Topic 4

Design By Contract

References:
Design by Contract by Example
Pragmatic Programmer
Eiffel and JML from Slides by Professor Tevfik Bultan
Applying Design by Contract by Betrand Meyer
Design by Contract with JML by Gary T. Leavens and Yoonsik Cheon
Wikipedia

DBC and Quality Software

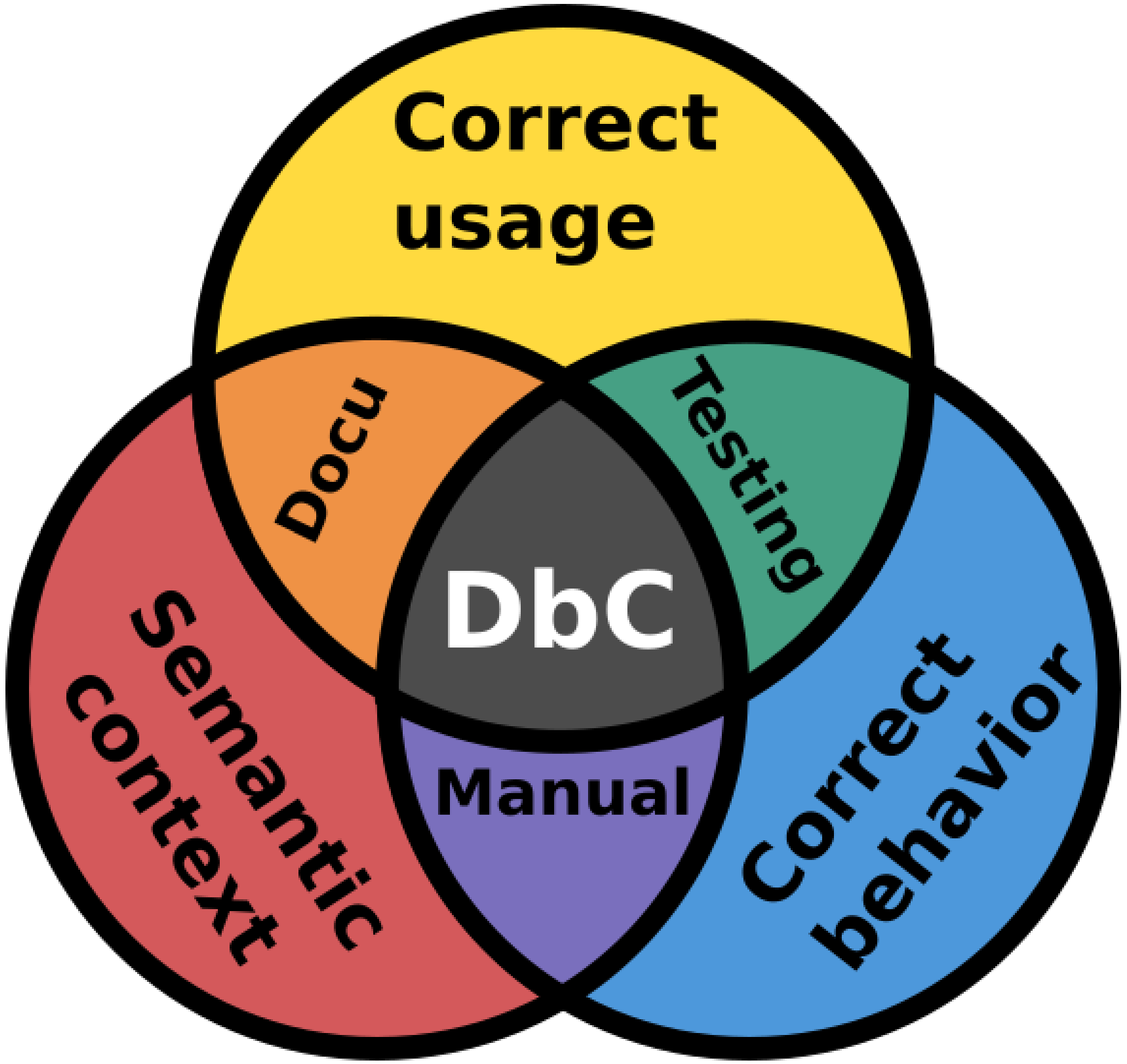
- A key aspect of O-O approach is reuse – as the same components may be used several times with different applications
- Hence reliable components are necessary if quality OO software is to be developed
- Reliability refers to the combination of
  - correctness
  - robustness (e.g. dealing with erroneous input, long running execution, fault tolerance, recoverability)
- Design by Contract helps improve the Reliability of software modules by clearly specifying certain conditions of operation
- High Cohesion suggests that OO programs are made up of many co-operating objects
- Design by Contract also helps improve the Interoperability of software modules by clearly specifying how objects interact

Why a Contract?

