A method for aspect mining using production rules, dependency graphs and two-level grammars

Amir Abdollahi Foumani

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

September 2005

Your file Votre référence
ISBN: 978-0-494-16254-5
Our file Notre référence
ISBN: 978-0-494-16254-5

ABSTRACT

A method for aspect mining using production rules, dependency graphs and two-level grammars

Amir Abdollahi Foumani

Adopting aspect-oriented technologies requires revisiting and restructuring the entire traditional software lifecycle in order to identify and represent occurrences of crosscutting during software requirements engineering and design, and to determine how these requirements are composed. In this research, we propose an aspect mining approach to identify and model crosscutting concerns (aspects) by restructuring the meta-level at the breakpoints of: requirements elicitation, analysis, design, and implementation activities. The main purpose of this research is twofold: (1) "isolation" of crosscutting concerns in the early steps of software life cycle (2) identification of crosscutting concerns in legacy systems. The proposal outlined in this research illustrates a method to reformulate existing object-oriented artifacts by mining and explicitly modeling crosscutting concerns. In this method of restructuring we represent the semantics of UML artifacts by using a set of production rules, which refer to object definitions and relationships metadata. The result of our proposed restructuring process is an aspect-oriented design whereby aspects may be introduced into the object-oriented design or certain classes may be restructured as aspects.

Table of contents

Table of Figures

1. Introduction

Despite the success of object-orientation in the effort to achieve separation of concerns, certain properties in object-oriented systems cannot be directly mapped in a one-to-one fashion from the problem domain to the solution space, and thus they cannot be localized in single modular units. This is due to the fact that the decomposition of requirements is performed along one axis (the notion of class). This tyranny of dominant decomposition [16] dictates that the implementation of certain concerns would cut across parts of the system. Such concerns are called crosscutting concerns or aspects. Requirements may be implemented by a number of different design decisions and artifacts. Some object-oriented design decisions decompose the problem domain into better modular units such that the crosscutting phenomenon is minimized. However, the phenomenon may not be totally eliminated from an object-oriented model. The symptoms imposed by this phenomenon manifest themselves as (1) the scattering of concerns across the decomposition hierarchy of the system and (2) the tangling of concerns in modular units. As a result, developers are faced with low level of cohesion of modular units, strong coupling between modular units, low level of reusability of code, low level of adaptability and difficult comprehensibility resulting in programs that are more error prone.

New software engineering technologies are emerging to allow richer specifications of programs and better modularization of these specifications. Aspect-Oriented Programming (AOP) originally proposed by [5] is a collective term that refers to a growing family of approaches and technologies that provide solutions to separate and implement crosscutting concerns by providing a second axis of decomposition that enables the identification and separation of core functionality and crosscutting requirements and thus a better modularization for the specified requirements. The description of crosscutting leaves space for its manifestation in disciplines that precede programming. Aspect-Oriented Software Development (AOSD) has extended AOP to provide a systematic support for identification, separation, representation and composition of crosscutting requirements as well as mechanism that make them traceable throughout software development process.

The main purpose of this project is twofold:

1. Isolation of crosscutting concerns in the early steps of software life cycle
2. Identification of crosscutting concerns in legacy systems.

In this respect we try to define an aspect mining approach that uses the same rules and techniques for aspect mining throughout the entire software life cycle. At the analysis stage by analyzing "analysis artifacts" we distinguish potential aspects or aspect candidates and then with applying the same techniques on "design artifacts" we trace these aspects to the design level with different granularity. At the design level we may introduce some more aspect candidates. Aspect mining can be continued to the implementation stage for identifying missing aspect candidates from analysis and design stages. For analyzing implementation artifacts we introduce a new method of parsing. With this new method in addition to typical parsing of the source code we are able to analyze the object relationships and definitions and also identify scattered and duplicated codes. This approach also identifies aspect candidates from the source code. As a result by just compiling source codes of a legacy system we are able to identify aspect candidates. In Figure 1 we illustrate our proposal for a process of restructuring an object-oriented design into an aspect-oriented design. According to this proposal for a single set of requirements we may have different object-oriented designs and we show that by using production rules and dependency graphs we are able to restructure these object oriented designs to an aspect-oriented design. At the end of this restructuring process for each object-oriented design we will have a corresponding aspect-oriented design with the same set of aspects. According to this proposal any object-oriented design with any degree of modularity can be restructured to a corresponding aspect-oriented design by analyzing object-oriented design artifacts.

We define the following:

1. `R: {r1, r2, r3, ...}`, a set of requirements.
2. `D: {D1(m1), D2(m2), D3(m3), ...}`, a set of object-oriented designs, where `m1`, `m2` and `m3` constitute certain degree of design modularity. With degree of modularity we mean a general metrics for evaluating the degree of cohesion of modular units, degree of coupling between modular units, degree of reusability of code, degree of adaptability and difficult comprehensibility. So if `m1>m2` then we evaluate `D1` as a better object-oriented design than `D2`.
3. `E:` $\{e_1, e_2, e_3, ...\}$, a set of entities where each member corresponds to an object-oriented design `Di` which define real world applications.
4. `C: {c1, c2, c3, ...}`, a set of potential crosscutting concerns.
5. `A: {A1, A2, A3, ...}`, a set of aspect-oriented designs that are the result of restructuring `D1, D2, D3,...` object-oriented designs.

Our proposal is based on the deployment of dependency graphs and production rules in order to be able to restructure an object-oriented design say `Di` to an equivalent aspect-oriented design say `Ai` such that:

`E:` $\{e'_1,\ e'_2,\ e'_3,\ \ldots\}$ is a set of entities that together with `Aspect: {a1, a2, a3, ...}` as a set of aspects define the real world applications for the same set of requirements. Our argument is that different restructurings from an object-oriented to an aspect-oriented context would yield the same set of aspects.

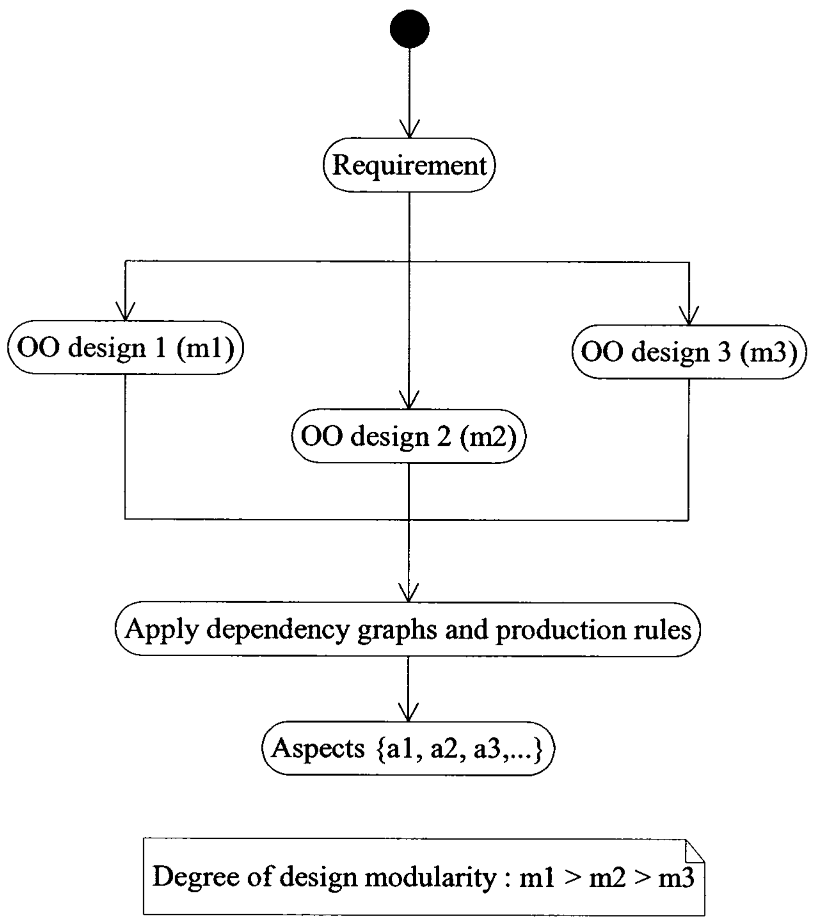

Requirement
OO design 1 (m1)
OO design 2 (m2)
OO design 3 (m3)
Apply dependency graphs and production rules
Aspects {a1, a2, a3,...}
Degree of design modularity : m1 > m2 > m3

Figure 1. Restructuring of different object-oriented designs into aspect-oriented design

Adopting aspect-oriented technologies requires revisiting and restructuring the entire traditional software lifecycle in order to identify and represent occurrences of crosscutting during software requirements engineering and design, and to determine how these requirements are composed. In this research, we propose an approach to detect and model crosscutting concerns by restructuring object-oriented artifacts at the breakpoints of:

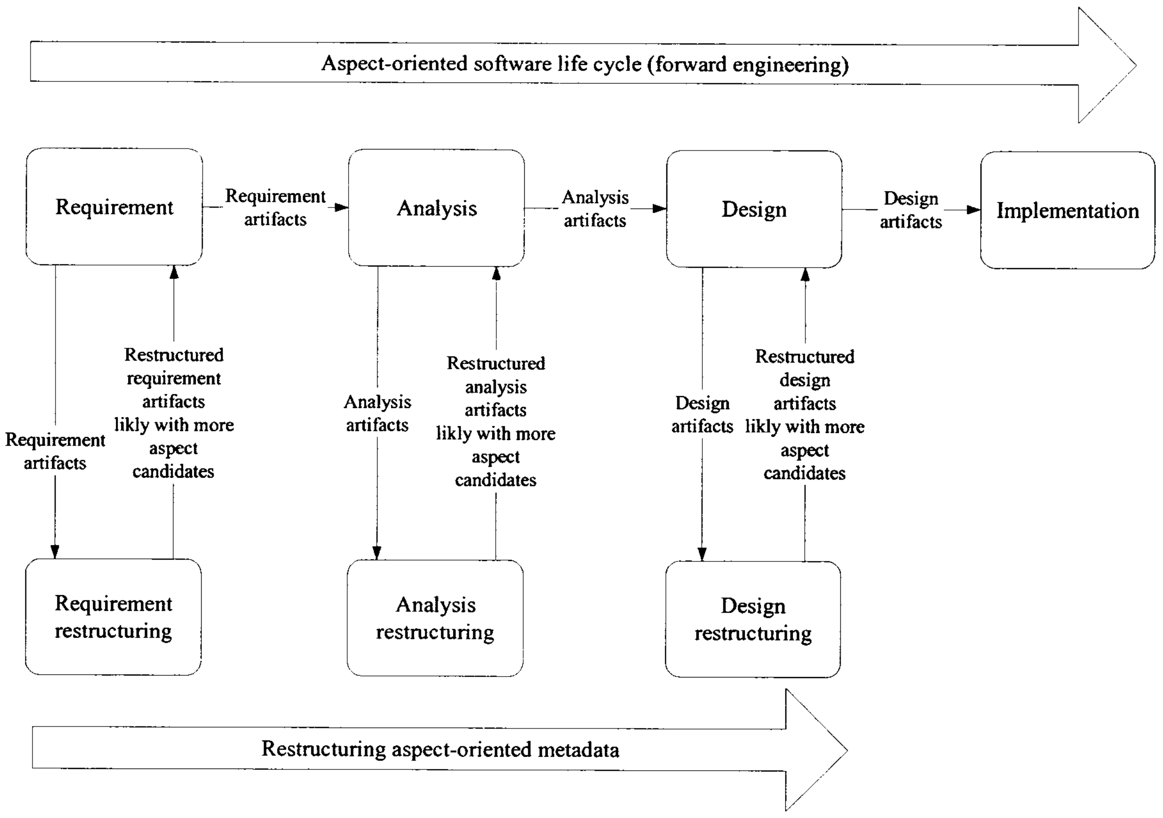
1) Requirements elicitation,

2) Analysis

3) Design and

4) Implementation activities.

During software development, we move from requirements to implementation (forward engineering) we incrementally refine the system. This is applicable to both the linear as well as the iterative development process. Earlier work by the authors and others [13] has demonstrated how we can identify crosscuttings at the requirements phase. Together with possible initial crosscuttings, we believe it is possible that crosscutting concerns may arise at different stages during development and at various levels of granularity. Our proposal (shown in Figure 2) builds on earlier work on forward engineering and reverse engineering [8, 9] and considers the set of artifacts produced at certain milestones during forward engineering. By analyzing the artifacts, we are able to restructure each set of artifacts in order to identify any possible "missing" crosscuttings. As a result, each stage of development is enhanced with a corresponding restructuring activity that will deliver a refined set of artifacts to the stage.



Aspect-oriented software life cycle (forward engineering)
Requirement
Requirement artifacts
Analysis
Analysis artifacts
Design
Design artifacts
Implementation
Requirement artifacts
Restructured requirement artifacts likly with more aspect candidates
Analysis artifacts
Restructured analysis artifacts likly with more aspect candidates
Design artifacts
Restructured design artifacts likly with more aspect candidates
Requirement restructuring
Analysis restructuring
Design restructuring
Restructuring aspect-oriented metadata

Figure 2. Restructuring requirements, analysis, and design artifacts in different stages of aspect-oriented software life cycle

The Thesis is organized as follows:

Section 2: Problem and motivation

Section 3: Related work

Section 4: Theoretical background

Section 5: Proposal and methodology

Section 6: Strategies for aspect mining

Section 7: Case studies

Section 8: Conclusion and recommendations

2. Problem and motivation

It is not always straightforward to identify aspects over a set of design artifacts. This would be particularly true in applications of complex domains. A great deal of the research in AOP has focused on identifying aspects over an implementation in legacy systems (a set of code artifacts) and a number of automated tools [1, 2, 3, 4] are now available to developers. However, less attention has been placed on identifying aspects at the earlier stages of development. The motivation behind this project is:

1. To define a set of rules in order to identify aspects in a crosscutting object-oriented design.

2. To define a set of principles by which we can migrate a crosscutting object-oriented design to an aspect-oriented context.

3. To show that any crosscutting object-oriented design with any degree of modularity can be restructured to a corresponding aspect-oriented design by analyzing object-oriented design artifacts.

4. To define an aspect mining methodology that can be applicable in entire software life cycle.

5. Apply this aspect mining approach to object-oriented compilers to identify aspect candidates during parsing implementation artifacts. The result is a new method of parsing that helps developers to develop code that follows the semantics defined in the design level in addition to identifying potential crosscutting concerns and also it is applicable for aspect mining in the legacy systems. This approach enables us to identify crosscuttings behavior with considering all possible scenarios implemented by the source code.

3. Related work

Most of the current aspect mining approaches operate on source code. In [1] the authors describe an automatic dynamic aspect mining approach. This approach uses program traces that are generated in different program executions. These traces are then investigated for recurring execution patterns based on different constraints, such as the requirement that the patterns have to exist in a different calling context in the program trace. In [2] the authors describe an automatic static aspect mining approach, where the control flow graphs of a program are investigated for recurring executions based on different constraints, such as the requirement that the patterns have to exist in a different calling context. In [3] the authors introduce a concern graph representation that abstracts the implementation details of a concern and it makes explicit the relationships between different elements of the concern for the purpose of documenting and analyzing concerns. In [4] the authors describe concerns based on class members. This description involves three levels of concern elements: use of classes, use of class members, and class member behavior elements (use of fields and classes within method bodies). Use of classes is expressed by the class-use production rules. The rules specify that a concern either uses the entire class to implement its behavior or only part of a class, as well as what parts of the class participate in the implementation of the particular concern. In [12] the authors use Abstract syntax tree to detect duplicated code ("clones"). In [17] the authors introduce a general-purpose, multidimensional, concern-space modeling schema that can be used to model early-stage concerns.

4. Theoretical background

In many situations developers work on software systems that other people have designed and developed. In the literature, the term "reverse engineering" is defined as *"[t]he process of analyzing a subject system to identify its components and their interrelationships and create representations of the system in another form, or at a higher level of abstraction"* [7]. Reverse engineering involves only analysis, not change. The goal of reverse engineering is to obtain understanding (comprehension) of the system. A related term, "restructuring" and its object-oriented equivalent "refactoring", refer to reformulating a program without first abstracting it to a higher level. Imperative to the restructuring process is to maintain the same level of functionality and semantics of the system.

These transformations are illustrated in Figure 3. To restructure (refactor) a set of object-oriented implementation artifacts (code) into an aspect-oriented implementation would also have to involve a propagation of backward activities in order to maintain the synchronicity between design artifacts and implementation. Our proposal entails the refactoring of a system at the design level and the deployment of forward engineering in order to transform an object-oriented design to aspect-oriented.

4.1. Describing Aspects

In this section we describe the main concepts of this research such as "Separation of concerns", "Crosscuttings", "Aspect", "Aspect mining".

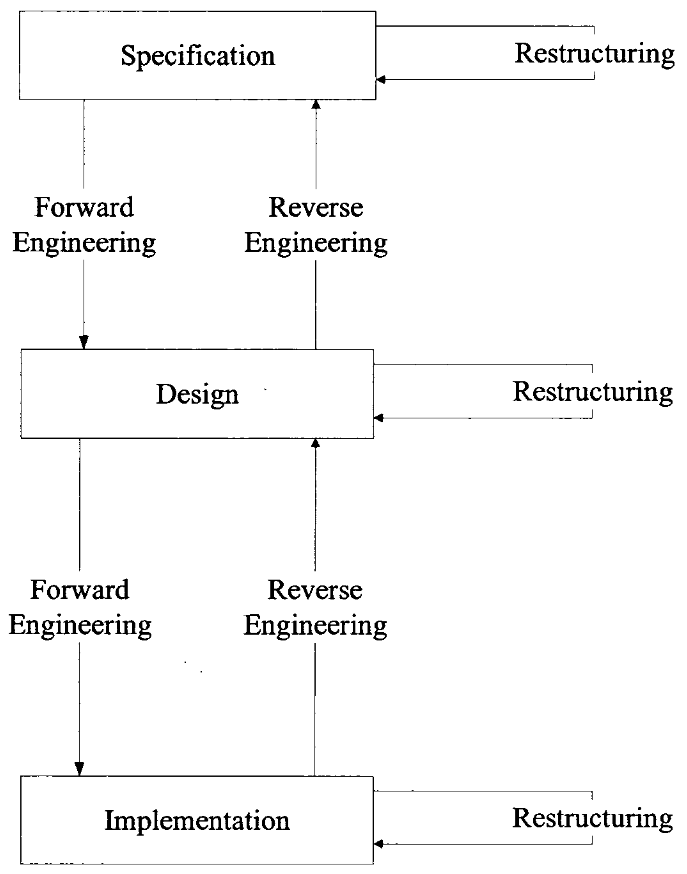


Figure 3. Reengineering transformations

4.1.1.Separation of concerns

“Separation of concerns” is defined as realization of problem domain concepts into separate units of software (Parnas, Dijkstra). There are many benefits associated with having a concern of a software system being expressed in a single modular unit such as better analysis and understanding of the system, easy adaptability, maintainability and high degree of reusability. Although separation of concerns is crucial to software development, but how to best achieve it is an open issue. Object-oriented programming provides linguistic mechanisms to support functional decomposition along the notion of class and allows us to view computation as a set of collaborating objects. Overall OOP gets closer to the way we think and it has been a great success towards separation of concerns.

4.1.2. Crosscutting

In OOP, decomposition (through mechanisms provided by current languages) is one-dimensional focusing on the notion of a class. In large systems, interaction of components is very complex. Current programming languages do not provide constructs to address certain properties in a modular way. OOP cannot address the design or implementation of behavior that spans over many modules (often unrelated). Certain properties cannot be localized in single modular units, but their implementation cuts across the decomposition hierarchy of the system. These properties are called crosscutting concerns, or aspects. Two issues are of interest here. For these operations code spans over many methods of potentially many classes and packages and implementation of some operations do much more than performing some code functionality, it means that these operations contain code for more than one concerns.

Crosscutting imposes two symptoms on software development:

1. Code scattering: implementation of some concerns not well modularized but cuts across the decomposition hierarchy of the system.
2. Code tangling: a module may contain implementation elements (code) for various concerns.

The origin of crosscutting is: there might not always be a one-to-one mapping from problem domain to solution space and requirements space is n-dimensional, but implementation space is one-dimensional (in OOP everything must belong to a class). Figure 4 illustrates the reasons for crosscutting phenomena.

