Flexible Call-by-call Settlement
An Opportunity for Dynamic AOP

Christian Hofmann
TU Ilmenau
Ehrenbergstraße 29
98693 Ilmenau, Germany
hofmann_ch@gmx.de

Robert Hirschfeld
DoCoMo Euro-Labs
Landsberger Straße 312
80687 Munich, Germany
hirschfeld@acm.org

Jeff Eastman
Windward Solutions
1081 Valley View Ct.
Los Altos, CA 94024, USA
jeff@windwardsolutions.com

ABSTRACT

Dynamic aspect-oriented programming is gaining interest due to its ability to provide attractive solutions to challenging technical problems. Most scenarios presented to date are motivated by the technical capabilities of a particular platform rather than application-level requirements to be addressed. In this paper, we describe a scenario taken from telecommunications where settlement systems of operators and call-by-call providers need to be integrated in a flexible manner after the system's initial deployment. By comparing static and dynamic object-oriented and aspect-oriented approaches, we present a case for dynamic AOP.

1. CALL-BY-CALL SCENARIO

In the telephony domain, rate plans vary from provider to provider. There are many different tariffs, for for example: local, long distance, international, or toll-free calls. There are also variations in how and when customers are billed. Besides other ways to compete in this market, there are call-by-call providers offering dial-around services. Here customers can choose their actual connectivity provider for each individual call by dialing a specific prefix or toll-free number prior to the actual destination phone number to be called (Figure 1).

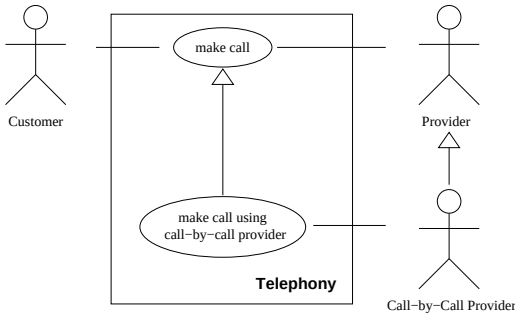

Figure 1: Dial-Around Services

The great flexibility offered to customers to select the best (and in most cases cheapest) connectivity provider demands a corresponding flexibility in the core telephony network and its supporting systems. Network operators and call-by-call providers can choose to start or terminate business relationships at any time. Such decisions are rarely synchronized with the design, development, or deployment of the responsible software systems. Most of these partnerships are formed and terminated while their systems are being operated long after their initial deployments. Entering into a partnership requires both operators and providers to integrate their operations environments. Such integration requires agreement on data and data formats to be exchanged as well as the way this data is to be transmitted.

Settlement is one such area to be addressed (Figure 3). Both operators and providers run their own settlement pipelines that are unlikely to be exactly compatible in the formats of call data records to be processed. Nor are the interfaces required to interchange raw or processed call data records standardized. Finally, security policies to be enforced and communication protocols and endpoints to be utilized have to be agreed upon. For customer convenience, some operators offer combined invoices to simplify customer billing. Items charged for by a call-by-call provider need to be listed separately from the ones charged for by the operator to make cost distribution explicit and transparent to the customer. The way items are listed may also vary from partnership to partnership.

As it might have become apparent from the problem description so far, integrating settlement systems is a rather complex task. Complexity increases by the general requirement to keep system down time minimal, on both the operator and the provider side. An operator can partner with several call-by-call providers over a period of time, as similarly a call-by-call provider might partner with several operators to render its services to its customers.

In our paper we discuss selected system functionality as indicated by the use-cases marked gray in Figure 3. We identify variation points needed to support the aforementioned establishment and termination of partnerships, and describe

