ATOLL: Aspect-Oriented Toll System[*]

Luis Daniel Benavides Navarro[1], Christa Schwanninger[2], Robert Sobotzik[2], Mario Südholt[1]

[1]OBASCO group
EMN-INRIA, LINA
Ecole des Mines de Nantes
4 rue Alfred Kastler, 44307 Nantes cedex 3, France
{lbenavid,sudholt}@emn.fr

[2]Software & Engineering, Architecture
Siemens Corporate Technology
Siemens AG
Otto-Hahn-Ring 6, 81730 Munich, Germany
Christa.Schwanninger@siemens.de, R.Sobotzik@gmx.de

ABSTRACT

Product line development places emphasis on quality attributes like understandability, maintainability, reusability and variability. Better modularization techniques like aspect-oriented programming are supposed to improve these attributes.

In the context of an industrial case study in the domain of infrastructure software for toll systems from Siemens AG, Germany, we have investigated how OO designs can be enhanced using AO techniques. We have explored, in particular, how sequential crosscutting concerns can be modularized using AspectJ and how distributed ones can be modularized using AWED, a system that features aspects with explicit distribution. Concretely, we show how sequential and distributed aspects improve the implementation of the charge calculation functionality that is central to real-world tolling systems.

General Terms

Design, Experimentation, Languages.

Keywords

Software Product Lines, Aspect-oriented Software Development.

1. INTRODUCTION

Automatic tolling systems are becoming popular in many European countries for charging toll for the usage of roads and motorways. There are considerable differences between existing toll system installations. Light-weight systems rely mainly on road-side equipment, while for advanced solutions vehicles carry sophisticated embedded on-board units (OBU) featuring position determination using GPS and mobile communication that interact with powerful back-end servers (electronic tolling back office, ETBO). In the latter case, road side equipment and OBUs are used to record the position data of vehicles. The calculation of toll charges can be distributed between the OBU and the back end ETBO in different ways. Depending on the capability of the embedded device the position data can be processed on the OBU itself and the resulting information is sent to the back end server for billing, or position data is immediately transmitted to the back end server for further processing. Despite the variability, core tasks like road recognition, charge calculation and billing are essentially done in a similar manner for every instance of a toll system. Therefore, companies like Siemens build product lines for toll systems to provide cost-effective toll systems that exploit such similarities while supporting the necessary variability.

Software product line engineering aims to reduce development time, effort, cost and complexity by taking advantage of the commonality within a portfolio of similar products [1]. Members of a product line differ in the number and the specific properties of features they include. Features are not always well localized but often crosscut multiple software artifacts in every stage of the development life cycle. As a result, effective variability management is a predominant engineering challenge in software product line development.

For the toll system product line a number of crosscutting concerns have to be accommodated by the system design like variability in the tariff models, different ways how toll charges are calculated and the distribution between server and on-board units of road recognition and charging responsibilities. Aspect-oriented software development should be useful in this context because it improves the way software is modularized by providing modularization constructs for the encapsulation of crosscutting concerns.

We have implemented the toll system in an industrial case study using AOP whenever it improved the design. To make a realistic case and taking the degree of adoption of AOSD in the industry into account, the main design technique used for the toll system demonstrator is still plain object-orientation. All features were first implemented in pure OO and only enhanced or replaced by aspects when the original implementation showed limitations in fulfilling quality attributes that could be addressed using aspects. The two concerns we report on here arise from the variability in the charge calculation, a central functionality of any toll system.

[*]Work supported by AOSD-Europe, the European Network of Excellence in AOSD (`http://www.aosd-europe.net`)

Charge calculation is split into several sub tasks that are executed using a pipes and filters architecture. The processing steps can be performed in a pipelined fashion either on the OBU or on the ETBO. Depending on how the charge calculation is distributed between the two main components of the toll system architecture, different steps of the processing pipeline need to be adapted. This variability leads to two challenges that need to be considered in the implementation, both of which are motivated from real-world application scenarios, we are interested in. First, communication of data between different steps may be managed differently and, second, the charge calculation pipeline may be split, in principle, at arbitrary elements to switch charging between OBU and ETBO that are physically distributed entities. In both cases, parts of the charging pipeline have to be modified to accommodate these adaptations.