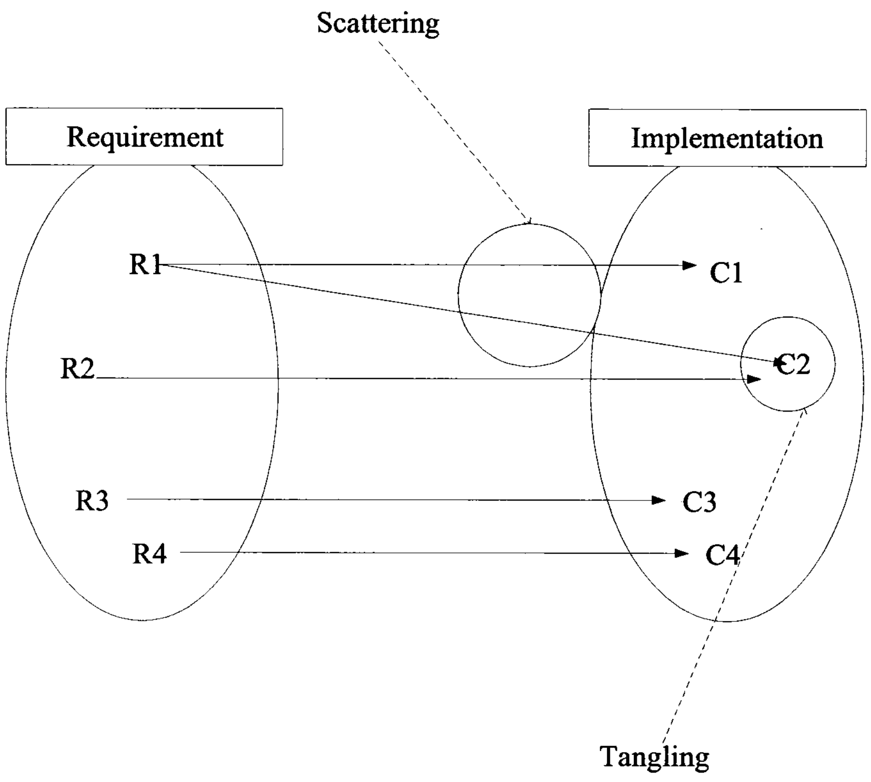


Figure 4. Reasons for crosscutting phenomena

The fact that there is a single axis of decomposition has been termed *"decomposition tyranny"*. Decomposition tyranny is the source of crosscutting. The decomposition tyranny (and crosscutting) applies to artifacts across the life-cycle of software (not confined to implementation).

As a result of crosscutting, the benefits of OOP cannot be fully utilized, and developers are faced with a number of implications:

1. Poor traceability of requirements: Mapping from an n-dimensional space to a single dimensional implementation space.

2. Lower productivity: Simultaneous implementation of multiple concerns in one module breaks the focus of developers.

3. Strong coupling between modular units in classes that are difficult to understand and change.

5. Low degree of code reusability. Core functionality impossible to be reused without related semantics, already embedded in component.
6. Low level of system adaptability.
7. Changes in the semantics of one crosscutting concern are difficult to trace among various modules that it spans over.
8. Programs are more error prone.
9. Difficult evolution.
10. Crosscutting affects the quality of software.

4.2. What is Aspect Mining?

Identifying crosscutting concerns is an important part of a process referred to as *aspect mining*. One of the goals of aspect mining is to identify opportunities for transforming (parts of) the code of an application into aspect-oriented code. Since aspects are specifically designed to deal with crosscutting concerns, aspect mining is naturally focused on crosscutting concerns. Aspect mining is typically described as a specialized reverse engineering process, which is to say that legacy systems (source code) are investigated (mined) in order to discover which parts of the system can be represented using aspects. A major problem in re-engineering legacy code based on aspect-oriented principles is to find and to isolate these crosscutting concerns. The detected concerns can be re-implemented as separate aspects, thereby improving maintainability and extensibility as well as reducing complexity.

As a general explanation we can say that aspect mining is:

- New research area
- Identification of crosscutting concerns in legacy systems
- "Isolation" of crosscutting concerns
- Helpful for program understanding
- Useful for refactoring
- Applicable in entire software life cycle

In our aspect mining approach we analyze object-oriented artifacts in different ways because we believe that there is no single rule or algorithm to identify all types of aspects in object-oriented artifacts. We categorize aspects in different groups and for identifying each of them we proposed different methods. In this section we introduce our aspect mining methodology at the breakpoints of: requirements elicitation, analysis, design, and implementation activities.

5. Proposal and methodology

Analyzing and restructuring of UML artifacts or in general object-oriented requirement, analysis, design and implementation artifacts, in order to detecting potential aspects, are based on the concepts of dependency graphs and production rules, and two-level grammars. In this section we explain the methodology that we apply to identify aspect candidates throughout software life cycle. The following steps are applied to identify and detect aspect candidates in different activities of software life cycle:

(1) Defining **dependency graphs** in order to identify entities with an independent role in a scenario.

(2) Defining semantics and metadata of UML artifacts by a set of **production rules** in order to identify patterns of crosscuttings.

(3) Identifying code duplication and scattering by using **object dependency graphs**, and

(4) Parsing source codes by two-level grammars.

Figure 5 illustrates our aspect mining methodology in different activities of software lifecycle.

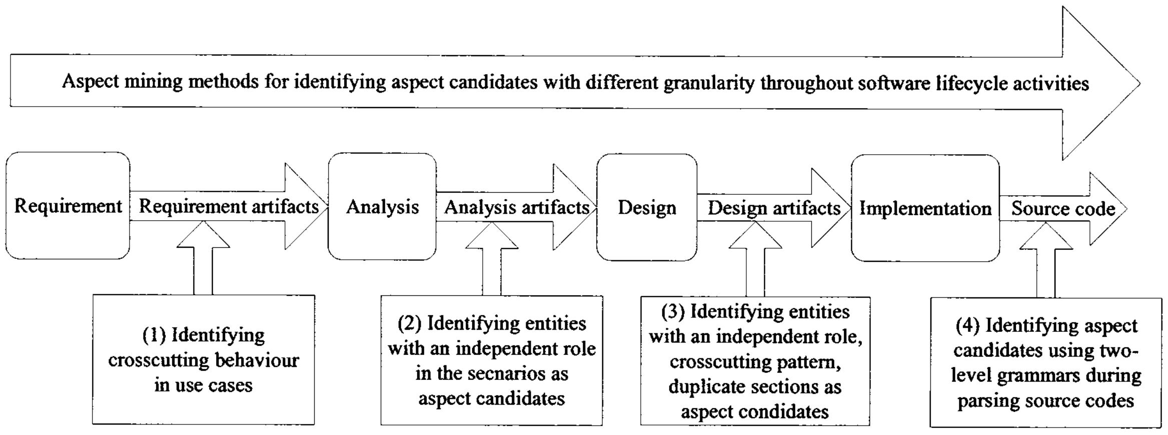

Aspect mining methods for identifying aspect candidates with different granularity throughout software lifecycle activities
Requirement
Requirement artifacts
Analysis
Analysis artifacts
Design
Design artifacts
Implementation
Source code
(1) Identifying crosscutting behaviour in use cases
(2) Identifying entities with an independent role in the secnarios as aspect candidates
(3) Identifying entities with an independent role, crosscutting pattern, duplicate sections as aspect condidates
(4) Identifying aspect candidates using two-level grammars during parsing source codes

Figure 5. Aspect mining methodology

In each step of this methodology we identify aspect candidates in a crosscutting object oriented artifacts such that a candidate aspect identified in requirement stage is traceable in analysis, design, and implementation but with different granularity. We use dependency graphs, production rules, and two-level grammars as our tools for aspect mining. The following table illustrates the usage of these tools in different stages of software life cycle.

ASPECT MINING TOOLS	SOFTWARE LIFE CYCLE ACTIVITY
Dependency graph	1,2,3
Production rules	1,2,3,4
Object dependency graph	3,4
Two-level grammar	4

6. Strategies for aspect mining

With UML we are able to model and visualize a real world system based on object definitions and object relationships. The semantics and metadata behind the model can be represented as a set of abstract rules, which we refer to as production rules. To identify aspects in object oriented artifacts we need to analyze them from different dimensions. The reason for our multi-dimensional analysis is that production rules do not have vertical order of message passing sequence but they carry information about horizontal order of message passing sequence. On the other hand, in the parse trees we do not have horizontal order of message passing sequence but we have the vertical order of message passing sequence. By extending the parse tree with production rules using two-level grammars [11], we will have a vertical and horizontal order of message passing between objects of a scenario and we are able to identify vertical and horizontal aspects.

We introduce the following two dimensions for our analyzing process:

a) Vertical analyzing

b) Horizontal analyzing

We also categorize aspects in three groups: vertical aspects, horizontal aspects and hybrid aspects. Horizontal aspects are the aspects that can be identified by analyzing the sequence of message passing between involved objects in the scenario or part of the scenario in the different level of objects interaction relationships. Our strategy to detect this kind of aspects is identifying objects with an independent role in scenarios and also by detecting some crosscutting patterns in our design artifacts. In this case, part of a class (some properties) or even a class itself can be modeled as an aspect.

Vertical aspects are the aspects that can be identified by analyzing sequence of message passing between involved objects in the scenario or part of the scenario in a single level of objects interaction relationship. Our strategy to detect vertical aspects is analyzing object-oriented artifacts by using two-level grammars and detecting clones. In this case, typically part of some classes are involved to define this kind of concerns, as we said we can identify this kind of aspects by detecting scatted and duplicate codes (Figure 6).

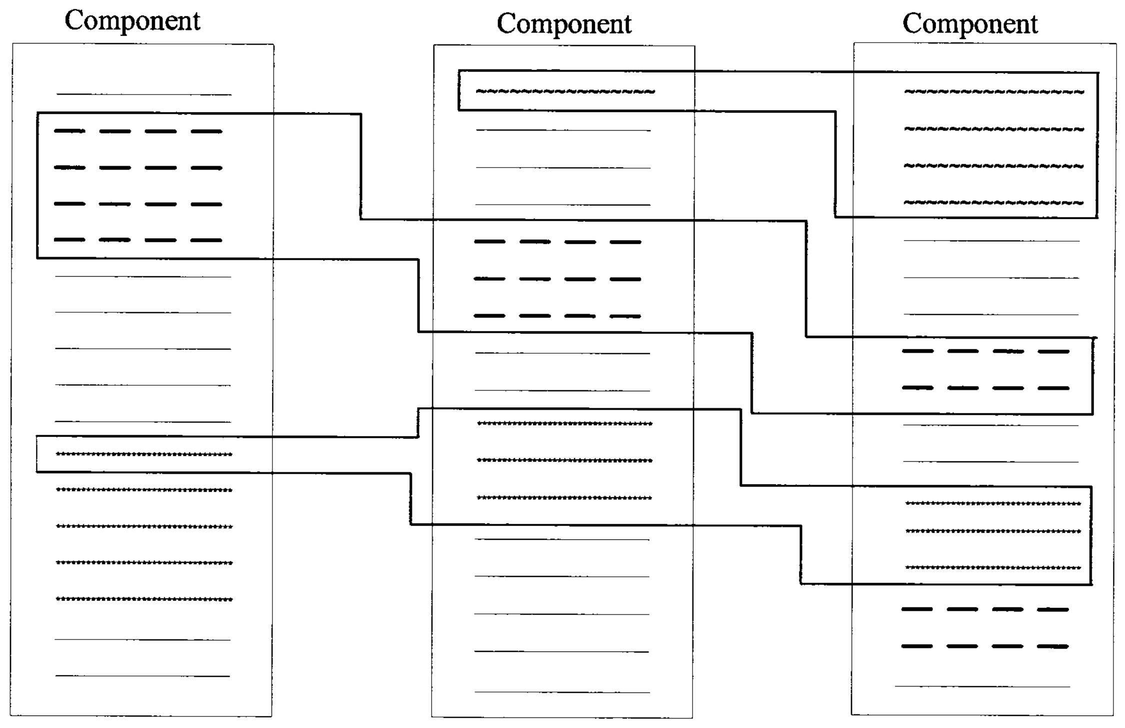


Figure 6. Initial picture of crosscutting

Horizontal aspects are different from vertical aspects. The reason of having vertical aspects in object-oriented systems is that OOP cannot address the design or implementation of behavior that spans over many modules (often unrelated) and certain properties cannot be localized in single modular units, but their implementation cuts across the decomposition hierarchy of the system.

In object-oriented design, a system is modeled as a collection of cooperating objects, and individual objects are treated as instances of classes within a class hierarchy. The result of our proposed restructuring process is an aspect-oriented design such that aspects may be introduced into the object-oriented design or certain classes may be restructured as aspects. From this point of view, we categorize aspects in two different groups: behavioral and business aspects.

1. **Behavioral Aspects:** They provide behavior/ability to objects. For example, a behavior aspect is one that implements a synchronization policy or applies a contract checking mechanism, or persistence (in database applications) to one or more objects. In object oriented modeling we can use composition patterns to model this category of aspects.
2. **Business Aspects:** They implement some business in the problem domain. As an example, consider a sales software system where both the ordering and purchasing subsystems make use of an accounting subsystem. In order to perform their tasks correctly, both ordering and purchasing subsystems require the use of the accounting subsystem. In this scenario, the level of coupling between the accounting subsystem (the business aspect) and the rest of the system is high.

6.1. Vertical aspect

Vertical aspects are the aspects that can be identified by analyzing sequence of message passing between involved objects in the scenario or part of the scenario in a single level of objects interaction relationship.Listing 1 illustrates an example of vertical aspect. `Method1()` and `Method2()` both of them call `C1.M1()`, `C2.M2()`, `Method3()`, and also after doing some other business they call `C1.M3()`. We can define these sections of code as a vertical aspect.

Listing 1: Vertical aspect

```
Method1() {
    C1.M1();
    C2.M2();
    Method3();

   // do something …

   C1.M3();
}
```

```
Method2 () {
    C1.M1();
    C2.M2();
    Method3();

   // do something …

    C1.M3();
}
```

6.2. Horizontal aspect

Horizontal aspects are the aspects that can be identified by analyzing the sequence of message passing between involved objects in the scenario or part of the scenario in the different level of objects interaction relationships. Listing 2 illustrates an example for horizontal aspects. Consider the derivation sentences for `Method1()` and `Method2()` defined in listing 2:

```
Method1()::=[call]<C1.M1>::=[call]::=<C2.M2()>::=[call]<C2.M3()>

::=<rest of Method1>

Method3()::=[call]<C1.M1>::=[call]::=<C2.M2()>::=[call]<C2.M3()>

::=<rest of Method2>
```

According to these derivation sentences we can introduce a horizontal aspect defined by the following section of derivation sentences:

```
[call]<C1.M1>::=[call]::=<C2.M2()>::=[call]<C2.M3()>
```

Listing 2. Horizontal aspect

Method1() {
    C1.M1();
    // do something
}
	Method2() {
	    C1.M1();
	    // do something
	}

C1.M1() {
    // do something
    C2.M2();
}
	C2.M2() {
	    // do something
	    M3();
	}

6.3. Hybrid aspect

Listing 3 illustrates a concern that we named it as hybrid concern that is the most popular type of aspects in an object-oriented code. In `Method1()` and `Method2()` we identify a vertical aspect that consists of `C1.M1()`, `Method3()`, and `C1.M3()`, on the other hand `C1.M1()` itself originate a horizontal aspect then we can define a hybrid aspect with the following definition:

```
<Hybrid aspect>::=

<begin of aspect>::= <C1.M1()>→[call]<C2.M2()>→[call]<C2.M3()>

                ::=[call]Method3()

<end of aspect>::=[call]<C1.M3()>
```

Listing 3. Hybrid aspect

```
Method1()
{
    C1.M1();
    Method3();

   // do something …

    C1.M3();
}
```

```
Method2()
{
    C1.M1();
    Method3();

   // do something …

    C1.M3();
}
```

```
C1.M1()
{
     // do something
     C2.M2();
}
```

```
C2.M2()
{
     // do something
     M3();
}
```

6.4. Ambiguity in identifying aspects

In the section 6.3 the example of hybrid aspects is an example of ambiguity to identify aspects. In this example we define `{C1.M1(), Method3(), C1.M3()}` as an aspect, such that `C1.M1()` and `Method3()` execute before the main body of `Method1()` and `C1.M3()` executes at the end of this method. Our strategies to identify aspects are based on detecting clones or duplicate codes, even if the duplicated parts are completely unrelated. Although this strategy is useful for identifying aspects, but defining a well-defined aspect is not always guaranteed.

For example If we just consider `Method1()` and `Method2()` then the hybrid aspect defined in section 6.3 can be a valid aspect definition, regardless of `C1.M1()`, `Method3()`, and `C1.M3()` are unrelated methods and modularizing them into an aspect is not a good modularization. We may encounter to some other scenarios that detecting correct aspects is not straight forward and is ambiguous or even can change our aspect definition, for example we may have the following choices of aspects depend on the code artifacts that we consider for analyzing.

1: Define all three methods as a single aspect

`Aspect: {C1.M1(), Method3(), C1.M3()}`

2: Define two aspects with the following definition

`Aspect 1: {C1.M1(), Method3()}`

`Aspect 2: {C1.M3()}`

3: Define two aspects with the following definition

`Aspect 1: {C1.M1(),C1.M3()}`

`Aspect 2: {Method3()}`

4: Define three aspects with the following definition

`Aspect 1: {C1.M1()}`

`Aspect 2: {C1.M3()}`

`Aspect 3: {Method3()}`

5: ....

Identifying a well-defined aspect in ambiguous situations needs to analyze the artifacts with different strategies such as identifying objects with an independent role in a scenario and detecting duplicated codes and clones may lead us to an ambiguous situation.

Consider the following example. In this example, authentication, contract checking and logging are tangled with the core operation (methods) of the component (class). Two issues are of interest here:

1. Implementation for authentication, contract checking and logging is not localized.

Code spans over many methods of potentially many classes and packages.

2. Implementation of `someOperation()` does much more than performing some core functionality. It contains code for more than one concern.

```
Public class BusinessLogic {
    … data members for business logic
    … data members for authentication,
    … contract checking and logging
    public void someOperation {
         // perform authentication
         // ensure preconditions
         // log the start of an operation
         … perform core operation
         // authentication
         // ensure postconditions
         // log successful termination of operation
    }
  // more operations similar to the above
}
```

With using "identifying duplicate code strategy" we identify an aspect with the following definition:

```
<Duplicated code aspect>::=
<begin of aspect> ::=<perform authentication>
                  ::=<ensure preconditions>
                  ::=<log the start of an operation>
<end of aspect>   ::=<authentication>
                  ::=<ensure postconditions>
                  ::=<log successful termination of operation>
```

This definition is not a well-defined aspect since it still contains responsibility for more than one aspect. By analyzing the dependency graph for `BusinessLogic`, `Authenticator`, `Contract checker`, and `Logger` classes it is clear that

Authenticator, Contract checker, and Logger classes have independent role in the scenario and can be defined as a separate aspects. The following aspect definitions are well-defined:

```
<authenticator aspect>::=
<begin of aspect> ::=<perform authentication>
<end of aspect>   ::=<authentication>


<contract checker aspect>::=
<begin of aspect> ::=<ensure preconditions>
<end of aspect>   ::=<ensure postconditions>


<logger aspect>::=
<begin of aspect> ::=< log the start of an operation>
<end of aspect>   ::=<log successful termination of operation ensure>
```

6.5. Dependency graphs

As their name suggests, dependency graphs illustrate dependencies between entities and they can be deployed at different stages throughout development. In a dependency graph, each vertex represents an object and each edge represents a dependency between a pair of adjacent vertices signified by the direction of the edge. The arrow illustrates the sending of a message from source to destination. As a message is sent in order to fulfill a responsibility, we say that the source is dependent on the destination. We use dependency graph to identify objects that play an independent role in a scenario but they complete part of the scenario. This kind of objects in dependency graph can be identified by the following rules:

(1) The in-degree of node representing the object is greater than one.

(2) There is no direct or indirect feedback from target node to the destination nodes.

Figure 7 illustrates classes C1, C2, and C3. In Figure 7.a, C1 depends on C2 implying that C2 is required to fulfill one or more of the responsibilities assigned to C1. We also define {D(Ci)} as the set of all classes that Ci is dependent on. In this figure, {D(C1) = {C2, C3}. In Figure 7.b, however, there is no dependency of C1 on either C2 or C3. In a directed cyclic graph, vertices that form part of a cycle do not pose crosscutting behavior (see Figure 7.a). In Figure 7.b, even though C1 may delegate to another class (not shown), we can regard C1 as a candidate crosscutting concern because of the unidirectional direct dependency from C2 and C3 on C1.

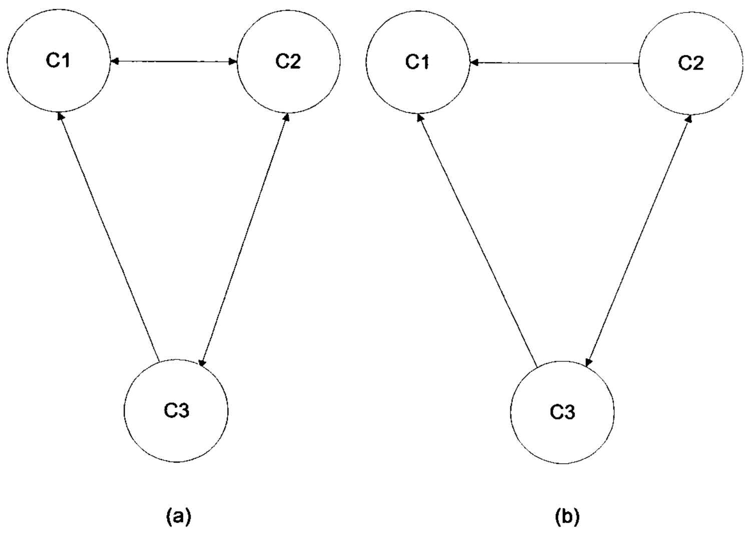


Figure 7. Dependency graph

Existing a cycle between instances involved in a scenario is not a reason that they are not crosscutting each other. For example in observer pattern in respect of existing a cycle we will have crosscutting behavior (state-preserving concerns) [10]. In other words, first we build dependency graph if we cannot find any independent instances then as a second step we look for some crosscutting patterns and as a third step we build object-dependency graph for finding duplicate or scattered sections.

