Aspect Oriented Programming

2013-2014

Course 1

Objectives

- Introduction to Aspect Oriented Paradigm (AOP)
  - AspectJ
  - Spring AOP
  - other ...
- Development of software systems using aspect oriented programming

Bibliography

- AspectJ Project homepage:
  http://www.eclipse.org/aspectj/
- Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004
- Ramnivas Laddad. *AspectJ in Action. Enterprise AOP With Spring Applications*, Second Edition, Manning Publications, 2009.
- Ramnivas Laddad. *AspectJ in Action. Practical Aspect-Oriented Programming*, Manning Publications, 2003.
- Craig Walls, *Spring in Action*, Third Edition, Ed. Manning, 2011

Final Mark

- Lab activity: 50%
- Practical exam or project : 30%
- AOP Report 20%

Course 1 Outline

- What is AOP?
- Crosscutting concerns
- AOP Concepts
- A simple example
- Tools for AOP

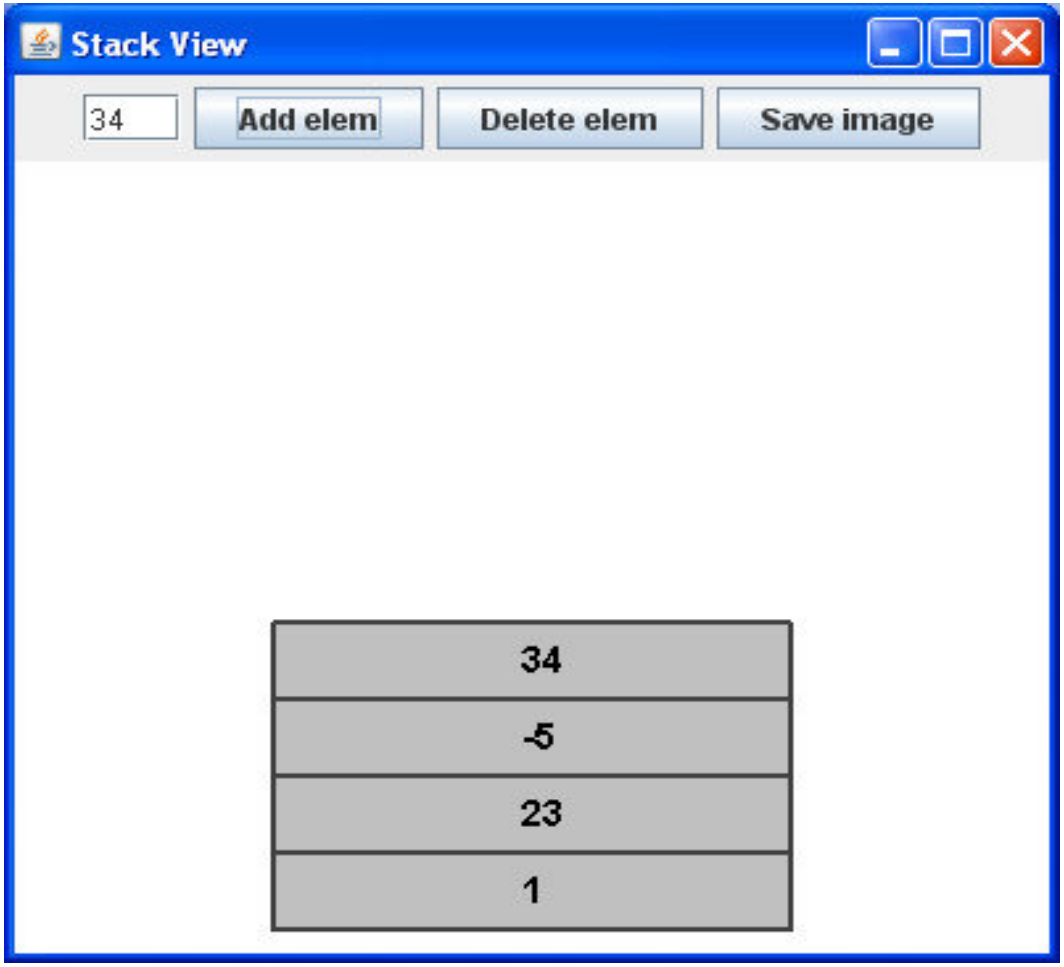
What is AOP?

- Motivation
  - Increase complexity of software systems
  - Separation of concerns (David Parnas, 1979)
  - Modularization
  - Different type of concerns:
    - *core concerns*: business functionalities, data access, presentation logic. Eg. for banking system: customer management, account management, and loan management.
      - Object Oriented Programming
    - *system-wide concerns (crosscutting concerns)*: security, logging, caching, performance monitoring, transaction management, etc.
      - Aspect Oriented Programming

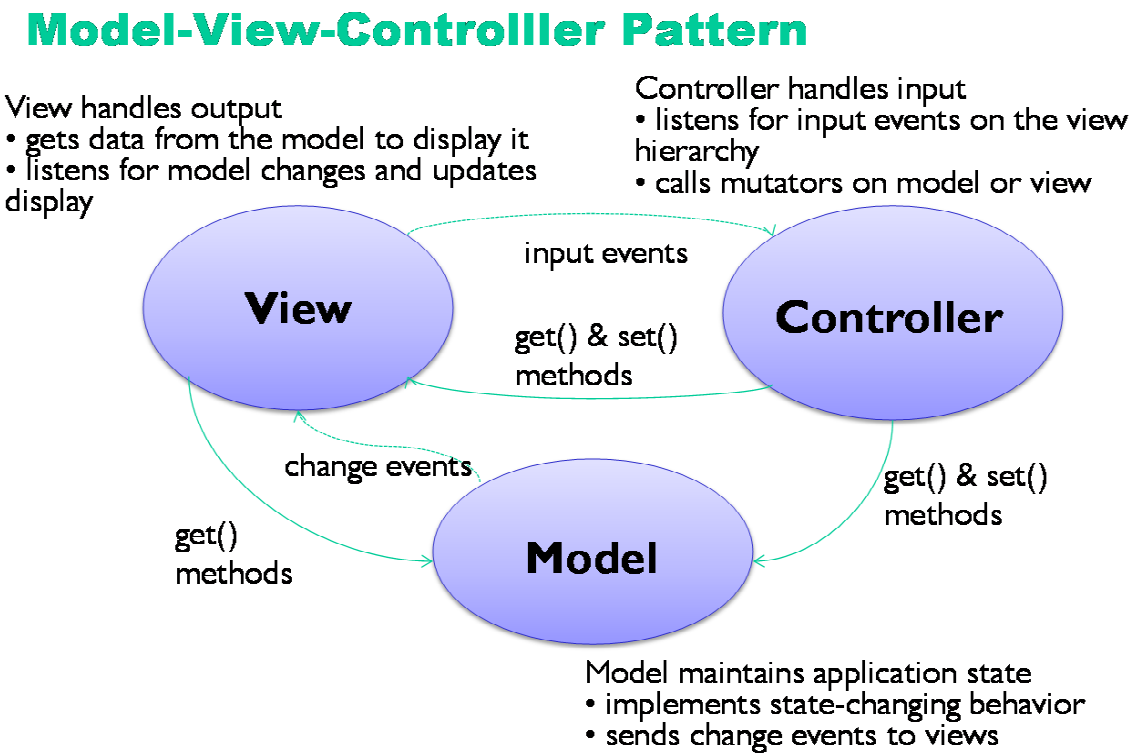
Separation of concerns

- A conceptual separation exists between multiple concerns at design time, but implementation tangles them together.
- Implementation without using AOP breaks:
  - The *Single Responsibility Principle* (SRP): a class should be responsible for only one feature.
  - When implementing a crosscutting concern (without AOP) a single class is responsible for implementing multiple concerns (a core concern and some crosscutting concerns).
  - The *Open/Close Principle* (OCP)—open for extension, but closed for modifications (a class once written should not be modified in ways that affect the clients, only extended).
  - The overall consequence is a higher cost of implementing features and fixing bugs.