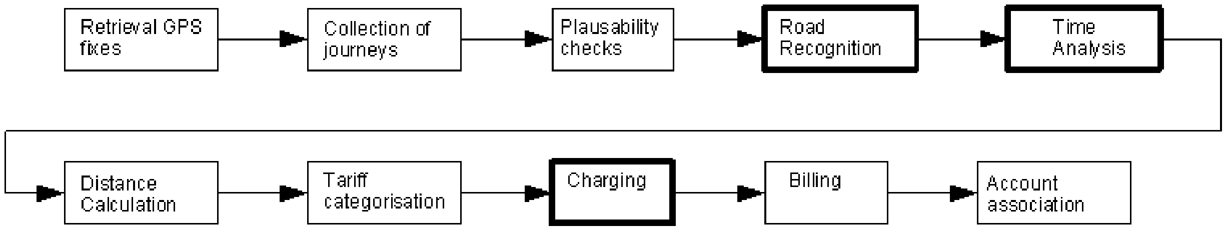


Figure 1: Data flow in the Toll System

2. THE TOLL SYSTEM FAMILY

The key functionality of the toll system is the usage of GPS information to track vehicle movement on roads in order to charge vehicle owners. Depending on the capabilities of OBUs, position data or billing data is transmitted from the OBU to the ETBO. Depending on stakeholders' requirements the responsibility split between OBU and ETBO can vary.

An OBU is responsible for collecting information on each journey of a vehicle. The time interval between switching on and off the ignition is treated as one distinct journey. During this time incoming GPS fixes, i.e., position measurements, are associated to this journey. Additionally the journey data will be extended with predefined properties like trailer attached, ecological vehicle classification, towed or carried. After finishing a journey several plausibility checks have to be done, like comparing start and end time of the journey. Each GPS position is compared to a map to determine the corresponding kind of road, e.g., a simple street or a motorway. Depending on the kind of roads the list of GPS positions will be split into sections. Afterwards the data is passed to do additional time analysis on the journey to split the sections once again, e.g., depending on the attributes peak or non-peak time. Using streets during peak-time is usually more expensive. The outcome is a set of fine granular sections with respect to kinds of roads and driving time. Then, the distance driven in each of these sections will be calculated. Depending on a specific tariff model, the charge of a journey is the sum of sections' costs. To associate the charge with the account holder, the owner of the OBU has to be identified. The journey charges are invoiced to the customer after a specific period of time.

3. ARCHITECTURE

In this section we describe the architecture of our system by first presenting its basic OO structure, followed by two crosscutting concerns for which AOP support has proven valuable.

3.1. Basic OO Design

The basic architecture is data flow driven; it essentially follows the Pipes & Filters [2] architectural pattern. Fig. 1 shows the overall data flow within the demonstrator.

Depending on the capabilities of an OBU, charge calculation can be done either on the OBU or on the ETBO. The main data flow stays the same, but the distribution of work between OBU and ETBO changes. Currently, we support the following charge calculation models[1]:

Central model: Raw GPS positions are transmitted to the ETBO. Roads are identified and the charges are calculated on the ETBO.

De-central model: The OBU identifies roads and calculates charges. This requires that the OBU is aware of the tariff model and the road map. Current costs can be displayed by the OBU in real-time.

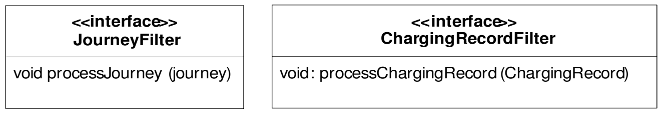
The filters that are mainly responsible for the charge calculation process are highlighted in Fig. 1 by boldly-stroked boxes:

- **Road Recognition** identifies road usage based on GPS positions.
- **Time Analysis** associates the time stamps with peak or non-peak time.
- **Charging** is responsible for the charge calculation.