6.6. Object dependency graph

We can extend the dependency graph to illustrate dependencies between instances. On the other words, Class level dependency graphs at design level can be extended to instance level dependency graph at implementation artifacts level to identify crosscutting phenomena.

Consider the code on Listing 4 where introduce tracing to two methods, `Order::placeOrder()` and `Invoice::assignInvoice()` and consider the corresponding dependency graph in Figure 8.a. We see that the behavior in object `Trace` crosscuts methods `order.placeOrder()` and `invoice.assignInvoice()`.Figure 8.b is a dependency graph defined for instances of `Order`, `Invoice`, and `Trace`. Each node shows an instance and each directed edge shows a method invocation. The numbers on an arc describe the level and order of message passing. For example, `message# 1.1` constitutes the first step in the scenario and in the first level of message passing. Methods `traceEntry()` and `traceExit()` in class `Trace` are crosscutting methods `Order::placeOrder()` and `Invoice::assignInvoice()`.

Listing 4.

```
Order::placeOrder()                Invoice::assignInvoice()
{                                  {
   Trace trace1;                       Trace trace2;
   Inovice invoice;                    trace1.traceEntry();
   trace1.traceEntry();                // do something
   invoice.assignInvoice();            trace2.traceExit();
   trace1.traceExit();             }
}
```

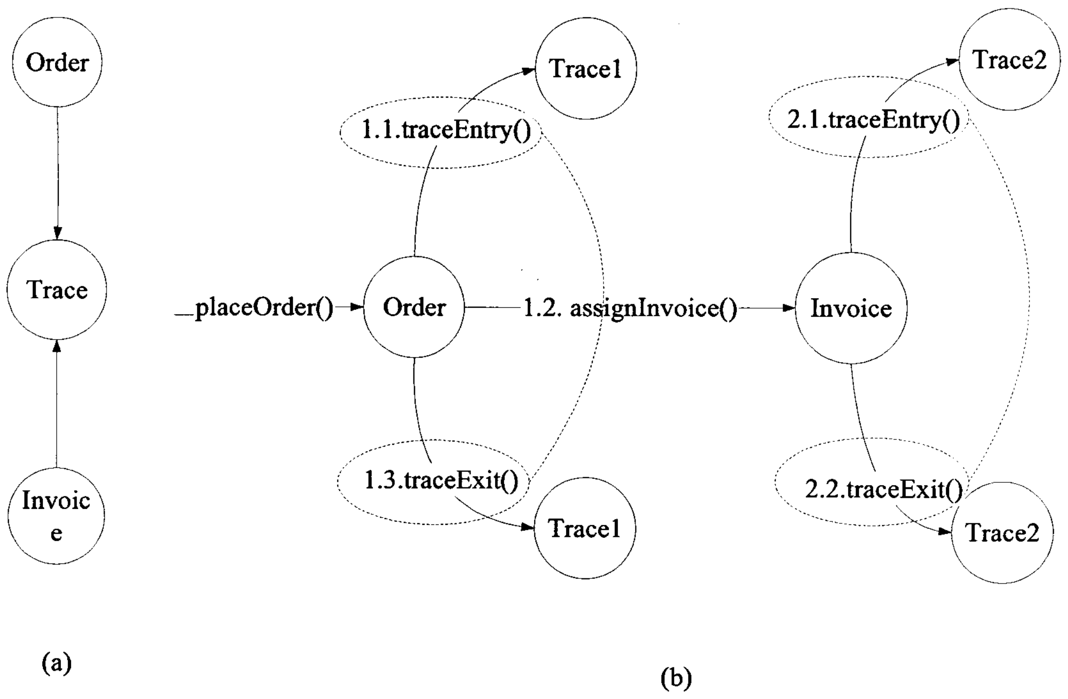


Figure 8. Object dependency graph

6.7. Dependency types

Suppose C1, C2, and C3 are three different entities (classes) associated with a scenario defined by the dependency graph of (Figure 7). We define a scenario as a set of five elements shown in Table 1.

Table 1. A scenario as a set of elements

S (Entities): $\{e_1, e_2, e_3, e_4, \ldots\}$	Set of all entities that cooperate and play a role in the scenario.
S (Associations): $\{a_1, a_2, a_3, a_4, \ldots\}$	Set of all associations between entities in S (Objects) set.
C (Postconditions): $\{p_1, p_2, p_3, p_4, \ldots\}$	The types of postconditions are: 1.Creation or deletion of an instance of an entity S (Entities). 2.Formation or breaking of an association between two entities S (Associations). 3. Modification of one or more attributes of an entity.

We can say that C1 has an independent role in the scenario if the following requirements are met:

- R1: C1 ? C2 (entities): C1 belongs to entity set defined by C2, or C1 is visible by C2 as an attribute, then we say C2 depends on C1 to fulfill some part of its responsibility [direct dependency from C2 to C1], on the other words, A *direct dependency* from C2 to C1 exists when C2 makes a reference to C1. and
- R2: C2 !? C1 (entities): C2 doesn't belong to entities set defined by C1, or C2 is not visible by C1 directly [no direct dependency from C1 to C2] and
- R3: if we define $\{D_1$ (C1)$\}$ as a set of the first level entities the C1 depends on them then C2 !? $\{D_1$ (C1)$\}$.

- R4: if we define $\{D_2\ (C1)\}$ as a set of the second level entities the C1 depends on them then C2 !? $\{D_2\ (C1)\}$.
- R5: In general if we define {Di (C1)} as a set of the i-th level entities the C1 depends on them then C2 !? {Di (C1)}.

According to the requirement R1 there is a direct dependency from C2 to C1 which forms the starting point for a dependency graph. In R2, object C1 does not refer to C2, implying that there is no direct dependency from C1 to C2. R3 implies that objects defined in C1 do not refer to C2. In R4, objects defined in dependent objects of C1 do not refer to C2. R5 implies that objects which are visible by objects defined in the dependent objects of C1 do not refer to C2 and so on. This implies that there is not a direct or indirect feedback from C1 to C2.

Figure 9 illustrates different possibilities of cycles between entities by using a dependency graph. We distinguish between different types of cyclic dependencies: The dependency is *bidirectional* if the converse is also true. *Direct cyclic dependency* from `c1` to `c2` exists when `c1` makes a reference to `c2` and `c2` makes a direct reference to `c1`. An *indirect dependency with direct feedback (or direct dependency with indirect feedback)* from `c1` to `c2` exist when `c1` uses a reference, `c3`, which in turn makes a reference to `c2` and `c2` makes a direct reference to `c1`, and also with the same concept we can define *multi-level indirect dependency with direct feedback (or direct dependency with multi-level indirect feedback)*, and *multi-level indirect dependency with multi-level indirect feedback.*

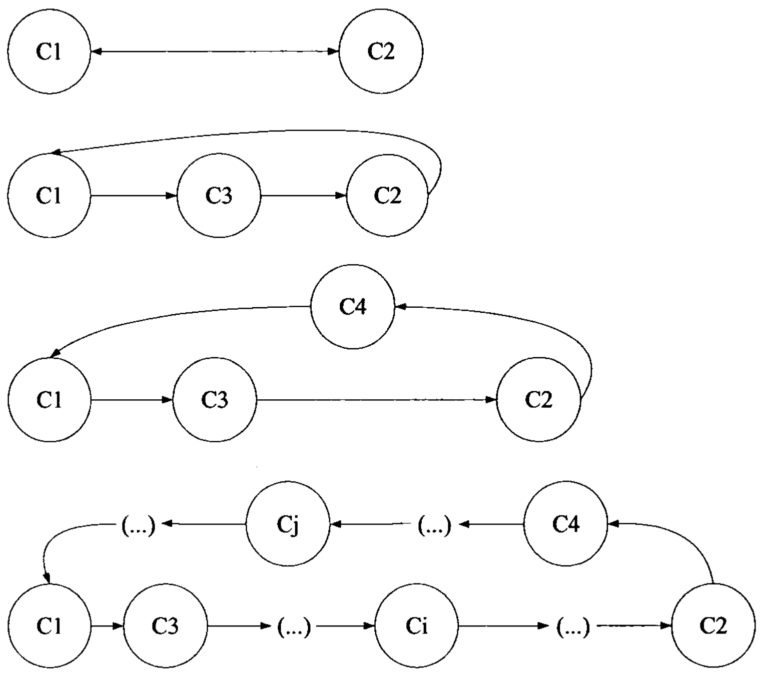


Figure 9. Different types of cyclic dependencies

Example 1: Dependency by message sending

This kind of dependency can be defined as direct (between two classes) or indirect (between more than two classes) where the classes are defined as [C→B→A→C][Cycle]. An example with class definitions is given below:

```
class C {
   B b; // declares an instance of B
   CF (){
      b.BF (); // sends a message to B
   }
}
class B {
   A a; // declares an instance of A
   BF () {
      a.AF (); // sends a message to A
   }
}
class A {
   C c; // declares an instance of C
   AF () {
      c.CF (); // sends a message to C
   }
}
```

Example 2: Dependency by message sending

In this kind of message passing dependency, class `A` is visible inside the function `BF()` of class `B` and not as an attribute. The classes are defined as [`C`→`B(BF)`→`A`→`C`][Cycle]. An example of class definitions is given below:

```
class C {
   B b; // declares an instance of B
   CF () {
     b.BF (); // sends a message to B
   }
}
class B {
   … // A is not visible in B as an attribute
   BF () {
      A a; // declares an instance of A
      a.AF (); // sends a message to A
   }
}
class A {
   C c; // declares an instance of C
   AF () {
      c.CF (); // sends a message to C
   }
}
```

Example 3: Dependency by message sending

This message passing dependency is similar to dependency defined in example 1 with this difference that class `C` sends a message to class `B` through its parent class. The classes are defined as [`C`→`B`→`A`→`C`][Cycle]. An example of class definitions is given below:

```
class P {
   B b; // declares an instance of B
   PF () {
      b.BF (); // send a message to B
   }
}
class C extends P {}
class B {
   A a; // declares an instance of B
   BF () {
       a.AF (); // sends a message to A
   }
}
```

```
class A {
   C c; // declares an instance of C
   AF () {
      c.PF (); // sends a message to C
   }
}
```

Example 4: Dependency by parameter passing

In this type of dependency a class will be visible by another class through parameter passing. For example, class `C` is visible to function `BF()` in class `B` as a parameter. The classes are defined as [`B→C and B→A→C`][No Cycle]. An example of class definitions is given below:

```
class C {
   Public int attr1;
   CF (){}
}
class B {
   A a; // declares an instance of A
   BF (C c) {
      a.AF (); // sends a message to A
      c.attr1 = 1;
   }
}
class A {
    C c; // declares an instance of C
    AF () {
       c.CF (); // sends a message to C
    }
}
```

6.8. Production rules and derivation sentence

With UML we are able to model and visualize a real world system based on object definitions and object relationships. The semantics and metadata behind the model can be represented as a set of abstract rules, which we refer to as production rules. By definition, production rules must be finite, implying that in order to represent semantics of UML artifacts we need a limited number of production rules. As a consequence, our knowledge, as incorporated in the production rules, must also be finite. It is the process of using this knowledge that will be "productive". In effect, the production rules can define an infinite set of scenarios. We use a set of production rules to represent use cases and objects relationships. By analyzing these production rules we can identify crosscutting concerns in this case by building derivation sentences we can identify classes with independent roles in a scenario. In other words, production rules complement dependency graphs for identifying cycles in the graphs at different phases of the development process. In addition, by using production rules we define some patterns that define crosscuttings behavior in general and identifying similar patterns in the UML artifacts lead us to detecting potential crosscuttings. Table 2 defines different transformations for production rules.

Production rules can define an infinite set of scenarios, derivation sentences are the way that we use these production rules for generating a scenario.

Definition. Semantics of object-oriented artifacts, G, can be defined in terms of a set of five elements, each of which is finite. Let `G = (C, A, M, P, R)`, such that

6.9. C is a set of classes

6.10. A is a set of attributes

6.11. M is a set of methods

6.12. P is a set of transformation rules. These rules not only define the structure of a class but also they can be used for defining all the system scenarios in terms of message passing between involved objects in the scenarios. The left hand side of these rules can be a class or a method of a class and the right hand side of a transformation is a member of set R -that defines the semantics of the transformation- and a class, a method of a class, or an attribute of a class.

6.13. R is set of relationships and concepts defined by object-oriented methodology we define this set as {[declare], [has], [call], [extend], [declare/receive], [set], [supplement]}.

Table 2. Transformation definitions in production rules

Transformation	**Definition**
`<C>`	C class.
`<C>::=[declare]<B>`	C declares an instance of B as an attribute.
`<C>::=[has]<CF()>`	C has a `CF()` method.
`<C.CF()>::=[call] <B.BF()>`	Method `C.CF()` calls method `B.BF()`.
`<C>::=[extend]<P>`	C class extends/inherits class P.
`<C.CF()>::=[declare/receive] <B>`	`C.CF()` method receives `B` class as a parameter.
`<C>::=[declare]<a>`	C declare a as an attribute.
`<B.BF()>::=[set]<C.a>`	`B.BF()` sets the value of `C.a`
`<C>::=<supplement> <B>`	C class supplements some behavior or functionality for `B` class. For example we can define the semantics of a Composition pattern by supplement transformation.

Production rules for dependency graph of example 1

For this example G, the object-oriented semantic, can be defined as:

```
C = {A, B, C}

A = {C.b:B, B.a:A, A.c:C}

M = {A.AF(), B.BF(), C.CF()}

P = {<C>::=[declare]<B>,

   <C>::=[has]<CF()>,

   <C.CF()>::=[call]<B.BF>,

   <B>::=[declare]<A>,

   <B>::=[has]<BF()>,

   <B.BF()::=[call]<A.AF()>,

   <A>::=[declare]<C>,

   <A>::=[has]<AF()>,

   <A.AF()>::=[call]<C.CF()>}

R = {[declare], [has], [call], [extend], [declare/receive], [set],

[supplement]}
```

The derivation sentence when `C.CF()` is called, is defined as follows:

```
{call C.CF()} → <C.CF()>::=[call] <B.BF()>::=[call] <A.AF()>

::=[call] <C.CF()> → cycle
```

Production rules for dependency graph of example 2

```
C = {A, B, C}

A = {C.b:B, A.c:C}

M = {A.AF(), B.BF(), C.CF()}

P = {<C>::=[declare]<B>,

  <C>::=[has]<CF()>,

  <C.CF()>::=[call]<B.BF()>,

  <B>::=[has]<BF()>,

  <B.BF()>::=[declare]<A>,
  <B.BF()>::=[call]<A.AF()>,
  <A>::=[declare]<C>,
  <A>::=[has]<AF()>,
 <A.AF()>::=[call]<C.CF()>}
R = {[declare], [has], [call], [extend], [declare/receive], [set],
[supplement]}
```

The derivation sentence when `C.CF()` is called, is defined as follows:

```
   {call C.CF()} → <C.CF()>::=[call] <B.BF()>::=[call] <A.AF()>
::=[call] <C.CF()> → cycle
```

Production rules for dependency graph of example 3

```
C = {A, B, C, P}
A = {P.b:B, B.a:A, A.c:C}
M = {A.AF(), B.BF(), P.PF()}
P = {<P>::=[declare]<B>,
  <P>::=[has]<PF()>,
  <P.PF()>::=[call]<B.BF()>,
  <C>::=[extend]<P>,
  <B>::=[declare]<A>,
  <B>::=[has]<BF()>,
  <B.BF()>::=[call]<A.AF()>,
  <A>::=[declare]<C>,
  <A>::=[has]<AF()>,
   <A.AF()>::=[call]<C.PF()>}
R = {[declare], [has], [call], [extend], [declare/receive], [set],
[supplement]}
```

The derivation sentence when `C.PF()` is called, is defined as follows:

```
  {call C.PF()} → <C.PF()>::=[call] <B.BF()>::=[call] <A.AF()>
::=[call] <C.CF()> → cycle
```

Production rules for dependency graph of example 4

```
C = {A, B, C}
A = {C.attr1:int, B.a:A, A.c:C}
M = {A.AF(), B.BF(), P.PF()}
P = {<C>::=[declare]<attr1>,
  <C>::=[has]<CF()>,
  <B>::=[declare]<A>,
  <B>::=[has]<BF()>,
  <B.BF()>::=[declare/receive]<C>,
  <B.BF()>::=[call]<A.AF()>,
  <B.BF()>::=[set]<C.attr1>,
  <A>::=[declare]<C>,
  <A>::=[has]<AF()>,
 <A.AF()>::=[call]<C.CF()>}
R = {[declare], [has], [call], [extend], [declare/receive], [set],
[supplement]}
```

The derivation sentence when `B.BF()` is called, is defined as follows:

```
{call B.BF()} →<C.PF()>::=[call]<A.AF()>::=[call]<C.CF()>
::=[set]<C.attr1>
```

6.9. Two-level grammar: a formal definition

We can apply our aspect mining approach to the object-oriented compilers such that we are able to identify candidate aspects during compiling implementation artifacts or source codes. The result is a new generation of compilers that help developers to develop code that follows the semantics defined in the design level in addition to identify potential crosscutting concerns. The main idea is to design an intelligent compiler that regenerate programming language grammar according to the semantics defined in the design artifacts, that is by modifying design artifacts the semantics beyond the design will be added to the programming language grammar as a set of rules and generate a new grammar that can be used for parsing and analyzing source codes. In effect, what has been referred to as "semantics" reintroduce as part of the syntax. What is particularly important is that we will use the idea of a two-level grammar and the related idea of dynamic syntax such that the programming language grammar could in part define its own grammatical (syntax and semantics) rules [11].

Each two-level grammar (TLG), W, can be defined as a 4-tuple

$$W = (Gm, G, G', \$)$$

Where Gm is the meta-grammar that extends G by adding semantics rules defined by G' and results a new grammar $\underline{\boldsymbol{G}}$ that contains syntax and semantics rules, G is the grammatical form of any object oriented language grammar for example Java language grammar, G' is a set of rules that define semantics of UML artifacts for a specific design and \$ is the uniform replacement rule for completing the instantiation of the grammatical form.

In turn, the meta-grammar is itself a 3-tuple consisting of

(1) Nm, a set of meta-variables which are assigned values to be transmitted to the grammatical form.

(2) T, a set of terminal symbols to be transmitted to the grammatical form; these are "terminal" only with the respect to the meta-level analysis;

(3) R, a set of meta-rules for rewriting of the meta-variables. These have the form $a \rightarrow b$ for a in Nm and b in (Nm U T)*.

6.10. Crosscuttings patterns

6.10.1. Rules for identifying composition patterns

A composition pattern is a design model that specifies (1) the design of a crosscutting requirement, independently from any design it may potentially crosscut, and (2) how that design may be reused wherever it may be required. Composition patterns are based on a combination of the subject-oriented model for decomposing and composing separate, potentially overlapping designs, and UML templates. In this example, we define a pattern that simplifies the composition pattern. In Figure 10, suppose that class `B` defines a set of objects that will be persistent through method `CF1()` of class `C`. Class `C` provides persistence for class `B` or even for another class `D` through different methods.

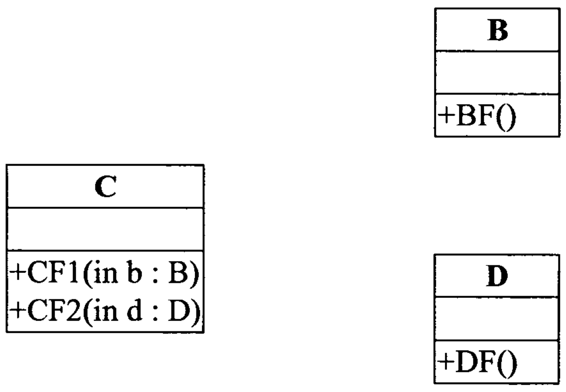


Figure10. Class C provides more functionality for Class B through CF1() method

The model in Figure 10 can be defined by the following production rules:

```
<P>::=[has] <PF()>

<C>::=[extend] <P>

<C>::=[has] <CF()>

<C.CF()>::= [declare/receive] <B>

<B>::=[has] <BF()>
```

The rules are defined as follows:

1. R1: C is not visible in B.
2. R2: B is only visible in C via parameter passing (class B is visible in class C through function CF()).

In this model class C defines a new behavior for class B, and in this case we can define the following rule:

```
<C>::=<supplement> <B>
<B>::=<inherit> C.CF()
```

The above implies that we can define class C as an aspect to declare a super type for class B. This type of pattern (composition pattern) can be implemented as a behavioral aspect.

6.10.2. Observer-pattern: State-preserving concerns

Consider the Observer design pattern that defines a one-to-many dependency between objects [6]. The key objects in this pattern are the subject and the observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with that of the subject's. Consider the UML class diagram of the observer pattern illustrated in Figure 11.