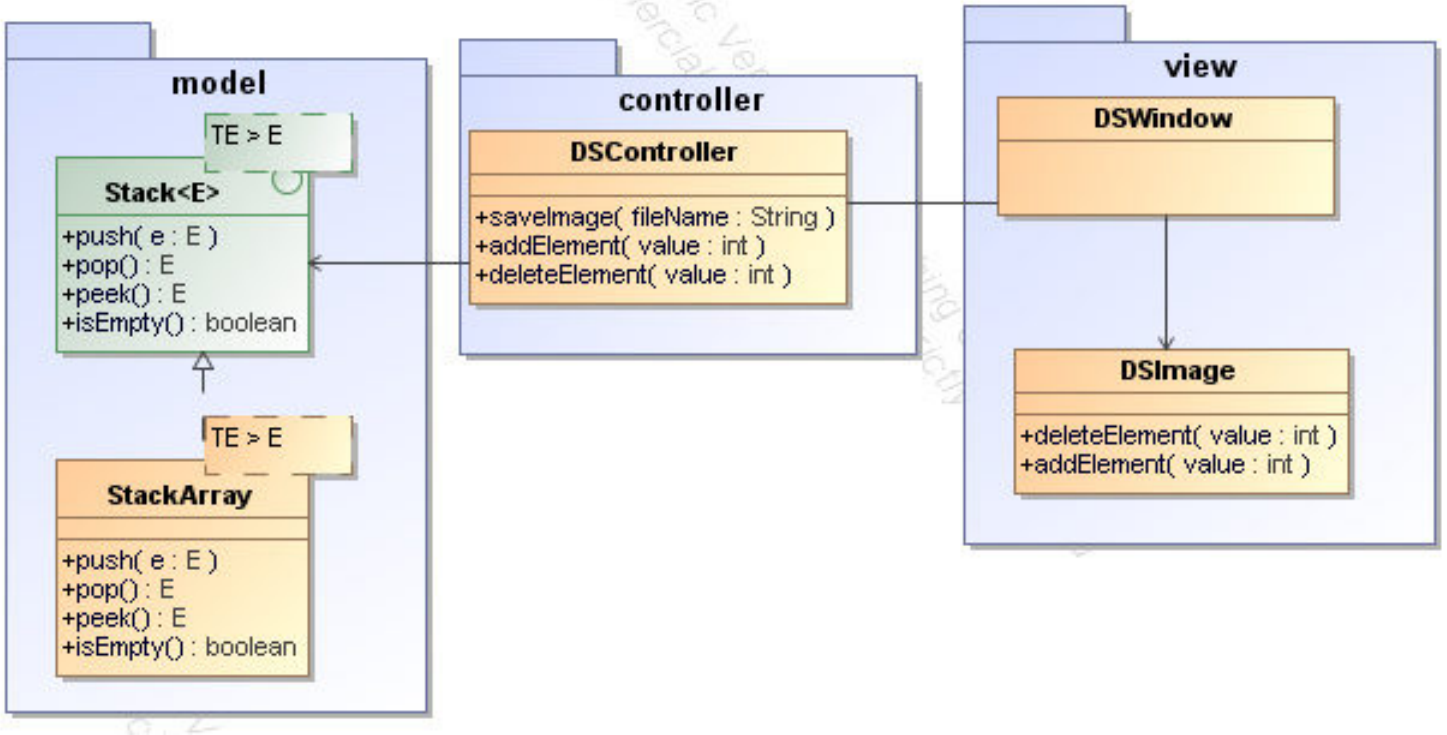
Example – DSView App



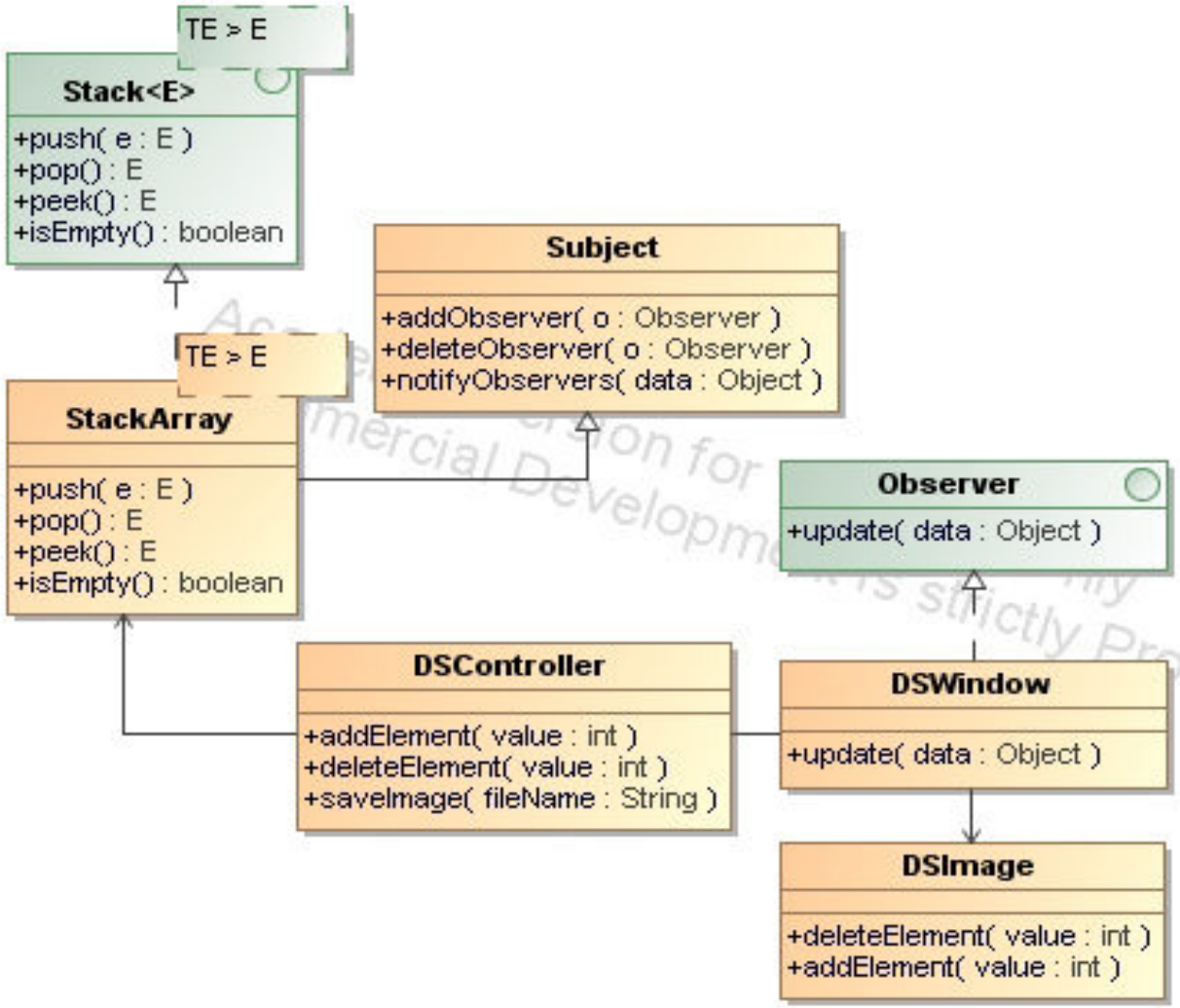
MVC Pattern


Model-View-Controlller Pattern
View handles output
• gets data from the model to display it
• listens for model changes and updates display
Controller handles input
• listens for input events on the view hierarchy
• calls mutators on model or view
View
Controller
Model
input events
get() & set() methods
change events
get() methods
get() & set() methods
Model maintains application state
• implements state-changing behavior
• sends change events to views