- Trading between unknown entities relies on many mechanisms such as guarantees and contracts
- Contracts are used to define the rights and **responsibilities** (remember CRC!) of trading parties
- Contracts also define repercussions when one party fails to abide by the contract
- When you sign a contract with an employer the obligations and benefits are spelt out
  - ❑ Obligations include period of work and expected outcomes (KPIs)
  - ❑ Benefits include your pay
- The same concept can be used in programming between the *client* (calling method) and *supplier* (called method)

What is Design By Contract?

- The idea of Design by Contract was developed in Santa Barbara by a UCSB professor, Bertrand Meyer
- He developed a language that implements the Design by Contract principles (called Eiffel)
- DBC is a conceptually simple but powerful technique documenting the rights and responsibilities of software modules
- By meeting such obligations modules work together to promote program correctness and dependability (reliability and interoperability)



Preconditions and Postconditions

- Every method in a module does something such as returning the square root of a passed parameter
- Each such method may have some assumptions which when met result in some specific outcomes
  - ❑ A square root method, if given a positive number will return another number which when multiplied by itself returns the original number
  - ❑ let's look at the API for `java.lang.Math.sqrt()`
- These expectations and claims can be described as preconditions and postconditions respectively
  - ❑ Preconditions are what must be true before the routine is called (but the supplier/library coder has no control of whether they are met)
  - ❑ Postconditions are what the routine guarantees to do (useful for self testing the routine/method*)
  - ❑ *Use in conjunction with test driven design/unit testing (TDD)

Preconditions and Postconditions

- Pre and postconditions are assertions, i.e. they are expressions which evaluate to true or false
  - ❑ The precondition expresses the requirements that any caller (client) must satisfy
  - ❑ The postcondition expresses the properties that are ensured at the end of the procedure execution (supplier must satisfy)

- If there is no precondition or postcondition, then the precondition or postcondition is assumed to be true (which is equivalent to saying there is no pre or postcondition)

When not to use preconditions?

- For a method that needs to access a file, the condition file-exists should not be made a precondition
- Why?
- It is unpredictable – the file may have been deleted after the existence of the file was verified by the client
- Therefore the supplier is in a better position to enforce the condition
- Preconditions should be enforced only for things that are under the control of caller e.g. parameters

Class Invariants

- A class invariant is an assertion that ALWAYS holds for all instances (objects) of the class
  - –A class invariant must be satisfied at all stages after creation of every instance of the class
  - –i.e. The invariant must be preserved by every *public* method of the class, i.e., if we assume that the invariant holds at the method entry it should hold at the method exit

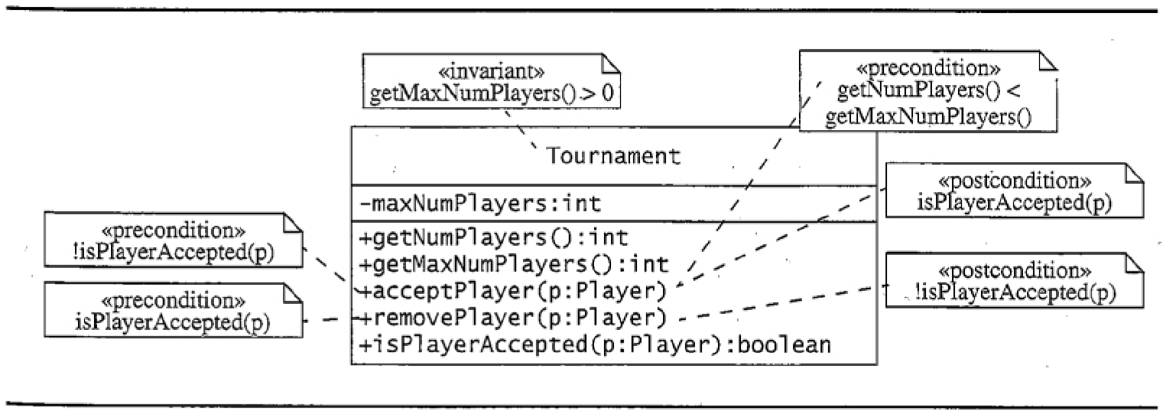
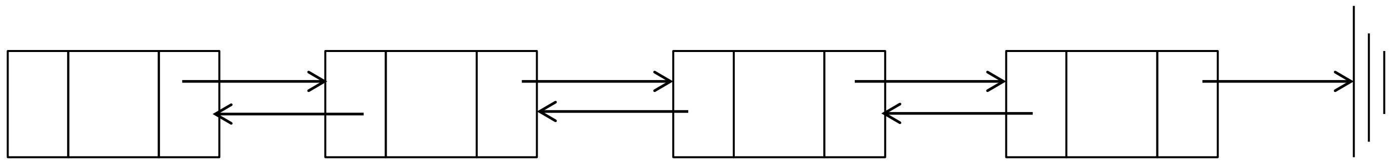

«invariant» getMaxNumPlayers() > 0
«precondition» getNumPlayers() < getMaxNumPlayers()
Tournament
-maxNumPlayers:int
+getNumPlayers():int
+getMaxNumPlayers():int
+acceptPlayer(p:Player)
+removePlayer(p:Player)
+isPlayerAccepted(p:Player):boolean
«precondition» !isPlayerAccepted(p)
«precondition» isPlayerAccepted(p)
«postcondition» isPlayerAccepted(p)
«postcondition» !isPlayerAccepted(p)

Figure 9-4 Examples of invariants, preconditions, and postconditions in OCL attached as notes to the UML model (UML class diagram).