Depending on the charge calculation model, either only raw GPS data is transmitted to the ETBO once a journey has ended, or journey data is constantly processed to display the current cost in real time. This has the effect that the data structures that have to be passed between processing steps are either simple GPS lists or GPS data enhanced with pre-calculated information, such as

[1] Other distributions of work between OBU and ETBO are possible: many of these, however, are not useful in practice, e.g., distributions that switch multiple times between OBU and ETBO.

the breakdown into sections. This would result in two interfaces for each affected filter, one for sending GPS data and one for sending calculation results in addition. One way to minimize the complexity to address this variability is to use the more complex data structures necessary for the de-central model also for the central model and deal with the variability only in the business logic in a way such that it can be used for both models. This means that the implementation of each filter should be the same, no matter whether it will be executed on the OBU or on the ETBO. The usage of stable interfaces to encapsulate the business logic allows an exchange of the calculation model with minimal effort; all variations of a filter have to implement the same interface.



<<interface>>
JourneyFilter
void processJourney (journey)
<<interface>>
ChargingRecordFilter
void: processChargingRecord (ChargingRecord)

Figure 2: Filter Interfaces

Filters are not hard wired with each other, rather we use dependency injection to connect the processing steps. A factory is responsible for creating the filters and setting up their next neighbour (conforming to a common interface) as a parameter. Data passing to the following filter is managed by each filter itself. For this loosely coupled control model to work, a common data structure usable by all filters is necessary. The benefit of using dependency injection is that it establishes an abstraction via interfaces, which reduces the dependency on components and facilitates a plug-in based architecture [4].

Because the charging calculation has to support two different data types between the filters (depending on whether plain GPS journeys or journeys containing pre-computed information is to be transmitted), the filters have to implement at least one of the following interfaces:

The class ***Journey*** is used as a data container to store raw GPS positions plus predefined properties like trailer attached.

The class ***ChargingRecord*** contains a reference to a particular journey and information about the journey's pre-calculated sections with respect to kinds of roads and time.

In the case of central charging the computation within each filter is done on a completed journey. Therefore each filter has to be passed only once because all data is available at the beginning of the computation chane. But in the case of de-central charge calculation the whole filter chain needs to be traversed for each new GPS position to calculate the current charging in real-time.

As mentioned above the data structures including all components between “Retrieval of GPS fixes” and “Charging” are chosen such that they fit both charge calculation variations which causes parts of the data structures to be empty or not complete in the de-central case.

Without optimization, this proceeding introduces redundancy and method calls that are useful only in one of the variants. E.g. at some point in the processing journeys are passed to a data buffer for sending them to the remote server. This makes sense for the central charging model, but in the de-central case the data buffer refuses to accept data before it is complete. Only compete journeys will be sent to the ETBO. These kinds of redundancy should, however, be avoided as far as possible to increase the performance and code understandability. At the same time as much code as possible should be reused unchanged for both variants.

3.2. Data Passing Concern

Handling of the data structures between pipeline filters is subject to two crosscutting concerns:

- Transition between the filters (pipe logic).
- Keeping track of processing states.

The latter point is due to the decision to minimize the variations in data structures. In every charge calculation variant whole journeys are passed between filters, not only single positions. Therefore it is necessary for each filter to keep track of the last position that was already processed by this filter to avoid rerun of road recognition, time analysis and charging in the de-central case. To keep this tracking functionality centralized, the class `Journey` was extended by providing getter-methods for each filter to get its specific last processed position in the GPS list. Adding or removing filters within the charge calculation model therefore always would cause changes in `Journey`.

The passing of data between two filters always follows the same algorithm: testing for “null” references (in order to ensure that structures that are used for holding the processing results have already been created), getting the last processed position before processing and passing the data to the next filter after processing.

Since the data transfer functionality is identical in every processing step and only necessary to accommodate the variability in charge data processing, it would be good to have it modularized in one place. First it reduces the error-proneness because of reusing the data transfer functionality and second optimizations could be applied to it, e.g. asking for the last processed position is not necessary in central charge calculation since all processing is done in pass through.

