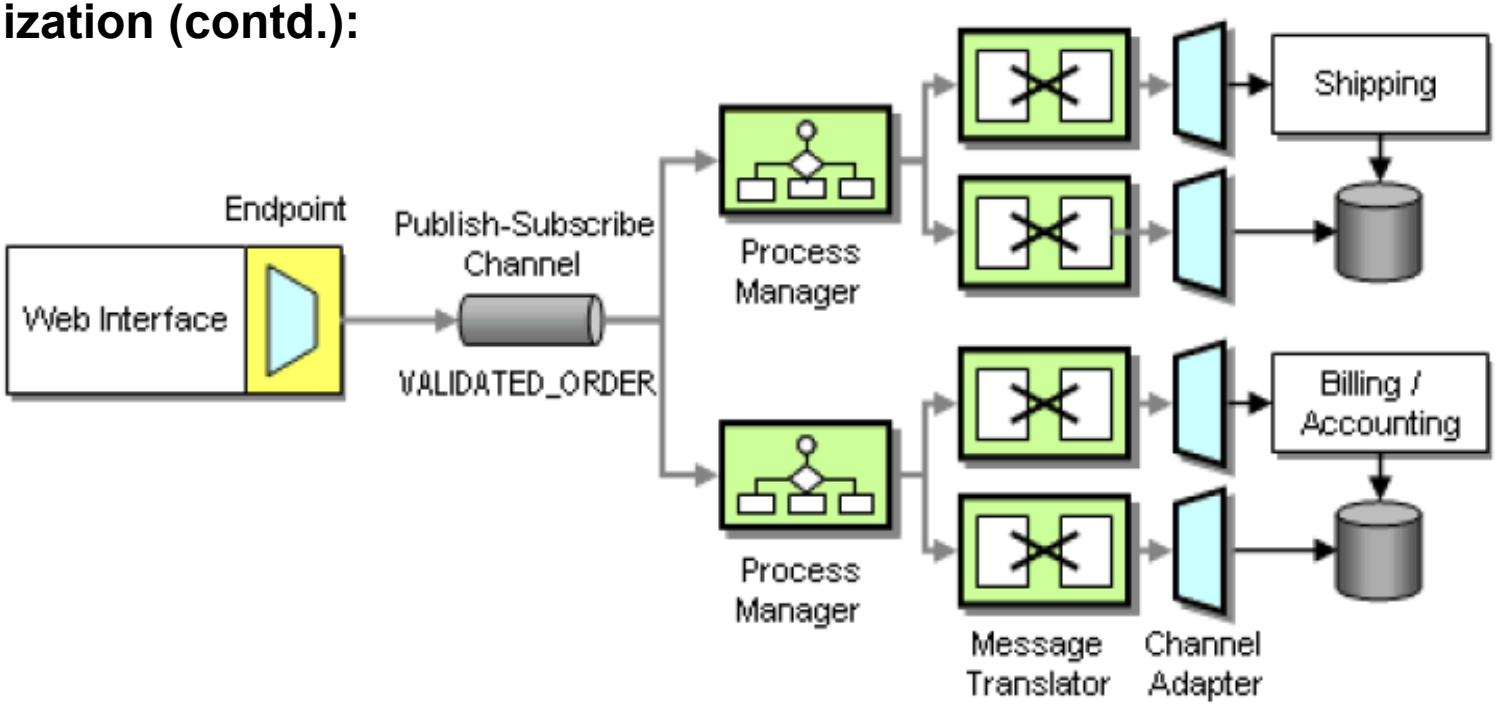
- update the local address
- **THEN** shipping the item/invoice to the updated address

Their order matters ⟶ **Process Manager is needed!**

4. function: Change address III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

a. Realization (contd.):



Because applications use their **special data format** and the order message is in canonical data format, **Message Translators** have to be applied (now it is not built into the Process Manager). Then **Channel Adapters** are for **inserting the data** into the database and for **invoking a function** of the applications' API.

4. function: Change address IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

b. Data replication:

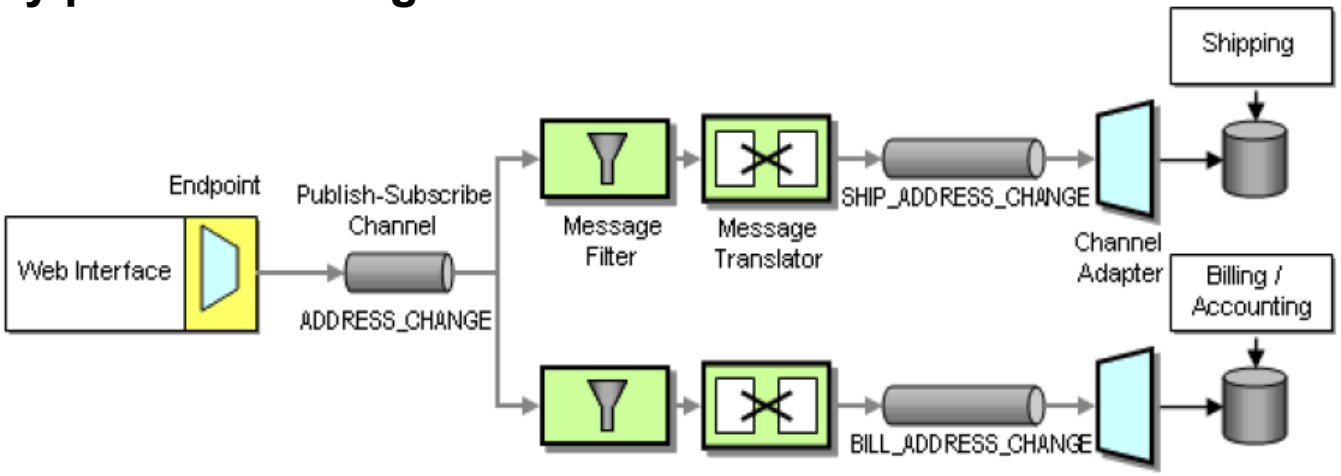
An **address change** in the Web interface **is propagated to the local databases** of all interested systems **using a specific Publish-Subscribe Channel**. Any system which uses address subsribes to this channel.

Advantage: it **reduces data traffic and coupling** between the systems

Disadvantage: **another interface** is needed for the systems

Since an order message may contain more than one address, we have to guarantee that each system gets only **the right type of address**. That's why we apply ***Message Filter***. It **only passes messages that fit to a certain criterion**.

We define the **data format of the Web interface to be the canonical data format**. So, Message Translator is not needed, but it limits the flexibility.



4. function: Change address V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

How to decide between solutions a. and b.?

Message traffic + internal structure of the applications are important: Maybe we can not insert data directly into the local databases of the applications, only e.g. after validation, conformation mail, etc. It is disturbing if each order message contains the address.

Granularity of application interface (designed for integration vs. standalone application):
- **fine-grained interfaces** (e.g. custom functions for every field of the address): huge number of remote calls, lot of **overhead**, nead for **synchronization** of the messages, **tight coupling**, but **efficient for single application**.
- **coarse grained interfaces**: **in integration it is more efficient** and **less tightly coupled**. However, it **limits the flexibility** (we got also information pieces that we do not need).

Happy medium is the best
(taking into account the necessary **trade-offs**).

5. function: New catalog

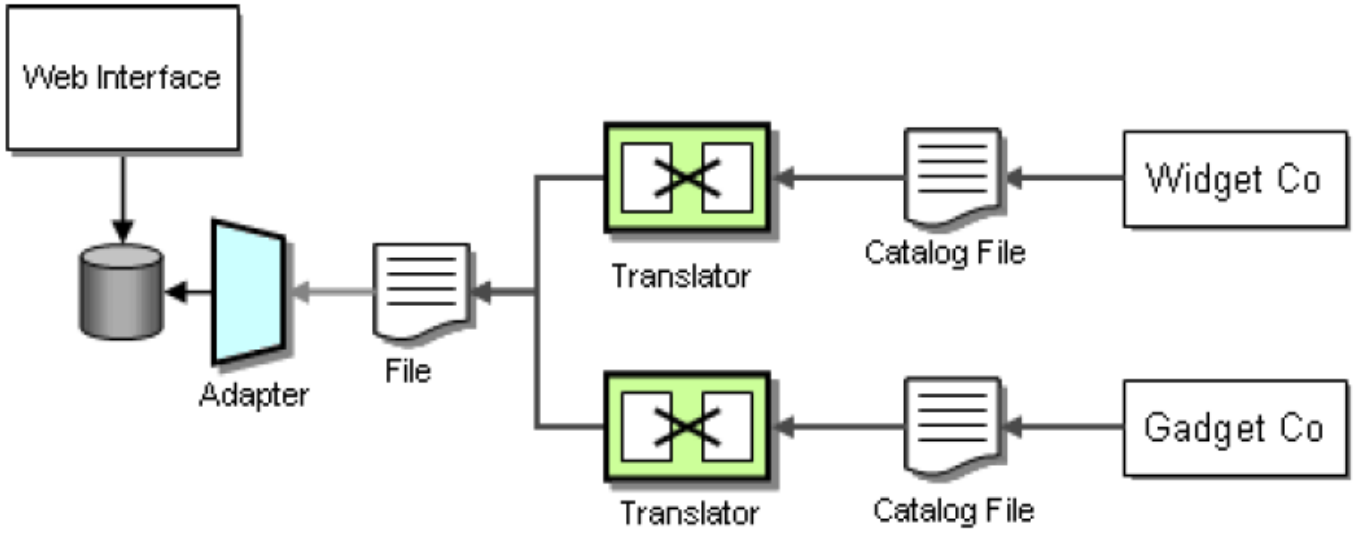
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Customers intends to know the **currently offered items and their prices** via **Web interface**. For that we should create an **information portal** (it has multiple data sources).

We suppose that both suppliers update their catalog **in every 3 months**. That's why a real-time messaging system – which works poorly on the public Internet - is unnecessary. **File Transfer integration** suits better: easy and efficient.

Translators produce data in our **internal catalog format**.

Translators handle **one catalog at the same time** (it is more efficient than item-by-item).



6. function: Announcements I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

We would target **special customer groups** with special messages, moreover allow the customers to subscribe for messages (announce specials) of their **special interest.**

The problems with Publish-Subscribe Channel (what is our tool for multiple receivers):

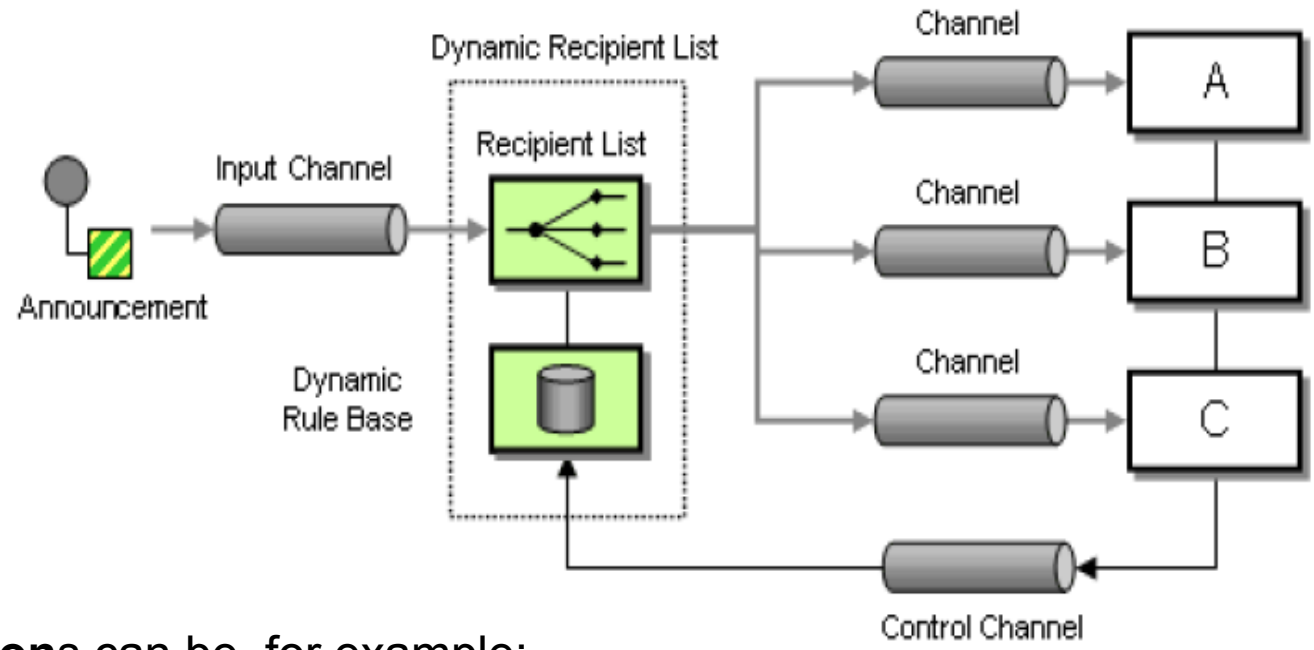
- it allows any subscriber to listen **without the knowledge of the publisher**
- it is appropriate for LANs; in case of **WAN**s we need **separate copies** for each recipient.

Dynamic Recipient List is needed. It is a combination of:

- **Recipient List**: a router that propagates a message to a **set of recipients**, **addressing each recipient specifically**
- **Dynamic Router:** a router whose routing algorithm **can change based on control messages**

6. function: Announcements II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



Its **realizations** can be, for example:

- email: **mailing list** features
- **Web services** interface: each recipient channel is a **SOAP** request; channel address is the URI of the Web service

↳ Patterns are independent of a specific transport technology

7. function: Testing and Monitoring I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

For **monitoring** the correct execution of messages **Message Store** can provide some basics, however we need **more detailed data**.

Example: verify the invoice of an external agency for assessing customers' credit standing.

We request the service only for new customers and late answers are not charged

↓

we can not calculate it based on the number of orders

So, we need the **number of our external requests** and the **response time**

Since the external agency handles **parallel requests** from the company as a **Shared Service**, we have to

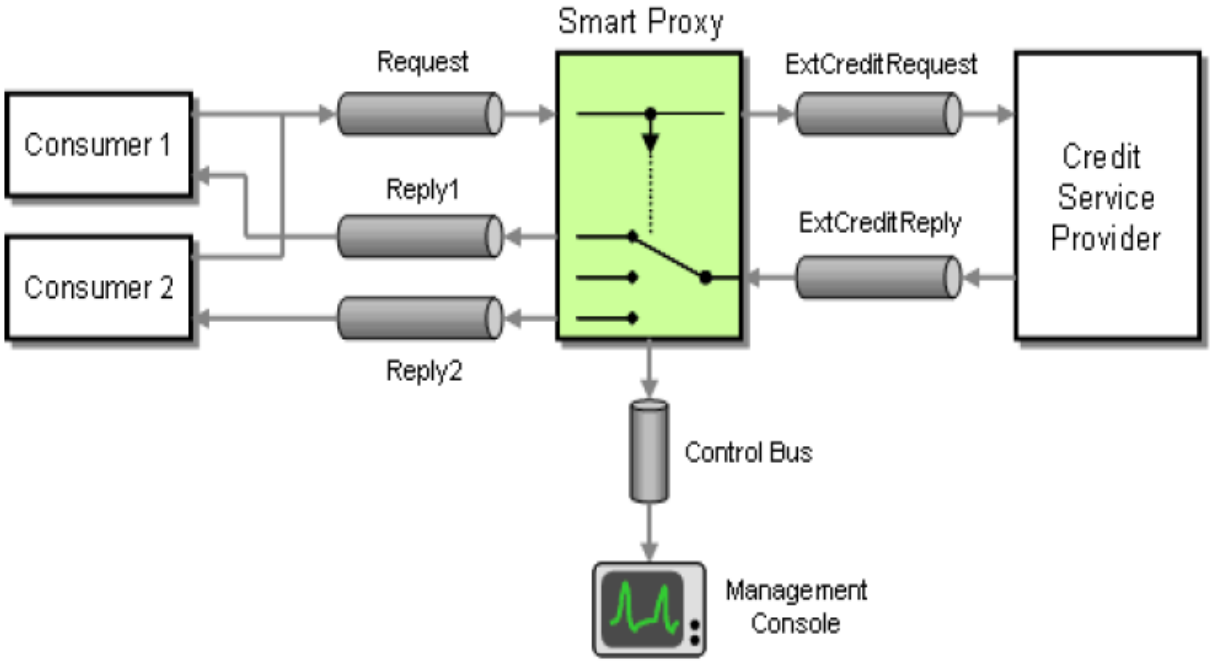
(1) match requests and responses

(2) define **Return Address**.

Smart Proxy can do that!

7. function: Testing and Monitoring II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



- **Smart proxy** is inserted **between any consumer and the external service**

- each **Return Address is replaced by a specific Reply Channel**

-the original **Return Address is stored** in the Smart Proxy for the reply

- Smart Proxy **measures the delay** between the request and the reply and it publishes these data to the ***Management Console*** through the ***Control Bus***

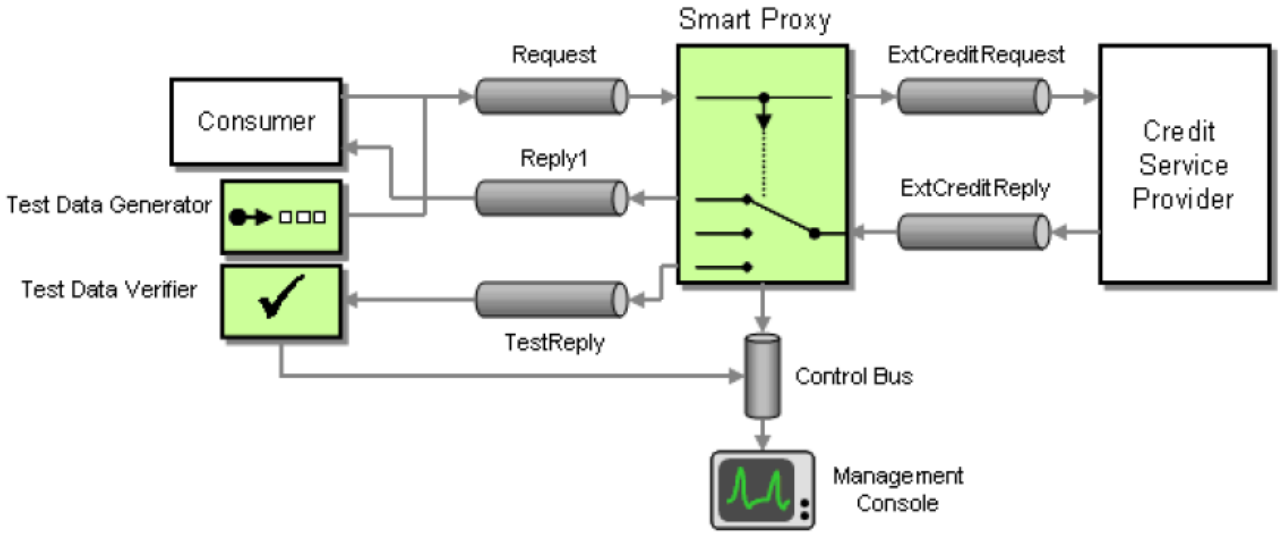
- Smart Proxy can report **timeouts**, too

7. function: Testing and Monitoring III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Ways of **incorrect result detection** from the message:

- Periodical **Test Message** with known results verified by a ***Test Data Verifier***. ***Test Data Generator*** specifies an own Reply Channel.
- **Statistic**al sampling: e.g. if we can suppose that less than 1/10 of the orders fail, in opposite (**suspicious**) cases the management console can email the orders to an administrator for verification



In this example we have described a solution more or less **accurately** by a **vendor- and technology-neutral language.**

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

THANK YOU FOR THE ATTENTION!

Reference:

Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns – Designing, Building and Deploying Messaging Solutions, Addison Wesley, 2003, ISBN 0321200683

www.enterpriseintegrationpatterns.com

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

ENTERPRISE INTEGRATION PATTERNS

4. Integration Styles

Author: Tibor Dulai

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

Difficulties of integration

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- **custom** (in-house) developed ⟷ **3rd party** applications

- **multiple locations, computers and platforms**

- **outside-running applications** (e.g. at business partners, at customers)

- some applications have **not been developed for integration** and cannot be changed

Application Integration Criteria I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Integration needs to be varied (otherwise one integration style would be enough). We have to **take into account**:

- **Application integration**: applications have to **work together** (vs. a standalone application)

- **Application coupling**: **assumptions and dependencies should be reduced** (because of possible changes, evolution of applications). The interface should be **specific enough** to implement useful functionality, **but general enough** to allow changes as needed.

- **Integration simplicity**: developers intend to **minimize changes** of the applications and **minimize the additional code** for integration. However, approaches with the least changes **may not ensure the best integration**.

- **Integration technology**: for the different integration needs usually different **specialized software and hardware** are required. They can be **expensive, vendor-locked and hard-to-learn**.

Application Integration Criteria II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- **Data format**: there is a need for a **common format or** an intermediate **translator**. Data format may change over time (**data format evolution + extensibility**)

- **Data timelines**: we intend to **minimize delays** (data may **expire**). Prefer **frequent small data exchange** to waiting for large amount of unrelated messages.

- **Data or functionality**: if we use shared functions instead of shared data, we have to be careful: **assumptions of remote method invocation differs to local method call!**

- **Asynchronicity**: usually preferred, because remote endpoint or the network may be **unavailable**, or the sender intends to **continue its operation** instead of waiting for the result.

Which integration approaches to apply to which of these criteria?

Application Integration Options

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Integration approaches can be classified into one of 4 **integration styles**:

- **File transfer**: each application produces **files of shared data** for others to consume

- **Shared database**: applications store the data they wish to share in a **common database**

- **Remote Procedure Invocation**: applications expose some of its procedures so that they can be **invoked remotely**

- **Messaging**: applications connect to a **common messaging system**, and exchange data and invoke behavior using messages

The pattern order reflects an **increasing order of sophistication**. These styles can be **combined**.

These **patterns** have **similar context** and the **same problem statement**:
the need to integrate applications

File transfer pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Problem:</u> How can I integrate multiple applications so that they can work together and they can exchange information?

An enterprise's **software system's parts** may **diverge**, because …:

- some of them were developed by a **3rd party vendor**
- they were implemented at **different times**, it involves possible **different technologies**
- they were created by **different developers** who differ in their **experiences** and **preferences**
- to **start operating the application is more important** than making it ready for integration

File transfer pattern II.

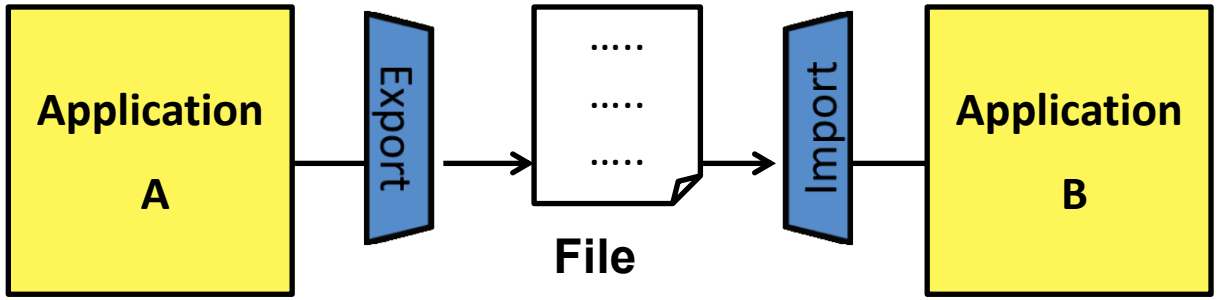
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The goal is to **integrate** applications **both on the business level and on the technical level**, **minimizing the knowledge** what the integrators have to know **about the applications**.

A **common data transfer mechanism** is needed, independent from platforms and languages.

Files prove to be the **simplest** approach what ensure an **universal storage** form.

Solution: **each application produces files containing information that other applications can consume. Integrators have to deal with transforming files into different formats. Files are produced at regular intervals according to the business needs.**



File transfer pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Decisions we have to make:

- **File Format**: the output format of one application and the input format need of another usually **differ**. Moreover, files often have to be **processed** during their way. Suitable **standard file formats** were born **(e.g. COBOL for mainframes, text based files for UNIX, or XML)**. It triggered the development of **readers, writers** and **transformation tools** for these formats.

- **Time of producing and consuming files**: these actions need effort, so **their frequency should not be too high**. Usually **business cycle** drives their rhythm: e.g. nightly, weekly, etc. In timing the **freshness needs** of the consumers also have to be taken into account.

File transfer pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Properties of file transfer based integration:

- **no knowledge of the internals** of the applications needed
- either the application **developers** have to deal with the **file format** or the **integrators** have to care with **transformation**
- applications are **decoupled** from each other **(they can change freely while produce the same data in the same format into the file)**
- **no extra tools or packages** are needed
- **developers** have to:
  - agree on **filename** and **location**
  - keep the **filenames unique**
  - agree on **who and when deletes** the file
  - ensure the **locking** file and **timing** the file **access**
  - ensure the **file transfer** from one disk to another

File transfer pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Frequency of file update or new file creation:

Frequent

- lots of extra **effort**
- extra **resource** need
- handling **lots of files:**

ensuring their reading and

not to let them be lost

Infrequent

- the system gets **out of synchronization**
- results **inconsistency**
- more likely and painful **problems**

Sometimes **delay** is acceptable, sometimes not.
If **fine grained files** have to be applied, it is **easier to use messaging** instead.

File transfer pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Related patterns:

- **Shared Database:** in case of more **quicker data availability** and **agreed-upon data format**

- **Remote Procedure Invocation:** for cases when we intend to integrate application **functionalities rather than their data**

- **Messaging:** for **frequent** data exchange (even for remote **method invocation**)

Shared Database pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Problem:</u> How can I integrate multiple applications so that they can work together and they can exchange information?

Rapid and **consistent data share** are needed.

Possible **problems**:

- **File Transfer** lacks **timelines**, it may cause **staleness of data, inconsistency**. If applications always have the **latest data**, it increases **people's trust** and **reduces errors**. In case of inconsistent data update in the hurry we have to determine which data is valid, so a **master source** is needed.

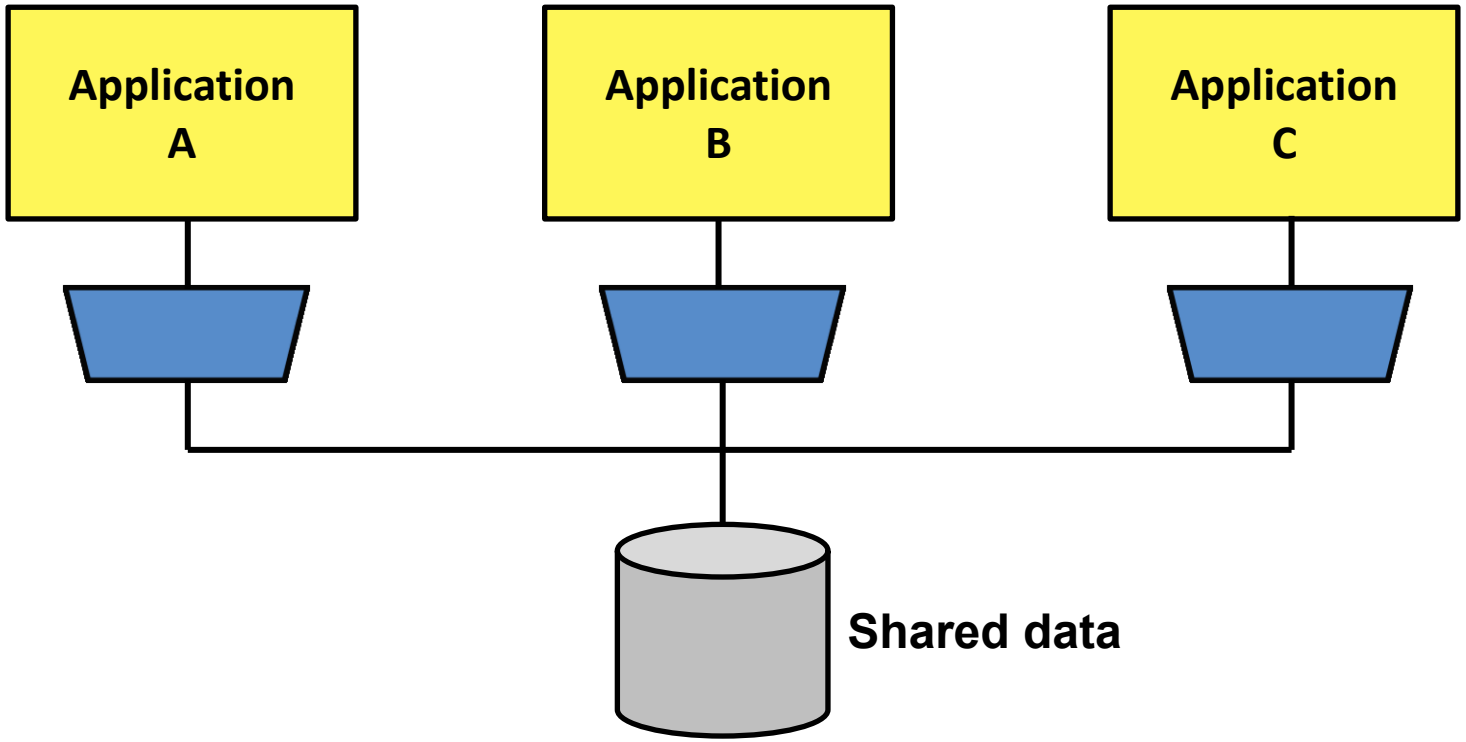
- **incompatible ways of looking at the data may happen (e.g. one database defines oil well as a single drilled hole, while another contains oil well as multiple holes covered by a single equipment)**

Shared Database pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A **central, agreed-upon datastore** is needed. It should be **accessible** by any of the applications **whenever they need** it.

<u>Solution:</u> Integrate applications by having them store their data in a single Shared Database.



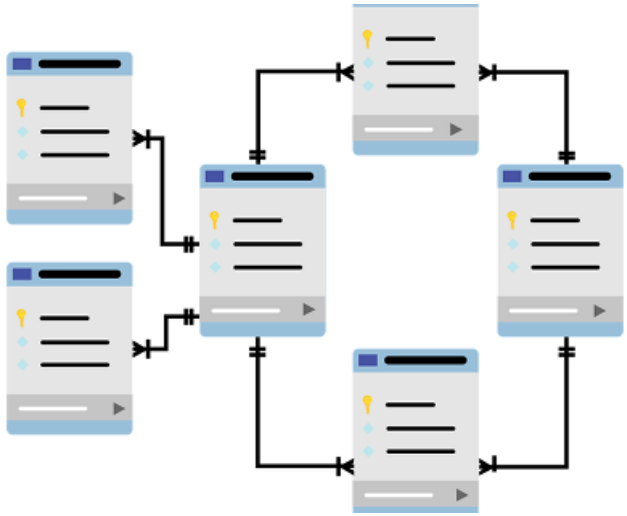
Shared Database pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

One **single database** ensures data **consistency**.If **simultaneous update** origins from different data sources to the same data element, **transaction management system** has to be applied.

SQL-based relational databases are popular, most application development frameworks are able to handle them.

It eliminates the problem of multiple file formats, however, **semantic dissonance** may happen. It has to be handled before large amounts of incompatible data is collected.



Shared Database pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Problems to be solved:

- **Database schema** has to be designed to **meet the needs of multiple** applications

- **More time for development may be required** because of preparing the application to work together with a unified schema (so, some departments may prefer application separation)

- **External packages** sometimes do not work well **only with their own schema**

- **Frequent read and write** may cause **performance bottleneck**

- **Locking conflicts** may happen (even deadlocks)

- In case of **distributed** applications **which computer to store** the database?

Shared Database pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Related patterns:

- **File Transfer**

- **Remote Procedure Invocation:** for cases when we intend to integrate application **functionalities rather than their data**

- **Messaging:** for **frequent** data exchange (even for remote **method invocation**), where **data format may vary** from message to message **(instead of one universal schema)**

Remote Procedure Invocation pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Problem:</u> How can I integrate multiple applications so that they can work together and they can exchange information?

Data and function share is needed in a responsive way.

Shared database and file transfer enable data sharing, but sometimes **function sharing** also needs. To invoke a function it requires **more knowledge** about the internals of the remote application.

In **RPC** applications **hide their data through a function call** interface /**encapsulation**/ (e.g. data change or data access via a function call).

Shared database provides **unencapsulated** data

Change of an application results in **changes of the shared database** what requires **changes of the related other applications**.

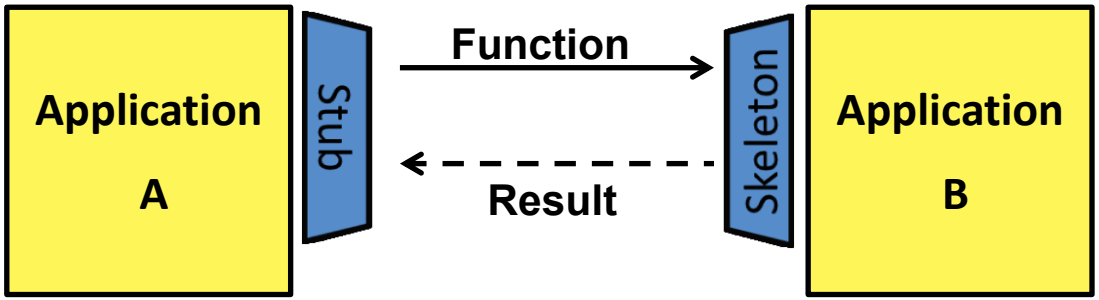
In **file transfer** pattern application can **change after reads** the changed data from the file, but there is **delay**.

Remote Procedure Invocation pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A pattern is needed for letting applications **pass data** to another application by **invoking a function** that **determines what to do with the passed data**.

<u>Solution:</u> Develop each application as a large-scale object or component with encapsulated data. Provide an interface to allow other applications to interact with the running application.



Applications get their **own local data store** and have the responsibility to **maintain the integrity** of data. Moreover, local data can be **modified without affecting other applications**.

Remote Procedure Invocation pattern III.

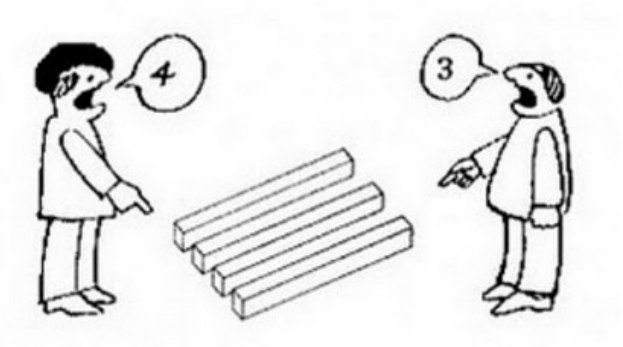
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

RPC approaches (like CORBA, COM, .NET Remoting, Java RMI) differ in their **capabilities** (e.g. transactions), **ease of use** and **how many systems support them**.

Web service related standards suit well for RPC: **XML, SOAP** and **HTTP** (it goes through firewalls). Applying them, data is wrapped, so semantic dissonance is handled.

The **same data** can be handled **through different interfaces**:

- it supports **multiple points of view**
- **transformation components** are needed and applications have to **negotiate their interfaces**



Remote Procedure Invocation pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Applying **RPC**:

- developers are **familiar** with function call
- there are huge **differences** **(performance, reliability)** **between local function call and RPC**

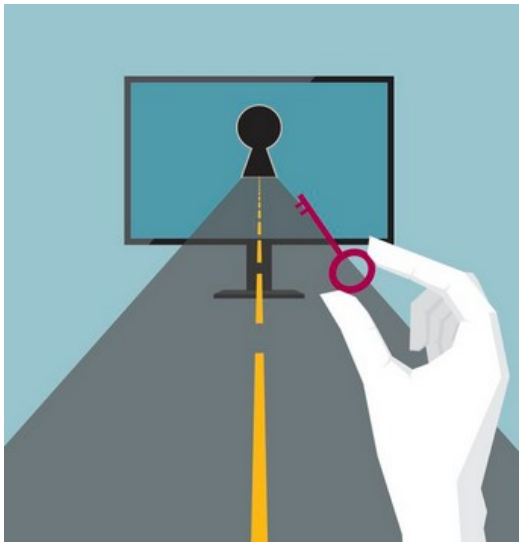
Coupling:

- RPC's **encapsulation reduces** it
- applications are **still coupled** (e.g. function signature)
- especially **sequencing** (strict order of actions) is sensitive for individual changes

The issues that are **not important** in case of a **single application become often significant** during **integration**.

Remote Procedure Invocation pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



Related patterns:

- **File Transfer**

- **Shared Database**

- **Messaging:** for **frequent** data exchange (even for **remote functionality invocation**)

Messaging pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Problem:</u> How can I integrate multiple applications so that they can work together and they can exchange information?

Data and function share is needed in a responsive way.

	File Transfer	Shared Database	Remote Procedure Invocation
share functionality, too (not only data share)	-	-	+
decoupled applications	+	-	-
responsive	-	+	+
reliability	+	+	-

Messaging pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

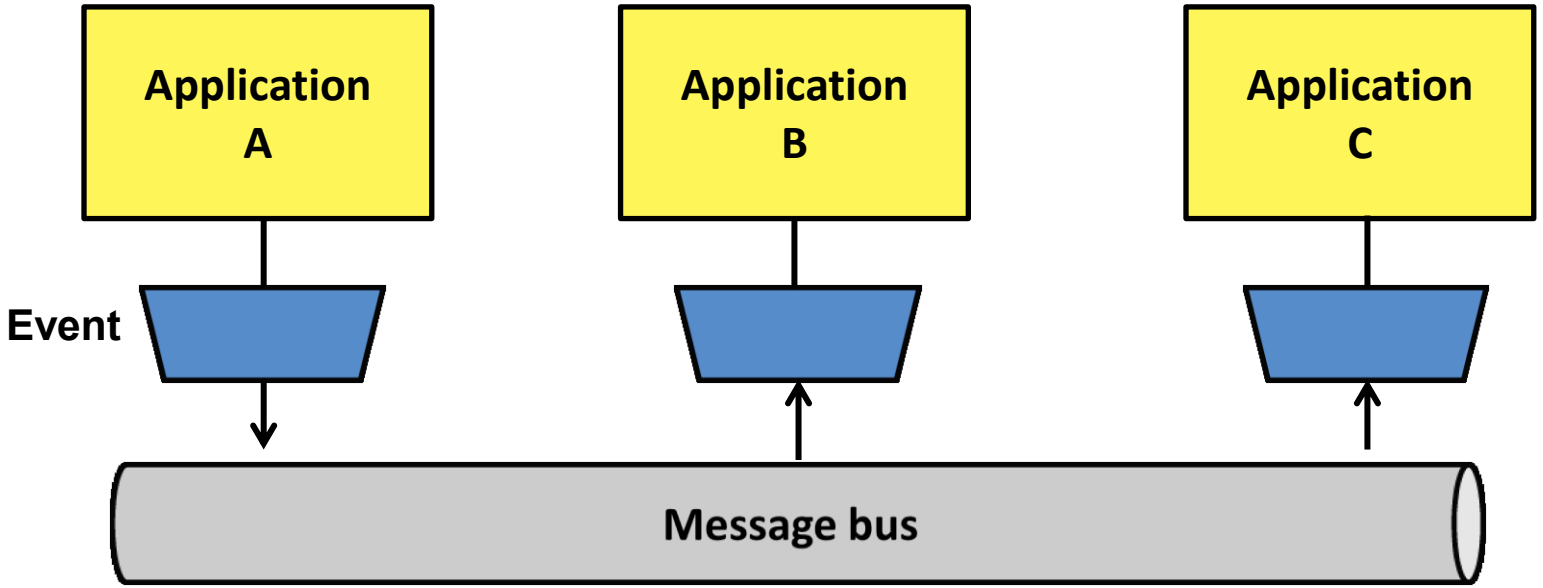
We need a solution with the following properties:

- like in File Transfer, several data pieces have to be **produced quickly, transferred easily**, the receiver has to be **notified about data arrival**
- it ensures successful transmission by **retry**
- an **internal data structure** has to be **hidden**, its **modification (e.g. because of changing needs of the enterprise)** should have **no influence** on the related applications
- **fault-tolerant access** of **remote functionalities**
- **asynchronous** data transfer

<u>Solution:</u> Use Messaging to transfer packets of data frequently, immediately, reliably, and asynchronously, using customizable formats.

Messaging pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



Asynchronous messaging is well applicable solution **for distributed systems** where the applications are **not surely available continously**.

Since in remote messaging **delay** may happen, designers should place **lots of work locally** and **only selective work remotely**.

Messaging pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Advantages of messaging:

- **decoupled** solution: neither the sender nor the receiver has to know about the transformation → **broadcasting** and **multicasting** become possible.

- because of the transformation, applications may have **quite different conceptual models** **(still semantic dissonance can remain)**

- **separates application development from application integration** **(fits to human nature)**

- by frequent and small messages **fast request-reply pairs can be made** (moreover, the sender does not have to block). **/e.g. financial services also often apply messaging for sending 1000s of messages per seconds/**

Messaging pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

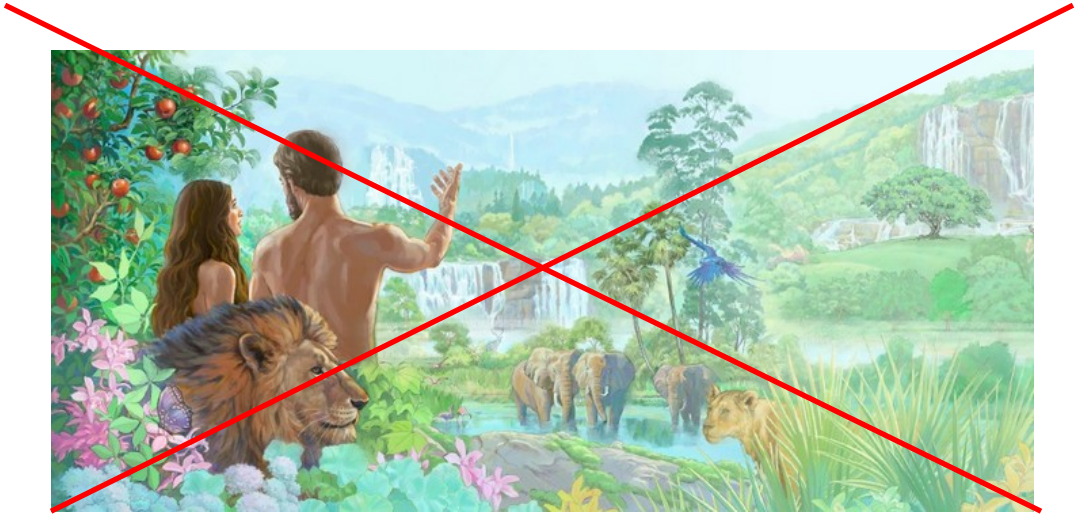
Problems of messaging:

- even high frequency messaging **can not eliminate** totally **inconsistence**.

- **asynchronous** design/thinking is **not easy**. Just like debugging and testing.

- Although applications are **decoupled,** it places **more work on integrators**

(e.g. translations)



Messaging in not a paradise!

Messaging pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Issues to be considered and practises to be employed if using messaging:

- **How to transfer packets of data? /answer: Message and Message Channel/**
- **How to find out where to send the data? /answer: Message Router/**
- **Which data format to use? /answer: Message Translator/**
- **How do application developers connect the application to the messaging system? /answer: Message Endpoints/**

Related patterns: File Transfer, Shared Database, Remote Procedure Invocation Message, Message Channel, Message Endpoint, Message Router, Message Translator

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

THANK YOU FOR THE ATTENTION!

Reference:

Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns – Designing, Building and Deploying Messaging Solutions, Addison Wesley, 2003, ISBN 0321200683

www.enterpriseintegrationpatterns.com

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

ENTERPRISE INTEGRATION PATTERNS

5-6. Messaging systems

Author: Tibor Dulai

SZÉCHENYI 2020

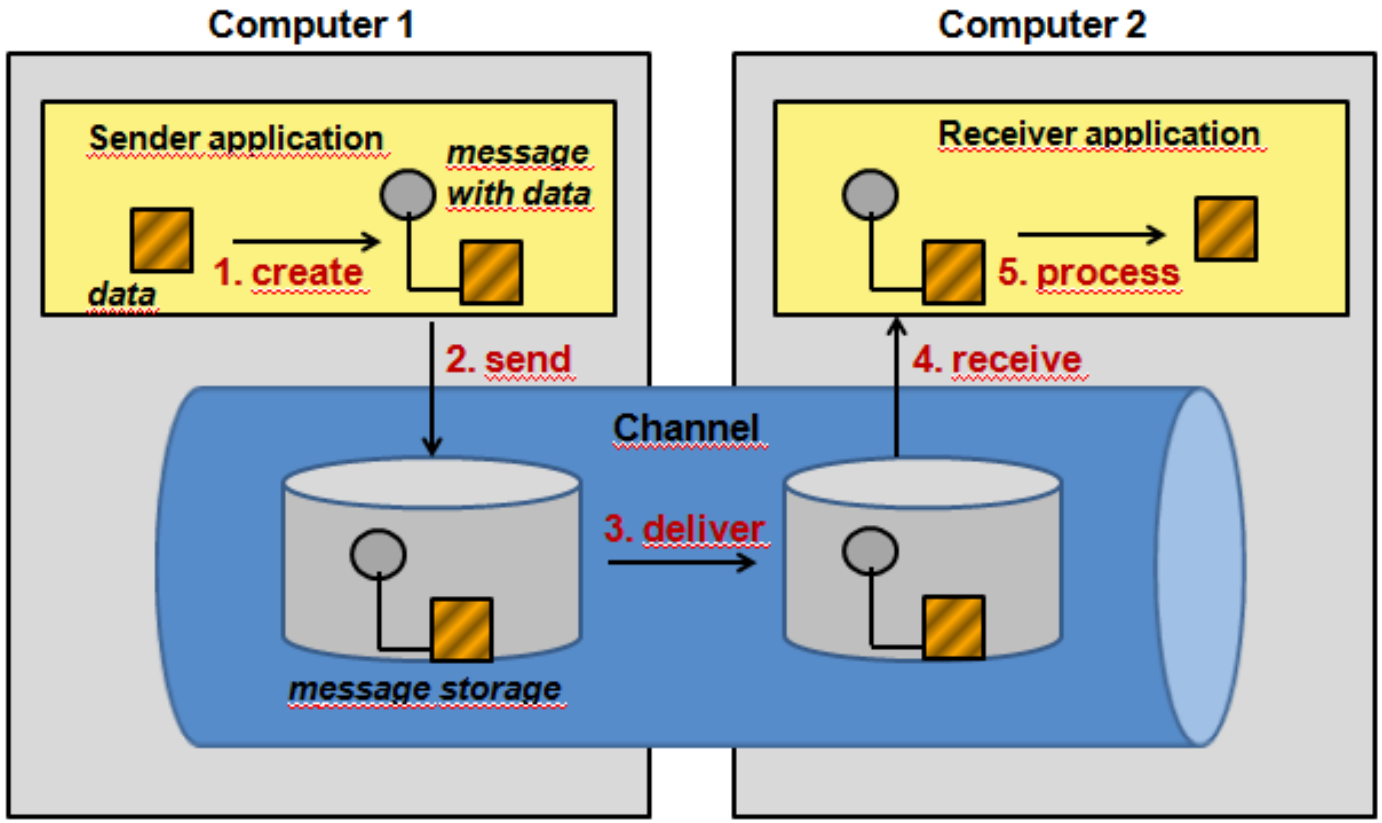
MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

Messaging

- loosely coupled
- asynchronous
- reliable



Basic Concepts of Messaging I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- **Message Channel:** a **virtual pipe** to **connect** the sender and the receiver and to **transmit** data. A newly installed messaging system does not contain any. Integrators have to create them.

- **Message: atomic packet of data**. Data to be sent is **broken** into one or more packet, these packets are **wrapped** into messages and messages are **sent** through the channel.

- **Multi-step delivery (Pipes and Filters): actions** often have to be performed on the message **while it is transmitted** by the channel between the sender and the receiver (e.g. validation, translation)

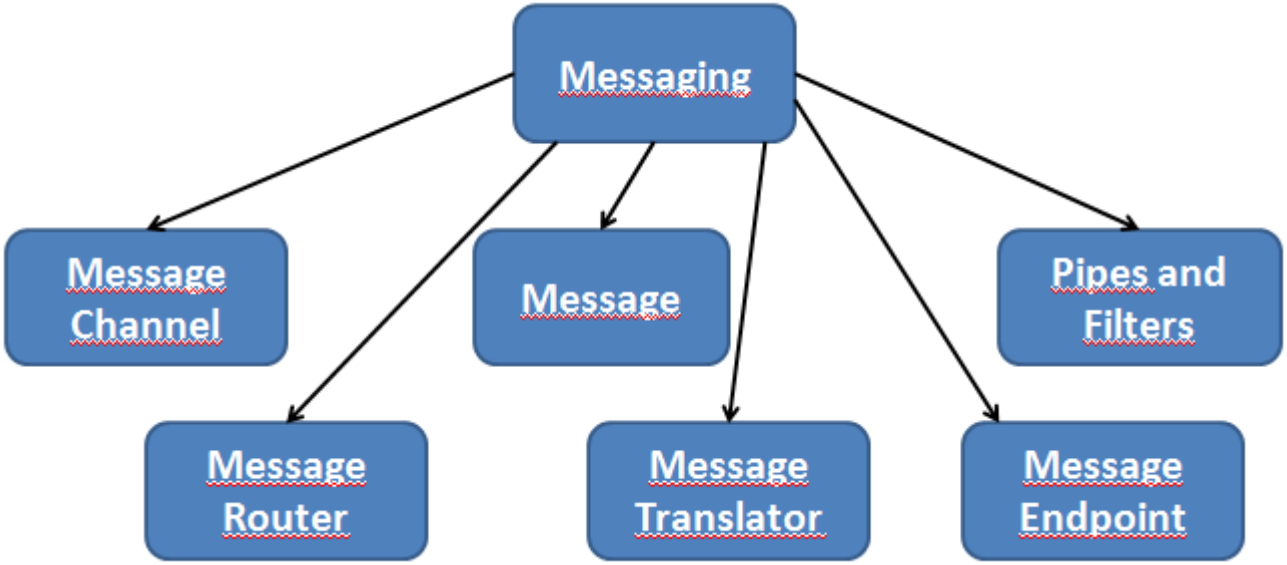
- **Routing (Message Router):** the sender often does not know **which channel** transmits the message to the receiver. The message may travel through several channels on its way. In these cases the sender sends the message to the Message Router, whose task is to **navigate** to the receiver (or to the following router).