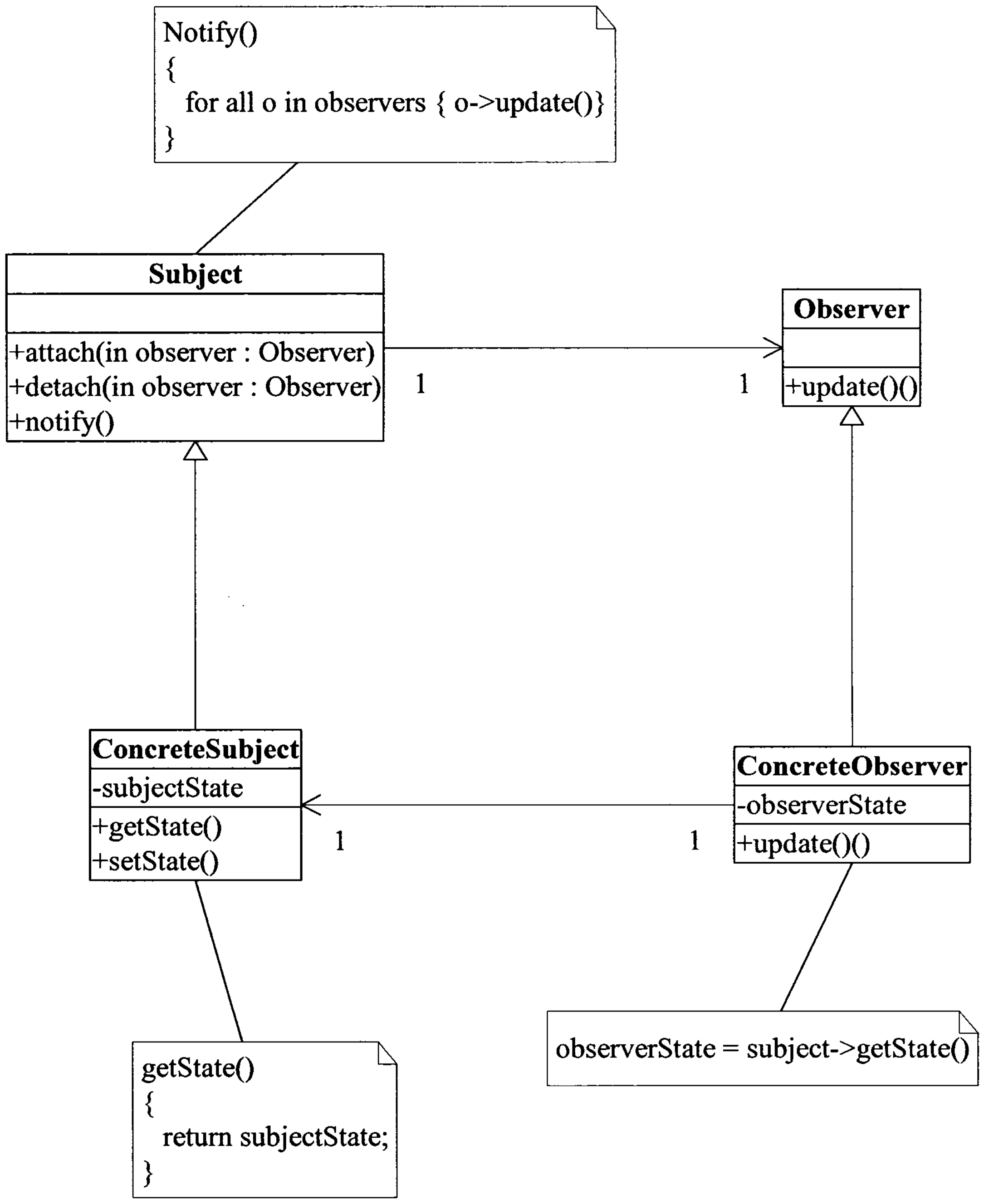


Figure11. UML class diagram for the observer pattern

The corresponding production rules for the observer pattern are defined as follows:

```
<Observer>::=[has]<Update()>

<ConcreteObserver>::=[extend]<Observer>

<ConcreteObserver>::=[declare] <observerState>
```

```
<ConcreteObserver>::=[has]<Update()>

<ConcreteObserver.Update()>::=[set] <ConcreteObserver.observerState>

<Subject>::=[has] <Attach()>

<Subject>::=[has] <Detach()>

<Subject>::=[has] <Notify()>

<Subject>::=[collection] <Observer>

<Subject.Attach()>::=[declare/receive] <Observer>

<Subject.Detach()>::=[declare/receive] <Observer>

<ConcreteSubject>::=[extend]<Subject>

<ConcreteSubject>::=[declare]<subjectState>

<ConcreteSubject>::=[has]<GetState()>

<ConcreteSubject>::=[has]<SetState()>

<ConcreteSubject.SetState()>::=[call]<Subject.Notify()>

<ConcreteSubject.SetState()>::= <set><ConcreteSubject.subjectState>

<Subject.Notify()>::=[call]<Observer.Update()>
```

Consider a scenario that corresponds to a call to `ConcreteSubject.SetState()`:

```
<ConcreteSubject.SetState()> ::=

      [set]<ConcreteSubject.subjectState] &&

      [call]<Subject.Notify()>

::=[call]<Observer.Update()>

::=[call]<ConcreteObserver.Update()>

::=[call]<ConcreteSubject.GetState()> &&

 [set]<ConcreteObserver.observerState>
```

Figure 12 illustrates the above scenario by a dependency graph. To create the dependency graph we use the following two rules: (1) each class in the scenario represents a node in the graph, where objects related by inheritance are merged into one node, and (2) each function call represents an edge from the caller object to the object that the called method belongs to.

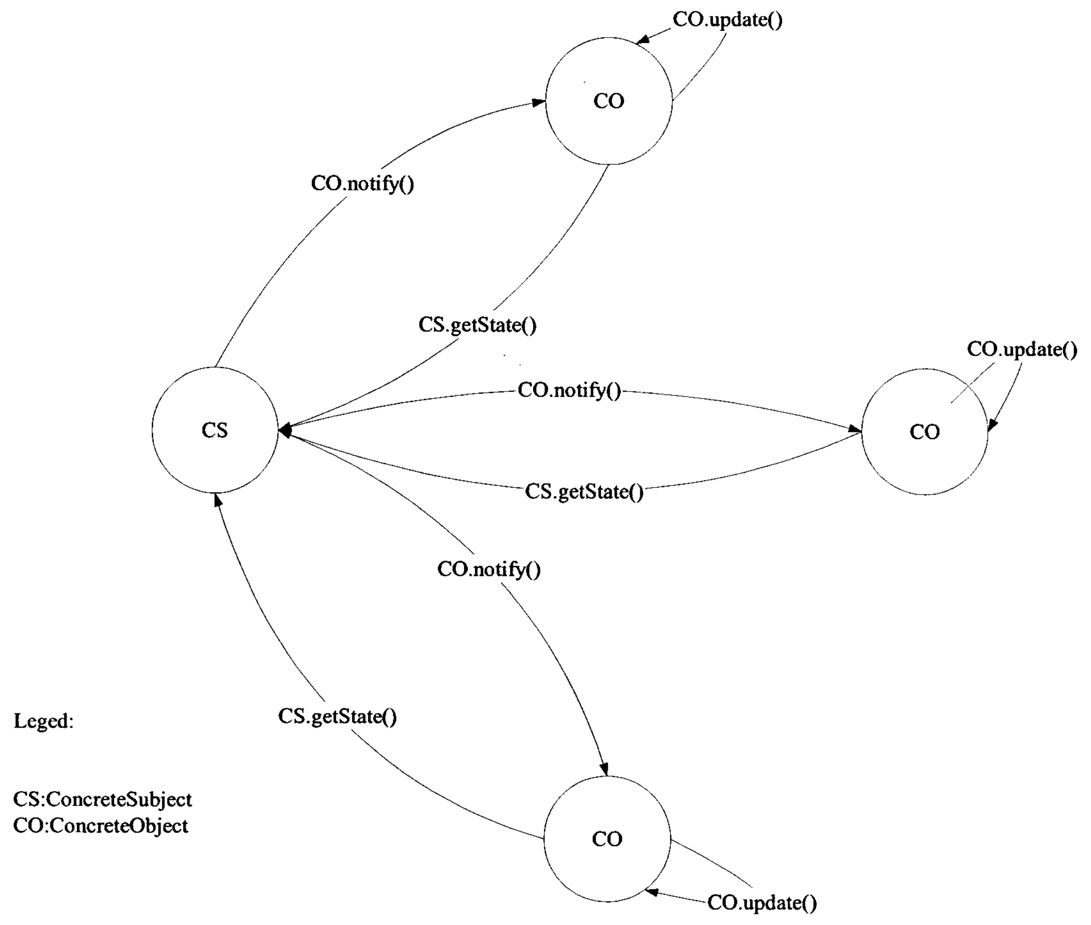


Figure12. Dependency graph for the Observer design pattern

The dependency graph in Figure 12 shows a cycle between `ConcreteSubject` and `ConcreteObserver`, where `cs` calls `co.Notify()` and `co` calls `cs.GetState()`. In this example, in spite of the existence of a cycle between concerns, we can still define an aspect. Suppose `C1` and `C2` are dependent class instances according to Figure 13. This scenario refers to state-preserving if `S1 = S1'` (`S1`: set of

`C1.states` before calling `C2.Fx()` and `S1'`: set of `C1.states` values after calling `C2.Fx()`). This implies that the state of `C1` before and after calling `C2.Fx()` remains the same. In the observer pattern when `ConcreteSubject` calls `ConcreteObserver.Notify()`, then after ending this call the state of `ConcreteSubject` does not change. For this reason the scenario is considered state-preserving. As a result, if no cycle exists in the concerns dependency graph, then according to the rules `R1, R2, R3, R4`, and `R5` defined in section 4.8 we will have crosscutting concerns and we can define an aspect, and if there is a cycle in concern dependency graph but concerns are state-preserving then the target concern, can be defined as an aspect. In the Observer design pattern, `observer` preserves the state of `subject` and it is considered as state-preserving concern. It can, therefore, be defined as an aspect.

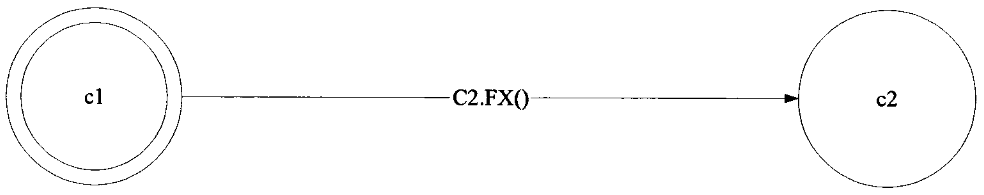


Figure 13. State-preserving concerns

Figure 14 illustrates two different scenarios for state-preserving and non-state-preserving cases. In some situations the receiver concern may not be a state-preserving concern, but we still may have crosscutting. To identify this type of crosscutting concerns we use a set of production rules and a transition matrix.

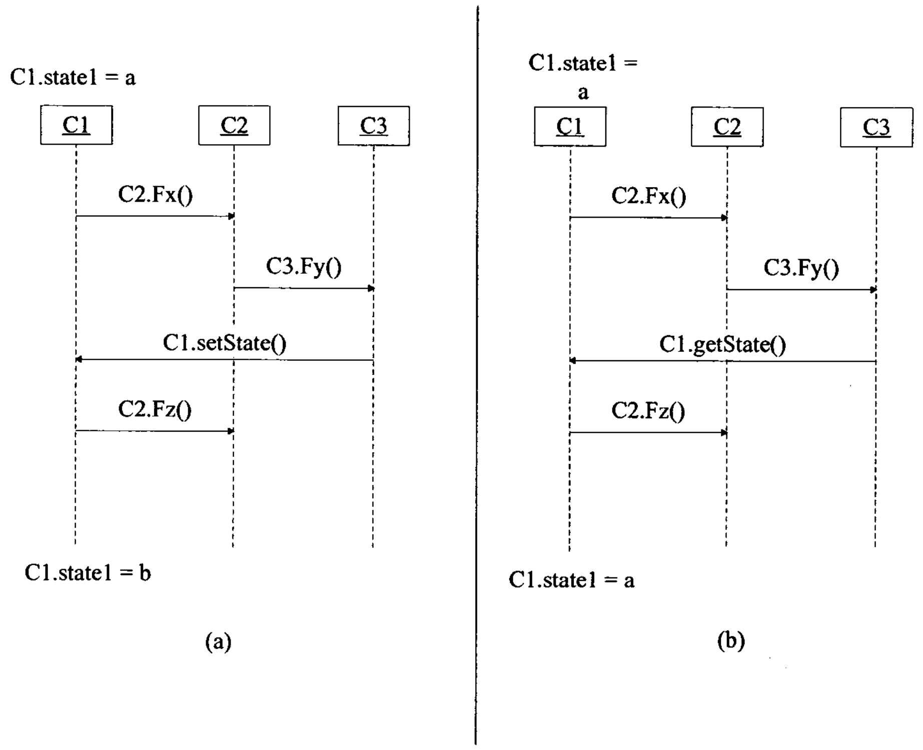


Figure 14. State-preserving scenario and non-state-preserving scenarios

Figure 14.a illustrates a non-state-preserving scenario, in this scenario the state of `C1` will change after calling `C2.Fx()`. Figure 14.b illustrates a state-preserving scenario it means the state of `C1` after calling `C2.Fx()` does not change

6.11. Identifying code duplication and scattering using production rules

Consider the following code segments, where `method1()` creates a sale order item and inserts its corresponding accounting transaction. Further, `method2()` creates a purchase order and inserts its corresponding accounting transaction.

```
// creating an invoice order
method1() {
    AccountingSystem acc = AccountingSystem();
    OrderItemBilling orderItem = OrderItemBilling();
    orderItem.setSalesOrderId(sid);
    orderItem.setSalesInvoiceId(sid);
    orderItem.setQuantity(quantity);
    orderItem.setAmount(amount);
    if (orderItem.insert()){
        acc.insert();
    }
}
method2() {
     AccountingSystem accountTransaction = AccountingSystem();
     OrderItemPayment orderItem = OrderItemPayment();
     orderItem.setPurchaseOrderId(sid);
     orderItem.setPurchaseInvoiceId(sid);
     orderItem.setQuantity(quantity);
     orderItem.setAmount(amount);
     if (orderItem.insert()) {
         accountTransaction.insert();
     }
  }
```

We can define the following production rules that describe the algorithm for `method1()` and `method2()`. The underlined code cuts across `method1()` and `method2()`. By using production rules we can identify concerns that are cutting across other concerns.

```
   Method1 ::=[declare] <AccountingSystem>

   Method1 ::=[declare] <OrderItemBilling>

   Method1 ::=[call] <OrderItemBilling.SetSalesOrderId()>

   ...

   Method1 ::=[call] <OrderItemBilling.insert()>

Method1 ::=[call] <AccountingSystem.insert()>
```

```
   Method2::=[declare]<AccountingSystem>

   Method2::=[declare]<OrderItemPayment>

   Method2::=[call]<OrderItemPayment.setPurchaseOrderId()>

   ...

   Method2::=[call]<OrderItemPayment.insert()>

Method2::=[call]<AccountingSystem.insert()>
```

6.12. Identifying code duplication and scattering using a transition matrix

A transition matrix is a two-dimensional matrix where in each row and column we can place classes and their methods and each cell of this matrix will be set to the values of 1 or 0. A value of 1 implies that [class.function] in ith row will call [class.function] in jth column. Figure 15 illustrates a transition matrix defined for classes `C1` and `C2`.

		C1			C2	
		f1	f2	f3	f4	f5
C1	f1				1	
	f2					1
	f3					
C2	f4					
	f5	1				

Figure 15. Transition matrix for class C1 and

`C2: C1.f1()→ C2.f4(); C1.f2()→c2.f5() and C2.f5()→C1.f1()`

Consider the following methods:

```
C1.M1()
{
    authenticator.authenticate();
    logger.log();
    contractChecker.precondition();
    do some business.
    C3.M3();
}
```

```
C2.M2()
{
    authenticator.authenticate();
    logger.log();
    contractChecker.precondition();
    do some business.
  C4.M4();
}
```

Figure 16 shows an associated transition matrix. From the table, we can observe that methods `logger.log()`, `contractChecker.precondition()` and `authenticator.authenticate()` cut across `C1.M1()` and `C2.M2()`. These three methods can be considered as separate aspects.

		C1			C2		C3		C4	Authenticator	Contract checker	Logger
		M1			M2		M3		M4	authenticate	precondition	log
C1	M1						1			1	1	1
C2	M2								1	1	1	1
C3	M3											
C4	M4											
Authenticator	authenticate											
Contract checker	precondition											
Logger	log											

Figure 16. Transition matrix for identifying aspects in C1.M1(), C2.M2()

7. Case study: Analysis of object-oriented artifacts metadata throughout the software life cycle

This section illustrates how we have applied our aspect mining methodology in the context of a case study throughout the software life cycle. The case study we have chosen is a sales order system.

Requirements

The system is capable of receiving multiple orders requests at the same time. The system requires its users to have certain level of privileges to access any of the above functionalities. The privileges are granted automatically upon a successful authentication. Considering the top-level use case diagram (Figure 17), we can define this use case by the following production rules:

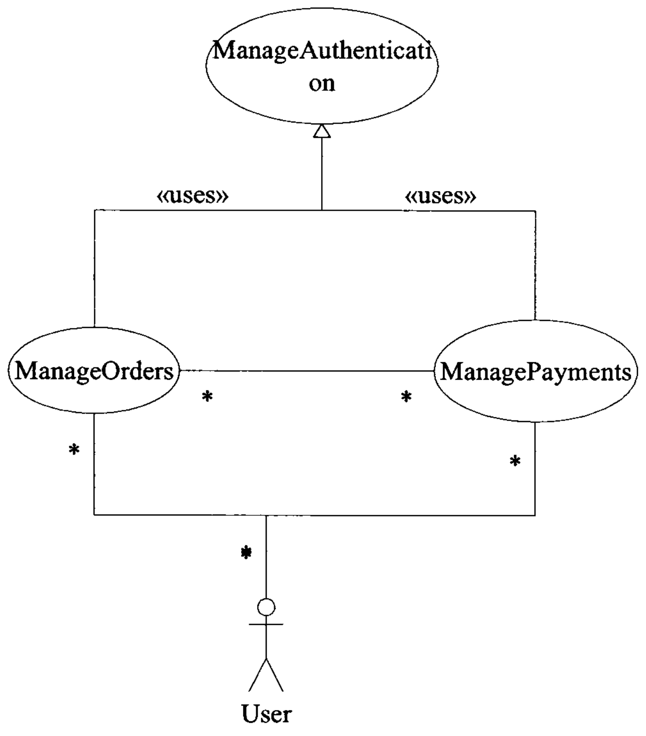


Figure 17. Top level use case diagram for invoicing system

```
<User>::=[use]<ManageOrders>

<User>::=[use]<ManagePayments>

<ManageOrders>::=[associate]<ManagePayments>

<ManageOrders>::=[use]<ManageAuthentication>

<ManagePayments>::=[use]<ManageAuthentication>
```

Using the production rules we can deduce certain derivations which illustrate that `ManageOrders` and `ManagePayments` use cases are dependent on `ManageAuthentication`, but `ManageAuthentication` is independent from these two and can therefore be considered as a crosscutting use case.

```
<User>::=[use]<ManageOrders>::=[use]<ManageAuthentication>
```

```
<User>::=[use]<ManagePayments>::=[use]<ManageAuthentication>
```

`ManageOrder` use case can be modeled as two main use cases `placeOrder` and `makePayment`. Figure 18 illustrates next level of sales order system use case that extends `ManageOrder` use case defined at the top level use case.

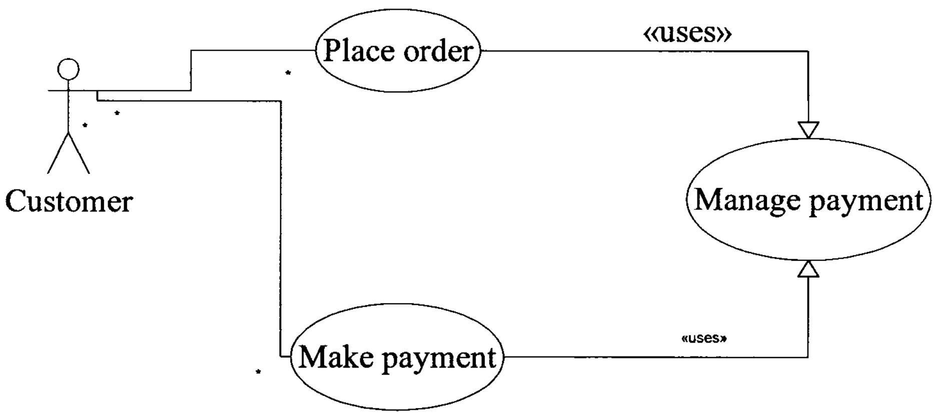


Figure 18. Main use cases of manage order functionality

Production rules corresponding to this use case are as follow:

```
<Customer>::=[use]<Place order>

<Customer>::=[use]<Make payment>

<Place order>::=[use]<Manage payment>

<Make payment>::=[use]<Manage payment>
```

We see that `Manage Payment` poses a concern which crosscuts use cases `Place Order` and `Make payment`. We choose to note this knowledge on a refined use case diagram (Figure 19) even though we have to stress that at this stage we do not have enough knowledge on the exact nature of crosscutting, until these use cases will be further refined into their constituent components.

