Process-Related Test Message

Karl Flieder

CAMPUS 02, University of Applied Sciences
A-8010 Graz, Austria
karl.flieder@gmx.at

Abstract. Enterprise integration patterns [7, 8, 17, 18] are well-known for designing, building and deploying messaging solutions. Testing loose-coupled systems during productive operation is a challenge due to several reasons such as the time needed for delivering a message, co-existence with productive messages and some others. Integration on process level has emerged as a promising paradigm for managing enterprise application integration strategies. However, there is a missing link between established messaging infrastructures, utilising transport of messages and a process-coupled view on it. Testing and analysing the results are two tightly coupled and normally inseparable mechanisms. The messaging model used, as well as the test model and the term of end-to-end testing will be introduced.

The idea of splitting a messaging infrastructure into different layers (business, logical, physical), in order to gain coherent test results for different business processes, is a novel approach. Likewise, injecting test messages on application-level during operation is a challenge. As a result, this paper proposes a pattern for an end-to-end test of loose-coupled systems on application level to ensure process-related testing through a logical grouping of the components involved.

Introduction

Messaging – often referred to as *Message-Oriented Middleware* (MOM) [3, 4] – is frequently used as a base service for integrating distributed systems on application level. For usability reasons, different departments often request their own view of the components involved during interaction. System administrators are not always available to answer questions concerning efficiency proof, system availability and further more. Obviously, a customized view of the related messaging components could help the department's super-user a lot. In this paper the authors propose a concept to meet this aim very efficiently without using a professional and expensive system monitoring tool. The proposed pattern focuses on the logical grouping of the physical message paths (Figure 6). It incorporates ideas of testing on application level and looking at the test results on process level.

Messaging technology enables asynchronous, high-speed, program-to-program communication with reliable delivery. The two basic components of messaging are messages and queues (Figure 2). Programs communicate by exchanging packets of data called messages with each other. A sender or producer is a program that sends a message by writing the message into a queue. A receiver or consumer is a program that receives a message by reading it from a queue. This asynchronous transmission makes delivery more reliable and decouples the sender from the receiver. Basically, a message

Copyright © 2004 Karl Flieder. Permission is granted for the purpose of EuroPLoP 2004 to Hillside e.V.

consists of two parts: a *header* and a *body*. The header contains metadata about the message; the body contains the payload generated by the sending application.

There are two messaging models available: *point-to-point* (one-to-one) and *publish/subscribe* (one-to-many) [5]. Point-to-point messaging is used to send data to a single application. This does not guarantee that every piece of data sent will necessarily go to the same receiver, because the channel might have multiple receivers. Publish/subscribe provides of the receiver applications with data. In this case, the data is copied for all of the receivers. Additionally, this messaging model works with events. The sender publishes messages based on events; the receivers subscribe certain topics.

Messaging systems are peer-to-peer facilities. Generally, each client can send messages to, and receive messages from any client. Each client connects to a messaging agent that provides facilities for creating, sending and receiving messages. Messaging capabilities are typically provided by a separate software system. This software defines a set of services to mediate between various applications across different languages and platforms. E-Mail messaging is based on the *store-and-forward* concept, too. Several messaging systems compete on the market, for example:

- IBM's WebSphere MQ, can be regarded as a leader on commercial markets [9].
- Sun's Java Messaging Service (JMS) as part of the open source J2EE JDK [13].
- Microsoft's Message Queuing (MSMQ) as part of the *System.Messaging* libraries in Microsoft .NET [11].
- Sonic Software's SonicMQ [12].
- Emerging Web services toolkits that support asynchronous Web services.

Information integration issues in enterprises are well-known as *Enterprise Application Integration* (EAI). The intention is to achieve effective communication across various business applications and delivery channels, thereby ensuring seamless data integration from different legacy systems. We distinguish between three main levels of integration:

- Exchanging data on *data level* relies on data management. Typically, JDBC and ODBC middleware or system interfaces are involved. This type of integration supports data exchange between disparate data stores when applications in different businesses must share information. This level of integration can be regarded as the lowest one in enterprise application integration.