Class Invariants

- The invariant may not hold during internal processing of a method* but it must hold at the end of the method.
- * Including private method calls within the class
- For the doubly linked list below, an invariant may specify that for all the nodes other than the last node node.next.prev = node.
  - This invariant however, may not hold during the time delete method is deleting a node. Why?



Delete Node

Design by Contract: Benefits

- **Pre-condition**:
  - Simplifies method since it does not need to deal with preventable fringe or exceptional cases
  - Avoids expensive double checking where both the method caller and method check pre-conditions.
  - Explicitly states call requirements and serves as documentation
  - can be checked with assertions or Exceptions (exceptions safer since assertions may not be enabled in release versions)
  - use Exceptions for public APIs with wide use (that may be misused beyond your control!)

- **Post-condition**:
  - Explicitly documents and checks expected outcome of method and simplifies debugging since it avoids error propagation
  - Guarantees behaviour for the caller/client
  - can be checked with assertions (self-testing code)

- **Invariant**:
  - Provides a class wide mechanism for checking correctness and removes need to do explicit checks in multiple (i.e. all public) methods

Design by Contract in Eiffel

In Eiffel procedures are written in the following form:

```
procedure_name(argument declarations) is
    -- Header comment
require
    Precondition
do
    Procedure body
ensure
    Postcondition
end
```

Design by Contract in Eiffel

An example:

```
feature put_child(the_node: NODE) is
    -- Add the new node to the children of
    -- the current node.
require
    the_node /= Void
do
    ... Insertion algorithm ...
ensure
    one_more_child: child_count = old child_count + 1;
    the_node_is_a_child: the_node.parent = Current
end
```

- `Current` refers to the current instance of the object (`this` in Java)
- `old` keyword is used to denote the value of a variable on entry to the procedure
- Note that “=“ is the equality operator (== in Java) and “/=“ is the inequality operator (!= in Java)

Class Invariants in Eiffel

- Example, a class invariant for a binary tree could be (in Eiffel notation)
- NOTE *implies* is basically an AND here which is only evaluated if the first condition is true

```
invariant
  left /= Void implies (left.parent = Current)
  right /=Void implies (right.parent = Current)
```

Example Contract

- Here is an example written contract for **adding a node to a tree**.
  - As shown, the Eiffel language enables the software developer to write this contract formally using require and ensure constructs
  - Note the related obligations of client/supplier
  - 3rd party frameworks (covered later in lecture) allow this in Java

Party	Obligations	Benefits
Client	Use as argument a reference, say `the_node`, to an existing object of type `Node`.	Get an updated tree where the `Current` node has one more child than before; `the_node` now has `Current` as its parent.
Supplier	Insert `the_node` as required.	No need check if the argument actually points to an object.

Eiffel Stack Example

```
class stack [T] export
          is_empty, is_full, push, pop, top
    feature
          buffer :  array [T];
          stack_top : integer ;
          Create (n : integer) is
             do
               stack_top := 0;
               buffer.Create (1, n);
             end; -- Create
          is_empty: boolean is
             do
               Result := stack_top <= 0;
             end; -- is_empty
          is_full: boolean is
             do
               Result := stack_top >= buffer.size;
             end; -- is_full
          top: T is
             require
               not is_empty
             do
               Result := buffer.entry (stack_top);
             end; -- pop
pop: T is
   require
      not is_empty
   do
      Result := buffer.entry (stack_top);
      stack_top := stack_top - 1;
   ensure
      not is_full;
      stack_top = old stack_top - 1;
   end; -- pop
push (x : T) is
   require
      not is_full;
   do
      stack_top := stack_top + 1;
      buffer.enter (stack_top, x);
   ensure
      not is_empty; top = x;
      stack_top = old stack_top + 1;
   end; -- push
invariant
      stack_top >= 0 AND stack_top < buffer.size;
end; -- class  STACK
```

See Caspar's converted Java example using Cofoja in Topic4.zip on Canvas

Defensive Programming and Design by Contract

- Defensive Programming is a form of design intended to ensure the continuing function of a piece of software in the presence of unforeseeable usage.
- DP advocates many checks at every stage for both the method and method callers
  - may result in redundant code (e.g. rechecking pre-conditions)
  - adds to code complexity
- DbC can help support a defensive programming approach however defensive programming is stricter.
- e.g. a defensive programming approach would NOT assume that pre-conditions are met

You just *had* to divide by zero, didn't you?
It's okay! I wrote an exception!