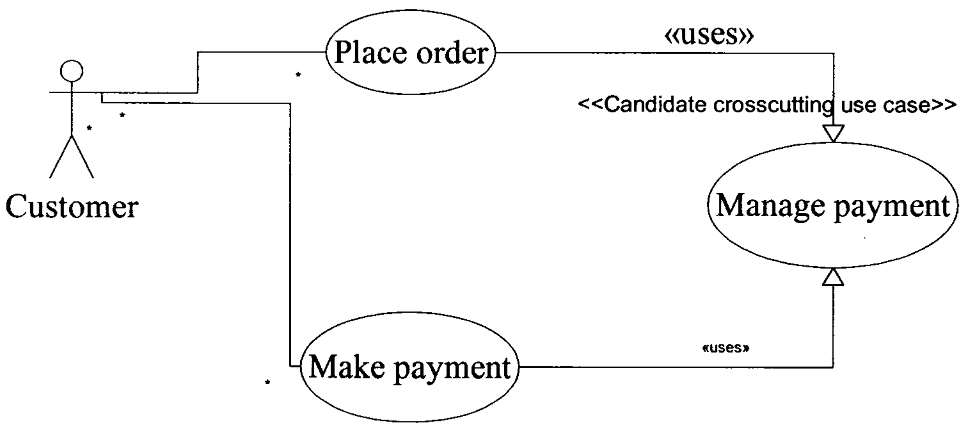


Figure 19. Use case diagram after applying metadata analysis.

Analysis

We model each top level use case in a domain model (Figure 20). A customer can be related to one or more sales orders. When items have been ordered, it is critical for the enterprise to make sure that it requests payments; this is done through invoices. In this simplified model order items have a one-to-one relationship with invoice items. Invoices are issued to request payment. Order transaction and invoice transaction affect the accounting of the organization. The creation of an invoice will result in a related accounting transaction, namely a sales accounting transaction. The creation of a payment also will result in a related accounting transaction [11]. The interaction between an actor and the system as captured in a use case scenario can be illustrated in a system sequence diagram (SSD) where the system is viewed as a black box. We may also view the entire system as a group of black-box subsystems. Figure 21 illustrates the system sequence diagrams for the success scenarios of placing an order and making a payment.

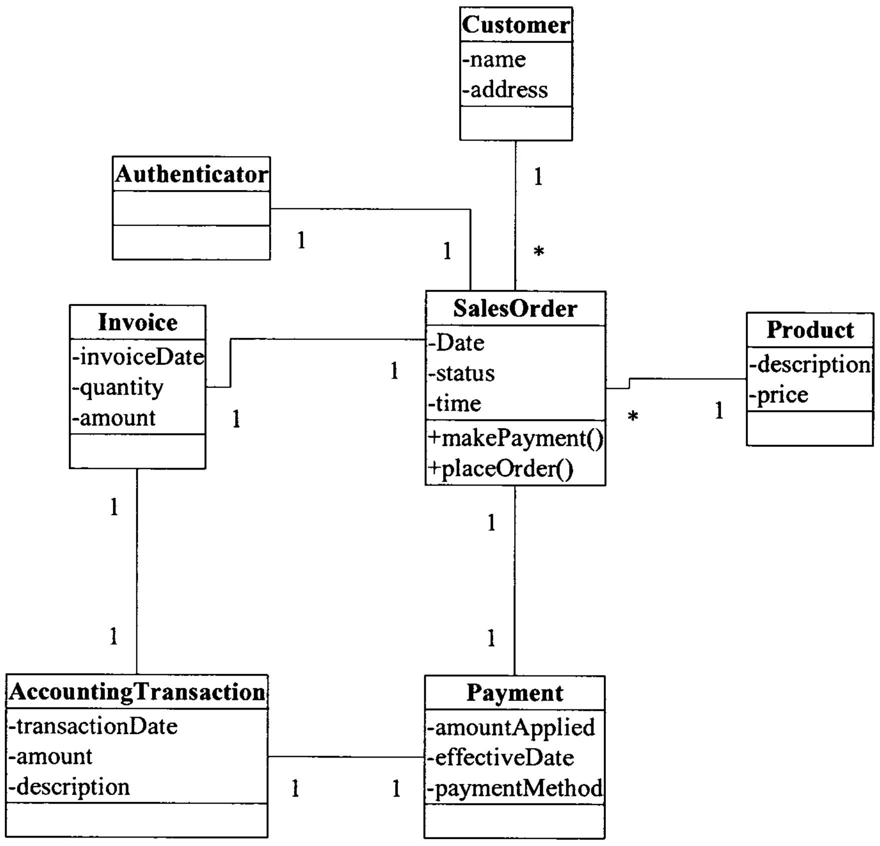
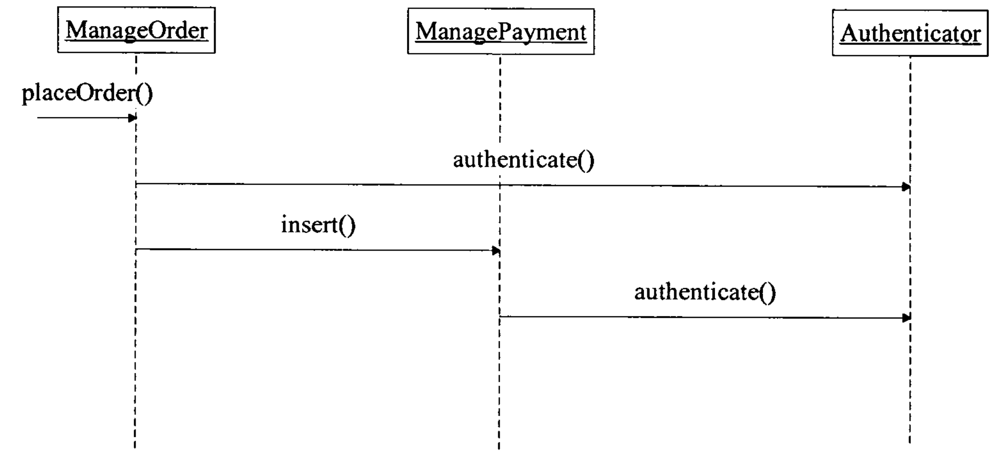


Figure 20. Sales order domain model

Based on the two system sequence diagrams, we can define a sequence of production rules, one for each system operation. From the production rules, we observe that the subsystem for `Authenticate()` method of Authenticator class crosscuts `placeOrder()` and `makePayment()` methods the main system operations. As you can see there is a traceability of crosscuttings behaviors in different steps with different granularity from requirement to analysis, design and at last to the implementation.

```
-- Place Order production rules

<User>::=[call]<ManageOrder.placeOrder()>

::=[call]<manageAuthentication.authenticate()>

::=[call]<managePayment.insert()>

::=[call]<manageAuthentication.authenticate()>
```

```
-- Make payment production rules
<User>::=[call] <manageOrder.makePayment()>
::=[call]<manageAuthentication.authenticate()>
::=[call]<managePayment.insert()>
::=[call]<manageAuthentication.authenticate()>
```



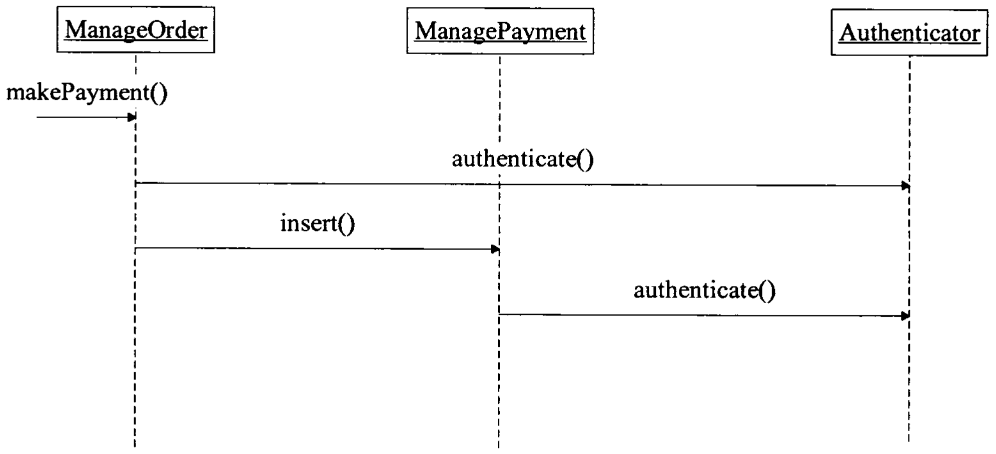


Figure 21. SSDs for placing an order and making a payment

Design

As top level uses case can be modeled as components with a well-defined interface (Figure 22). The interface corresponding to `ManageAuthentication` component is defined as an aspect that will contain functions to implement the interface and advice definitions. Our design is illustrated in Figure 22 and the rationale behind certain important design decisions is described next:

(1) As the system would require maintaining history information on any modifications of Order, Invoice and Payment instances, we can deploy the Observer design pattern to implement this requirement. Request for instance modification would initially notify History Observer to produce a copy of the instance, before the modification takes place.

(2) The synchronization policy and persistency can be implemented by the Composite design pattern.

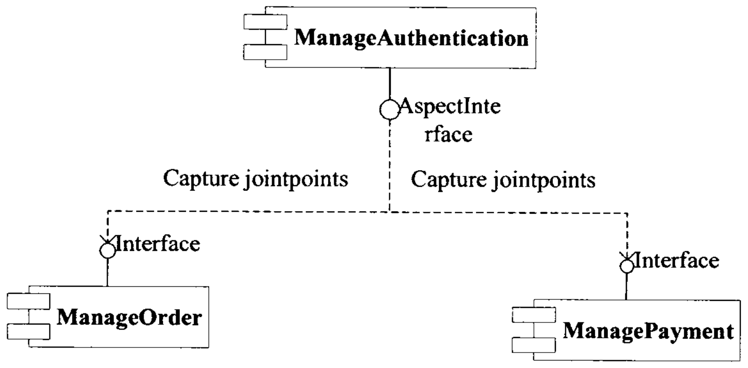


Figure 22. Component diagram representation of the system

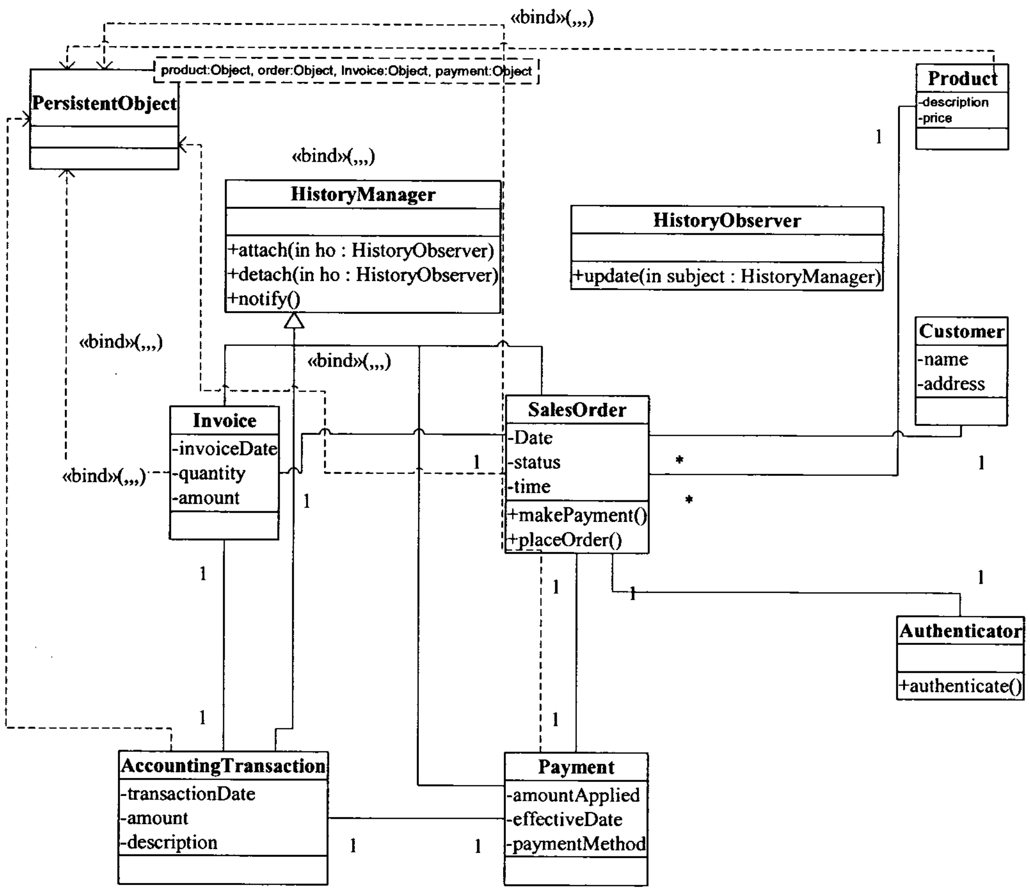


Figure 23. Class diagram

Accounting Transaction

We implement each system operation from an SSD, as an interaction diagram (Figure 34), illustrating how the responsibilities can be implemented. In the corresponding interaction diagrams of `placeOrder()` and `makePayment()` system operations, we can observe the following dependencies: Both Invoice and Payment are dependent on Accounting Transaction (but not vice versa). Based on these semantics, we can formulate the following production rules. From the rules we can identify the behavior of Accounting Transaction as crosscutting which is initiated by the calls to `invoice.insert()` and `payment.insert()`, both of which can be considered candidate joinpoints. In this case study, we will see the traceability of crosscuttings from the requirement to design but with different granularity.

```
-- insert invoice

<invoice.insert()> ::=[call]<invoice.insert()>::=

[call]<accountingTransaction.insert()>


--insert payment

<payment.insert()>::=[call]<payment.insert()>::=

[call]<accountingTransaction.insert()>
```

`accountingTransaction` has an independent role in these two scenarios and then can be defined as an aspect.

PersistentObject

In this model we use a composition pattern `PersistentObject` that is used for supplementing persistency behavior for all business objects such as product, customer, sales order, invoice, payment, and accounting transaction. In Figure 24, class `PersistentOrder` is a parameterized class that supplement persistency for class Order through the `_insert()` method. This implies that we can define class `PersistentOrder` as an aspect to declare a super type for class Order.

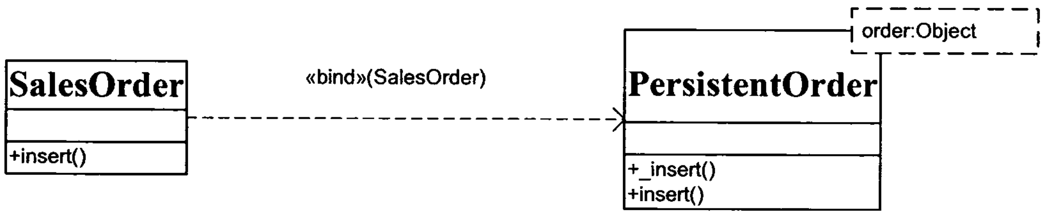


Figure 24. Composition pattern for modeling persistent objects

Class PersistentOrder adds more functionality (persistency) to Class Order through _insert () method

```
<PersistentObject>::=<supplement><Customer>

<PersistentObject>::=<supplement><Product>

<PersistentObject>::=<supplement><SalesOrder>

<PersistentObject>::=<supplement><Invoice>

<PersistentObject>::=<supplement><Payment>

<PersistentObject>::=<supplement> <AccountingTransaction>

<Customer>::=<inherit>PersistentObject.insert()

<Customer>::=<inherit>PersistentObject.update()

<Customer>::=<inherit>PersistentObject.delete()

<Product>::=<inherit>PersistentObject.insert()

<Product>::=<inherit>PersistentObject.update()

<Product>::=<inherit>PersistentObject.delete()

...
```

Observer design pattern: History manager

Consider the Observer design pattern that defines a one-to-many dependency between objects [8]. The key objects in this pattern are the subject and the observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with that of the subject's. In [1] we demonstrated how the Observer design pattern can be regarded as a crosscutting concern. In the current case study we can use the Observer pattern to model a History Manager that will maintain information on modifications of Order, Invoice, Payment, and Accounting transaction.

Implementation

In case study 3 we will show how to use dynamic syntax grammars to identify aspect candidates by parsing code artefacts.

8. Case study: Reengineering object-oriented designs by analyzing dependency graphs and production rules

To illustrate our approaches we will adopt the case study of a point-of-sale system, described in [6]. The success scenario of the Process Sale use case and the corresponding system sequence diagram are illustrated in Figure 25, and the domain model is illustrated in Figure 26.

Use case: Process sale (Successful scenario)

Pre conditions:

1. An Instance of customer ***c*** exists.
2. An Instance of product ***p*** exists.
3. Number of available product ***p*** is greater that the order quantity.

Post conditions:

1. An instance of sales item ***si*** for each purchase item was created.
2. Customer ***c*** was associated with an on hold sales order ***so***.
3. Product ***p*** was associated with sales item order ***si***.
4. Quantity of sales item ***si*** deducted from number of available product ***p.***
5. An instance of payment ***pt*** associated with sales order ***so*** was created.
6. Status of sales order ***so*** set to paid.

Steps:

1. The Customer arrives at a POS checkout with items to purchase.

2. The cashier records the identifier for each product item.
3. The system determines the item price and adds the item information to the running sales transaction.
4. On completion of item entry, the Cashier indicates to the POS system that item entry is complete.
5. The System calculates and presents the sale total. The Cashier tells the customer the total.

6. The Customer gives cash payment ("cash tendered") possibly greater than the sale total.

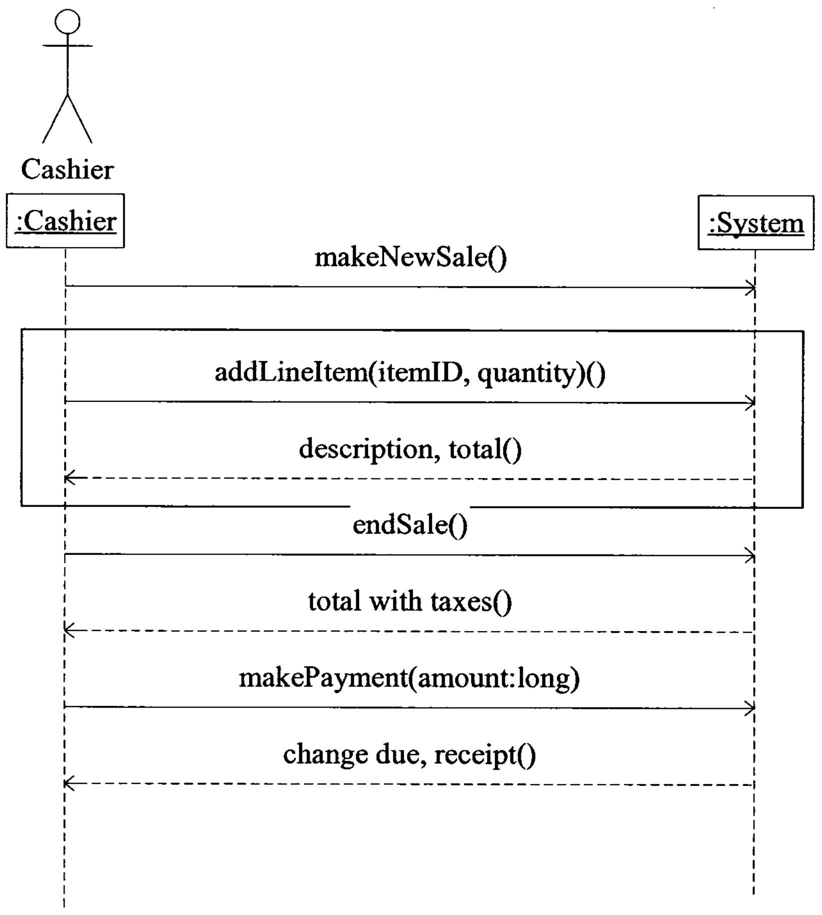


Figure 25. Process sale scenario

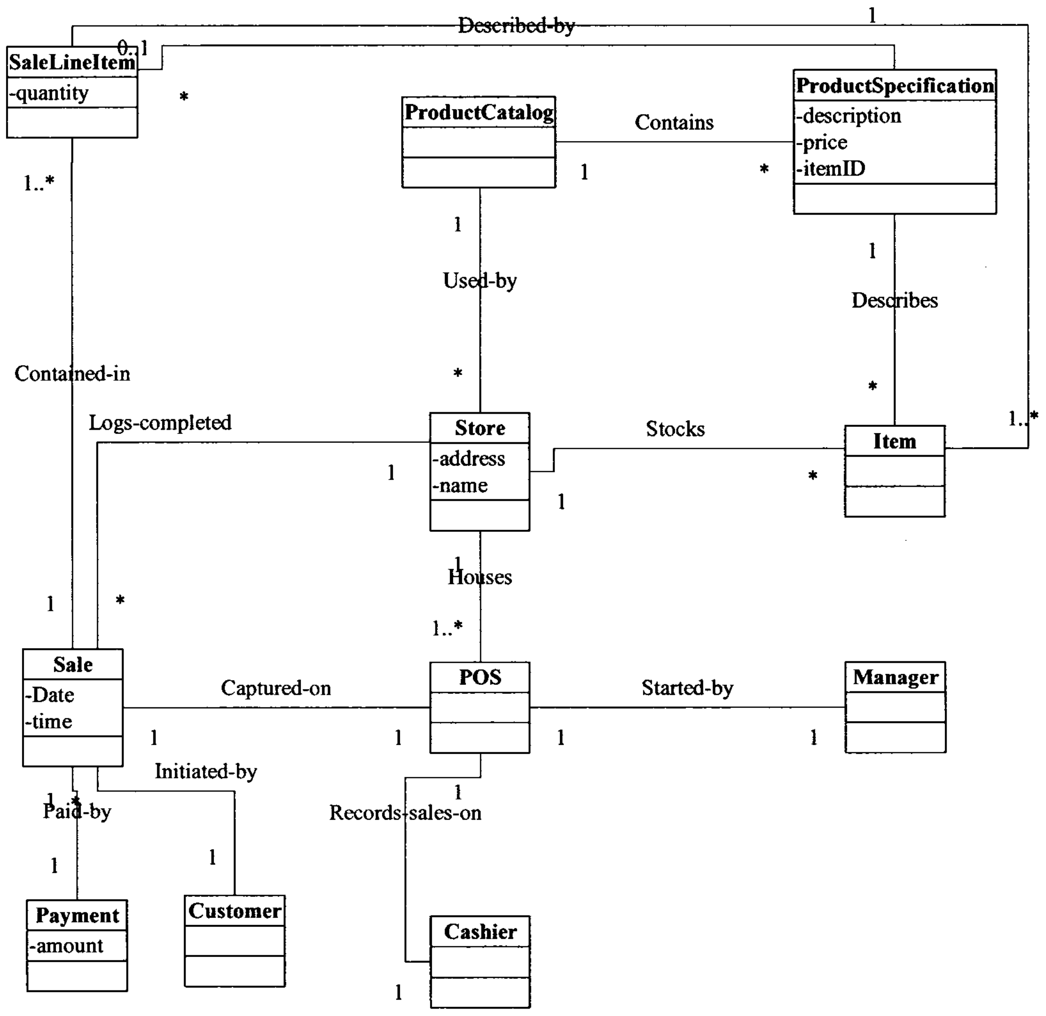


Figure 26. Domain model for the point of sale system

We design four models for applying data consistency and integrity and contract checking mechanism in this system. We categorize this contract checking or validation mechanism in two levels (1) Instance level (in terms of object invariants), (2) scenario level (in terms of preconditions and postconditions). The designs are described in the following subsections.