- The approach on *application level* operates under the paradigm that applications can easily be wrapped using some form of middleware technology. Integrated application can be built on top of this middleware, for example messaging. As a matter of fact, this is a powerful concept in software engineering because middleware represents a layer of abstraction to prevent the top layers from knowing details about the layers below.

- Integration on *process level* represents the highest level of abstraction. Different applications work together to fulfill a certain business process. Typically, this level can be reached in assistance with an integration server dealing with application server tasks. Thereby, a lot of transformations, mappings and flow-control activities are being done this way.

Messaging Interaction Scheme

Point-to-Point Mode

In message queuing systems, messages are stored in first-in-first-out (FIFO) queues. Producers append (push) messages asynchronously at the end of a queue, while consumers pull them synchronously at the front of a queue as shown in Figure 1. Message queues implement unidirectional message paths. Messages are concurrently pulled by consumers with *one-of-n* semantics. This interaction model is called point-to-point. The solution proposed in this paper focuses on this messaging mode.

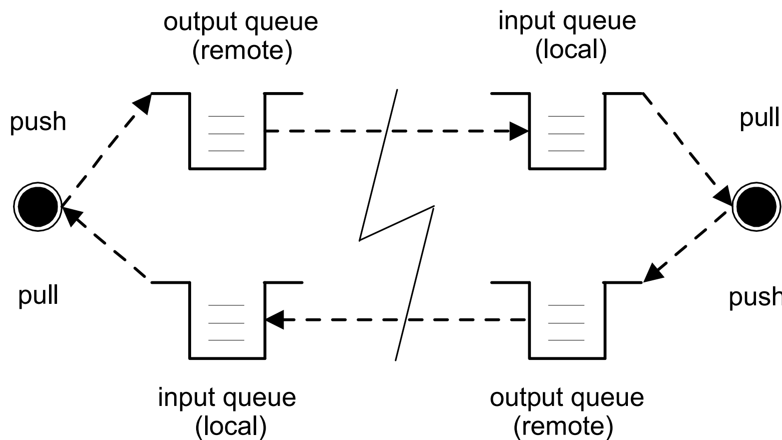

Figure 1: Point-to-point messaging

Message Structure

A single message within a message-oriented communication infrastructure consists of multiple sections:

a) The *message header* includes a number of predefined fields. These fields are used by the messaging system to decide how to process the message. They contain values that both, clients and providers, use for identifying and routing messages.

b) The *message body* contains the business information (payload) of the message. This can be text, binary and in case of JMS a Java object that represents one of the five JMS message types: *BytesMessage, TextMessage, MapMessage, StreamMessage* or *ObjectMessage.*

c) JMS provides an additional section for supporting compatibility to other messaging vendors: *message properties*. This part includes arbitrary properties assigned in addition to the header fields. They might be application-specific or provider-specific.

Object-Oriented Analysis

With the knowledge acquired so far, we can depict a simple messaging system as shown in Figure 2. A message queue is "*part of*" a queue manager ("*part of*" relationship). This situation represents a composition. Every queue manager owns at least one (in diagram: 1), but mostly more (in diagram: 1..*) message queues. The life-time of a message queue is bound to that of a queue manager. A message queue does not own messages ("*has-a*" relationship). In object-oriented analysis, this situation is known as an aggregation. The life-time of a message is not bound to that of a queue. For example, an already gathered message can survive as a text file. One or more producers respectively consumers are allowed to put messages into a queue or to get messages out of a queue. Nevertheless, restrictions to only one consumer may be appropriate, depending on the messaging system used [8, pp. 508]. Both the test messages to be implemented and the productive messages are specialized elements of the generalized parent "message". This represents a "*kind-of*" relationship.