Common Defensive Programming Techniques

- Check data structure bounds (buffer overflow problem)
  - problematic in languages such as C/C++
- Encrypt/authenticate all important data transmitted over networks
- All data is important until proven otherwise
- All data is tainted until proven otherwise
- All code is insecure until proven otherwise
- Verify correctness not incorrectness

How much/many obligation(s) should a method specify?

- What if a module places too many obligations?
  - The stronger the preconditions the heavier the burden on the client
  - This could be an indication of a poor design – which must be refactored
  - For example, a module which takes many arguments with separate precondition for each argument
  - Makes it cumbersome to use the method or service
  - May result in a lot of duplicated code in the client that could have been handled in one place in the provider

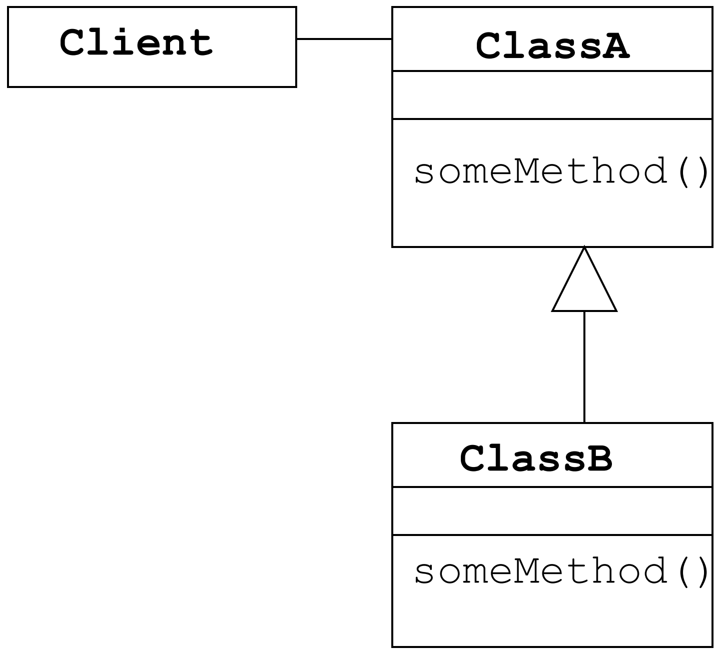
Design by Contract and Inheritance

- Inheritance enables declaration of subclasses which can override some of the methods of the parent class, or provide an implementation for the abstract methods of the parent class

- Polymorphism and dynamic binding combined with inheritance are powerful programming tools provided by object-oriented languages
  - How can the Design by Contract be extended to handle these concepts?

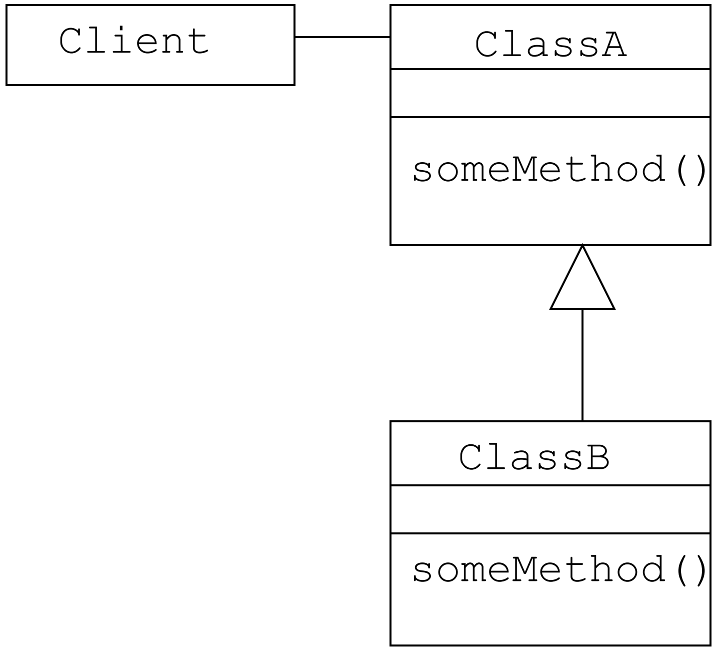
Inheritance: Preconditions

- If the ***precondition*** of `ClassB.someMethod` is ***stronger*** than the precondition of the `ClassA.someMethod`, then this is *not **fair*** to the `Client` (i.e. may inadvertently cause the Client to fail)
- The code for `ClassB` may have been written after `Client` was written, so `Client` has no way of knowing its contractual requirements for `ClassB`
- This breaks the Liskov Substitution Principle which says any subclass can replace a superclass and behave correctly