Encapsulating this concern with classic OO could be done in several ways. For example the `JourneyFilter` and `ChargingRecordFilter` could be classes instead of interfaces and implement the common behavior, while each deriving filter implements the specific logic, an implementation of the template method pattern [11]. Since there are filters that have to implement both interfaces, this is not possible, so a separate class could hold the common behavior and be inherited from. Methods for the pre- and post functionality of the processing steps would be the only functionality of this class, though. Also this would require some mechanism for calling the right getter method from the Journey, since these getters need to be specific to each filter. The information about the processing position in the Journey of each filter could be directly stored in each filter, which would pollute the filter with information only relevant for the de-central case and deprive us of the opportunity to do optimizations.

Modularizing the pre-and post functionality in an aspect has the effect that the logic only exists once and can be reused for each component. A suitable aspect can be defined by capturing the respective method calls of the interface and would thus be really simple and stable. Concretely, the aspect is defined per target, an instance of the aspect thus existing for each filter.

Also the issue with the filter specific processing state variables and the associated getters in the Journey class can be solved easily. The solution to this is to embed these variables in the aspect mentioned above. Since an instance of the aspect exists for each filter, only a single variable is necessary to keep track of the last processed position. Thus adding or removing filters within the charge calculation chain does not cause any need for adaptation.

3.3. Distribution Concern

Partitioning of the application into a distributed system is an orthogonal concern to that of data passing in the pipelined charge calculation. As mentioned before we want a solution flexible enough to incorporate variability into the distribution concern. Thus the specific point where the calculation chain is partitioned can be changed at will, deciding which steps are performed in the OBU and which are performed in the ETBO.

Distribution has been shown before to be a crosscutting concern in distributed applications implemented using OO techniques (see, e.g, [9] for such evidence in the domain of replicated caches). Furthermore, it has been shown that encapsulating such a concern using sequential AOP languages presents limitations and requires preparation of the base code (harming encapsulation and separation of concerns) [8]. To avoid these problems the AWED aspect system ("Aspects with Explicit Distribution") [9,10] provides explicit support for distribution. In the context of the toll system, AWED allows to modularize the code for distribution of arbitrary sets of pipeline segments and the code for communication between them in one aspect.

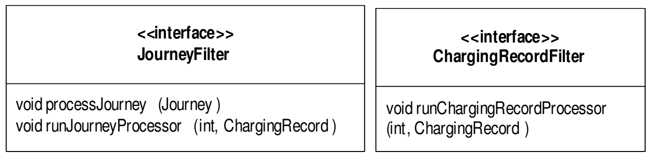
4. IMPLEMENTATION

We now present how we implemented the two crosscutting concerns using AO techniques.

4.1. Data Passing Concern

In the original OO design the functionality for checking data types and the position up to where they are already processed is implemented in each filter. In our AO-refactored version, two aspects are required (for each interface) to extract the pipes logic from the filters, thus only the business logic remains in the filters.

To accomplish the extraction of the pipe logic, both interfaces displayed in figure 3 have to be adapted.



<<interface>>
JourneyFilter
void processJourney (Journey)
void runJourneyProcessor (int, ChargingRecord)
<<interface>>
ChargingRecordFilter
void runChargingRecordProcessor (int, ChargingRecord)

Figure 3: Filter interface using aspects

While `processJourney` is used as a hook, the business logic of classes implementing the interfaces is coded in `runJourneyProcessor` method and `runChargingRecordProcessor`. Note that the `ChargingRecord` method in the `JourneyFilter` has a parameter of type `ChargingRecord`. Such parameter is used to allow delegation of creation of an object of type `ChargingRecord` to the aspect. Thus the method signature allows the aspect to communicate with the base program in a more flexible way.

Before the business logic of each filter being executed the aspects handle the non-functional requirements, e.g. like getting the last processed position. After execution of the business logic the same aspect delegates the data to the next neighbor. Filters whose methods should not be captured by the aspect have been labelled with the annotation `@Exclude`. The reason is that the logic of some filters differ comparison to the other, like the in memory database. Although the application still would work correctly without excluding them, keeping track of processed positions and testing for "null" references are not necessary and decrease the performance.