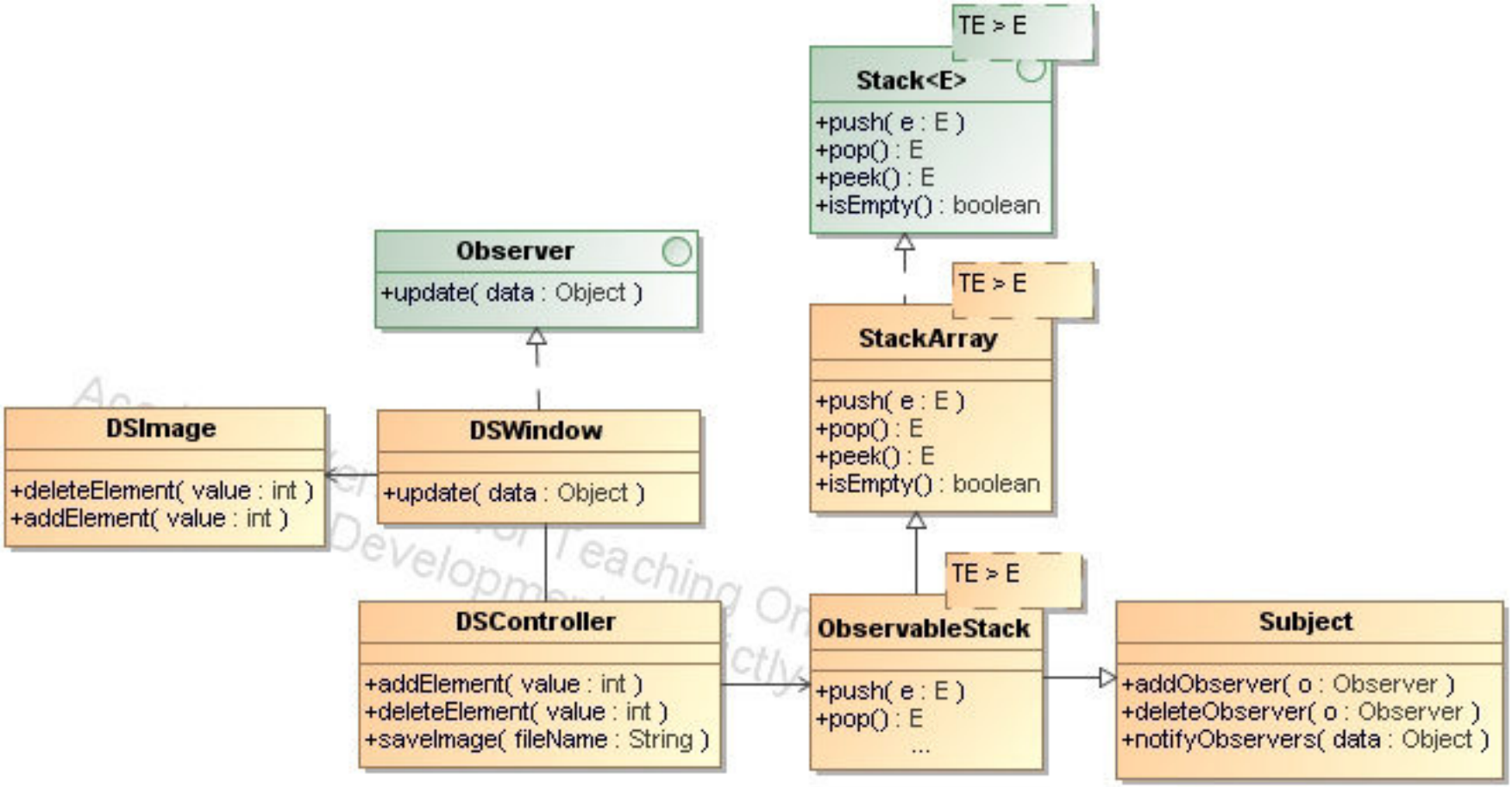
DSView App - MVC



DSApp Architecture – solution 1



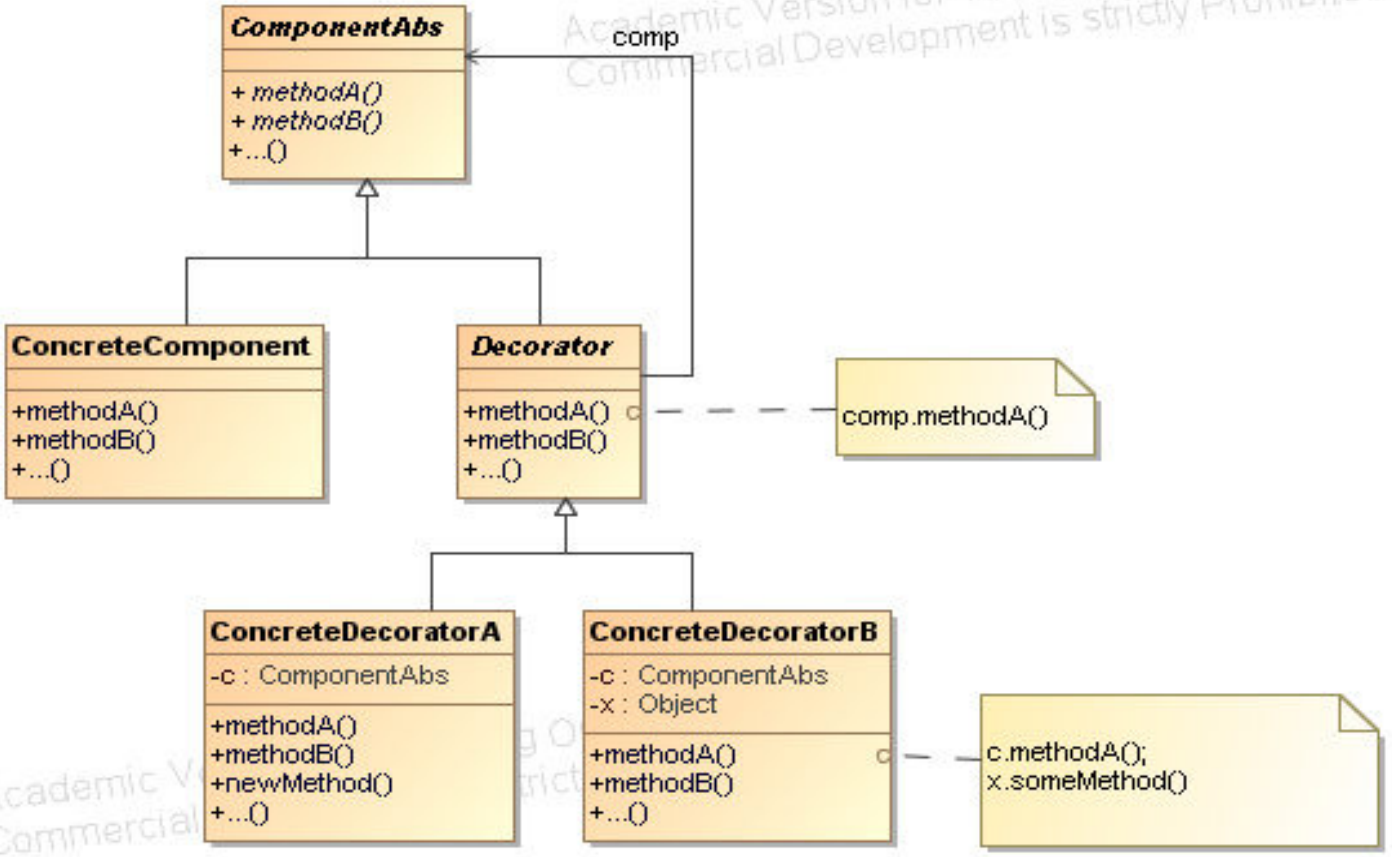
DSApp Architecture – solution 2



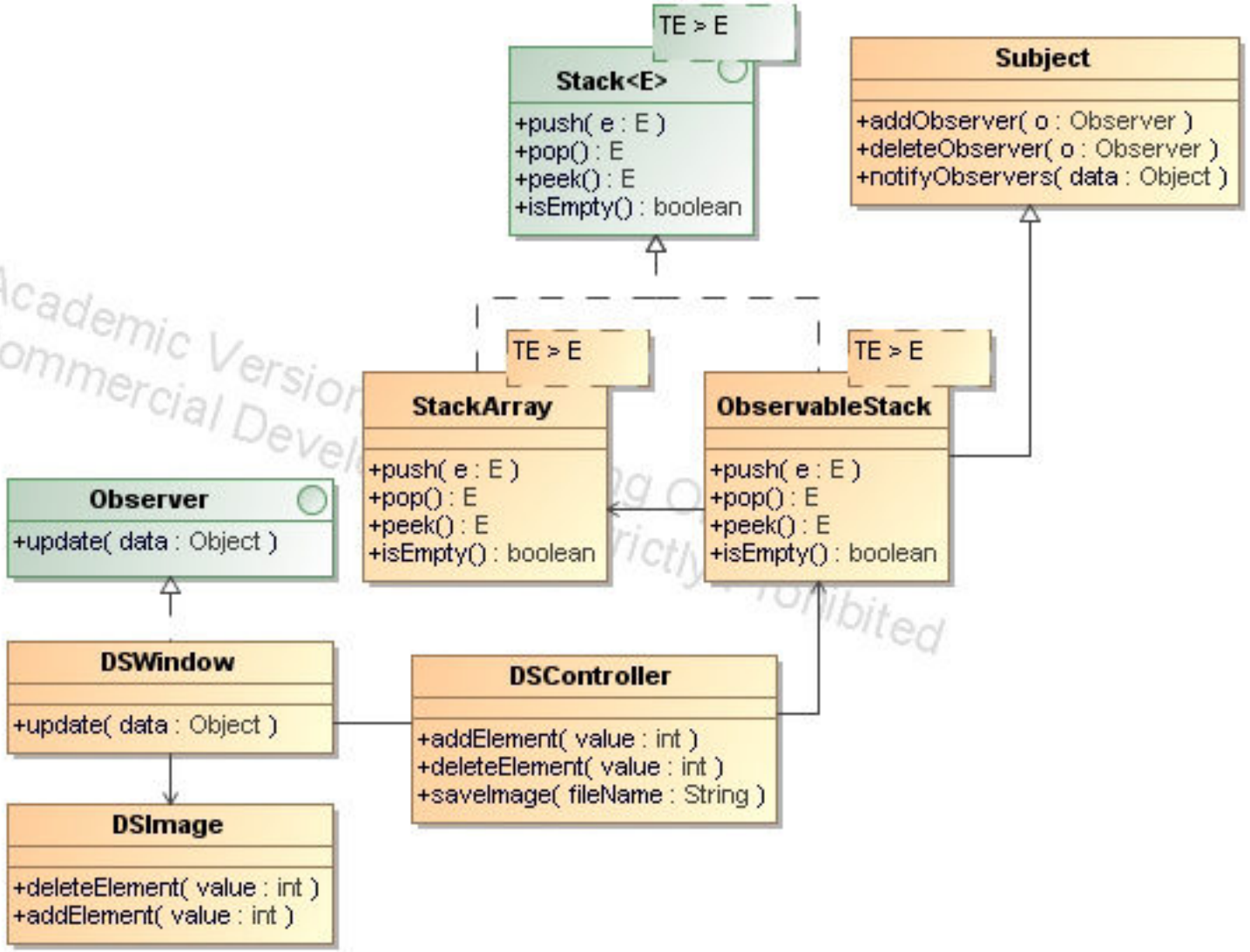
Decorator Pattern

- Attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.
- Applicability:
  - to add responsibilities to individual objects dynamically and transparently, without affecting other objects.
  - for responsibilities that can be withdrawn.
  - when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.

Decorator Pattern



DSApp Architecture – solution 3



Common Crosscutting Concerns

- Logging
- Performance monitoring
- Transaction management
- Caching
- Design patterns (Observer, ...)
- Security

Modularization without AOP

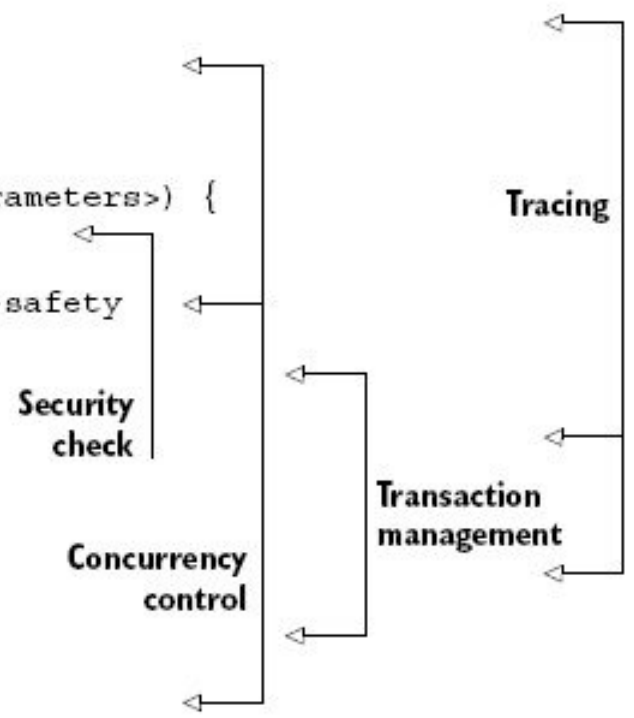
- With conventional implementations, core and crosscutting concerns are tangled in each module.
- Each crosscutting concern is scattered in many modules.
- The presence of code tangling and code scattering are symptoms of the conventional implementation of crosscutting concerns.

Code tangling & scattering

- *Code tangling* is caused when a module is implemented to handle multiple concerns simultaneously.
  - Developers often consider concerns such as business logic, performance, synchronization, logging, security, etc. when implementing a module.
  - This leads to the simultaneous presence of elements from each concern's implementation and results in code tangling.
- *Code scattering* is caused when a single functionality is implemented in multiple modules. Crosscutting concerns, by definition, are spread over many modules, related implementations are also scattered over all those modules.

Modularization without AOP

```
public class SomeBusinessClass extends OtherBusinessClass {
    ... Core data members
    ... Log stream
    ... Concurrency control lock

    ... Override methods in the base class

    public void someOperation1(<operation parameters>) {
        ... Ensure authorization

        ... Lock the object to ensure thread-safety

        ... Start transaction

        ... Log the start of operation

        ... Perform the core operation

        ... Log the completion of operation

        ... Commit or rollback transaction

        ... Unlock the object
    }

    ... More operations similar to above addressing multiple concerns

}
```

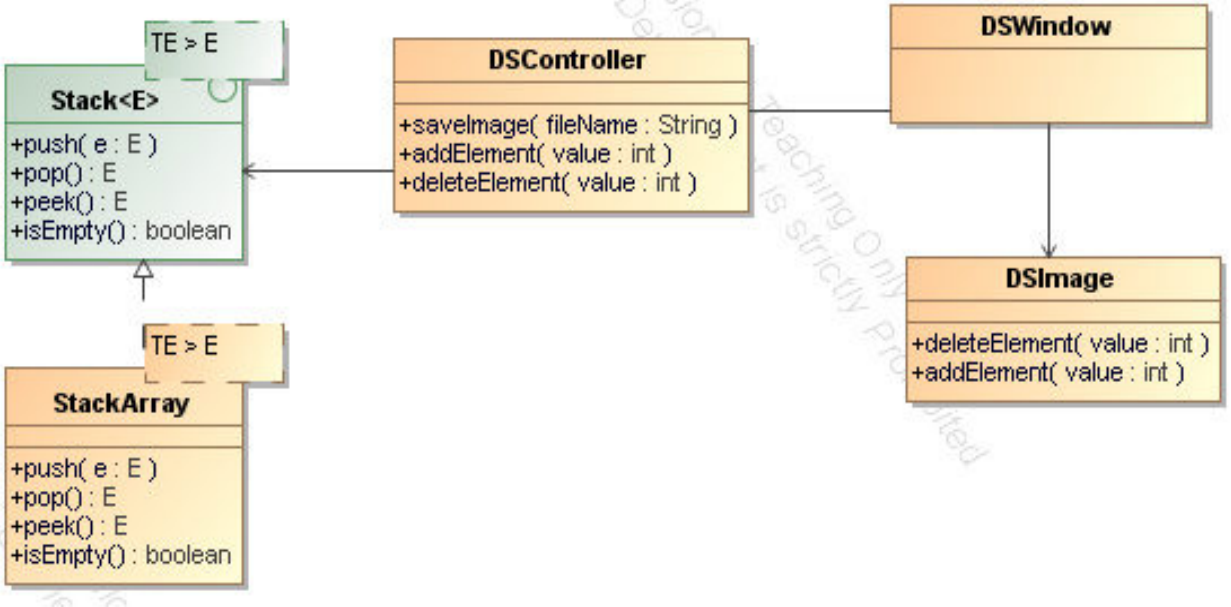


Aspect Oriented Programming

- AOP is a methodology that provides separation of crosscutting concerns by introducing a new unit of modularization, called *aspect*.
- Each aspect focuses on a single specific crosscutting functionality.
- The core classes do not contain anymore code from crosscutting concerns.
- A special tool, called *aspect weaver,* composes the final system by combining the core classes and crosscutting aspects through a process called *weaving*.
- AOP helps to create/develop applications that are easier to design, implement, and maintain.

Benefits of AOP

- Simplified design
- Cleaner implementation
- Better code reuse

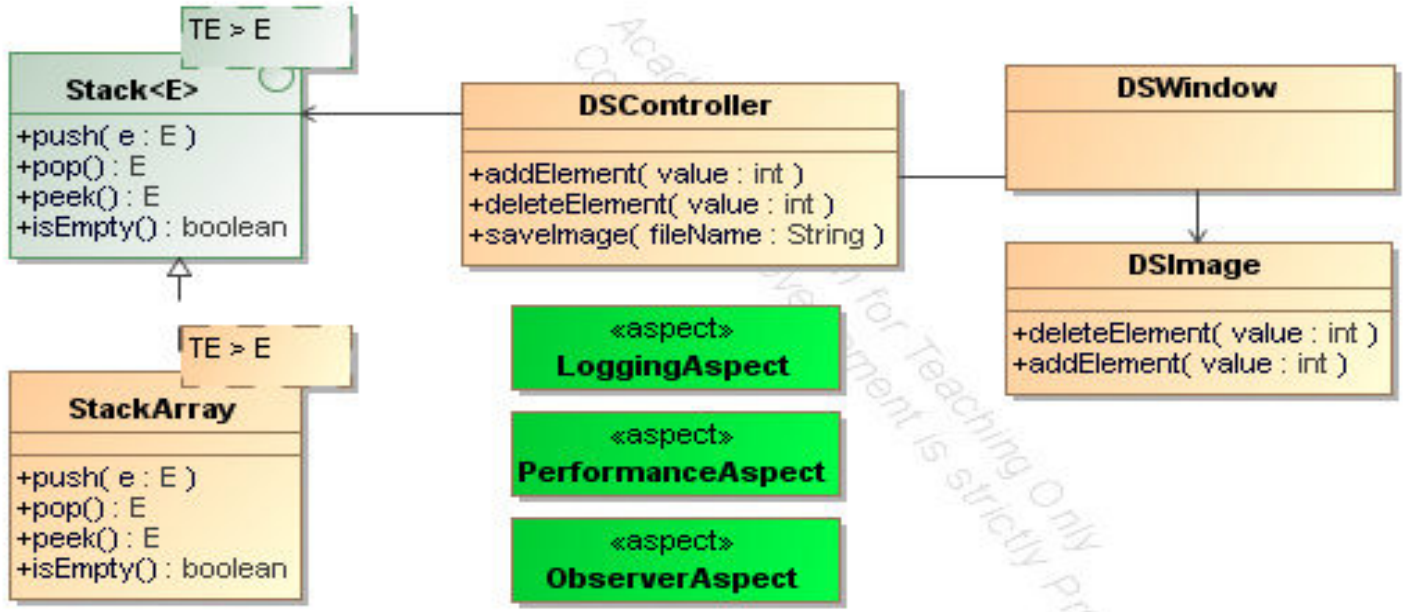


Costs of AOP

- Tools not mature enough
  - (AJDT for Eclipse, IntelliJ Idea plugin)
- Need for greater skills
- Complex program flow

Modularizing with AOP

- AOP keeps the independence of the individual concerns (from design). Implementations (classes) can be easily mapped back to the corresponding concerns, resulting in a system that is simpler to understand, easier to implement, and more adaptable to changes.



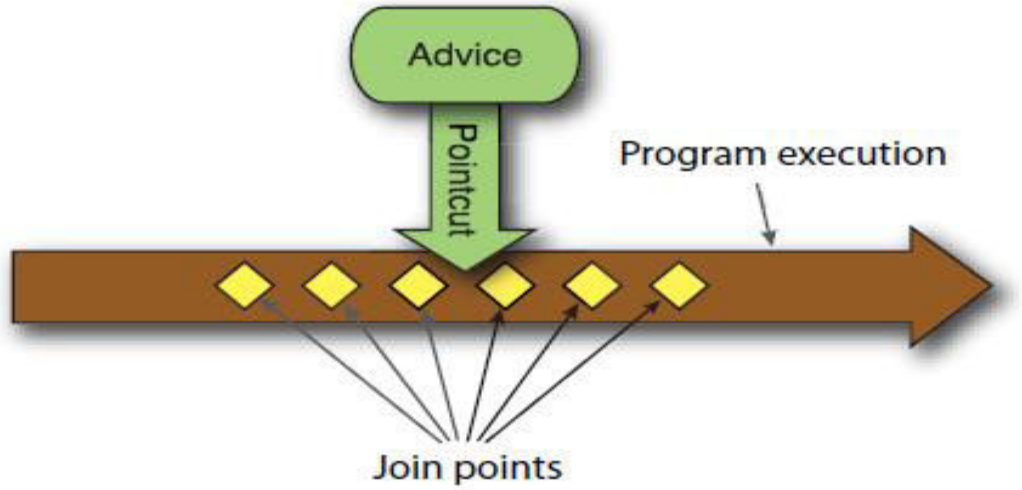
Alternatives to AOP

- The design and implementation of crosscutting concerns is not a new problem.
- It has appeared when complex software systems were starting to be developed.
- Other technologies that deal with the same problem:
  - frameworks
  - code generation
  - design patterns : observer pattern, chain of responsibility, decorator and proxy, interceptor
  - dynamic languages.

Fundamental concepts in AOP

- ***Join points*** (Identifiable points in the execution of the system):
  - The system exposes different points during its execution (execution of methods, creation of objects, etc). Such identifiable points in the system are called ***join points***.
- ***Pointcuts*** (A construct for selecting join points):
  - The pointcut construct selects any join point that satisfies some criteria. Pointcuts also collect context at the selected points (i.e. method arguments).
- ***Advice*** *(A construct to alter program behavior):*
  - The join points are augmented with additional or alternative behavior. An advice can add behavior *before*, *after*, or *around* the selected join points.
  - Advice is a form of *dynamic crosscutting* because it affects the execution of the system.

Fundamental concepts in AOP



Fundamental concepts in AOP

- ***Introductions (Inter-type declarations)*** *(Constructs to alter static structure of the system).*
  - In order to implement a crosscutting functionality effectively, the static structure of the system must be changed (add new atributes/methods to existing classes).
  - These mechanims are referred to as *static crosscutting*.
  - There are also situations when certain conditions must be detected (e.g. the existence of particular join points), before the execution of the system (at compile-time). Weave-time declaration constructs allow such possibilities. (AspectJ)

Fundamental concepts in AOP

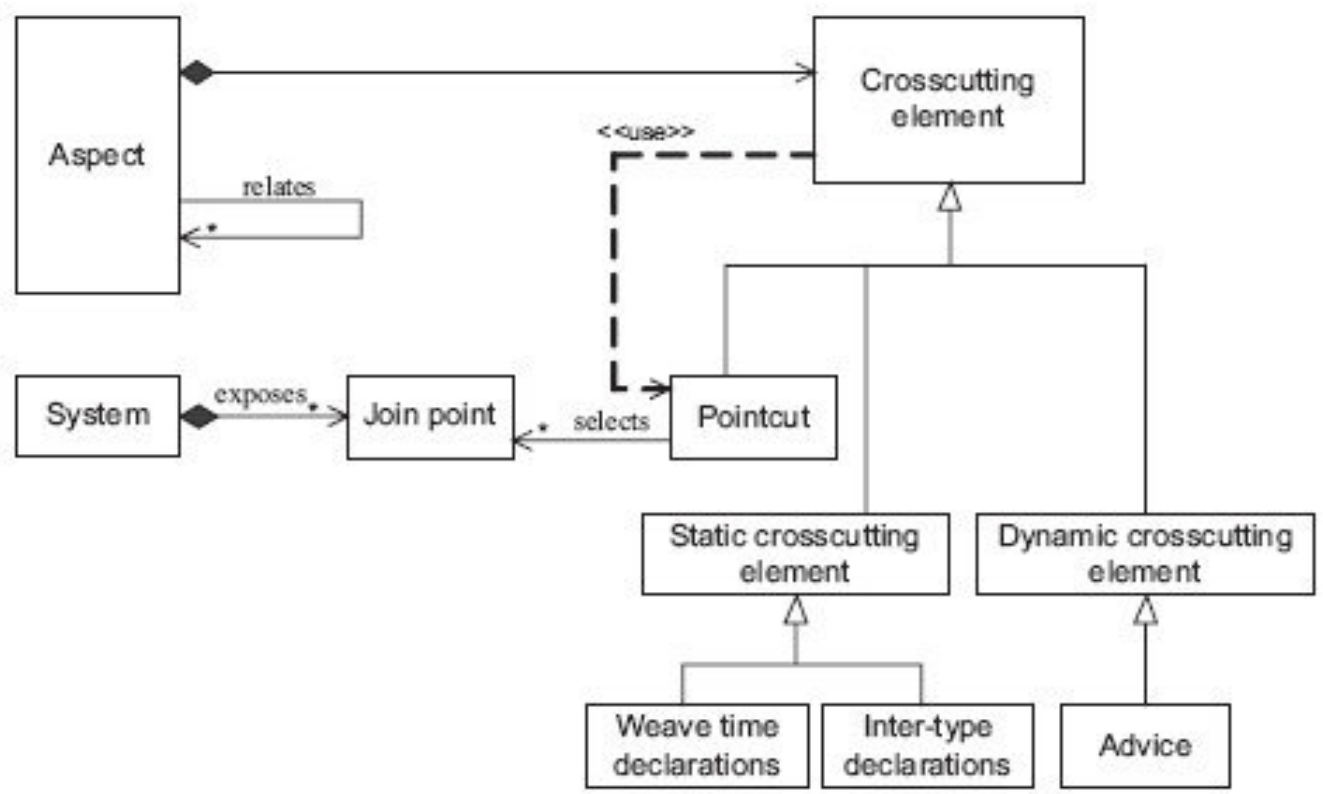
- ***Aspect*** *(A module to express all crosscutting constructs)*. The aspect construct provides a module that contains the crosscutting logic.
  - The aspect contains pointcuts, advice, and static crosscutting constructs.
  - It may be related to other aspects.

  Aspects become a part of the system and use the system.

Remark:

A software system cannot be developed using only aspects. They are only aprox. 15% of the final system.

AOP Model


Aspect
relates
*
Crosscutting element
<<use>>
System
exposes
*
Join point
*
selects
Pointcut
Static crosscutting element
Dynamic crosscutting element
Weave time declarations
Inter-type declarations
Advice

Anatomy of an AOP language

- An AOP implementation consists of two parts:
  - The *language specification* that describes the language constructs and syntax to express implementation of the core and crosscutting concerns.
  - The *language implementation* verifies the code's adherence to the language specification and translates the code into an executable form.

The AOP language specification

- Any implementation of AOP must specify a language to implement the individual concerns and a language to implement the weaving rules.
  - Implementation of concerns: the concerns of a system are implemented into modules that contain the data and behavior needed to provide their services. To implement the business logic of each concern, standard languages (Java, C++, and C#) are normally used.
  - Weaving rules specification: weaving rules specify how to combine the implemented concerns in order to form the final system.
- The strength of AOP comes from the economical way of expressing the weaving rules.
- Weaving rules can be general or specific in the ways they interact with the core modules.

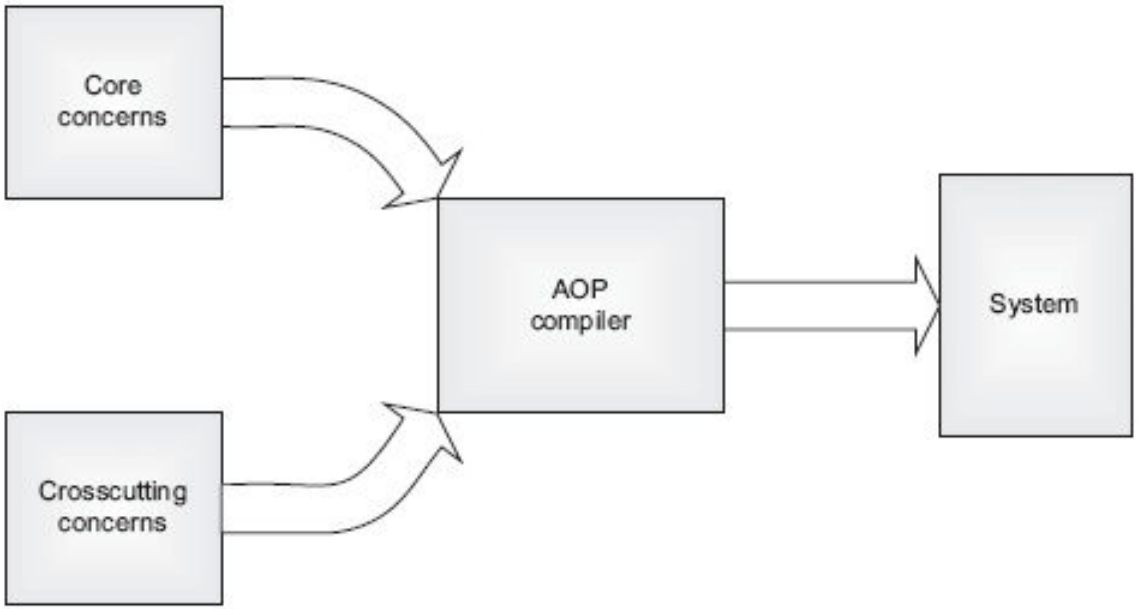
The AOP language specification

```
public class SomeBusinessClass extends OtherBusinessClass {
    ... Core data members

    ... Override methods in the base class

    public void someOperation1(<operation parameters>) {
        ... Perform the core operation
    }

    ... More operations similar to above
}
```

The AOP language implementation

- The AOP language implementation performs two logical steps:
  - It first combines the individual concerns using the weaving rules.
  - It converts the resulting information into executable code.
- AOP implementation requires the use of a processor, called *weaver,* to perform these steps.
- An AOP system can implement the weaver in various ways:
  - *Source-to-source translation.*
  - *Byte-code modification.*
  - *Class-loading weaving.*
  - *Dynamic proxies.*

The AOP language implementation



AOP Languages

- Java: AspectJ (AspectWerkz), Spring AOP, JBoss AOP, etc.
- C++: AspectC++
- C#: Aspect#, PostSharp
- Ruby: Aquarium
- etc.