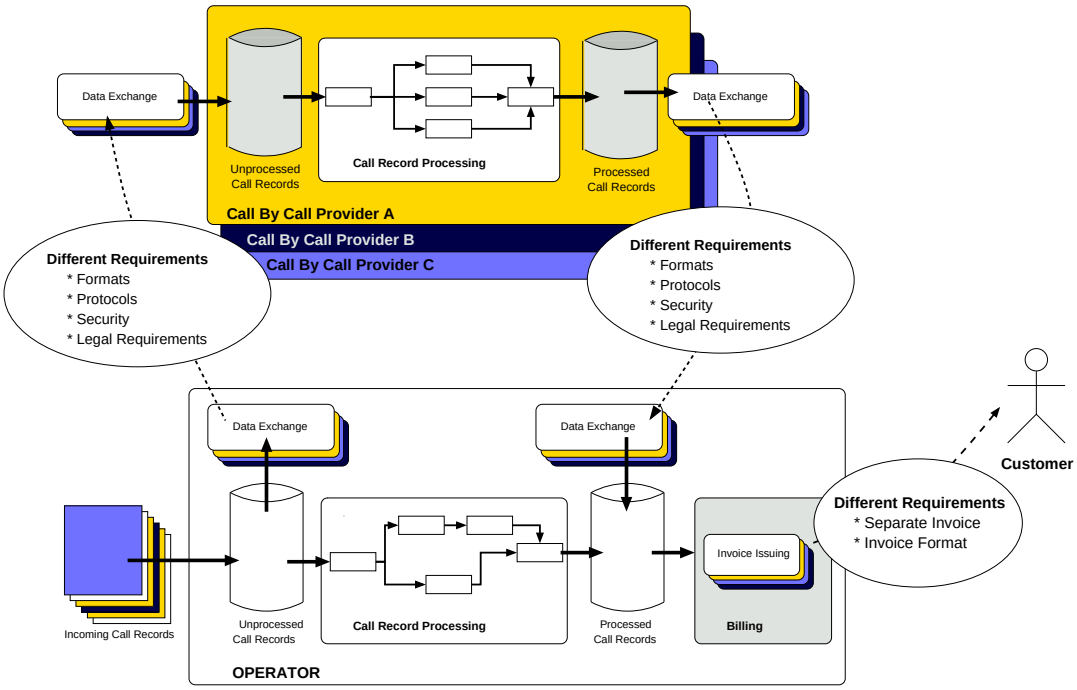

Figure 2: Variation Points

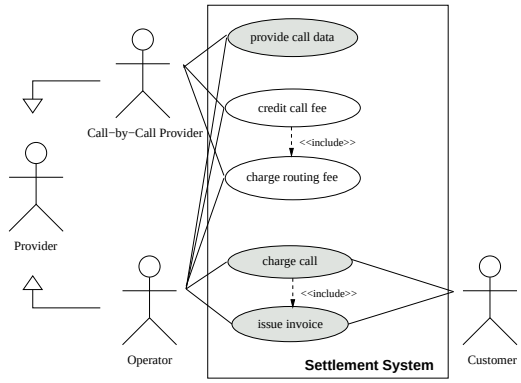

Figure 3: Call-by-call Settlement

how these variation points can be realized by applying both object-oriented and aspect-oriented approaches. We show how dynamic composition is of benefit in both cases; recognizing the dynamic aspect-oriented approach as the most beneficial one.

2. SELECTED VARIATION POINTS

As indicated in the previous section, integrating settlement systems offer interesting challenges. From the point of view of each participant, its settlement system is considered the stable part in such an activity. The settlement of calls can be described as the processing of call data records. It is typically performed as a sequence of processing steps that can be conveniently arranged within a settlement pipeline. Each step is responsible for one particular task. Example tasks are data record collection, correlation of ingress and egress records, fraud detection, duplicate removal, and billing. In general, a settlement system converts raw data records into processed records that in turn are used for billing all participating parties (Figure 2).

While operators process their call data records themselves, records of call-by-call providers are processed preferably at a provider's site. After sorting and collecting records for every individual provider (for example according to the dialing prefix used to initiate the call), all such records are transmitted to each corresponding provider. There they are handled similarly to the processing performed at the operator's site. At the end of a provider's settlement pipeline, calls are settled either by the provider directly or, for customer convenience, by the operators customers originated their calls. In the former case, the provider uses its own billing system. In the latter case, all processed call data records need to be transferred back to the operator where they are phased back into the operator's settlement system. There this data is eventually used to prepare the customer's bill that enlists provider charges separately and differently.