4.1 Design 1: Contract checking in each class: Figure 27 illustrates the first design for applying contract checking mechanism. In this model, each class is responsible to make sure to validate rules for invariants, and also validate preconditions and post-conditions of its operations.

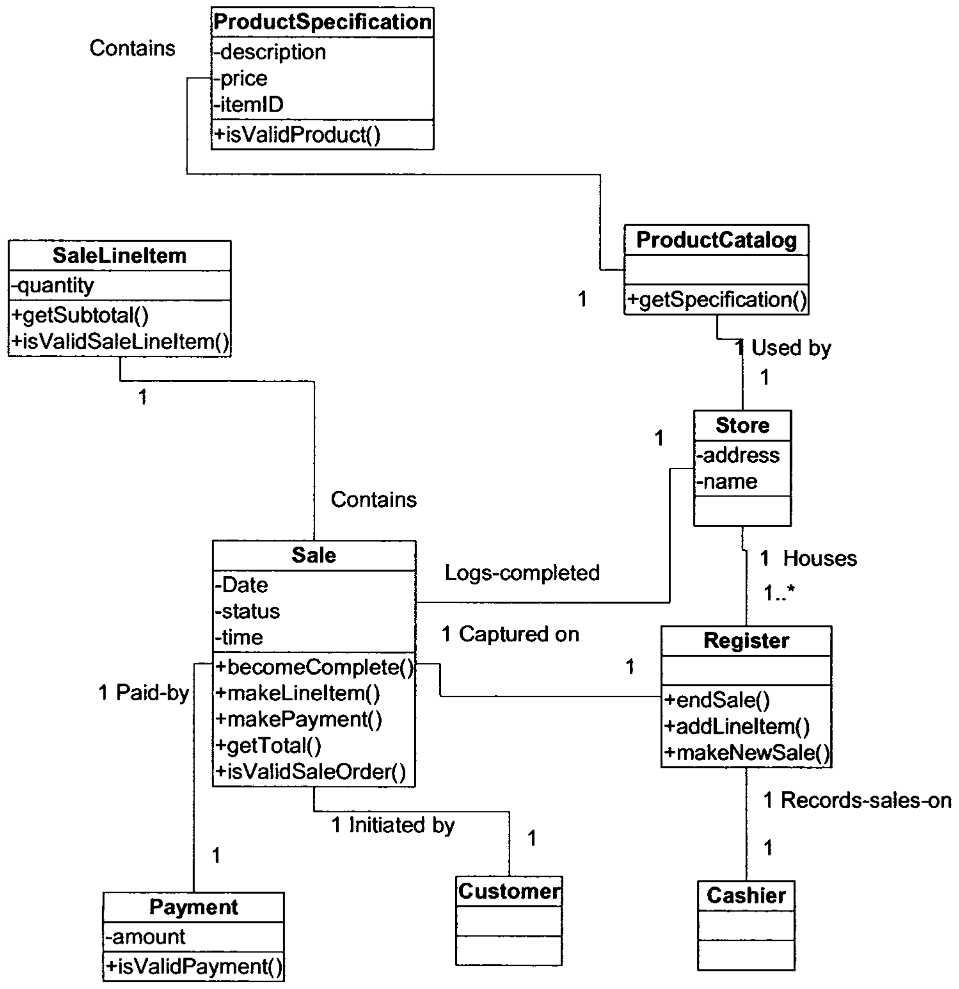


Figure 27. Class diagram for sales system

- Method `Product::isValidProduct()` validates that the unit price of the product is greater that zero, the `productCode` is unique, and `availableQuantity` is not negative.

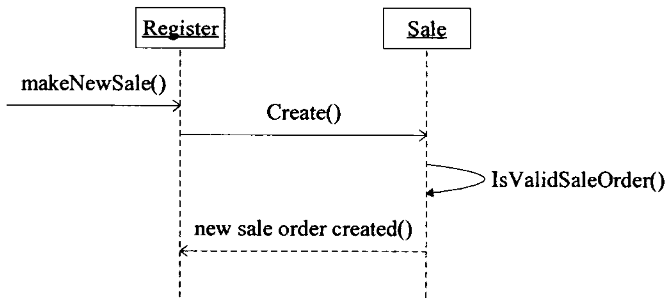
- Method `SalesLineItem::isValidSaleLineItem()` validates that quantity of sale item is greater than zero and that there is enough available quantity of requested product.
- Method `Payment::isValidPayment()` validates that the payment amount is equal to corresponding sale order total price.
- Method `SalesOrder::isValidSaleOrder()` is used for placing and order and it checks that order date is correct date format and not empty, the order status is not completed.

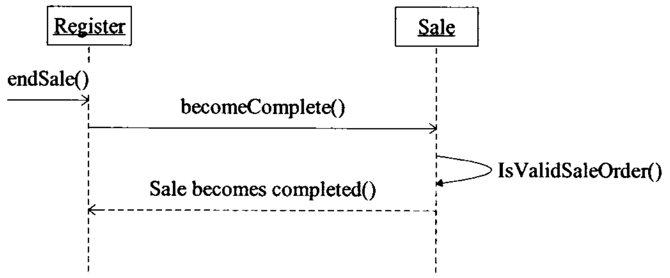
The following sequence diagrams (Figure 28) illustrate placing an order by a cashier. According to this sequence diagram when placing an order the following validations are done:

1. Date of order is correct and a unique sales order ID is exists.
2. For each valid sale item (sale item with quantity greater than zero) a valid product with unitPrice value greater than zero and enough quantity must be available.

According to these sequence diagrams contract checking methods such as `isValidSaleOrder()` are scattered all over the code and then can be implemented as an aspect. The production rules in Listing 5 show the scattering of `isValidSaleOrder()` method in the three different scenarios.

Further, if we consider some other scenarios then we will see that other contract checking methods are scattered too.





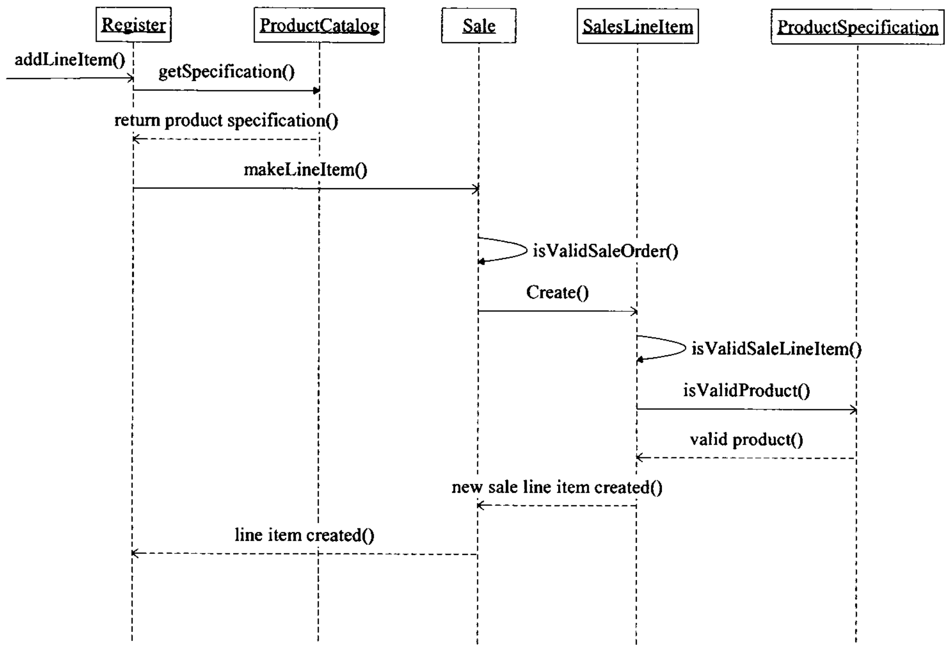


Figure 28. Processing a sale for the first design model

Listing 5.

```
{call Register.makeNewSale()}
-> <Register.makeNewSale()>
::=[call]<Sale.Create()>
::=[call]<Sale.isValidSaleOrder()>


{call Register.endSale()}
-> <Register.endSale()>
::=[call]<Sale.becomeComplete()>
::=[call]<Sale.isValidSaleOrder()>


{call Register.addLineItem()>
-> <Register.addLineItem()>
::=[call]<ProductCatalog.getSpecification()>
::=[call]<Sale.makeLineItem>
::=[call]<Sale.isValidSaleOrder()>
::=[call]<SaleLineItem.Create()>
...
```

4.2 Design 2: Contract checking in a single class: In this second design, we adopted a separate class to validate the contracts defined for each object and each scenario (Figure 29). An alternative design would use the composition pattern.

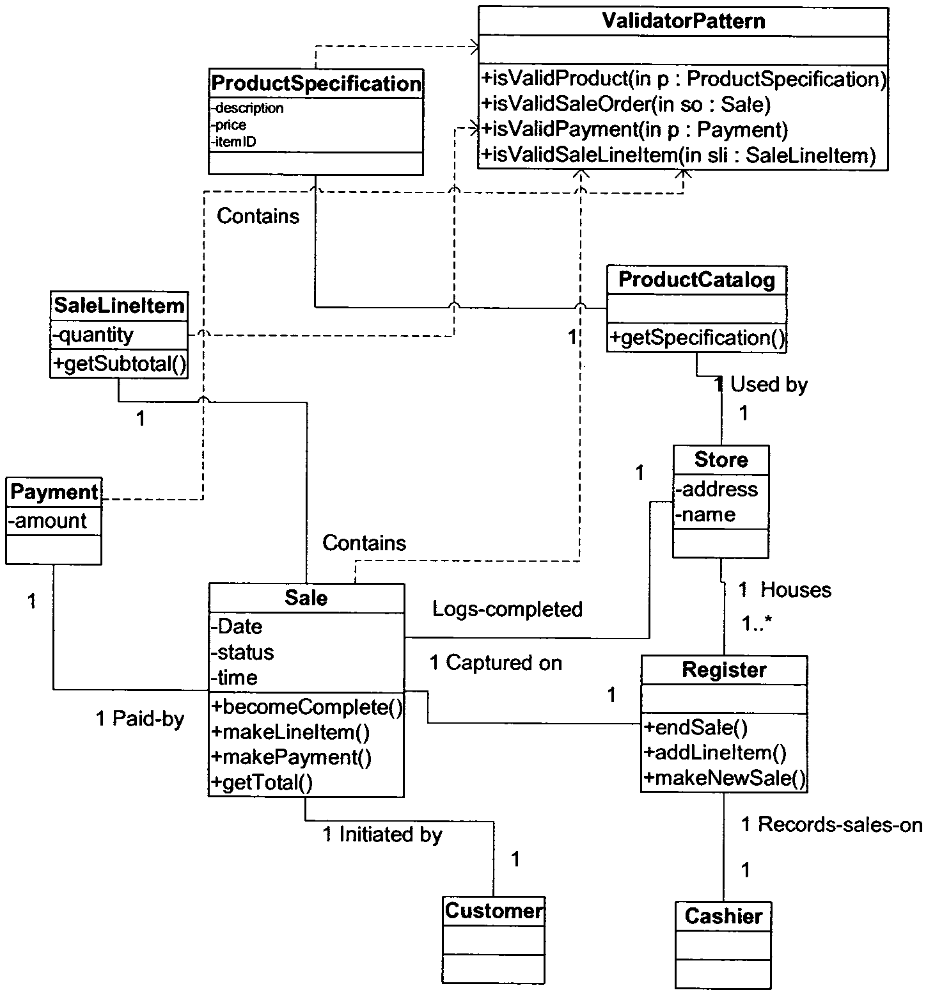


Figure 29. Validator pattern

In this model the Validator pattern has different functions for each class or even each scenario. Method `isValidCustomer()` accepts a customer objects and apply contract checking defined for the customer class to this object. For example it checks the customer object has a unique ID and a non empty name. Method `isValidProduct()` validates that the unit price of the product is greater that zero, the `productCode` is unique, `availableQuantity` is not negative. We can define the following production rules for this model, shown in Listing 6.

Listing 6.

```
<Validator>::=<supplement><Product>

<Validator>::=<supplement><SalesOrder>

<Validator>::=<supplement><Payment>

...

<Product>::=<inherit>Validator.isValidProduct()

<SalesOrder>::=<inherit>Validator.isValidSalesOrder()

<Payment>::=<inherit>Validator.isValidPayment()
```

By using these production rules we can define the corresponding dependency graph (Figure 30) which implies that class `Validator` can be defined as an aspect with the responsibility to validating all other objects.

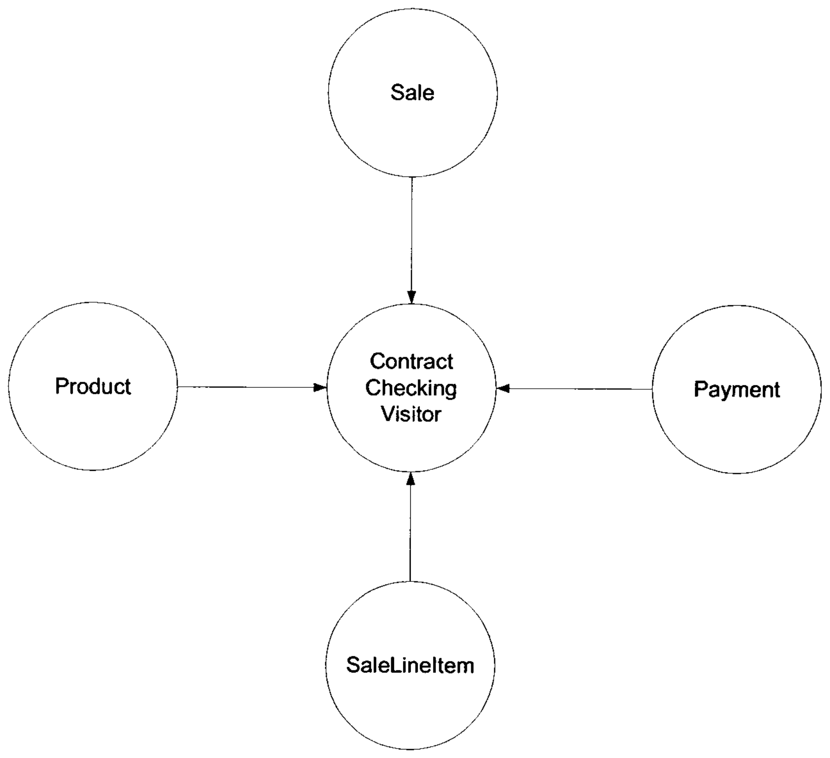


Figure 30. Dependency graph for the validator pattern

4.3 Design 3: Deploying the Observer design pattern for validation: The third developer uses the Observer pattern (Figure 31) to consider a status for each object and defines four different levels: "uncommitted", "committed", "invalid", and "valid". When an object of any type is created its status set to "uncommitted" that means the object can be an invalid object but no contract checking mechanism is applied to it yet to identify if it is invalid or not. Whenever in the scenario we need to validate an object and apply appropriate contract checking mechanism, we change the object's status to "commited" in this moment an observer notify about this modification and apply appropriate contract checking mechanism that at the end it sets the status of the object to "valid" or "invalid".

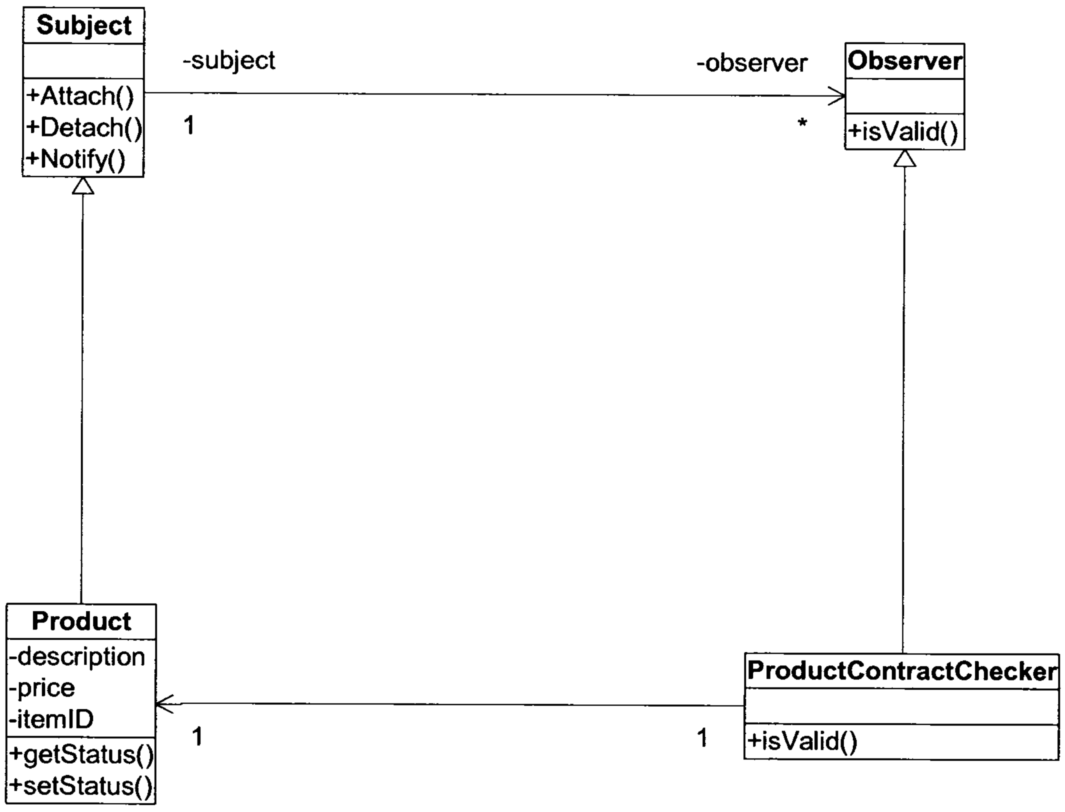


Figure 31. Observer pattern for contract checking

In this model the observer can be modeled as a crosscutting concern such that whenever method `Product::setStatus()` is called to change the status of the object from 'uncommitted' to 'committed', the aspect will be activated and apply contract checking defined for the object and change the status of the object to "invalid" or "valid". Figure 32 illustrates the object dependency graph for this scenario with different instances of `Product` and `ProductContractChecker`. According to this dependency graph a similar interaction will take place for each `Product` instance whenever its status changes to "committed" by calling `Product.setStatus()` method.