Basic Concepts of Messaging II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- **Transformation (Message Translator): converts** one **message format** into another (suitable to the applications)

- **Endpoints:** a layer of a code that **bridges** the **messaging system** and the **application**, knowing the working mechanism of both systems.



EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message Channel pattern

Message Channel pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: two separate applications should communicate with each other via Messaging

Problem: How does one application communicate with another using messaging?

We may think: applications can communicate with each other anytime they want: one application, that gives something to the messaging system and the receiver application grabs anything from the messaging system.
But: it is more **predetermined**: the sender application **knows what sort of information** he sends, and the receiver application doesn't look for just any information, but for particular sorts of information they can use.

Solution: Connect the applications using a *Message Channel, where one application writes information* to the channel and the other one reads that information from the channel.

Message Channel pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

When an application sends or consumes a message, it does not give or grab the message simply to/from the messaging system, but uses a **particular message channel**.

The messaging system has different Message Channels for different types of information which the applications want to communicate.

Channels are logical addresses in the messaging system. Their **implementation can be different** (direct connections between endpoints or a central hub). Sometimes several logical channels are combined into one physical channel.

First, **application developers** have to decide **what channels** will be needed for the communication. Then the **system administrator** who installs the messaging system software must configure it to **set up the channels** that the applications expect.

The **number and the purpose of the channels should be fixed at deployment time** (because every application should know about the channels).

Message Channel pattern III.

EFOP-3.4.3-16-2016-00009

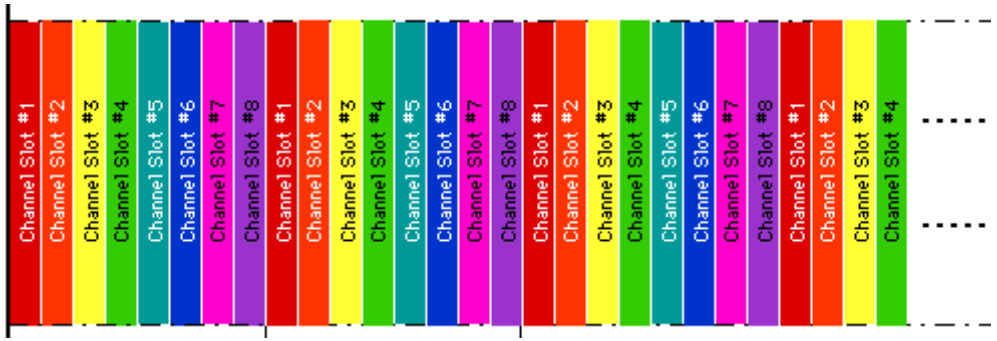
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

When an application developer **creates a channel**, he really has to create only a **reference to an existing message channel** that was created before in the messaging system.

The number of channels: because of the **different type of data** and **several applications**, sometimes 1000s of channels are required. Although they are cheap they **necessitate memory/disk space**.

Datatype Channel helps you determine when you need another channel.

Selective Consumer makes it possible to use **one physical channel as multiple logical channel.**



Message Channel pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Endpoint names:

Sender	Receiver
Producer	Consumer
Publisher	Subscriber
Talker	Listener
Provider	Requester
Server	Client

Channel names:

Usually **alphanumeric**, **e.g.: MyChannel**
Often **hierarchical**: **e.g. MyEnterprise/Dep1/NewOrders**

Channel types:

- Point-to-Point Channel
- Publish-Subscribe Channel

- Datatype Channel: for different data types

- Invalid Message Channel

For connecting applications to a messaging system often **Channel Adapter**s are used.
A set of channels can create a **Message Bus**.

Message Channel pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

J2EE JMS example

j2ee: command to start the reference server
j2eeadmin: tool to configure queues (Point-to-Point Channel) and topics (Publish-Subscribe Channel)

Channel creation:

```
j2eeadmin -addJmsDestination jms/mytopic topic
j2eeadmin -addJmsDestination jms/myqueue queue
```

Their access from the JMS client:

```
Context jndiContext = new InitialContext();
Queue myQueue = (Queue) jndiContext.lookup("jms/myqueue");
Topic myTopic = (Topic) jndiContext.lookup("jms/mytopic");
```

Message Channel pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

IBM WebSphere MQ example

IBM WebSphere MQ for Java implements JMS. Apply IBM WebSphere MQ JMS administration tool.

Channel creation:

```
DEFINE Q(myQueue)
```

Its access from an application (without the full WebSphere Application Server):

```
Session session = // create the session
Queue queue = session.createQueue("myQueue");
```

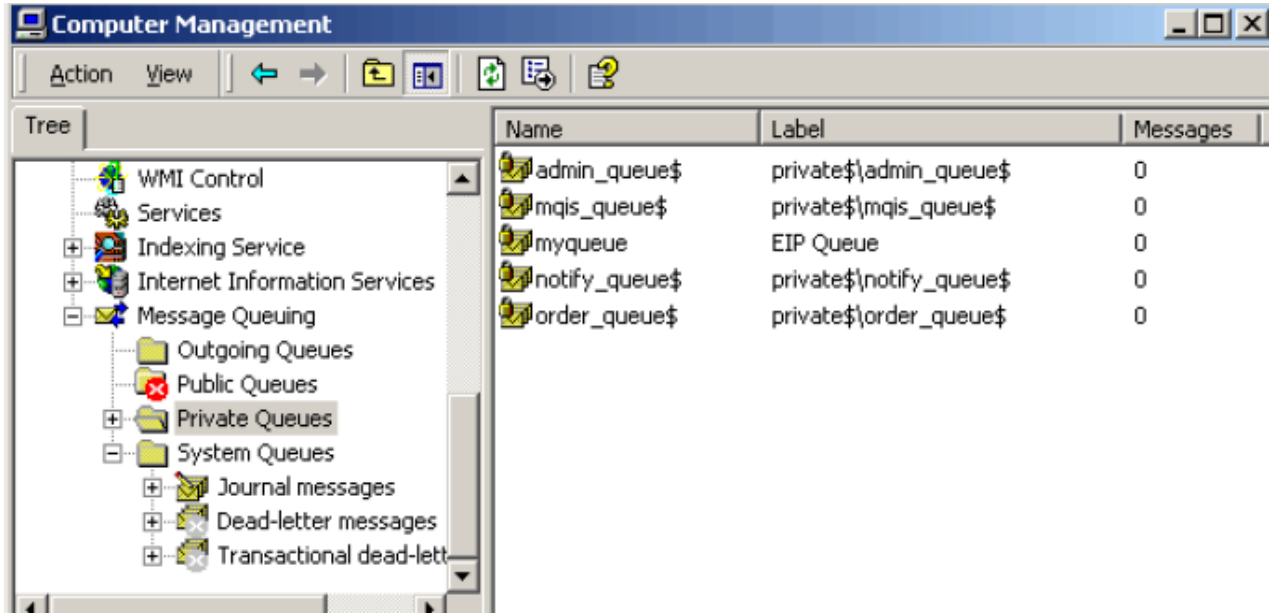
Message Channel pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Microsoft MSMQ example – 1.

Channel creation:

From Microsoft Message Queue Explorer or the Computer Management console:



Message Channel pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Microsoft MSMQ example

Channel creation:

From code:

```
using System.Messaging;
...
MessageQueue.Create("MyQueue");
```

Its access from an application:

```
MessageQueue mq = new MessageQueue("MyQueue");
```

Related patterns: Channel Adapter, Datatype Channel, Invalid Message Channel, Message Bus, Message Endpoint, Selective Consumer, Messaging, Point-to-Point Channel, Publish-Subscribe Channel

Message pattern

Message pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: two separate applications should communicate with each other via Messaging through Message Channel

<u>Problem:</u> How can two applications connected by a message channel exchange a piece of information?

Since data is **not a continuous stream**, a channel transmits **units of data** (like records, objects, etc.)
In local function call, parameters can be passed by **referencing by pointers**, but it **does not work** in a remote call (**because of the different memory spaces**).
So, the sender process has to **marshal the data into byte form, then copy that** between the originating and the receiving computers, which will **unmarshal the data**/

<u>Solution:</u> Package the information into a Message, a data record that the messaging system can transmit through a message channel.

Message pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Data has to be **fragmented into** one or more **messages that can be sent through the message channel**.

Parts of a message:

- **Header**: information for the messaging system (e.g. receiver, sender)
- **Body**: data to be transmitted **(usually transparent to the messaging system)**

Application developers differentiate the messages:

- **Command Message**
- **Event Message**
- **Document Message**
- if reply is required: **Request-Reply**

For the messaging system:

Every message is **the same**: **data to be transmitted** as described by the header.

Message pattern III.

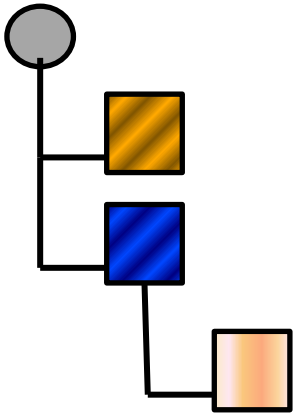
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Other concepts related to the Message:

- **Message Sequence:** it is obtained if the original data is fragmented into more than one message

- **Message Expiration**: indicates time-limited usability of the data contained by the message

- **Canonical Data Model**: an agreed-upon data format between the applications



Message pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

JMS Message Types **(they differ only in their body format)**

- TextMessage **/textMessage.getText() returns a String/**
- BytesMessage **/bytesMessage.readBytes(byteArray) returns byte array/**
- ObjectMessage **/objectMessage.getObject() returns a Serializable/**
- StreamMessage **/readBoolean(), readChar(), readDouble(), etc./**
- MapMessage **/e.g. getBoolean(„isEnabled”), getInt(„numberOfItems”)/**

.NET Message

The Message class has the following properties:

- Body **/Object/**
- BodyStream **/Stream/**
- BodyType **/int/**

Message pattern V.

A SOAP Message example

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Header>
      <t:Transaction
         xmlns:t="some-URI"
         SOAP-ENV:mustUnderstand="1">
            5
      </t:Transaction>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m="Some-URI">
         <symbol>DEF</symbol>
      </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This SOAP message can be included into a message of the messaging system.

It demonstrates the **recursive nature of messages**.

Related patterns: Canonical Data Model, Command Message, Document Message, Event Message, Message Channel, Message Expiration, Message Sequence, Messaging, Request-Reply

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Pipes and Filters pattern

Pipes and Filters pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: some **events trigger a sequence of actions** **(e.g. arrival of an order requires encryption, authentication and to ensure avoiding duplication)**

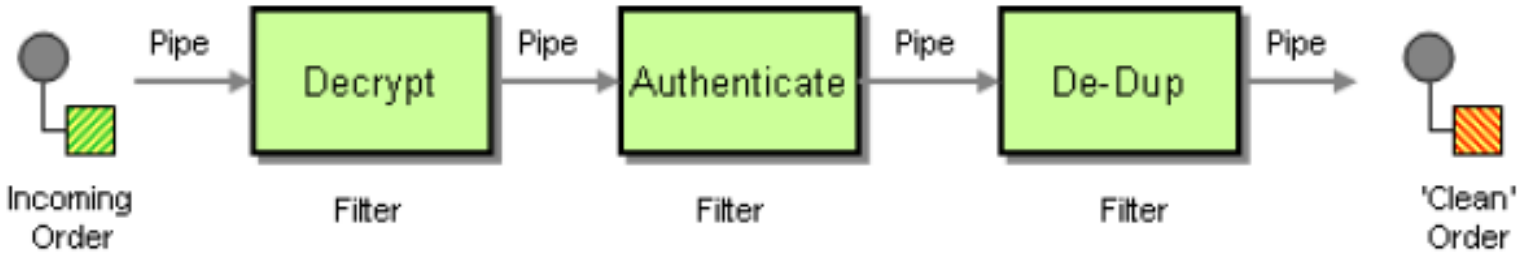
Problem: How can we perform complex processing on a message while maintaining independence and flexibility?

If we develop **one comprehensive solution**, that is **inflexible**, it is **not possible to reuse** and **difficult to test**.
Moreover, the pieces of the required action-sequence may **differ in** the used **programming language**, applied **technology** and **computer** they run on.
The **components** should be **independent** from each other, so that they can be **interchangeable**, offering **generic external interfaces**.
Asynchronous messaging is well applicable for the communication of this kind of components.

Solution: Use the *Pipes and Filters* architectural style to divide a larger processing task into a sequence of smaller, independent processing steps (*Filters*) that are connected by channels (*Pipes*).

Pipes and Filters pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



Each **filter** (a component with arbitrary function) **receives** messages on the inbound pipe, **processes** the message, and **publishes** the results to the outbound pipe.

Every components use the **same interface** → they can be **composed** into arbitrary structures.

The connection between a filter and a pipe is called **port**. Basically each filter has exactly one input and exactly one output port, but in complex structures they can have more.
If we apply messaging, e.g. **Message Router** or **multiple Message Filters** (**applying Point-to-Point Channel** to ensure exactly one consumer per message) we allow the connection between one component and **more than one input/output channel**.

Pipes and Filters pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Pipes and Filters are **basic architectural style** for messaging systems: individual processing steps (filters) are chained together through the messaging channels (pipes). So individual components can be **combine**d easily into larger solutions.

Several **other messaging patterns** (transformation, routing) are based on this architectural style.

Implementation variations of a **pipe**:

- **message channel**: provides platform-, language- and location-independence
- **simple in-memory queue**: if all the components are on the same machine

Abstract pipe interface and **messaging gateway** (for adjusting components) are required.

Pipes and Filters pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Flexibility

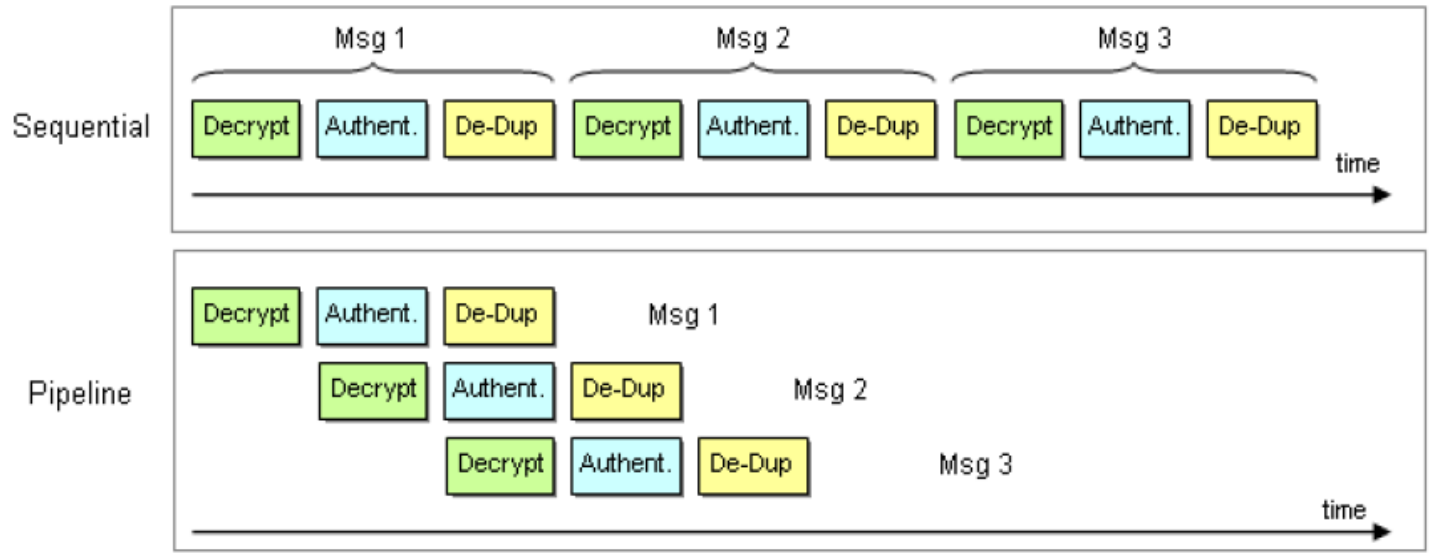
Reuseability

Easy to test (separetely each component)

Lots of channels (memory, CPU)

Lower performance (because of the translations between the components)

Processing structure:



Asynchronous Message Channels make **processing pipeline** possible (each filter has its own **thread**)

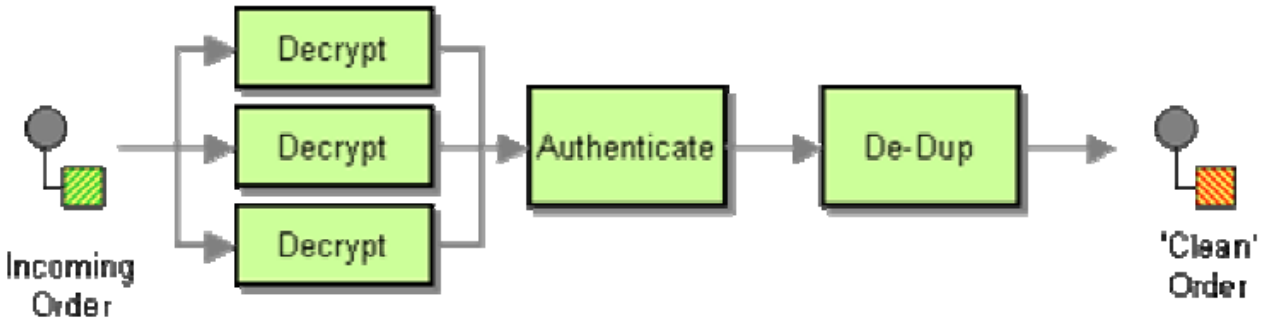
increased throughput

Pipes and Filters pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

In pipelining, the **throughput's bottleneck** is the **slowest component** in the chain.

Parallel instances of the slow filters can eliminate the problem. For that a ***Point-to-Point Channel with Competing Consumers*** is needed. It ensures that each message on the channel is consumed by **exactly one of the N available** consumers.



This solution can be applied **ONLY if the order of the messages does not matter,** or a ***Resequencer*** is applied.
Moreover, it is **easily** applicable **ONLY if each component is stateless (after processing the message it returns to the previous state) /not like de-duper/.**

Pipes and Filters pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A C# / MSMQ example

```
using System;
using System.Messaging;

namespace PipesAndFilters
{
    public class Processor
    {
        protected MessageQueue inputQueue;
        protected MessageQueue outputQueue;

        public Processor (MessageQueue inputQueue, MessageQueue outputQueue)
        {
            this.inputQueue = inputQueue;
            this.outputQueue = outputQueue;
        }

        public void Process()
        {
            inputQueue.ReceiveCompleted += new
ReceiveCompletedEventHandler(OnReceiveCompleted);
            inputQueue.BeginReceive();
        }
```

event-driven consumer

contd.

Pipes and Filters pattern VII.

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

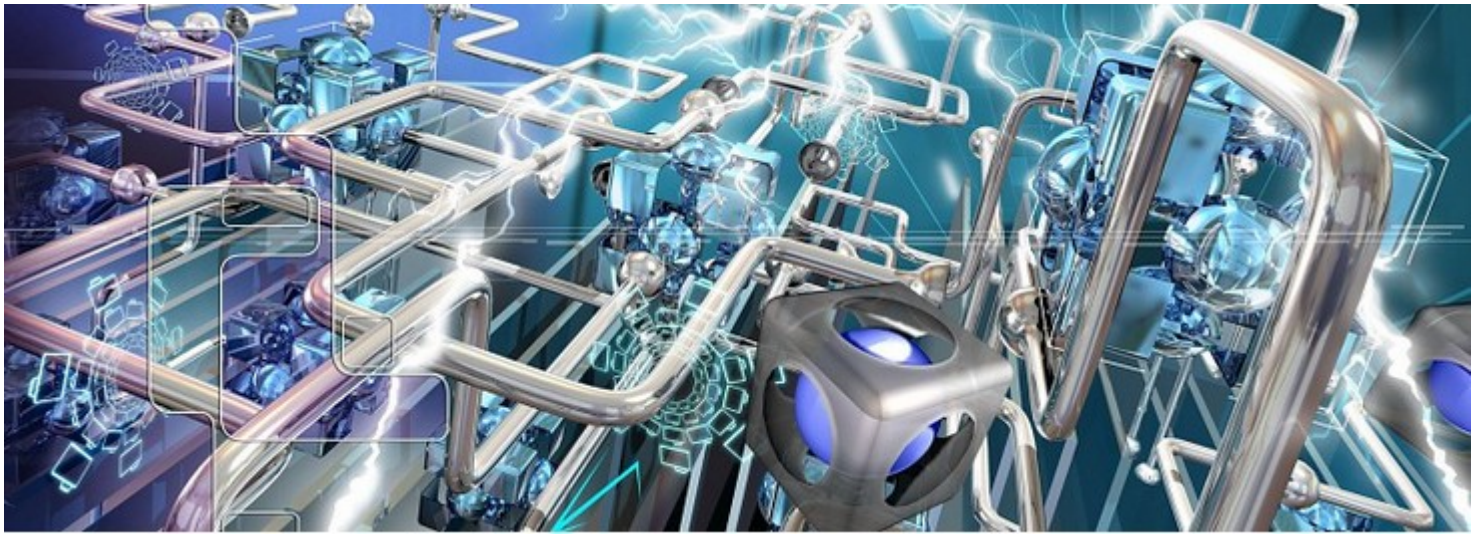
```
      private void OnReceiveCompleted(Object source, ReceiveCompletedEventArgs
asyncResult)
      {
         MessageQueue mq = (MessageQueue)source;

         Message inputMessage = mq.EndReceive(asyncResult.AsyncResult);
         inputMessage.Formatter =  new System.Messaging.XmlMessageFormatter(new
String[] {"System.String,mscorlib"});

         Message outputMessage = ProcessMessage(inputMessage);

         outputQueue.Send(outputMessage);

         mq.BeginReceive();
      }

      protected virtual Message ProcessMessage(Message m)
      {
         Console.WriteLine("Received Message: " + m.Body);
         return (m);
      }
   }
}
```

Pipes and Filters pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The example is **not transactional**: an error during message processing – before sending that to the output – may result in **message lost**.
Transactional Client can be a solution.

Related patterns: Competing Consumers, Content Filter, Event-Driven Consumer, Message Filter, Message Channel, Message Router, Messaging, Messaging Gateway, Point-to-Point Channel, Resequencer, Test Message, Transactional Client



EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message Router pattern

Message Router pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: Filters – representing different processing steps – are connected by message channels.

<u>Problem:</u> How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?

Contrary to the large data sets which require the same handling, individual **messages** usually **require different processing steps**.

One message channel usually transmits **several messages of different sources**, that require different handling.
a. If they are separated into different channels **and treated based on their types**, then:

- **the sender has to send** the message **to the proper channel**
- it results in **lots of channels**
- sometimes **destination depends not only on the originator (e.g. on the message number transmitted on that channel so far)**

Message Router pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

b. Applying **Publish-Subscribe Channels** consumers can subsribe to the channel:

- the **sender does not have knowledge about the destinations**
- a subsribed consumer will **get all messages** from that channel **(no individual message handling)**
- **no flexibility** in the structure of the chain

c. If we use **Point-to-Point Channel delegating the decision** about consuming a message **to the destination**:

- a **misconsumed message cannot be put back** onto the channel
- only some messaging systems make it possible for a component to **inspect the content** of a message while that is still **in the channel (moreover, in these cases the application is bounded to specific message formats that reduce the possibility of reuse and composability)**

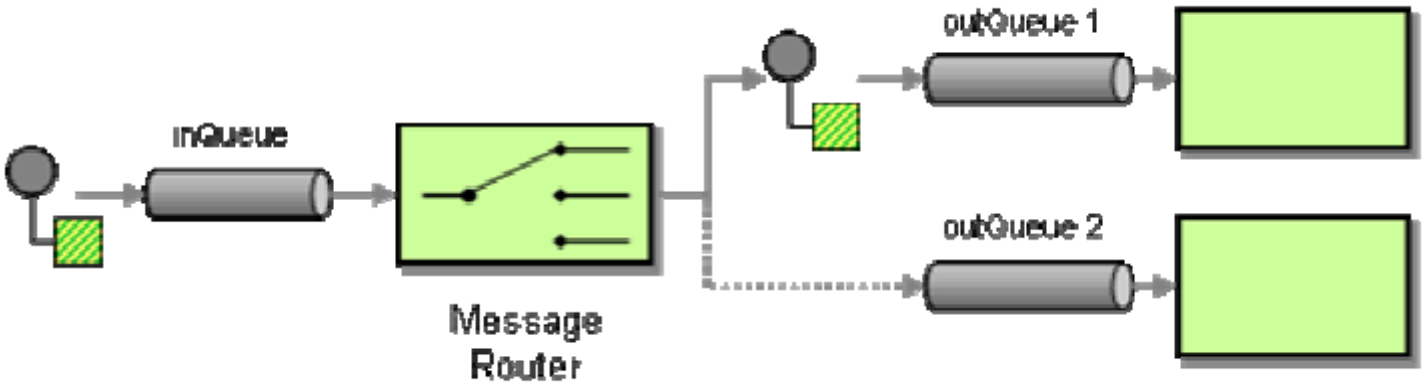
Message Router pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

These solutions require the **modification of the components** that is **impossible** in several cases.

However, **Pipes and Filters** model makes it possible to **insert a new component** between two existing – **unmodified** – applications **which component will determine the next processing step.**

<u>Solution</u>: Insert a special filter, a Message Router, which consume a Message from one Message Channel and republishes it to a different Message Channel depending on a set of conditions.

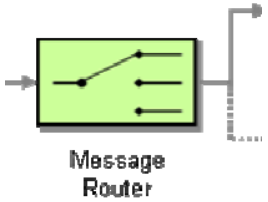


Message Router pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Speciality of the Message Router is that it connects to **multiple output channels**. Connecting **components do not have to be modified**, moreover, the **message content** also **stays unmodified**.

Message Router **determines the destination** of the message, so **decision criterion is handled only here (new message types, new components, changed routing rules do not influence the other components)**. Moreover, the **message order remains**.



Knowledge need:

The message router **decouples** components **/predictive routing/**. Its price is that the **router has to have knowledge** about the possible destinations.
If these vary frequently, it causes **bottleneck** at the router. In these cases it is **better if the destinations decide** based on their interest – **Publish-Subscribe Channel** and **Message Filters** are needed **/reactive filtering/**.

Message Router pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Applying a Message Router, messages are **put from one channel to another**. It may require **decoding/coding**, resulting in **decrease of performence**.
Solution can be a **paralell** application of a set of routers.

„Big picture" of the overall message flow:
Message Routers support the creation of **loosely coupled** systems. However, in these systems it is **hard to determine how messages really flow** from the source to the destination. It **makes testing, debugging and maintenance very complex**.
Solution:

- use of ***Message History*** **(follows which components a message goes through)**
- collect **channel subscription information** **(it helps to draw the graph of the possible message flows)**

Message Router pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

What the decision criteria can be based on? - (router types)

- **no criterion**: a ***fixed router***: **only one input and one output** channel. Role of a **relay** **(e.g. to sepatare into subsystems or to connect multiple integrated solutions – usually combined with Message Translator or Channel Adapter)**

- **properties of the message**: ***Content-Based Router***

- **environment conditions**: ***Context-Based Routers*** **/e.g. for load-balancing, test or failover functionality/**
A simple **Message Channel also ensures** a type of **load balancing** (the quickest consumer gets the message), but **Context-Based Routers may have more complex built-in intelligence**.

- ***stateless*** vs. ***stateful routers***: are the content of the **previous messages** needed for making the decision? **/e.g. de-douping/**

- *hard-coded logic*, ***Control Bus,*** or ***Dynamic Router*****:** on Control Bus the **middleware solution** can modify the decision criteria **/e.g. switch between 'test' or 'production' mode/**, while Dynamic Router gets control messages from the **potential recipients**.

Message Router pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

In commercial EAI tools ***Message Broker*** has routing functionality:

- **receives** messages, **validates** them, **transforms** them and **routes** them
- usually **applications** which we want to connect, **can not be modified**

A C# / MSMQ example for Message Router

```
class SimpleRouter
{
    protected MessageQueue inQueue;
    protected MessageQueue outQueue1;
    protected MessageQueue outQueue2;

    public SimpleRouter(MessageQueue inQueue, MessageQueue outQueue1, MessageQueue
outQueue2)
    {
        this.inQueue = inQueue;
        this.outQueue1 = outQueue1;
        this.outQueue2 = outQueue2;

        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();
    }
```

Message Router pattern VIII.

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

```
    private void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        if (IsConditionFulfilled())
            outQueue1.Send(message);
        else
            outQueue2.Send(message);

        mq.BeginReceive();
    }

    protected bool toggle = false;

    protected bool IsConditionFulfilled ()
    {
        toggle = !toggle;
        return toggle;
    }

}
```

Message Router pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The example is **not transactional**: an error during message processing – before sending that to one output – may result in **message lost**.
Transactional Client can be a solution.

Related patterns: Channel Adapter, Content-Based Router, Control Bus, Datatype Channel, Dynamic Router, Message Filter, Message, Message Channel, Message History, Message Translator, Pipes and Filters, Publish-Subscribe Channel, Transactional Client

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message Translator pattern

Message Translator pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: **Applications** to integrate often have their **own proprietary data model (e.g. customer-related data: tax payer ID or email address)**. This data model has **impact on the interface/database** of the applications and on the message, too. However, **integration solutions** usually **apply standardized data format (e.g. based on consortia like RosettaNET, ebXML, OAGIS)** e.g. to be opened for external parties.

Problem: How can systems using different data formats communicate with each other using messaging?

Possible approaches:

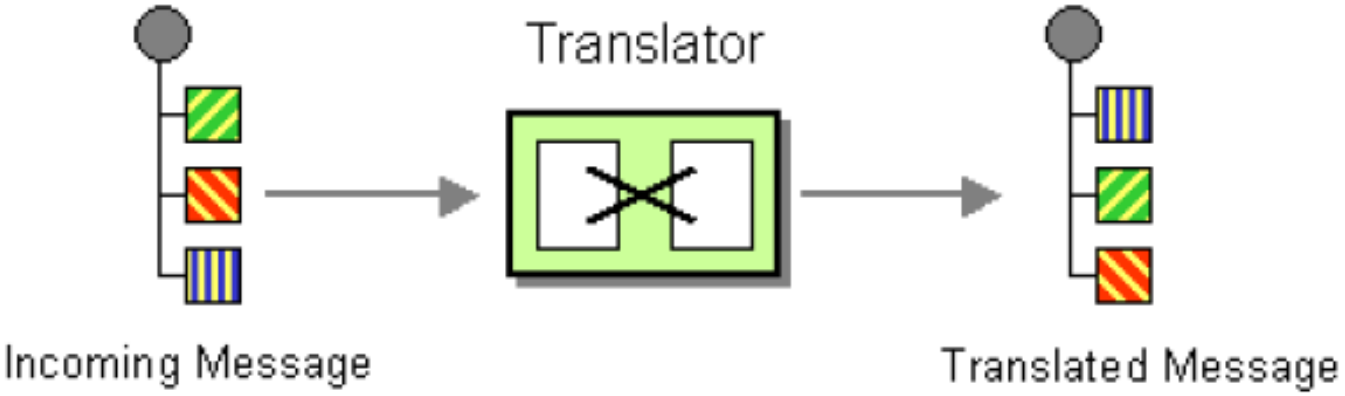
1. **Modifying the application** to use the common data format:
   - is often **impossible (e.g. packaged application)**
   - is **difficult, risky, expensive (e.g. Y2K)**
   - the **physical data representations** may **differ**
   - results in **tightly coupling** solution

2. Hard coding the **translation into the Message Endpoint**:
   - requires **access to the code** of the Endpoint
   - **reduces code reuse**

Message Translator pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution</u>: Use a special filter, a Message Translator, between other filters or applications to translate one data format into another.



Improvement of decoupling:

Message Channel applications do not have to know each other's **location**

Message Router applications do not have to agree on a **common message router**

Message Translator applications do not have to know each other's **data format**

Message Translator pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Levels of translation

Layer	Deals With	Transformation Needs (Example)	Tools / Techniques
Data Structures (Application Layer)	Entities, associations, cardinality	Condense many-to-many relationship into aggregation.	Structural Mapping Patterns Custom code
Data Types	Field names, data types, value domains, constraints, code values	Convert zip code from numeric to string. Concatenate *first name* and *last name* fields to single *name* field. Replace US state name with two character code.	EAI visual transformation editors XSL Database lookups Custom code
Data Representation	Data formats (XML, name-value pairs, fixed-length data fields etc., EAI vendor formats) Character sets (ASCII, UniCode, EBCDIC) Encryption / compression	Parse data representation and render in a different format. Decrypt/ encrypt as necessary.	XML Parsers, EAI parser / renderer tools Custom APIs
Transport	Communications Protocols:TCP/IP sockets, http, SOAP, JMS, TIBCO RendezVous	Move data across protocols without affecting message content.	*Channel Adapter* EAI adapters

Message Translator pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Layer-related properties/questions:

- **Transport Layer:** ensures **complete** and **reliable** data transfer

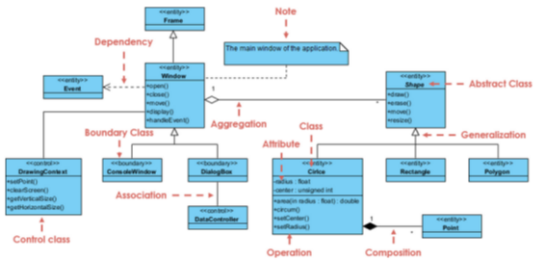
- **Data Representation** (syntax) **Layer**: it is needed because transport layer deals only with **character or byte streams**

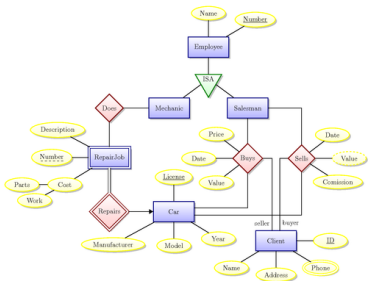
- **Data Types Layer:** usually applies **Data Dictionaries**

- **Data Structures Layer:** defines **logical entities and the relationship between them (e.g. Customer, Account, Address).**

  E.g.: - can a customer have multiple addresses or accounts?
  - can customers have shared address or account?
  - is the address part of the account?

  Class diagrams and **entity-relationship diagrams** are useful in this layer.



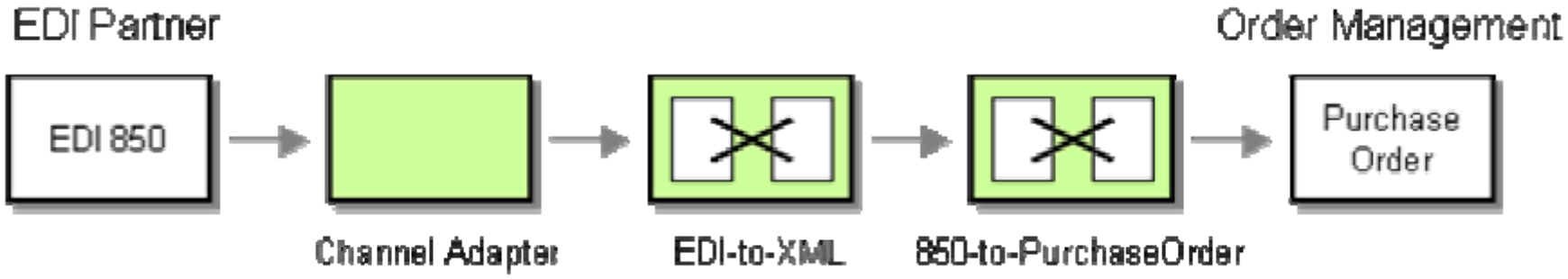


Message Translator pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Translator chaining:

Transformation is often required in **more than one layer**. However, the layers can be treated **separatedly**.
It allows **reuse** and **interchangeability**.
Message translators of each layer can be **chained by Pipes and Filters**.



The above example:

EDI 850 PurchaseOrder record as a fixed-format file

Order object represented by an XML document sent by http protocol

Message Translator pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message Translator variations

- **Content Enricher**: **adds data** item to the content of a message

- **Content Filter**: **removes data** item from the content of a message

- **Claim Check**: removes data element from the message but **stores for later usage**

- **Normalizer**: converts messages into a **consistent message format**

- **Canonical Data Model**: results in **data format decoupling**

- **Message Bridge**: **connects multiple messaging systems** at the Transport Layer

Message Translator pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Transformation by XSL

W3C defined **XSL (Extensible Stylesheet Language)** as a standard language for processing XMLs. **XSLT (XSL Transformation)** is a part of XSL for transforming XML documents.

The **incoming XML** document:

```
<data>
    <customer>
        <firstname>Joe</firstname>
        <lastname>Doe</lastname>
        <address type="primary">
            <ref id="55355"/>
        </address>
        <address type="secondary">
            <ref id="77889"/>
        </address>
    </customer>
    <address id="55355">
        <street>123 Main</street>
        <city>San Francisco</city>
        <state>CA</state>
        <postalcode>94123</postalcode>
        <country>USA</country>
        <phone type="cell">
            <area>415</area>
            <prefix>555</prefix>
            <number>1234</number>
        </phone>
        <phone type="home">
            <area>415</area>
            <prefix>555</prefix>
            <number>5678</number>
        </phone>
    </address>
    <address id="77889">
        <company>ThoughtWorks</company>
        <street>410 Townsend</street>
        <city>San Francisco</city>
        <state>CA</state>
        <postalcode>94107</postalcode>
        <country>USA</country>
    </address>
</data>
```

Message Translator pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The desired **output XML** format:

```
<Kunde>
    <Name>Joe Doe</Name>
    <Adresse>
        <Strasse>123 Main</Strasse>
        <Ort>San Francisco</Ort>
        <Telefon>415-555-1234</Telefon>
    </Adresse>
</Kunde>
```

The **transformation** has to:

- change the **tag names**
- **merge** some fields into one
- **pick** only one address and one phone number **(e.g. based on some business logic)**

Message Translator pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The appropriate **XSL** for making the transformation (1/2):

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" indent="yes"/>
   <xsl:key name="addrlookup" match="/data/address" use="@id"/>
   <xsl:template match="data">
      <xsl:apply-templates select="customer"/>
   </xsl:template>
   <xsl:template match="customer">
      <Kunde>
         <Name>
            <xsl:value-of select="concat(firstname, ' ', lastname)"/>
         </Name>
         <Adresse>
            <xsl:variable name="id" select="./address[@type='primary']/ref/@id"/>
            <xsl:call-template name="getaddr">
               <xsl:with-param name="addr" select="key('addrlookup', $id)"/>
            </xsl:call-template>
         </Adresse>
      </Kunde>
   </xsl:template>
   <xsl:template name="getaddr">
      <xsl:param name="addr"/>
      <Strasse>
```

Message Translator pattern X.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The appropriate **XSL** for making the transformation (2/2):

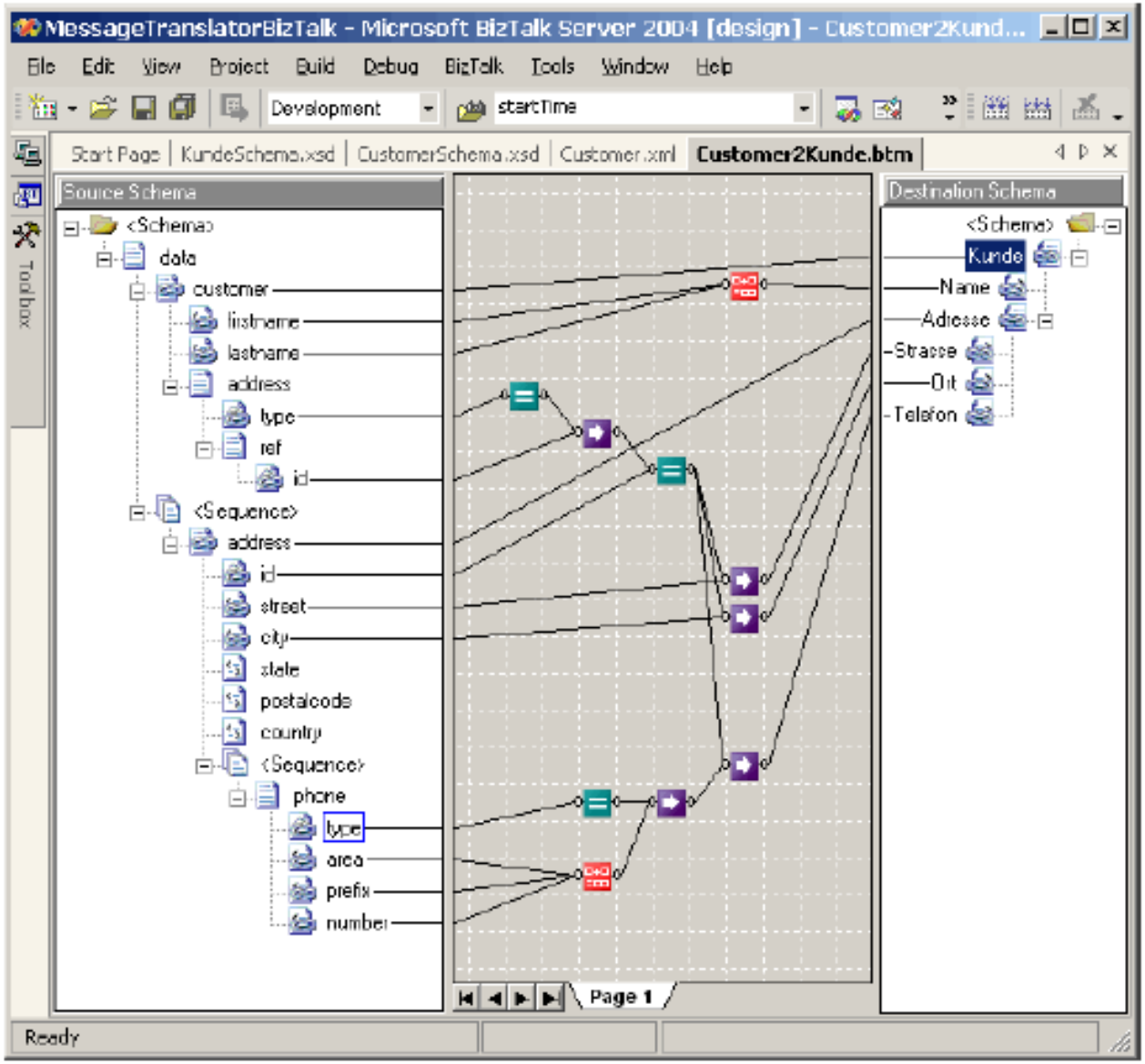
```
            <xsl:value-of select="$addr/street"/>
        </Strasse>
        <Ort>
            <xsl:value-of select="$addr/city"/>
        </Ort>
        <Telefon>
            <xsl:choose>
                <xsl:when test="$addr/phone[@type='cell']">
                    <xsl:apply-templates select="$addr/phone[@type='cell']"
mode="getphone"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:apply-templates select="$addr/phone[@type='home']"
mode="getphone"/>
                </xsl:otherwise>
            </xsl:choose>
        </Telefon>
    </xsl:template>
    <xsl:template match="phone" mode="getphone">
        <xsl:value-of select="concat(area, '-', prefix, '-', number)"/>
    </xsl:template>
    <xsl:template match="*"/>
</xsl:stylesheet>
```

Message Translator pattern XI.

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Visual Transformation Tool
(Microsoft BizTalk Mapper of Visual Studio)



Related patterns:
Canonical Data Model, Channel Adapter, Content Filter, Content Enricher, Message Channel, Message Endpoint, Message Router, Message Bridge, Normalizer, Pipes and Filters, Shared Database, Claim Check

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message Endpoint pattern

Message Endpoint pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: Applications are communicating by sending **messages through message channels**.

Problem: How does an application connect to a messaging channel to send and receive messages?

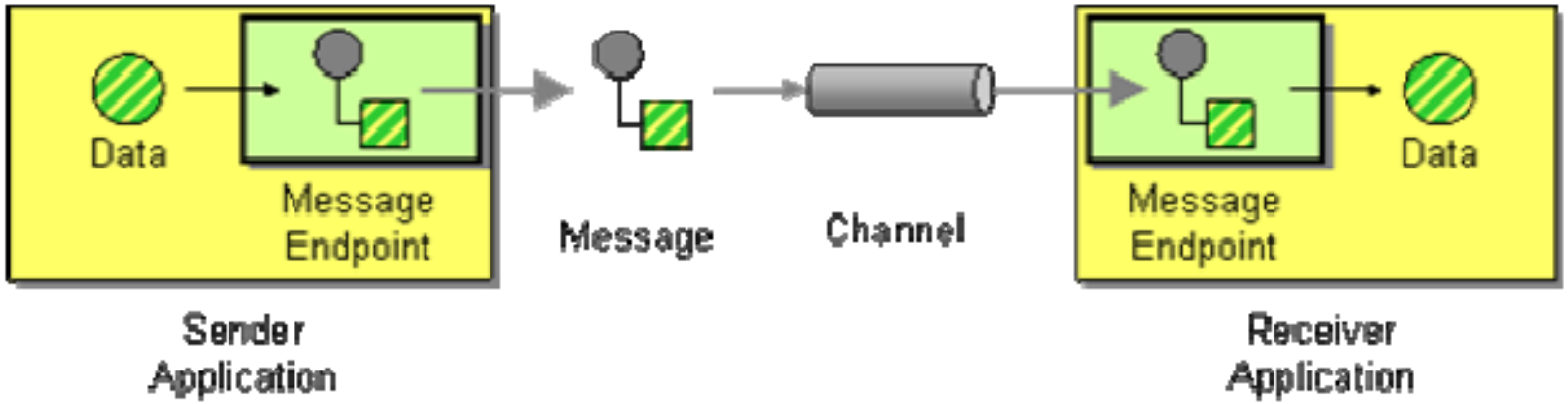
Since **application** and **messaging system** are **separate**, they have to be connected for working together.

A messaging system acts like a **messaging server** for the applications (clients) **/accepts and delivers messages/**. For the interaction a **client API** is defined that is **messaging domain-specific**.
So applications have to contain a **code that connects** itself to the messaging domain.

Solution: Connect an application to a messaging channel using a Message Endpoint, a client of the messaging system that the application can then use to send or receive messages.

Message Endpoint pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



From the **application** side:

it has to **send a piece of data** to an application and/**or it expects data**.

From the **message endpoint** side:

it **takes the data** from the application, **wraps** it into a message and **sends it into an appropriate message channel**;
or it **receives a message**, **extracts** its content, and **gives that** to the application

Message Endpoint pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A message endpoint offers a **general messaging API** to a **specialized application**.
So, **in case of changing** the messaging (communication) system, **only the message endpoint has to be rewritten**.

A message endpoint is **channel-specific**, so **for multiple channel or for a duplex communication multiple message endpoint** instances are required.

Sometimes **more than one message endpoint** is used **for one message channel (e.g. in case of multiple threads on the same channel).**

Message Endpoint pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

a Message Endpoint:

- is a **Channel Adapter specialized for and built into** an application

- has to be created as a **Messaging Gateway** that **hides** the messaging system from the application

- can implement a **Message Mapper** that **transfers data between domain objects and messages**

- can be a **Service Activator** that makes **asynchronous message access possible from a synchronous service**

- can **control transaction** with the messaging system as a **Transactional Client**

Message Endpoint pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message receiving types by a Message Endpoint:

Polling Consumer ⟷ **Event-Driven Consumer**

How to get messages from a channel? :

Competing Consumer ⟷ via a **Message Dispatcher**

Another related concepts:

- **Selective Consumer**: decide whether to consume or discard a message?

- **Durable Subscriber**: does **not miss** a message which was published **while** the consumer **was not online**

- **Idempotent Receiver**: eliminates duplications

Message Endpoint pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example **terminologies:**

JMS: ***MessageProducer*** or ***MessageConsumer***

.NET: ***MessageQueue*** **(just like a Message Channel)**

Related patterns: Channel Adapter, Competing Consumers, Durable Subscriber, Event-Driven Consumer, Idempotent Receiver, Message, Message Channel, Message Dispatcher, Selective Consumer, Service Activator, Messaging Gateway, Messaging Mapper, Polling Consumer, Transactional Client

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

THANK YOU FOR THE ATTENTION!

Reference:

Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns – Designing, Building and Deploying Messaging Solutions, Addison Wesley, 2003, ISBN 0321200683

www.enterpriseintegrationpatterns.com

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

ENTERPRISE INTEGRATION PATTERNS

7-8. Messaging channels

Author: Tibor Dulai

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

Data exchange by messaging I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Sender application **may not know the exact receiver** application, but knows that if the **appropriate channel** is selected to put the data on, then the receiver will be an application who **intends to consume this type of data** from this channel.

Message channel basics 1/2

- **Fixed set of channels**: applications have to know what type of data which channel they can put on, or consume from. So, message channels have to be **agreed upon at design time**. /**Exceptions**: (1) **Reply channel**, whose Return Address is specified in the Request Message; (2) **hierarchical channels** where the receiver subscribes to the parent and the sender sends to a new child/

- **Determination responsibility of the channels**: by the application or by the messaging system? Usually the **application determines what kind of channels it requires from the messaging system**. But, **later coming** applications may use the existing channels (or **join** to existing applications **or require** a new channel).

Data exchange by messaging II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message channel basics 2/2

- **A channel is unidirectional or bidirectional?** It is better like **a bucket distibuted accross multiple computers**. However, since one application sends message onto that and in case of bidirectional channel it could comsume its own message, **message channels are unidirectional**. It means that **a duplex connection requires two channels**.

Message channel decisions I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- **One-to-one or one-to many?:**
  ***Point-to-point channel*:** it can also have multiple receivers, but it ensures that one message will be **consumed only exactly once**.
  ***Publish-Subscribe channel*:** the message will be **copied to all** of the receivers

- **What type of data?:**
  The receiver understands the message only if that **suits to a predescribed type** (just like data types of byte sequences in the computer memory).
  Data on a ***Datatype Channel*** have to be of the same type. However, it can result in a plenty of channels.

- **Invalid and dead messages**:
  receiver applications have **expectations** about the type and meaning of the consumed message. If they **do not meet**, the application puts the message onto the ***Invalid Message Channel*** (and hopes in its monitoring).
  Messages that **can not be successfully delivered**, are put on the ***Dead Letter Channel*** (again, hopefully monitored).

Message channel decisions II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- **Crash proof:**

  what happens to still undelivered messages in case of crash of the messaging system? **Default, they are stored in the memory**, that's why **they will be lost.** However, in case of ***Guaranteed Delivery*** messages are **stored on the disk**, so they **stay alive**. It is worse for performance but better for reliability.

- **Non-messaging clients:**

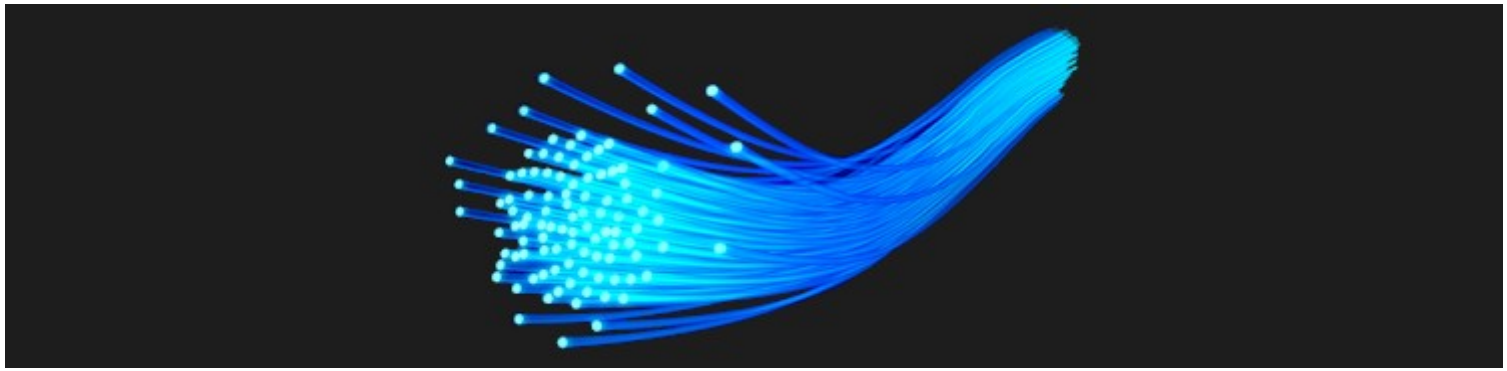
  Some applications **can not connect to the messaging system**, but the messaging system can connect to the application (e.g. via its user interface, TCP/IP, HTTP or through its database). In these cases a ***Channel Adapter*** can be applied to connect the channel and the application **without any modification of the application**.

  In other cases the **application uses another messaging system**. Then an application that is **client to both messaging systems** can build a ***Messaging Bridge***.

Message channel decisions III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- **Communications backbone:**
  As more and more applications connect to the messaging system, it gets a **central role** through that different **functionalities can be reached**. This communication backbone is called ***Message Bus***. Using it an application only has to know **which channels to use** to request a functionality and which ones to use for getting the response.



EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Point-to-Point Channel pattern

Point-to-Point Channel pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: an application uses messaging to transfer a document or for RPC.

Problem: How can the caller be sure that exactly one receiver will receive the document or perform the call?

In case of a simple RPC, we can be sure that one function is called only once if we want it so. But in case of messaging **more receivers can see** a procedure call wrapped in **message on the channel**.

Solution possibilities:

- **one receiver** is allocated to **one channel** (unnecessary restrictions)
- the process could be **coordinated by all** the receivers of the channel (too much communication, complexity and coupling)

Solution: Send the message on a *Point-to-Point Channel,* which ensures that only one receiver will receive a particular message.

Point-to-Point Channel pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A Point-to-Point Channel ensures that even it has multiple receivers, **only one receiver will consume** succesfully **any of its messages**.

From a Point-to-Point Channel **multiple receivers can consume different messages concurrently.**

Multiple receivers of a Point-to-Point Channel are ***Competing Consumers*****. They can even run on multiple computers (e.g. for load-balancing).**

For **RPC a pair** of Point-to-Point Channels is required (***Request-Reply*** pattern, where the call is a ***Command Message*** and the reply is a ***Document Message***).

Point-to-Point Channel pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: in JMS a Point-to-Point Channel is a Queue

Sending a message:

```
Queue queue = // obtain the queue via JNDI
QueueConnectionFactory factory = // obtain the connection factory via JNDI
QueueConnection connection = factory.createQueueConnection();
QueueSession session = connection.createQueueSession(true, Session.AUTO_ACKNOWLEDGE);
QueueSender sender = session.createSender(queue);

Message message = session.createTextMessage("The contents of the message.");

sender.send(message);
```

Point-to-Point Channel pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Receiving a message:

```
Queue queue = // obtain the queue via JNDI
QueueConnectionFactory factory = // obtain the connection factory via JNDI
QueueConnection connection = factory.createQueueConnection();
QueueSession session = connection.createQueueSession(true, Session.AUTO_ACKNOWLEDGE);
QueueReceiver receiver = session.createReceiver(queue);

TextMessage message = (TextMessage) receiver.receive();
String contents = message.getText();
```

Note: JMS 1.1 **unifies** a Point-to-Point Channel and a Publish-Subscribe Channel, applying *Destination, ConnectionFactory, Connection, Session, MessageProducer* and *MessageConsumer*.

Point-to-Point Channel pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: in .NET a Point-to-Point Channel is a MessageQueue (MSMQ). Prior to version 3.0 only Point-to-Point messaging was supported.

Sending a message:

```
MessageQueue queue = new MessageQueue("MyQueue");
queue.Send("The contents of the message.");
```

Receiving a message:

```
MessageQueue queue = new MessageQueue("MyQueue");
Message message = queue.Receive();
String contents = (String) message.Body();
```

Related patterns: Command Message, Competing Consumers, Document Message, Message, Message Channel, Messaging, Publish-Subscribe Channel, Request-Reply

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Publish-Subscribe Channel pattern

Publish-Subscribe Channel pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: an application uses messaging to announce an event

<u>Problem:</u> How can the sender broadcast an event to all interested receivers?

Event notification is wrapped into a message and is sent to a Message Channel.

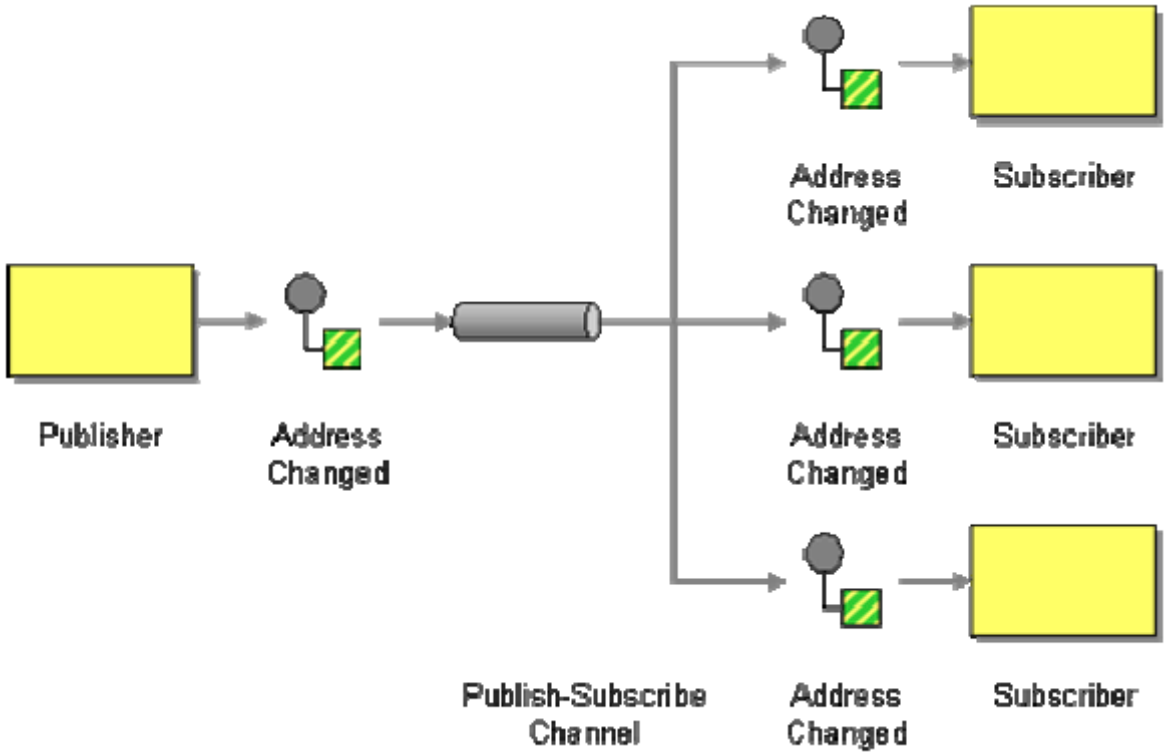
Requirements:

- **all receivers** have to be notified, but **only once**
- the message **can not be consumed till all the receivers** do not receive that
- when all the receivers got the message, it **has to disappear** from the channel (its coordination by the receivers would cause coupling)
- concurrent consumers **should not compete but share** the message

<u>Solution:</u> Send the event on a *Publish-Subscribe Channel,* which delivers a copy of a particular event to each receiver.

Publish-Subscribe Channel pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



A Publish-Subscribe Channel can be imagined as it **splits its one input** channel **into multiple output** channels (one for each subscriber) and an input message is **copied** to each output channel.

Publish-Subscribe Channel pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A Publish-Subscribe Channel can **easily used for debugging**: additional receiver can be added without disturbing the others. So, monitoring the messaging is easy.

However, additional subscribers without control **can be harmful**, too. In these cases **security policies for restricting subscribers** are needed. Moreover, **logging active subscribers** also can be useful.

While publishers **always send the message to a specific channel**, in some systems **subscribers can subscribe to multiple channels** at once, applying **wildcards (e.g. MyCorp/Prod/OrderProcessing/* : both NewOrders and CancelledOrders)**

Event Messages are usually sent to a **Publish-Subscribe Channel**.

There can be ***Durable Subscribers*** and ***Non-Durable Subscribers***.

If the event notification has to be acknowledged, ***Request-Reply*** pattern should be applied.

Publish-Subscribe Channel pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: in JMS a Publish-Subscribe Channel is a Topic.

Sending a message:

```
Topic topic = // obtain the topic via JNDI
TopicConnectionFactory factory = // obtain the connection factory via JNDI
TopicConnection connection = factory.createTopicConnection();
TopicSession session = connection.createTopicSession(true, Session.AUTO_ACKNOWLEDGE);
TopicPublisher publisher = session.createPublisher(topic);

Message message = session.createTextMessage("The contents of the message.");
publisher.publish(message);
```

Publish-Subscribe Channel pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Receiving a message:

```
Topic topic = // obtain the topic via JNDI
TopicConnectionFactory factory = // obtain the connection factory via JNDI
TopicConnection connection = factory.createTopicConnection();
TopicSession session = connection.createTopicSession(true, Session.AUTO_ACKNOWLEDGE);
TopicSubscriber subscriber = session.createSubscriber(topic);

TextMessage message = (TextMessage) subscriber.receive();
String contents = message.getText();
```

Note: JMS 1.1 **unifies** a Point-to-Point Channel and a Publish-Subscribe Channel, applying *Destination, ConnectionFactory, Connection, Session, MessageProducer* and *MessageConsumer*.

Publish-Subscribe Channel pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: .NET supports one-to-many messaging since MSMQ 3.0 **(it can be accessed through the COM interface)**

Its two approaches are:

- **Real-Time Messaging Multicast**: similar to Publish-Subscribe Channel but depends on **IP multicasting with Pragmatic General Multicast** (PGM) protocol

- **Distribution Lists and Multiple-Element Format Names**: enables message sending to a **list of receivers** + creates **a symbolic channel specifier that dynamically maps to multiple real channels**.

Related patterns: Durable Subscriber, Event Message, Message, Message Channel, Messaging, Point-to-Point Channel, Request-Reply

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Datatype Channel pattern

Datatype Channel pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: an application uses messaging to transfer different types of data.

<u>Problem:</u> How can the application send a data item in a way that the receiver will know how to process it?

For the messaging system each message has the same type (e.g. a sequence of bytes). However, **the receiver has to know the message content's data structure and format** for processing that (we will also call it „message type").

Structure can be e.g. character array, byte array, XML document, serialized object, etc. Format can be e.g. the record structure of the bytes or characters, the class of the serialized object, the DTD of the XML document.

Messages of different types usually have to be processed differently. If they arrive on the same message channel, the receiver will be confused.

Datatype Channel pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The **sender knows what type of data** it sends.

How to tell it to the receiver?

- a **flag in the header** of the message (***Format Indicator***) **/receiver needs a case statement/**
- wrap the data into a ***Command Message*** where command tells what to do **/message only need to transfer the data and not to control the receiver/**

If we put **different type** of messages onto the same channel, we **can not handle** them **in the same way** **(like iterators in case of homogeneous collections vs. heterogeneous collections).**

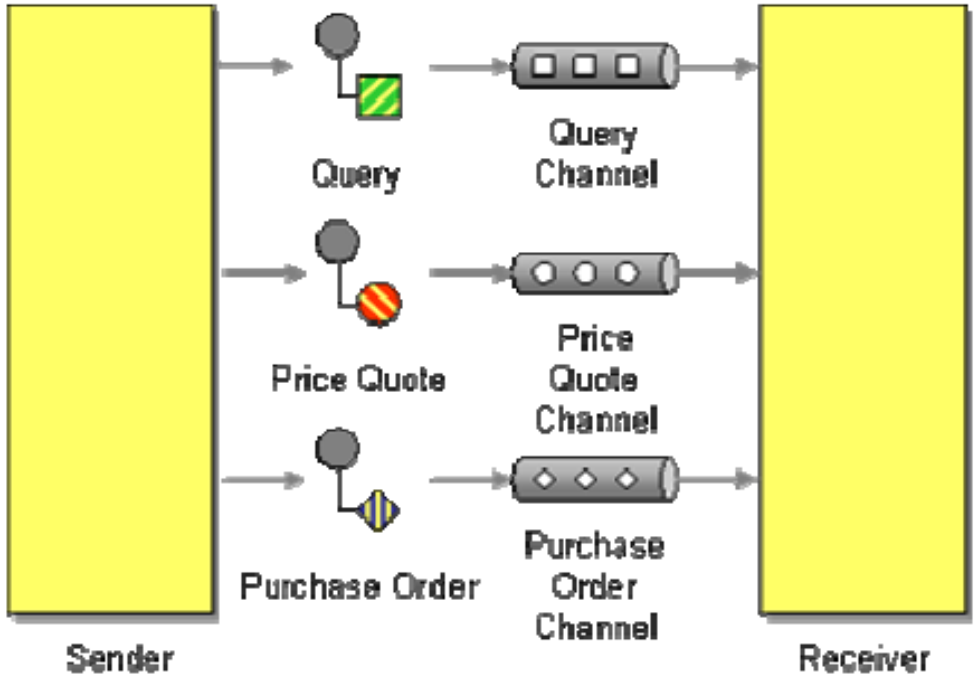
<u>Solution:</u> Use a separate *Datatype Channel* for each data type, so that all data on a particular channel can be of the same type.

Datatype Channel pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The sender knows **what type of data** he sends → **selects the channel**

The receiver knows **which channel** he consumed the message from → knows **the type of the received data**



Since a channel is not free (but cheap) in some cases it would be **too expensive to establish one separate channel for each message type**. In these cases multiple types of data use the same channel and different ***Selective Consumer*** has to be applied for each type.

Datatype Channel pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Canonical Data Model defines a unified data model so **that all messages on all channels** can follow in an enterprise.

A ***Message Dispatcher*** can be applied to **receive messages based on the general type** of the messages and **dispatch them based on their more specific type** to the appropriate receiver.

Related patterns: Canonical Data Model, Command Message, Format Indicator, Message Channel, Message Dispatcher, Selective Consumer, Messaging

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Invalid Message Channel pattern

Invalid Message Channel pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: an application uses messaging to receive messages.

Problem: How can a messaging receiver gracefully handle receiving a message that makes no sense?

In some cases the **receiver can not process** the consumed message. Source of the problem can be, e.g.:

- The message **header misses required elements** **(e.g. return address)** or it has **fields that make no sense**.
- The message **body** may cause **parsing errors**, it may have **lexical or validation errors** **(e.g. an XML document is not valid for its DTD).**
- The sender may send a valid message onto a **wrong channel**.
- A **malicious sender** sends **invalid messages**.

Invalid Message Channel pattern II.

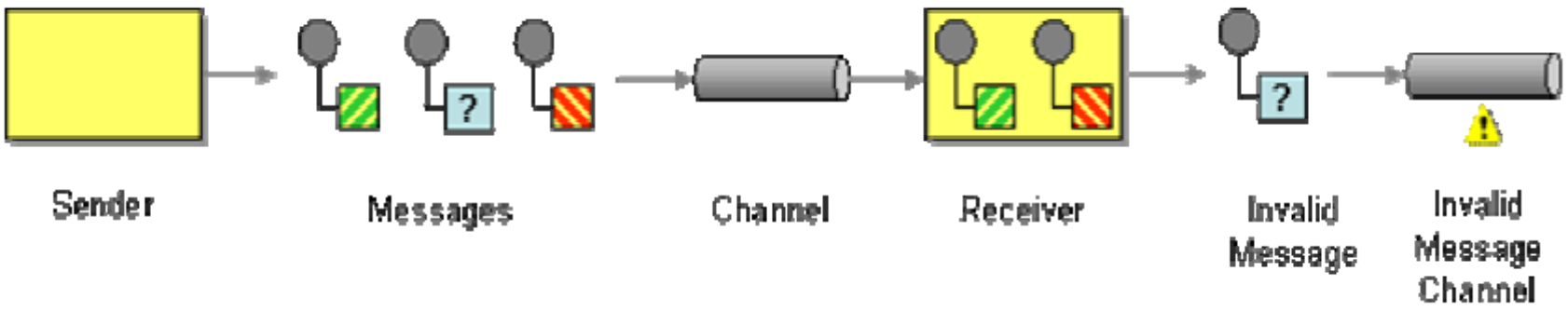
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

What to do when the **receiver recognizes** that the received **message is not valid**?

- **put the message back** onto the channel **/the message can be reconsumed; moreover, the channel can be flooded and the preformance decreases/**

- the receiver can **throw away** the message **/it would hide the problem/**

<u>Solution:</u> The receiver should move the improper message to an *Invalid Message Channel,* a special channel for messages that could not be processed by their receivers.



Invalid Message Channel pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The **messaging system administrator has to define** one or more Invalid Message Channels for the applications.

Since this channel contains only invalid messages, **its flooding is not a problem**.

An **error handler** can use a receiver on this channel.

This channel acts **like and error log** for the messaging. If **the application** – which puts the invalid message on the channel – wants, it **can log the error with more details**.

Whether to put the message on the Invalid Message Channel or not: **it is the decision of the receiver**.

Invalid Message Channel pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

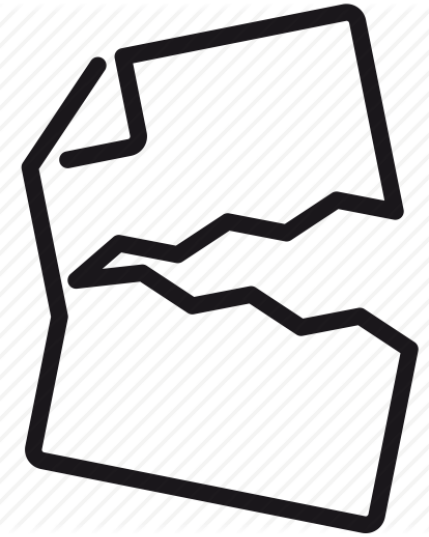
A message **that is valid (does not have the intended structure) for one receiver should be valid for all the other receivers** of the same channel. A message that is invalid for one receiver should be invalid for all the other receivers of the same channel.

The situation differs **for valid messages with semantically incorrect content**:
(e.g. a message that initiates the receiver to delete a non-existing database record)
It is **not a messaging error but an application error**. **(The message itself is valid but the application request is invalid.)**
To differentiate message-processing errors and application errors the receiver should be implemented as a ***Service Activator*** **or** a ***Messaging Gateway***. They separate message-processing code from the rest of the application.
So, **invalid messages** are routed onto the **Invalid Message Channel**, **but invalid data from valid message** results in an **application error** (that does not belong to messaging) – however, **they can also be put onto the Invalid Message Channel** by the application.

Invalid Message Channel pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

It is useful to **monitor** the Invalid Message Channel **/correction is hard to be automatized/** and **alert the system administrator** when it has content **(means often coding or configuration error)**.



Related patterns: Command Message, Correlation Identifier, Datatype Channel, Dead Letter Channel, Message, Message Channel, Message Sequence, Messaging, Service Activator, Messaging Gateway, Return Address

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Dead Letter Channel pattern

Dead Letter Channel pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: an enterprise is using messaging to integrate applications.

Problem: What will the messaging system do with a message it cannot deliver?

Possible reasons for the system being not able to deliver the message:

- the **channel is not configured** properly
- before receiving the message the **channel is deleted**
- the message **expired** before delivery
- a message without expiration **timed out** **(no delivery for a very long time)**
- **none of the Selective Consumers** intends to consume the message
- a **wrong header** of the message prevents the successful delivery

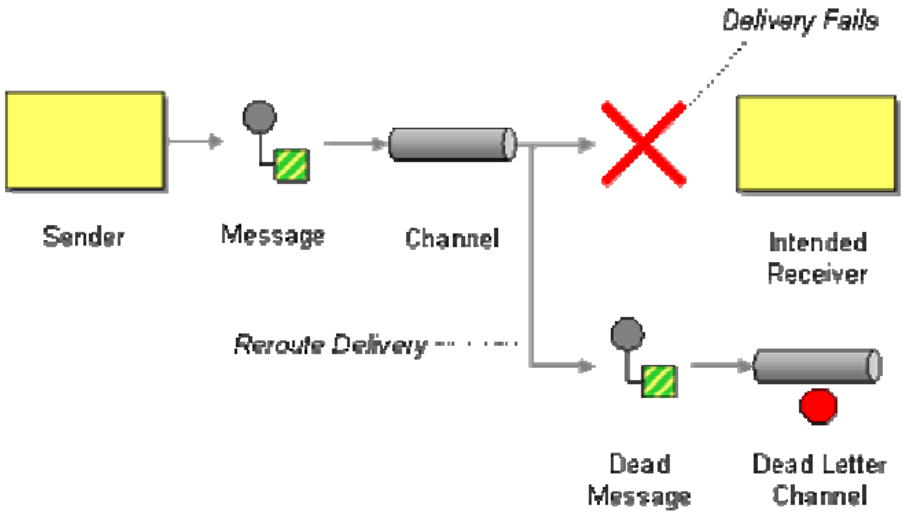
Dead Letter Channel pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

What a messaging system can do if it turns out that the message can not be delivered?

- leaves the message on the channel **(it clutters up the system)**
- can send back the message to the sender **(but the sender is not reveiver)**
- deletes the message **(but the sender thinks that the delivery was successful)**

<u>Solution:</u> When a messaging system determines that it cannot or should not deliver a message, it may elect to move the message to a *Dead Letter Channel.*



Dead Letter Channel pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Each machine which has an installed messaging system has its **own local Dead Letter Channel**. From the local channel the dead message can be moved to another queue **(usually recording which machine the message died on and the original channel of the message)**.

Dead message	Invalid message
Can not be delivered successfully	Can be delivered successfully but **can not be processed**
Noticed by the **messaging system** **(based on the header of the message)**	Noticed by the **receiver** **(based on the header or the body of the message)**

Dead message handling can be **offered by the messaging system** but can be **delevoped by the developer**, too.

It is useful to **monitor** the Dead Letter Channel as well.

Related patterns: Invalid Message Channel, Message Expiration, Selective Consumer, Messaging

Guaranteed Delivery pattern

Guaranteed Delivery pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: an enterprise is using messaging to integrate applications.

Problem: How can the sender make sure that a message will be delivered, even if the messaging system fails?

Because of the „**store and forward**” method of the asynchronous messaging, the temporary unavailability of the network or the receiver is not a problem. Till one of them is not available, the messaging system stores the message – **by default in the memory**.

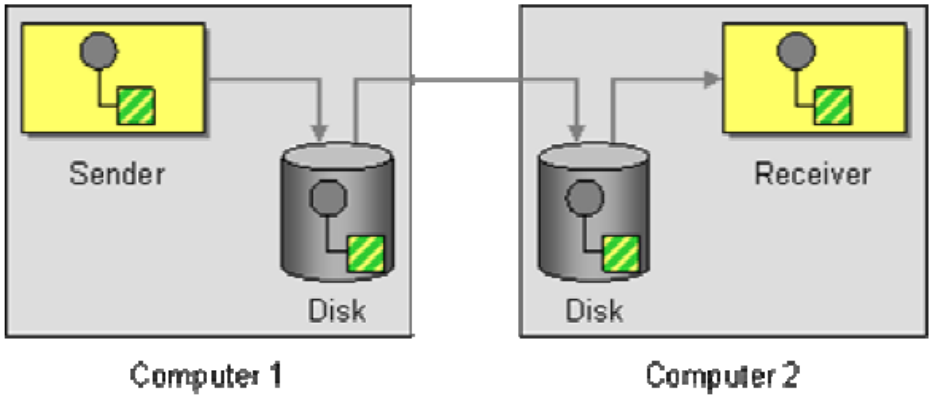
Storing happens till a successful delivery to the next storage point. However, **in case of a crash** of the messaging system **(loss of power, unexpected abort of the process, etc.)** the **content of the memory is lost**.

If an application is **sensitive for memory loss**, it usually **saves its data onto the disk (into a file or a database)**. Something similar is needed for messaging with Guaranteed Delivery.

Guaranteed Delivery pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution:</u> Use the *Guaranteed Delivery* to make messages persistent so that they cannot be lost even if the messaging system crashes.



Each computer which has an installed messaging system has its own, **local built-in store**. In case of Guaranteed Delivery the message is **stored here until it is sent successfully to the next storage point**. At least one storage exists where the message is stored till a **successful delivery and till an acknowledgement** from the receiver.

Guaranteed Delivery pattern III.

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Persistence is **more reliable** but **more expensive** **(it may require large amount of disk space)** with **lower performance** **(message delivery becomes slower)**.

Since a sender with 1000s of messages to send through an unavailable network may require huge amount of disk space**, in some messaging systems *retry timeout*** can be set – defining **how long to store messages in the messaging system**.
(lots of systems with high traffic use Event Messages, on which Message Expiration can be set practically)

It is adviced **not to use Guaranteed Delivery during tests and debugging**. Otherwise, **stored messages could disturb** the purging of the channels and restart. **(Some messaging systems make it possible to purge the message channels individually during testing.)**

Of course, **Guaranteed Delivery is not 100% (disk failures, out of disk space, etc.).**

Guaranteed Delivery pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Implementation-specific Guaranteed Delivery:

- in **.NET** MSMQ: the **sender** has to be a **Transactional Client** for Guaranteed Delivery
- in **JMS Publish-Subscribe Channels** ensure Guaranteed Delivery **only for active** subscriber **or to *Durable Subscribers***. So, if a subscriber intends to take part in Guaranteed Delivery even in its inactive state, it has to be Durable Subscriber.

If there are **subscribers for the same data with different requirements**: some of them want Guaranteed Delivery and others do not want, then **two different channels should be created**: one with Guaranteed Delivery and one without. The sender should **send the messages to both channels, less frequently to the persistent channel (because of its overhead)**.

Guaranteed Delivery pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: persistent messages by JMS

JMS makes it possible to **set the persistence message-by-message**
(it means that there can be both persistent and non-persistent messages on the same channel):

```
Session session = // obtain the session
Destination destination = // obtain the destination
Message message = // create the message
MessageProducer producer = session.createProducer(destination);
producer.send(
        message,
        javax.jms.DeliveryMode.PERSISTENT,
        javax.jms.Message.DEFAULT_PRIORITY,
        javax.jms.Message.DEFAULT_TIME_TO_LIVE);
```

Guaranteed Delivery pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Although, JMS makes it possible to **set all messages of an application to be persistent**:

```
producer.setDeliveryMode(javax.jms.DeliveryMode.PERSISTENT);
```

Then all message of this producer will be sent in a persistent way:

```
producer.send(message);
```

Example: persistent messages by .NET

In .NET persistent messaging is possible when MessageQueue is created to be transactional.

```
MessageQueue.Create("MyQueue", true);
```

Guaranteed Delivery pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: persistent messages by IBM WebSphere MQ

In WebSphere MQ persistence can be set to both channels **(at the time of their creation)** and messages.
If the channel is non-persistent, the messages can not be persistent.
If the channel is persistent, it can be configured in two ways:

(1) all of its messages are persistent
(2) persistence can be set **for each message separately**

(1):
```
DEFINE Q(myQueue) PER(PERS)
```

(2):
```
DEFINE Q(myQueue) PER(APP)
```

In this latter case the intention of the sender related to persistence can be set through JMS MessageProducer as described in the previous example.

Related patterns: Durable Subscriber, Event Message, Message Expiration, Messaging, Point-to-Point Channel, Publish-Subscribe Channel, Transactional Client

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Channel Adapter pattern

Channel Adapter pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: many enterprises use messaging to integrate their heterogeneous applications.

Problem: How can you connect an application to the messaging system so that it can send and receive messages?

Several applications were **developed not to take part in messaging**, like standalone (one-in-all) applications.

Some **messaging vendors** developed **special messaging API**, so an application developer should have prepared the **application ready for different messaging APIs.**

If an application requires **general interface** for data interchange with multiple applications, it usually applies **file or database**. Moreover, applications often **save their data** into a database or file, so **it can be easily extended** to store the data **for data exchange**, too, this way and offered to the communication partner.

Channel Adapter pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Moreover, applications may **offer their functions through an API**, that can be reached e.g. by messaging.

Some applications can communicate through **TCP/IP** or **HTTP**. **(it is less reliable than messaging and sometimes the data format is specific and not compatibe with common messaging solutions)**

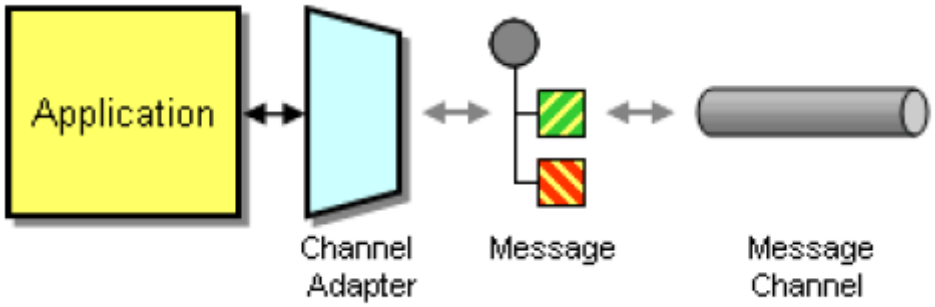
If **custom applications** have to fit to messaging, **additional code** should be insterted into them:

- it has to be done carefully **not to disturb the existing functionality**
- it requires experts who **know both the application and the messaging system**
- it is usually **not an option in case of 3rd party applications (the code may not be changed)**

Channel Adapter pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution:</u> Use a *Channel Adapter* that can access the application's API or data and publish messages on a channel based on this data, and that likewise can receive messages and invoke functionality inside the application.

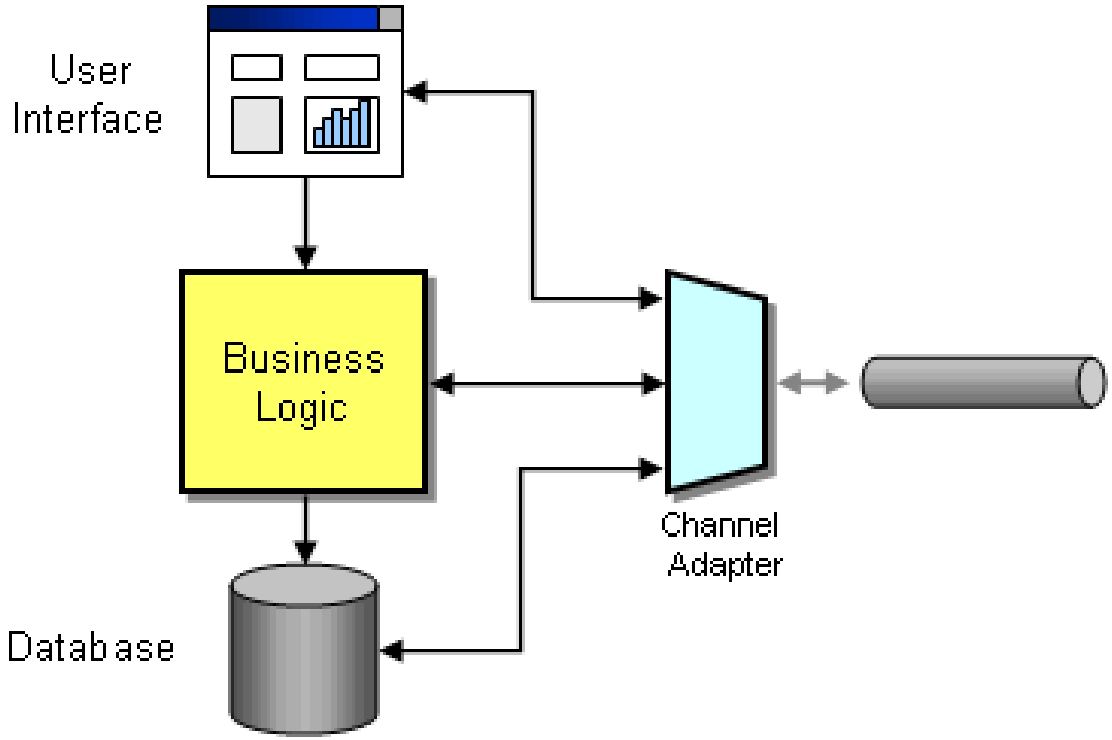


The **Channel Adapter** behaves as a **messaging client to the messaging system** and **reaches the application's functionality via the interface** of the application. This way **any application can be connected** to the messaging system.

Channel Adapter pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Channel Adapters can **connect to different layers** of the application:



Channel Adapter pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

a. User Interface Adapters:

- it is often applied if **the platform of the application does not support the messaging system** or the **application does not support integration**.

- it **does not require access to the** apllication's **internal functions.**

- in case of a **HTML-based user interface** it is easy to **start a HTTP request** and **parse the result**.

- **rigid** and **slow** solution (parsing)

- in case of **UI change** the **Channel Adapter** also has to be changed

Channel Adapter pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

b. Business Logic Adapters:

- Some aplications have their **well defined interface** exposing the **core functions**
(components like EJBs, COM objects, CORBA components or direct programming API like C++, C# or JAVA libraries)

- They are **created for access** by other applications

- They are **stable** and their use is **efficient**. If an application has a well defined API **it is suggested to connect to that**.

Channel Adapter pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

c. Database Adapters:

- Several applications persist their data in a **relational database**.

- Channel Adapter could **read/write directly** these databases without disturbing the application. **/e.g., by inserting a trigger into a database table that sends a message every time when a record changes/**

- it is an **efficient** and **universal/generic** solution **/only some database vendors dominate/**

- **Drawbacks** are: we dig in the **deep internals**: writing the data directly in the database is **dangerous** + if the application developers **change the (often unpublished) database schema** that influences the Channel Adapter, too.

Channel Adapter pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Limitations of Channel Adapters:

- They usually **require messages formatted suitable to the applications' details (e.g. data field names)** – so they can not be applied generally.
For improving it, **Message Translator is often added** that translates the application-specific message to a message with canonical data format.

- Their **location (computer) often differs** from that of the application's.
Applied **HTTP, ODBC or other protocols** for brigding the gap **does not provide the same QoS** than messaging.

- They are often **unidirectional**: e.g., they can invoke applications' functions but not to be notified about data changes in the application.

Channel Adapter pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Special variations of Channel Adapters:

- **Metadata Adapters (Design-Time Adapters)**: they extract **metadata about the application**, e.g. the internal data structure **(like system tables /that contain description of the tables/ of a database).**
It can be used to **adjust the Message Translator** or to **detect changes in data representation**, for example.

- **Messaging Bridge**: **connects messaging systems**. In this case Channel Adapter is **implemented as a Transactional Client**, ensuring that every piece of work succeeds in all systems.

Related patterns: Canonical Data Model, Message Channel, Message Translator, Messaging, Messaging Bridge, Transactional Client

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Messaging Bridge pattern

Messaging Bridge pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: an enterprise uses more than one messaging system

<u>Problem:</u> How can multiple messaging systems be connected so that messages available on one can also be available on the others?

The possible **reason** of using more than one messaging system:

- divisions **standardized around different technologies**
- **different companies** that use different messaging systems
- an application, which **takes part in multiple companies** with different messaging systems
- a **too big company that separates** its message channels and endpoints into more than one messaging system

Messaging Bridge pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Messaging Bridge needed if two messaging system intend to communicate with each other.

Unfortunately, a **standardized messaging API** – like JMS – **is not enough**:

- the **same message format** is needed
- usually a Message Store of one vendor can work together only with another **Message Store from the same vendor**.

Possible solutions:

- each application creates a **client for each messaging system**

(complex, redundant, one more messaging system influences every application, etc.)

- each application is **bounded only to one messaging system**

(not a robust solution)

Messaging Bridge pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution:</u> Use a *Messaging Bridge,* a connection between messaging systems, to replicate messages between systems.

„Connecting two complete messaging systems" means in practice to **connect sets of channels of different messaging systems**.
So, Messaging Bridge is **a set of Channel Adapters**: each pair of adapters connects a pair of corresponding channels.

Messaging Bridge: **a map from one set of channels to the other while transforming the message format**.
When a message is produced onto a channel of interest the Message Bridge consumes that and puts a message with the same content onto the corresponding channel of the other messaging system.

Messaging Bridge pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Many **messaging vendors have their Messaging Bridges** to other vendors.

Examples:

MSMQ Bridges

In **MSMQ connector servers** are applied to send messages to non-MSMQ systems

There is an **MSMQ-MQSeries Bridge** in the **Microsoft's Host Integration Server** .

Envoy Connect connects **MSMQ** to (non-Windows based) **BizTalk** servers.
It makes **.NET** based **and J2EE** based communication bind.

SonicMQ Bridges

SonicMQ Bridges support **IBM MQSeries, TIBCO TIB/Rendezvous and JMS**.

Related patterns: Channel Adapter, Message Channel, Message Endpoint, Messaging

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message Bus pattern

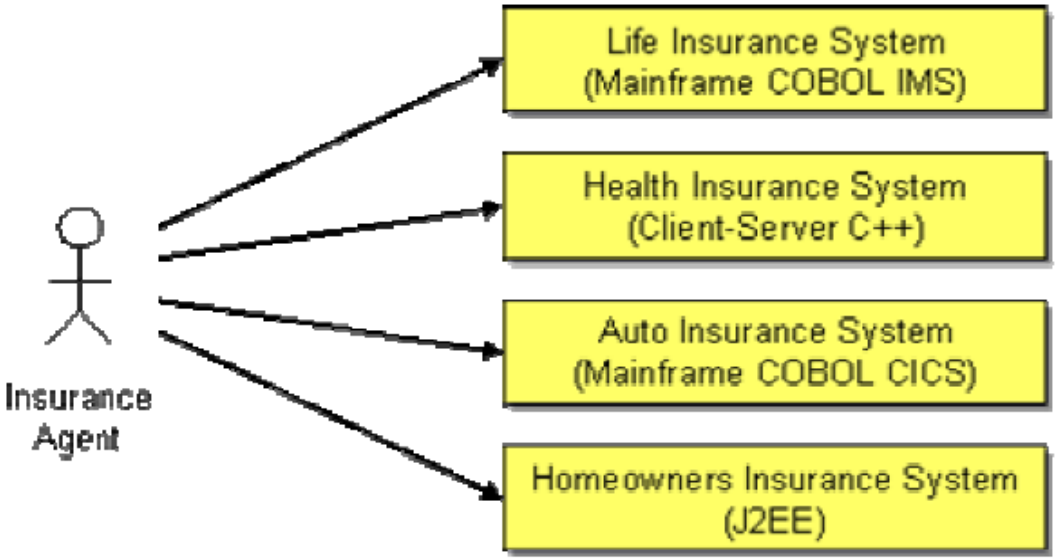
Message Bus pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: an enterprise has several systems that should share data and operate together in a unified way.

<u>Problem:</u> What is an architecture that enables separate applications to work together, but in a decoupled fashion such that applications can be easily added or removed without affecting the others?

An **example** for EAI, where an agent has to log in every application, separately.



Message Bus pattern II.

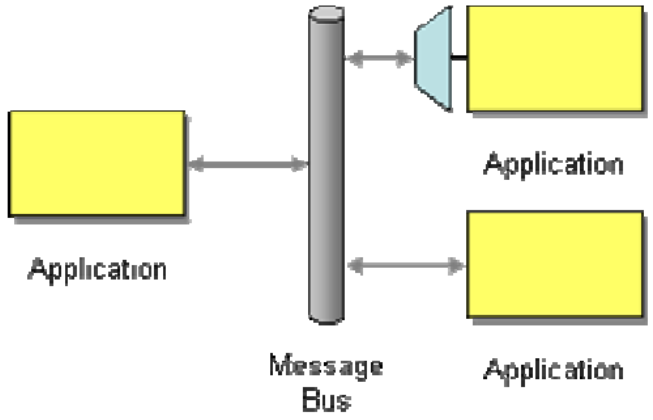
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Possible solutions for the problem, where some agents need to work together with some different systems:

- **unify/rewrite all** the applications **(tremendous work)**
- **develop a unified application** **(creates a newer, not integrated application)**

Moreover, what happens if applications **change, disappear, become unavailable or new applications** are involved? → **loosely coupled** solution is needed.

<u>Solution:</u> Structure the connecting middleware between these applications as a *Message Bus* that enables them to work together using messaging.



Message Bus pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A Message Bus is a combination of a **common data model, a common command set and a messaging infrastructure. /just like a communication bus of a computer/**

Its **components** are:

- **Common Communication Infrastructure**: typically a **messaging system** that provides a cross-platform and cross-language solution **/including especially Message Routers and Publish-Subscribe Channels/. (like e.g. pins and wires of a PCI-bus)**

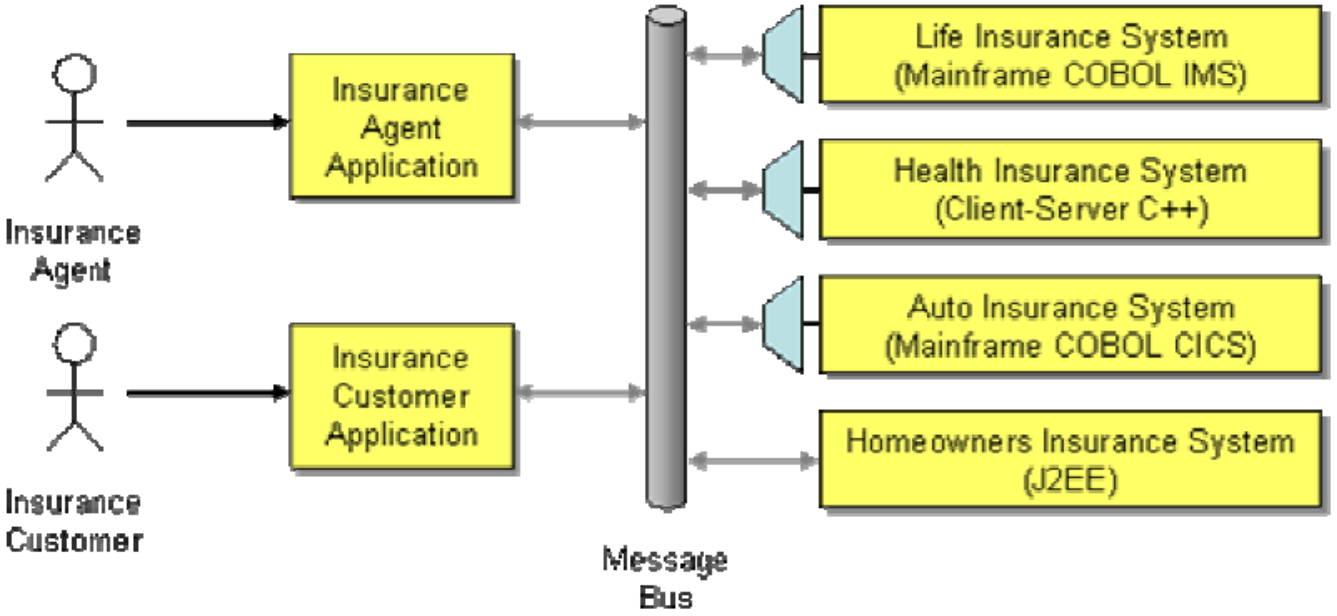
- **Adapters**: **Channel Adapters** and **Service Activators**. They also require the existance of a **Canonical Data Model**.

- **Common Command Structure**: **set of commands that are understood by all** participants of the Message Bus. **/like Command Messages/**

Message Bus pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The previous example with Message Bus:



The two GUIs know only about the Message Bus. The GUI originated **command messages** can be handled by:

- An **adapter** that interprets the command and communicates with the system
- the **command processing logic that is built into** the application

Message Bus pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

As soon an a **Message Bus** is created **for one GUI**, that can be **used by another ones**, easily. Maybe the security aspects differ, but **their work with the backend is the same.**

A **Message Bus realizes a SOA** for the enterprise:

- each service has **a request channel for the well formatted messages** and optionally a reply channel.
- the set of the request channels **acts as a directory** of the available services

Applications on a Message Bus:

- have to use the **same Canonical Data Model**.
- **may depend on the Message Routers**
- may require a **Channel Adapter/Service Activator**

Related patterns: Canonical Data Model, Channel Adapter, Command Message, Datatype Channel, Message Router, Service Activator, Publish-Subscribe Channel

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

THANK YOU FOR THE ATTENTION!

Reference:

Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns – Designing, Building and Deploying Messaging Solutions, Addison Wesley, 2003, ISBN 0321200683

www.enterpriseintegrationpatterns.com

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

ENTERPRISE INTEGRATION PATTERNS

9-10. Message

Author: Tibor Dulai

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

Basics of messages I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

When two **applications wish to exchange a piece of data**, they do that by **wrap**ping it in a message.
A **Message Channel cannot transmit raw data**, it can transmit the data **wrapped in a message**.

Intention of sending a message:

1. **Command message**: the sender wants to **invoke a function/method** (that has to be specified, what code to run)
2. **Document message**: sender **transmits a data structure** to the receiver (not specifying what to do with it)
3. **Event message**: **notification** of the receiver **about a change** in the sender (not specifying how to react)

Basics of messages II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Returning a response (Request-Reply scenario):

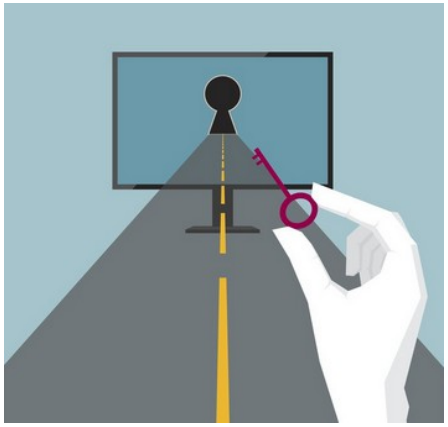
- The **request** is usually a **Command Message**
- The **reply** message is usually a **Document Message** containing the result or an exception
- The requestor should specify a **Return Address** in the request specifying the **channel** for the reply
- A **Correlation Identifyer** is needed for **pairing the reply with the corresponding request** (because the requestor may have multiple request)

The two common Request-Reply scenarios

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

1. **Messaging RPC**:

   The requestor invokes a function and needs the result value.



2. **Messaging Query**:

   The requestor performs a query that the replier executes and returns the result.

Both of them uses a Command Message request and a Document Message reply.

Other Message-related concepts

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message sequence

If the amount of data to be sent is quite a big, it **has to be broken into chunks**. It is important to send the chunks in a **sequence and not as a bunch** of chunks: the receiver has to reconstruct the original data.

Message expiration

If the content of the message is time-sensitive, Message Expiration can be applied.

- If the messaging system does not deliver the message by the expiration, it should discard the message.

- If the receiver gets a message after its expiration, it has to discard the message.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Command Message pattern

Command Message pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: an application needs to invoke functionality provided by an other application (Remote Procedure Invocation via Messaging)

Problem: How can messaging be used to invoke a procedure in another application?

RPI is a **synchronous** process: **immediate** call, blocked caller, but does not work if the **network is down** or the remote procedure is **not available**.

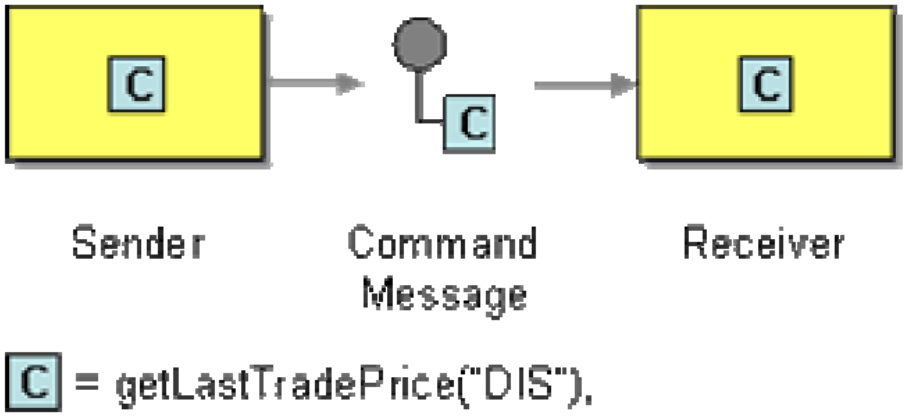
Messaging is **asynchronous**: **the procedure does not have to be remotely available**; it can be handled like a local procedure in its own process.

The request can be wrapped into a command message and put onto a message channel. Anyway, the **command's state can be stored in the message's state**.

Solution: Use a Command Message to reliably invoke a procedure in another application.

Command Message pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



There is no specific message type for commands: **Command Message is a simple message that contains a command**.

JMS examples: an **ObjectMessage** that contains a Serializable command object
a **TextMessage** that contains a command in XML form

In **.NET** a Command Message is a **Message** that stores a command.
A **SOAP request** is also a Command Message.

Command Messages are usually sent onto a **Point-to-Point Channel**, so they are consumed exactly once.

Command Message pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: SOAP and WSDL (Web Service Description Language)

In case of RCP-style SOAP message, the request is a Command Message. The body of the message has to contain **the name of the method** to invoke and the **parameter values** to pass. The method name must be the same as one of the message names defined in the receiver's WSDL.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m="Some-URI">
         <symbol>DIS</symbol>
      </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Command Message pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: SOAP and WSDL (Web Service Description Language)

cont.

In a SOAP command the method name should be some standard <method> element:

the method name is the name of the method element, prefixed by the *m* namespace.

Having a separate XML element type for each method makes XML data validation easier, so the method element type can specify the parameters' names, types, and order.

Related patterns: Remote Procedure Invocation, Message, Message Channel, Messaging, Point-to-Point Channel

Document Message pattern

Document Message pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: an application intends to transfer data to another application by messaging.

Problem: How can messaging be used to transfer data between applications?

If one application has data that another application needs, there are **several ways** to transfer that:

- **File Transfer**: **easy** to use, but the file may sit unused **for a while** before reading it; moreover who has the **responsibility** of deleting it in case of more than one receivers?

- **Shared Database**: new data requires **new/modified schema**; **access-control might be** needed (different parts of the database may differ in the sensibility of the data); **trigger of the receiver** to read the new data is difficult; determination of **who has to delete** the data may also be a complex task.

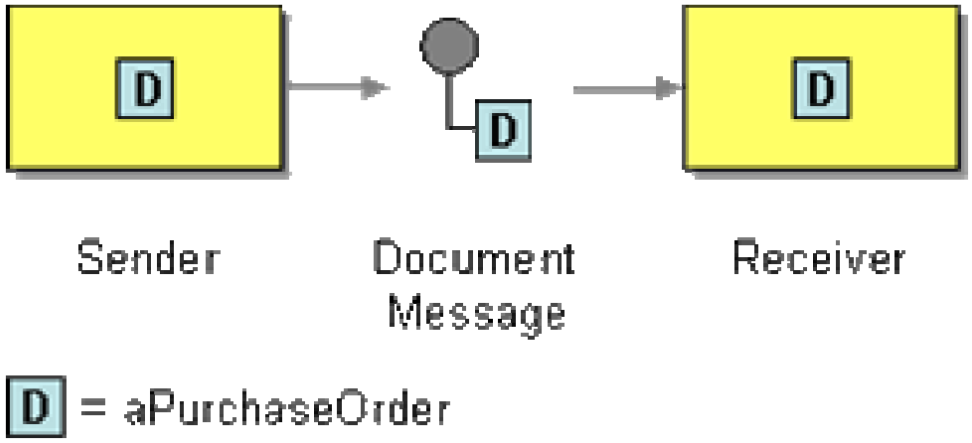
Document Message pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- **Remote Procedure Invocation:** it does not only send the data but tells the receiver **what to do with the data** (like Command Message)

- **Messaging**: more **reliable** than RPC. If we apply a **Point-to-Point Channel**, we **avoid duplication**. If we apply a **Publish-Subscribe Channel**, we can be sure that **every receiver gets** the data who intends to.

<u>Solution:</u> Use a *Document Message* to reliably transfer a data structure between applications.



Document Message pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The data (single object or a decomposable data structure) is just passed, **no certain behaviour of the receiver is determined**. (not like in the case of a Command Message)

Differs from Event Message in content and timing aspects: **only the successful transfer of the content**: the data **is important** (less important is when it happens). **Guaranteed Delivery** can be considered, **Message Expiration** not so much.

There is no specific message type for a Document Message.

JMS examples: an **ObjectMessage** that contains a Serializable data object
a **TextMessage** that contains data in XML form

In **.NET** a Document Message is a **Message** that stores the data.
A **SOAP reply** is also a Document Message.

Document Message pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Document Messages are **usually sent onto Point-to-Point Channel** (guaranteeing that ***no duplication*** happens). A popular screenplay is to implement a **simple workflow** by messaging: one application gets a document, modifies that and sends forward to other application.

If we realize broadcast of Document Messages by **Publish-Subscribe Channel**, we have to be **aware of multiple copies of the document that contain different data**. To **prevent** it, the document copies should be **read-only**.

In **Request-Reply** case, the **reply message** is usually a **Document Message** that contains the reply value as the document.

Document Message pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Java and XML

A simple XML document (purchase order) is sent as a message by JMS:

```
Session session = // Obtain the session
Destination dest = // Obtain the destination
MessageProducer sender = session.createProducer(dest);
String purchaseOrder =
"       <po id=\"48881\" submitted=\"2002-04-23\">
                <shipTo>
                        <company>Chocoholics</company>
                        <street>2112 North Street</street>
                        <city>Cary</city>
                        <state>NC</state>
                        <postalCode>27522</postalCode>
                </shipTo>
                <order>
                        <item sku=\"22211\" quantity=\"40\">
                                <description>Bunny, Dark Chocolate,
Large</description>
                        </item>
                </order>
        </po>";
```

```
TextMessage message = session.createTextMessage();
message.setText(purchaseOrder);
sender.send(message);
```

Document Message pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: SOAP and WSDL

A **document-style SOAP message** (that contains an XML document) or in case of an **RPC-style SOAP** message the **response message** (that contains the reply value as an XML document) is a Document Message.

Returning the answer from the called GetLastTradePrice method:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
      <m:GetLastTradePriceResponse xmlns:m="Some-URI">
         <Price>34.5</Price>
      </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Document Message pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Related patterns: Command Message, Remote Procedure Invocation, Event Message, File Transfer, Guaranteed Delivery, Message, Message Expiration, Messaging, Point-to-Point Channel, Publish-Subscribe Channel, Request-Reply, Shared Database

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Event Message pattern

Event Message pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: applications want to inform each other about some events for the sake of coordinated actions, by messaging.

Problem: How can messaging be used to transmit events from one application to another?

There are cases when **objects have to be notified about change in other object's state**.

e.g. - view has to be notified about the change in the model
- in B2B communication customers have to be notified about changes in of the prices or about a new catalog

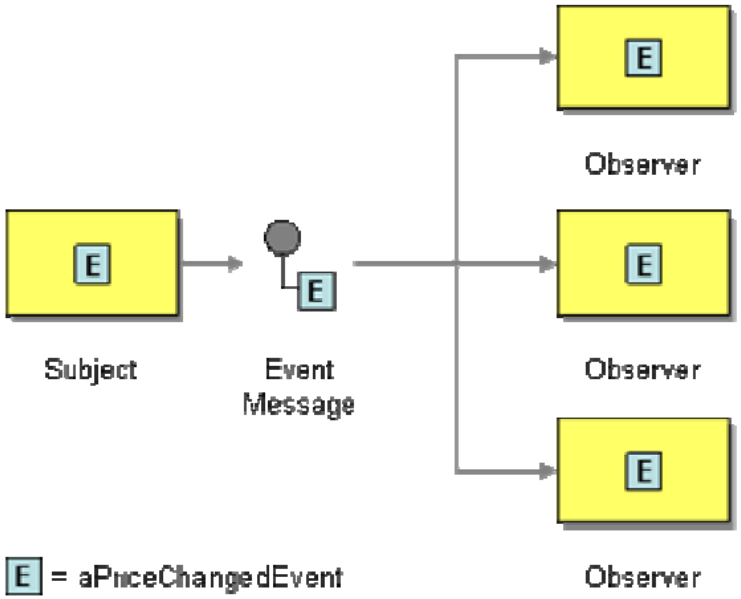
Also **RPC could** be used for event notification, **but RPC requires** the receiver to be **able to accept the event immediately**. Moreover, **the caller has to know all the listeners** and invoke an RPC on each of them.

If event notification is sent by a **message**, it can happen in **asynchronous** way.

Event Message pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution:</u> Use an *Event Message* for reliable, asynchronous event notification between applications.



The announcer creates the **event object**, wraps it into a **message**, puts that onto a **channel**; the receiver **receives** it, gets the event and **processes** that.
Messaging **does not change** the notification.

Event Message pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

There is no specific message type for an Event Message.

JMS examples: an **ObjectMessage** that contains the event as an object
a **TextMessage** that contains the event data

In **.NET** an Event Message is a **Message** that stores the event.

Differs from Document Message in content and timing aspects: **the timing is very important** (the message is often empty, only its occurence matters). The announcer should send the event notification as soon as the event happens, and the receiver should process that while it is relevant.

Guaranteed Delivery is not so useful (because of the frequent and quick transfer), but **Message Expiration** is very helpful (to verify whether the processing was quick or not).

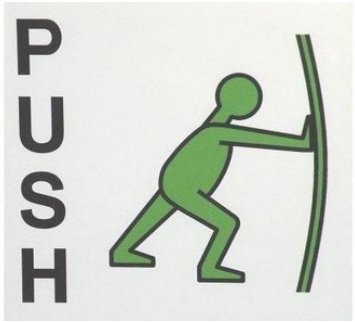
Event Message pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Examples:

- **Event Message**: a message that announces a new catalog and its URL
- **Document sent as an Event**: a message that announces and contains a new catalog

Push model	vs.	**Pull Model**
sends information about the change as part of the update		sends minimal information and that who needs more information can request it



Event Message pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Push model:

The message is a **combined Document/Event Message**. The **existence** of the message means that **state-change happened** and the **content** describes the new **state**. It is useful if **all the receivers need** the details.

Pull model:

It requires three messages:

- ***Update***: an **Event Message** that notifies about the state-change
- ***State Request***: a **Command Message** for asking the details of the event
- ***State Reply***: a **Document Message** that contains the details of the event

Advantages of the pull model: **only the interested requesters get only that part** of details that he is interested in. Anyway, the **Update is** usually **small**.

Disadvantage of the pull model: **3 messages** are needed.

Event Message pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

For Event Message **usually Publish-Subsribe Channel** is applied: there are usually **more than one application who is interested** in the event.

A **Document Message needs to be consumed**.

An **Event Message can often be ignored**
(when the receiver is too busy to process the message)

The subscribers can often be non-durable.

Related patterns: Command Message, Document Message, Durable Subscriber, Remote Procedure Invocation, Guaranteed Delivery, Message, Message Expiration, Messaging, Point-to-Point Channel, Publish-Subscribe Channel

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Request-Reply pattern

Request-Reply pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: Usual messaging is one-way communication. Sometimes applications need two way conversation.

Problem: When an application sends a message, how can it get a response from the receiver?

Messaging decouples the sender and the receiver: uses **asynchronous, reliable** communication and **one-way message channels**. So, the basic messaging ensures a one-way communication.

However, **two-way communication is often needed**:

- after a function call, we need the return value
- after query execution we intend to receive the query result
- event notifications are often followed by an acknowledgement

Request-Reply pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Possible ways of making messaging **to be two-way**:

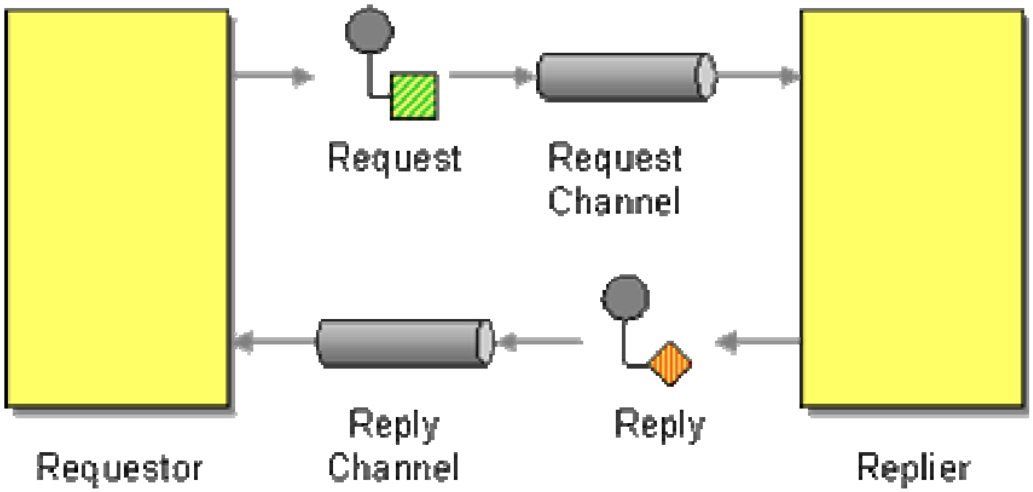
(a) If **a message could be shared** between applications and every application could modify the same message to inform the other application by that …. But it is **not possible** in messaging (one message can not be accessed by more parties at the same time).

(b) If a sender **keeps a reference** to the message, and **after the receiver inserts the answer into the message, the sender pulls that back** … But it is **not possible** in messaging (because of one-way message channels).

<u>Solution:</u> Send a pair of *Request-Reply* messages, each on its own channel.

Request-Reply pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



Two participants:

- **requestor:** sends the request and waits for the response
- **replier:** gets a request and responds

Request-Reply pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The **Request Channel** can be **either a Point-to-Point Channel or a Publish-Subscribe Channel** (depending on whether the request has to be broadcasted or not).

The **Reply Channel** is usually a **Point-to-Point Channel** (since the reply has to be sent to the requestor)

RPC always blocks the caller. In Request Reply **two approaches** exist:

1. **Synchronous Block**: a caller sends the request and **blocks** its process till the reply arrives **(as a Polling Consumer).**
   Advantage: **simple** to implement
   Disadvantages: if the caller crashes, **difficult to re-establish** the blocked thread. There can be **only one outstanding request**. **The reply channel is private** for the request thread.

Request-Reply pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

2. **Asynchronous Callback**: one thread of the caller **sends the request and sets the callback** for the reply. **Another thread is listening** for the reply. When the reply arrives, the **reply thread invokes the callback**, which re-establishes the caller's context and processes the reply.

Advantages: **multiple requests** can **share a single reply channel**. **One reply thread** can handle replies **for multiple request threads**. **In case of crash** of the requestor only **the reply thread has to be restarted**.

Disadvantage: more **complex**: the callback mechanism has to re-establish the requestor's context.

Request-Reply pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

What **Request and Reply** messages can **represent**:

- **Messaging RPC**: RPC by messaging. The **request** message is a **Command Message** (contains the function to invoke), the **reply** message is a **Document Message** (contains the return value or exception)

- **Messaging Query**: remote query by messaging. The **request** message is a **Command Message** (contains the query), the **reply** message is often a **Message Sequence** (contains the result of the query)

- **Notification/Acknowledgement**: event notification and acknowledgement by messaging. The **request** message is an **Event Message** (contains the notification), the **reply** message is a **Document Message** (contains the acknowledgement – it can be another request for more details about the event)

Request-Reply pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Since a request is like a method call, a **reply can be**:

- **Void**: simple notification about the finish of the called method

- **Result value**: return value of the method as a single object

- **Exception**: an exception object indicating that the method aborted without successful completion and may include more details about the reason

The **request should contain a *Return Address*** (telling the receiver where to send the reply).

The **reply should contain a *Correlation Identifier*** (telling which request this reply is for).

Request-Reply pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: SOAP

SOAP is a typical example for Request-Reply.
A SOAP request indicates the **service** that the requestor intends **to invoke** on the server. The SOAP response contains either the **result of the service invocation or a fault**.

Example: JMS

(1) TemporaryQueue: a **private Queue to the connection** (**lasts as long as the connection** that created it; only MessageConsumers created by the same connection can read from it)
A requestor creates the TemporaryQueue and specifies it in the reply-to property of a request message (like Return Address).
Although, when the connection closes, the TemporaryQueue and the messages on that are deleted: so, Guaranteed Delivery is not possible

Request-Reply pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: JMS (cont.)

(2) QueueRequestor: contains a QueueSender for sending requests and a QueueReceiver for receiving replies. **Each requestor creates its own temporary queue** for receiving replies and specifies that in the request's reply-to property:

```
QueueConnection connection = // obtain the connection
Queue requestQueue = // obtain the queue
Message request = // create the request message
QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
QueueRequestor requestor = new QueueRequestor(session, requestQueue );
Message reply = requestor.request(request);
```

The method *request()* sends the request and blocks until it receives the reply.

TemporaryQueue and **QueueRequestor** work with **Point-to-Point Channel**.

Similarly, **TemporaryTopic** and **TopicRequestor** work with **Publish-Subscribe Channel**.

Request-Reply pattern X.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Related patterns: Command Message, Correlation Identifier, Document Message, Remote Procedure Invocation, Event Message, Guaranteed Delivery, Message, Message Channel, Message Sequence, Messaging, Point-to-Point Channel, Polling Consumer, Publish-Subscribe Channel, Return Address

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Return Address pattern

Return Address pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: An application applies Request-Reply by messaging.

Problem: How does a replier know where to send the reply?

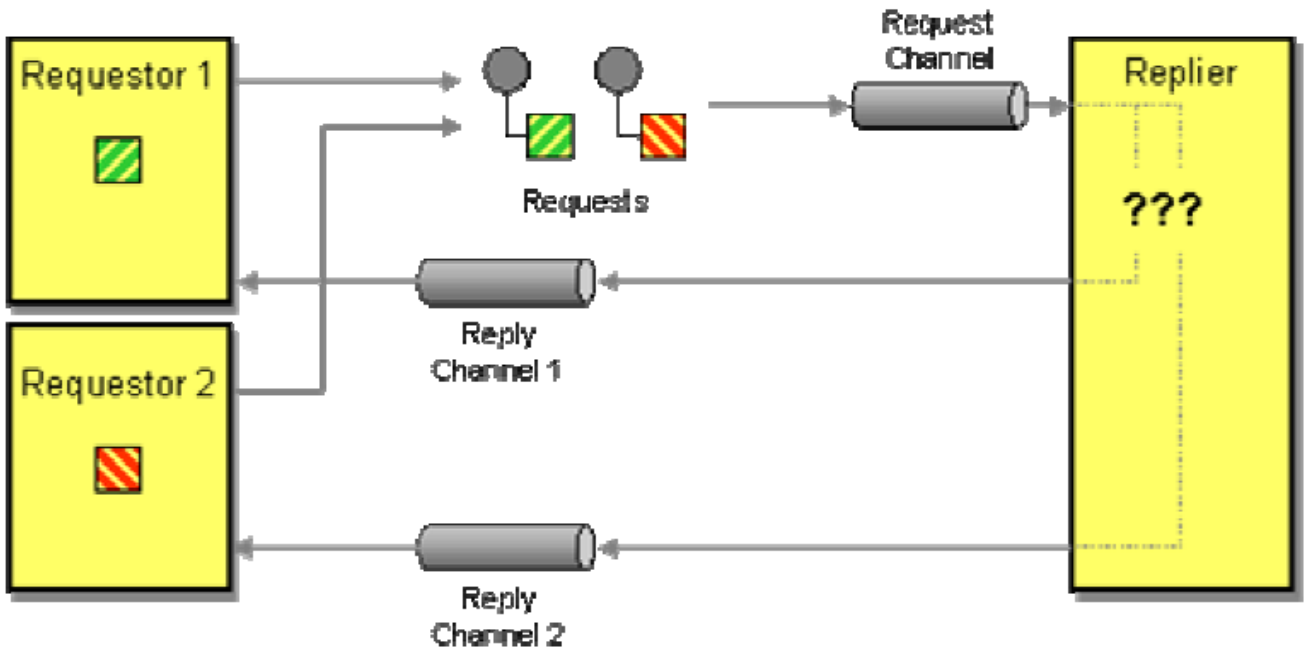
Usually **messages are independent**, however, there are exceptional cases, like Request-Reply: **the replier** after processing the request cannot simply send the reply on any channel it wishes, it **must send the reply on the channel the requestor expects that on**.

How to know the reply channel?

If it would be **hardcoded** in the replier, it were **difficult to maintain** and it **is not a flexible** solution. Moreover, there can be several different requestors and the reply channels should be different for them.

Return Address pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



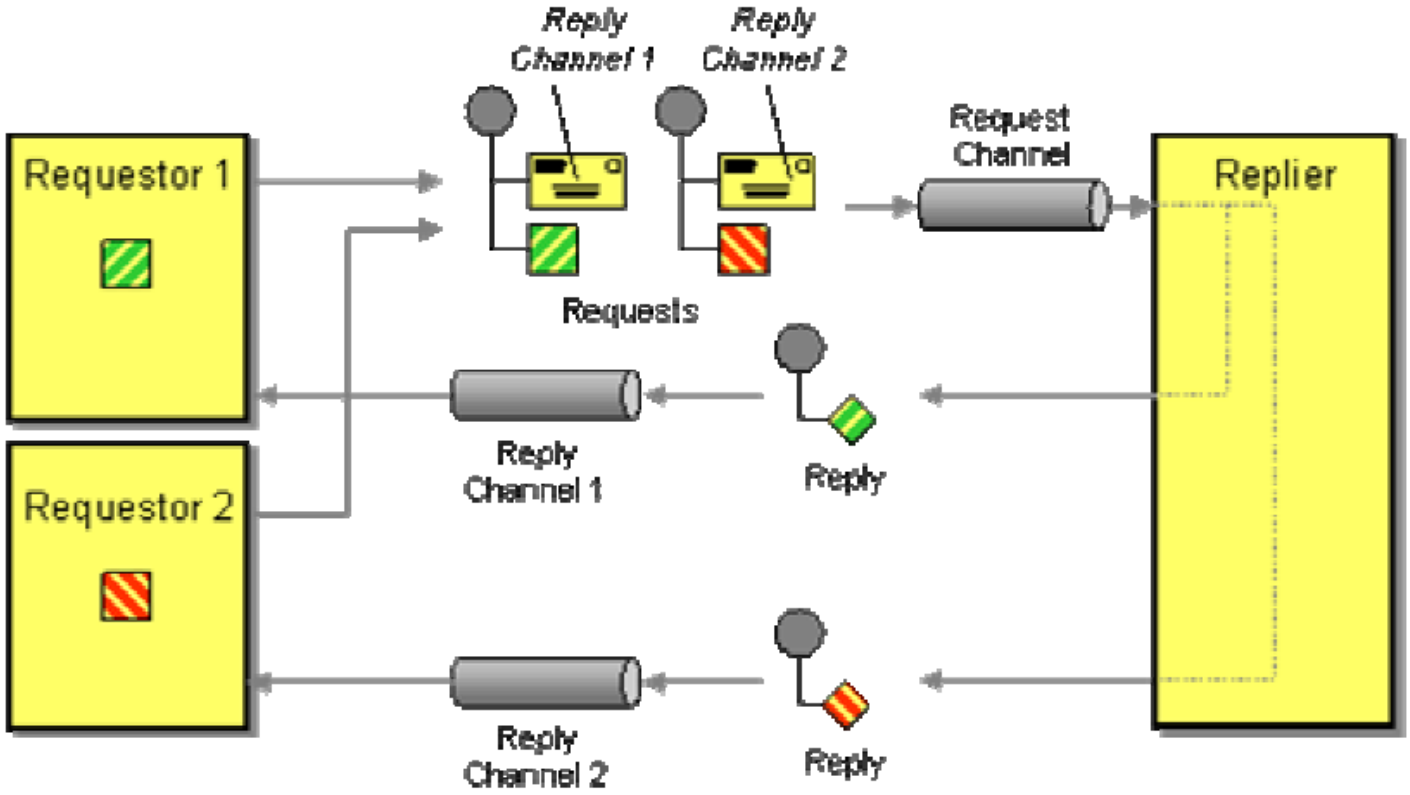
There are also cases when the requestor wants **the reply to be sent to a 3rd party** /a callback processor/ (sometimes **for different requests different reply channels**).

It would be an easy solution if the **request explicitly specifies** on which channel the reply is required.

Return Address pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution:</u> The request message should contain a *Return Address* that indicates where to send the reply message.



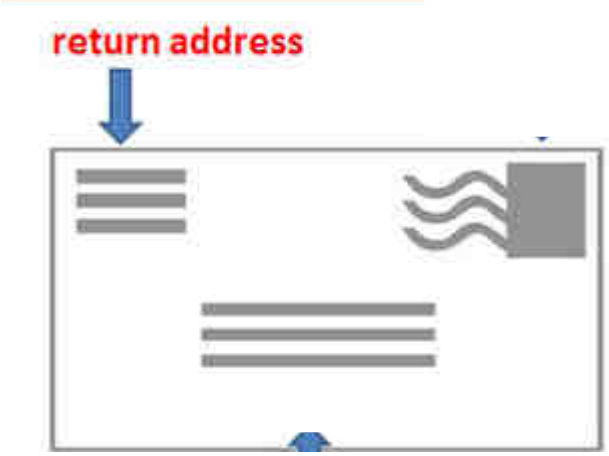
Return Address pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The **Return Address is** put **in the header** of a message because it's not part of the data being transmitted.

The Return Address is **similar to the reply-to field** in an e-mail.

When the reply is sent onto the channel that is identified by the Return Address, **Correlation Identifier also might need**: The Return Address tells the receiver which channel to put the reply message on; the Correlation Identifier tells the sender which request a reply is for.



Return Address pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: JMS: JMSReplyTo

It is not a simple string but has **Destination type** (Topic or Queue); ensuring that the destination really exists – at least in the moment the request is sent.

Sender site (specifying a queue as the reply channel):

```
Queue requestQueue = // Specify the request destination
Queue replyQueue = // Specify the reply destination
Message requestMessage = // Create the request message
requestMessage.setJMSReplyTo(replyQueue);
MessageProducer requestSender =
        session.createProducer(requestQueue);
requestSender.send(requestMessage);
```

Return Address pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: JMS: JMSReplyTo (cont.)

Receiver site (sending back the reply):

```
Queue requestQueue = // Specify the request destination
MessageConsumer requestReceiver =
        session.createConsumer(requestQueue);
Message requestMessage = requestReceiver.receive();
Message replyMessage = // Create the reply message
Destination replyQueue = requestMessage.getJMSReplyTo();
MessageProducer replySender = session.createProducer(replyQueue);
replySender.send(replyMessage);
```

Return Address pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: .NET: ResponseQueue

It is not a simple string but has **MessageQueue type**.

Example: Web Services

Based on the emerging WS-Addressing standard, a web service endpoint address intends to specify the Return Address.

Related patterns: Correlation Identifier, Message Channel, Messaging, Request-Reply

Correlation Identifier pattern

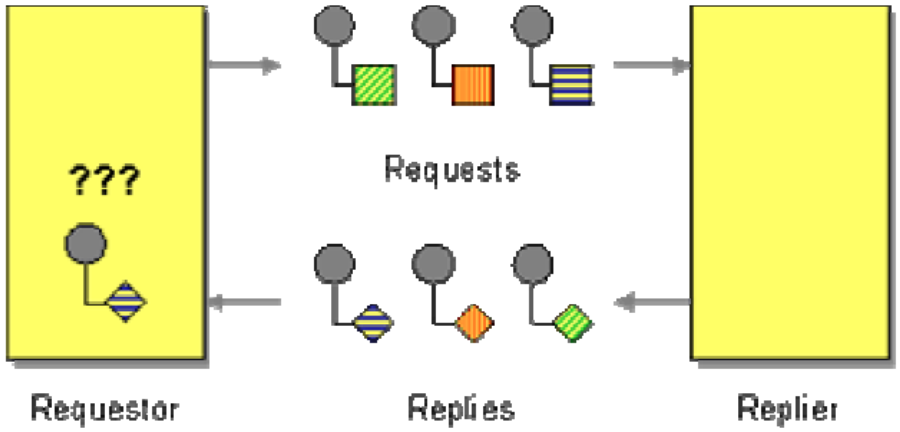
Correlation Identifier pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: An application received a reply message by messaging based on Request-Reply.

<u>Problem:</u> How does a requestor that has received a reply know which request this is the reply for?

In case of synchronous call (like RPC) there is no question about which call has produced the result. However, in case of **asynchronous** call (like messaging) the result can appear **later** or there are **many request**, so **the determination of which call the reply belongs to is not trivial**.



Correlation Identifier pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Possible solutions for coupling the reply with the request:

- to make **only one call at a time** (it will decrease the **throughput** highly)

- caller could assume that the **replies follow the same order** as the requests did (faulty assumption – **messaging does not guarantee** that)

- requests **do not need reply** (too strong **constraint**)

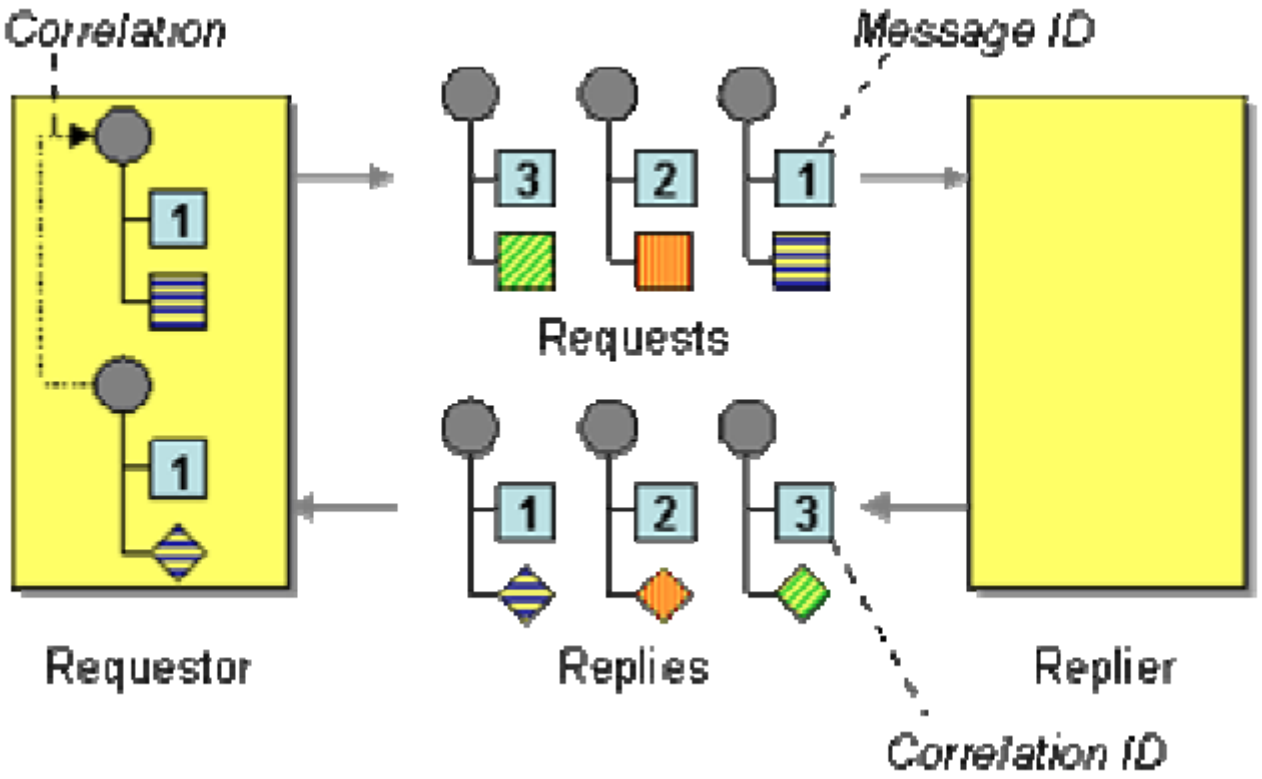
- replies should have some **reference or pointer** to the appropriate request message (but messages **do not have a stable space in the memory**)

- messages should have a **foreign key/unique identifier** (like a key for a row in a relational database)

Correlation Identifier pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution:</u> Each reply message should contain *a Correlation Identifier,* a unique identifier that indicates which request message this reply is for.



Correlation Identifier pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The 6 parts of Correlation Identifier:

- **Requestor**: the application that sends the request and waits for the reply
- **Replier**: the application that receives the request, processes it and sends back the reply. It gets the request ID from the request and puts it into the reply as the Correlation ID
- **Request**: the message from the requestor to the replier that contains the request ID
- **Reply**: the message from the replier to the requestor that contains the correlation ID
- **Request ID**: a token in the request that uniquely identifies the request
- **Reply ID**: a token in the reply that has the same value as the request ID of the request

Correlation Identifier pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The requestor **creates the request** message and **assigns a request ID** to that (the ID is different from those for all other currently outstanding requests).

The replier **processes the request**, **saves the request ID** and **adds that to the reply** message as correlation ID.

The requestor **processes the reply** and **uses the correlation ID** to know which request the reply is for.

The requestor and the replier have to **agree in the type and name of the request ID and the correlation ID**.
Usually **the two IDs have the same type**. **Otherwise**, the requestor **has to know how the replier converted** the request ID to correlation ID.

Correlation Identifier pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Concepts of a similar pattern:

Correlation Identifier pattern	Asynchronous Completion Token pattern
Requestor	Initiator
Replier	Service
The process in the requestor who handles the reply	Completion Handler
Correlation Identifier	Asynchronous Completion Token

Correlation Identifier pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The correlation ID (and the request ID, too) is **placed into the header**, since it is not part of the command or the data the requestor sends to the replier.

What to select to be the unique identifier?

(1) If it is a simple **message ID**, that is **not too interesting** to the requestor. (if it would remind the requestor of **what business task** the reply belongs to, that would be more useful)

(2) If the business tasks (like execute a stock trade or ship a purchase order) are identified by a **unique business object ID**, it can be used as the correlation ID. This way, **when a requestor gets a reply, it can bypass the request** and go straight to the business object.

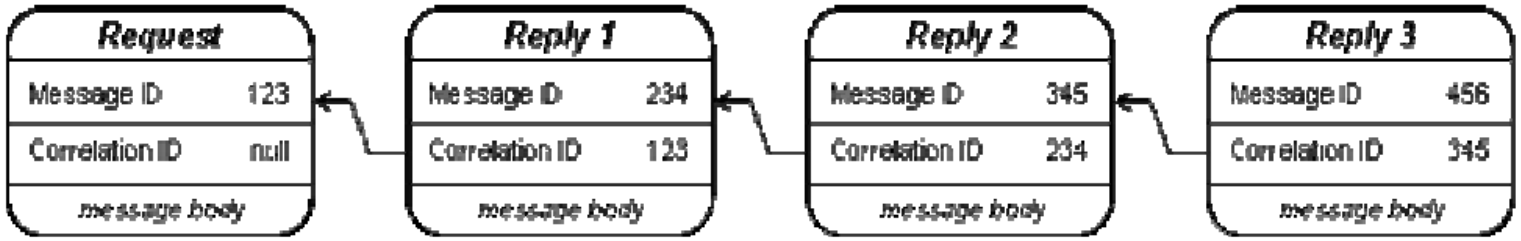
Correlation Identifier pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

What to select to be the unique identifier? (cont.)

(3) A compromise solution: the requestor keeps a **map of the request IDs and the business object IDs**. This is a good solution, e.g., if the requestor intends to **keep the object IDs private**.
This way, the requestor - after getting the reply -, looks for the business object ID (e.g., order ID) in the map based on the message ID, and continues the business task using the reply data.

If we want to ensure the **reply to be able to cause an other request**, there have to be **both a message ID and a correlation ID** in each message. It makes the **chaining of the messages** possible.



Correlation Identifier pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Chaining (and so different correlation IDs) **is useful** only if the path of the messages is intended to be **retrace**d from the last reply till the first request.

Often applications need **only to know the first request** of the chain. In these cases all the messages in the chain (excepting the original request) should have the message ID of the first request as their correlation ID.

Another possible fields of a message:

- **Return Address**: tells what channel to put the reply on

- **Message Sequence identifier**: specifies a **message's position** within a series of messages from the same sender.

Correlation Identifier pattern X.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: JMS

The predefined property is: **JMSCorrelationID**

The reply's correlation ID is set based on the request's – also predefined – **JMSMessageID** property.

```
Message requestMessage = // Get the request message
Message replyMessage = // Create the reply message
String requestID = requestMessage.getJMSMessageID();
replyMessage.setJMSCorrelationID(requestID);
```

Correlation Identifier pattern XI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: .NET

A .NET Message has both an **Id** and a **CorrelationId** property. Both of them is a string.

Moreover, for peeking at and consuming a specific message on a .NET MessageQueue there are **PeekByCorrelationId(string)** and **ReceiveByCorrelationId(string)** functions.

Example: Web services

SOAP 1.1 does not support asynchronous messaging

SOAP 1.2 starts to support asynchronous messaging **by Request-Response Message Exchange Pattern**, however, **does not support multiple ongoing requests**, so it **does not define** a standard **Correlation Identifier field**

Correlation Identifier pattern XII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Web services (cont. 1.)

Web Services Architecture Usage Scenarios discusses several **different asynchronous** web services scenarios.

4 of them:

- Request/Response
- Remote Procedure Call
- Multiple Asynchronous Responses
- Asynchronous Messaging

They use **message-id** and **response-to** fields in the SOAP header to correlate a response to its request.

Correlation Identifier pattern XIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Web services (cont. 2.)

Request/Response Usage Scenario (request example):

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <n:MessageId>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:MessageId>
    </n:MsgHeader>
  </env:Header>
  <env:Body>
      ........
  </env:Body>
</env:Envelope>
```

Correlation Identifier pattern XIV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Web services (cont. 3.)

Request/Response Usage Scenario (reply example):

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <n:MessageId>uuid:09233523-567b-2891-b623-9dke28yod7m9</n:MessageId>
      <n:ResponseTo>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:ResponseTo>
    </n:MsgHeader>
  </env:Header>
  <env:Body>
      ........
  </env:Body>
</env:Envelope>
```

Related patterns: Remote Procedure Invocation, Message, Selective Consumer, Message Sequence, Messaging, Request-Reply, Return Address

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message Sequence pattern

Message Sequence pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: An application needs to send or waits for a huge amount of data, more than may fit into a single message.

<u>Problem:</u> How can messaging transmit an arbitrarily large amount of data?

It would be nice to send arbitrary amount of data in one message, however, **there are some limitations**:

- some **messaging implementation limit the message size**

- too **large messages** can **hurt the performance**

- **message producers or consumers** may **have limit on the message size they can handle** (e.g., many COBOL- and mainframe-based systems handle date in 32 Kb chunks)

Message Sequence pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Possible solutions:

- limit the application to **never need more data** that what **the messaging layer can handle** (it can be **against the desired functionality**)

- in case of too large amount of reply data the caller could **send multiple request, one for each result chunk** (in this case the **caller should know the number of result chunks**)

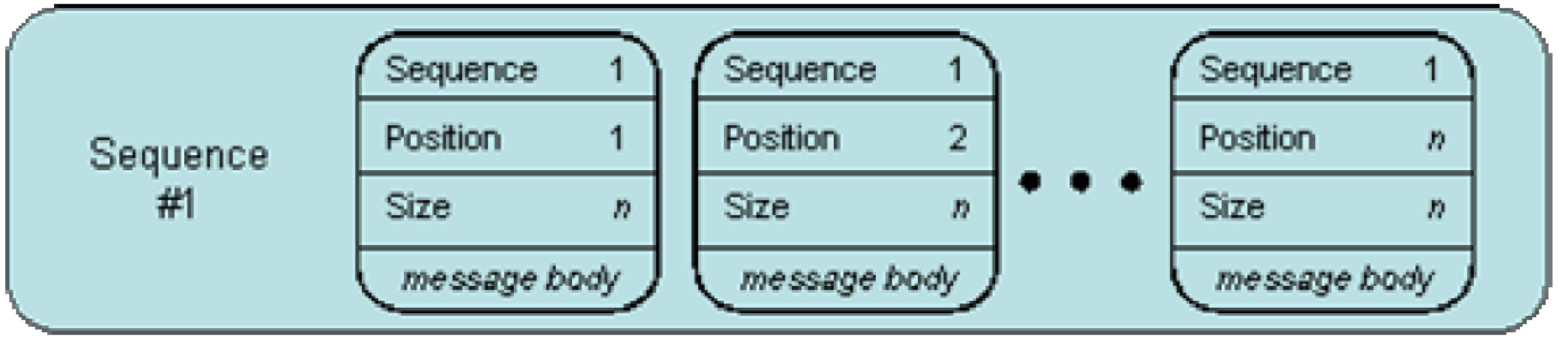
- the receiver could **listen for data chunks until there are no more** (the **receiver should know that there are no more chunks**, moreover, **it has to reassamble** the chunks without error)

- **marked messages like „1 of 3”, „2 of 3” and „3 of 3”**; it ensures that the **receiver knows which ones it has received** and **whether it has all of the messages**.

Message Sequence pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution</u>: Whenever a large set of data may need to be broken into message-size chunks, send the data as a *Message Sequence* and mark each message with sequence identification fields.



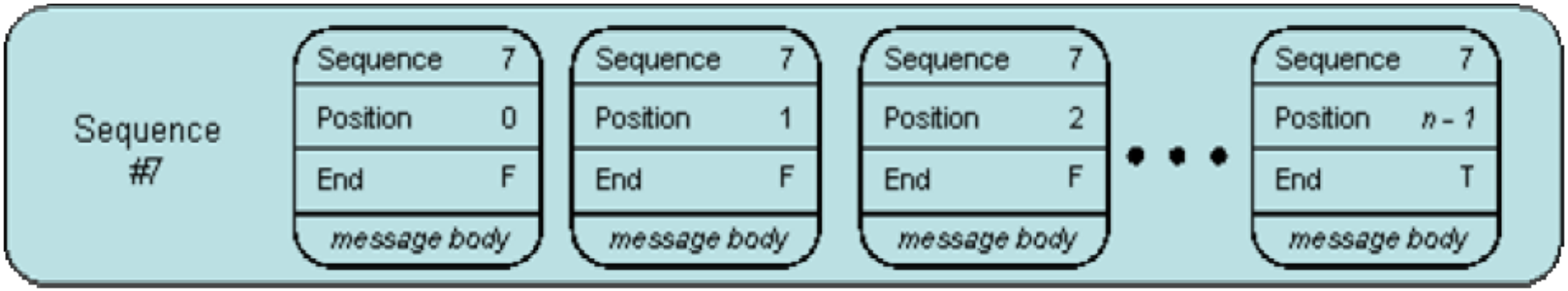
The **Message Sequence identification fields are**:

- **Sequence identifier**: **distinguishes** this sequence from the others
- **Position identifier**: uniquely **identifies** and sequentially **orders** the message of a message sequence
- **Size** or **End indicator**: shows the **number of messages** in a message sequence or marks the **last message** of a message sequence

Message Sequence pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Instead of the Size value in each message - that indicates the total number of messages in the sequence - it also can be indicated in each message whether that is the last one or not in that message sequence (**End indicator**):



If the **sender knows the number of messages** in the sequence, usually the **„Size" property** is applied. **Otherwise** (**e.g., in case of data streaming**) the sender only sends the messages with false value of the **„End indicator"** until it runs out of data, then in the last message the value of the „End indicator" property is true.

Message Sequence pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

No matter, which solution is applied of the two**, the Sequence identifier together with the Size/End indicator** property of the messages offer **enough information** the receiver **to reassemble the parts** (even if the parts are not received in sequential order).

If the receiver excepts a Message Sequence, then even a single message sized sequence **has to be sent as a message sequence** (**otherwise** the missing fields can cause the message to be judged as **invalid**).

If the receiver **never gets all of the messages** of a message sequence, the **received parts have to be routed onto the Invalid Message Channel**.

Message Sequence pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

If **Transactional Client** is used:

- The sender can send all the messages of the sequence in one transaction, so **none of the messages is delivered until all of them have been sent**.

- The receiver can receive all the messages of the sequence in one transaction, so it does **not consume any of the messages till all of them have been received**. If any of the messages in the sequence is missing, the receiver can decide to rollback the transaction and to consume the messages later.

- In **many** messaging system **implementations if the sending of the message sequence happens in one transaction**, the messages are received **keeping their order**.

Message Sequence pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

If Message Sequence is applied for the reply data of a Request-Reply, the **Correlation Identifier is practical to be used as the Sequence identifier, too**.
They **have to be separate fields only** when there are **multiple replies** to the same request and **at least one of them has multiple parts**.

Message Sequence should be transmitted by a **Message Channel with exactly one consumer**:
Message Sequence is **not compatible with Competing Consumers nor Message Dispatcher**. If different applications receive different messages of the same sequence, none of them will be able to reassamble the original data without data exchange between each other.

An **alternative solution** is ***Claim Check***:
It **stores** the large data **in a common database or a file**, and **transmits only the receipt**.

Message Sequence pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

In **MSMQ** the **maximal message size** is **4 MB**.

Usage examples:

- transmission of a **large document** that does not fit into a sinle message

- a **multi-item query**: each match to the same query is transmitted in a separate message **(e.g., the list of all books of an author)**

- **distributed query**: the parts of the same reply origin from different repliers and their sequence matters

Message Sequence pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

JMS vs. .NET

None of them has built-in properties for **support**ing **message sequences**.

JMS

Messaging applications **can define** their **own properties in the** message **header**.

.NET

Messaging applications **can NOT define** their **own properties in the** message **header**.

Solution can be to define message sequence-specific fields in the message **body**.

If the receiver has to **filter** for messages **based on** their **sequence ID**, it is **easier if the ID is stored in the header** and not in the body.

Message Sequence pattern X.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Web services

SOAP 1.1 does not support asynchronous messaging

SOAP 1.2 starts to support asynchronous messaging.

Web Services Architecture Usage Scenarios discusses several **different asynchronous** web services scenarios.

4 of them:

- Request/Response
- Remote Procedure Call
- Multiple Asynchronous Responses
- Asynchronous Messaging

They use **message-id** and **response-to** fields in the SOAP header to correlate a response to its request.
Moreover, **Multiple Asynchronous Responses** has **sequence-number** and **total-in-sequence** fields in the body to sequentially identify the responses.

Message Sequence pattern XI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Web services (cont. 1.)

Multiple Asynchronous Responses Usage Scenario (request):

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <n:MessageId>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:MessageId>
    </n:MsgHeader>
  </env:Header>
  <env:Body>
    ........
  </env:Body>
</env:Envelope>
```

Message Sequence pattern XII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Web services (cont. 2.)

Multiple Asynchronous Responses Usage Scenario (1st reply message):

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <!-- MessageId will be unique for each response message -->
      <!-- ResponseTo will be constant for each response message in the sequence-->
      <n:MessageId>uuid:09233523-567b-2891-b623-9dke28yod7m9</n:MessageId>
      <n:ResponseTo>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:ResponseTo>
    </n:MsgHeader>
    <s:Sequence xmlns:s="http://example.org/sequence">
      <s:SequenceNumber>1</s:SequenceNumber>
      <s:TotalInSequence>5</s:TotalInSequence>
    </s:Sequence>
  </env:Header>
  <env:Body>
    ........
  </env:Body>
</env:Envelope>
```

Message Sequence pattern XIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Web services (cont. 3.)

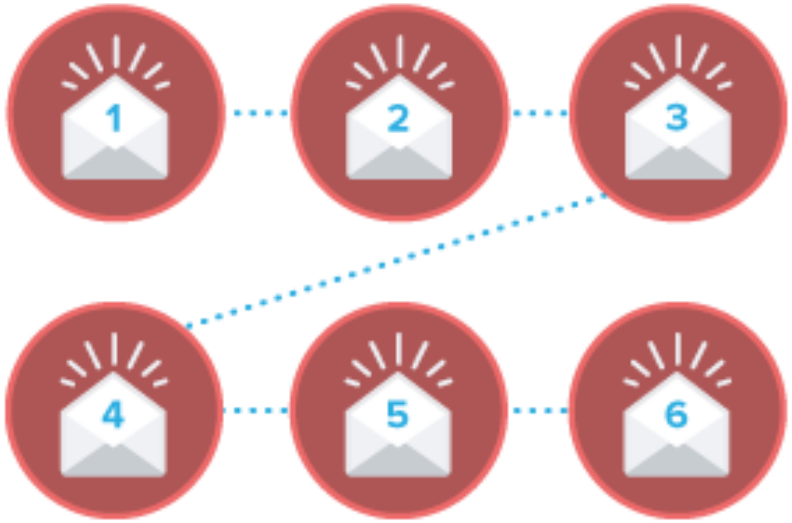
Multiple Asynchronous Responses Usage Scenario (final reply message):

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <!-- MessageId will be unique for each response message -->
      <!-- ResponseTo will be constant for each response message in the sequence-->
      <n:MessageId>uuid:40195729-sj20-pso3-1092-p20dj28rk104</n:MessageId>
      <n:ResponseTo>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:ResponseTo>
    </n:MsgHeader>
    <s:Sequence xmlns:s="http://example.org/sequence">
      <s:SequenceNumber>5</s:SequenceNumber>
      <s:TotalInSequence>5</s:TotalInSequence>
    </s:Sequence>
  </env:Header>
  <env:Body>
    ........
  </env:Body>
</env:Envelope>
```

Message Sequence pattern XIV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Related patterns: Competing Consumers, Correlation Identifier, Invalid Message Channel, Message Channel, Message Dispatcher, Request-Reply, Claim Check, Transactional Client



EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message Expiration pattern

Message Expiration pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: Using messaging, the data of a message is not delivered till the **desired time**. The message becomes **useless** and **should be ignored**.