From this description we can infer at least three variation points to be present to allow for operator-provider settlement system integration:

- Raw call record transfer from the operator to the provider,
- Processed call record transfer from the provider back to the operator, and
- Provider-specific billing by an operator.

Both the first and the second item most likely require data conversion from one system's representation to the other's. Also, it has to be decided which security policies (including encryption mechanisms) to enforce and which transmission protocol suites to use (Figure 4).

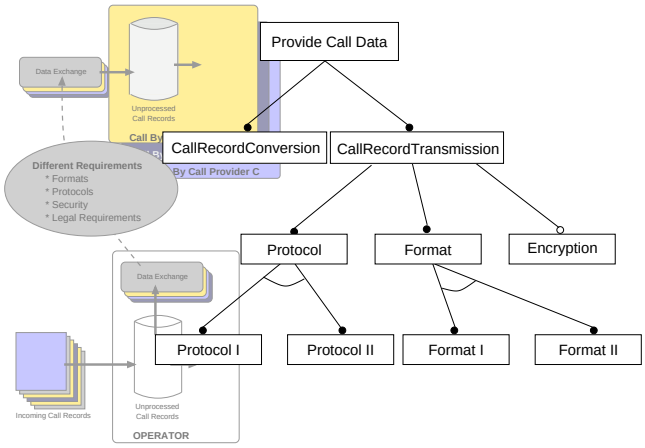

Figure 4: Call Record Exchange

The third item, provider specific billing needs to reflect legal constraints and requirements (such as privacy concerns) or company guidelines (such as style guides for bill rendering). Also, customers may often still select to receive their invoices by mail instead of viewing them online via the Web (Figure 5).

To integrate two such settlement systems, at least one if not both of them need to be adapted, at all of the variation points mentioned above. In addition to that, system downtime must be kept at a minimum. This requirement makes runtime adaptation very attractive for system integration. Partnership changes need not then equate to an off-line system modification. The following sections discuss four approaches to address system integration and show how system downtime can be avoided.

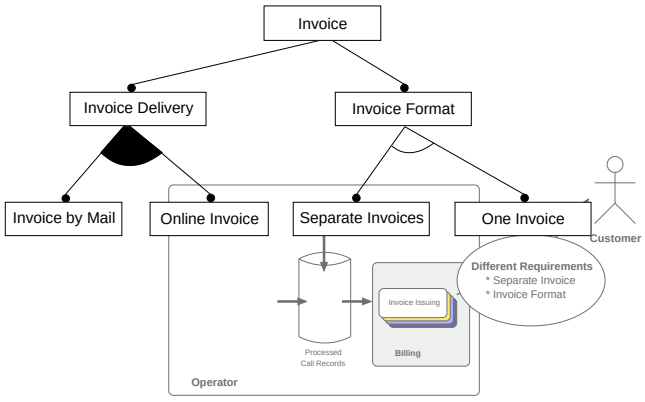

Figure 5: Invoice

3. OBJECT-ORIENTED VARIATION POINTS

First we show how some of the variation points outlined in the previous section can be implemented with object-oriented technology, relying on polymorphism or dynamic registries with explicit dispatch.

3.1 Static Object-Oriented

Figure 6 models call records, different kinds of calls, and the specifics needed by individual providers via single inheritance. Here we can see that the kind of call or call-type (in our example only long-distance and international are addressed) acts as the dominant decomposition criterion. Both implementations can offer different ways of charging, or - from the operator's point of view - different ways of creating provider bills. To further distinguish long-distance and international call records by provider, we need to subclass each of them and enhance every such subclass with provider-specific behavior that might be duplicated across neighboring branches of the class tree. Such behavior can comprise call-record conversions or the selection of transmission protocols.