The code shown below in figure 4 is the source code of the "Road Recognition" filter in the OO implementation. There the non-functional code for the filter pattern implementation is entangled with the functional code of the filter. Figure 5 shows the same class once the application has been refactored using aspects. Note that business logic code is present only in the filter implementation.

```
public class RoadRecognitionFilter implements JourneyFilter{
// …
 public RoadRecognitionFilter (double tolerance, ChargingRecordFilter nextFilter) {
    this.tolerance = tolerance;
    this.nextFilter = nextFilter;
 }

 // businessLogic
 public void identifyRoad (int lastProcessedPosition, Journey journey) {
    // code of the businesslogic
 }

 // non-functional requirements
 public void processJourney (Journey journey) {
    // null testing
    // get last processed journey
    // call identifyRoad(..)
    // pass data to the next filter
 }
}
```

Figure 4: Road recognition filter implementation in the OO solution

```
public class RoadRecognitionFilter implements JourneyFilter{
  // …
  public RoadRecognitionFilter (double tolerance, ChargingRecordFilter nextFilter) {
     this.tolerance = tolerance;
  }

  // businessLogic
  public void identifyRoad (int lastProcessedPosition, Journey journey) {
  // code of the businesslogic
  }

  // non-functional requirements
  public void processJourney (Journey journey) {}
}
```

Figure 5: Road Recognition Filter after extracting the aspect

```
01 public aspect JourneyFilterAspect pertarget(execution  JourneyFilter+.new(..))){
02 int lastProcessedPosition = 0;
03
04 pointcut getSuccessor(ChargingRecordProcessor nextFilter) :
05    execution( *.new(..))  &&   args(..,nextFilter)   &&this(JourneyProcessor);
06
07 pointcut preparationForProcessing(Journey journey, JourneyProcessor
currentFilter) :
08    execution (public void JourneyProcessor.processJourney(Journey))
09    && args(journey)
10    && target(currentFilter)
11    && !execution (@Exclude * *(..);
12
13 after(ChargingRecordProcessor nextFilter) : getSuccessor(nextFilter){
14    // get reference of successor;
15 }
16
17 void around (Journey journey, JourneyProcessor currentFilter) :
18    preparationForProcessing(journey, currentFilter){
19    // testing of „null“ reference; get last processed state
20    currentFilter.runJourneyProcessor(lastProcessedPosition, record);
21    // pass data to the next filter
22 }
```

Figure 6: JourneyFilterAspect

The pipe functionality can be extracted from the filters and encapsulated into a single aspect as shown in figure 6. Because the aspect is declared as `pertarget` (line 1), an instance of the aspect exists for each filter which is implementing the interface `JourneyFilter`. Lines 4 to 5 define the pointcut to capture the reference of the next filter. The `around` advice of the pointcut `preparationForProcessing` (lines 7 to 11) prepares the filter to execute his business logic, i.e. testing for “null” reference and getting the last processed. After executing the business logic (line 20) the data will be passed to the next filter.

The benefit of this aspect is encapsulation of the same functionality that is applied in several places, thus improving maintainability and the correction properties of the code.

4.2. Distribution Concern

Evolution of the charge calculation pipeline into a distributed pipeline can be achieved using AWED’s remote pointcuts, distributed advices and group support. The basic idea is that AWED aspects include a declarative description of the distributed partitioning at specific points (i.e., method invocations) of the pipeline.

Figure 7 shows an aspect implementing pipeline distribution using AWED. It implements the distributed partitioning of the calculation pipeline after journey(s) have been gathered. Lines 2 to 5 present a pointcut definition that matches all the calls to the method `receiveJourneys` (in the class `ec.noe.tollsystem.etbo.Communication`) that are executed in a host that belongs to the group OBUs (through use of the term `host("OBUs")` in the pointcut). The pointcut definition also restricts the execution of the advice to the group ETBOs (by use of the term `on("ETBOs")`).