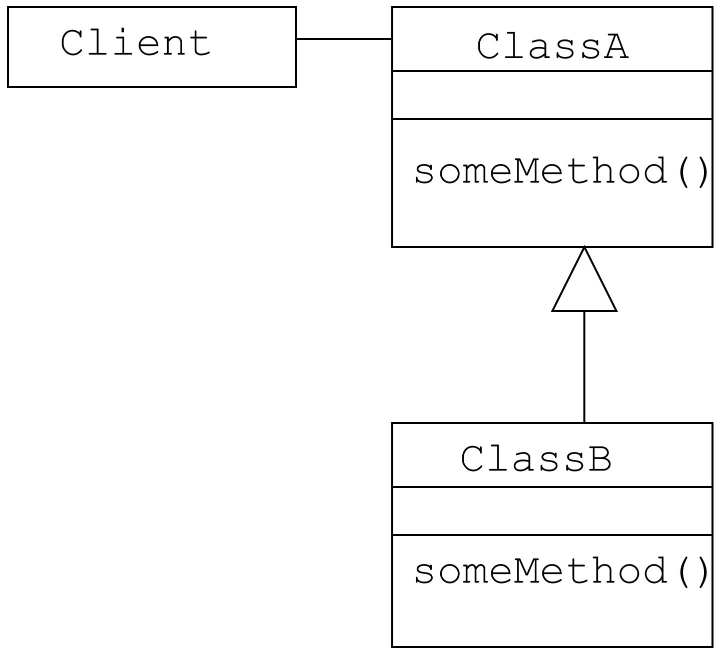
Inheritance: Postconditions

- If the ***postcondition*** of the `ClassB.someMethod` is ***weaker*** than the postcondition of the `ClassA.someMethod`, then this is ***not fair*** to the `Client`
  - since the original contract (post-condition) may not be fulfilled by the new class
- Since `Client` may not have known about `ClassB`, it could have relied on the stronger guarantees provided by the `ClassA.someMethod`
- Again is an LSP problem



Inheritance: Invariants

- If the class ***invariant*** for the `ClassB` is ***weaker*** than the class invariant for the `ClassA`, then this is ***not fair*** to the `Client`
- Since `Client` may not have known about `ClassB`, it could have relied on the stronger guarantees provided by the `ClassA`
- LSP again!



What if a precondition is not met?

- If a pre-condition is not met the simplest solution is to throw an exception to notify the client
  - could also use assertions during debugging however these are turned off for release builds so not suitable for public APIs
- However consider a binary search on an array where the precondition is that the array must be sorted
  - if this precondition is not met the results will (probably) not be valid
  - the class designers chose this precondition since pre-sorting/checking an array is computationally expensive (and wasteful if it is already sorted)
  - so what do we do? (think post-condition)
  - See `java.util.Collections.binarySearch(…)` for an example
- **NOTE**: post-conditions are easier, if post-condition is not met then the code is broken so you need to ***fix it***!

What if no support for DBC?

- The main advantage of using DbC is that it specifies the requirements and guarantees from the beginning
- Specifying these explicitly goes a long way towards writing better software even in languages with no built-in support
- In Java you can add these contract specifications as comments (for example using javadoc)
  - serves as documentation
  - can reinforce with assertions or exceptions (see next slide)
- However these techniques can only partially emulate DbC
  - No built-in support for OLD values in assertions (value at method entry)
  - No support for propagating specifications along the hierarchy – it must be done manually
  - You must explicitly call/check class invariants before exiting any public method within the class

Assertions in Java

- Java assertion statements
  - `assert` ***<condition>*** `;`
  - `assert` ***<condition>*** `:` ***<message>***`;`
- enabling assertions
  - when compiling: `javac -source 1.5 ClassName.java`
  - (can configure source level >= 1.5 in Eclipse)
  - when running: `java -ea ClassName`
  - (can use VM args in *run configuration* in eclipse)

http://docs.oracle.com/javase/1.5.0/docs/guide/language/assert.html

Example:

To check a sort algorithm works you can write the following post-condition:

```
 for (int i=0; i < num_entries-1; i++)
    assert(sorted[i] <= sorted[i+1]);
```

Assertion in Java

```
/** Returns index of value n in array a.
 *  @pre.condition The array a is in sorted order.
 */
public static void binarySearch(int[] a, int n) {
 for (int i=0; i< num_entries-1; i++)
   assert sorted[i] <= sorted[i+1] : "Array must be sorted";
   …


}
```

See `topic4.dbc.AssertionTest.java`

NOTE: `@pre.condition` is informal syntax

NOTE2: It only makes sense to check this precondition during debugging (as discussed previously)