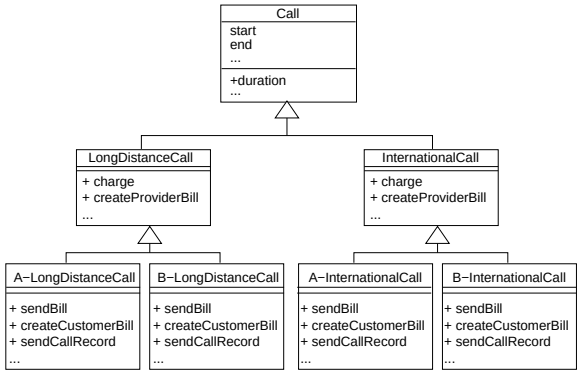

Figure 6: Dominant Decomposition by Call-type

If we decide to use provider-specifics as the dominant decomposition criteria, we might end up with a class tree as shown in Figure 7. Information specific to long-distance or international calls needs to be modeled by further sub-classing our provider-specific classes. This modeling step will introduce duplication across neighboring branches of the class tree similar to the dominant decomposition by call-type described above.

In languages providing multiple inheritance or mix-in behaviors, we might end up with a class model as depicted in Figure 8. While with mix-ins there can still be a dominant decomposition (here kind of call, or call-type), mix-ins are a means to help us avoiding code duplications by combining crosscutting concerns in our implementation model.

Note that all models described above aim not only for implementation reuse, but also for simplicity of method dispatch by using polymorphism. Such dispatch is necessary to select the appropriate implementation that matches, at each variation point, the correct provider and call type. Dispatch is also needed when transmitting raw call records from an

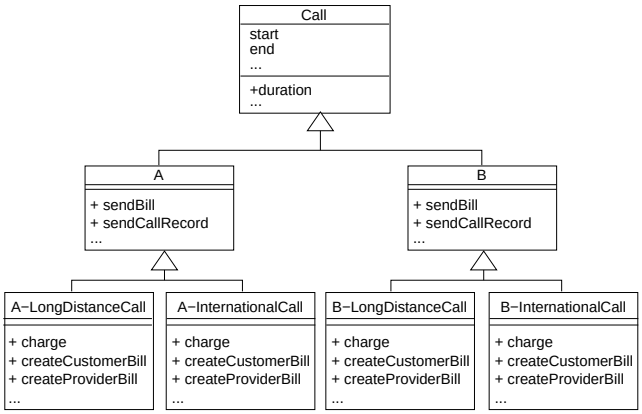

Figure 7: Dominant Decomposition by Provider-specifics

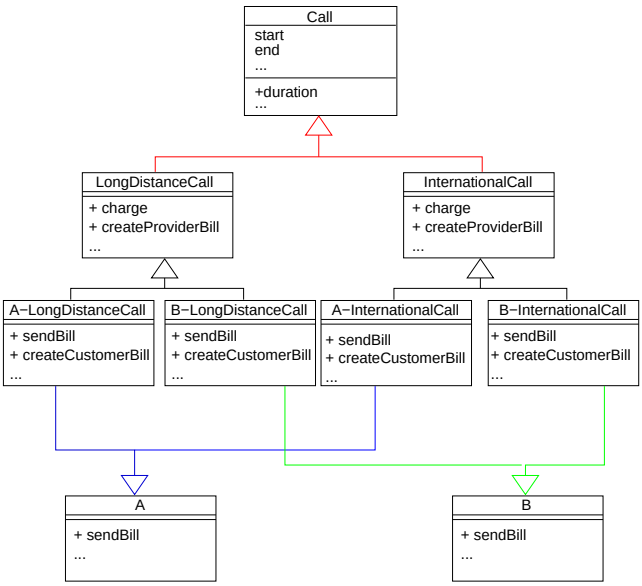

Figure 8: Mix-in Solution

operator to one of its partnering call-by-call providers, and when transmitting processed call records from a provider back to to one of its partnering operators. Finally, similar selection criteria also apply for provider-specific customer billing at the operator's site.

When using a static object-oriented approach, we need to know all possible partners in advance at development time in order to provide a complete set of implementations covering all possible dispatches that might be necessary during system operation. Since, in a static object-oriented approach, code cannot change at runtime, all necessary dispatch code needs to be provided initially. Code that turns out to be incomplete then requires the exchange of deployed system components. This requires expensive hardware and software fail-over solutions to avoid system outages and down-time in the high availability (99.999%) telecom world.