Problem: How can a sender indicate when a message should be considered stale and thus shouldn't be processed?

Messaging – as follows store-and-forward method – is highly reliable, however it is not surely the case for the circumstances (sender, network, receiver). It causes that – though, **the message will be delivered**, – **it can not be guaranteed how long the delivery takes**.

In several cases, the **content of a message is useful only for a limited time (e.g., a stock quote request)**.

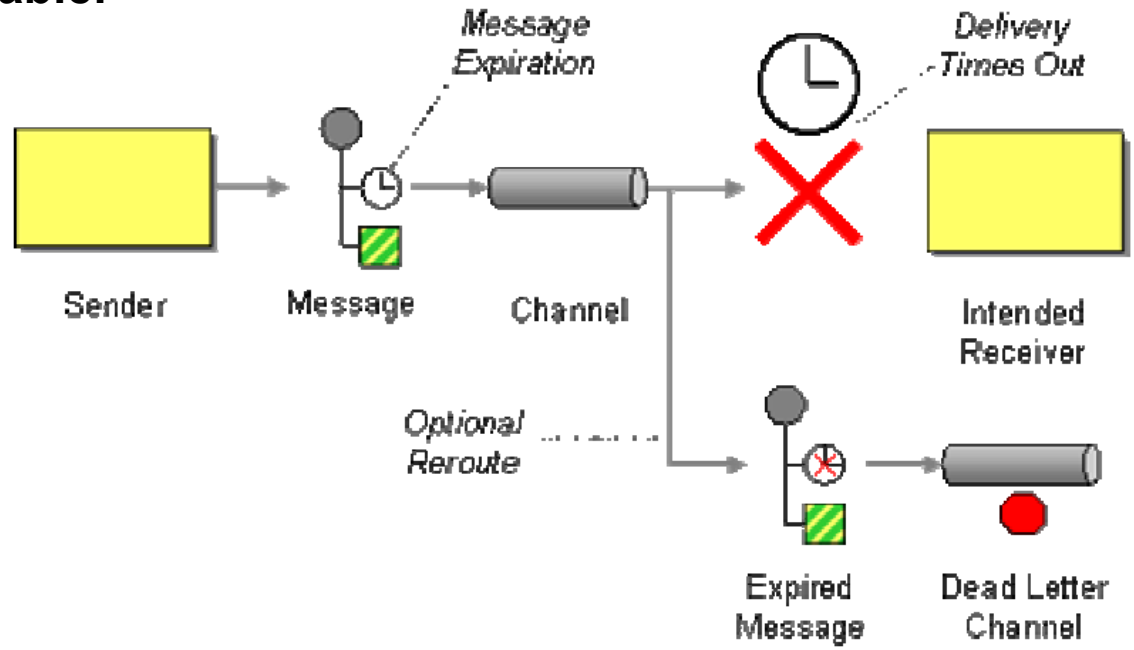
After the sender **sent the message**, it **can not be cancelled or recalled (e.g., if the reply does not arrive)**, so, somehow the receiver has to handle the problem.

Message Expiration pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The **receiver can check when the message was sent**. However, the receiver **should know, what time duration is too old** (it depends on the intension of the sender). So, the **sender should specify** somehow **the lifetime** of the message.

<u>Solution</u>: Set the *Message Expiration* to specify a time limit how long the message is viable.



Message Expiration pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The messages **expires** when the time for which that is viable passes and the message still **has not been consumed**. Expired messages should be either sent onto the **Dead Letter Channel or discarded by the messaging system**. If a **consumer** nevertheless receives an expired message, it **should trow it away, move it onto the Invalid Message Channel**.

A **Message Expiration** is a **timestamp** (date and time), that

(1) either shows **how long** the message **will live** /relative term/

(2) or gives **when** the messages **will expire**. /absolute term/

The **messaging system** will use the time when the message is sent to **convert the relative setting into an absolute** one.
Moreover, the **messaging system** has to:

- adjust the **different timezones**
- adjust the **daylight savings times**
- deal with other issues for ensuring that the **clocks of the different sites agree in what time it is**.

Message Expiration pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

If the **Message Expiration** is an **absolute value**, it **must be later than** the message's ***sent time*** **property** **(otherwise the message expires immediately).**

Senders usually **send relative message expiration** and the **messaging system calculates the absolute expiration time** **(sent time + time to live).**

In case of **Publish-Subscribe Channel**: it may happen that **some of the subscribers consume** their copy of the message **successfully but some copies of the message expires** before other subscribers consume them.

Message Expiration does not work well with Request-Reply: if the reply message expires, **the sender won't know whether the request was received** or not. This case **has to be handled in the design of the sender** of the request.

Message Expiration pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: the two ways of JMS:

- a predefined **JMSExpiration** property, but senders should not set it by ***Message.setJMSExpiration(long)***, because the **JMS Provider overrides it** when the message is sent.

- the sender's MessageProducer (QueueSender or TopicPublisher) should set the **timeout** value (in milliseconds):

  a. **for all messages** by the ***MessageProducer.setTimeToLive(long)***

  b. **for individual messages** by the ***MessageProducer.send(Message message, int deliveryMode, int priority, long timeToLive)*** function.

Message Expiration pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: the two ways of .NET:

- **TimeToBeReceived** property: equivalent to JMS's JMSExpiration property. Limits the **time to transmitting** the message to its destination **plus** the time that the message can **wait in the queue at the destination**.

- **TimeToReachQueue** property: defines the maximum **time length for reaching the destination's queue** (after that the message can sit in the queue indefinitely).

Both cases express the lenght of time by a value of **type System.TimeSpan**.

Related patterns: Dead Letter Channel, Guaranteed Delivery, Invalid Message Channel, Message, Messaging, Publish-Subscribe Channel, Request-Reply

Format Indicator pattern

Format Indicator pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: Applications communicate by messages that follow an **agreed-upon data format** (e.g. an **enterprise-wide Canonical Data Model**). The format may **need to change**.

<u>Problem:</u> How can a message's data format be designed to allow for possible future changes?

There can be **need for data format change**, e.g.:

- **new applications having new format requirement** have to be integrated
- **new data have to be added** to the messages
- developers want to **change the structure** of the same data

Even if the challengig task of designing **a common data format for all the involved applications** is successful, it **may have to be changed**.

Format Indicator pattern II.

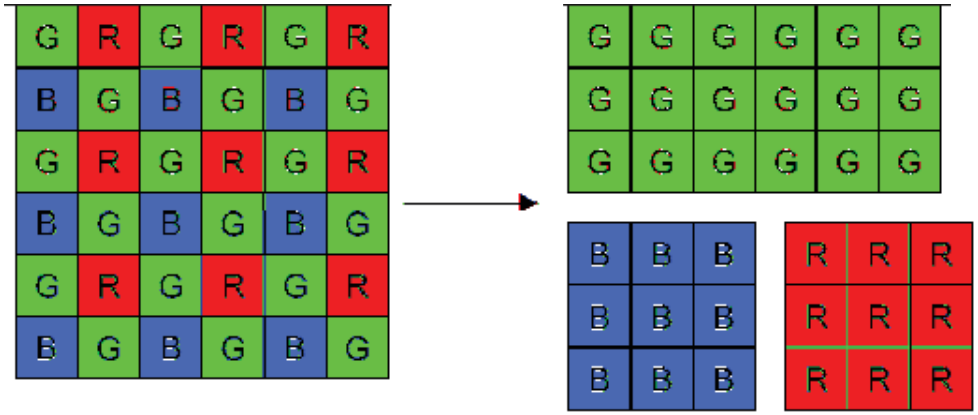
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The **timing of the format change can cause problems**:

- if **some applications change** the format **before others** (even some rarely used aplications **never change** that)

- if there still remain some **unconsumed messages** on the channel **after** the applications **change the format**.

Conversion would be simple only if all the applications change the format **exacty the same time** and that time there is **no messages on the channels**.



Format Indicator pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The real solution could be that the **applications support both the old and the new format,** parallel. For that, the applications have to **differentiate between the messages with the old and the new format.** It can be done by:

(1) applying **separate sets of channels** for the old and new messages. It is **too complex** solution with **duplicated design** and **lots of channels**.

(2) sharing the same channels by both formats, however, the receivers can do the differentiating based on the **format specification contained in the messages**.

Solution: Design a data format that includes a *Format Indicator*, so that the message specifies what format it is using.

This way, **a sender can tell the receiver the format** of the message. Moreover, the receiver – getting the format of the message – will know **how to interpret** the content of the message.

Format Indicator pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Alternatives to implement the Format Indicator:

- **Version Number**: a **string or number**, that – based on the agreement between the sender and the receiver – **identifies the format** of the message.

- **Foreign Key**: a **unique ID** that **specifies a format document (like a filename, a database row key, a primary key or an Internet URL)**. Both the **sender and the receiver have to know the mapping of the key** to the document **and the format of the document**.

- **Format Document**: a schema document that is **included in the message** and **describes the data format**. Both the sender and the receiver **have to know the format of the document**.

Format Indicator pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Where to place the format Indicator?

- if it is a **Version Number** or a **Foreign Key**, it can be stored **in the header**
(in an agreed upon field by the sebder and the receiver)

- if it is a **Format Document**, it can be **too long or complex** to be stored in the header. In that case the **message body** should contain **two parts**: one for the **schema document** and one for the **data**.

Format Indicator pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: XML

(1)
```
<?xml version="1.0"?>
```

Version Number: indicates the document's conformance to the version 1.0 of the XML specification.

(2)
```
<!DOCTYPE greeting SYSTEM "hello.dtd">
```

Foreign Key: indicates the file containing the DTD document that describes this XML document's format.

(3)
```
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
```

Format Document: an embedded schema document that describes the XML's format.

Related patterns: Canonical Data Model, Message

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

THANK YOU FOR THE ATTENTION!

Reference:

Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns – Designing, Building and Deploying Messaging Solutions, Addison Wesley, 2003, ISBN 0321200683

www.enterpriseintegrationpatterns.com

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

ENTERPRISE INTEGRATION PATTERNS

11-12-13-14. Message routing

Author: Tibor Dulai

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE

Introduction

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The main purpose of a Message Router is to **decouple the sender from the destination**.

The main categories of Message Routing pattern

- **Simple Routers**: variants of the Message Router; they **route** a message **from one inbound** channel **to one or more outgoing** channels.

- **Composed Routers**: a **combination of more Simple Routers** for ensuring a more complex message flow

- **Architectural Patterns**: describe **architectural styles based on Message Routers**.

Simple Routers I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

1. **Content-Based Router**: routes the message **based on the content of the message**. The **sender does not have to know the exact destination**, it is determined by the router.

2. **Message Filter**: a special Content-Based Router. **Examines** the message content and **sends** the message to another channel **if the content matches cetrain criteria**. **Otherwise**, it **discards** the message.

   It is often applied together with **Publish-Subscribe Channels**: the message is **published and the recipients filter out** the irrelevant messages.

   Its function is **similar to** the operation of **a Selective Consumer**, however, a **Message Filter is part of the messaging system**, while a **Selective Consumer is part of a Message Endpoint**.

Simple Routers II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The **behaviour of a Content-Based Router** also **can be achieved by Message Filters** (applying a Publish-Subscribe Channel).
However, the **Content-Based Router** makes routing decision **predictively** to a single channel and in a **centralized** way. Contrary to that, the using of **Message Filters** works **reactively** and **spreads the routing logic** across many Message Filters.

3. **Dynamic Router**: while a basic Message Router uses fixed rules, the Dynamic Router allows the **routing logic to be modified by control messages sent to a control port**. It can be applied to most forms of Message Routers.

4. **Recipient List**: a Content-Based Router that allows to route the message to **more than one destination channel**, **maintaining the control** over the recipients.

Simple Routers III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

5. **Splitter:** if a message contains a list of individual items, Splitter **splits the message into individual messages**, that can be routed and processed individually.

6. **Aggregator: receives a stream** of messages, i**dentifies the related messages and combines them** into a single message. An Aggregator has a specific behaviour: it can **consume several messages before it publishes one**. So, Aggregator is **stateful**: it stores the messages until specific conditions are fulfilled.

7. **Resequencer:** because **messages** can be processed parallel by different processes **(e.g. messages can be consumed off a single channel by several processes, some of them are quicker than the others)**, they **can get out of order**. Resequencer **puts out-of-order messages into a correct sequence** of individual messages. Resequencer is also **stateful**, since it may have to store messages until the messages arrive that complete the sequence. However, it **publishes as many messages as it consumes**.

Simple Routers IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Pattern	# of messages it consumes	# of messages it publishes	Stateful?	Comment
Content-Based Router	1	1	No (mostly)	
Message Filter	1	0 or 1	No (mostly)	
Recipient List	1	multiple (including 0)	No	
Splitter	1	multiple	No	
Aggregator	multiple	1	Yes	
Resequencer	multiple	multiple	Yes	Publishes the same number as consumes

Any of them can be implemented as a **dynamic** variant.

Composed Routers I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

1. **Composed Message Processor**: **combines multiple Message Routers** – following the Pipes and Filters architecture – **for retrieving information from multiple sources and recombining it into a single message**. Composed Message Processor **maintains control** over the sources.

2. **Scatter-Gather**: **combines multiple Message Routers** – following the Pipes and Filters architecture – **for retrieving information from multiple sources and recombining it into a single message**. Applies **Publish-Subscribe Channel**, so **any interested component can participate in the process**.

Both solutions manage the **parallel routing** of a message: they **route** a message **to several participants concurrently** and then **reassamble the replies** into a single message.

Composed Routers II. + Architectural Patterns

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

3. **Routing Slip: specifies the path** the message should take, that follows a sequence of individual steps. Routing Slip manages **sequential routing** of a message from a **central point**.

4. **Process Manager**: **specifies the path** the message should take, that follows a sequence of individual steps. Process Manager realizes **sequential routing** in a **flexible way** but requires the message to **return to a central component after each function**.

Architectural Patterns

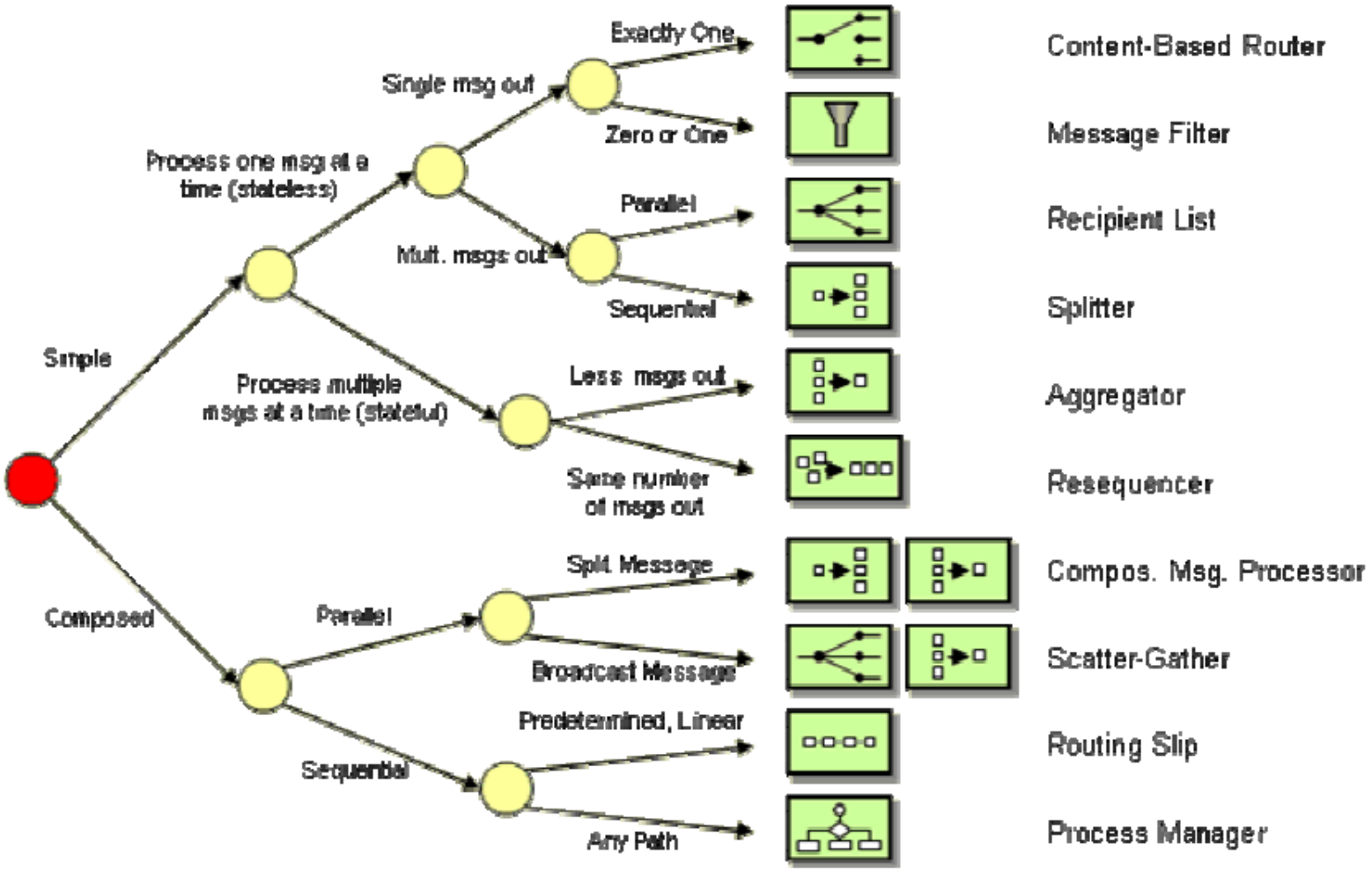
These patterns describe a **hub-and-spoke architectural style**.

1. **Message Broker**: a **central element** for integration solutions.

Router selection

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

How to find the proper Router pattern for the proper purpose:



The diagram also illusrates how closely the individual patterns are related.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Content-Based Router pattern

Content-Based Router pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: In an order processing system there is a part for **inventory check**, that is **spread across more than one inventory systems,** where each system is **able to handle only specific items**.

Problem: How do we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems?

Business functions are rarely encapsulated in a single system, since **applications were mainly developed without integration in mind**. That's why there are different applications that **contain the same business function (like place order or check inventory, for example)**. These application can **belong to the same or to differenct companies**, too.

Content-Based Router pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Let's have an example of a company that sells **two types of product**: A and B. When the company gets an order, it has to verify whether the product exist in the desired number in the inventory. However, there are **separated inventories for product types** A and B. The **desired product's type is included in the order** as part of the item's unique item number.
The company has to decide which inventory to pass the order to.

(1) It can be done by using **separated channels for the incoming orders based on the type** of the ordered item. It pushes the decision to the **customer**, who **is required to know the internal structure** of the company (distinguishing types A and B). It is **not the desired** way.

Orders for different products are expected to arrive on the same channel.

Content-Based Router pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

(2) The order can be **forwarded** **to all inventories by a Publish-Subscribe Channel** and each **inventory has to decide** whether it can deal with the order.

- In this case the **adding of a new inventory system is easy** (it does not change any of the existing components).

- This approach assumes **distributed coordination**:
  - What happens if **none of the systems can process** the order?
  - What happens if **more than one system processes** the order? **(duplicate shipments?)**
  - If the system that **can not process** the order treats it as an error, **how to differentiate it to real errors**?

Content-Based Router pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

(3) The **item number could be used as channel address**. So, each item has its own, dedicated channel. Customer only has to put the order onto the channel that is associated with the desired product's number, but he is not required to know about which inventory handles which items. **Inventories have to listen on all the channels that belong to them**.

- this solution has the danger of possible **explosion of the number of channels**, that can lead to huge **run-time and management overhead**.

It is not a good idea to create new channels for each ordered item.

Content-Based Router pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

(4) The **order** could be **passed from one inventory system after the other, in a sequence**. The **first inventory that can process** the order, **consumes** the order message. **If an inventory can not process** the order, **passes** the order message **to the next system**.

- This case **eliminates** the possibility of **duplication**.
- **We can realize that nobody is able to handle** the order, if the last system can not process it.
- Good solution to **minimize message traffic**.

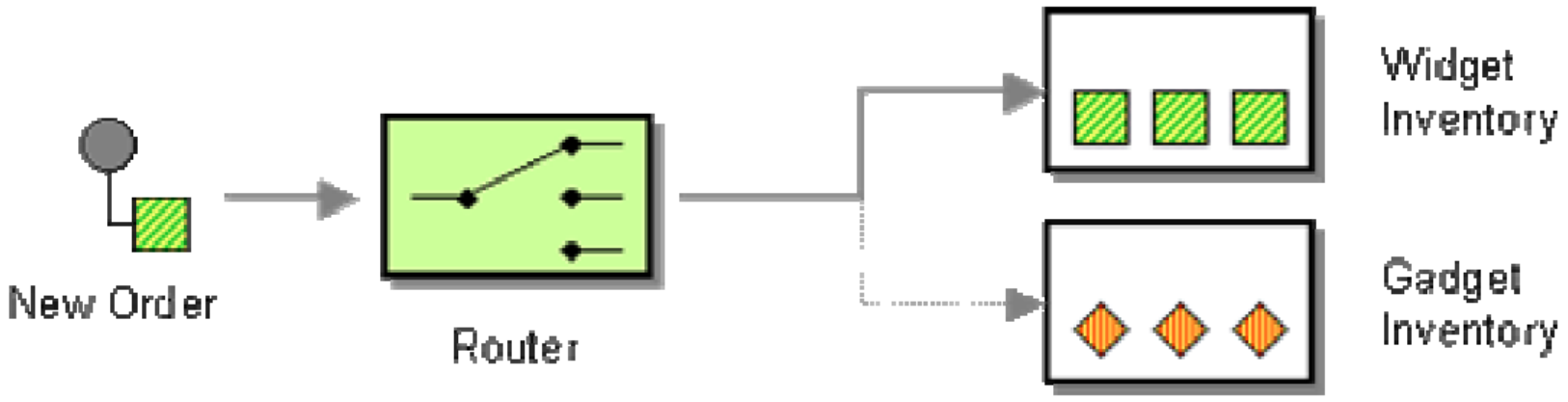
- This solution **requires the systems to have enough knowledge about each other** for passing the order from one to the next.
- If some of the systems are out of our control (e.g. systems of the business partners) their **collaboration is nearly impossible**.
- The chaining of possible destinations could mean **big overhead**.

Content-Based Router pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A **solution is needed** that handles a business function that is splitted **across systems**, **efficient** in message traffic and in channel requirements, and ensures that an **order is processed by exactly one** system.

<u>Solution</u>: Use a *Content-Based Router* to route each message to the correct recipient based on message content.



Content-Based Router pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A Content-Based Router **examines the content** of the messages **and selects the outbound channel** based on the content of the message.

The **selection** of the outbound channel can happen:

- **based on the existence of fields**
- **based on the value of a field**, etc.
- (sometimes) based on a set of **configurable rules**.

It is important to keep the **routing function easy to maintain** (maintenance can happen frequently).

Content-Based Router pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Content-Based Router

predictive routing

+

- efficient: each outgoing message goes **directly to the desired destination**

-

- has to **know all possible recipients** and their capabilities (since it may change frequently, **maintenance is hard**)

reactive routing

recipients have more **control**: each participants filter the relevant messages (e.g., Message Filter / Routing Slip)

+

- no need for centralized control of routing

-

- **less efficient**

Dynamic Router: each **recipient can inform** the Content-Based Router of its capabilities and the router maintain a list of them for routing.

+: more flexible -: more complex, difficult to debug

Content-Based Router pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: C# and MSMQ (1/3)

A **stateless router**, that routes messages **based on the first letter** of their body:
W → widgetQueue G → gadgetQueue none of them → dunnoQueue

```
class ContentBasedRouter
{
    protected MessageQueue inQueue;
    protected MessageQueue widgetQueue;
    protected MessageQueue gadgetQueue;
    protected MessageQueue dunnoQueue;

    public ContentBasedRouter(MessageQueue inQueue, MessageQueue widgetQueue,
MessageQueue gadgetQueue, MessageQueue dunnoQueue)
    {
        this.inQueue = inQueue;
        this.widgetQueue = widgetQueue;
        this.gadgetQueue = gadgetQueue;
        this.dunnoQueue = dunnoQueue;

        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();
    }
```

acts as an Invalid Message Channel

handler for incoming messages

Content-Based Router pattern X.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: C# and MSMQ (2/3)

what type of messages it expects (simple strings)

```
private void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
{
    MessageQueue mq = (MessageQueue)source;
    mq.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
    Message message = mq.EndReceive(asyncResult.AsyncResult);

    if (IsWidgetMessage(message))
        widgetQueue.Send(message);
    else if (IsGadgetMessage(message))
        gadgetQueue.Send(message);
    else
        dunnoQueue.Send(message);
    mq.BeginReceive();
}
```

Content-Based Router pattern XI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: C# and MSMQ (3/3)

```
    protected bool IsWidgetMessage (Message message)
    {
        String text = (String)message.Body;
        return (text.StartsWith("W"));
    }

    protected bool IsGadgetMessage (Message message)
    {
        String text = (String)message.Body;
        return (text.StartsWith("G"));
    }
}
```

The example is **not transactional**: an error during message processing – before sending that to one output – may result in **message lost**.
Transactional Client can be a solution.

Content-Based Router pattern XII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: TIBCO (1/3)

Many EAI tools – like **TIBCO ActiveEnterprise suite**, too – provide **built-in solution** for common elements of a messaging system, like a message router. For constructing a Content-Based Router – **instead of writing a long code** – we can use **drag-and-drop** operations and only **have to code the decision logic** of the router.

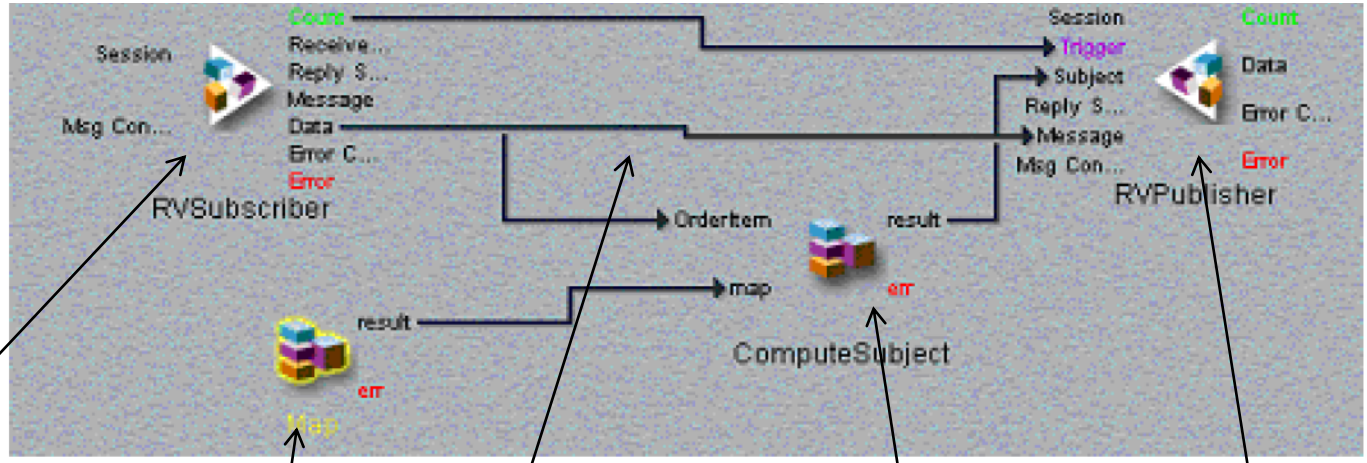
Simple message flows – including **transformation and routing** – can be created by **TIB/MessageBroker** of the suite.

Content-Based Router pattern XIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: TIBCO (2/3)

The former router, that works based on the first letter of the message body can be created using TIB/MessageBroker as follows:



Consumes messages off the channel router.in (channel name is specified in a property box)

Translation table between message contents and the destination channel name

Message content is transmitted unmodified

Determines the correct output channel

Uses the result as the output channel

Content-Based Router pattern XIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: TIBCO (3/3)

The map/translation table contains the following data:

Item code	Channel Name
G	gadget
W	widget

The implemented ComputeSubject function is:

```
concat("router.out.",DGet(map,Upper(Left(OrderItem.ItemNumber,1))))
```

So, the output channel name is either **„router.out.gadget”** or **„router.out.widget”** in case of a valid input message.

+: requires **less code,** ensures **transactionality, thread management, system management,** etc. **for free.**
-: Instead of UI, **many important settings are hidden in property fields**, so **hard to document.**

Related patterns: Dynamic Router, Message Filter, Invalid Message Channel, Message Router, Pipes and Filters, Publish-Subscribe Channel, Routing Slip, Transactional Client

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Message Filter pattern

Message Filter pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: A **company publishes price changes and promotions** to customers. However, there are **customers** who are **interested in information only related to some special items**.

Problem: **How can a component avoid receiving uninteresting messages?**

Simple solution for receiving only relevant messages:
Applying **separate Publish-Subscribe Channels for different messages** and the customer subscribes to channels only of his interest.

+

New subscribers do **not require any change** of the existing system

-

For **finer granularity** of interest, **lots of channels** are required. This solution is **difficult to manage**.

Message Filter pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

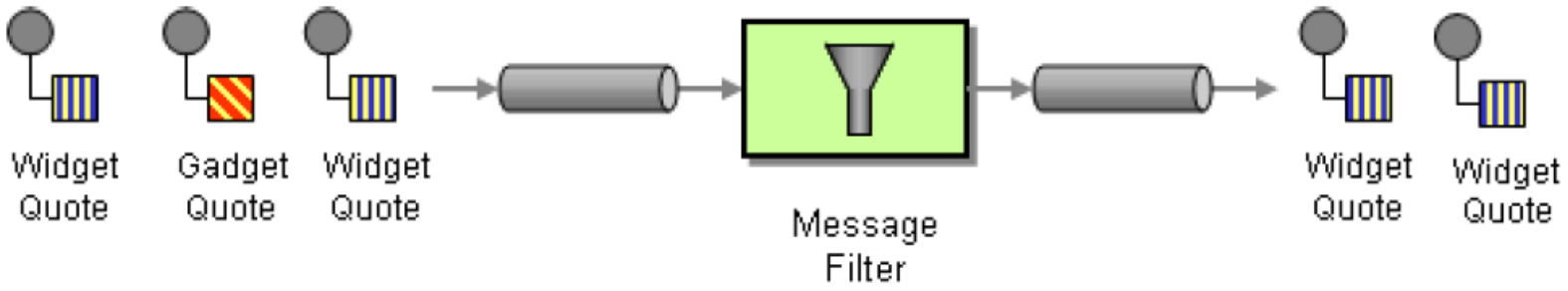
Another **requirement**: **accomodation to the frequent changes of the customer preferences.**
If we use a **modified Content-Based Router** that is able to route the incoming messages to **multiple receivers** (like **Recipient List**), we **devolve the handling of the preferences** of the receivers **to the message originator**. In case of frequent preference change, it is a **hard task**.

If we choose the solution of **broadcasting** the message and **leave the receiver components to filter out** the irrelevant messages, we **have to have control over those components**. It is **impossible** in case of **packaged applications**, **legacy applications** or **applications that do not belong to our company**.
Moreover, if the **filter logic** is **inside the receiver** components, receivers become **tightly coupled to the message type**, so their filter logic **can not be flexible** **(e.g., a generic price watch criteria can not be differentiated to specific product types)**.

Message Filter pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution</u>: Use a special kind of Message Router, a *Message Filter*, to eliminate undesired messages from a channel based on a set of criteria.



The Message Filter can be **considered as a Content-Based Router** that has **only one output** channel. If the content of the incoming message **matches the criteria** specified by the Message Filter, the message is **routed to the output channel**. **Otherwise**, the message is **discarded** **(sent into the null channel, that behaves like /dev/null or the Null Object)**.

The possible customers can listen on a **Publish-Subscribe Channel** and apply **Message Filter** to eliminate messages that do not meet their criteria.

Message Filter pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Stateless Message Filters

The stateless Message Filter **makes its decision based on the content only of the single message it receives**. It **does not have to maintain state** across messages.
One important **advantage** of stateless Message Filters is that **multiple instances** of them can be **applied parallel** to speed up the process.

Stateful Message Filters

Maintains **state** across messages: **keeps track of message history**.
Common usage of stateful Message Filters is the **elimination of duplicate messages** **(the unique message IDs of the past messages are stored, based on them duplications can be recognized).**

Message Filter pattern V.

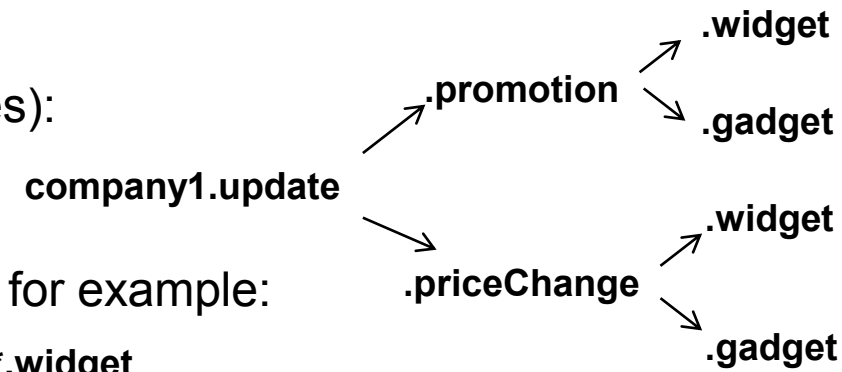
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

There can be **built-in filtering functions** in messaging systems:

1. **Hierarchical structure of Publish-Subscribe Channels** (e.g. in JMS)

   In this case subscribers can use wildcards to subscribe to a subset of messages.

   E.g. (channel names):

   Possible wildcards, for example:

   company1.update.*.widget
   company1.update.promotion.*

+: **flexible**, lets to **refine the semantics** of a channel **by appending qualifying parameters**
-: the **flexibility is limited** compared to Message Filters **(e.g. problem with price change over 34% updates)**

Message Filter pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

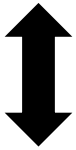
2. **Selective Consumers inside the receiving applications**

Message Selectors are expressions that **evaluate header or property** elements of an incoming message before the application gets to see the message. **If the condition does not evaluated to be true** the message is **ignored and not passed to the application logic**.
A Message Selector is **like a Message Filter that is built into the application**, it means that it **requires the access to the application**. However, the **execution of selection rules is built into the messaging infrastructure**. It means that a consumer with **Selective Consumer does not consume** the message **if that does not fit** to the desired criteria, in contrast to **Message Filter** that **consumes all messages from its input channel** but publishes only those messages that fit to the desired criteria.

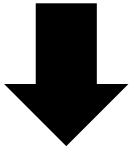
Message Filter pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

If the **filter expression is implemented in the messaging infrastucture**, then the messaging infrastructure is able to make **smart routing decisions**:
e.g., if the message is not desired by the distant receiver, it won't be transmitted over the network.



If **Message Filters** are applied then the **routing decision is not centralized** but **delegated to the recipients**.



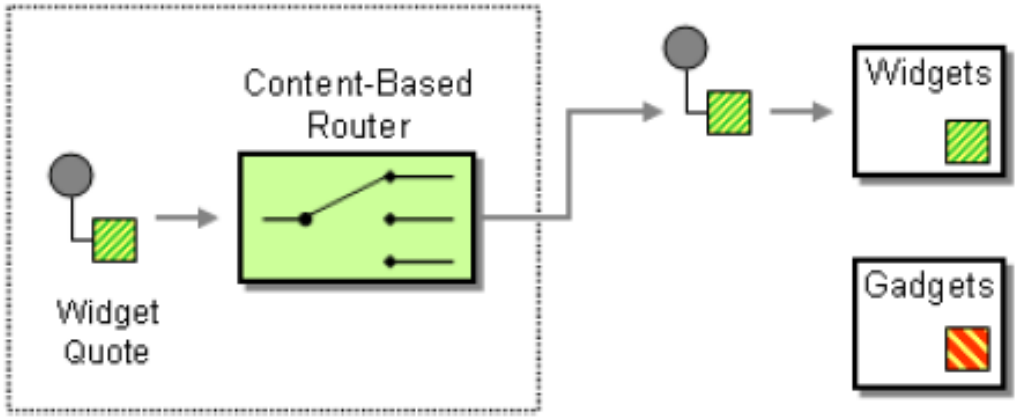
The **Message Filter** can be made the **part of the API of the messaging system** that is provided to the subscribers. This way the **control get closer to the subscribers** but **allows the messaging infrastructure to avoid unnecessary network traffic** (similarly to the **Dynamic Recipient List**). **Dynamic Recipient List** allows the consumers to **express their preferences and stores them (in a database or a rule base)**. The incoming message is forwarded to all consumers meeting the criteria.

Message Filter pattern VIII.

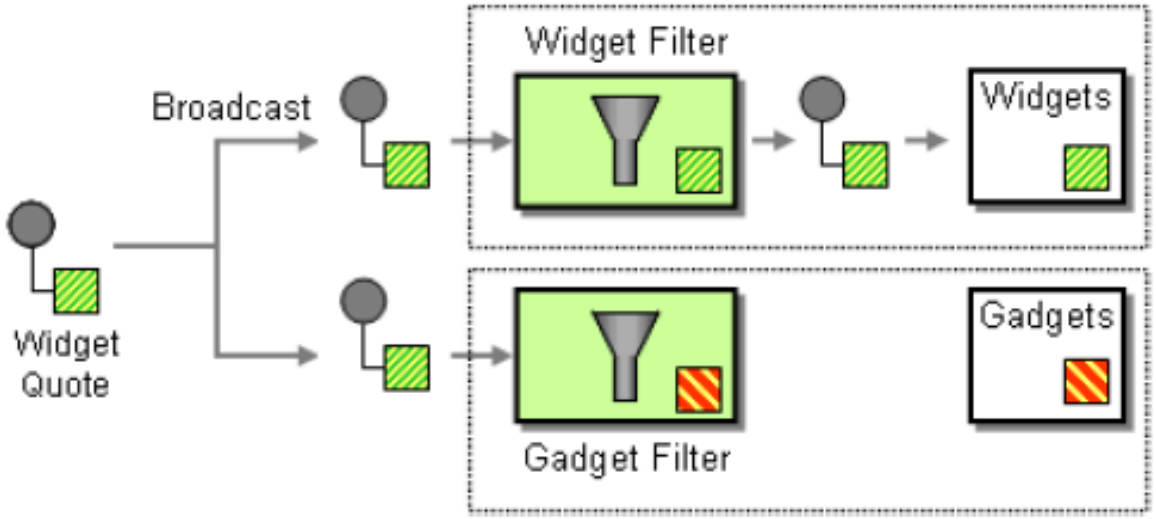
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

2 equivalent functionalities (1/2):

(1)



(2)



Message Filter pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

2 equivalent functionalities (2/2):

(1) Content-Based Router	(2) Publish-Subscribe Channel + Message Filters
Exactly one consumer receives each message	**More than one consumers** can consume each message
Centralized routing	**Disributed** routing
Predictive routing	**Reactive filtering**
Receivers (and their adding/removing) **have to be known**	**No knowledge of consumers** is required
Common use for **business functions** (e.g., for taking orders)	Common use for **event notifications** and **informal messages**
Efficient with **queue-based channels**	Efficient with **Publish-Subscribe Channel** **(like, e.g., IP multicast)**

Message Filter pattern X.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Related patterns: Content-Based Router, Message Router, Selective Consumer, Publish-Subscribe Channel, Recipient List

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Dynamic Router pattern

Dynamic Router pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: A **Message Router** is used to route messages between **multiple destinations**.

Problem: How can you avoid the dependency of the router on all possible destinations while maintaining its efficiency?

Message Router: **efficient**, routes a message directly to the **correct destination**. However, has to **know about all destinations and the routing rules**. In case of frequent change its **maintenance is hard**.

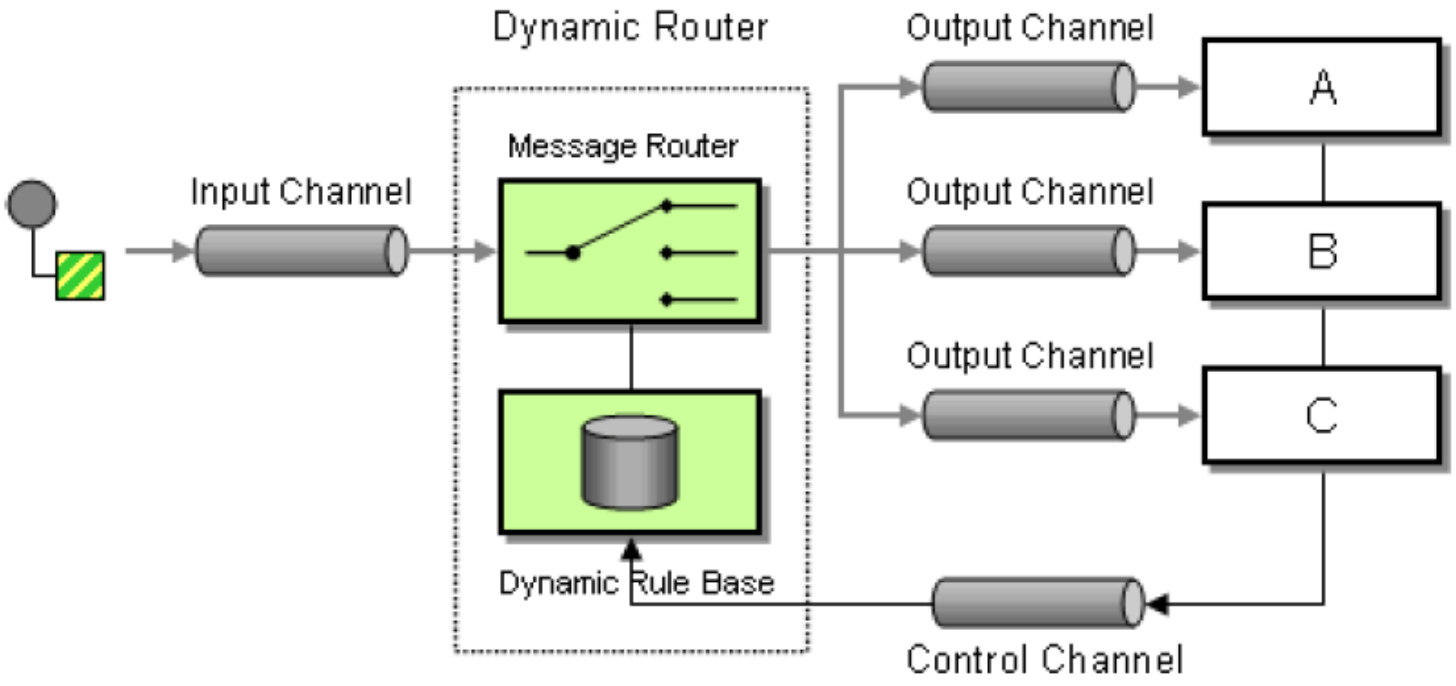
Routing Slip – as a **reactive filtering** solution – uses a **trial-and-error** approach, that is **less efficient**
(routes the message to the first destination, if it is not the correct destination then it passes the message on the next destination)

Distributed routing solutions – like the ***Message Filters*** (they realize **reactive filtering**, too) – **can cause multiple or none consumer** of the message that remains **undetected**.

Dynamic Router pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution</u>: Use a *Dynamic Router*, a Router that can self-configure based on special configuration messages from participating destinations.



Dynamic Router pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A Dynamic Router has not only usual input and output channels, but one **control channel**, too. During **system start-up**, the **consumers announce their presence and preferences** in a control message via this control channel. This information is **stored in a rule base**. It makes flexible, efficient predictive routing without hard maintenance possible.

To do it,

(1) the consumers **have to know** about the control channel of the dynamic router, and

(2) storing the rules has to happen in a **persistent** way **(e.g., for the cases of rouer restarts)**.

A robust alternative for (2) a **broadcast message from the router to trigger the sending of control messages** is **(it requires an additional Publish-Subscribe Channel)**.

Dynamic Router pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A possible **addition**:

Consumers can send **not only 'subscribe' but 'unsubscribe' control messages**, too, during runtime.

Because of the **individual consumers**, **conflict of rules** can happen. Possible ways to resolve them are:

- **Ignoration of control messages that conflict with existing ones**.
  +: routing table is **free of confict**
  -: **unpredictable rule set** because the sequence of control messages (consumer start-ups) matters

- Routes the message **to the first consumer whose preferences match**. This solution **lets rule conflicts inside the rule base**.

- Sends the message **to all consumers whose criteria match**. This behaviour is **tolerant to conflicts**, but **realizes Recipient List** instead of a Content-Based Router **(where there is only one output message for each input message).**

Dynamic Router pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Dynamic Routers are **complex** solutions, moreover, it is **difficult to debug** dynamically configured systems.

Dynamic Routers operate **similar to the dynamic routing tables of IP routing**. The protocol that the consumers use **for the control messages** is analogous to the IP **Routing Information Protocol** (RIP).

Dynamic Service Discovery in SOA is a common usage of Dynamic Router: **Services register their name and the channel** they listen on into the **service directory**. This way, a client application can use only a single channel, without the need for the knowledge of the location of the desired service.

Dynamic Router pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: C# and MSMQ (1/4)

```
class DynamicRouter
{
    protected MessageQueue inQueue;
    protected MessageQueue controlQueue;
    protected MessageQueue dunnoQueue;

    protected IDictionary routingTable = (IDictionary)(new Hashtable());


    public DynamicRouter(MessageQueue inQueue, MessageQueue controlQueue, MessageQueue
dunnoQueue)
    {
        this.inQueue = inQueue;
        this.controlQueue = controlQueue;
        this.dunnoQueue = dunnoQueue;

        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();

        controlQueue.ReceiveCompleted += new
ReceiveCompletedEventHandler(OnControlMessage);
        controlQueue.BeginReceive();
    }
```

The control message has the format „x:QueueName”, requesting the Dynamic Router to route all input messages whose body starts with character „x” to the channel „QueueName”.

Dynamic Router pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: C# and MSMQ (2/4)

```
protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
{
    MessageQueue mq = (MessageQueue)source;
    mq.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
    Message message = mq.EndReceive(asyncResult.AsyncResult);

    String key = ((String)message.Body).Substring(0, 1);

    if (routingTable.Contains(key))
    {
        MessageQueue destination = (MessageQueue)routingTable[key];
        destination.Send(message);
    }
    else
        dunnoQueue.Send(message);
    mq.BeginReceive();
}
```

acts as an Invalid Message Channel

Dynamic Router pattern VIII.

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: C# and MSMQ (3/4)

```
   protected void OnControlMessage(Object source, ReceiveCompletedEventArgs
asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        mq.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        String text = ((String)message.Body);
        String [] split = (text.Split(new char[] {':'}, 2));
        if (split.Length == 2)
        {
            String key = split[0];
            String queueName = split[1];
            MessageQueue queue = FindQueue(queueName);
            routingTable.Add(key, queue);
        }
        else
        {
            dunnoQueue.Send(message);
        }
        mq.BeginReceive();
    }
```

```
    protected MessageQueue FindQueue(string queueName)
    {
        if (!MessageQueue.Exists(queueName))
        {
            return MessageQueue.Create(queueName);
        }
        else
            return new MessageQueue(queueName);
    }
}
```

Dynamic Router pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: C# and MSMQ (4/4)

The presented example **resolves conflicts** of rules in the way, that **the last interested recipient will get the message**: the hashmap stores only one queue for each key value.

Two types of messages are routed to „**dunnoQueue**”:

- incoming messages without matching routing rule
- wrongly formatted control messages

Related patterns: Content-Based Router, Message Filter, Message Router, Publish-Subscribe Channel, Recipient List, Routing Slip

Recipient List pattern

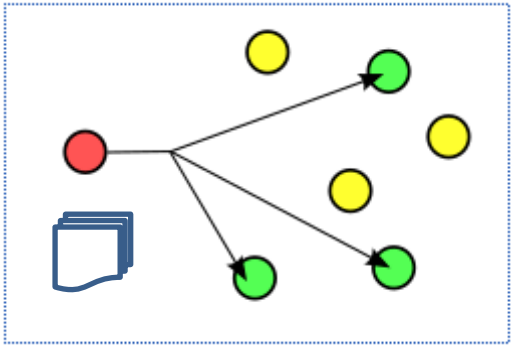
Recipient List pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: Content-Based Router routes the incoming message to the correct recipient based on the content of the message. However, in some cases **the sender intends to specify one or more recipients** for the message. **(e.g., the data of the requestor of small orders are sent to one credit agency, while that of large ordes are sent to more agencies parallel, for the sake of a more reliable evaluation. So, the list of recipients depends on the value of the order. Similar case are the multiple quotation requests for a product.)**

Problem: How do we route a message to a list of dynamically specified recipients?