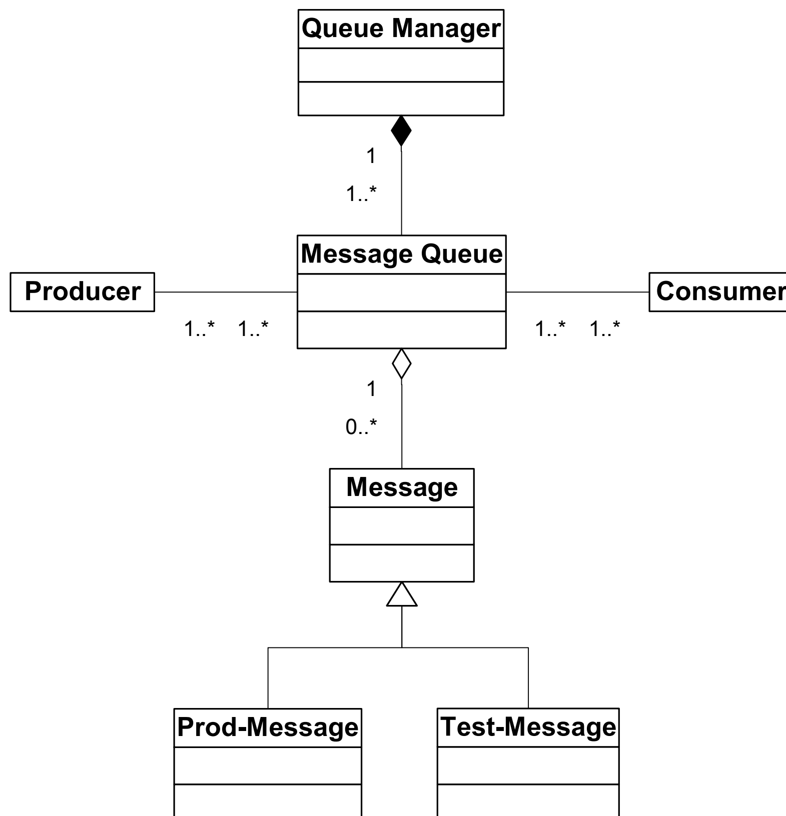

Figure 2: Objects of a simple messaging system

Test Pattern Template

Patterns are proofed solution concepts, filling the gap between a high-level vision of integration and the present system implementation. In other words, a pattern represents a decision that must be made and the considerations that influence that decision. CHRISTOPHER ALEXANDER [1] stated: *"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution."* A pattern template adds technology-specific elements to these basic items. ROBERT V. BINDER [2] suggested in his book a test design template, including the following unique and essential elements:

- **Fault Model**
  Why is this approach better than just poking around?
- **Test Model**
  What facets of the implementation under test should be considered and how should they be abstracted?
- **Test Procedure**
  How can an application model be transformed into test cases?
- **Oracle**
  How can latest results be evaluated?
- **Entry Criteria**
  Complying with certain entry criteria solves two common problems: The false confidence that may result by skipping component tests and pass system scope tests. Furthermore, the waste of time that results when components are not stable enough to perform the test.
- **Exit Criteria**
  The exit criteria should give an answer to the question: How much testing is enough?

The essential elements of pattern templates may vary, depending on the domain they focus on. After many considerations about what template style to use, I decided in favour of a mixture of a widely used variant, often used at PLoP conferences, and ROBERT V. BINDER'S test design template. As a result of my first experience with patterns, the interested reader should be able to find a coherent pattern proposal.

Process-Related Test Message

Intent

We aim to test the message paths between loose-coupled systems, including the sender and receiver applications, for multiple business processes.

Context

In loose-coupled systems, senders and receivers can be in different states (Figure 3). For business-critical applications it is sometimes necessary to test whether the whole system is reliable or not. Basically, this is possible by means of administrative tools provided by some vendors. However, there is a missing link called "process-oriented testing" on application level. In this context, *process* stands for a logically grouped amount of queues, senders and receivers involved to fulfil the information transfer for a certain business process.