3.2 Dynamic Object-Oriented

A dynamic object-oriented solution benefits from language platforms that allow to load and integrate additional code into or remove code from a running system. Examples of such platforms range from Java with its dynamic class loaders to Smalltalk or Lisp. In the latter two there is no distinction between code and data, thus any data made available at runtime can also be interpreted as computation. This allows partnership-specific processing data to be incorporated into the system at runtime, or removed to storage for use at another time. Since such behavior is difficult to be visualized in rather abstract models, we will provide code to describe one such solution. Due to its flexibility, usability and open source availability, we opted for Squeak/Smalltalk [4, 2] to do so.

Smalltalk has a powerful mechanism for expressing small units of computation as blocks and block contexts. A block is an object that embodies a sequence of operations. It is only executed after a value message is received by the block. A block context holds the dynamic state, such as parameter values, for execution of a block. Since blocks are, as anything else in Smalltalk, regular objects, they can be manipulated and stored as any other object.

A simple dynamic and customizable dispatch mechanism can be implemented by using a dictionary (similar to a hash table in Java) as a dispatch table. The dictionary allows us to store a set of associations where we use an association's key part to store the information necessary for dispatch selection, and the association's value part to store the code to be activated in the context of a dispatch. The code is provided as a block. On dispatch we would simply use the dispatch criteria to look up an associated block. Once obtained, this block is executed by simply sending a value message as mentioned above.

In our scenario, we can create a dispatch dictionary for each variation point with call prefixes or identifiers as keys and the appropriate provider-specific sequence of operations as values.

```
Object subclass: #RawCallRecordExchange
 instanceVariables: 'dispatchTable
                     defaultAction'
```

```
RawCallRecordExchange>>
initialize
  dispatchTable := Dictionary new.
  defaultAction := [:anObject | self error].

RawCallRecordExchange>>
addSelector: anArray
  action: aBlockContext
   dispatchTable add: anArray -> aBlockContext.

RawCallRecordExchange>>
removeSelector: anArray
  dispatchTable removeKey: anArray
                ifAbsent: [].

RawCallRecordExchange>>
dispatchOn: anArray with: aCallRecord
   (dispatchTable at: anArray
    ifAbsent: [defaultAction])
   value: aCallRecord.

RawCallRecordExchange>>
sendCallRecord: aCallRecord
   | prefix |
   prefix := aCallRecord callPrefix.
   self dispatchOn: { prefix. #before.
                      #sendCallRecord:. }
        with: aCallRecord.
   ...
   self dispatchOn: { prefix. #after.
                      #sendCallRecord:. }
        with: aCallRecord.

rawExchange := RawCallRecordExchange new.
rawExchange
 addSelector:
  { 01071 . #before. #sendCallRecord:. }
 action:
  [:aCallRecord |
   aCallRecord server
   authenticateUsingKerberos.
   aCallRecord authenticated
    ifTrue: [aCallRecord server connect]].
rawExchange
 addSelector:
  { 01071. #after. #sendCallRecord:. }
 action:
  [:aCallRecord |
   aCallRecord server disconnect]].
...
rawExchange sendCallRecord: aCallRecord.
```

Every time a new partner (an operator or a provider) needs to be added to the system, all dictionaries at our variation points are populated with the call prefixes or identifiers of the new partner and the code block specific to the new partner and the concerned variation point. If a dispatch for the new partner needs to be carried out, we simply look up and evaluate the code block associated with that partner. Terminating a partnership only involves removing all dictionary entries associated with the partner separated from. Starting or terminating relationships does not require a system rebuilt and exchange as in the static object-oriented case. With this dynamic registry and dispatch mechanism we can modify the running system without requiring elaborate failover mechanisms required by static methods.

4. ASPECT-ORIENTED VARIATION POINTS

As with the static and dynamic object-oriented implementations of our variation points, there are static and dynamic aspect-oriented implementations as well. In the following we see how aspect-orientation helps us to avoid the coding of explicit selections or explicit dispatches by using the dispatch mechanisms already built-in into an aspect-oriented composition platform.