Dynamic Design-by-Contract Monitoring

- Enforce contracts at run-time
- A contract
  - Preconditions of modules
    - What conditions the module requests from the clients
  - Postconditions of modules
    - What guarantees the module gives to clients
  - Invariants of the objects
- Precondition violation, the client is to blame
  - Generate an error message blaming the client (caller)
- Postcondition violation, the service is to blame
  - Generate an error message blaming the server (callee)
  - Fix the bug!
- The Eiffel compiler supports dynamic design-by-contract monitoring. You can run the program with design-by-contract monitoring on, and it will report any contract violations at runtime
  - this is similar to enabling/disabling assertions in Java

Design by Contract with Cofoja

- Cofoja is an open source project based on annotations that simplifies using contracts
- It is based on the ASM library which facilitates bytecode manipulation of Java classes for aspect weaving (adding functionality to compiled code)
- It supports three main annotations (supporting inheritance)
- **Pre-condition:**

```
@Requires("expression")
ContractedDeclaration
...
```

- **Post-condition**

```
@Ensures("expression")
ContractedDeclaration
...
```

- **Invariant**

```
@Invariant({
  "class invariant 1",
  "class invariant 2"
})
```

- See https://github.com/nhatminhle/cofoja for more details (documentation)

Using Cofoja in Eclipse (Java 8)

Requires some configuration!

- Use a JDK 8 adding the following (adjusting path) to start of `eclipse.ini` (in root of eclipse installation) to make sure eclipse runs from a JDK and not a JRE (since it needs access to `javac`)

```
-vm
C:/Java/jdk1.8.0_241/bin
```

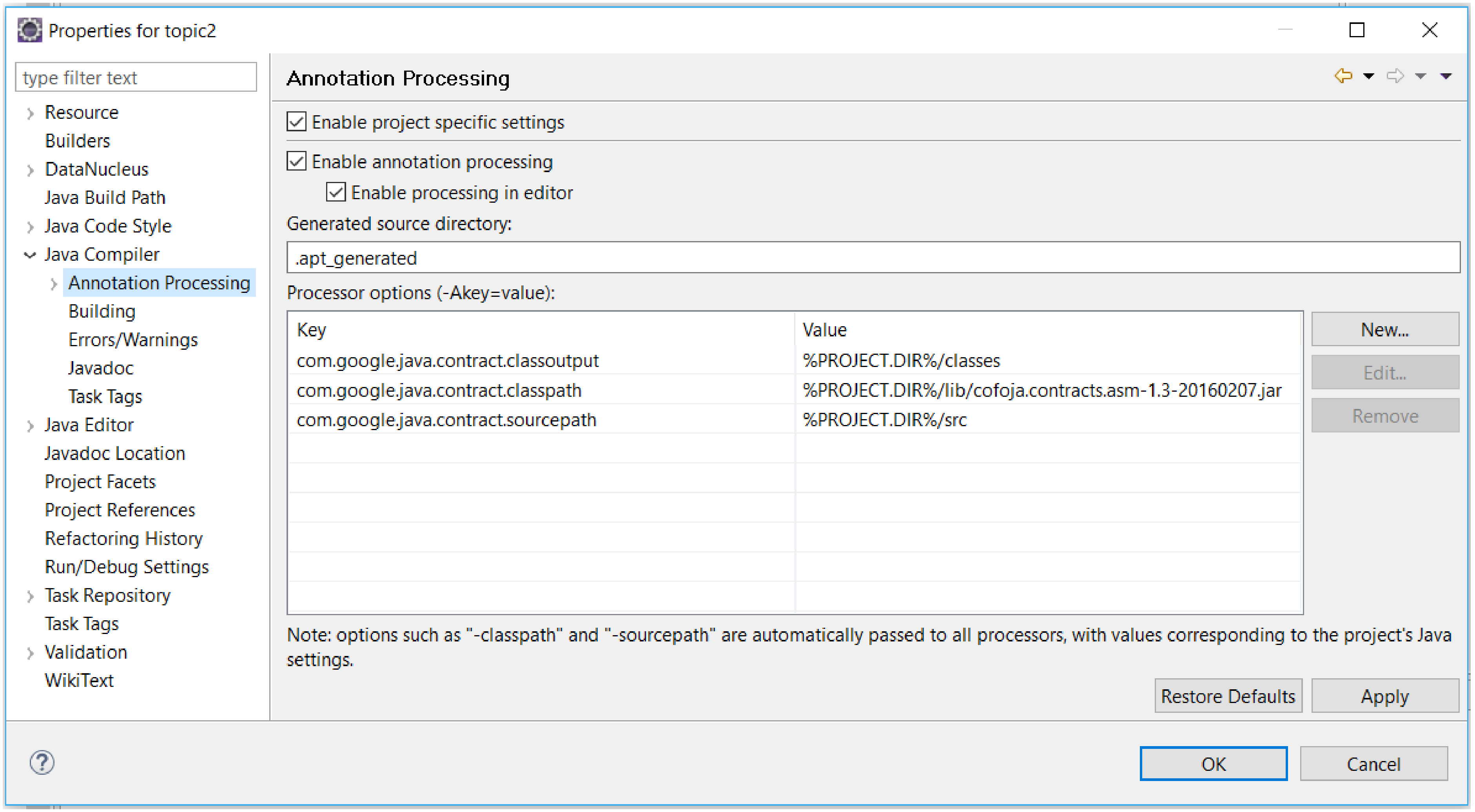
- NOTE: must be on two lines as above!
- configure Project -> Properties -> Java Compiler -> Annotation Processing according *exactly* to the following two slides **adjusting paths** for your own installation
- See https://github.com/nhatminhle/cofoja for details of the meanings of the properties
- Download .jar releases at https://github.com/nhatminhle/cofoja/releases
- add the cofoja jar (e.g. `cofoja.contracts.asm-1.3-20160207.jar`) to the project build path
- add `-javaagent:lib/cofoja.contracts.asm-1.3-20160207.jar` to the VM args in your `AssertionTestCofoja.java` Eclipse run configuration (Run -> Run Configurations) providing full path to the lib if it is external to the project
- you should now be able to run the provided example code `AssertionTestCofoja.java` in the topic4 source code examples