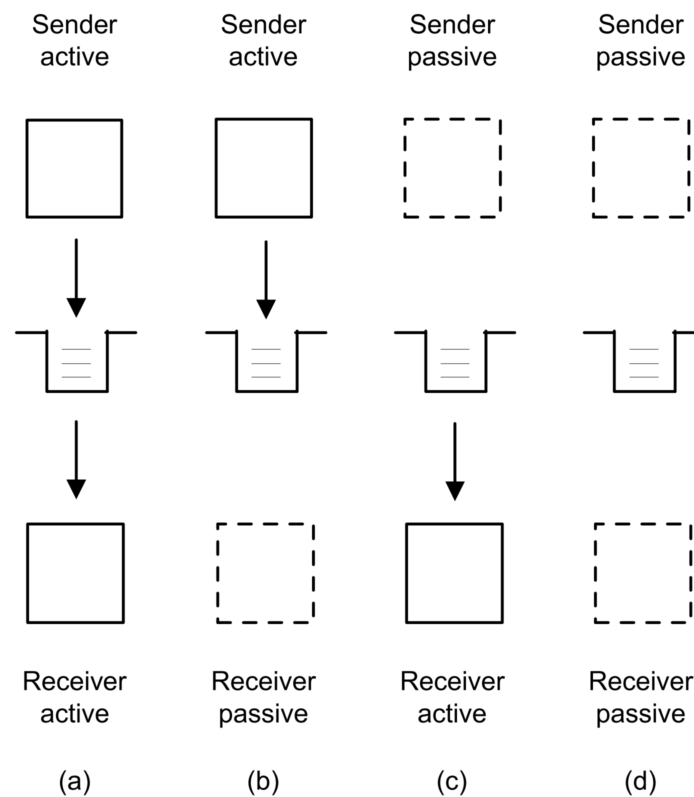

Figure 3: Four different states in loose-coupled systems.

A system that fails will not adequately provide the services it was designed for. In our case, testing on network level – for example, in assistance with *Simple Network Management Protocol* (SNMP) – is not appropriate. This due to the fact that the main components of interest are located on application level: producers, consumers and queues. Ideally, all the active and passive components involved during operation should be checked under productive conditions.

Fault Model

Defining predictable failure situations on application level implies that failures occurring on the levels below might be caught, too [15]: network failures, crash failures, omission failures, timing failures, response failures and arbitrary failures. The following possible error situations should be checked within the time frame for which the test results are valid:

- Queue managers are not available.
- Message queues are constipated or have already reached their maximum limit of messages.
- Message queues have accidentally got a wrong naming.
- Sender and receiver applications are not working properly.
- Messages are not able to arrive within the chosen time restrictions due to reasons like the time needed for database transactions of the systems behind.
- Encoding errors.
- The partner systems are not able to answer just-in-time.

Test Model

The following active and passive components are of main interest: sender applications, message queues (messaging system), receiver applications and – optional – the partner systems behind. An end-to-end test includes verifying transactions through each application involved, start to finish, assuring that all related processes are performed correctly.

Problem

How to test message paths of loose-coupled systems for multiple business processes?

You want to perform an end-to-end test on application level to find out whether the entire messaging system for a whole process is reliable or not.

Forces

- The productive messages within a messaging system must not be disturbed by test messages: they are not allowed to overtake the productive ones.
- For application reasons, the correct delivery sequence of the productive messages must not be changed. A strict FIFO (first-in-first-out) order is necessary.
- The frequency of the tests applied must not provide side-effects. Refrain from testing through a periodic heartbeat to avoid additional burden to the system.
- Other components involved during operation, such as senders and receivers of messages should be tested, too.
- The systems behind the receiver application (databases, legacy systems, etc.) must have enough time to answer the test messages.

Example

A network of queue managers was established, each of which is responsible for a reliable information exchange between internal and external systems.

- Scenario 1 (Figure 4) explains a *deterministic* use case: Consequently, for each queue to be tested, an equal number of reply queues set up for answering the test messages. A correlating reply message is expected for each message queue. No other answers than the expected ones will join this queue. This situation represents a deterministic use case which, in general, is easier to implement.