4.1 Static Aspect-oriented

Figure 9 shows an aspect-oriented model of call records, kinds of calls, and the specifics needed by individual providers. In contrast to our object-oriented models, for instance the one in Figure 6, we decided to model only kinds of calls within a call records hierarchy. This hierarchy acts as the base system of our aspect-oriented model in which we express call record exchange and issuing invoices to customers.

Besides the reduction of code duplication, our aspect-oriented model also frees us from explicitly selecting a particular implementation for a specific operator or provider since such conditionals are hidden within the static compositions or residual test of the aspect composition.

Within a static aspect-oriented model the solution space must be known completely at compile-time, requiring us to know all possible combinations and their implications on our integrated system when it is deployed. If, after the deployment of our integrated system, we discover that we were wrong, we would have to perform similar corrective actions as the ones needed to update a statically modeled object-oriented system.

4.2 Dynamic Aspect-oriented

As with the dynamic object-oriented solution, the dynamic aspect-oriented solution benefits from the malleability of dynamic programming and composition platforms at runtime. Examples of such platforms are Steamloom, Prose, or AspectS. Since AspectS [3] is our research platform, we provide our sample code for this system.

We do not describe a different aspect model for the dynamic case since the static one discussed previously will do just fine. To implement our variation point for exchanging raw call-records, we first extend AspectS so that we can easily express aspects and their associated advice constructs that are active or inactive depending on a particular provider. This requires nothing else than allowing aspects to be provider-specific. We implement a provider-specific activation block and make it accessible to an advice qualifier object via the #providerSpecific attribute.

```
AsMethodWrapper class>>
providerSpecificActivator
  ^[:aspect :baseSender |
    | result receiver |
    receiver := baseSender receiver.
    result := aspect hasProvider:
    receiver callerNumber prefix.
    aspect := baseSender := nil.
    result] copy fixTemps
```

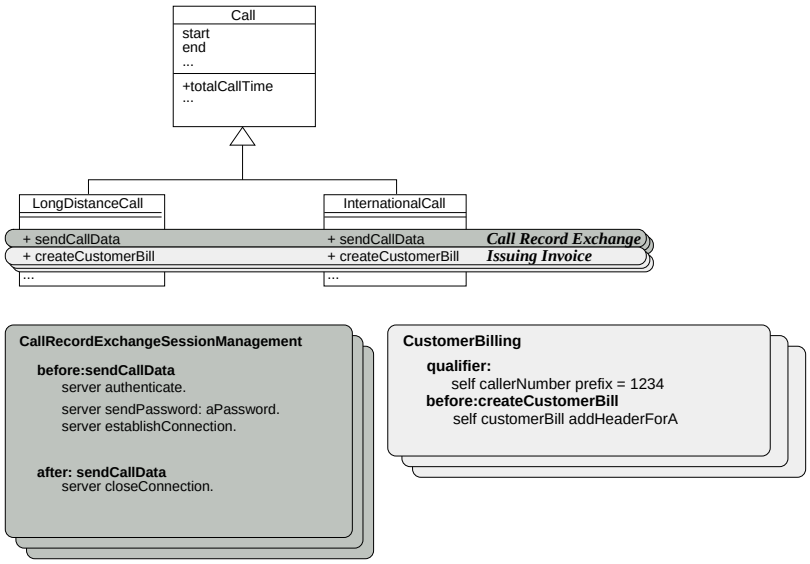

Figure 9: Aspect-Oriented Composition

More code is necessary to make the provider-specific advice qualifier attribute work. It requires the following changes to class AsAspect:

```
Object subclass: #AsAspect
  instanceVariableNames:
   'receivers senders
    senderClasses projects providers
    clientAnnotations advice installed'
  classVariableNames: ' '
  poolDictionaries: ' '
  category: 'AspectS-Aspects'

AsAspect>>
providers
  ^providers

AsAspect>>
providers: anIdentitySet
  providers := anIdentitySet.

AsAspect>>
initialize
  self receivers: IdentitySet new;
  senders: IdentitySet new;
  senderClasses: IdentitySet new;
  projects: IdentitySet new;
  providers: IdentitySet new;
  clientAnnotations: IdentityDictionary new;
  advice: nil;
  installed: false

AsAspect>>
addProvider: aProvider
  ^self providers add: aProvider

AsAspect>>
removeProvider: aProvider
  ^self providers remove: aProcess
                 ifAbsent: []

AsAspect>>
hasProvider: aProvider
  ^self providers includes: aProvider
```