Since the desired operation is based on that of a **Content-Based Router**, the **forces and alternatives are similar** for the ones of that pattern.



Recipient List pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

(1) Publish-Subscribe Channel + Message Filter/Selective Consumer

Subscribers of a Publish-Subscribe Channel can not change per message. So, if a **Publish-Subscribe Channel** is applied, receivers should use either **Message Filter or Selective Consumer** to filter the incoming messages based on their content. This solution **delegates the logic over who receives the message to the** individual **subscribers**. The maintenance of the **recipient list** can be **centrally controlled if** the list is **part of the message**.

Disadvantages of this solution:

- **inefficient**: requires **each potential recipient to process** the message **(even if the message will be discarded)**
- **unsecure**: no recipient can be prevented from porcessing the message

Recipient List pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

(2) The message originator sends the message to each desired recipient, individually

This solution requires both the **task of delivery** to all recipients and the **decision logic** to be **in the originator application**.

Disadvantages of this solution:

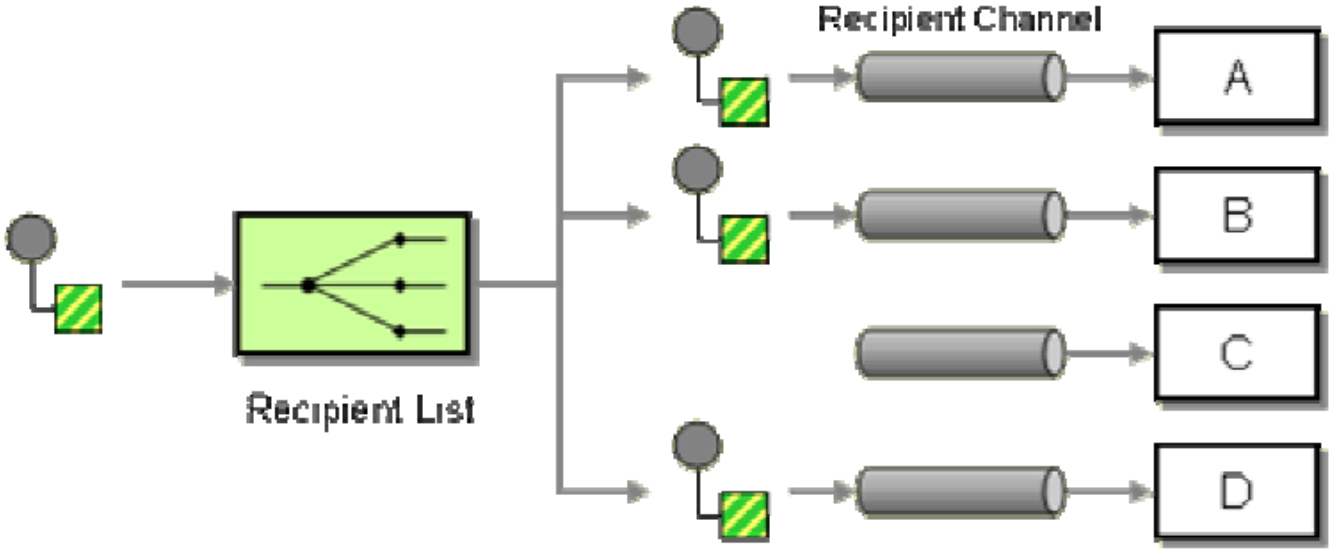
- causes **tight coupling** of the originator application and the integration infrastructure

- the **modification of the applications** – even if they are 3rd party applications – may be **hard or impossible**

Recipient List pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution</u>: Define a channel for each recipient. Then use a *Recipient List* to inspect an incoming message, determine the list of desired recipients, and forward the message to all channels associated with the recipients in the list.



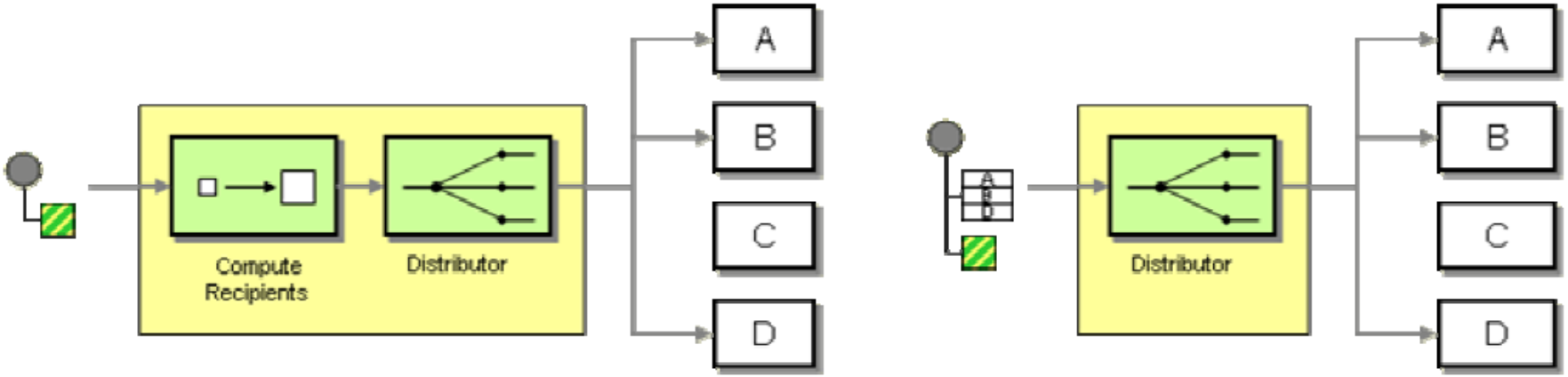
Recipient List pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The **2 parts of the logic** embedded into Recipient List are:

(1) **computes or gets** from somewhere **a list of recipients**

(2) processes the list and **sends a copy of the** incoming **message to all** of the recipients in the list, usually without modifying the message content



Recipient List pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

(1) If the origin of the list of the recipients is external to the Recipient List **(e.g., another application's user selection for each message)**, it usually **attaches the list** to the incoming message. The **Recipient List usually removes** the list from the message, because:

- it **reduces the size** of the outgoing message
- **prevent the recipients to see each other** in the list.

(2) Often the Recipient List computes the list of recipients (based on the **content of the incoming message** and the – hard-coded or configurable – **rules** inside the component)**.**

Predictive routing causes **tight coupling**, because it requires the center **having knowledge about the recipients**.

If we intend Recipient List **to have control** of routing, we **have to ensure** that **recipients cannot subscribe directly** – bypassing the Recipient List - **to the input channel** of the Recipient List.

Recipient List pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Robustness of a Recipient List (1/2)

To ensure all the desired recipients to get the message, Recipient List can **consume** that **only after** that is **sent successfully to all** of the desired recipients. How can a Recipient List provide **robustness** taking into account **possible failures**?

(1) Single transaction: applies **transactional channel**, that **does not commit messages till all the messages are placed on the channels**.So, either all or no messages are sent.

(2) Persistent recipient list: **remembers the sent messages**. The list of recipient is stored **on disk or in a database** and messages can be **sent to remaining recipients in case of a crash**.

Recipient List pattern VIII.

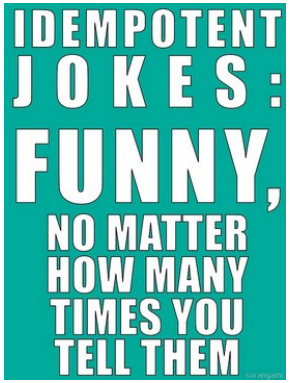
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Robustness of a Recipient List (2/2)

(3) Idempotent receivers: Recipient List - in case of restart - sends the messages again to all the desired reveivers, but they **are not sensitive for possible duplications**, because:

- messages are idempotent: **does not harm if** they are **duplicated**, or
- receivers are idempotent: have **Message Filter for eliminating duplications**

Idempotence is useful, since it **allows to resend** messages if we do not know whether the receiver got it, **without causing any problem (TCP/IP applies something similar).**

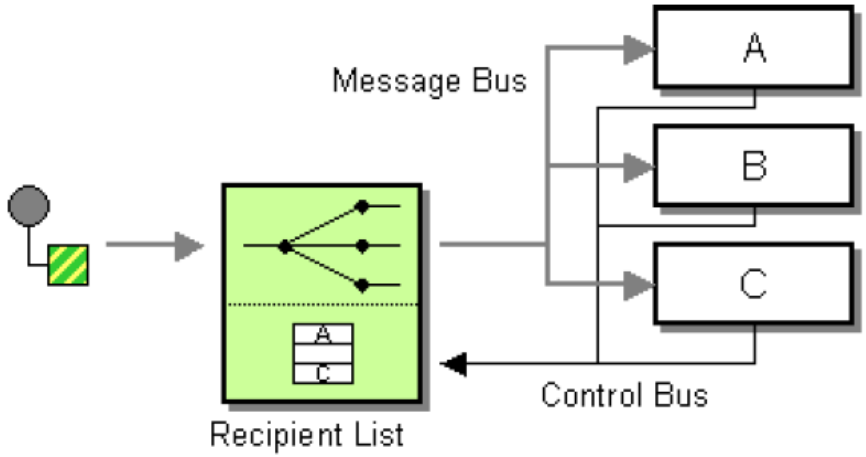


Recipient List pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Dynamic Recipient List

In some cases – contrary to the centralized Recipient List – we intend to **let recipients to control the rule set of the Recipient List**. **E.g., if they want to subscribe to special sets of messages but to build a hierarchy of Publish-Subsribe Channels is not an option.**
For minimizing network traffic, the **decision logic should remain in the Recipient List**.



It is a **combination of a Dynamic Router and a Recipient List**.

Recipients use a **control channel** to modify the rule set of the Recipient List.

Recipient List pattern X.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Network Efficiency

Individual messages to the desired recipients

vs.

One message to all recipients who can filter messages

The efficiency often depends on the messaging **infrastucture**:

One can think the more the desired recipients are the more the network traffic is caused. However, **some multicasting solutions (IP Multicast) route messages to multiple receivers with a single network transmission (where retransmission is needed only for the lost messages).**
IP Multicast is based on the broadcast channel property of an Ethernet segment and its bus architecture. However, it works only for local networks and not for the whole Internet with point-to-point TCP/IP connections.
So, **for further recipients Recipient List is usually more effective than Publish-Subsribe Channel**.

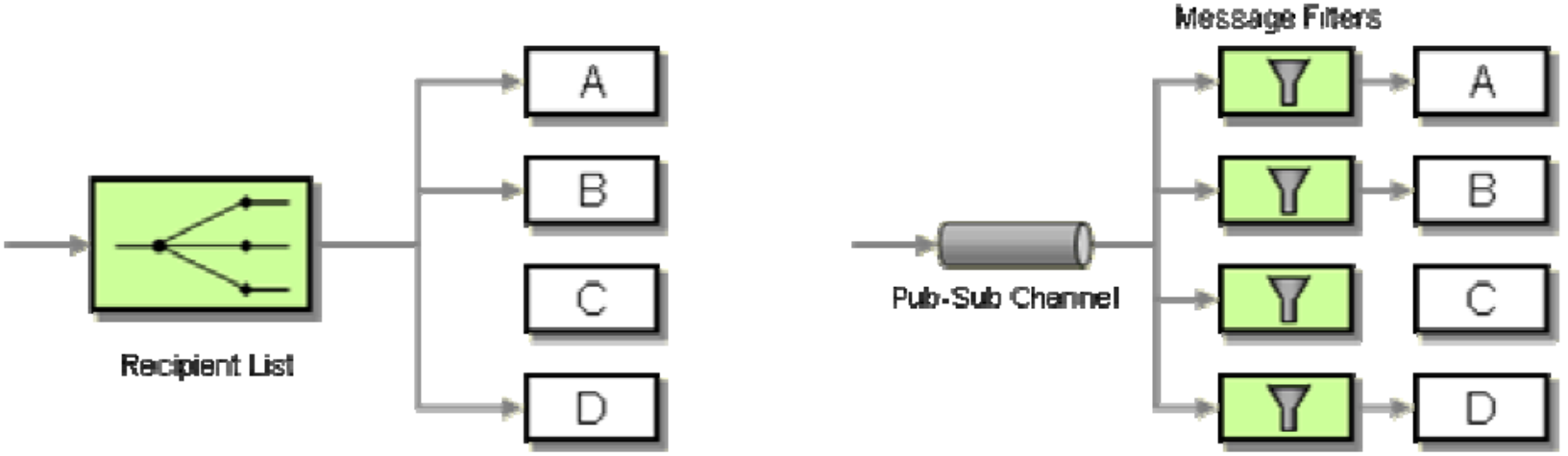
Moreover, **the more percentage of the desired reveivers from the receivers is, the more effective to use message broadcast and filters**.

Recipient List pattern XI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Recipient List or **Publish-Subscribe Channel with Message Filters? (1/2)**

This decision is very **similar to Content-Based Router vs. Message Filters**.
However, in case of Recipient List - because of the **multiple receivers** - **filters seem more attractive** to apply.



Recipient List pattern XII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Recipient List or **Publish-Subscribe Channel with Message Filters? (2/2)**

Recipient List	Publish-Subscribe Channel + Message Filters
Central control and maintenance	**Disributed** control and maintenance
Predictive routing	**Reactive filtering**
Receivers (and their adding/removing) **have to be known** (excepting the Dynamic Recipient List, at expence of losing control)	**No knowledge of consumers** is required
Common use for **business functions** (e.g., for requesting quotes)	Common use for **event notifications** and **informal messages**
Efficient with **queue-based channels**	Efficient with **Publish-Subscribe Channel** **(like, e.g., IP multicast)**

Recipient List pattern XIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

If there are multiple receivers, there are situations that require the **results to handle all in one (e.g., to compare them and to select the best one)**. For better message throughput **we may take the first** available response. These strategies are typically implemented in the component called ***Aggregator***.

Scatter-Gather sends a single message to multiple receivers and **recombines their responses into a single message**.

Dynamic Recipient List is an appropriate tool to **realize a Publish-Subscribe Channel** in an environment **where only Point-to-Point Channels are available**.

Dynamic Recipient List is also a good component **if special criteria have to be applied for allowing recipients to subscribe** - if the messaging system is able to ensure that **recipients don't have direct access to the input channel** of the Recipient List.

Recipient List pattern XIV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Dynamic Recipient List in C# and MSMQ (1/4)

```
class DynamicRecipientList
{
    protected MessageQueue inQueue;
    protected MessageQueue controlQueue;

    protected IDictionary routingTable = (IDictionary)(new Hashtable());

   public DynamicRecipientList(MessageQueue inQueue, MessageQueue controlQueue)
   {
       this.inQueue = inQueue;
       this.controlQueue = controlQueue;

       inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
       inQueue.BeginReceive();

       controlQueue.ReceiveCompleted += new
ReceiveCompletedEventHandler(OnControlMessage);
       controlQueue.BeginReceive();
   }
```

The control message has the format „xyz:QueueName”, requesting the Dynamic Router to route all input messages whose body starts either with character „x” or with character „y” or with character „z” to the channel „QueueName”.

Recipient List pattern XV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Dynamic Recipient List in C# and MSMQ (2/4)

```
   protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
   {
      MessageQueue mq = (MessageQueue)source;
      mq.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
      Message message = mq.EndReceive(asyncResult.AsyncResult);

      if (((String)message.Body).Length > 0)
      {
         char key = ((String)message.Body)[0];

         ArrayList destinations  = (ArrayList)routingTable[key];
         foreach (MessageQueue destination in destinations)
         {
            destination.Send(message);
            Console.WriteLine("sending message " + message.Body + " to " +
destination.Path);
         }
      }
      mq.BeginReceive();
   }
```

Recipient List pattern XVI.

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Dynamic Recipient List in C# and MSMQ (3/4)

```
   protected void OnControlMessage(Object source, ReceiveCompletedEventArgs
asyncResult)
   {
      MessageQueue mq = (MessageQueue)source;
      mq.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
     Message message = mq.EndReceive(asyncResult.AsyncResult);

     String text = ((String)message.Body);
     String [] split = (text.Split(new char[] {':'}, 2));
     if (split.Length == 2)
     {
        char[] keys = split[0].ToCharArray();
        String queueName = split[1];
        MessageQueue queue = FindQueue(queueName);
        foreach (char c in keys)
        {
           if (!routingTable.Contains(c))
           {
              routingTable.Add(c, new ArrayList());
           }
           ((ArrayList)(routingTable[c])).Add(queue);
           Console.WriteLine("Subscribed queue " + queueName + " for message " + c);
        }
     }
     mq.BeginReceive();
  }
```

```
   protected MessageQueue FindQueue(string queueName)
   {
      if (!MessageQueue.Exists(queueName))
      {
         return MessageQueue.Create(queueName);
      }
      else
         return new MessageQueue(queueName);
   }
}
```

Recipient List pattern XVII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Dynamic Recipient List in C# and MSMQ (4/4)

The presented implementation – for optimizing the processing of incoming messages – maintains a **Hashtable keyed by the first character** of the incoming messages **that contains an ArrayList of all subscribed recipients (instead of a single destination).**

A Recipient List typically **doesn't regard it to be an error if there are zero destinations** for a message. That is the reason for **not to apply Invalid Message Channel** for incoming messages that do not match any criteria.

This implementation **does not let recipients to unsubscribe**, moreover, - unlike typical publish-subscribe semantics - **doesn't detect duplicate subsriptions**.

Related patterns: Aggregator, Scatter-Gather, Content-Based Router, Dynamic Router, Message Filter, Idempotent Receiver, Message Router, Selective Consumer, Point-to-Point Channel, Publish-Subscribe Channel

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Splitter pattern

Splitter pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: **One input message** may contain **multiple items** that **require their individual handling** **(e.g. order of items that require to be checked in different inventory systems)**. We should process a complete order, but treat each item of the input message individually.

Problem: How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?

The solution should be generic enough to handle **arbitrary number and type** of elements.

The **control should stay at the component centralized**, **avoiding duplications and lost items** **(so, transmission via a Publish-Subscribe channel letting the recipients to filter the items they can handle is not an option)**. This way **network traffic** remains lower, too.

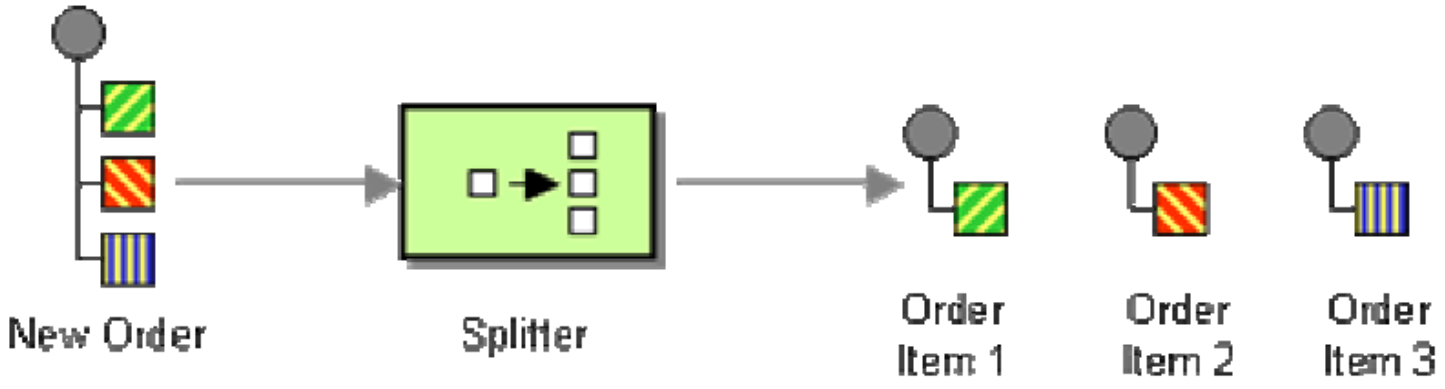
Splitter pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The original input **message could be splitted** into pieces: **each piece should contain items for a well defined destination**. However, this solution **requires the knowledge of the types of the items and their associated destinations**. This way the **change of the routing rules is difficult**.

It is better to apply **Pipes and Filters** architecture and breaking out the processing into well-defined, composable components.

<u>Solution</u>: Use a *Splitter* to break out the composite message into a series of individual messages, each containing data related to one item.

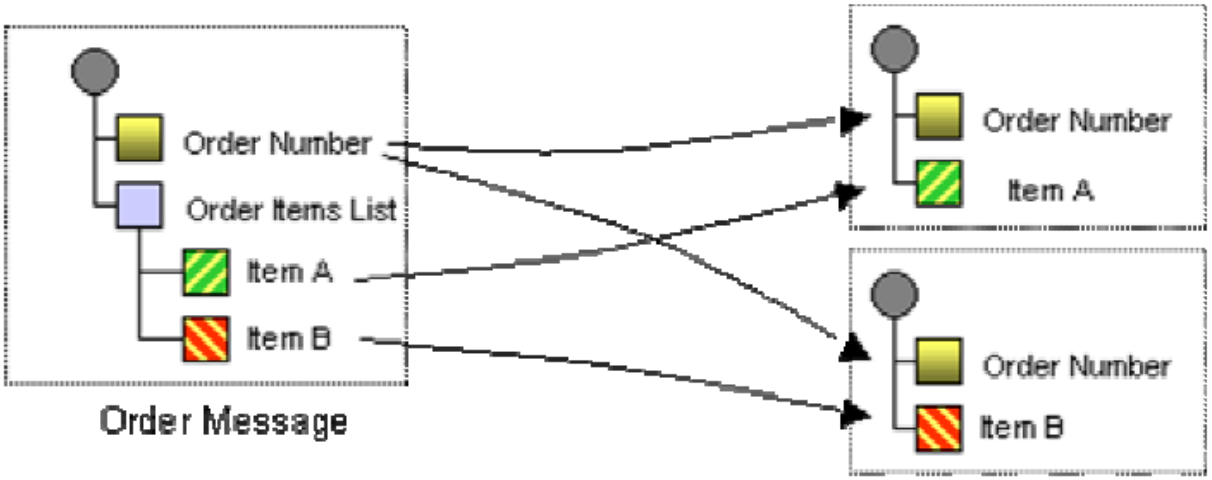


Splitter pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A Splitter **consumes one message** (usually containing a list of repeating elements) and **publishes one or more** outbound message.

Several cases, **each resulting message is supposed to have some common elements**, e.g., the order ID of the original message **(so, the originator – for example - can be determined for each resulted message)**. It results **self-contained child messages** that enable **stateless processing** and makes it possible **to reconciliate the associated results** later.

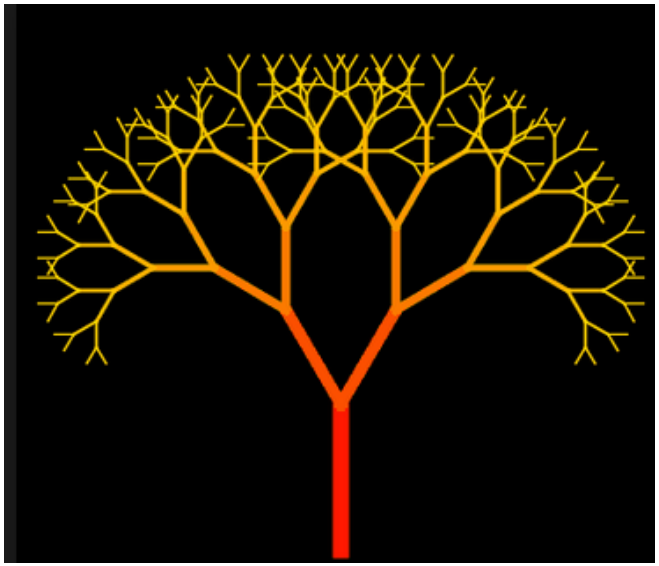


Splitter pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Iterating Splitter

The **content** of a message often follows a **tree structure**, that is a **recursive** data structure.
Iterating Splitter is a Splitter that is configured to **iterate through all children under a specified node and separetes each child node into an individual message**. Some EAI tools call them ***Iterator*** or ***Sequencer***. This component acts general: does not depend on the number and type of the child elements.



Splitter pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Static Splitter (1/2)

There are situations, where **large messages** has to be handled, where each large message has **well-defined structure** **(e.g., a specified comprehensive B2B message format, composed by separate parts that are centered around a specific portion of the large message).**
It is worth to **split** these mega-messages into individual messages **based on the different parts** of them. It results that:

- the **subsequent transformations** become **easier to develop**

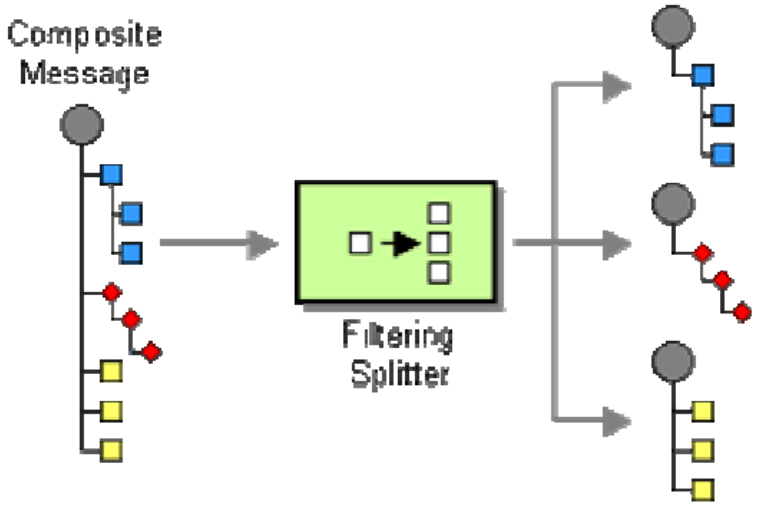
- requires **less network bandwidth** **(smaller messages can be sent to recipients who can handle only that part of the original message)** – the resulting messages (usually with different sub-types) are sent often to **different channels**

Splitter pattern VI.

Static Splitter (2/2)

In contrast to a general Splitter (that assumes a variable number of items) in case of a Static Spitter there is **generally a fixed number of resulting messages**.

A Static Splitter is **functionally equivalent** to a **broadcast channel** followed by a **set of Content Filters**.



Splitter pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Issues related the child messages

If child messages are equipped with **sequence numbers**, it is easier to **trace** them and helps the task of the **Aggregator**, too.

The results of the processing of the **child messages can be correlated to the original message** if the child messages are equipped with a **reference (like a Correlation Identifier) to the original message**.

If **message envelopes** are used **(e.g., a messaging system requires a timestamp in the header of each message)**, then **each child message has to be supplied with** its own message envelope **(e.g., the timestamp of the header of the original message has to be propagated to the header of each child message).**

Splitter pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: splitting an XML document that contains incoming orders (1/9)

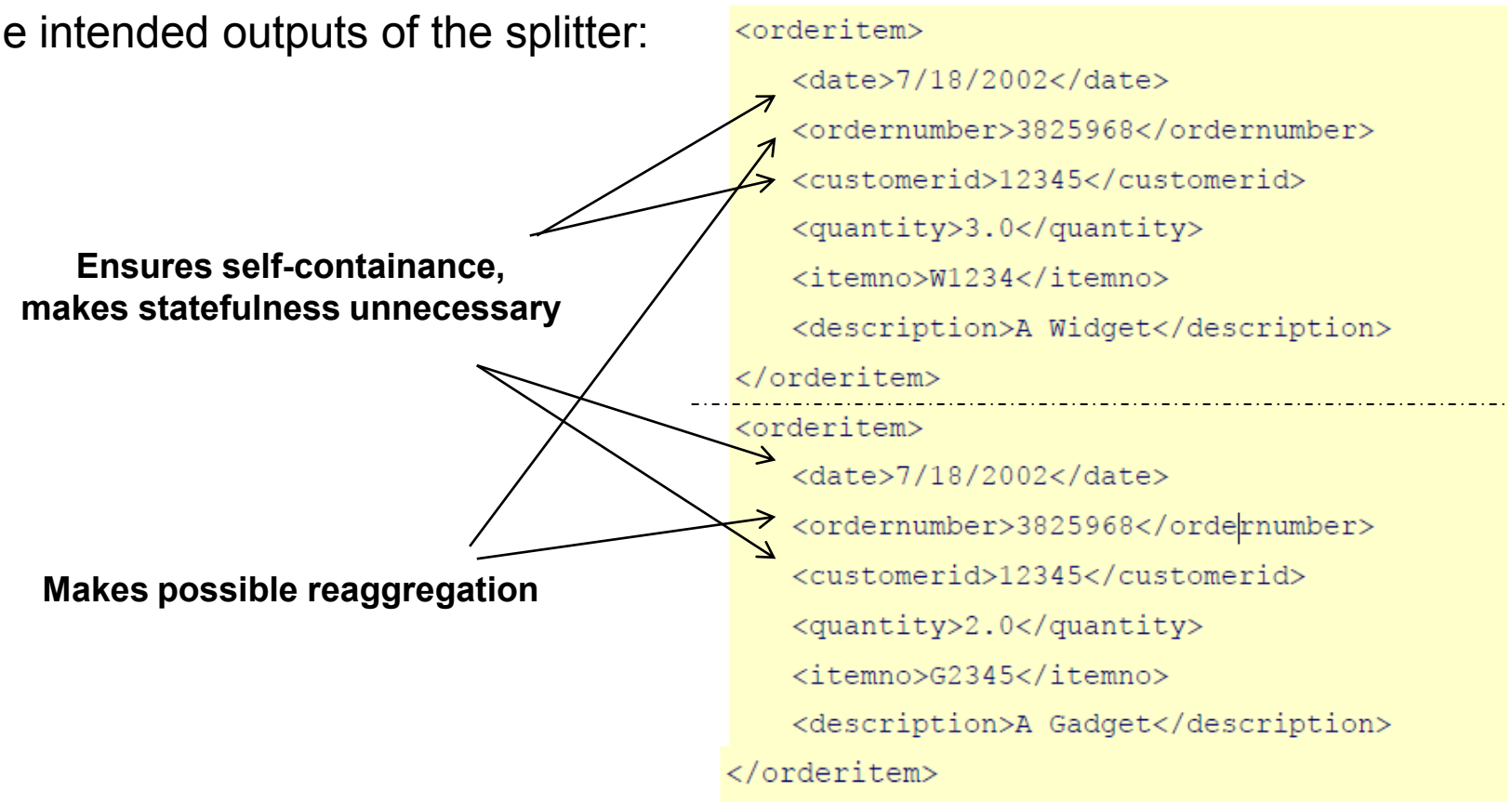
The incoming order:

```
<order>
    <date>7/18/2002</date>
    <ordernumber>3825968</ordernumber>
    <customer>
        <id>12345</id>
        <name>Joe Doe</name>
    </customer>
    <orderitems>
        <item>
            <quantity>3.0</quantity>
            <itemno>W1234</itemno>
            <description>A Widget</description>
        </item>
        <item>
            <quantity>2.0</quantity>
            <itemno>G2345</itemno>
            <description>A Gadget</description>
        </item>
    </orderitems>
</order>
```

Splitter pattern IX.

Example: splitting an XML document that contains incoming orders (2/9)

The intended outputs of the splitter:



No need for specific order item, that's why item number is not used.

Splitter pattern X.

Example: splitting an XML document that contains incoming orders (3/9)

Using **only C#** - The Splitter (1/3): /it also has **Event-Driven Consumer** structure/

```
class XMLSplitter
{
    protected MessageQueue inQueue;
    protected MessageQueue outQueue;

    public XMLSplitter(MessageQueue inQueue, MessageQueue outQueue)
    {
        this.inQueue = inQueue;
        this.outQueue = outQueue;

        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();

        outQueue.Formatter = new ActiveXMessageFormatter();
    }
```

Splitter pattern XI.

Example: splitting an XML document that contains incoming orders (4/9)

Using **only C#** - The Splitter (2/3):

```
protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
{
    MessageQueue mq = (MessageQueue)source;
    mq.Formatter = new ActiveXMessageFormatter();
    Message message = mq.EndReceive(asyncResult.AsyncResult);

    XmlDocument doc = new XmlDocument();
    doc.LoadXml((String)message.Body);

    XmlNodeList nodeList;
    XmlElement root = doc.DocumentElement;

    XmlNode date = root.SelectSingleNode("date");
    XmlNode ordernumber = root.SelectSingleNode("ordernumber");
    XmlNode id = root.SelectSingleNode("customer/id");
    XmlElement customerid = doc.CreateElement("customerid");
    customerid.InnerText = id.InnerXml;
```

converts the message body into an XML document

Splitter pattern XII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: splitting an XML document that contains incoming orders (5/9)

Using **only C#**

-

The Splitter (3/3):

```
        nodeList = root.SelectNodes("/order/orderitems/item");

        foreach (XmlNode item in nodeList)
        {
            XmlDocument orderItemDoc = new XmlDocument();
            orderItemDoc.LoadXml("<orderitem/>");
            XmlElement orderItem = orderItemDoc.DocumentElement;

            orderItem.AppendChild(orderItemDoc.ImportNode(date, true));
            orderItem.AppendChild(orderItemDoc.ImportNode(ordernumber, true));
            orderItem.AppendChild(orderItemDoc.ImportNode(customerid, true));

            for (int i=0; i < item.ChildNodes.Count; i++)
            {
                orderItem.AppendChild(orderItemDoc.ImportNode(item.ChildNodes[i],
true));
            }

            outQueue.Send(orderItem.OuterXml);
        }

        mq.BeginReceive();
    }

}
```

Splitter pattern XIII.

Example: splitting an XML document that contains incoming orders (6/9)

Using **XSL and C#**

The C# code (1/2):

The C# code transforms the input XML by a separate XSL file, after that creates the output messages.

This way in case of format change only the XSL has to be changed without recompiling the C# code.

```
class XSLSplitter
{
    protected MessageQueue inQueue;
    protected MessageQueue outQueue;

    protected String styleSheet = "..\\..\\Order2OrderItem.xsl";
    protected XslTransform xslt;

    public XSLSplitter(MessageQueue inQueue, MessageQueue outQueue)
    {
        this.inQueue = inQueue;
        this.outQueue = outQueue;

        xslt = new XslTransform();
        xslt.Load(styleSheet, null);

        outQueue.Formatter = new ActiveXMessageFormatter();

        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();
    }
```

Splitter pattern XIV.

Example: splitting an XML document that contains incoming orders (7/9)

```
protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
{
    MessageQueue mq = (MessageQueue)source;
    mq.Formatter = new ActiveXMessageFormatter();
    Message message = mq.EndReceive(asyncResult.AsyncResult);

    try
    {
        XPathDocument doc = new XPathDocument(new
StringReader((String)message.Body));

        XmlReader reader = xslt.Transform(doc, null, new XmlUrlResolver());

        XmlDocument allItems = new XmlDocument();
        allItems.Load(reader);

        XmlNodeList nodeList =
allItems.DocumentElement.GetElementsByTagName("orderitem");
```

Using **XSL and C#**

The C# code (2/2)

the Transform method provided by the XslTransform class converts the input document into an intermediate document format

```
        foreach (XmlNode orderItem in nodeList)
        {
            outQueue.Send(orderItem.OuterXml);

        }
    }
    catch (Exception e) { Console.WriteLine(e.ToString()); }
    mq.BeginReceive();
  }
}
```

Splitter pattern XV.

Example: splitting an XML document that contains incoming orders (8/9)

Using **XSL and C#**

The XSL document **/Order2OrderItem.xsl/**:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

   <xsl:template match="/order">
      <orderitems>
         <xsl:apply-templates select="orderitems/item"/>
      </orderitems>
   </xsl:template>

   <xsl:template match="item">
      <orderitem>
         <date>
            <xsl:value-of select="parent::node()/parent::node()/date"/>
         </date>
         <ordernumber>
            <xsl:value-of select="parent::node()/parent::node()/ordernumber"/>
         </ordernumber>
         <customerid>
            <xsl:value-of select="parent::node()/parent::node()/customer/id"/>
         </customerid>
         <xsl:apply-templates select="*"/>
      </orderitem>
   </xsl:template>
```

creates an <orderitems> tag for each <order> tag of the original document

```
  <xsl:template match="*">
     <xsl:copy>
        <xsl:apply-templates select="@* | node()"/>
     </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Splitter pattern XVI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: splitting an XML document that contains incoming orders (9/9)

Comparison of the **preformance** of the two solutions:

Both solutions were feeded by **5000 pieces** of the introduced input message. The time after the 10000. message appeared on the output was:

only C#	XSL + C#
7 sec	5.3 sec

/a simple solution that copied the input message 2 times to the output required 2 sec/

Related patterns: Aggregator, Content-Based Router, Content Filter, Correlation Identifier, Envelope Wrapper, Event-Driven Consumer, Pipes and Filters, Publish-Subscribe Channel

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Aggregator pattern

Aggregator pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: Based on one message **multiple receivers were destinated (e.g., by a Splitter, a Recipient List or a Publish-Subscribe Channel)**. How to do **further processing** if that **depends on the processing of the sub-messages**? **/e.g., we want to select the best offer for the same product, or the client has to be billed after collecting products from the different inventories./**

Problem: How do we combine the results of individual, but related messages so that they can be processed as a whole?

Because messaging is **asynchronous**, it is a **challenging** task **to collect information across multiple messages**. How do we know:

- the **number of the messages**? **(e.g., in case of a broadcast channel the number of responses is usually unknown)**
- the **sequence of the messages**? **(it can be a problem even in case of a Spitter; It is caused by both the different network paths of the sub-message based responses and the possible different processing speed)**

Aggregator pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Usually, messaging infrastructures work in „**guaranteed, ultimately**” mode: guarantees the **reliable delivery but** there is **no guarantee for its timing**.

I**f a message delays**:

- **that can delay the process, too**, or
- moving ahead causes that we have to **work with incomplete information**.

If the missing message arrives, we may:

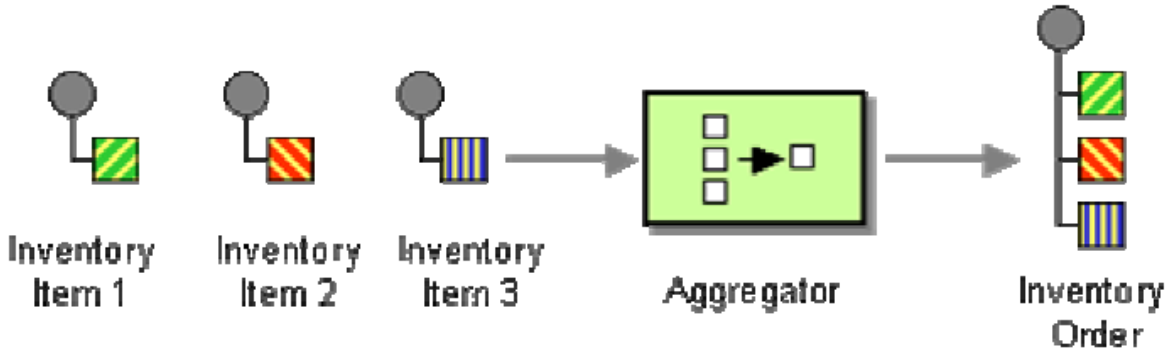
- **process it individually**, but **sometimes** it leads to **duplications**, or
- **ignore that**, however, this case **leads to the loosing its content**.

There should be a **component** who cares these problems and **produces a single messages that depends on the presence of all individual sub-messages.**

Aggregator pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution</u>: Use a stateful filter, an *Aggregator*, to collect and store individual messages until a complete set of related messages has been received. Then, the Aggregator publishes a single message distilled from the individual messages.



An Aggregator works as follows:

Receives a stream of messages,

↘

identifies the correlated messages,

↘

collects information from the correlated messages,

↘

publishes a single message based on the collected information.

Aggregator pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Aggregator is **stateful***:* it has to **store each incoming message until all the messages that belong together have been received**.

However, Aggregator **does not have to store** neceserrily **all the received messages**: sometimes only some (parts) of the incoming messages have to be stored **(e.g., the current best offer and its owner)**, but nevertheless, something has to be stored.

For aggregation we have to know:

(1) Which messages belong together? (**correlation**) **/the decision, e.g., can be based on the type of the message or on an explicit Correlation Identifier/**

(2) How do we know that all the messages have arrived? (**completeness condition**)

(3) How to combine the messages into the one resulted message? (**aggregation algorithm**)

Aggregator pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Because of the properties of messaging, Aggregator **can receive related messages at any time in any order**.

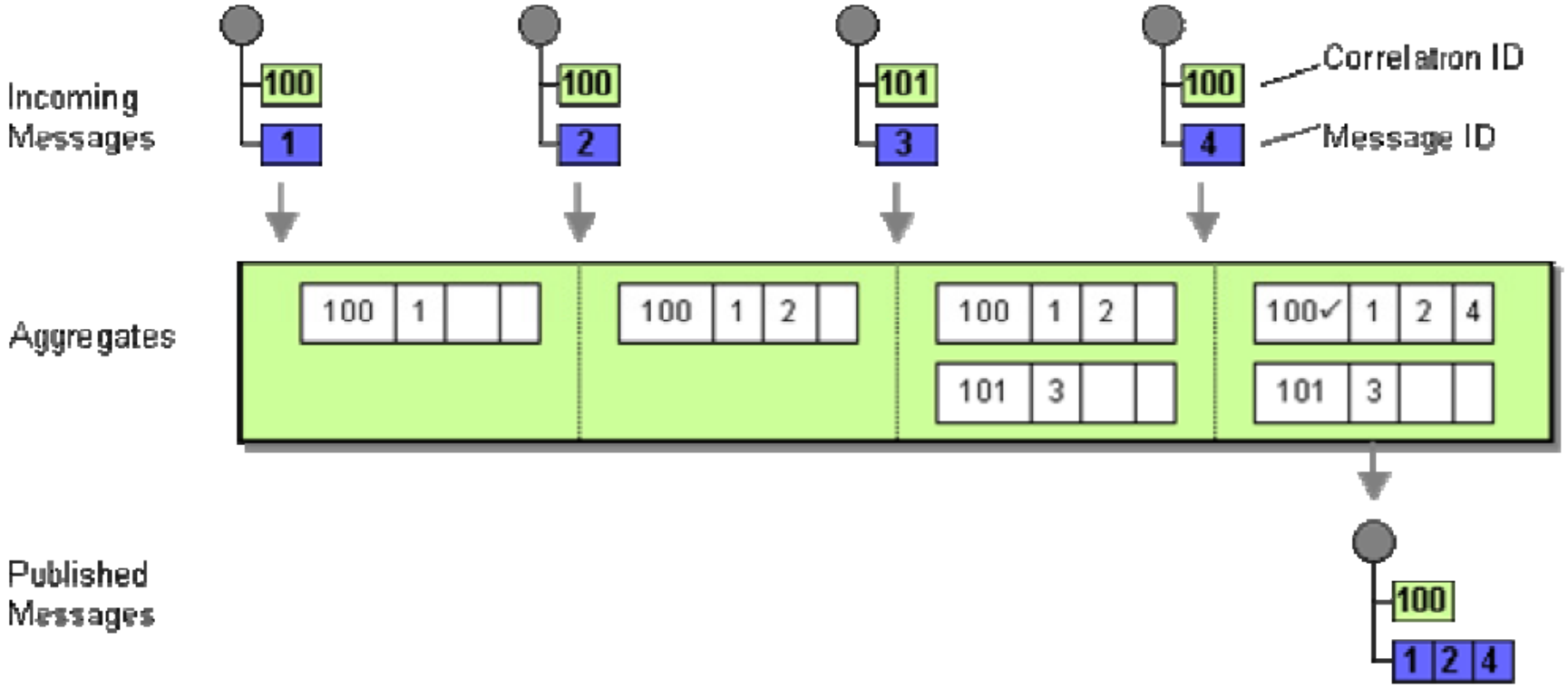
The operation inside an Aggregator:
It keeps a **list of active aggregates** **(aggregates for which it received some messages already)**.
When a **new message** is received:

- **if** the message is **not part of an existing aggregate**: it **starts a new aggregate** and **adds the received message** to that as its first message
- **if** the message is **part of an existing aggregate**: it **adds the message** to the aggregate and **evaluates the completeness condition** for that. If its result is:
  - **true**, then **a new aggregate message is formed and published** onto the output channel
  - **false**, **keeps the aggregate active** for additional messages to arrive.

Aggregator pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

An example, where the completeness condition of an aggregate is to include at least 3 messages:



Aggregator pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Aggregators that starts a new aggregate anytime when getting a message that does not belong to an existing aggregate – so, not making prior knowledge of the aggregates necessary – are **called *Self-starting Aggregators***.

Depending on the aggregation strategy **it can happen that a message arrives to an aggregate that is already closed**. **Avoinding starting a new aggregate** for that message, **a list of closed aggregates** have to maintained **(of course, we do not have to store the whole aggregate that is closed)**. Because of its size, **this list has to be purged periodically**. It **requires to have some assumption about time frame** in which related messages can arrive. **Message Expiration** can be applied for this purpose.

Some Aggregators use **control channel to allow manual purging of all act aggregates or a specific one (useful feature in case of recovering from error without restarting the Aggregator).**

Some Aggregators are able to **publish the list of active aggregates** onto a special channel **upon request (useful for debugging).**

typical features of Control Bus

Aggregator pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Strategies for aggregator **completeness conditions**
/they differ in how much information the Aggregator has about the number of sub-messages to except – this information may exist, for example, because the Aggregator received a copy of the original message or all the sub-messages contain the total count/

(1) „Wait for All”: **wait until all sub-messages are arrived**. The **slowest** and the **least flexible** solution. If not all sub-messages arrive within the **time-limit** period, an **error** occurs **/it can be caused even by a single delayed or missing sub-message/**.

- the **knowledge of the number of expected sub-messages** is required
- it should be predetermined **how long to wait** for the sub-messages
- for **re-requesting** a **missing sub-message**, its **source** has to be known.

Typical example: splitted order handling

Aggregator pattern IX.

(2) „**Time Out**”: wait for a **well-determined duration**, after its expire the **decision is made based on the received sub-messages**. If no sub-message has arrived, either an **exception** is returned **or a retry** happens.

Typical example: selection of the best answer

(3) „**First Best**”: wait for **the fastest response** and ignore the others. This method is **the quickest** but **looses a lot of information**.

Typical example: applications where response time is critical

(4) „**Time Out with Override**”: wait for a **specified duration or** until **a sub-message with a preset minimum score** arrives. If **no required sub-message** has arrived **until the time-limit** is reached, a **rank ordering** is performed among the so far received sub-messages.

Aggregator pattern X.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

(5) „**External Event**”: wait for an **external business event**. It can be, e.g. a **fixed timer** **(it eliminates flexibility)** or an **Event Message** **(that can arrive on a special control channel, for example).**

Typical example: „end of the trading day” signal

Aggregator pattern XI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Strategies for **aggregation algorithm**

(1) „**Select the „Best" Answer**": very **simple** selection criteria and **assumes that there is a single best answer** **(e.g. lowest bid; however, in real life the best bid usually depends on several other factors, like whether the vendor is preferred, the time of delivery, etc.)**

(2) „**Condense Data**": its goal is to **reduce traffic** from a high-traffic source. For example, the outgoing single message can be the **average or the collection** of the data of the multiple incoming messages.

(3) „**Collect Data for Later Evaluation**": **combines** the data of the individual incoming messages **into one single outgoing message**. This case is applied when the **Aggregator can not select** the best message, **but a later component might be able** to do that.

Aggregator pattern XII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Aggregators **often require a parameter (e.g. the maximum wait time, the number of sub-messages, or a specific threshold that a message has to exceed).**
If these parameters are **set at run-time**, it can be done through an additional input **by control messages**. The process is:

When the **first message arrives**, the Aggregator:

starts a new aggregate

AND

receives and stores the parameter information with the aggregate **(it can origin e.g. from the original request message, like in case of Scatter-Gather message).**

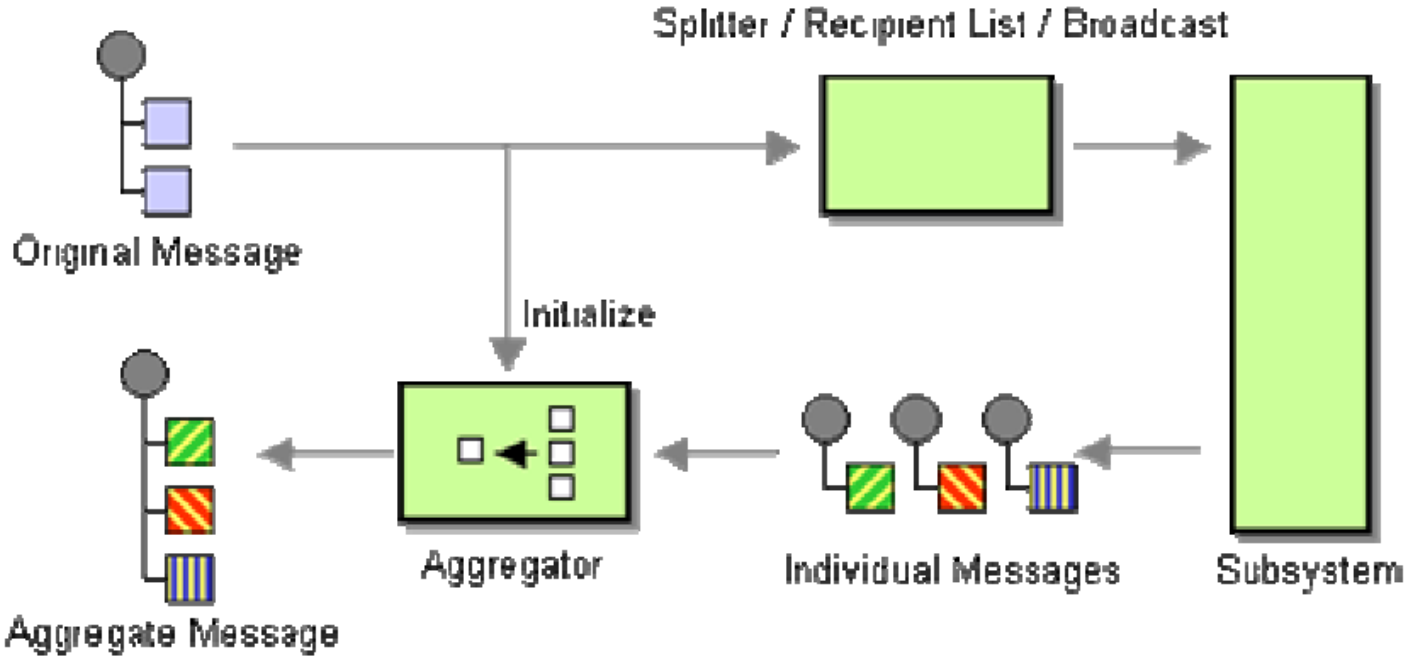
The **further individual messages are associted with the corresponding aggregate**.