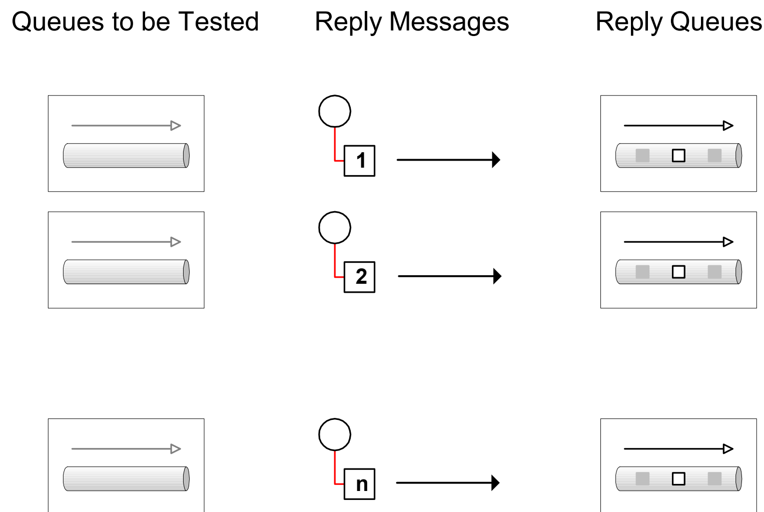

Figure 4: Deterministic use case

- Scenario 2 (Figure 5) describes a *non-deterministic* use case: It represents the more common test case where multiple "replies" join a single reply queue or a single message path. The answers might arrive at different times, which will lead to an unordered sequence of the reply messages within the common reply queue.

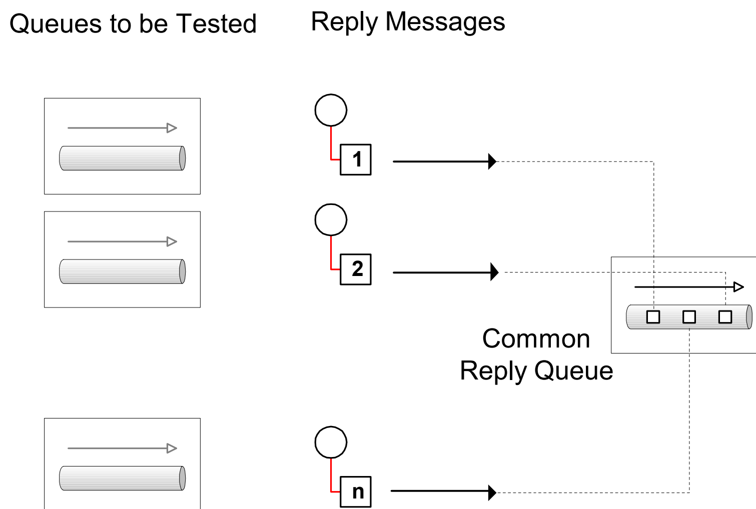

Figure 5: Non-deterministic use case

Solution

Group all the message paths in subsets that represent the business processes and send event-based test messages on application level.

Using messaging middleware, business process integration requires a logical orchestration of the components involved. A variety of business processes use messaging as their base service. We distinguish between three layers of abstraction:

a) The *business layer* focuses on analysing and defining the business objects, independently of the solution used to meet a company's requirements. This layer can be compared to a generic abstraction.

b) The *logical layer* specifies how business data can be exchanged in a structured fashion, following a number of rules. The purpose of the logical design is to refine the physical model in order to structure it for each business case and to parameterize the test application. This layer can be compared to specific requirements.

c) The *physical layer* delivers the messages and rules by means of syntax and code. Most of the business information involved will be turned into data elements composing the messages. The entire set of the physical message paths to be tested is represented by M in Figure 6. Out of this M, different subsets (M_1, M_2 … M_n) represent the individual business processes the message paths were set up for.

