Aspect Oriented Programming

2018-2019
Course 4

Course 4 Contents

- AspectJ Language:
  - Annotations and AspectJ
  - Static crosscutting
  - Examples

Annotations -Recap

- From version 1.5
- Annotations provide data about a program code (class, method, package, etc) that is not part of the program itself.
- They have no direct effect on the operation of the code they annotate.
- Usages:
  - *Information for the compiler* — Annotations can be used by the compiler to detect errors or suppress warnings.
  - *Compiler-time and deployment-time processing* — Software tools can process annotation information to generate code, XML files,etc.
  - *Runtime processing* — Some annotations are available to be examined at runtime.

Defining Annotations

```
[meta-annotations declaration]
public @interface AnnotationName {
  [annotation's elements]
}
```

4 meta-annotations (`java.lang.annotation` package) :

- **`@Target(ElementType)`:** where the annotation can be applied.
  - **`CONSTRUCTOR`:** Constructor declaration
  - **`FIELD`:** Field declaration (including enum constants)
  - **`LOCAL_VARIABLE`:** Local variable declaration
  - **`METHOD`:** Method declaration
  - **`PACKAGE`:** Package declaration
  - **`PARAMETER`:** Parameter declaration
  - **`TYPE`:** Class, interface(including annotation) or enum declaration.

Remark:

If **`@Target`** is omitted, it can be applied to any of the mentioned elements.

Defining Annotations

Meta-annotations (cont.):

- **`@Retention(RetentionPolicy)`** – How long the annotation information is kept:
  - **`SOURCE:`** Annotations are discarded by the compiler.
  - **`CLASS:`** Annotations are available in the class file generated by the compiler but can be discarded by the VM.
  - **`RUNTIME:`** Annotations are retained by the VM at run time, so they may be read reflectively.
- **`@Documented:`** Include this annotation in the Javadocs.
- **`@Inherited:`** Allow subclasses to inherit parent annotations.

Annotation Elements

- Syntax:

```
Type elementName() [default default_value];
```

where `Type` may be:

  - All primitives (`int`, `float`, `double`, `byte`, etc.)
  - `String`
  - `Class`
  - Enums (`enum`)
  - Annotations (`annotation`)
  - Arrays of any of the above.

Remarks:

1. The compiler will report an error if you try to use any other types.
2. An annotation without any elements, is called a *marker annotation*.

Annotations and AspectJ

- AspectJ allows the use of annotations in pointcuts.
- Annotations used as part of a statically determinable pointcut must have at least class-retention policy so the compiler retains them in the class file.
- Others need runtime retention so the compiler retains them in the class files and the VM makes them available at runtime.
- Annotation-based type signature pattern
- Annotation-based method signature pattern
- Annotation-based field signature pattern
- Annotation-based pointcuts

Annotation-based type signature pattern

- AspectJ allows the use of annotations in type signature patterns.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Sensitive {
    int level();
}


@Sensitive(level=5)
public class MedicalRecord {
   ...
}
```

Annotation-based type signature pattern

- Examples

`@Secured Account`

The **`Account`** type with the **`Secured`** annotation.

`@Sensitive *`

Any type marked with the **`Sensitive`** annotation.

```
@Sensitive(level=5)
class MedicalRecord {...}
```

```
@Sensitive(level=10)
class NuclearDesign { ...}
```

`@Business* Customer+`

The **`Customer`** type or its subtypes that carry an annotation of a type whose name starts with **`Business`**

```
@BusinessEntity
class Customer {...}
```

```
@BusinessCritical
class PlatinumCustomer extends Customer {...}
```

Annotation-based method signature pattern

- AspectJ supports selecting methods based on the annotations they carry.

```
public class Account {
    ...
    @Transactional
    public void credit(double amount) {
       ...
    }
    ...
}
```

Annotation-based method signature pattern

- Examples

```
@Secured * *(..)
```

Any method marked with the `@Secured` annotation.

```
@Secured @Transactional * *(..)
```

Any method marked with both `@Secured` and `@Transactional` annotations.

```
@(Secured || Transactional) * *(..)
```

Any method marked with either a `@Secured` or `@Transactional` annotation.

```
(@Sensitive *) *(..)
```

Any method that returns a type marked with a `@Sensitive` annotation.

```
MedicalRecord getRecord()
```

Annotation-based method signature pattern

- Examples

`* (@BusinessEntity *).*(..)`

Any method defined in a type annotated with the **`@BusinessEntity`** annotation.

`* *(@RequestParam (*))`

Any method with one parameter marked with the **`@RequestParam`** annotation. The parentheses around that last * are used to group the parameter type.

`eg. void show(@RequestParam Long id)`

`* *(@Sensitive *)` or **`* *((@Sensitive *))`**

Any method with one parameter whose type carries the **`@Sensitive`** annotation.

eg. **`void create(MedicalRecord mr)`**

Annotation-based method signature pattern

- Examples

`* *(@RequestParam (@Sensitive *))`

Any method with one parameter marked with the **`@RequestParam`** annotation, where the parameter's type is marked with the **`@Sensitive`** annotation.

eg. **`void create(@RequestParam MedicalRecord mr)`**

Annotation-based field signature pattern

- AspectJ can use field-level annotations.

```
public class Account {
    @Id private Long id;
    ...
}
```

- Examples

`@Sensitive * *.*`

Any field that is marked with the `@Sensitive` annotation, regardless of the field's type, declaring type, or name

eg. `private @Sensitive SSN socialSecurityNumber;`

`(@Sensitive *) *.*`

Any field whose type is marked with the `@Sensitive` annotation.

`* (@Sensitive *).*`

Any field defined in a type annotated with the `@Sensitive` annotation.

Annotation-based pointcuts

- AspectJ allows selection based on annotations carried by types, methods, and fields.
- Annotation-based pointcuts come in two forms:
  - selection based on matching annotation types
  - collection of the matching annotations.

Eg.

1. If `MedicalRecord` is annotated with `@Sensitive`, `@this(Sensitive)` selects all join points where `this` is of `MedicalRecord` type.

2. If the `@Sensitive` annotation is marked as `@Inherited` it also matches join points where `this instanceof MedicalRecord` is true.

Annotation-based pointcuts

- **`@this(TypePattern or ObjectIdentifier)`**

  Any join point where the *this* object's type carries the annotation of the **`TypePattern`** type.

- **`@target(TypePattern or ObjectIdentifier)`**

  Any join point where the *target* object's type carries the annotation of the **`TypePattern`** type.

- **`@args(TypePattern or ObjectIdentifier, ..)`**

  Any join point where the *arguments' type* carries annotations of the **`TypePattern`**.

- **`@within(TypePattern or ObjectIdentifier)`**

  Any join point in the *lexical scope* of a type that carries an annotation matching the specified **`TypePattern`**.

Annotation-based pointcuts

- **`@withincode(TypePattern or ObjectIdentifier)`**

  Any join point where the matching program element (method or constructor) carries an annotation matching the **`TypePattern`**.

- **`@annotation(TypePattern or ObjectIdentifier)`**

  Any join point where the subject carries the specified annotation:

  - For method, constructor, and advice-execution join points, the subject is the same as the program element.
  - For field-access and exception handler join points, the subject is the field or exception being accessed
  - For initialization and pre-initialization join points, the subject is the first called constructor matching the specified signature.
  - For static initialization join points, the subject is the type being initialized.

Static Crosscutting

- Sometimes there is a need to modify the static structure of the system in order to implement a crosscutting functionality.
- Static crosscutting modifies the static structure of the types (classes, interfaces, and other aspects) and their weave-time behavior.
- There are three types of static crosscutting:
  - Inter-type declaration (ITD)
  - Weave-time error and warning declarations
  - Exception softening

Static Crosscutting

- *Inter-type declaration (ITD):* One type (an aspect) makes declarations for another type (an interface, a class, or an aspect). It consists of support for member introduction, type-hierarchy modification, and annotation supplementation.
- *Weave-time error and warning declarations:* It detects the presence of a join point and issues errors and warnings during the weaving process.
- *Exception softening.* It lets the programmer deal with checked exceptions in a crosscutting manner.

Inter-type declarations

- Member introduction rules:

1. *An aspect may only introduce members with* `public` *or* `private` *access specification:*
    - *public access* - the introduced member is visible to other parts of the system (plain Java classes or aspects).
    - *private access* - the introduced member is visible only to the introducing aspect. When using a private introduction, the name of the member woven into the types is a mangled version.
2. *Multiple aspects may introduce the same named members as long as they have private access.*
    - It is an error to introduce the same public member through multiple aspects.

Inter-type declarations

3. *An aspect may introduce fields (final as well as non-final), methods, and constructors to classes and interfaces.*
   - The aspect may introduce methods along with their implementations to an interface.
   - Contrary to standard Java rules (version <8), interfaces may also contain method implementations (not only declarations) and may also contain non final instance fields.
4. *If a class contains a method, and an aspect introduces the same method to its base interface, the implementation in the class takes precedence.*
   - Similar to overriding a method, but the super() method cannot be called.
5. *A member-introduction declaration can use only one type.*
   - The use of wildcards is prohibited.
   - You can combine a member declaration with a type modification to get the effect of introducing a member into multiple types.

Inter-type declarations - Examples

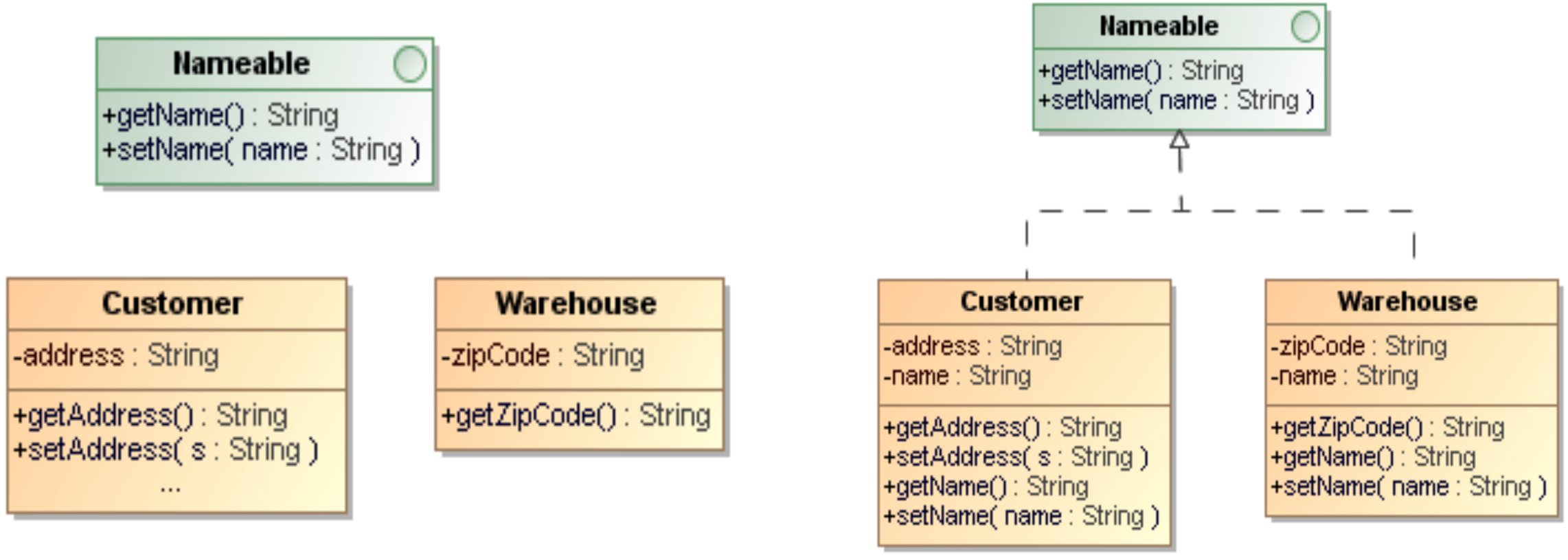
```
public class Customer  {
    private String address;
    public String getAddress() {
    return address;
    }
    public void setAddress(String address) {
    this.address = address;
    }
}
```

Inter-type declarations - Examples

```
public aspect CustomerIdentifiable  {
    private int Customer.id;
    private static int idGen;
    public int Customer.getId() {return id;    }

    public void Customer.setId(int idn) {id = idn;}



    pointcut customerCreation(Customer c):
    initialization(Customer.new(..)) && this(c);



    after (Customer c) returning: customerCreation(c){
    c.setId(idGen++);
    }
}
//... main
    Customer c1=new Customer(), c2=new Customer();
    System.out.println("c1.id="=c1.getId()+" c2.id="+c2.getId());



Output: c1.id=0 c2.id=1
```

Idiom: default interface implementation

- Java (version <8) does not allow interfaces to contain implementation code; only classes can implement methods.
- Sometimes, it would be useful to have a default implementation for interfaces as well.



Idiom: default interface implementation

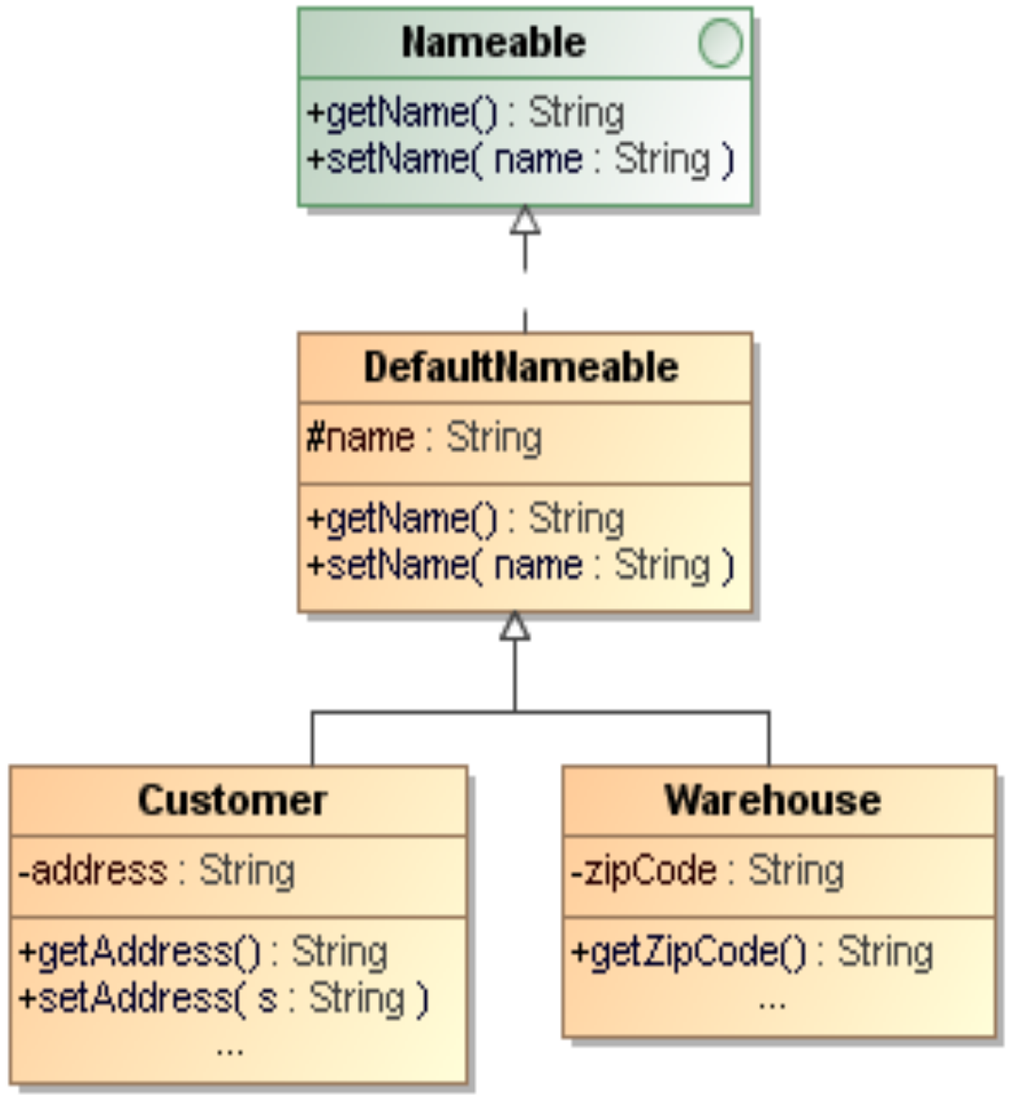
```
public interface Nameable {
    public void setName(String name);
    public String getName();
}
//Java solution
public class Customer implements Nameable {
    private String name;
    public void setName(String name) {
    this.name = name;
    }
    public String getName() {
    return this.name;
    }
 //... other members (methods, attributes)
}
```

Idiom: default interface implementation

Solution without AspectJ: to create a default implementation class for the interface and let the classes extend this class.

Disadvantages:

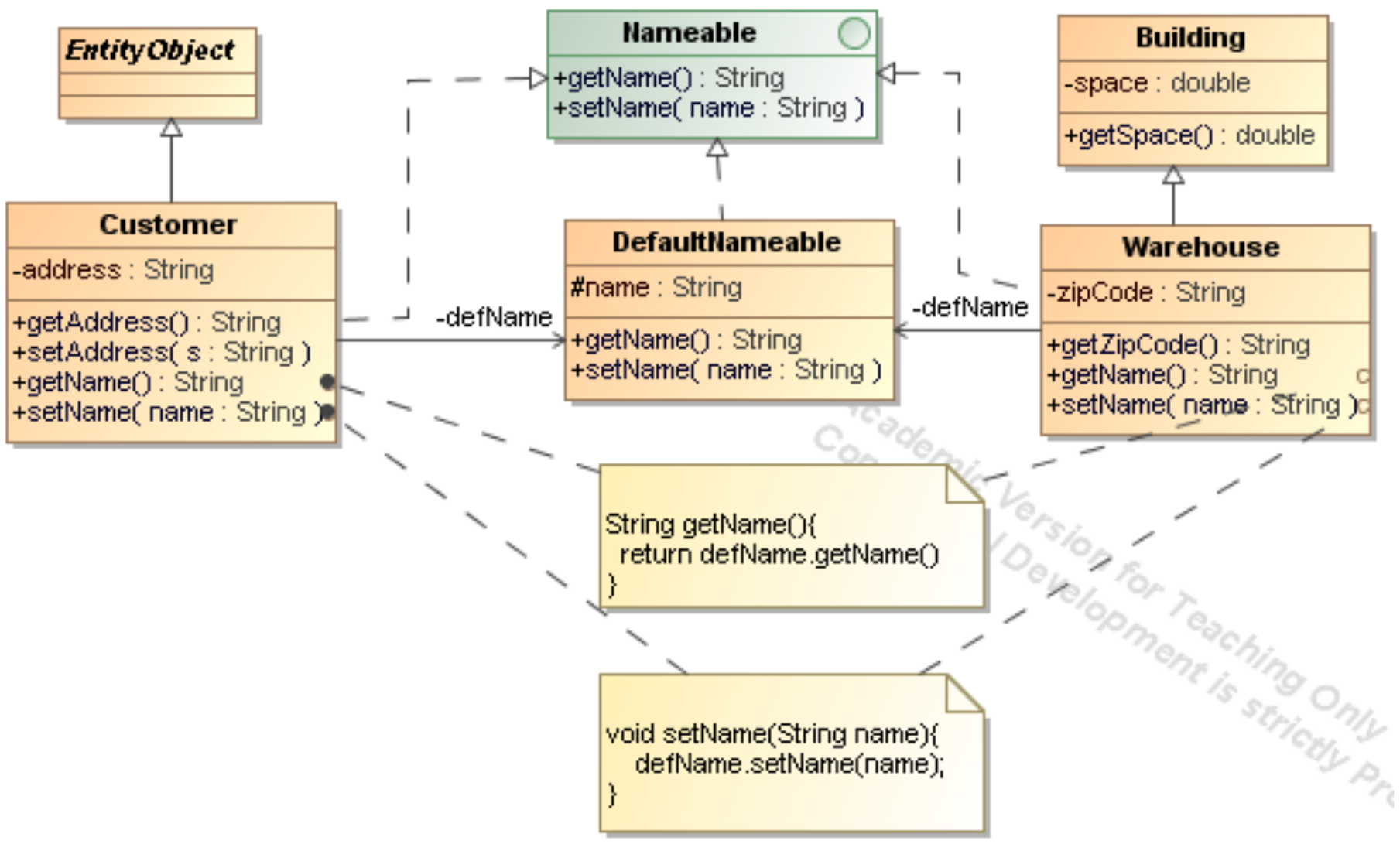
- It works well as long as the implementing classes need to extend only this class, but the solution does not work if you need to implement two or more such interfaces.
- It breaks down if you need to extend another class and implement an interface using its default implementation.



Idiom: default interface implementation

Alternatives:

- Use the delegation pattern by delegating each method to an instance of the default implementation class.
  - You end up with several one-line methods, which causes code scattering (one of the symptoms of a crosscutting concern).



Idiom: default interface implementation

```
//AspectJ solution
public interface Nameable {
    public void setName(String name);
    public String getName();
    static aspect Impl {
    private String Nameable.name;
    public void Nameable.setName(String name) {
        this.name = name;
    }
    public String Nameable.getName() {
        return this.name;
    }
    }
}
public class Customer implements Nameable {
    //... other members (methods, attributes)
}
```

Idiom: default interface implementation

Remarks:

- This solution saves you from writing a lot of code.
- It facilitates making changes. If you need to modify the default implementation, all you need to do is change the nested aspect.
- Although the classes that implement these interfaces no longer have to implement their methods, in some cases you may need to customize a few methods.
  - When such methods are directly implemented in classes, they override the default implementation introduced by the aspect.
- Another variation that can be used provides only a partial default implementation for an interface.

Modifying the Type Hierarchy

- The inheritance hierarchy of existing classes and interfaces can be modified using the `declare parents` construct.
- It has two forms:

```
declare parents : [TypePattern] implements [InterfaceList];
```

or

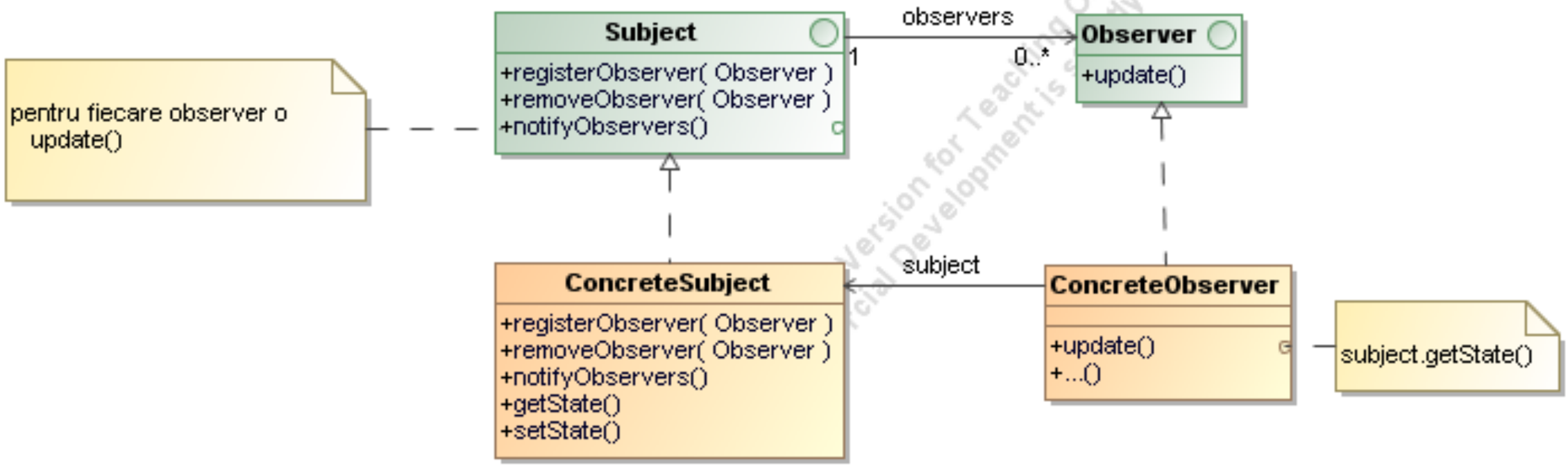
```
declare parents : [TypePattern] extends [Class or InterfaceList];
```

- The declaration of parents must follow the regular Java object-hierarchy rules:
  - You cannot declare a class to be the parent of an interface.
  - You cannot declare parents in such a way that it results in multiple inheritance.

Modifying the Type Hierarchy

- Evaluation order: static and dynamic crosscutting
- AspectJ applies all static crosscutting prior to dynamic crosscutting.
- It applies all `declare parents` statements before evaluating matches for a pointcut.
- Matching is not affected by how a static structure came to be: it could be due to the way classes were written or due to a static crosscutting construct.

Modifying the Type Hierarchy



Introducing Members to Multiple Types

- A combination of member introduction and type-hierarchy modification provides a useful idiom to introduce members to multiple types:
  - create an interface,
  - introduce members to it,
  - declare it the parent of the target types.
- Similar to creating default implementations of interfaces.
- Example: Trace the last-accessed time for each service class.
  - We assume that a service class is marked with the @Service annotation.
  - You need to introduce a field for the last-accessed time in all service classes. *(We cannot use wildcards in a member-introduction statement.)*

Introducing Members to Multiple Types

```
public aspect TimeTracker {
   private static interface LastAccessedTimeHolder {
     static aspect Impl {
       private long LastAccessedTimeHolder.lastAccessedTime;
       public long LastAccessedTimeHolder.getLastAccessedTime() {
         return lastAccessedTime;
       }
       public void LastAccessedTimeHolder.setLastAccessedTime(long time) {
          lastAccessedTime = time;
       }
     }
  }
declare parents : @Service * implements LastAccessedTimeHolder;
before(LastAccessedTimeHolder service)
: execution(* LastAccessedTimeHolder+.*(..)) && this(service) && !within(TimeTracker) {
     service.setLastAccessedTime(System.currentTimeMillis());
   }
}
```

Supplying Annotations

- AspectJ provides support for selecting join points based on annotations carried by the program elements.
- It also offers static crosscutting constructs to supply annotations in a crosscutting manner.
- You can associate annotations with methods, constructors, fields, and types.
- General form:

```
declare @<target-kind>: <Target-element-pattern>: @<Annotation-
    type>[Annotation-properties];
```

- Program elements annotated using the `declare` statement can be used the same way as normally annotated program elements.
- Eg. if a pointcut specifies an annotation as part of the matching specification, program elements annotated using a `declare` statement will match the same way as program elements directly marked with an annotation.

Supplying Annotations

- The declaration has three parts:
  - **`declare @<target-kind>`** declares the kind of elements annotated by the statement
  - **`<Target-element-pattern>`** uses a signature pattern to select the program elements to be annotated
  - **`@<Annotation-type>[Annotation-properties]`** declares the annotation, which may contain any properties available for the annotation.
- Like normal annotations, **`declare`** statements must use annotations that are compatible with the elements being annotated.

Supplying Annotations

- Method:

```
declare @method: <Method signature pattern>: <Annotation>;
```

Eg. `declare @method: * AccountService.*(..):`
`@Transactional(Propagation.Required);`

- Constructor:

```
declare @constructor: <Constructor signature pattern>: <Annotation>
```

Eg. `declare @constructor: AccountService+.new():@ConfigurationOnly;`

- Field:

```
declare @field:<Field signature pattern>: <Annotation>;
```

Eg. `declare @field:* MissileInformation.*: @Classified;`

- Type:

```
declare @type:<Type signature pattern>: <Annotation>;
```

Eg. `declare @type: banking..* :@PrivacyControlled;`

Declaring weave-time errors and warnings

- AspectJ provides a static crosscutting mechanism to declare weave-time errors and warnings based on certain usage patterns.
- This mechanism is often used with the AspectJ compiler as the weaver and therefore is also referred to as a compile-time error and warning construct.
- With this mechanism, you can implement behavior similar to the #error and #warning preprocessor directives supported by some C/C++ compilers.
- **`declare error`**
- **`declare warning`**

Declaring weave-time errors and warnings

- The **`declare error`** construct provides a way to declare a weave-time error when the compiler detects the presence of a join point matching a given pointcut. The compiler issues an error, prints the given message for each detected use, and aborts the compilation process:

```
declare error : <pointcut> : <message>;
```

- The **`declare warning`** construct provides a way to declare a compile-time warning, but it doesn't abort the compilation process:

```
declare warning : <pointcut> : <message>;
```

Remark:

You cannot use the pointcuts that use runtime checks to select the matching join points—**`this(), target(), args(), @this(), @target(), @args(), if(), cflow(), and cflowbelow()`**— in their declaration.

Declaring weave-time errors and warnings

- A typical use of these constructs is enforcing rules, such as prohibiting calls to certain unsupported methods, or issuing a warning about such calls.

```
declare error : callToUnsafeCode() : "This third-party code is known to
    result in a crash";


declare warning : callToBlockingOperation() : "Please ensure you are
    not calling this from an AWT thread";
```

Softening Checked Exceptions

- Java specifies two categories of exceptions that a method may throw: checked and unchecked.
  - Checked - callers must deal with it either by catching exception or by declaring that they can throw it.
  - Unchecked (it directly or indirectly extends `RuntimeException` or `Error`) - callers need not deal with it explicitly and the exception is automatically propagated up the call stack.
- Exception softening lets you treat checked exceptions thrown by specified pointcuts as unchecked ones.
- It eliminates the need to explicitly deal with them in the caller code.
- The exception-softening feature helps modularize the crosscutting concerns of exception handling.

Softening Checked Exceptions

- To soften exceptions, you use the **`declare soft`** construct:

```
declare soft : <ExceptionTypePattern> : <pointcut>;
```

- If a method is throwing more than one checked exception, you must soften each one individually.
- Exception softening is a quick way to avoid tangling the concern of exception handling with the core logic.

Remark:

Do not overuse this technique. One of the reasons to use checked exceptions is that it forces you to handle them by making a conscious decision about processing the exceptions or propagating them to the caller.

Softening Checked Exceptions

```
import java.rmi.RemoteException;
public class TestSoftening {
   public static void main(String[] args) {
     TestSoftening test = new TestSoftening();
     test.perform();
   }
   public void perform() throws RemoteException {
     throw new RemoteException();
   }
}
```

Compilation error: `main` does not catch the `RemoteException`, nor does it specify that it throws the exception.

Softening Checked Exceptions

```
import java.rmi.RemoteException;
public class TestSoftening {
   public static void main(String[] args) {
     TestSoftening test = new TestSoftening();
     test.perform();
   }
   public void perform() throws RemoteException {
     throw new RemoteException();
   }
}
import java.rmi.RemoteException;
public aspect SofteningTestAspect {
    declare soft : RemoteException : call(void TestSoftening.perform());
}
ajc TestSoftening.java SofteningTestAspect.aj
    //no compilation errors
```