This type of Aggregator is called ***Initialized Aggregator***.

Aggregator pattern XIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Initialized Aggregators need to have access to the originating message:

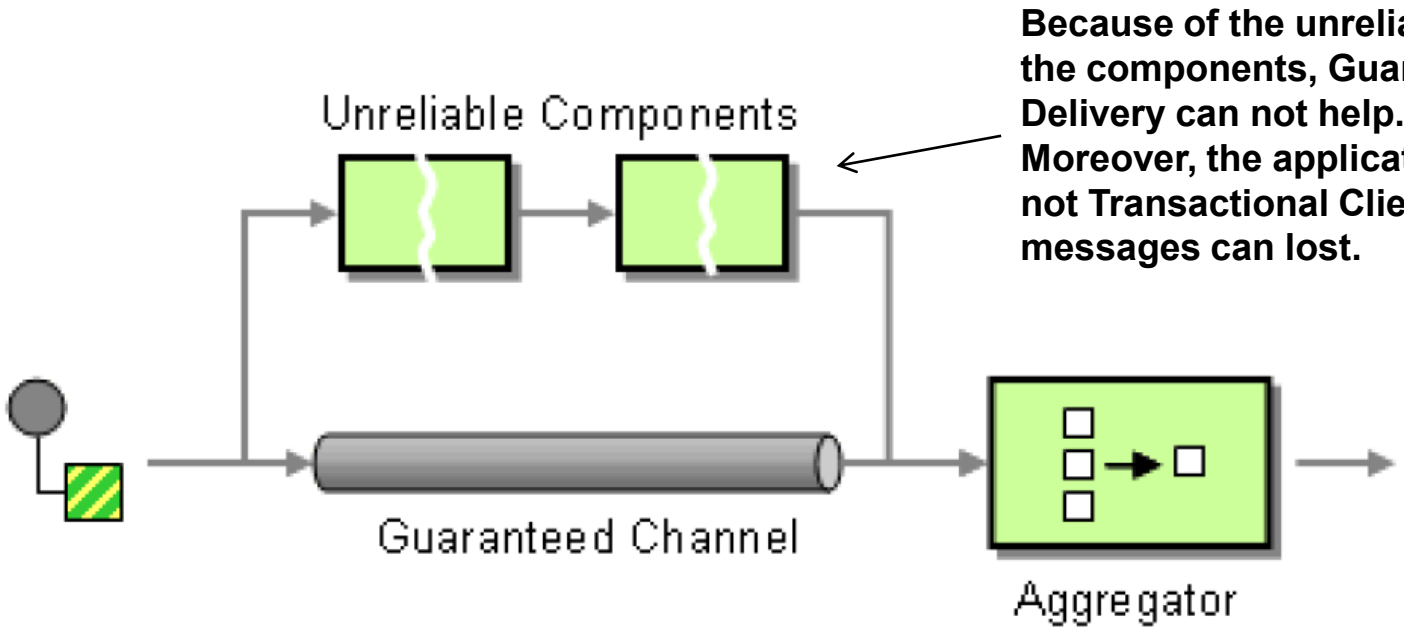


Aggregator is often **coupled with** other component, like **Splitter or Recipient List**.

Aggregator pattern XIV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Detection of missing messages by Aggregator



Because of the unreliability of the components, Guaranteed Delivery can not help. Moreover, the applications are not Transactional Clients, so, messages can lost.

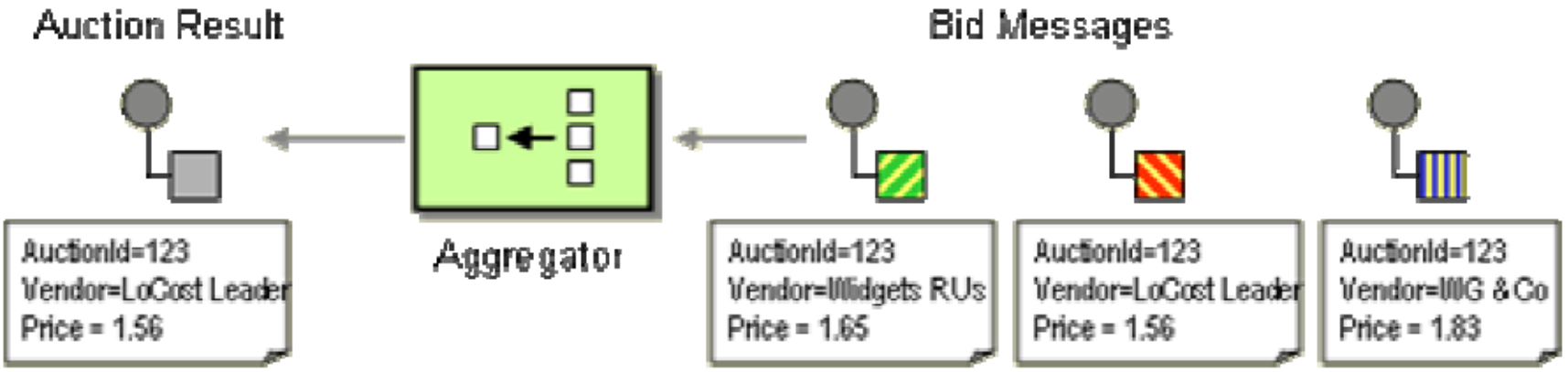
The Aggregator uses „Time Out with Override" completeness condition: waits either till time out occurs or until the two associated messages arrive.
The aggregation algorithm: (1) if both messages are received, the processed message is passed on. (2) In case of timeout, error message is sent, that alerts the operator to restart the failed unreliable component and to resend the lost message.

Aggregator pattern XV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: JMS (1/11)

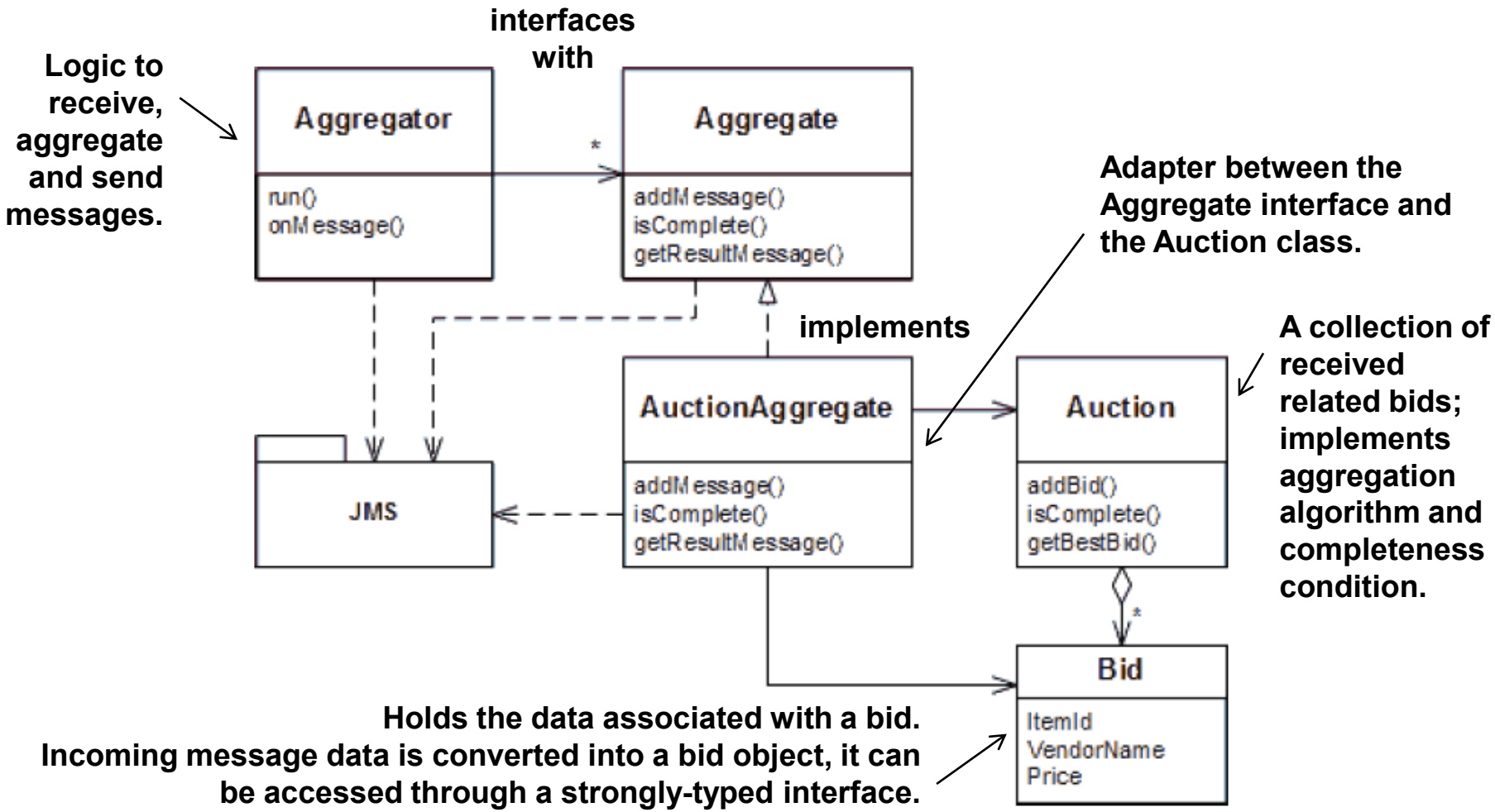
Correlation identifier: **AuctionId**
Completeness condition: receive **minimum 3 bids**
Aggregation algorithm: select the message with **the lowest bid**
The Aggregator is **self-starting** without any external initialization.



Aggregator pattern XVI.

Example: JMS (2/11)

The main classes



Aggregator pattern XVII.

Example: JMS (3/11), Aggregator class 1/3.

```
public class Aggregator implements MessageListener
{
    static final String PROP_CORRID = "AuctionID";

    Map activeAggregates = new HashMap();

    Destination inputDest = null;
    Destination outputDest = null;
    Session session = null;

    MessageConsumer in = null;
    MessageProducer out = null;

    public Aggregator (Destination inputDest, Destination outputDest, Session session)
    {
        this.inputDest = inputDest;
        this.outputDest = outputDest;
        this.session = session;
    }
```

This abstraction allows independence from the type of the channel: e.g. for testing Publish-Subscribe Channel can be used, but for production Point-to-Point Channel can be applied.

Aggregator pattern XVIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: JMS (4/11), Aggregator class 2/3.

```
public void run()
{
    try {

        in = session.createConsumer(inputDest);
        out = session.createProducer(outputDest);
        in.setMessageListener(this);
    } catch (Exception e) {
        System.out.println("Exception occurred: " + e.toString());
    }
}
```

Aggregator pattern XIX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: JMS (5/11), Aggregator class 3/3.

```
public void onMessage(Message msg)
{
    try {
        String correlationID = msg.getStringProperty(PROP_CORRID);
        Aggregate aggregate = (Aggregate)activeAggregates.get(correlationID);
        if (aggregate == null) {
            aggregate = new AuctionAggregate(session);
            activeAggregates.put(correlationID, aggregate);
        }
        //--- ignore message if aggregate is already closed
        if (!aggregate.isComplete()) {
            aggregate.addMessage(msg);
            if (aggregate.isComplete()) {
                MapMessage result = (MapMessage)aggregate.getResultMessage();
                out.send(result);
            }
        }
    } catch (JMSException e) {
        System.out.println("Exception occurred: " + e.toString());
    }
}
}
```

The Aggregator is an Event-Driven Consumer

checks whether an active aggregate exists for this correlation ID

tests whether the termination condition has been fulfilled

It is a very generic Aggregator code.

Aggregator pattern XX.

Example: JMS (6/11)

the Aggregate interface:

```
public interface Aggregate {
    public void addMessage(Message message);
    public boolean isComplete();
    public Message getResultMessage();
}
```

the Auction class (1/2):

```
public class Auction
{
    ArrayList bids = new ArrayList();

    public void addBid(Bid bid)
    {
        bids.add(bid);
        System.out.println(bids.size() + " Bids in auction.");
    }
```

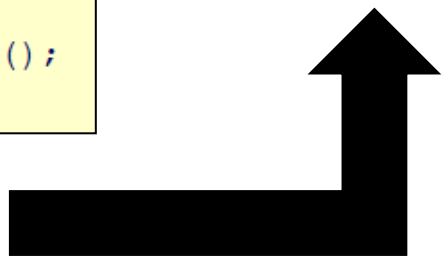
Aggregator pattern XXI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: JMS (7/11)

the Auction class (2/2):

```
public boolean isComplete()
{
    return (bids.size() >= 3);
}

public Bid getBestBid()
{
    Bid bestBid = null;

    Iterator iter = bids.iterator();
    if (iter.hasNext())
        bestBid = (Bid) iter.next();
    while (iter.hasNext()) {
        Bid b = (Bid) iter.next();
        if (b.getPrice() < bestBid.getPrice()) {
            bestBid = b;
        }
    }
    return bestBid;
  }
}
```



This class implements the aggregation strategy.

It provides three methods similar to the Aggregate interface, but their signatures differ: they use the strongly typed Bid class instead of the Message class.

Aggregator pattern XXII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: JMS (8/11), AuctionAggregate class 1/2.

This class is an **Adapter**, so, **converts the interface** of class Auction to the interface Aggregate.

```
public class AuctionAggregate implements Aggregate {
    static String PROP_AUCTIONID = "AuctionID";
    static String ITEMID = "ItemID";
    static String VENDOR = "Vendor";
    static String PRICE = "Price";

    private Session session;
    private Auction auction;

    public AuctionAggregate(Session session)
    {
        this.session = session;
        auction = new Auction();
    }
```

```
public void addMessage(Message message) {
    Bid bid = null;
    if (message instanceof MapMessage) {
        try {
            MapMessage mapmsg = (MapMessage)message;
            String auctionID = mapmsg.getStringProperty(PROP_AUCTIONID);
            String itemID = mapmsg.getString(ITEMID);
            String vendor = mapmsg.getString(VENDOR);
            double price = mapmsg.getDouble(PRICE);
            bid = new Bid(auctionID, itemID, vendor, price);
            auction.addBid(bid);
        } catch (JMSException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Aggregator pattern XXIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: JMS (9/11), AuctionAggregate class 2/2.

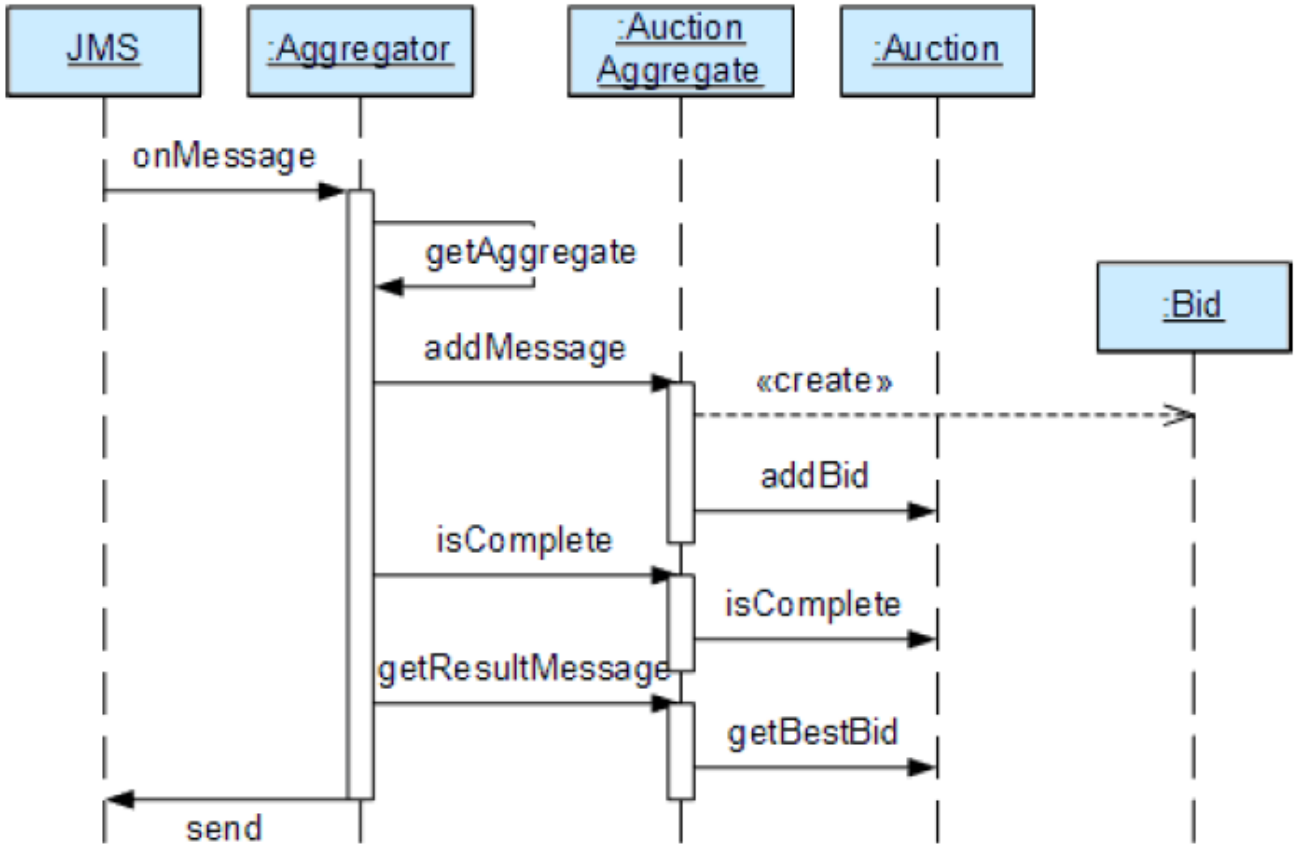
```
    public boolean isComplete()
    {
        return auction.isComplete();
    }

    public Message getResultMessage() {
        Bid bid = auction.getBestBid();
        try {
            MapMessage msg = session.createMapMessage();
            msg.setStringProperty(PROP_AUCTIONID, bid.getCorrelationID());
            msg.setString(ITEMID, bid.getItemID());
            msg.setString(VENDOR, bid.getVendorName());
            msg.setDouble(PRICE, bid.getPrice());
            return msg;
        } catch (JMSException e) {
            System.out.println("Could not create message: " + e.getMessage());
            return null;
        }
     }
}
```

Aggregator pattern XXIV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: JMS (10/11)

The interaction between the classes:



Aggregator pattern XXV.

Example: JMS (11/11)

The example assumes that AuctionID is unique.

We do not have to worry about cleaning up the open auction list.

Although, the example is more likely to use **Point-to-Point Channel** (the **only one receiver** of the bids is the Aggregator), the code references **JMS destinations**, so, it can be **easily applied with Publish-Subscribe Channel, too**. The latter case is perfect for debugging purposes (e.g., by subscribing to topics with name of „*”, if wildcards can be used).

Related patterns: Scatter-Gather, Content-Based Router, Control Bus, Correlation Identifier, Composed Message Processor, Event-Driven Consumer, Event Message, Guaranteed Delivery, Message Expiration, Point-to-Point Channel, Publish-Subscribe Channel, Recipient List, Resequencer, Splitter, Transactional Client

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Resequencer pattern

Resequencer pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

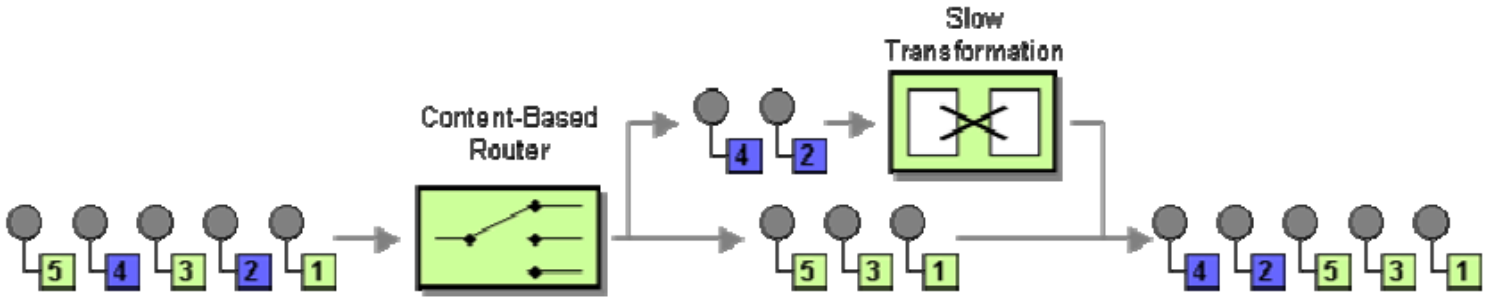
Context: Because of individual routing of messages, the messages can follow **different routes**. It can result in **out-of-sequence** messages. However, some processes require in-sequence messages (e.g., to maintain referential integrity).

Problem: How can we get a stream of related but out-of-sequence messages back into the correct order?

Possible approach:

> **Keeping the messages in sequence**: it is easier than getting them back in order, but hard to realize in case of asynchronous messaging.

Messages can get out of sequence because of **different processing paths**.



Resequencer pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

We can avoid getting the messages out of sequence if we apply a **loop-back (acknowledgement)** mechanism:

The next messages is not handled
until the processing of the previous one has not finished.

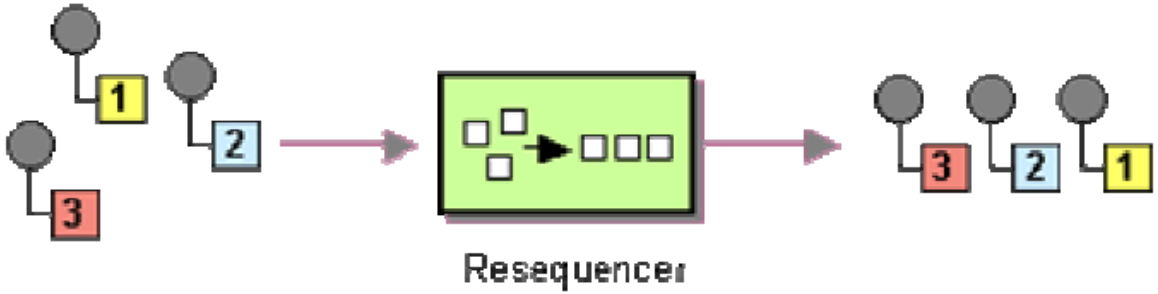
The **drawbacks** of this solution:

- **slow** (eliminates the advantages of parallel processing)
- we **should have control of the message origin** (we are often at the receiving end of an out-of-sequence message flow)

Aggregator solves the out-of-sequence problem by **storing the messages until all related messages arrives**. The result message is published only after that.

Resequencer pattern III.

<u>Solution</u>: Use a stateful filter, a *Resequencer*, to collect and re-order messages so that they can be published to the output channel in a specified order.



A Resequencer can receive a stream of out-of-sequence messages as its input. It stores the out-of-sequence messages in its **internal buffer** until a complete sequence is obtained. Then the in-sequence messages are published to the **output channel**, that **preserves the order**; so it is guarenteed that the next component receives the messages in order.

Resequencer usually **does not modify the content** of a message.

Resequencer pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Sequence Numbers

Resequencer requires each message to have a **unique Sequence Number**.

It **differs both from Message Identifier and from Correlation Identifier**:

The latters are often random values, not in sequence, non comparable, and sometimes even not numbers (but, e.g., time values)

Sequence Number is often part of the **message header**.

The **generation** of Sequence Numbers is not an easy and time consuming task **(and sometimes bottleneck of the system)**:

the numbers **have to be not only in ascending order but to be consecutive**, too. That's why their generation often happens by **a single counter** that assigns number **across the system** **(while unique identifiers can be generated easily in a distributed way, e.g., by combining time and location information).**
In case of Splitter the generation of Sequence Numbers are the best to be built into the Splitter.

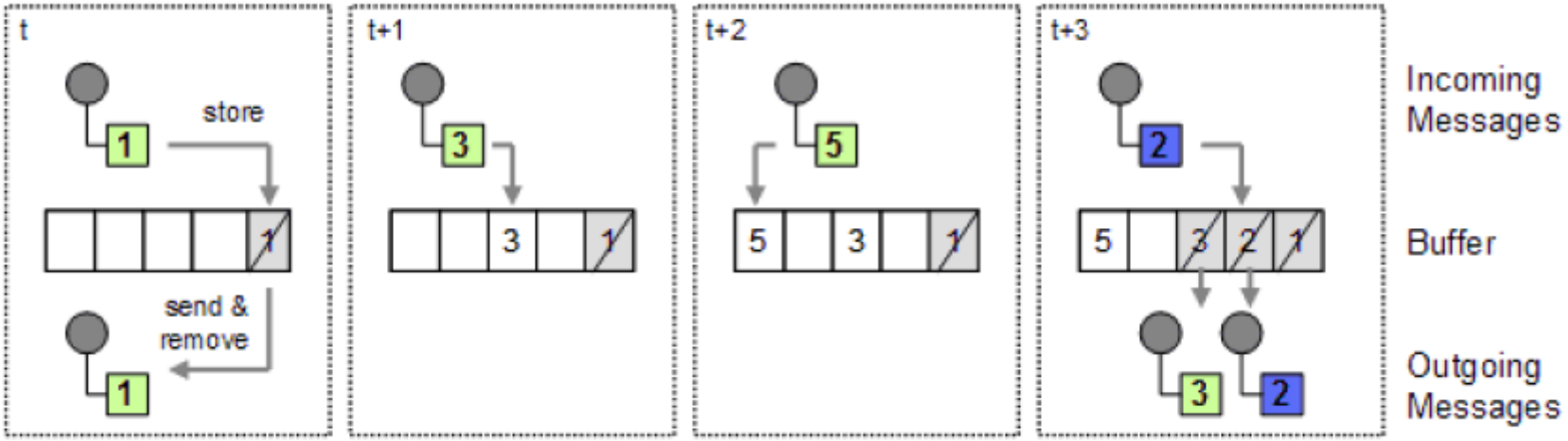
Resequencer pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

How Resequencer operates?

If a message with **a higher sequence number arrives before a message with a lower sequence number**, the Resequencer has to **store** that **until it receives all the "missing" messages** with lower sequence numbers.

When the buffer contains a **consecutive sequence** of messages, they are **sent in-sequence to the output** channel, **and** after that they are **removed from the buffer**.



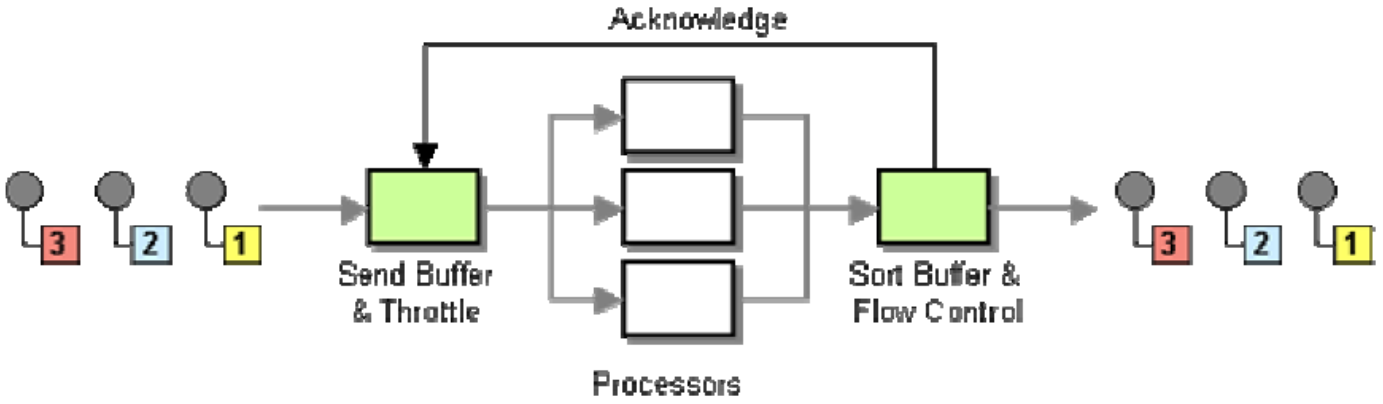
Resequencer pattern VI.

Avoiding buffer overrun (1/2)

Problem can happen, e.g., if either **long message sequences** are applied or **parallel processing units** are used **(for example, if any of them fails that results missing message)**.

Sometimes applying **queue that lets to read messages based on some selection criteria can help** (e.g., reading always the oldest message), however, in case of missing messages it is not a solution against buffer overrun (for example, in the storage of the queue), too.

Throttling the message producer by **active acknowledgement** also can be a solution:



Resequencer pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Avoiding buffer overrun (2/2)

Sending only a single message at a time is unefficient. That's why instead of acknowledging messages one-by-one, **the Resequencer could tell the message producer, how many available slots it has in its buffer**. However, this method requires that we have access to the original in-sequence message stream.

TCP protocol applies similar flow control (including preventing Silly Window Syndrome).

Because IP traffic is asynchronous and unreliable, TCP/IP solutions can serve good patterns for messaging solutions.

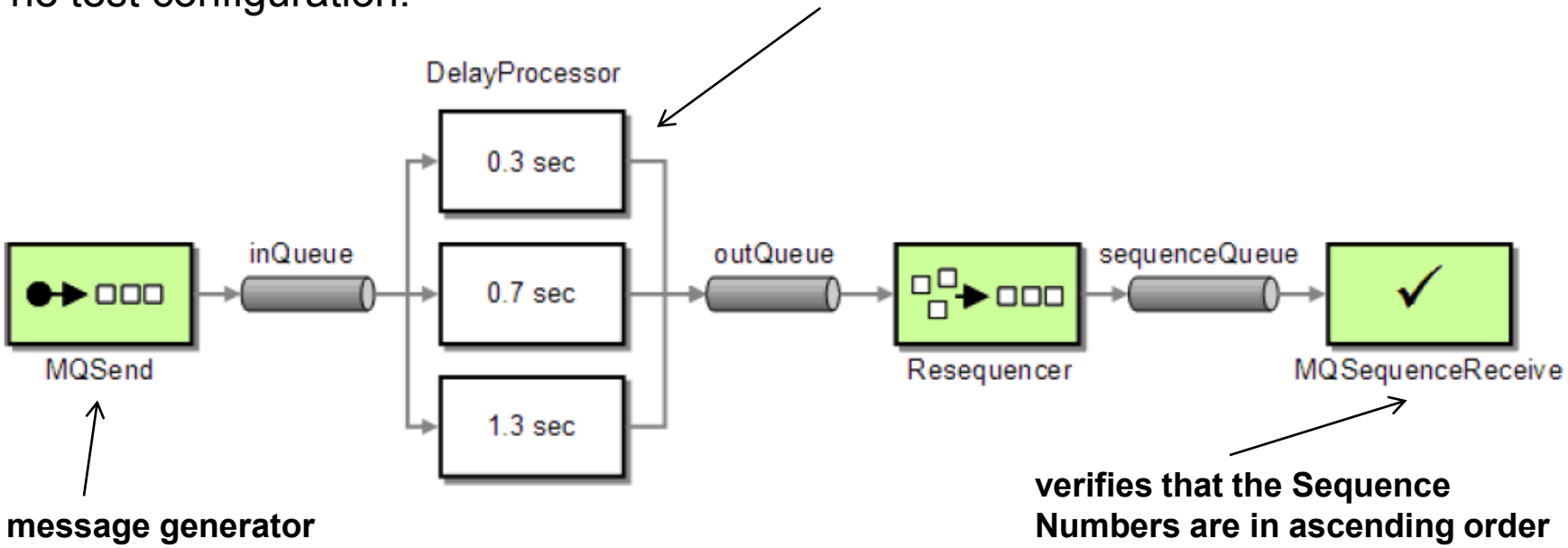
Another solution for buffer underrun is to **compute stand-in messages for the missing ones**. This is quicker but less accurate solution than the re-request (e.g., VoIP applies that).

Resequencer pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: MSMQ, .NET (1/9)

The test configuration:



The queues are MSMQ message queues, provided by the Message queuing service that is part of Windows 2000 and Windows XP.

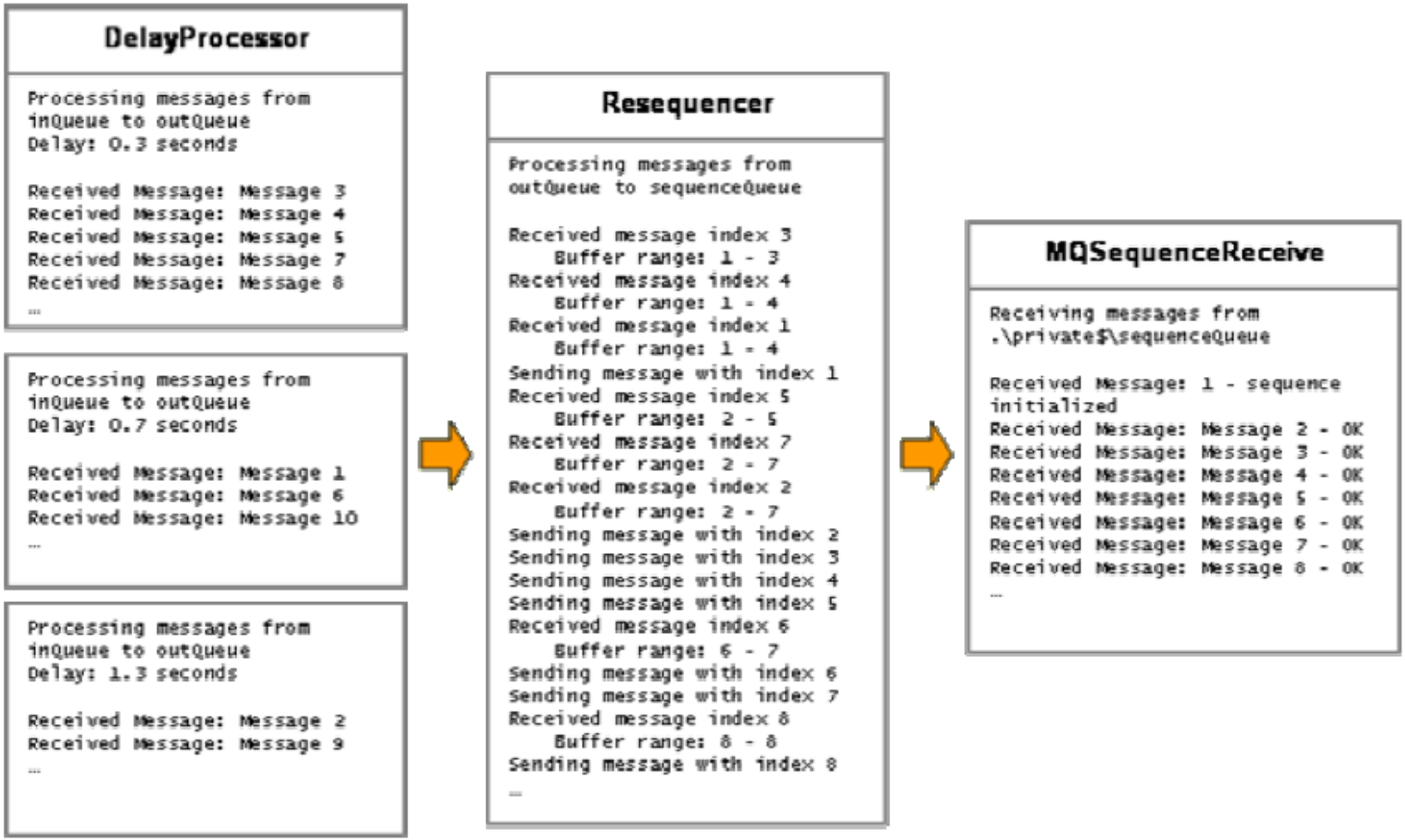
The messages have their Sequence Number in the AppSpecific property. The number of messages can be inputted from command line. The Sequence Numbers start with 1.

Resequencer pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: MSMQ, .NET (2/9)

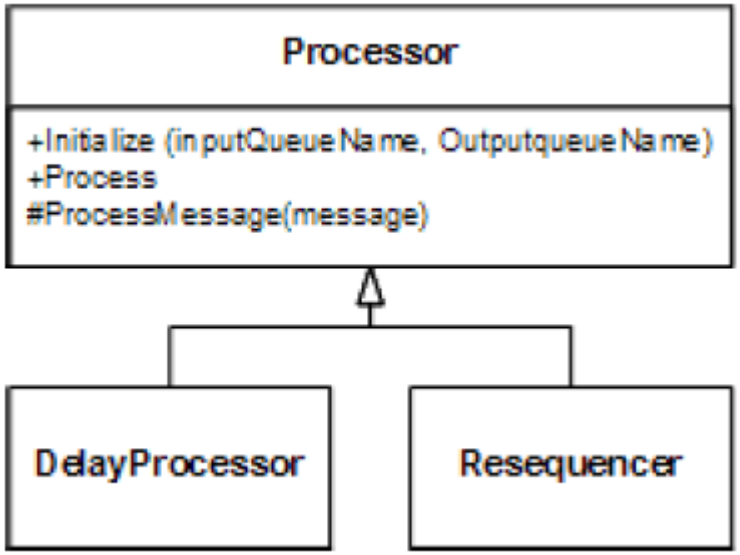
Output from the components of the example

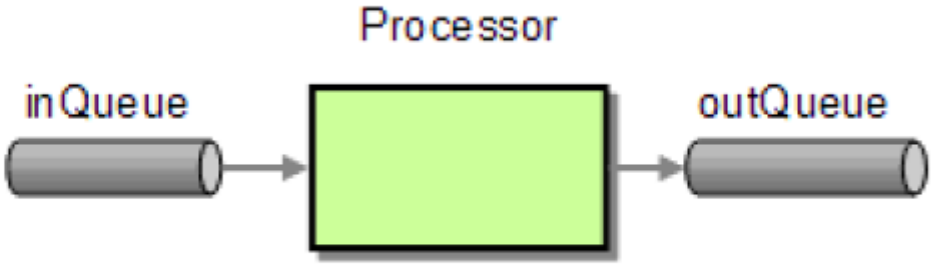


Resequencer pattern X.

Example: MSMQ, .NET (3/9)

In the example solution both DelayProcessor and Resequencer are inherited from the same base class:





The default ProcessMessage method simply copies the messages from the inputQueue to the outputQueue.

Resequencer pattern XI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: MSMQ, .NET (4/9) Code of the Resequencer (1/3)

```
using System;
using System.Messaging;
using System.Collections;
using MsgProcessor;

namespace Resequencer
{

    class Resequencer : Processor
    {
        private int startIndex = 1;
        private IDictionary buffer = (IDictionary)(new Hashtable());
        private int endIndex = -1;

        public Resequencer(MessageQueue inputQueue, MessageQueue outputQueue) : base
(inputQueue, outputQueue) {}

        protected override void ProcessMessage(Message m)
        {
            AddToBuffer(m);
            SendConsecutiveMessages();
        }
```

The Resequencer assumes that the message sequence starts with Sequence Number 1. Moreover, the Resequencer and the message producer have to maintain the same sequence over the lifetime of the components. It could be made more flexible by negotiation about the sequence start number, like in TCP.

the buffer is implemented as a Hashtable

Resequencer pattern XII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: MSMQ, .NET (5/9) Code of the Resequencer (2/3)

```
private void AddToBuffer(Message m)
{
    Int32 msgIndex = m.AppSpecific;
    Console.WriteLine("Received message index {0}", msgIndex);
    if (msgIndex < startIndex)
    {
        Console.WriteLine("Out of range message index! Current start is: {0}",
startIndex);
    }
    else
    {
        buffer.Add(msgIndex, m);
        if (msgIndex > endIndex)
            endIndex = msgIndex;
    }
    Console.WriteLine("    Buffer range: {0} - {1}", startIndex, endIndex);
}
```

the Sequence Number is stored in the AppSpecific property

the message in the buffer is keyed by its Sequence Number

Resequencer pattern XIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: MSMQ, .NET (6/9) Code of the Resequencer (3/3)

This solution does not care with buffer overrun. For more safe implementation the message producer and the Resequencer should negotiate a window size (the maximum number of messages the Resequencer can buffer) and an error handler (how to deal the missing message).

```
        private void SendConsecutiveMessages()
        {
            while (buffer.Contains(startIndex))
            {
                Message m = (Message)(buffer[startIndex]);
                Console.WriteLine("Sending message with index {0}", startIndex);
                outputQueue.Send(m);
                buffer.Remove(startIndex);
                startIndex++;
            }
        }
    }

}
```

Resequencer pattern XIV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: MSMQ, .NET (7/9) Code of the Processor (1/3)

```
using System;
using System.Messaging;
using System.Threading;

namespace MsgProcessor
{
    public class Processor
    {
        protected MessageQueue inputQueue;
         protected MessageQueue outputQueue;


         public Processor (MessageQueue inputQueue, MessageQueue outputQueue)
         {
             this.inputQueue = inputQueue;
             this.outputQueue = outputQueue;
             inputQueue.Formatter = new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});
             inputQueue.MessageReadPropertyFilter.ClearAll();
             inputQueue.MessageReadPropertyFilter.AppSpecific = true;
             inputQueue.MessageReadPropertyFilter.Body = true;
             inputQueue.MessageReadPropertyFilter.CorrelationId = true;
             inputQueue.MessageReadPropertyFilter.Id = true;
             Console.WriteLine("Processing messages from " + inputQueue.Path + " to " +
outputQueue.Path);
         }
```

Resequencer pattern XV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: MSMQ, .NET (8/9) Code of the Processor (2/3)

```
        public void Process()
        {
            inputQueue.ReceiveCompleted += new
ReceiveCompletedEventHandler(OnReceiveCompleted);
            inputQueue.BeginReceive();
        }

        private void OnReceiveCompleted(Object source, ReceiveCompletedEventArgs
asyncResult)
        {
            MessageQueue mq = (MessageQueue)source;

            Message m = mq.EndReceive(asyncResult.AsyncResult);
            m.Formatter =  new System.Messaging.XmlMessageFormatter(new String[]
{"System.String,mscorlib"});

            ProcessMessage(m);

            mq.BeginReceive();
        }
```

Resequencer pattern XVI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: MSMQ, .NET (9/9)

Code of the Processor (3/3)