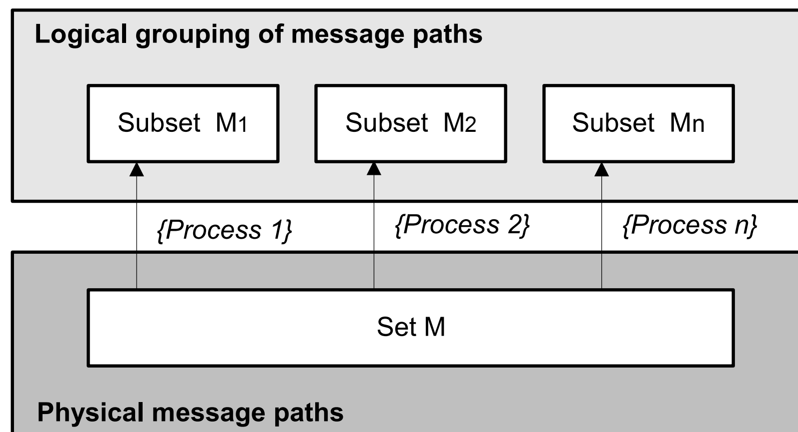

Figure 6: Creating process-related subsets

Discussing Messaging Issues

Testing asynchronous communication paths depends significantly on several criteria and conditions. What makes this approach so different and important is its focus on adaptations for the practical use in response to the different requirements and constraints.

Send and Forget. Once a *push* operation is complete, a sender does not have to wait for the answer. It does not even have to wait for the messaging system to deliver the

message. Nevertheless, many loose-coupled implementations are time-critical for several reasons.

Delivery Sequence. Within a messaging system there is no guarantee for the point of time a message is delivered. Problems might occur if messages get out of sequence. For example, for the exchange of records between different databases during production, the messages have to be delivered in a strict FIFO order to guarantee the consistency of distributed applications. This can be achieved either by a FIFO delivery sequence or by applying the same priority to all the messages. Using multiple receivers, there is the risk that the first message in the queue takes longer to be processed than the second one. This is due to different time frames needed for database transactions. Practical tests showed that a load balanced system with a running receiver application for each instance may cause this problem. In this case, the messaging system was not able to manage multiple consumers in a coordinated way [8]. To avoid running into these troubles, sometimes a turnaround might be necessary. A single reply queue offers a way out: only the receiver module of the test application will pull messages from this queue.

Frequency of Tests. In general, active testing adds burden to a system. As a requirement, the frequency of the tests applied must not produce side-effects. Since a periodic heartbeat would add this burden, but would also leave a gap of regularly untested periods, we implement event-based testing. Moreover, to avoid troubles caused by old test messages in a queue, an idempotent receiver was considered. This receiver discards answers of test messages that do not belong to the current test run.

Time Constraint. Because of the characteristics of asynchronous communication we consider a well defined time frame between sending out the test messages and receiving their related answers. The receiver application will pull them according to a strict FIFO order. After this time frame, missing answers of the test messages get classified as wrong. The time constraint depends on the average time used to process productive messages and the number of messages to be processed within a certain business process.

Expiry Time. Because of the time constraint discussed earlier, we consider an expiry time for the test messages after which they are no longer valid and therefore discarded by the queue manager. This contributes to reducing the burden to the messaging system.

Transaction Certainty. As shown in Figure 2, a single queue allows multiple sender and receiver applications. In this case, a sophisticated messaging system must be able to coordinate all the messages to make sure senders do not overwrite each other's messages. As a result, we have to ensure *exactly-once* delivery semantics of the messages [14]. Transactional queues, for example, will fulfill this requirement.

Parameters. For the logical orchestration and other requirements to meet the aims we need several parameters, depending on the implementation, for example (Figure 7):

- Set up a well defined time frame so that receivers may answer the test messages in time.
- Arrange the message paths to be tested in logical groups for each business process.

- Apply a unique message sequence for each test run to distinguish this cluster of messages from others. This allows identifying the test results which belong together.
- Consider to parameterize the return address at which the answers are expected, depending on the use case.
- Define a database connection string for persisting outgoing and incoming messages.
- Define additional properties for displaying the results, for example direction indicators or colours for the status of the test results.
- Use a flag whether a message path (queue) is allowed to be tested or not at runtime.

Consequences

As a result of the pattern, several of the forces previously described are resolved or balanced.