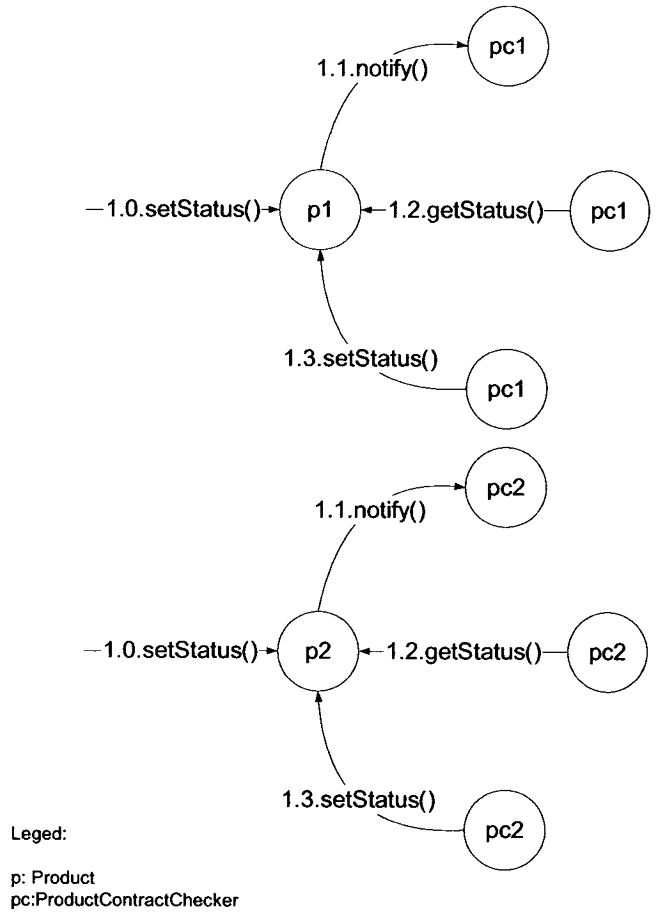


Figure 32. Object dependency graph for contract checker observer design pattern

4.4 Design 4: Deploying the Visitor design pattern for validation: In this design we deployed the Visitor design pattern [6] to apply contract checking mechanisms (Figure 33). When an element accepts a visitor it sends a request to the visitor that encodes the corresponding class. The visitor will then execute the contract checking for that element (as opposed to the operation being part of the class of the element as in the first design).

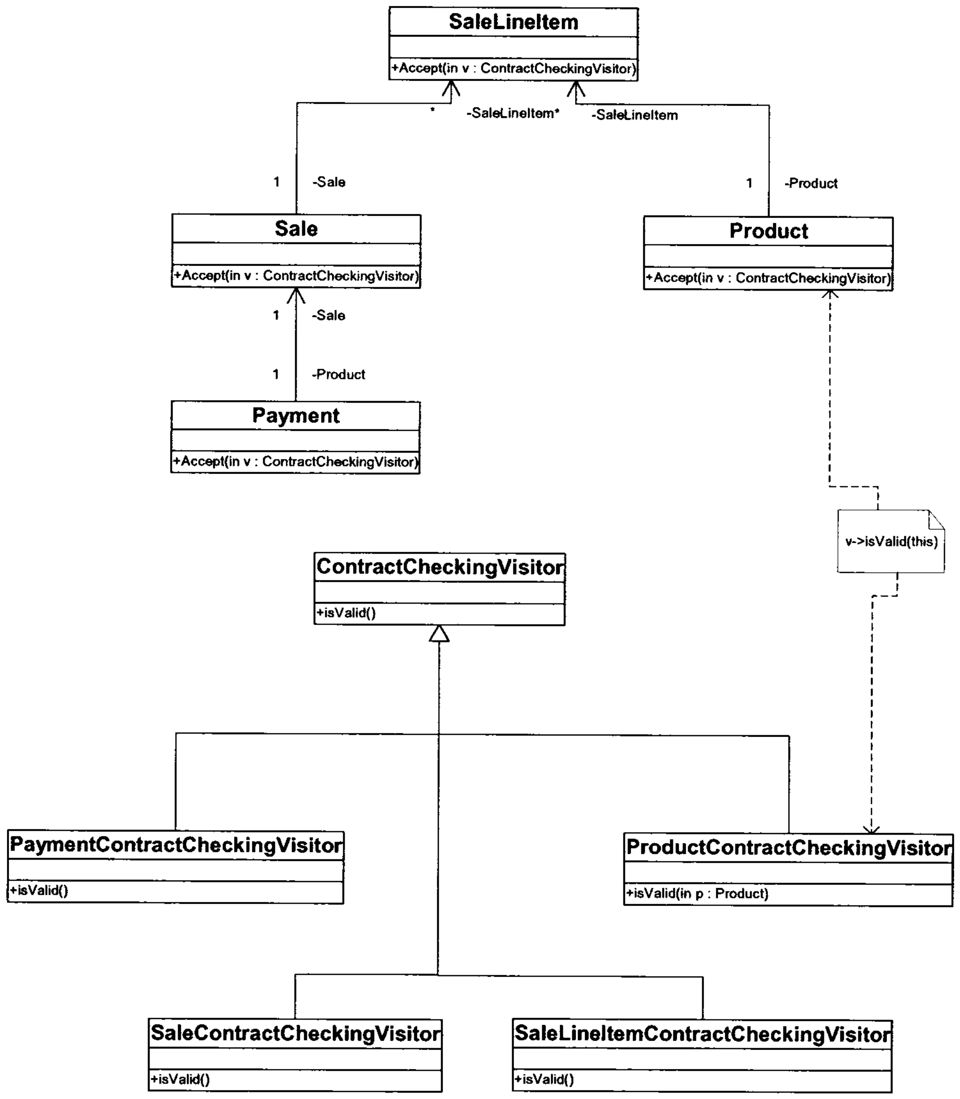

SaleLineItem
+Accept(in v : ContractCheckingVisitor)
-SaleLineItem*
-SaleLineItem
1
-Sale
1
-Product
Sale
+Accept(in v : ContractCheckingVisitor)
Product
+Accept(in v : ContractCheckingVisitor)
1
-Sale
1
-Product
Payment
+Accept(in v : ContractCheckingVisitor)
v->isValid(this)
ContractCheckingVisitor
+isValid()
PaymentContractCheckingVisitor
+isValid()
ProductContractCheckingVisitor
+isValid(in p : Product)
SaleContractCheckingVisitor
+isValid()
SaleLineItemContractCheckingVisitor
+isValid()

Figure 33. Deployment of Visitor pattern

In this model, the developer has tried to separate contract checking mechanism from the rest of the algorithms and business logic. By analysis of the dependency graph of this model (Figure 34) one can identify that the visitor classes are providing and supplementing contract checking mechanism without effecting the main functionality of the scenarios, that can be considered as an aspect.

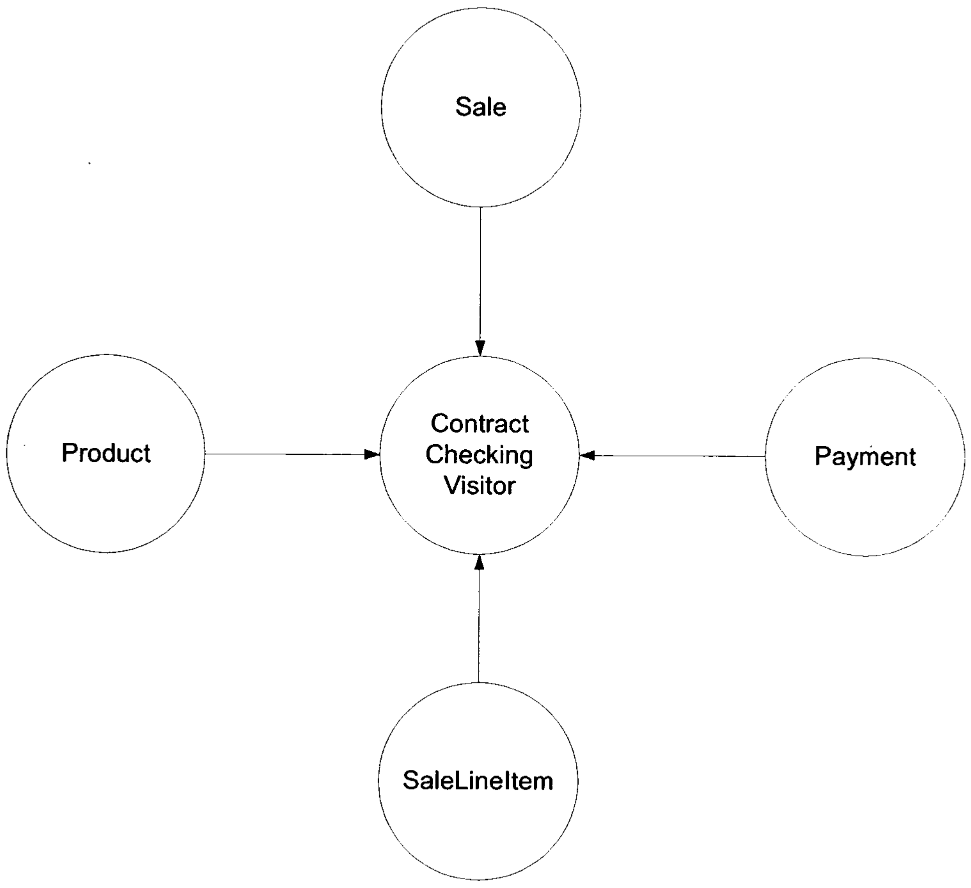


Figure 34. Dependency graph for the Visitor design pattern

9. Case study: Using dynamic syntax grammars for detecting crosscutting concerns

This section illustrates how we have applied dynamic syntax grammars to identify crosscutting concerns.

9.9. Using dynamic syntax grammar for identifying horizontal concerns

Example 1: Sales system: identifying horizontal concerns

Figure 35 illustrates a class diagram for sales system.

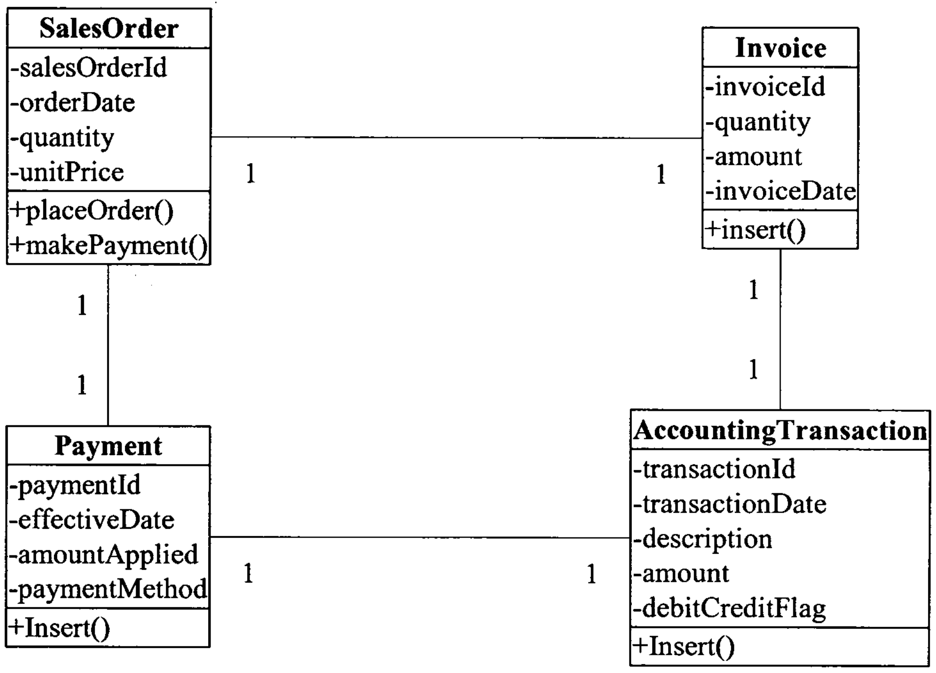
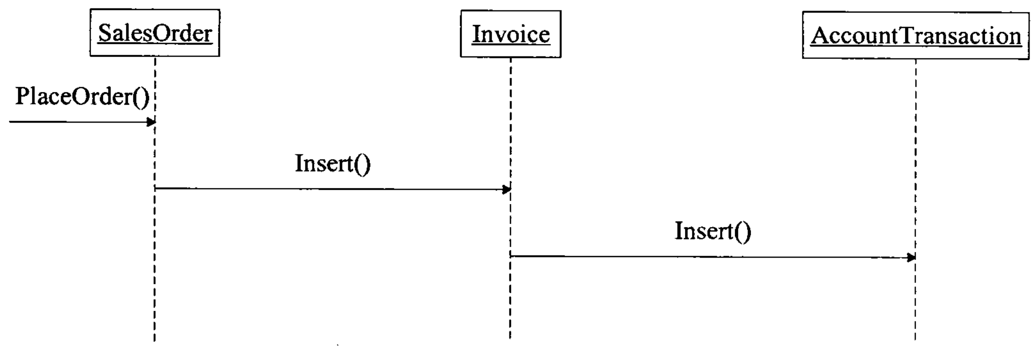


Figure 35. Class diagram for sales system

Figure 36 illustrates interaction diagrams for `placeOrder()` and `makePayment()` methods of `SalesOrder` class.



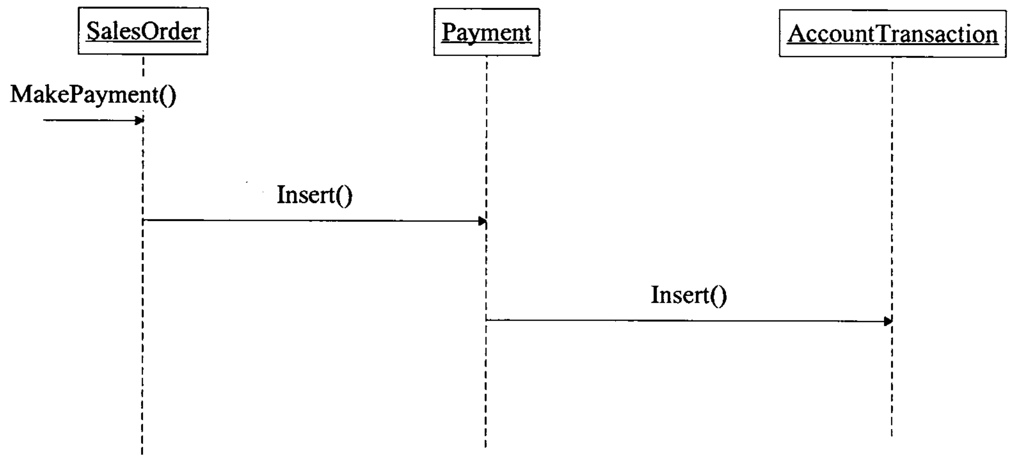


Figure 36. Interaction diagram for PlaceOrder() and MakePayment() methods of SalesOrder class

We implement our design by using Java language (Listing 7):

Listing 7: Implementation of Sales system by Java language

```
public class SalesOrder {

      String salesOrderId;

      String orderDate;

      String quantity;

      int unitPrice;

      Invoice invoice;
```

```
        Payment payment;

        //...

        public int placeOrder() {

                int errorCode = 0;

                //...

                invoice.insert();

                //...

                return errorCode;

        }

        public int makePayment() {

                int errorCode = 0;

                //...

                payment.insert();

                //...

                return errorCode;

        }

}
```

```
public class Invoice {

        String invoiceId;

        int quantity;

        int amount;

        String invoiceDate;

        AccountingTransaction accountTransaction;

        //...

        public int insert() {

                int errorCode = 0;

                //...

                accountTransaction.insert();
```

```
            //...
            return errorCode;
        }
}
```

```
public class Payment {
   String paymentId;
   String effectiveDate;
   int amountApplied;
   int paymentMethod;
   AccountingTransaction   accountTransaction;
   //...
   public int insert() {
      int errorCode = 0;
      //...
      accountTransaction.insert();
      //...
      return errorCode;
   }
}
```

```
public class AccountingTransaction {
      String transactionId;
      String transactionDate;
      String description;
      int amount;
      int debitCreditFlag;

      public int insert() {
            int errorCode = 0;
```

```
              //...
              return errorCode;
       }
}
```

First we use a top-down parsing strategy to show the parse tree for `makePayment()` and `placeOrder()` methods. Listing 8 illustrates BNF rules of java language.

Listing 8. BNF rules of Java

```
class_declaration ::= "class" identifier "{"  <field_declaration>  "}"
field_declaration ::= (method_declaration | constructor_declaration |
variable_declaration)
method_declaration ::= < modifier > type_specifier identifier "("
[parameter_list] ")"  statement_block
modifier ::= "public" | "private" | "protected" | "static" | "final"
type_specifier ::= "boolean" | "byte" | "char" | "short" | "int" |
"float" | "long" | "double" | class_name
class_name ::= identifier
constructor_declaration ::=  < modifier > identifier  "("  [
parameter_list ]  ")" statement_block
statement_block ::= "{"  < statement >  "}"
variable_declaration ::= < modifier > type identifier ";"
parameter_list ::= parameter  <  "," parameter >
parameter ::= type identifier
statement ::= variable_declaration | (expression ";") |
(statement_block) | (if_statement) | (do_statement) | (while_statement)
 | ("return" [expression] ";") | (";")
if_statement ::= "if"  "(" expression ")" statement ["else" statement]
```

```
do_statement ::= "do" statement "while" "(" expression ")"  ";"

while_statement ::= "while"  "(" expression ")" statement

expression ::= numeric_expression | logical_expression |
string_expression | casting_expression | creating_expression | "null" |
"super" | "this" | identifier | ("(" expression ")") | (expression
(("(" [arglist] ")") | ("." expression) ))

numeric_expression ::= (("-" | "++" | "--") expression) | (expression
("++" | "--"  )) | (expression ("+" | "+=" | "-" | "-=" | "*" | "*=" |
"|" | "/=" | "%" | "%=") expression)

testing_expression ::= (expression (">" | "<" | ">=" | "<=" | "==" |
"!=") expression)

logical_expression ::= ("!" expression) | (expression ("ampersand" |
"ampersand=" | "|" | "|=" | "^" | "^=" | ("ampersand""ampersand") |
"||=" | "%" | "%=") expression) | "true" | "false"

string_expression ::= (expression ("+" | "+=") expression)

bit_expression ::= ("~" expression) | (expression (">>=" | "<<" | ">>"
| ">>>") expression)

casting_expression ::= "(" type ")" expression

creating_expression ::= "new" classe_name

arglist ::= expression <"," expression>

integer_literal ::= "1..9"

float_literal ::= decimal_digits "." [ decimal_digits ]

decimal_digits ::= "0..9"

character ::= "based on the unicode character set"

string ::=  "''" <character> "''"

identifier ::= "a..z,$,_" <"a..z,$,_,0..9,unicode character over 00C0">
```

Figure 37 illustrates a partial parse tree for `class SalesOrder` and `placeOrder()` and `makePayment()` methods.

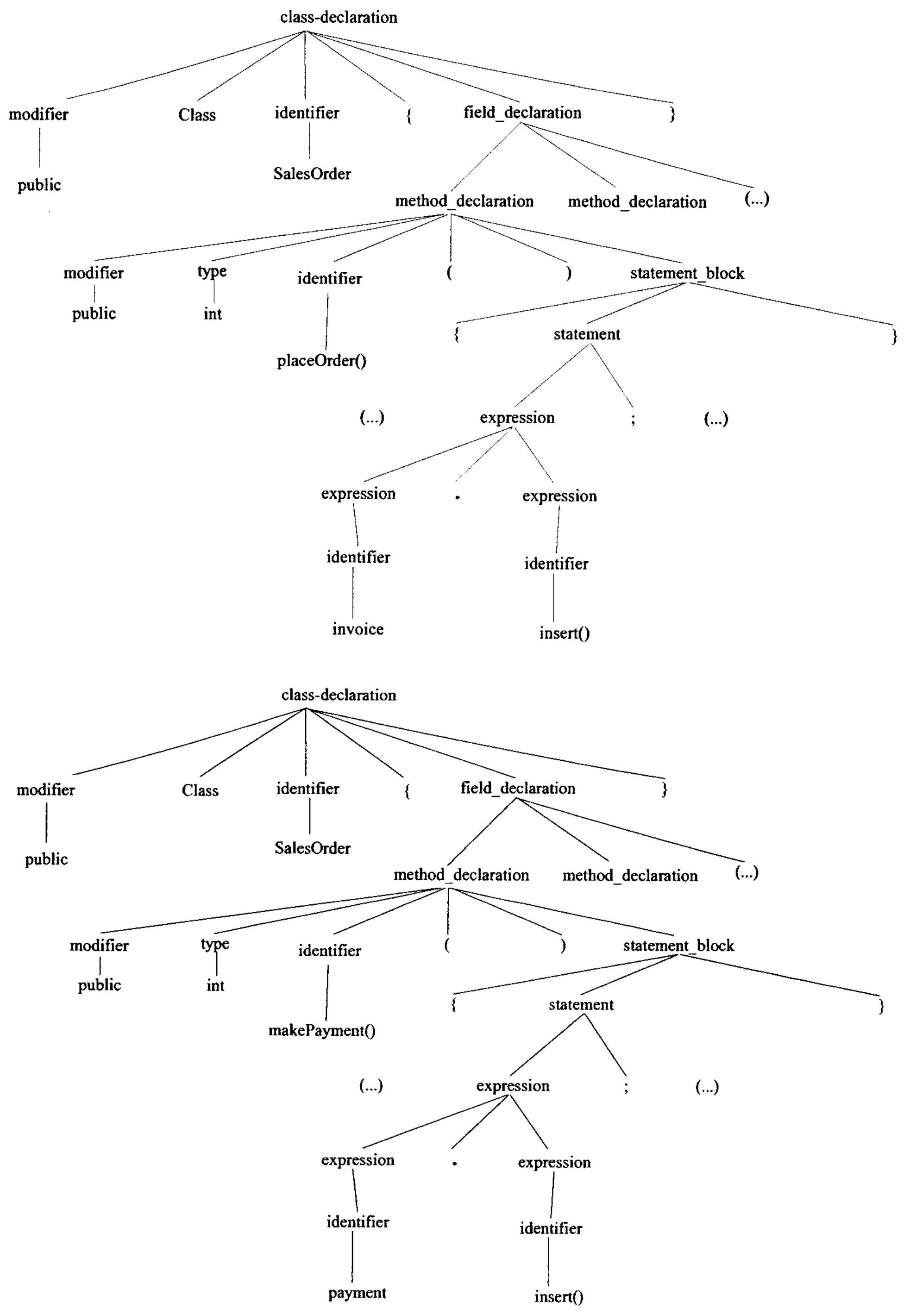


Figure 37.Parse tree for class SalesOrder class

The grammatical form of Java language dose not have any design semantics information to analyze the code, we inject design semantics to the grammatical form with using design level production rules and defining a two-level grammar.