Now we can implement our raw call record exchange, making use of our newly defined advice qualifier attribute. Here it is interesting to note that the association key we used in the dynamic object-oriented example to select the appropriate dispatch is represented as a provider specific activator in our dynamic aspect-oriented example. The values of the previously used associations (code blocks) can now be found in our AspectS code as the before and after blocks of the respective advice constructs.

```
AsAspect subclass: #RawCallRecordExchangeAspect

SessionManagementAspect class>>
prefix: anInteger
   ^self new
    addProviderPrefix: anInteger

RawCallRecordExchangeAspect>>
adviceSendCallDataProvider
   ^AsBeforeAfterAdvice
     qualifier: (AsAdviceQualifier
       attributes: { #receiverClassSpecific.
                     #providerSpecific.})
     pointcut: [{ AsJoinPointDescriptor
       targetClass: InternationalCall
       targetSelector: #sendCallData.
      AsJoinPointDescriptor
       targetClass: LongDistanceCall
       targetSelector: #sendCallData. }]
     beforeBlock: [receiver server
        authenticateUsingKerberos.
       receiver server authenticated
         ifTrue: [receiver server connect]]
     afterBlock: [receiver server disconnect]

aspect := RawCallRecordExchangeAspect
   prefix: 12345.
aspect install.
```

Looking at the code above we can see that the dynamic aspect-oriented solution does not require an explicit dispatch to be provided by a developer since this dispatch is intrin-

sic to all aspect-oriented platforms. Whenever necessary, the underlying aspect system accesses the activation block provided by us, evaluates this block, and, depending on the outcome of this evaluation, activates the associated advice code or not. So, in addition to all the flexibility gained by our dynamic object-oriented solution, we also achieve simplification of our code by the utilization of a hidden but well known and proven system-provided dispatch mechanism.

5. SUMMARY

In this paper we provide a scenario taken from telecommunications to motivate the need for dynamic aspect-oriented programming languages and systems. Our scenario describes how constantly changing relationships between operators and call-by-call providers affect their system integration requirements (here in the context of settlement), and how one of the most important of these requirements – keeping system downtime to a minimum – can be supported by employing dynamic composition in general, and dynamic aspect-oriented composition in specific. While focusing on dynamic aspect composition, we do not argue for or against the merits of aspect-orientation in general; this is done adequately elsewhere [1].

	OO	AO
static	implicit dispatch fixed set of providers explicit selection	implicit dispatch fixed set of providers
dynamic	explicit dispatch	implicit dispatch

Table 1: Properties of the Proposed Solutions

Table 1 summarizes the properties of the proposed solutions ranging from static and dynamic object oriented techniques to dynamic aspect-oriented composition. It is no surprise that all decisions made in advance of building a software system can be modeled, implemented, and optimized via early-bound object- or aspect-oriented systems. Later decisions, more precisely decisions made after the construction and deployment of a software system can be modeled, implemented, and, most importantly to us, adapted only in late-bound object- and aspect-oriented systems. In addition to adaptability, an aspect-oriented system has many other advantages enabling the support of unanticipated software evolution[5]. In our example, use of the built-in dynamic dispatch mechanism allows us to avoid explicit and critical dispatching code.

6. ACKNOWLEDGEMENTS

We would like to thank Matthias Wagner and Monika Fuchs for their valuable discussions and contributions.

7. REFERENCES

[1] http://www.aosd.net.

[2] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[3] R. Hirschfeld. AspectS - aspect-oriented programming with squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays 2002*, LNCS 2591, pages 216–232, Erfurt, 2003. Springer.

[4] D. Ingalls, T. Kaehler, J. Maloney, W. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. *ACM SIGPLAN Notices*, 32(10):318–326, Oct. 1997.

[5] G. Kniesel, J. Noppen, T. Mens, and J. Buckley. Unanticipated software evolution. *Lecture Notes in Computer Science*, 2548:92–107, 2002.