Benefits

- Structuring the physical message paths into logical groups by means of parameters and mappings (e.g. the return paths) enhances process-oriented testing.
- Different use cases, for example internal as well as external queue managers might be tested.
- Message paths beyond the scope of interest do not need to be tested.
- The test results can be evaluated and displayed for each business process separately.
- Launching non-periodic test runs on demand does not add much burden to the system.
- Testing on process level provides a higher value of integration during testing.
- The ability to persistently store all of the test messages and their answers provides great flexibility in reporting and analysis.

Liabilities

- Compared to a basic set of test tools, this approach requires more effort and resources for analysing the requirements as well as for implementing.
- Sometimes, transactional queues cannot provide support for message priorities.
- To restrict a system to only one consumer can be regarded as a drawback
- Implementation costs are relatively high.
- Applying a strict FIFO-ordering automatically disallows priority-ordering.

Implementation Example

In queue-based messaging systems, in general, a standard message format with message header and message body is used. The necessary parameterization will most likely be applied to the message header. Sometimes, it is also possible to put all the orchestration logic in a user-defined telegram within its message body to gain more flexibility. Figure 7 shows an example with additional information about the business processes, also called *header*, within the message body.

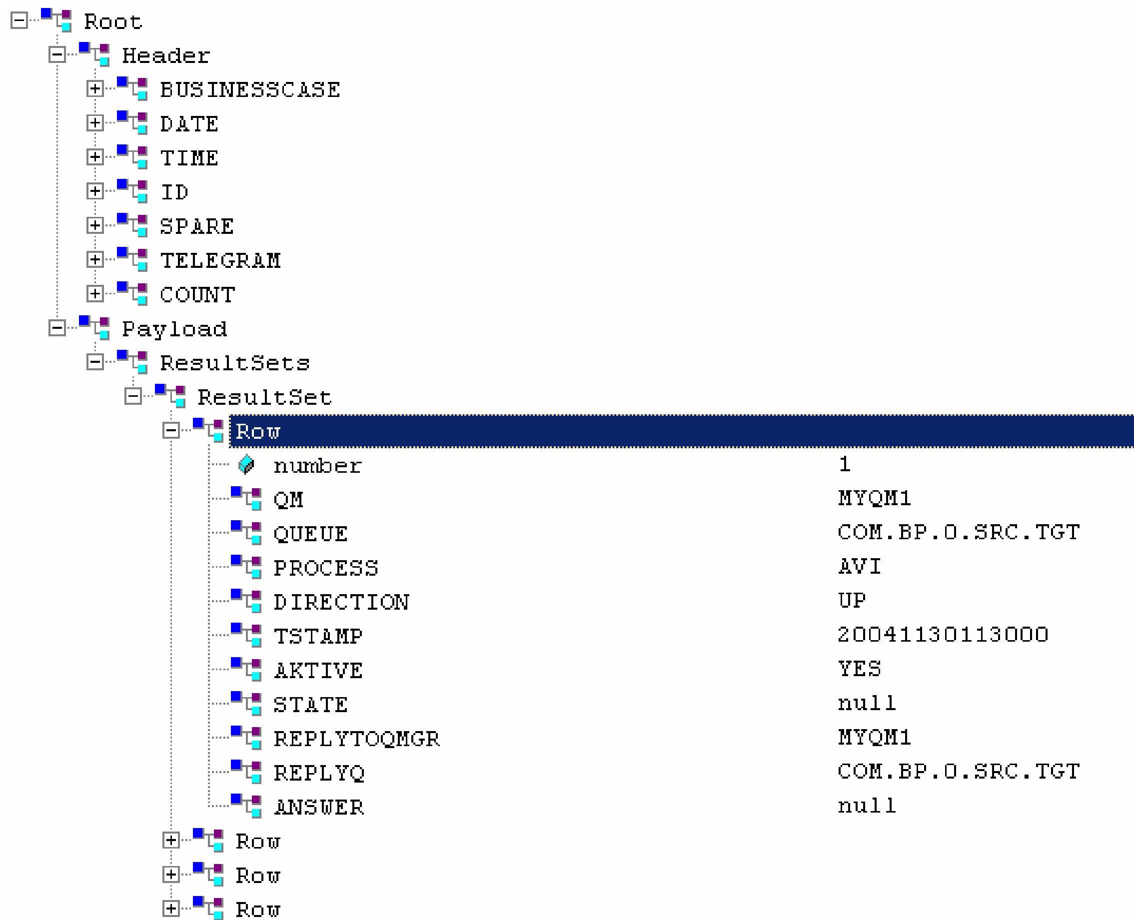

Figure 7: Attributes of a test message

Known Uses

- Magna Steyr's *MQWatchog* [6] reflects the basic ideas discussed in this paper.

- *Blat* [16]
  When sending e-mails to a certain list of receivers, a confirmation of delivery or a confirmation of having opened the e-mail may be requested. Depending on certain characteristics of the address list different categories can be implemented.

- *Microsoft Outlook* as well as *Outlook Express* supports the key features of this pattern. Outgoing e-mails use an address list for multiple recipients. Each of them is provided with the same subject, representing a certain business process. A reply message, including a return path, might be set up via the rules editor.