We can define the following production rules to present the design semantics for sales system (Listing 9):

Listing 9. Sales order semantics.

```
<SalesOrder>::=[has]<MakePayment()>

<SalesOrder>::=[has]<PlaceOrder()>

<SalesOrder>::=[declare]<Payment>

<SalesOrder>::=[declare]<Invoice>

<SalesOrder.MakePayment()>::=[call]<Payment.Insert()>

<SalesOrder.PlaceOrder()>::=[call]<Invoice.Insert()>

...

<Payment>::=[has]<Insert()>

<Payment>::=[declare]<AccountTransaction>

<Payment.Insert()>::=[call]<AccountTransaction.Insert()>

...

<Invoice>::=[has]<Insert()>

<Invoice>::=[declare] <AccountTransaction>

<Invoice.Insert()>::=[call]<AccountTransaction.Insert()>

...

<AccountTransaction>::=[has]<Insert()>

...
```

Meta-variables for two-level grammar are defined as:

```
CLASS_SET = {SalesOrder, Invoice, Payment}

METHOD_SET = {SalesOrder.placeOrder(),

SalesOrder.makePayment(), Invoice.insert(),

Payment.insert(), AccountingTransaction.insert(), …}
```

Then for two-level grammar we need to modified java grammatical form with using Meta-variables. Listing 9 illustrates java grammatical form with using meta-variables:

```
class_declaration ::= "class" CLASS_SET "{"  <field_declaration>  "}"

field_declaration ::= (method_declaration | constructor_declaration |
variable_declaration)

method_declaration ::= < modifier > type_specifier METHOD_SET "("
[parameter_list] ")"
statement_block

modifier ::= "public" | "private" | "protected" | "static" | "final"

type_specifier ::= "boolean" | "byte" | "char" | "short" | "int" |
"float" | "long" | "double" | CLASS_SET

class_name ::= CLASS_SET

constructor_declaration ::= < modifier > CLASS_SET "("  [
parameter_list ]  ")" statement_block

(…)

expression ::= numeric_expression | testing_expression
            | logical_expression | string_expression
            | bit_expression | casting_expression
            | creating_expression | literal_expression
            | "null" | "super" | "this" | identifier
            | ("(" expression ")")
            | (CLASS_SET "." METHOD_SET) | (METHOD_SET)

(…)

identifier ::= "a..z,$,_" <"a..z,$,_,0..9,unicode
               character over 00C0"> | CLASS_SET
               | METHOD_SET
```

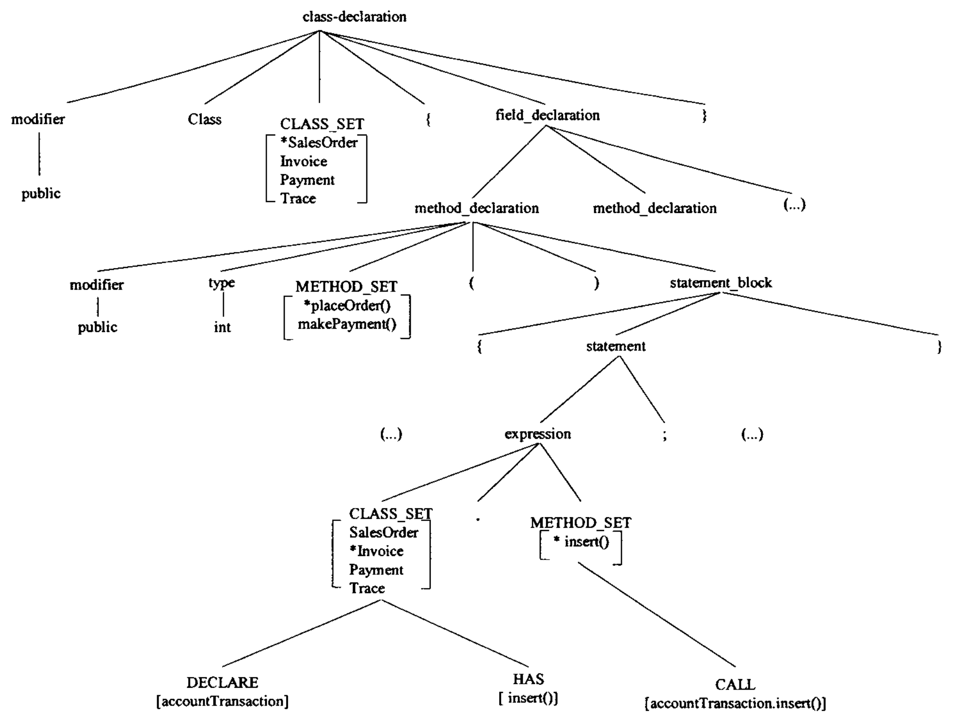
Steps for building a parse tree with using two-level grammar are defined as:

- Start building the parse tree from start symbol of the grammar class_declaration.

- Continue building parse tree with top-down method with extending Non-terminal symbols to the next level of the tree
- If the Non-terminal is a meta-grammar, continue parsing by using meta-variable definition that is coming from production rules (semantics of object-oriented design). for example when we encounter a `CLASS-SET` meta-variable that defines the classes defined in our design, we mark the main class for the parse tree, in this example the main class for the parse tree is `SalesOrder` that we mark it with a *.

At `METHOD_SET` meta-variable node we mark the method that we suppose to build its parse tree and then we continue parsing by using production rules related to the marked method. For example in `Invoice.insert()` node of the parse tree we extend the tree the following by production rule. `<Invoice.Insert()>::=[call]<AccountTransaction.Insert()>.`

Figure 38 illustrates pares tree corresponding to the two-level grammar:



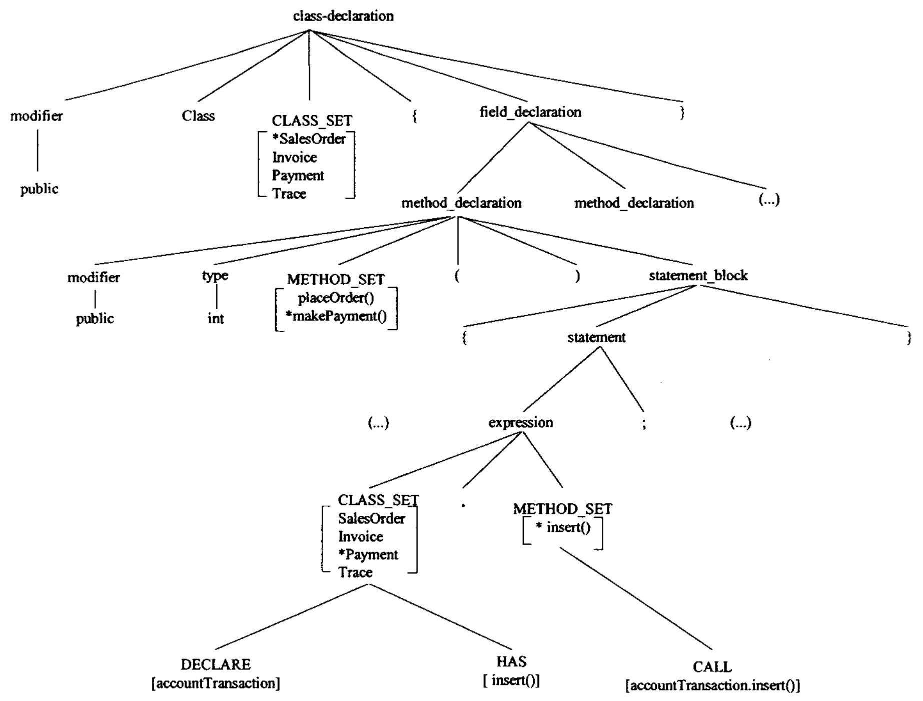


Figure 38. Parse tree for class SalesOrder method with using two-level grammar

This pars tree illustrates the `accountingTransaction.insert()` method is crosscutting `invoice.insert()` and `payment.insert()` methods:

```
<SalesOrder.placeOrder()>::=[call]<Invoice.insert()>

::=[call]<accountTransaction.insert()>

<SalesOrder.makePayment()>::=[call]<Payment.insert()>

::=[call]<accountTransaction.insert()>
```

9.10. Using dynamic syntax grammars for identifying vertical concerns

We modify `placeOrder()` and `makePayment()` methods such that we add tracing mechanism to these methods (Listing 10):

Listing 10. `SalesOrder` class with tracing mechanism

```
Public class SalesOrder
{
      String salesOrderId;
      String orderDate;
      String quantity;
      int unitPrice;
      Invoice invoice;
      Payment payment;
      Trace  trace;

      //...

      public int placeOrder()
      {
            trace.traceEntry();

            int errorCode = 0;
            //...
            invoice.insert();
            //...
            trace.traceExit();
            return errorCode;
      }

      public int makePayment()
      {
            trace.traceEntry();
            int errorCode = 0;
            //...
            payment.insert();
            //...
            trace.traceExit();
            return errorCode;
      }
}
```

Figure 39 illustrates the parse tree for these two methods by using two-level grammar:

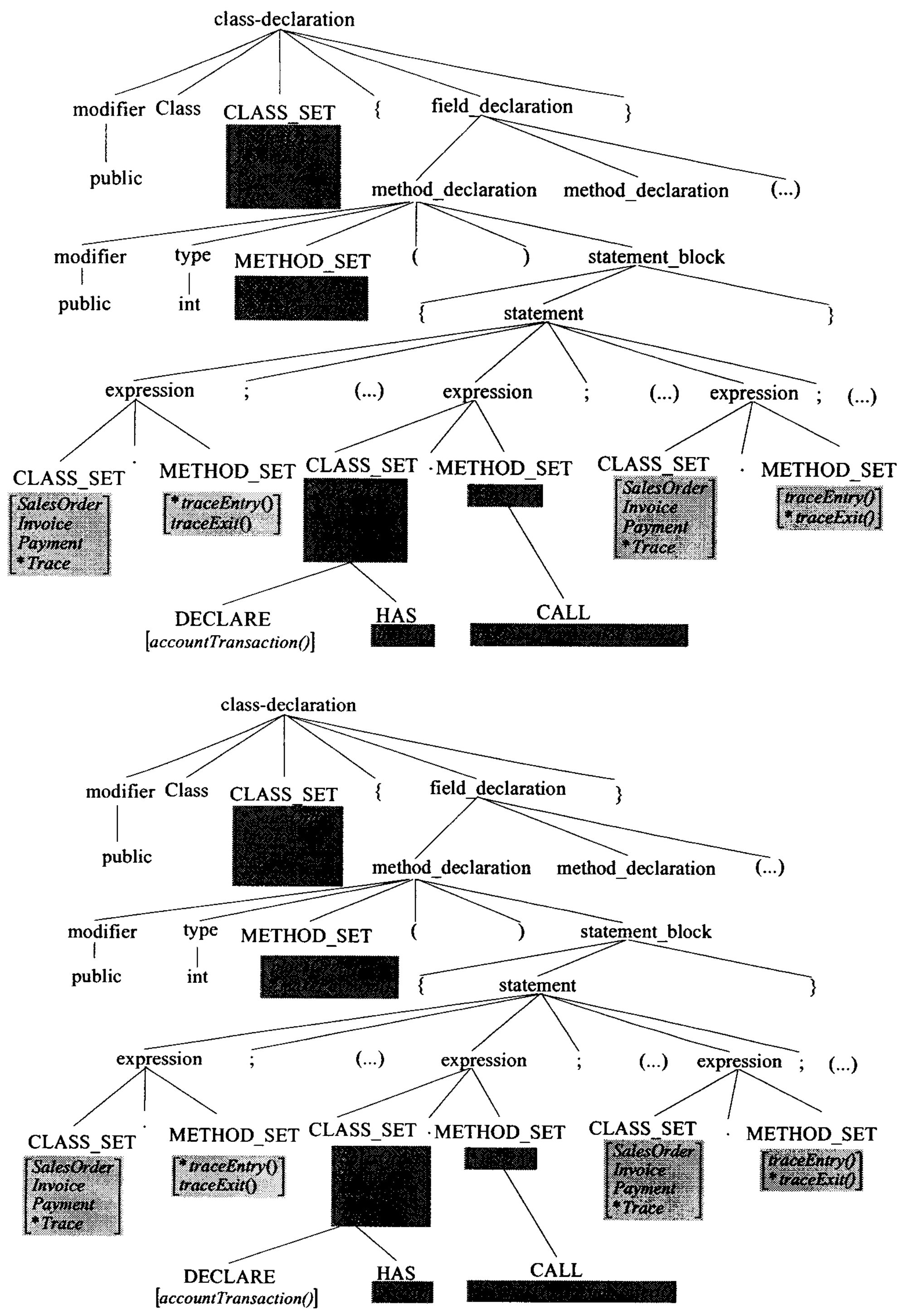


Figure 39. Parse tree for SalesOrder methods illustrates that there is a vertical concern defined by traceEntry() and traceExit() methods.

10. Conclusion, Limitations, and Future work

Software design is an important activity in the development lifecycle but its benefits are sometimes not realized. Scattering and tangling of crosscutting behavior with other elements cause problems with comprehensibility, traceability and reusability. A single set of requirements may be implemented by a number of different object-oriented design decisions and artifacts. Some object-oriented design decisions decompose the problem domain into better modular units such that the crosscutting phenomenon is minimized. However, the phenomenon may not be totally eliminated from an object-oriented model. Achieving a well modularized design in a complex software system involves, among a number of activities, the identification and modeling of crosscutting concerns (aspects). By analyzing the semantics behind any object-oriented design with any degree of modularity we can restructure object-oriented design into a corresponding aspect-oriented design.

Several proposals in the literature are addressing this problem in the programming domain but the problem has not been addressed effectively at earlier stages in the lifecycle. This text discusses an approach to addressing this problem throughout software life cycle activities with restructuring object-oriented artifacts into aspect oriented artifacts. There is no single rule or algorithm to identify all aspects in object-oriented software life cycle. Since aspects are associated with different types (behavioral, business, vertical, and horizontal), different mining rules and algorithms are required to identify each type.

This text outlined how to define a concern dependency graph and how to analyze it by using a set of production rules that can be used to identify aspect candidates. We have proposed a categorization of aspects into behavioral and business types. To identify the former we need to look for a composition pattern, and to identify the later we need to look into dependency graphs, production rules and the transition matrix. As a result, if no cycle exists in the concerns dependency graph we will have crosscutting concerns and we can define an aspect. Furthermore, if there is a cycle in concern dependency graph but concerns are state-preserving then the target concern can be considered as an aspect. In some situations the receiver concern may not be state-preserving, but we still may have crosscutting. To identify this type of crosscutting concerns we use a set of production rules and parsing with two-level grammars. By analyzing the semantics of a crosscutting object-oriented design with any degree of modularity we can restructure it into a corresponding aspect-oriented context as a result:

- No single rule or algorithm to identify all types of aspects.
- With **dependency graph** we identify objects with an independent role in scenarios.
- With representing object-oriented artifacts by **production rules** we are able to identify aspect by detecting clones and crosscutting patterns.
- With using **two-level grammar** we inject semantics of object-oriented design to the syntax of object-oriented language, and then by analyzing implementation artifacts during parsing we are able to detect aspect candidates.

Limitations: In this research we analyzed the behavior of some of the 'Gang of Four' (GoF) design patterns such as observer design pattern and visitor design pattern for identifying crosscutting design patterns. We believe that there are other design patterns that can be modeled as an aspect. The definition of production rules for representing object-oriented concepts can be extended to cover more semantics defined at design level such as defining constraints in terms of invariants, post-conditions, and preconditions.

Future work: In our opinion, In fact, there is a significant amount of work to do in the direction addressed by this research. As a further research and study we suggest the following areas:

- Define methods for detecting well-defined aspects in ambiguous situations.
- Validating restructured aspect-oriented context such that it maintains the same level of functionality and semantics of the object-oriented model.
- Define metrics and measurements that evaluate the process of proposed restructuring from object-oriented to aspect-oriented context and compare the original object-oriented design and restructured aspect-oriented context.
- Using data mining techniques for identifying aspects.
- Using natural language processing algorithms for detecting well-defined aspects.
- As a further development for this proposal, we would like to automate the activities described here and build tool that will use semantics of UML artifacts and also an object-oriented source code and identify aspect candidates.

11. References

1. S. Breu and J. Krinke. Aspect mining using event traces, Proceedings of the 19th International Conference on Automated Software Engineering, (ASE 2004), Linz, Austria, September 20 - 24, 2004, pp. 310-315.
2. J. Krinke and S. Breu, Control-flow-graph-based aspect mining, Proceedings of the First Workshop on Aspect Reverse Engineering at Working Conference on Reverse Engineering (WCRE 2004), Delft, November 9, 2004.

3. M. P. Robillard and G. C. Murphy, Concern graphs: Finding and describing concerns using structural program dependencies, Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), May 19 - 25, Orlando, Florida, 2002, pp. 406-416.
4. M. P. Robillard and G. C. Murphy, Analyzing concerns using class member dependencies, Workshop on Advanced Separation of Concerns in Software Engineering at the International Conference on Software Engineering (ICSE 2001), Toronto, Canada, May 15, 2001.
5. G. Kiczales, J. Lamping , A. Mendhekar, C. Maeda, C. V. Lopes, J-M. Loingtier, and J. Irving, Aspect-oriented programming, Proceedings of the 11th European Conference on Object-Oriented Programming, Jyväskylä, Finland, June 9-13, 1997, pp. 220-242.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: Elements of reusable object-oriented software, Addison Wesley, 1995.
7. M. Nelson, A survey of reverse engineering and program comprehension, 1996. Cornell University Library e-Print Archive No. CS-0503068.
8. C. Constantinides and T. Skotiniotis, Providing multidimensional decomposition in object-oriented analysis and design, The IASTED International Conference on Software Engineering (SE 2004), Innsbruck, Austria, February 17-19, 2004.
9. S. Clarke and R. J. Walker, Composition patterns: An approach to designing reusable aspects, Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), May 12–19, 2001, Toronto, Canada, pp. 5-14.

10. Amir Abdollahi Foumani and Constantinos Constantinides, Aspect-Oriented Reverse Engineering, Proceedings of the 9th World Multiconference on Systemics, Cybernetics and Informatics (WMSCI 2005), Orlando, Florida, USA, July 10-13, 2005.

11. Gilbert K. Krulee, Computer processing of natural language, Prentice-Hall, 1991

12. Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, Lorraine Bier, Clone Detection Using Abstract Syntax Trees, the Proceedings of ICSM'98, November 16-19, 1998 in Bethesda, Mayland.

13. Mohamad Kassab, Constantinos Constantinides, Olga Ormandjieva, Specifying and separating concerns from requirements to design: a case study, The IASTED International Conference on Software Engineering (ACIT-SE 2005), Novosibirsk, Russia, June 20-24, 2005.

14. Barrett R. Bryant, Two-Level Grammar as an Object-Oriented Requirements Specification Language, Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9 - Volume 9, Page: 280, Year of Publication: 2002, ISBN:0-7695-1435-9

15. Amir Abdollahi Foumani and Constantinos Constantinides, Reengineering Object-oriented designs by analyzing dependency graphs and production rules, Proceedings of the Ninth IASTED International Conference on Software Engineering and Applications, Phoenix, Arizona, USA – November 14-16, 2005

16. P. Tarr, H. Ossher, W. Harrison, S. M. Sutton Jr., N degree of separation: Multi-dimensional separation of concerns, Proceedings of the 21st International

Conference on Software Engineering, Los Angeles, CA, USA, May 16-22, 1999, pp. 107-119.

17. T. Elrad, R. E. Filman, A. Bader, Aspect-Oriented Programming – introduction. Communications of the ACM 44(10):29-32(2001).
18. R.E. Filman, T.Elrad, S. Clarke, and M.Aktis, Aspect-oriented Software Development, Addison Wesley Professional (October 6, 2004).
19. C. Larman, Applying UML and Patterns; An introduction to object-oriented analysis and design and the Unified Process; Second edition, (Upper Saddle River, NJ: Prentice Hall Inc. 2002).
20. S. N. Sutton Jr., Early-stage concern modeling, Proceedings of the AOSD 2002 Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, Ensschede, The Netherlands, April 22, 2002.
21. Siobh´an Clarke, Robert J. Walker, Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J, Technical Report TCDCS200115 and UBCCSTR200105.