Note that the above aspect represents a general aspect for partitioning, the names of the groups are just selected for understandability in the context of the example. But they represent two groups of host, one where the call originated and the second where the method is executed. The `on` construct can be extended with a selection policy, e.g., to implement high availability cluster support.

```
01   aspect DistributionAsp{
02   pointcut partitioningMethod():
03     call(* etbo.Communication.receiveJourneys(..))
04            && host("OBUs")
05        && on("ETBOs");
06
07   around(): partitioningMethod(){
08      System.out.println("Remote execution");
09      LocalRegistry.getInstance().getCom().receiveJourneys(
10         (List) thisJoinPoint.getArgumentsArray()[0]);
11      return new Object();}
12  }
```

Figure 7: Distribution aspect using AWED.

Performance of our distributed system can be improved using an asynchronous advice, i.e. annotating the around advice with *asyncex*. This will return immediately after the distributed call a future object in the base application. Thus booth the OBU and the ETBO will run asynchronously and they will synchronize only if the future object receives a request to return an actual value.

Another important feature of distributed systems that can be implemented easily with AWED is failure safety. Using groups and a policy extension for the `on` pointcut. The expression `on(“ETBOs”, org.awed.policy.roundrobin)`, in the pointcut definition, will tell the application to execute the advice in the hosts bellowing to the ETBOs group, but it will restrict the selection of the executing hosts by the round-robin policy. Note that the system will support intermediately a cluster deployment for the ETBO server.

To complete the AWED implementation a configuration aspect is needed. The objective of such an aspect is to create the OBUs and ETBOs groups. This is a specialized aspect that is highly coupled with deployment decisions concerning the to-be-run application. Such a configuration aspect is typically defined based on configuration files, here we define it, for simplicity, based on method invocations. Figure 8 shows the implementation of the configuration aspect. The aspect basically includes a host in an specific group depending of the invoked method: one for the OBUs group and the other for the ETBOs group. The advice is executed locally wherever the corresponding method is called. Note that groups are created as they are requested, no previous group creation is required.

5. EVALUATION OF OO vs AO

The toll system demonstrator currently consists of 40 classes and 7 aspects and is still under development to introduce communication middleware adaptability.

The demonstrator was developed in several increments. For every increment the additional functional and non functional requirements first were designed and implemented using OOP only with Java. The resulting system then was analyzed. In case of weaknesses, e.g., complex designs necessary to accommodate variability, we investigated alternative designs using AOP. These designs were then finally implemented. The reason for this proceeding was, that industrial systems most frequently are not (yet) designed with aspects up front. AOP is employed for modularizing development concerns, e.g. tracing. Only if there are very clear benefits, AOP is also used for modularizing domain specific concerns.

```
aspect AppConfigurator{

  pointcut confAsClient():
    call(void OBU.init(..))
        && host(localhost)
        && on(localhost);

  pointcut confAsServer():
    call(void ETBO.init(..))
    && host(localhost)
    && on(localhost);

  after():confAsClient(){
    System.out.println("Adding Host to group OBUs");
    addGroup("OBUs");}

  after():confAsServer(){
    System.out.println("Adding Host to group ETBOs");
    addGroup("ETBOs");}
}
```

Figure 8: Configuration aspect using AWED.

5.1. Data Passing Concern

One of the goals for the design of the demonstrator was to reuse as much code as possible for different variations in the product family, in this example specifically for both charging variants. A pure OO design would either be more complicated and less open for optimizations or less efficient and intuitive. Changes, e.g. introducing new filters, would require to make changes to code in the Journey class, which is not obvious to a developer. With the AO solution only the business logic of a new filter has to be implemented, the aspect cares for pre- and post processing transparently without change.

5.2. Distribution

Implementation of a distributed version of the tool system could be achieved using existing distributed frameworks and middleware, e.g., J2EE[12], CORBA[13], Spring[14], Internet Communications Engine (ICE) [15] and several other research and industrial approaches. These approaches each provide their own model of distributed objects, specialized APIs and specific mechanisms for RPC-based communication. However, none of these approaches directly address the problem of distribution as a crosscutting concern and even though some of them provide AO mechanisms, e.g., Spring AOP, JBOSS AOP [16], these mechanisms are not intended to interact directly with the distribution model. Furthermore, these frameworks require the preparation of base code in advance in order to support the specific mechanisms provided, thus distinguishing between local and distributed objects. Our experiments using AWED propose an additional, and different, level of abstraction where distributed concepts are considered explicitly in the aspect language and the original model for objects is maintained without affecting the original semantics of the application and without need of advance preparation of code.