Using Cofoja in Eclipse (Java 11+)

Requires MORE configuration!

In Java 11 we need a newer version of ASM bytecode libraries used by Cofoja

- Configuration is similar to Java 8 but we use the Cofoja jar that does not contain the ASM libraries (e.g. `cofoja.contracts-1.3-20160207.jar`
- The rest of the configuration is the same as for Java 8 but you must point to the correctly named non ASM jar
- Additionally, you must also add the following three ASM libraries to the build path since there are is no longer an "asm-all" jar that contains them all
  - `asm-7.3.1.jar`, `asm-commons-7.3.1.jar` and `asm-tree-7.3.1.jar`
  - These are available at https://mvnrepository.com/artifact/org.ow2.asm
- The `Topic4.zip` I have provided already has the project configured for Java 11 although you need to add the java agent flag to the run configuration and may have to add the `–vm` argument to `eclipse.ini`
- NOTE There are newer versions of cofoja on the git but you must build them yourself if you want to use them!



Properties for topic2

type filter text

- Resource
- Builders
- DataNucleus
- Java Build Path
- Java Code Style
- Java Compiler
  - Annotation Processing
  - Building
  - Errors/Warnings
  - Javadoc
  - Task Tags
- Java Editor
- Javadoc Location
- Project Facets
- Project References
- Refactoring History
- Run/Debug Settings
- Task Repository
- Task Tags
- Validation
- WikiText

Annotation Processing

☑ Enable project specific settings

☑ Enable annotation processing

☑ Enable processing in editor

Generated source directory:

.apt_generated

Processor options (-Akey=value):

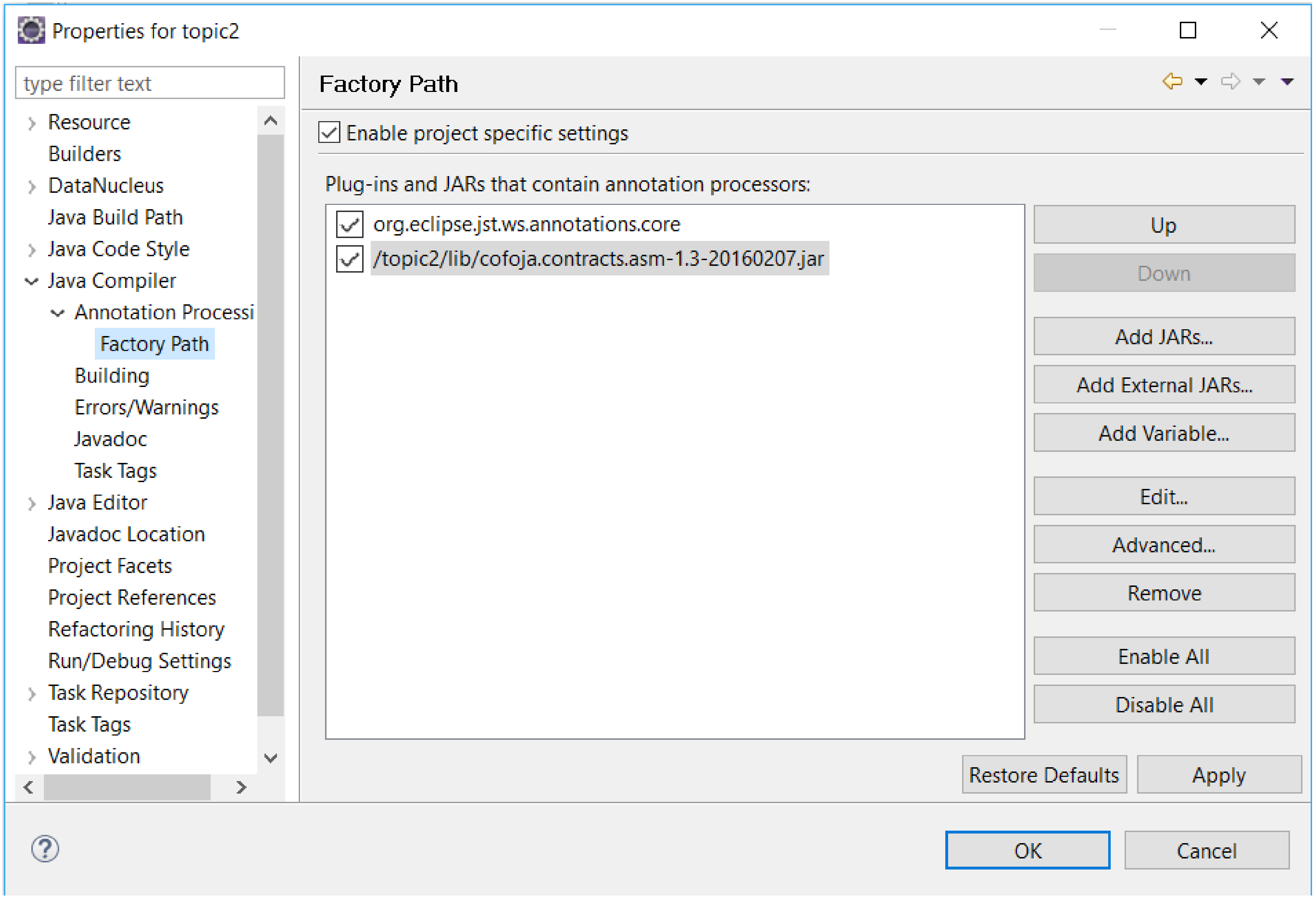
Key	Value
com.google.java.contract.classoutput	%PROJECT.DIR%/classes
com.google.java.contract.classpath	%PROJECT.DIR%/lib/cofoja.contracts.asm-1.3-20160207.jar
com.google.java.contract.sourcepath	%PROJECT.DIR%/src