Related Patterns

Test Message, Content-based Router, Point-to-Point Channel and some others. [8].

Acknowledgements

First of all, I would like to thank *Marcos C. d'Ornellas*, my EuroPLoP 2004 shepherd, for many helpful suggestions. I also appreciated the feedback from the members of workshop C to improve my paper.

References

[1] Alexander, C., Ishikawa, S., and Silverstein, M. *A Pattern Language*. Oxford University Press, 1977.

[2] Binder, R.V., *Testing Object-Oriented Systems: Model, Patterns and Tools*, Addison-Wesley, 1999.

[3] Banvavar, G., Chandra, T., Strom, R., and Sturman, D. A Case for Message Oriented Middleware, *13th International Symposium on Distributed Computing (DISC 99),* 1-18.

[4] Blakely, B., and Harris, H., *Messaging and Queuing Using the MQI*, McGraw-Hill, 1995.

[5] Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A.-M. The Many Faces of Publish/Subscribe, *ACM Computing Surveys*, Vol. 35, No. 2, (2003), 114-131.

[6] Flieder K., Test und Visualisierung einer Message Queuing Infrastruktur, diploma thesis, *CAMPUS 02 - Graz*, 2004.

[7] Herzner W., Message Queues – Three Patterns for Asynchronous Information Exchange, ARC Seibersdorf Research, *EuroPLoP 2003*.

[8] Hohpe G., and Woolf B., *Enterprise Integration Patterns*, Addison-Wesley, 2003,

[9] IBM Corp., WebSphere MQ formerly – MQSeries, http://www-306.ibm.com/software/integration/wmq/ (2004-12-17)

[10] Liebig, C., and Tai St., Advanced Transactions, *Lecture Notes in Computer Science*, Springer, Heidelberg, Vol. 1999 / 2001.

[11] Redkar A., Walzer, C., Boyd, S. et al., *Pro MSMQ: Microsoft Message Queue Programming,* Apress LP, 2004.

[12] Sonic Software, SonicMQ Product Documentation, http://www.sonicsoftware.com/developer/documentation/index.ssp (2004-12-17)

[13] Sun Microsystems, Java Messaging Service (JMS), http://java.sun.com/products/jms/ (2004-12-17)

[14] Tai St., and Rouvellou I., Strategies for Integrating Messaging and Distributed Object Transaction, *Lecture Notes in Computer Science*, Springer, Vol. 1795, 2000.

[15] Tanenbaum, A., and van Steen, M., *Distributed Systems – Principles and Paradigms*, Prentice Hall, 2003.

[16] Website: http://www.blat.net (2004-12-17)

[17] Website: http://www.enterpriseintegrationpatterns.com (2004-12-17)

[18] Woolf B., and Brown K., Patterns of System Integration with Enterprise Messaging, *PLoP 2002.*