Another set of approaches that can be employed to attain variability in the distribution implementation, is the use of systems for automatic partitioning, e.g., Addistant[17], J-Orchestra[18]. Those systems propose a technique for separated specification of the distributed behavior. In general, at configuration time, first a specific distributed schema for runtime is defined, then the application is partitioned (e.g., modifying the bytecode of a Java application). These techniques do address separation of crosscutting concerns but they do not provide the expressive power harnessed by AWED aspects. They just provide a declarative language for partitioning that is applied statically at deployment time. They do not allow the resulting system to be modified dynamically as AWED allows to do.

As mentioned before the main objective of our work was to achieve flexible variability of the implementation. This objective applies to the middleware layer in the distributed implementation. Currently, the AWED implementation relies on JGroups to manage group communication, the AWED model can, however, relatively straightforwardly be extended to support other distributed middleware.

6. Conclusion and future work

Software product lines attempt to reduce development time, effort, cost and complexity by taking advantage of the commonality within a portfolio of similar products. Therefore it is quite important to acquire as much commonality as possible to reduce the amount of variability to the required minimum but this leads always to a trade-off between customizing and reuse [7].

In this paper have we considered a particular product line for the development of automated tolling systems. We have motivated that the management of data flows in such a product line can benefit from aspect-oriented programming and have shown how to modularize two specific aspects, data passing and distribution control, benefit from Aspect-Oriented Programming.

Concretely, we have shown how an OO base application has been refactored using AspectJ for modularization of the data passing concern and AWED, an aspect system providing explicit distribution, for the distribution concern.

Future work will extend the demonstrator with new features to see if aspects support the unanticipated evaluation of a product line. Furthermore, we plan to develop middleware specific optimization strategies to better support AO-specific features for the adaptation of such product line architectures.

7. References

[1] Clements, P., and Northrop, L. Software Product Lines: Practices and Patterns. Addison-Wesley, 2001

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture – A System of Patterns, John Wiley & Sons, 1996

[3] Description of the Industrial Demonstrator Toll System, C. Schwanninger, I. Groher, M. Kircher, R. Chitchyan, A. Sampaio, A. Rashid, AOSD Europe, 10.10.2005

[4] M. Fowler, Inversion of Control Containers and the Dependency Injection pattern, http://www.martinfowler.com/articles/injection.html (Date: 01/22/2007)

[5] R. Laddad, AspectJ in Action, Manning, 2003

[6] G. Kiczales, J. Lamping, A. Nemdhekar, C.Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In Proceedings ECOOP'97, LNCS 1241, pages 220-242. Springer, 1997

[7] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering, Springer 2005

[8] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In Proceedings of OOPSLA'02, pages 174–190. ACM Press, 2002.

[9] L. D. Benavides Navarro, M. S¨udholt, W. Vanderperren, B. D. Fraine, and D. Suv´ee. Explicitly distributed AOP using AWED. In AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development, pages 51–62, New York, NY, USA, 2006. ACM Press.

[10] L. D. Benavides Navarro, M. S¨udholt, W. Vanderperren, B. D. Verheecke. Modularization of distributed web services using AWED. In Proceedings of On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE. 8th Int. Symposium on Distributed Objects and Applications (DOA'06), pages 1449—1466, Montpellier, France, 2006. Springer.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software.: Elements of Reusable Object-Oriented Software, Addison-Wesley 1995.

[12] J2EE Platform Specification 1.4. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf. Sun Microsystems, 2003.

[13] CORBA. http://www.omg.org. The Object Management Group (OMG).

[14] Spring Framework. http://www.springframework.org/ .

[15] The Internet Communications Engine. http://www.zeroc.com/ice.html.

[16] JBOSS AOP. http://labs.jboss.com/portal/jbossaop.

[17] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of legacy java software. In European Conference on Object-Oriented Programming 2002 (ECOOP 2002), LNCS 2072, pages 236–255. Springer, 2001.

[18] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In European Conference on Object-Oriented Programming 2002 (ECOOP 2002). Springer, 2002.