New... Edit... Remove

Note: options such as "-classpath" and "-sourcepath" are automatically passed to all processors, with values corresponding to the project's Java settings.

Restore Defaults Apply

OK Cancel


Properties for topic2

type filter text

- Resource
- Builders
- DataNucleus
- Java Build Path
- Java Code Style
- Java Compiler
  - Annotation Processi
    - Factory Path
  - Building
  - Errors/Warnings
  - Javadoc
  - Task Tags
- Java Editor
- Javadoc Location
- Project Facets
- Project References
- Refactoring History
- Run/Debug Settings
- Task Repository
- Task Tags
- Validation

Factory Path

☑ Enable project specific settings

Plug-ins and JARs that contain annotation processors:

☑ org.eclipse.jst.ws.annotations.core
☑ /topic2/lib/cofoja.contracts.asm-1.3-20160207.jar

Sample Question

The *StudentEnrol* class manages enrolment of students who are allowed to take up to five courses made up of cores and electives from the list of cores (*coresOffered*) and electives (*electivesOffered*) offered. There is an additional constraint that no student can enrol in more electives than core courses. You are required to write the pre and post-conditions for all methods (they require or ensure) as well as the class invariants. Specify invariants, pre and post conditions as comments using the annotations **@invariant, @pre.condition** and **@post.condition**. You may use the notation *OLD.cores* in the post-conditions to refer to the original value for cores.

```java
class StudentEnrol
{
   private final Set<String> coresOffered = new HashSet<String>(
         Arrays.asList("P1", "OS", "SEF", "DC", "WP"));
   private final Set<String> electivesOffered
        = new HashSet<String>
              (Arrays.asList("C++", "PHP", "Web3D", "IT"));
   private Set<String> electives = new HashSet<String>();
   private Set<String> cores = new HashSet<String>();

   // adding a core
   public void addCore(String core)
   {
      cores.add(core);
   }

   // adding an elective
   public void addElective(String elective)
   {
      electives.add(elective);
   }
```

```
    // removing a core
    public void removeCore(String core)
    {
       cores.remove(core);
    }

    // removing an elective
    public void removeElective(String elective)
    {
       electives.remove(elective);
    }

    // listing all courses enrolled
    public void display()
    {
       for (String course : cores)
          System.out.println(course);
       for (String course : electives)
          System.out.println(course);
    }
}
```

Summary

- DBC defines rights, obligations and correctness criteria for a software module:
  - If the class invariant and precondition are true before a supplier module is called by a client, then the invariant AND the postcondition (obligation) will be true after the service has been completed
  - When making calls to a supplier, a software module should not violate the supplier's preconditions (rights)
- DBC can be specified in Java as follows
  - comments in the code (e.g. using Javadoc)
  - exceptions and assertions
  - an external library such as Cofoja
- Unlike defensive programming the supplier module is not responsible for prevention or a course of action (other than raising a notification) if the required conditions are not met