```
        protected virtual void ProcessMessage(Message m)
        {
            string body = (string)m.Body;
            Console.WriteLine("Received Message: " + body);
            outputQueue.Send(m);
        }
    }
}
```

Related patterns: Aggregator, Competing Consumers, Message Router, Message Sequence, Pipes and Filters, Splitter, Test Message

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Composed Message Processor pattern

Composed Message Processor pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: An incoming message can contain **several individual orders** of different product types. Each order has to be **checked in the appropriate inventory** and the validated order has to be passed to the **next processing step**.

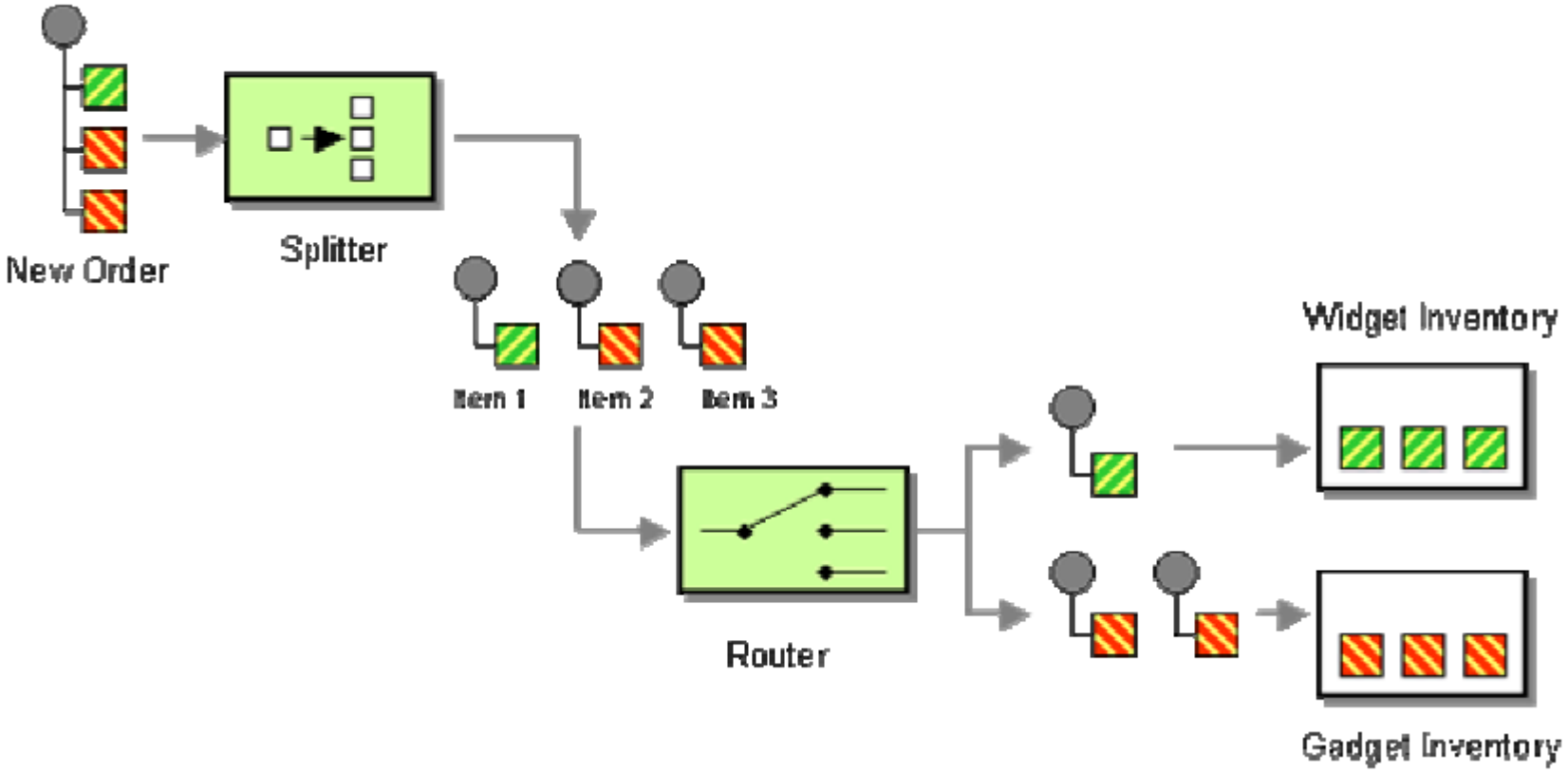
Problem: How you maintain the overall message flow when processing a message consisting of multiple elements, each of which may require different processing?

This problem can be easily solved by the previously presented components:

- a **Splitter** can split the message into multiple parts based on the individual orders
- a **Content-Based Router** can route each individual message to the desired processing step based on the content of the message
- **Pipes and Filters** architectural style makes possible to chain the different components

Composed Message Processor pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



In this solution each inventory gets only orders that can be processed by it.

Composed Message Processor pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The presented solution does not make possible to realize whether all ordered items are in stock. Moreover, **we would need** to know the prices of all ordered items presented in **a single invoice, however, the system split the original order** message into many sub-messages.

Possible solution:

> **Reassemble all orders that pass through a specific inventory** system into a separate order and handle that **as an independent process**. If we do not have control over the downstream process, this is the only available solution **(e.g. Amazon)**. However, this solution results in **more than one shipment and more than one invoice**. If the orders are not independent, problems of some orders may cause bigger problems **(e.g., if some pieces of a furniture order have problem, maybe we would cancel the other parts, too)**.

The asynchronous nature of a messaging system makes **distribution of tasks more complicated** than synchronous method calls.

Composed Message Processor pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

If we handle each order individually, after each other, **sequentially**:

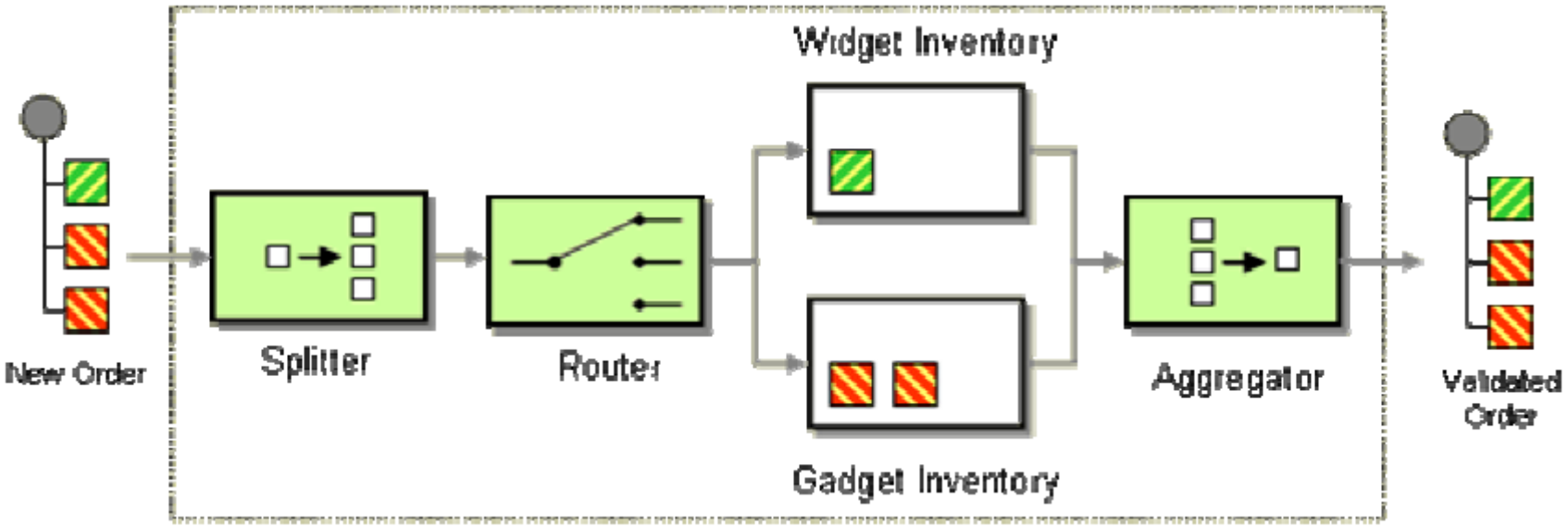
- it **simplifies the temporal dependencies**
- it is **inefficient**.

It would be **useful** to utilize the fact that **each system can process orders simultaneously**.

Composed Message Processor pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution</u>: Use *Composed Message Processor* to process a composite message. The Composed Message Processor splits the message up, routes the sub-messages to the appropriate destinations and re-aggregates the responses back into a single message.



Composed Message Processor pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The Composed Message Processor uses **Aggregator** to aggregate sub-messages.

Since the origin of all sub-messages is a single message, **additional informations** also can be passed to the Aggregator **(e.g., the number of sub-messages)**.

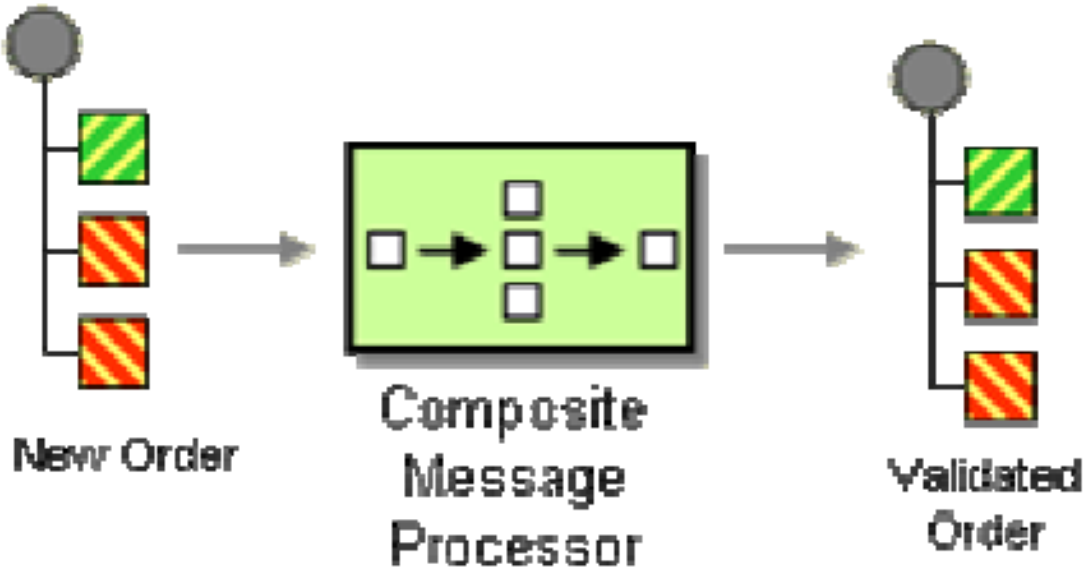
There are also **decisions** the Aggregator has to deal with: **what to do with missing/delayed messages?**

- Delay all related orders?
- Send them to an exception queue for further manual handling?
- Re-send the missing order requests?

Composed Message Processor pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

As this pattern shows, **individual patterns can be combined into a single, large, more complex pattern**.



Related patterns: Aggregator, Content-Based Router, Pipes and Filters, Splitter

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Scatter-Gather pattern

Scatter-Gather pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: Suppose an order request message that contains **order of several different items**. Some items are not in the stock, they have to be asked from **external suppliers**. There are different candidates as external suppliers, they may offer the requested item on different prices, with different speed, moreover, some of them may not have the desired item. **To satisfy the order in the best way, we should request all the external supplier and accept the best offer.**

Problem: How do you maintain the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply?

The solution has to be **flexible**:

- Maybe the **number of recipients is unknown** (e.g. they can subscribe),
- or the **recipients are determined centrally**.
- Perhaps **not all recipients give a response**.

So, the solution should **hide the number and identity of the individual recipients** from any subsequent processing.

Scatter-Gather pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

How to design the **subsequent message flow**?

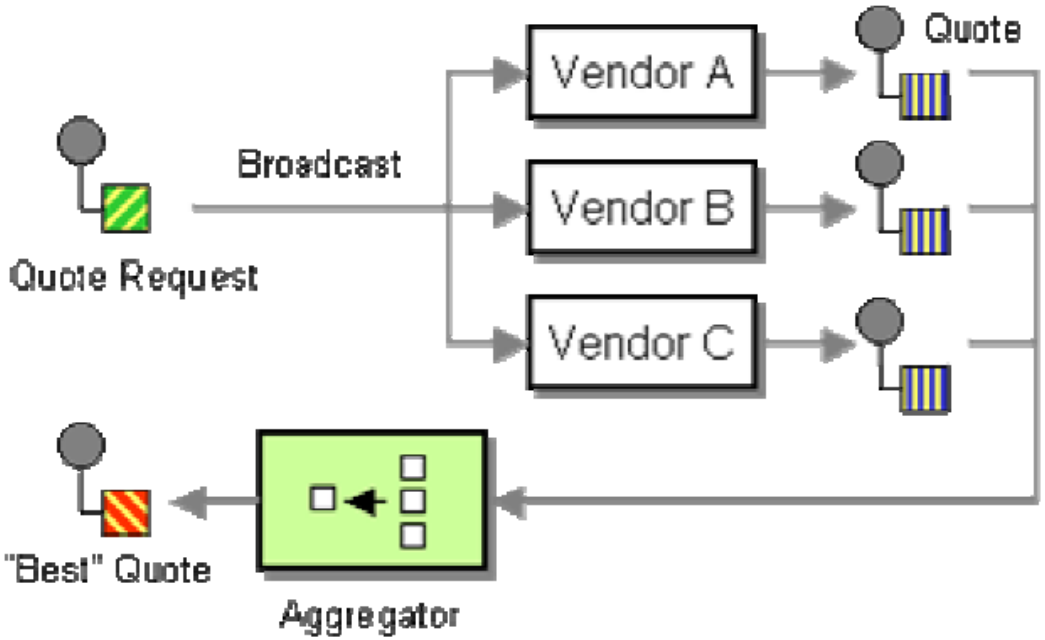
The simplest way is to let the **recipients** to **send their response on a channel**, **and** let the **subsequent process** to **handle the individual messages**.
This case requires the subsequent components to be **aware of the original message** (that was sent to multiple recipients). Moreover, the applied **routing logic** also **should be known**.

It makes sense to **combine the routing logic, the recipients and the post-processing of the individual messages into one logical component**.

Solution: Use a *Scatter-Gather* that broadcasts a message to multiple recipients and re-aggregates the responses back into a single message.

Scatter-Gather pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



Two variants **(they differ in how to send the original message to the multiple recipients)**:

(1) Using **Recipient List**: **centralized control** but **own message channel** for each recipient

(2) Applying **Publish-Subscribe Channel**: **distributed control** but one **single channel** **(Auction-style Scatter-Gather)**

Scatter-Gather pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

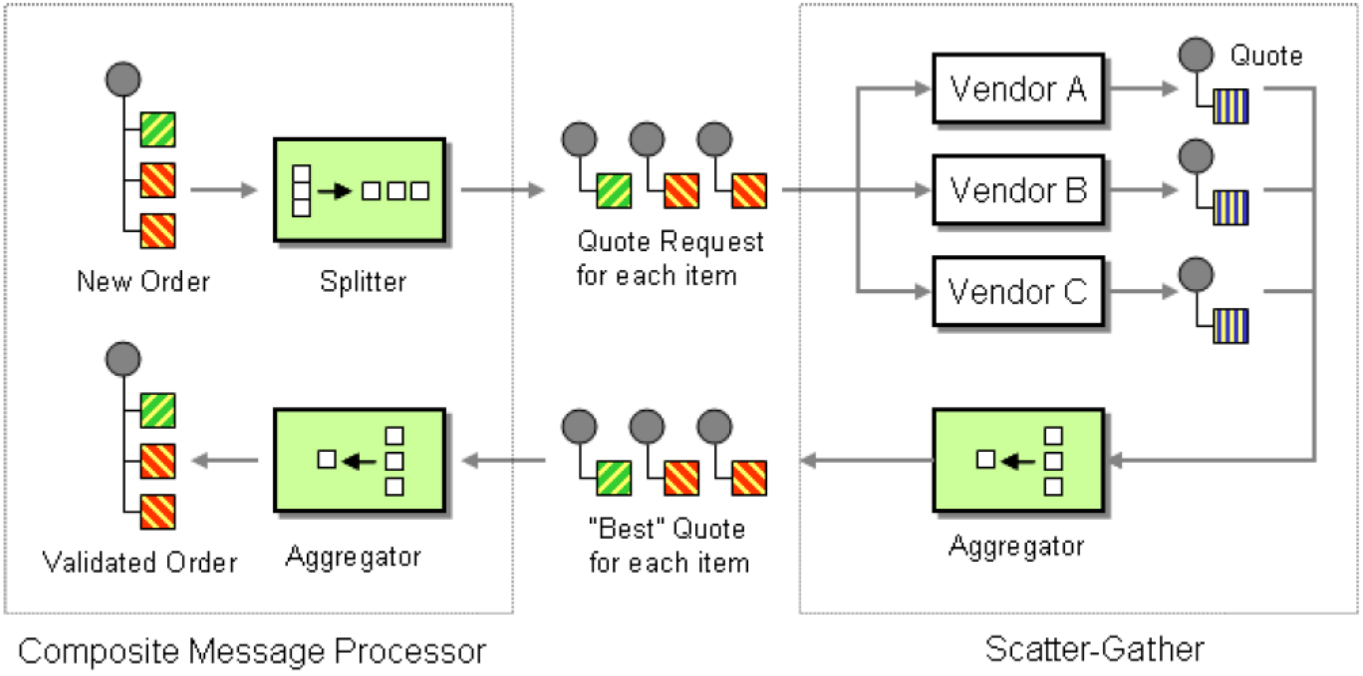
Scatter-Gather		Composed Message Processor
Broadcasts the complete message	⟷	Applies **Splitter**
Publish-Subscribe Channel, often Return Address, so all replies can be sent to a single channel	⟷	**Content-Based Router** (multiple recipients)
Responses are **aggregated** (e.g., best bid)	=	Responses are **aggregated**
Aggregation is more complex, since **the number of recipients can be unknown**.		**Synchronizes** the multiple parallel activities: subtasks may wait for each other

Scatter-Gather pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

A **compromise** between choosing Scatter-Gather or Composed Message Processor can be a ***cascading Aggregator***: it allows ***subsequent tasks to be initiated with only a subset of the results being available***.

Example: Combining Composed Message Processor and Scatter-Gather (1/2)



Scatter-Gather pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: Combining Composed Message Processor and Scatter-Gather (2/2)

The example contains **two different Aggregators**:

- the first one chooses the **best bid** (complex decision – e.g. one vendor is cheap but has not enough number of items) but **does not require response from all vendors**.

- the second one only **concatenates all responses** (simple algorithm), **but has to wait for all responses** and **has to handle errors/missing messages**.

The **composition** of patterns into larger patterns allows us to discuss the solution at a **higher level of abstraction and** to **modify details** of the implementation **without affecting other components**.

Related patterns: Aggregator, Composed Message Processor, Publish-Subscribe Channel, Recipient List, Return Address, Splitter

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Routing Slip pattern

Routing Slip pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: There are situations, when **the incoming messages have to be routed** to not only one single component but **through a series of components** (e.g., by applying **Pipes and Filters** architecture). Sometimes **the necessary components vary depending on the type of the message or other outer circumstances (e.g., validation depends on the applied payment method, or a customer who arrived on VPN does not need to be authenticated)**. So, we need to find a configuration that can route the message through a different sequence of filters depending on the type of the message.

Problem: How do we route a message consecutively through a series of processing steps when the sequence of steps is not known at design-time and may vary for each message?

The default **Pipes and Filters** architecture applies **fixed pipes**. Instead of the fixed pipes **we need to route the messages dinamically to different filters**. It necessitates the usage of **Message Routers as special filters** in the architecture.

Routing Slip pattern II.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The **requirements** are:

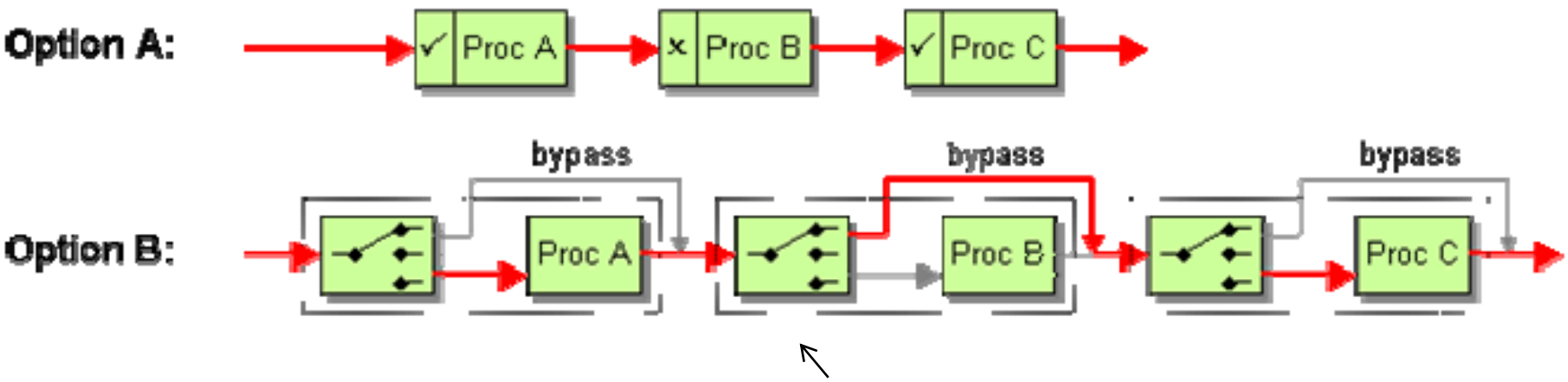
- **Efficient message flow**: messages avoid unnecessary components
- **Efficient use of resources**: does not use a big amount of resources (channels, routers, etc.)
- **Flexible**: the route of a message is easy to change
- **Simple maintenance**: there is a single point of maintenance - if a new type of message has to be supported, the error possibility remains low.

Routing Slip pattern III.

Alternative solutions for the following case:

the message has to be processed by A and C:

code added to each component (harder to reuse);
reactive filtering



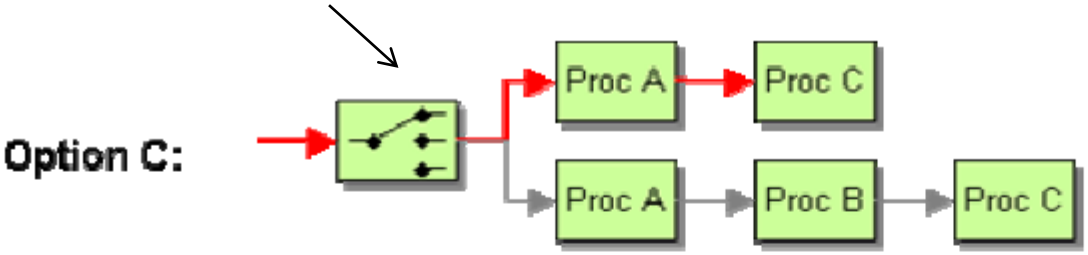
better solution for independent processing steps;
increases the composability but introduces more components (especially channels);
message has to go through several routers even only some processing steps required;
the routing logic is distributed; routers before each component that route to the next „available" step;
in case of a new message type every router has to be updated

In both cases, **messages are tied to executing steps in a common order**.

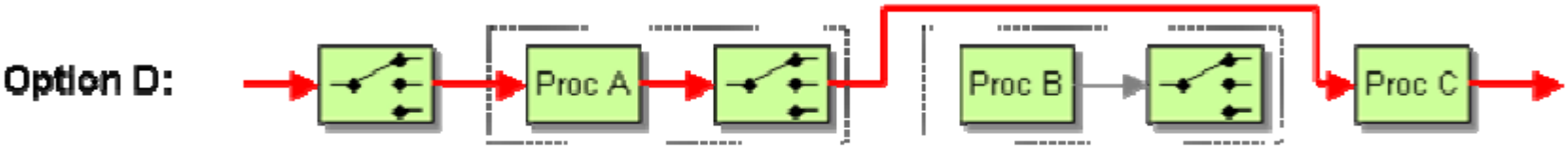
Routing Slip pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

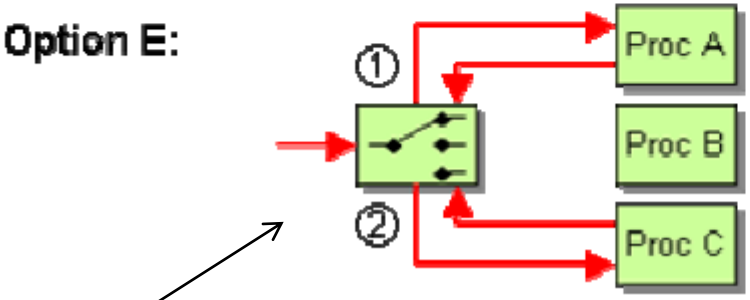
central point of control: an up-front Content Based Router, followed by all the possible sequences
only relevant steps (plus the router) → efficient
requires hard-wiring of any possibilities
multiple instances of components, lots of channels, maintenance nightmare



Routers after each step (plus at the very beginning), that route the message directly to the next „required" step (not like in Option B).



the routing logic is still distributed → large number of routers;



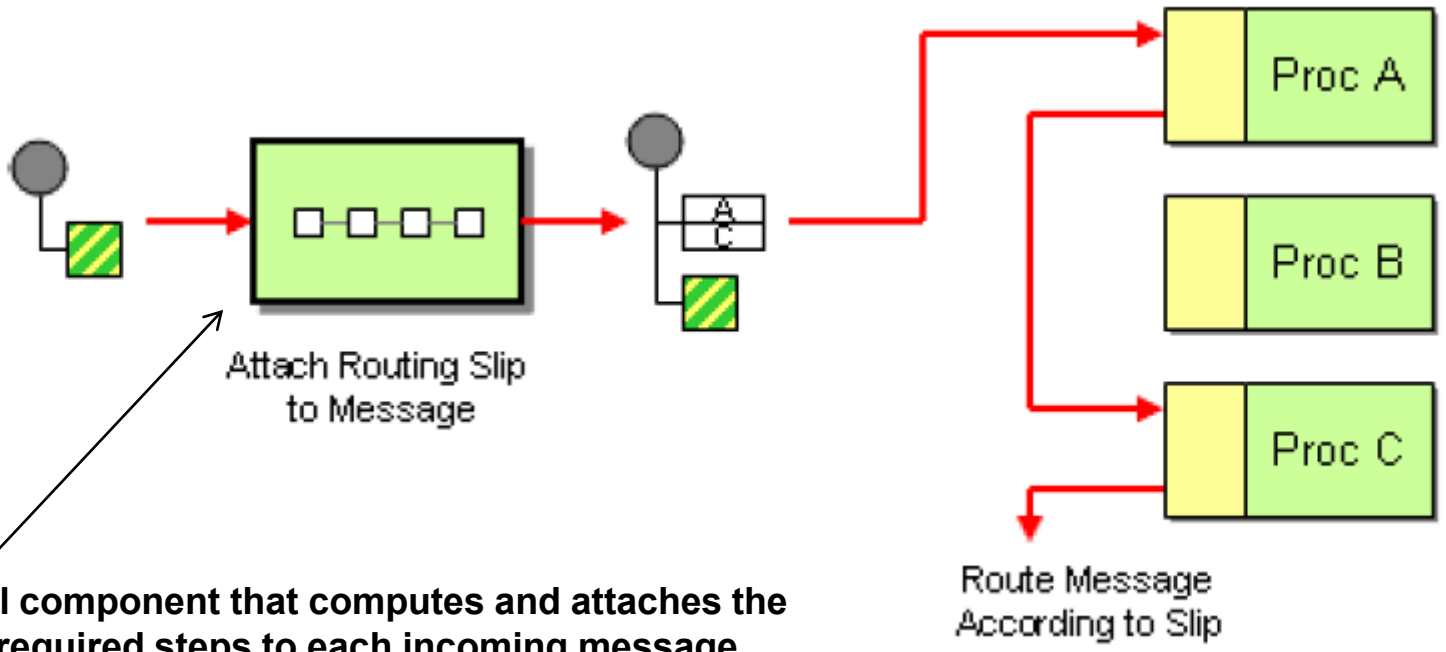
All router is combined into a „super-router". After each processing step the message is routed back to the „super-router", who has to determine the next step.

It requires to know which step is finished (either by a stateful router or by a tag that is attached to the message by the latest processing step). Requires about two times as much channel/traffic as Option C.

Routing Slip pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

<u>Solution</u>: Attach a *Routing Slip* to each message, specifying the sequence of processing steps. Wrap each component with a special message router that reads the Routing Slip and routes the message to the next component in the list.



A special component that computes and attaches the list of required steps to each incoming message.

Routing Slip pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

The Routing Slip **combines the central control** of the 'super-router' approach (Option E) **with the efficiency** of the hard-wired solutions (Option C): we do not have to return to the central router.

The **routing logic is built into the processing component** itself, but the **router** used in the Routing Slip **is generic** and does not have to change with changes in the routing logic (that is similar to the Return Address: a selection from a list).

The **computation of the routing table** can be done in a **central** place.

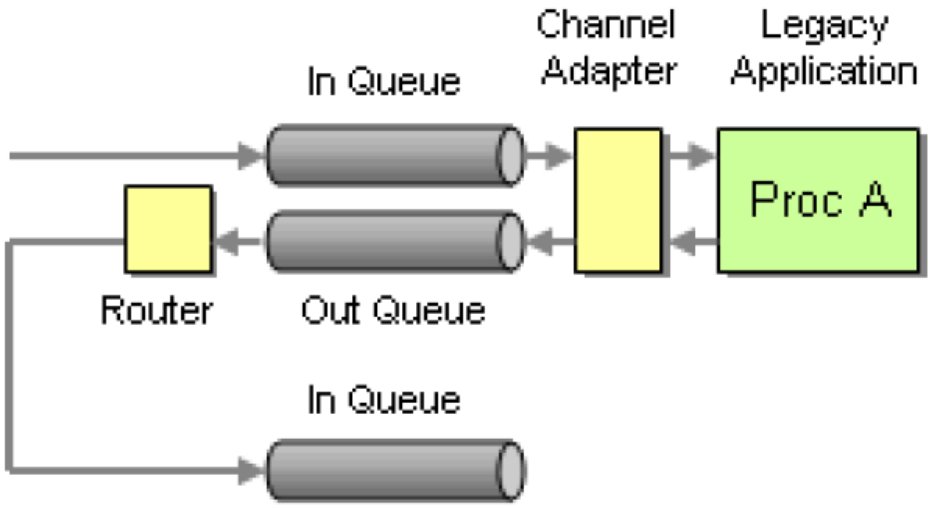
The **message size increases**.: **process state** (which steps have been completed) has to be included in the message. Because messages can be lost, it is useful to store the state of all messages in a central place.

The path of a message cannot be changed once it is under way: **if the route** of the message would **depend on the intermediate results** along the way, **it should be designed** in the central element **in advance**.

Routing Slip pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

If a component is a **packaged application** or a **legacy application**, we can not modify that by adding a routing logic. Instead of that, we need to use an **external router** that communicates with the component by messaging.



This solution requires **more channels and components**, however**, flexibility, maintainability and efficiency** remains.

Routing Slip pattern VIII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Main usage of Routing Slip:

- **A sequence of binary validation steps**: Each component has the choice between aborting the sequence due to error or to pass the message on to the next step.

- **Each step is a stateless transformation**: e.g., transformation of incoming orders from different partners

- **Each step gathers data, but makes no decisions**: for example, the original message contains only reference identifiers that have to be resolved from external sources to be able to make the final decision **(e.g., an order for DSL line contains only the phone number of the customer – the name, address, nearest central office, etc. have to be determined from external sources)**
In these cases if the flexibility of Routing Slip is not necessary, hard-wired chain of Pipes and Filters also may be enough.

Routing Slip pattern IX.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Content-Based Router

It has to have **knowledge about** all the **possible recipients and their capabilities**.
(in loosely coupled systems it is not desired)

Publish-Subscribe Channel + Message Filters

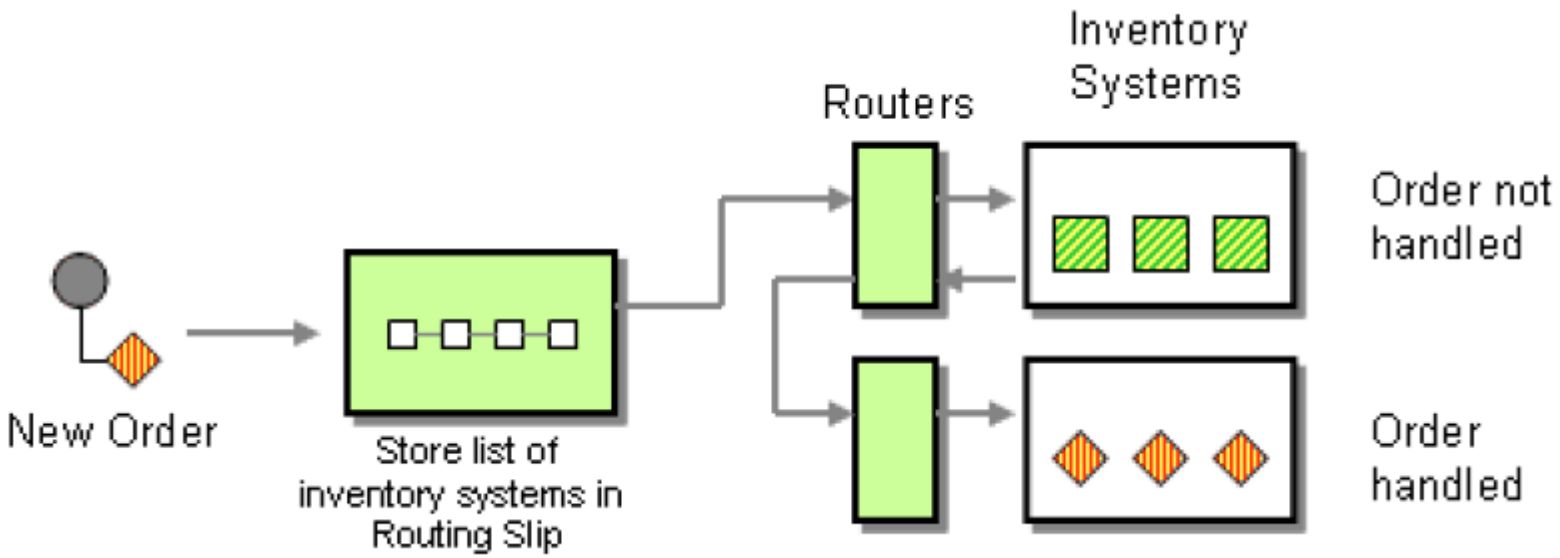
Distributed decision about processing, but there is a **risk of multiple message processing**.

Routing Slip

There is a central component that has to have **knowledge about all the possible recipient but not their capabilities**. It allows each component of the static list to accept a message or route it to the next component in the list. **No duplicate message processing**. **Unhandled messages** are **easy to determine**.

Routing Slip pattern X.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



This solution requires to publish **more messages than the single message of a Content-Based Router**: an average number of messages equals to 1/2 the number of systems. It can be improved by arranging the systems in such a way that the first systems to receive the message have a higher chance of handling the message.

Routing Slip pattern XI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Routing Slip vs. Process Manager

A ***Process Manager*** **supports branching conditions, forks and joins**. That is why it can be applied easily to realize **more complex** cases **than a simple sequential list** of components or for cases where the **flow of message has to be changed** based on intermediate results.

If we intend to realize these complex cases by a **Routing Slip**, analyzing and debugging the system may become **difficult** as the **routing state information is distributed across the messages**. Moreover, the configuration file may become **hard to understand and maintain**. We could include conditional statements inside the routing table and augment the routing modules in each component to interpret the conditional commands to determine the next routing location.

In these cases it is useful to use the **powerful Process Manager instead of the run-time efficient Routing Slip**.

Routing Slip pattern XII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Routing Slip as a composed service (1/3)

In **SOA** a single function is often composed of **multiple independent steps**:

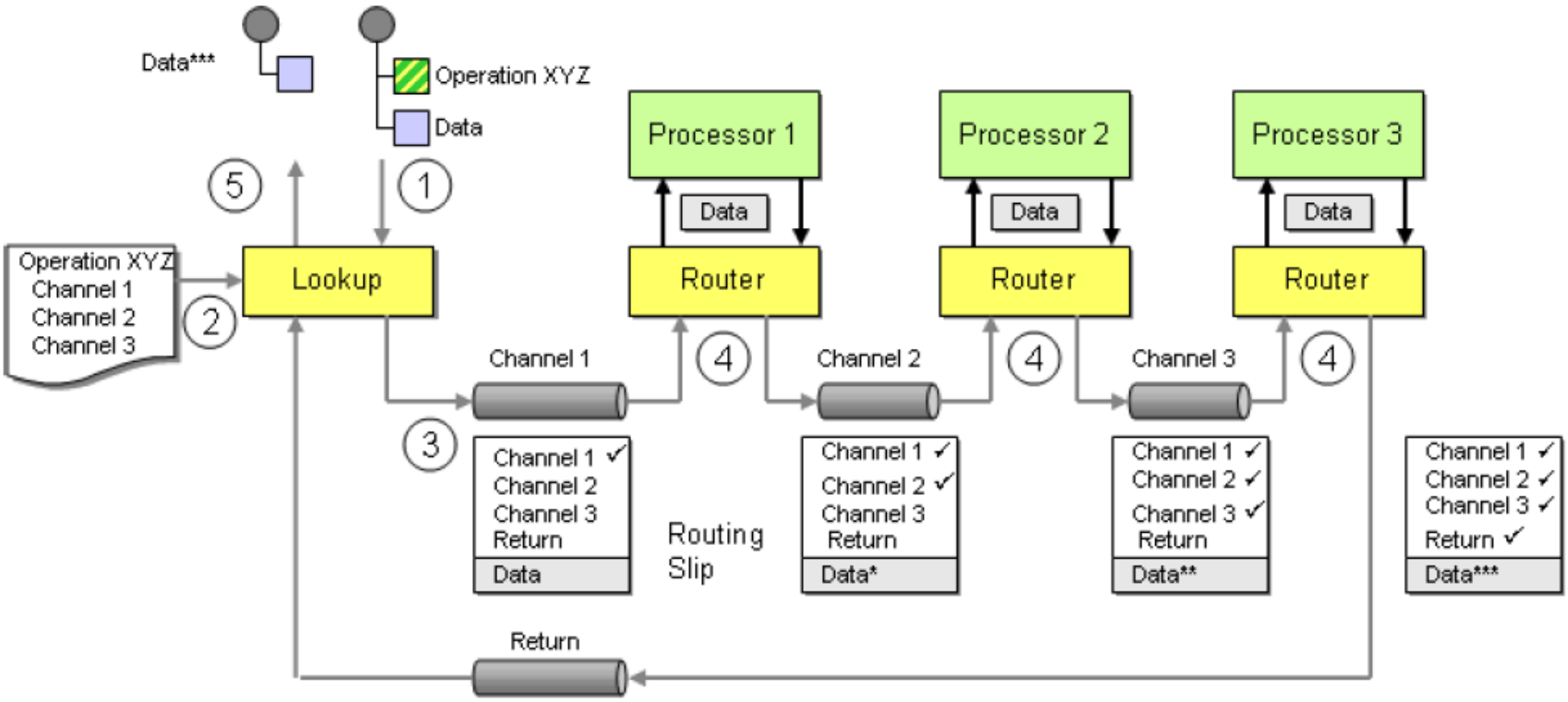
(A) … because **packaged applications** tend to expose **fine-grained interfaces** based on their internal APIs. Moreover, when integrating these packages into an integration solution, we want to work at a **higher level of abstraction**. **E.g., creating of a new account requires the creation of a new costumer, verification of credit data, setting his/her address, etc.**

OR

(B) … because a single logical function may be **spread across more than one system, but we want to hide this fact** from other systems.

Routing Slip pattern XIII.

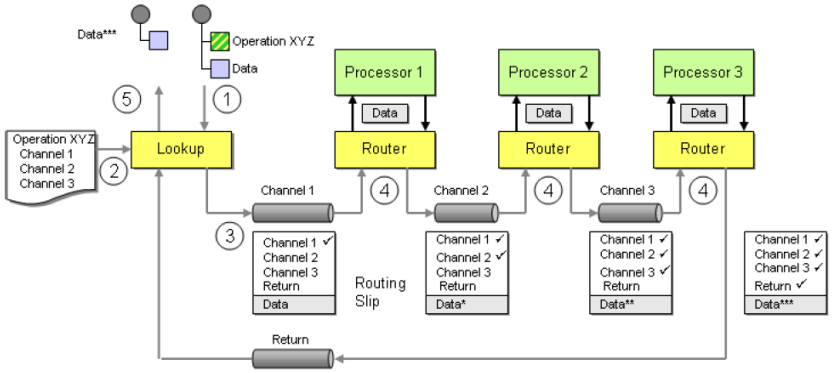
Routing Slip as a composed service (2/3)



Routing Slip pattern XIV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Routing Slip as a composed service (3/3)



1. Incoming request message
2. The lookup component retrieves the list of required processing steps from a **service directory**, then adds the list of channels and the return channel to the message header.
3. The lookup component publishes the message to the channel for the first activity.
4. Each router reads the request and passes it to the service provider. After the execution, the router **marks the activity as completed** and routes the message to the next channel based on its routing table.
5. The lookup component consumes the message and forwards it to the requestor.

To the outside, this whole process appears like a simple request-reply message exchange.

Routing Slip pattern XV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Example: WS-Routing (1/2)

Sometimes Web Service requests have to go through multiple intermediate stations.

Microsoft defined **Web Services Routing Protocol (WS-Routing)** – that is based on SOAP – for this purpose.

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://www.w3.org/2001/06/soap-envelope">
  <SOAP-ENV:Header>
    <wsrp:path xmlns:wsrp="http://schemas.xmlsoap.org/rp/">
      <wsrp:action>http://www.im.org/chat</wsrp:action>
      <wsrp:to>soap://D.com/some/endpoint</wsrp:to>
      <wsrp:fwd>
        <wsrp:via>soap://B.com</wsrp:via>
        <wsrp:via>soap://C.com</wsrp:via>
      </wsrp:fwd>
      <wsrp:from>soap://A.com/some/endpoint</wsrp:from>
      <wsrp:id>uuid:84b9f5d0-33fb-4a81-b02b-5b760641c1d6</wsrp:id>
    </wsrp:path>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    ...
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

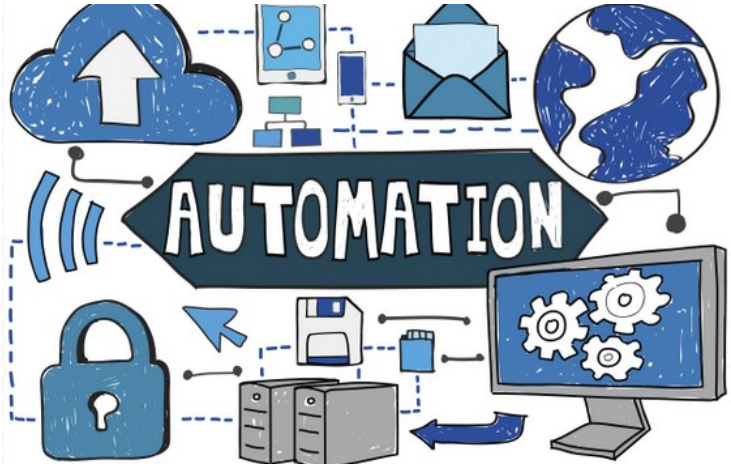
The message is routed from A to D, through B and C.

Routing Slip pattern XVI.

Example: WS-Routing (2/2)

WS-Routing can be used for **easy implementation of Routing Slip**.

However, it is worth to note that the field of WS-related standards and specifications is still a quickly improving and changing topic.



Related patterns: Content-Based Router, Message Filter, Message Router, Pipes and Filters, Process Manager, Publish-Subscribe Channel, Return Address

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Process Manager pattern

Process Manager pattern I.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Context: **Routing Slip** assumes that the **dynamic sequence of processing steps** is (1) **determined up-front** and (2) is **linear**. However, **in practice** these steps often need to be **executed parallel** or the **routing decisions** are **based** often **on intermediate results**.

Problem: How do we route a message through multiple processing steps when the required steps may not be known at design-time and may not be sequential?

Possible solutions:

(1) In a **Pipes and Filters** architecture (that can implement multiple independent processing steps) the **processing sequence can be changed** for each message if we insert multiple **Content-Based Router**. This solution is **flexible**, but the **routing logis is spread** across many routing components.

(2) If **Routing Slip** is applied, the **routing logic is centralized** (the message path is computed up-front), but the **message can not be re-routed** based on intermediate results and **lacks of parallel processing** of multiple steps.

Process Manager pattern II.

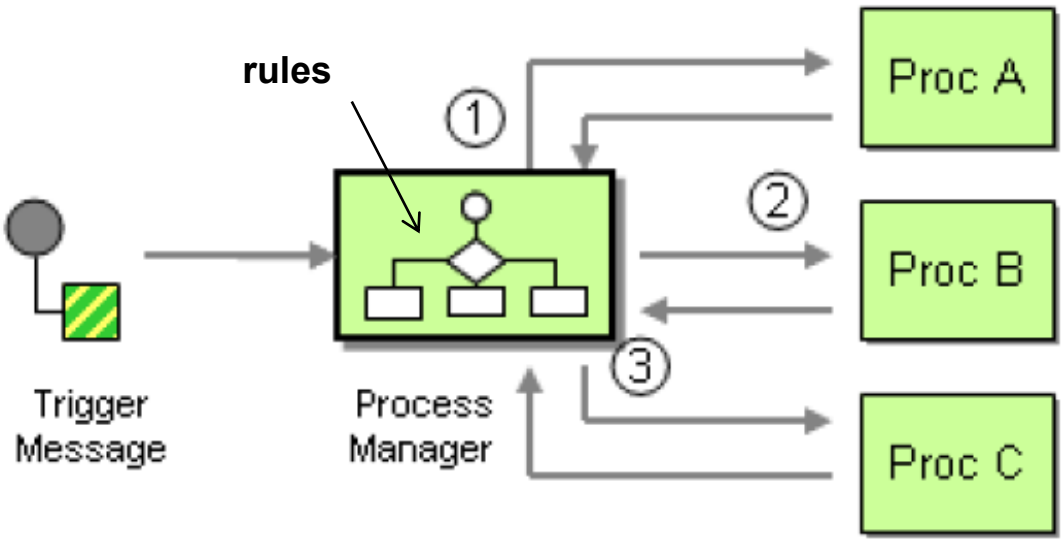
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

(3) A **centrally controlled and flexible** solution would be if we **route the message back to the central component after each processing step**. The central component has to **determine the next processing step(s)** to be executed **based on the intermediate results and the current 'step' in the sequence**. This would require the individual processing units to return sufficient information to the central unit to make this decision. However, **for determining the current state in the processing sequence, the central unit has to pass through extraneous information** that is not relevant to the processing unit, but only to the central component.
If we want to decouple the processing steps and message format from the central unit, **the central unit should have some memory**.

Solution: Use a central processing unit, a *Process Manager*, to maintain the state of the sequence and determine the next processing step based on intermediate results.

Process Manager pattern III.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



All message traffic runs through the **central 'hub'**, that's why it can become a **performance bottleneck**.
Process Manager **can execute any sequence of steps**, sequential or in parallel. Therefore, **almost any integration problem can be solved** with a Process Manager. However, it can distract from the core design issue and also cause **significant performance overhead**.

It is important to note that Process Manager, workflow or business process management and modelling is a pretty **extensive topic**.

Process Manager pattern IV.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Managing State

When the **central unit** gets a result from a processing unit, it **should know which step it was from the sequence** (e.g., the 5th step) of the multiple processing steps that the message has to go through. Since **the same processing unit may appear multiple times in the sequence**, this information should not been tied to the processing unit.

(For example, suppose that processing unit C is the component to execute both the 3rd and 5th steps of the sequence; so, when the Process Manager gets the result message from processing unit C, it can trigger either the execution of processing step 4 or 6, depending on the process context.)

If we do not want to make the message and the processing units too compicated, the Process Manager has to **store the current state of the process execution**.

Similarly, the **Process Manager can store other information, too (e.g., a processing result that is needed for a later processing unit)**. It makes the **individual processing units independent of each other** and makes the Process Manager to provide the function of ***Claim Check***.

Process Manager pattern V.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Process Instances (1/2)

A process execution may have several steps, so the **execution can take a long time**. During this time **other trigger messages may arrive** at the Process Manager. For the sake of efficiency, the Process Manager has to be able to carry out multiple **parallel executions**.
It is solved by creating a new **process instance** for each incoming trigger message. The process instance **stores the current execution step of the process and other additional data**. Each process instance has to be identified by a unique **process identifier**.

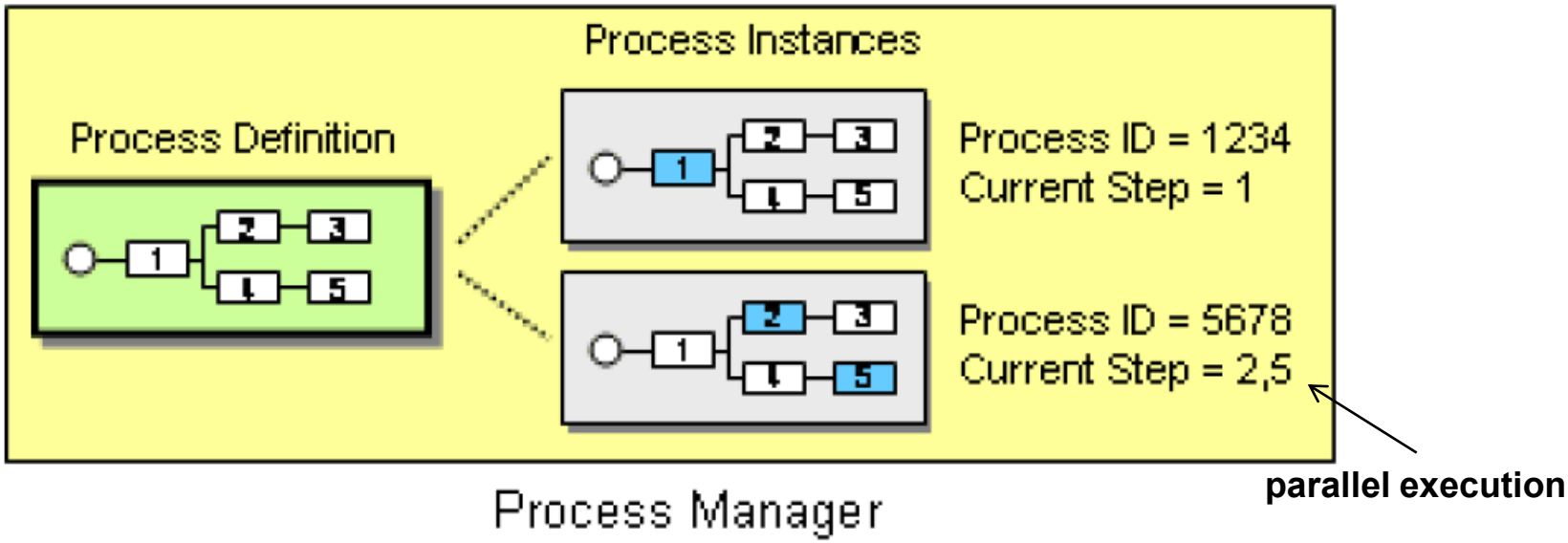
The **result messages that arrive from the processing units have to refer their process instance**. Since multiple instances may execute the same processing step, the Process Manager cannot derive the instance from the channel or the type of message. So, each message that is sent to a processing unit by the Process Manager has to include a **Correlation Identifier**, that is sent back as the part of the reply message by the processing unit.

Process Manager pattern VI.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Process Instances (2/2)

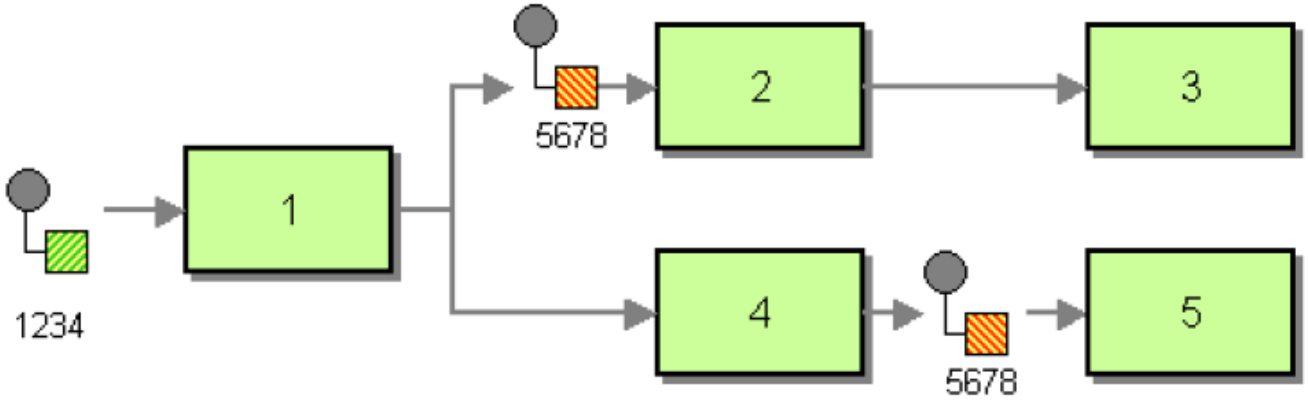
Each process instance follows a **process definition** (that is a **process template**). The process definition **describes the sequence of the processing steps** an execution has to follow.



Process Manager pattern VII.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

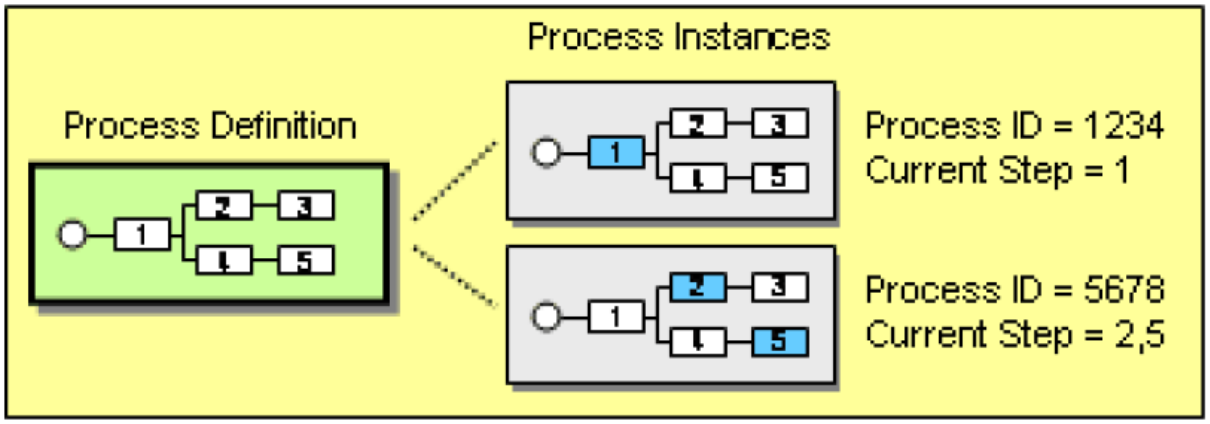
In Pipes and Filters arhitecture the **Message Channels** (pipes) have the same task as Process Manager: they **manage the state of the processing**:



Pipes and Filters

=

Process Manager



EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

THANK YOU FOR THE ATTENTION!

Reference:

Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns – Designing, Building and Deploying Messaging Solutions, Addison Wesley, 2003, ISBN 0321200683

www.enterpriseintegrationpatterns.com

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE