adbc - Design by Contract for AspectJ

User manual - version 0.2

Adbc is a small and lightweight library that adds support for design by contract to the AspectJ programming language. The library essentially consists of a number of aspects that monitor your contracts at runtime and will throw an exception whenever a contract is broken.

Requirements

- Java 6 (or later)
- AspectJ (tested on versions 1.6.12 and 1.7.2) are required

Installation

Include `adbc.jar` on the build path of your AspectJ project and contract enforcement should be enabled automatically. If you're using Eclipse+AJDT, this is done as follows:

- Right-click your AspectJ project and go to "Properties".
- Go to "AspectJ Build", "InPath".
- Click the "Add (External) JARs..." button and select the `adbc.jar` file.
  (If an exception is thrown, check the Troubleshooting section.)
- Close the Properties window with the OK button. You can now start writing contracts using the annotations available in the `be.ac.ua.ansymo.adbc.annotations` package.

Note that, because the aspects of adbc can advise *any* method call and advice execution, you probably want to hide AJDT's advice markers for those aspects. You can do this by right-clicking your AspectJ project, then go to "AspectJ Tools", "Configure advice markers...". Finally set the icon for all aspects in the `be.ac.ua.ansymo.adbc` package to None.

If you'd like to tinker with adbc on a small toy project before enabling it on your own projects, have a look at the included example in the `adbc/source/src/be/ac/ua/ansymo/example_bank` folder.

Usage

Contracts in adbc are specified using Java annotations. Adbc makes use of the following annotations:

`@requires` Specifies the preconditions of a method, a constructor or an advice. This annotation takes one or more strings as its value, where each string is a contract specified as a boolean expression[1].

`@ensures` Specifies the postconditions of a method, a constructor or an advice.

`@invariant` Specifies the invariants of a class or an aspect.

`@advisedBy` Specifies that a method *expects* to be advised by the listed advice, in the given order. This annotation can only be attached to methods, and has one or more strings as its value. Each string should contain the canonical/absolute name of an advice. Note that an `@advisedBy` annotation on a method is implicitly inherited by any overriding methods in subclasses.

`@pointcutRuntimeTest` If an advice is mentioned in an `@advisedBy` annotation, and its pointcut contains constructs that can only be determined at runtime (e.g. `if`, `cflow`), you should attach this annotation to the advice and copy these runtime tests into the annotation's value. (This information is completely redundant, but adbc currently makes use of this annotation because all pointcut-related information is lost after weaving.)

`@AdviceName` This annotation is part of AspectJ itself, and is used to give a name to an advice. An advice is required to have a name if you want to mention it in an `@advisedBy` annotation.

When writing contracts, the following variables and functions are available:

`$this` The this object

`parameters` You can simply access method/advice parameters via their name

`$result` The return value of a method/advice, available in postconditions

`$old(expr)` The old function evaluates an expression before the method/advice is executed, stores the result, such that it is available in postconditions. This is useful if, for example, you want to compare the old value of a field with the new value.

`$super` When used in a precondition of an overriding method, `$super` refers to the precondition of the overridden method. Likewise, when used in the postcondition, `$super` refers to the postcondition of the overridden method. Used in an invariant, it refers to the invariants of the super class.

`$proc` Depending on whether this keyword is used in the pre-or postconditions of an around advice, `$proc` usually refers to the pre-or postconditions of the method being advised. In general, `$proc` refers to the pre-or postconditions of the body that you *know* will be executed next when making a proceed call. The emphasis on *know* indicates that you should only be aware of any advice that have been mentioned explicitly in an `@advisedBy` annotation of the advised method.

With contract enforcement enabled, contracts are checked at runtime, guided by behavioural subtyping and the advice substitution principle. (See the Modular reasoning section for more information.) Whenever a contract is broken, a `ContractEnforcementException` is thrown, indicating which part of the contract was broken, and who is to blame.

[1] By default, contracts are JavaScript expressions. Other languages can be used by changing the scripting engine.

Modular reasoning

Modular reasoning is about the ability to reason about a method call by just looking at the method body's contracts (in the static type). In practical terms, it should be sufficient to inspect the tooltip you get when hovering over a method call in Eclipse. (In particular, that tooltip also includes the method body's contracts, since its annotations are displayed.) These contracts should be sufficient, in the sense that you can rely on them to be the minimum requirements and guarantees that will actually hold at runtime.

However, like most good things, modular reasoning is not something you get for free. Modular reasoning is not automatically guaranteed by Java, and certainly not by AspectJ. In Java, a method call might behave differently from what you expect. The method body being executed at runtime (in the dynamic type) could be different from the method body that you expected (in the static type). Java by itself does not prevent you from implementing completely different behaviour in an overriding method, compared to the behaviour of the overridden method.

To obtain modular reasoning in Java, there are some "**behavioural subtyping**" rules you need to take into account when implementing a class:

- The preconditions of methods should be equal to or weaker than those in the supertype.
- The postconditions of methods should be equal to or stronger than those in the supertype. (However, this constraint *only* applies if the precondition of the supertype held in the pre-state. If it did not hold, you know that the static type cannot be the supertype or any other ancestor, so you can't "surprise" the caller even if the postcondition is weaker than the supertype.)
- Invariants of the supertype must be preserved.

If a class is unable to comply with these rules, you can always write a wrapper class around whichever class you'd like to extend instead. (.. though you lose the benefit of being able to substitute for that class.)

Modular reasoning in AspectJ presents some additional complexity: Not only can the dynamic type differ from the static type at a method call, but an advice could also completely change the behaviour of that method call. Luckily, there is a very similar set of rules for writing advice, called the **advice substitution principle** (ASP). Note that, if we refer to "contracts of an advised join point", this refers to the contracts of the method body in the static type of an advised method call. (This also applies to advice that matches on execution join points; they modify the behaviour of method calls just the same..)

- The preconditions of the advice should be equal to or weaker than those of the join points it advises.
- The postconditions of the advice should be equal to or stronger than those of the join points it advises. (Again, this constraint only applies if the precondition of the advised join point held in the pre-state.)
- Invariants of the advised join points must be preserved.

The above rules apply to around advice. Since the contracts of before/after advice do not include the effects of their implicit proceed call, there are some small differences in their ASP rules. In case of **before** advice:

- The preconditions of the advice should be equal to or weaker than those of the join points it advises.
- If the preconditions of the advised join points held before executing the advice, they should still hold at the end of the advice.
- Invariants of the advised join points must be preserved.

In case of **after** advice:

- The preconditions of the advice should be equal to or weaker than the *post*conditions of the advised join point.
- If the postconditions of the advised join point held before executing the advice, they should still hold at the end of the advice.
- Invariants of the advised join points must be preserved.

In case an advice is unable to comply with the ASP, this means the advice cannot help but create surprising behaviour that was not expected by the caller of an advised method. To avoid such surprises, we should make the caller aware of the advice somehow. This is done in adbc by means of the `@advisedBy` annotation. If an advice is non-ASP-compliant, it should add its name to an `@advisedBy` annotation in all of its join point shadows. In other words, the advice name should be visible in all method bodies it advises. (Note that the `@advisedBy` annotation is automatically inherited by subclasses.) If one of these methods is ever called, the caller will notice the `@advisedBy` annotation and is aware the method's contracts are altered by the advice listed in an `@advisedBy` annotation.

In summary, this is adbc's approach to modular reasoning in AspectJ:

- When implementing classes, try to take into account the behavioural subtyping rules. If this is not possible, write a wrapper class instead.
- When implementing advice, try to take into account the rules of the ASP.
- If an advice cannot comply with the ASP, the advice's name should be mentioned in an `@advisedBy` annotation at the join point shadows.
  - The pointcut of a non-ASP-compliant advice may not include execution join points. (You can only determine at runtime which method calls are affected by advice on execution join points. As such, it's unclear where to put the `@advisedBy` annotations for such advice.)
  - If multiple non-ASP-compliant advice are advising the same join point, they should be executed in the order specfied by the `@advisedBy` annotation of the join point. When making a proceed call, a non-ASP-compliant advice must also be aware of the advice that follow it in the `@advisedBy` annotation.
  - If ASP-compliant and non-ASP-compliant advice are advising the same join point, the non-ASP-compliant advice get higher precedence. (This is because ASP-compliant advice do not have to take into account the contracts of other advice that share some of its join points.)

Examples

Writing contracts for classes

The following example of a simple `Square` class demonstrates the basic syntax of contracts:

```
@invariant("$this.getWidth()==$this.getHeight()")
class Square {
    @requires("s > 0"})
    @ensures({"$this.getHeight()==s", "$this.getWidth()==s"})
    public void setSize(int s) {...}

    @ensures("$result==$this.getWidth()*$this.getWidth()")
    public int getArea() {...}

    @ensures({"$this.getX()==$old($this.getX())+x",
      "$this.getY()==$old($this.getY())+y")
    public void move(int x, int y) {...}

    ...
}
```

Note that a contract can consist of multiple parts. For example, the postcondition of `setSize()` consists of parts `$this.getHeight()==s` and `$this.getWidth()==s`. This is equivalent to `$this.getHeight()==s && $this.getWidth()==s`. The benefit of writing a contract in multiple parts is: if a contract is broken, we can pinpoint which part was broken, which is more useful than just stating "this contract was broken".

Writing contracts for ASP-compliant aspects

The following is an example of a simple caching advice. Its preconditions are the same as those of the join point it advises. The postconditions are slightly stronger, due to the addition of `isCached(i,val)`. Consequently, the advice satisfies the advice substitution principle.

```
aspect ListCache {
  @requires("$proc"})
  @ensures("$proc && isCached(i,val)")
  void around(int i, Object val):
      call(void List.set(int, Object)) && args(i, val) {
    ...
  }

  ...
}
```

Writing contracts for advice isn't all that different from writing contracts for methods. What is mainly interesting here is the use of the `$proc` keyword. In the precondition, it refers to the preconditions of `List.set()`. Note that the pointcut could potentially also match on an overridden version of `List.set()`. If this is the case, the advice should technically also take into account the preconditions of overridden versions, as calls to those methods are advised as well. Likewise, the `$proc` keyword in the postcondition refers to the postconditions of `List.set`() (or an overridden version).

Writing contracts for non-ASP-compliant aspects

The following is an example of an authentication advice that does not satisfy the advice substitution principle. This is because the postcondition is `true` if the user is not logged in, which clearly is weaker than the postcondition of `Account.transfer()`.

```
public aspect Authentication {
  @requires("$proc")
  @ensures({"from.getOwner().isLoggedIn()?$proc:true"})
  @AdviceName("authenticate")
  void around(Account from, double amount, Account to):
      call(void Account.transfer(double, Account))
      && args(amount, to) && target(from) {
    if (from.getOwner().isLoggedIn()) {
      proceed(from, amount, to);
    }
  }

  ...
}
```

Because this advice does not satisfy the advice substitution principle, it could cause surprises for anyone calling `Account.transfer()`. More specifically, if the user is not logged in, nothing will happen. If this outcome is not specified in `Account.transfer()`, "nothing" isn't exactly what we expected to happen when calling `transfer`.

However, we can restore modular reasoning by adding an `@advisedBy` annotation to the join point shadows of the authentication advice. In this case, we'll add the annotation to the `Account.transfer()` method:

```
public class Account{
  @requires({"amount>0", "to!=null"})
  @ensures({"$this.getAmount()==$old($this.getAmount())-amount",
    "to.getAmount()==$old(to.getAmount())+amount"})
  @advisedBy({"com.bankapp.aspects.Authentication.authenticate"
  ,"com.bankapp.aspects.Authorization.authorize"})
  public void transfer(double amount, Account to) {...}

  ...
}
```

In this example, the `@advisedBy` annotation in `Account.transfer()` mentions two advice: `authenticate` and `authorize`. (Note that any advice in an `@advisedBy` annotation must have a name, i.e. an @AdviceName annotation.) Adding this `@advisedBy` annotation means that the `transfer` method is expecting to be advised by these two advice in the given precedence/order. The `@advisedBy` annotation is now part of `Account.transfer()`'s contracts, and anyone who wishes to call this method should now be aware of the advice mentioned in the annotation, and their contracts.

In effect, when calling `transfer`, you should now ensure the preconditions of `authenticate`. However, note that any use of the `$proc` keyword in `authenticate` will refer to the preconditions of the next advice in the `@advisedBy` annotation, i.e. `authorize`'s preconditions. In turn, the `$proc` keyword in `authorize`'s preconditions will refer to `transfer`'s preconditions. In this sense, every advice mentioned in the `@advisedBy` clause is can be viewed as a sort-of wrapper around the preconditions of `transfer`. The same logic also applies to the postconditions and invariants when calling `transfer`.

Finally, I should also mention that these `@advisedBy` annotations are automatically inherited by subclasses. This means that you don't have to add the annotation again when overriding the `transfer` method.