Tools for Java

- AspectJ homepage

  http://eclipse.org/aspectj
- AspectJ Development Tools (AJDT) plug-in
  - Eclipse
  - http://www.eclipse.org/ajdt
  - Contains also AspectJ files
- AspectJ Plugin
  - http://intellij.expertsystems.se/aspectj.html
  - IntelliJ IDEA version >9.0
  - only annotated style

Installing AJDT

- Without Internet connection:

1. Obtain the ajdt archives corresponding to your Eclipse version.

   e.g. ajdt_2.2.2_for_eclipse_3.7.zip

2. From Eclipse choose Help -> Install new software ...
3. Browse to the downloaded zip file.

Installing AJDT

- With Internet connection:

1. Go to the AJDT download page and search for the URL corresponding to your Eclipse version:

e.g. http://download.eclipse.org/tools/ajdt/37/update

2. From Eclipse choose Help -> Install new software ...

3. Paste the URL ....

Simple Example

```
public class MessageSender {
   public void send(String from, String to, String message){
    System.out.println("From: "+from+" to "+to+" ["+message+"]" );
   }
   public void send(String from, String message){
      System.out.println("From: "+from+" to ALL  ["+message+"]" );
   }
}
public class Test {
   public static void main(String[] args) {
        MessageSender sender=new MessageSender();
        sender.send("Ioana", "Maria", "where are you?");
        sender.send("Ioana", "Who is there?");
   }
}
```

Simple Example

- Compilation

```
javac MessageSender.java Test.java
```

- Execution

```
java Test
```

- Output

```
From: Ioana to Maria [where are you?]
From: Ioana to ALL  [Who is there?]
```

AspectJ Compiler

- Download AspectJ archive from:
  http://www.eclipse.org/aspectj/downloads.php
- Install AspectJ using java –jar archive_name
- Create ASPECTJ_HOME variable

```
set ASPECTJ_HOME=<install directory>
```

- Set the Path variable:

```
set PATH=%PATH%;%ASPECTJ_HOME%\bin
```

- AspectJ compiler: ajc

```
ajc <list of .java and .aj file names>
ajc –argfile file.lst  //file.lst constains the names of the .java
   //and .aj files one on each line
//simple.lst
MessageSender.java
Test.java
SimpleAspect.aj
```

Simple Example

- Compilation using AspectJ compiler

```
ajc MessageSender.java Test.java
```

- Execution

```
java Test
```

- Output

```
From: Ioana to Maria [where are you?]
From: Ioana to ALL  [Who is there?]
```

Simple Example - Aspect

```
public aspect SimpleAspect {
   pointcut sendMessage():execution(public * MessageSender.*(..));
   before(): sendMessage(){
        System.out.println("New message!!!");
   }
}
```

- Compilation using AspectJ compiler

```
ajc MessageSender.java Test.java SimpleAspect.aj
```

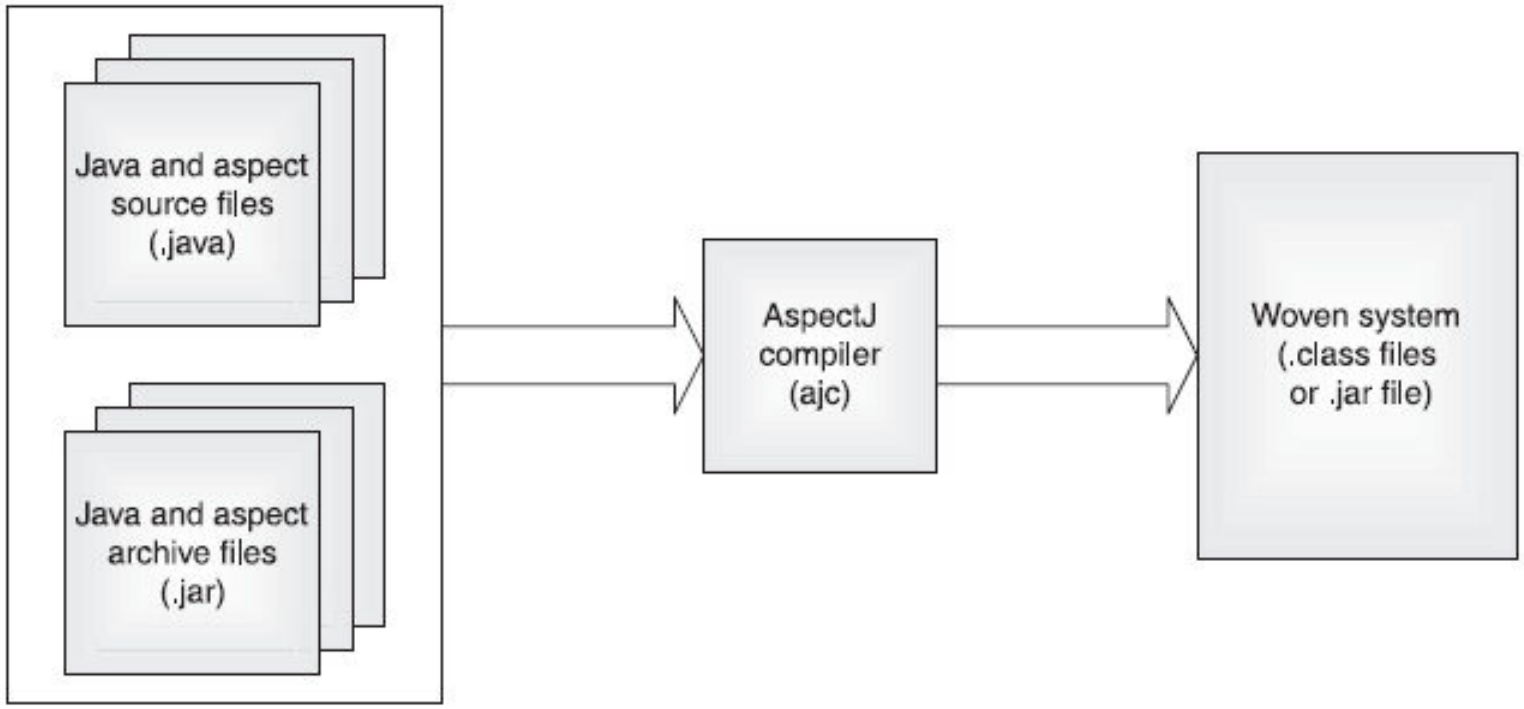
- Execution

```
java Test
```

- Output

```
New message!!!
From: Ioana to Maria [where are you?]
New message!!!
From: Ioana to ALL  [Who is there?]
```

AspectJ Compiler



AspectJ - Details

- *Aspects* are mapped to classes, with each data member and method becoming the members of the class representing the aspect.
- *Advice* is usually mapped to one or more methods. The calls to these methods are then inserted into the join points matching the pointcut specified within the advice. Advice may also be mapped to code that is directly inserted inside an advised join point.
- *Pointcuts* are intermediate elements that instruct how advice is woven and usually are not mapped to any program element, but they may have auxiliary methods to help perform matching at runtime.
- *Introductions* are mapped by making the required modification, such as adding the introduced fields to the target classes.

Simple Example - Transformed

```
public class SimpleAspect {
   public static final SimpleAspect aspectInstance;
   //before(): sendMessage(){
   //   System.out.println("New message!!!");
   //}
   public final void ajc$before$SimpleAspect$1$e248af() {
        System.out.println("New message!!!");
   }
   static {
    SimpleAspect.aspectInstance= new SimpleAspect();
   }
   //other methods ...
}
```

MessageSender - Transformed

```
public class MessageSender {
   public void send(String from, String to, String message){
    SimpleAspect.aspectInstance.ajc$before$SimpleAspect$1$e248af();
     System.out.println("From: "+from+" to "+to+" ["+message+"]" );
    }
   public void send(String from, String message){
    SimpleAspect.aspectInstance.ajc$before$SimpleAspect$1$e248af();
       System.out.println("From: "+from+" to ALL  ["+message+"]" );
    }
}
```

First Lab

- Design and implement a given application (with the Observer pattern)
  - Add logging to the application
  - Use a database for persistent data.
- Trace the execution of each public method using one of the folowing logging/tracing tools:
  - java.util.Logging (included in JSDK)
  - Log4J

Tracing tools

- A tracing tool/library allows the programmer to insert different type of messages into code for various purposes (debugging, further analysis, etc.).
- It usually defines levels for different types of messages: debug, warning, error, information, sever, etc.
- The configuration is often performed using configuration text files (properties files in Java), and can be turned on/off at runtime.

Package java.util.logging

- Provides the classes and interfaces of the platform's core logging facilities.
- Applications send logging calls on a **`Logger`** object. The **`Logger`** object creates **`LogRecord`** objects which are passed to **`Handler`** objects for storage. Both **`Loggers`** and **`Handlers`** usually use logging **`Levels`** and/or **`Filters`** to decide if they are interested in a particular **`LogRecord`**. A **`Handler`** can (optionally) use a **`Formatter`** to set the format of the message before storing.

Package java.util.logging

- Important classes:
  - **`Logger`**: is used to log messages for a specific system or application component.
  - **`Level`**: defines a set of standard logging levels that can be used to control logging output: SEVERE (highest value), WARNING, INFO,CONFIG, FINE, FINER, FINEST (lowest value), and OFF (to turn off logging) and ALL (to log all messages).
  - **`LogRecord`**: contains information about the message to be logged.
  - **`Formatter`**: provides support for formatting **`LogRecords`** (**`SimpleFormatter`**, **`XMLFormatter`**).
  - **`Handler`** object takes log messages from a **`Logger`** and exports them (console, file, network stream, etc).

Logger Methods

- The `Logger` class provides a large set of convenience methods for generating log messages.
- "Generic" Log methods: `logger.log(Level, String, ...)`
- It contains methods for each logging level, named after the logging level name: `logger.warning(...)`, `logger.severe(...)`, `logger.info(...)`, etc.
- There are convenience methods for tracing method entries (the "entering" methods), method returns (the "exiting" methods) and throwing exceptions (the "throwing" methods):
  - `entering(String sourceClass, String sourceMethod, ...)`
  - `exiting(String sourceClass, String sourceMethod,...)`
  - `throwing(String sourceClass, String sourceMethod, Throwable thrown)`

Example JSDK Logging

```
public class ParticipantRepositoryMock implements
   ParticipantRepository{
    private Map<String, Participant> participants;

    public ParticipantRepositoryMock() { ...}

    public void save(Participant p) {...}

    public void update(String codePart, Participant p) {...}

    public List<Participant> getAll() { ...}

    public Participant findById(String code) {...}

    public List<Participant> getByPoints() { ...  }

}
```

Example java.util.logging

```
import java.util.logging.*;
public class ParticipantRepositoryMock implements
   ParticipantRepository{
    private Map<String, Participant> participants;
    private static Logger logger=Logger.getLogger("contest");
    public ParticipantRepositoryMock() {
     logger.info("[Entering:] ParticipantRepositoryMock.init");
     //...
    }
    public void save(Participant p) {
    logger.info("[Entering:] ParticipantRepositoryMock.save");
    ...}
    public void update(String codePart, Participant p) {...}
    public List<Participant> getAll() { ...}
    public Participant findById(String code) {...}
    public List<Participant> getByPoints() { ...  }
}
```

Example JSDK Logging

- Configuration file:

```
//logging.properties
handlers =  java.util.logging.FileHandler
.level = ALL
java.util.logging.FileHandler.level = INFO
java.util.logging.FileHandler.directory = .
java.util.logging.FileHandler.formatter =
   java.util.logging.SimpleFormatter
java.util.logging.FileHandler.append=true
java.util.logging.FileHandler.pattern=contest.out
```

- Execution

```
//You do not have to modify the start of the application
public class StartApp {
    public static void main(String[] args) { ...}
}
//You have to add a virtual machine parameter
java StartApp -Djava.util.logging.config.file=logging.properties
```

Example concurs.out

```
Feb 14, 2012 1:17:03 AM concurs.repository.mock.ParticipantRepositoryMock
   <init> INFO: [Entering]
Feb 14, 2012 1:17:03 AM concurs.repository.mock.ParticipantRepositoryMock
   getAll INFO: [Entering:]
Feb 14, 2012 1:17:04 AM concurs.repository.mock.ParticipantRepositoryMock
   getByPoints INFO: [Entering:]
Feb 14, 2012 1:17:04 AM concurs.repository.mock.ParticipantRepositoryMock
   getByPoints INFO: [Entering:]
Feb 14, 2012 1:17:20 AM concurs.repository.mock.ParticipantRepositoryMock
   findById INFO: [Entering:] 245
Feb 14, 2012 1:17:20 AM concurs.repository.mock.ParticipantRepositoryMock
   update INFO: [Entering:] 245 Ionescu Maria 1 23
Feb 14, 2012 1:17:20 AM concurs.repository.mock.ParticipantRepositoryMock
   getByPoints INFO: [Entering:]
Feb 14, 2012 1:17:20 AM concurs.repository.mock.ParticipantRepositoryMock
   getByPoints INFO: [Entering:]
Feb 14, 2012 1:17:23 AM concurs.repository.mock.ParticipantRepositoryMock
   findById INFO: [Entering:] 345
Feb 14, 2012 1:17:23 AM concurs.repository.mock.ParticipantRepositoryMock
   update INFO: [Entering:] 345 Vasilescu Ioan 1 45
...
```

Log4j

- It is an open source project developed at Apache Software Foundation.
- You have to download the latest version of log4j from

http://logging.apache.org/log4j/1.2/download.html

and add the jar file to the project CLASSPATH.

- Log4j has three main components: *loggers*, *appenders* (for storage) and *layouts* (for formatting).
- Main classes(package `org.apache.log4j`):
  - `Logger`: for reporting messages.
  - `Level`: possible values: TRACE(lowest value), DEBUG, INFO, WARN, ERROR and FATAL(highest value).
  - `PropertyConfigurator`: configuration (what to report, where to store, how to format the messages).
  - etc.

Log4j Logger methods

- Creation & retrieval methods:

```
Logger getRootLogger();
Logger getLogger(String name);
```

- Printing methods:

```
trace(Object message);
debug(Object message);
info(Object message);
warn(Object message);
error(Object message);
fatal(Object message);
```

- Generic printing method:

```
log(Level l, Object message);
```

Example log4j

```
import org.apache.log4j.Logger;
public class ParticipantRepositoryMock implements
   ParticipantRepository{
    private Map<String, Participant> participants;
    private static Logger logger=Logger.getLogger("concurs");
    public ParticipantRepositoryMock() {
     logger.info("[Entering:] ParticipantRepositoryMock.init");
     //...
    }
    public void save(Participant p) {
    logger.info("[Entering:] ParticipantRepositoryMock.save");
    ...}
    public void update(String codePart, Participant p) {...}
    public List<Participant> getAll() { ...}
    public Participant findById(String code) {...}
    public List<Participant> getByPoints() { ...  }
}
```

Example log4j StartApp

- Configuration file

```
//log4j.properties
log4j.rootLogger=, A1
log4j.appender.A1=org.apache.log4j.FileAppender
log4j.appender.A1.File=log4jConcurs.out
log4j.appender.A1.Append=false
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
# Formatting: date, level,logger name and the message
log4j.appender.A1.layout.ConversionPattern=%d %-5p %c{2} - %m%n
```

- Execution

```
//You must modify StartApp to configure the use of log4j
import org.apache.log4j.PropertyConfigurator;
public class StartApp {
    public static void main(String[] args) {
         PropertyConfigurator.configure(args[0]);
       //...
     }}
//java StartApp log4j.properties
```

Example log4jconcurs.out

```
2012-02-13 01:52:01,093 INFO  concurs - [Entering:]
   ParticipantRepositoryMock.init
2012-02-13 01:52:01,109 INFO  concurs - [Entering:]
   ParticipantRepositoryMock.getAll
2012-02-13 01:52:01,125 INFO  concurs - [Entering:]
   ParticipantRepositoryMock.getByPoints
2012-02-13 01:52:01,140 INFO  concurs - [Entering:]
   ParticipantRepositoryMock.getByPoints
2012-02-13 01:52:07,781 INFO  concurs - [Entering:]
   ParticipantRepositoryMock.findById
2012-02-13 01:52:07,781 INFO  concurs - [Entering:]
   ParticipantRepositoryMock.update
2012-02-13 01:52:07,781 INFO  concurs - [Entering:]
   ParticipantRepositoryMock.getByPoints
2012-02-13 01:52:10,359 INFO  concurs - [Entering:]
   ParticipantRepositoryMock.findById
...
```