Configuration

Adbc exposes a few configuration options, such as enabling/disabling contract enforcement, whether or not postconditions or the substitution principle should be checked, or which scripting engine should be used to evaluate contracts. These options can be configured by simply modifying the static fields in the `AdbcConfig` class at any time.

Troubleshooting

- In case Eclipse throws an exception if you try to include `adbc.jar` to the AspectJ build path, you can get around this problem by simply putting the adbc source code into your project instead. This seems due to an AJDT bug similar to #244300. Note that you may be able to include `adbc.jar` on the Aspect Path instead of the Inpath, but then you will only get contract enforcement on classes, not aspects.
- If parameter names are not available in contracts, try passing the "-g:var" command-line parameter to the compiler. (This should be enabled by default when using AJDT.) Otherwise, if parameter names cannot be retrieved, you can use "arg0", "arg1", .. instead.

Caveats

- Keep in mind that adbc is currently still a proof of concept. This means some basic features are still missing:
  - Invariants can only be attached to a class or aspect, not directly to a field.
  - The performance of adbc has lots of room for improvement. (caching, pointcuts not relying on cflow, avoid relying on the dynamic parts of `thisjoinpoint`, ..)
  - There is no syntactic sugar to make `$this` implicit in method calls; it should always be mentioned explicitly.
- The advice substitution principle cannot be enforced yet on higher-order advice (advice that advises advice..), unless this advice accesses the non-static part of the `thisjoinpoint` object. (Our contract enforcement advice needs access to that object, but it is created lazily by the higher-order advice, so it may or may not be available..)
- Checking behavioural subtyping currently assumes that overriding methods use the same parameter names as the overridden method. (This could be solved using e.g. the Paranamer library..)

- There is basic support for the `@advisedBy` annotation, but several things can be improved:
  - An advice mentioned in an `@advisedBy` annotation has to be mentioned by its absolute/canonical name. It would be nicer if you could use its simple name (+ an import statement).
  - If multiple advice are mentioned in an `@advisedBy` annotation:
    * We do not enforce the ordering of the listed advice, but assume this is done by a separate `declare precedence` statement.
    * When resolving the `$proc` variable, we assume that the advice mentioned in the `@advisedBy` annotation use the same parameter names as the join point they advise. (Should be possible to figure out the mapping from the advice's names to those used by the join point.. Could be done by examining the advice's pointcut, but I'd rather not re-invent parts of the AspectJ compiler..)
    * If an advice is mentioned in an `@advisedBy` annotation and its pointcut makes use of constructs that can only be determined at runtime, like `cflow` or `if`, you'll currently need to copy them into a `@pointcutRuntimeTest` annotation attached to the advice. This is needed to determine the effective specification of methods that mention such an advice in their `@advisedBy` annotation. The `@pointcutRuntimetest` annotation is technically redundant information, but it's tricky to fix this since there's currently no such thing as a reflection API for pointcuts.. Another option would be to compile the effective specifications as a preprocessing step, as it can be done statically given the source code..
  - Even though advice can have names (using an `@AdviceName` annotation), AspectJ currently does not support overriding advice, so it's of course not possible either to make use of this feature in an `@advisedBy` annotation.. (The `@advisedBy` annotation could expect a certain aspect, but at runtime a subaspect could fill in that role just the same.)

Release notes

- Version 0.3
  - Added support for the $super keyword to inherit specifications from a superclass
  - Added cache when retrieving contracts
- Version 0.2
  - If an advice cannot comply with the advice substitution principle, modular reasoning can be restored with the `@advisedBy` annotation
  - An advice can refer to the contracts of the advised join point with the `$proc` keyword
  - Configurable scripting engine + configurable variable prefix (i.e. the dollar sign in `$this`, `$result`, etc.) This feature was mainly added so you can switch to the Groovy interpreter, which allows you to access private members in contracts, if desired. An identifier starting with a dollar sign is invalid in Groovy however, which is why you can change it to some other symbol..
  - More information is shown when a contract is broken (e.g. where in the source code is the contract that was broken)

  - Support for contracts on constructors
- Version 0.1
  - Initial release
  - Basic support for preconditions, postconditions and invariants
  - Enforces behavioural subtyping for classes, the advice substitution principle for advice

Contact

If you have any questions, suggestions or other feedback, feel free to contact me at tim.molderez@ua.ac.be.