CS189A - Capstone

Christopher Kruegel
Department of Computer Science
UC Santa Barbara
http://www.cs.ucsb.edu/~chris/

Design by Contract

- Design by Contract and the language that implements the Design by Contract principles (called Eiffel) was developed in Santa Barbara by Bertrand Meyer

- Meyer won the 2006 ACM Software System Award for the Eiffel!
  - Award citation: "*For designing and developing the Eiffel programming language, method and environment, embodying the Design by Contract approach to software development and other features that facilitate the construction of reliable, extendible and efficient software*."

- The company which supports the Eiffel language is located in Santa Barbara:
  - Eiffel Software (http://www.eiffel.com)

- The material in the following slides is mostly from the following paper:
  - "Applying Design by Contract," B. Meyer, IEEE Computer, pp. 40-51, October 1992.

Dependability and Object-Orientation

- An important aspect of object oriented design is reuse
  - For reusable components correctness is crucial since an error in a module can effect every other module that uses it

- Main goal of object oriented design and programming is to improve the quality of software
  - The most important quality of software is its dependability

- Design by contract presents a set of principles to produce dependable and robust object oriented software
  - Basic design by contract principles can be used in any object oriented programming language

What is a Contract?

- There are two parties:
  - Client which requests a service
  - Supplier which supplies the service

- Contract is the agreement between the client and the supplier

- Two major characteristics of a contract
  - Each party expects some benefits from the contract and is prepared to incur some obligations to obtain them
  - These benefits and obligations are documented in a contract document

- Benefit of the client is the obligation of the supplier, and vice versa.

What is a Contract?

- As an example let’s think about the contract between a tenant and a landlord

Party	Obligations	Benefits
Tenant	Pay the rent at the beginning of the month.	Stay at the apartment.
Landlord	Keep the apartment in a habitable state.	Get the rent payment every month.

What is a Contract?

- A contract document between a client and a supplier protects both sides
    - It protects the client by specifying how much should be done to get the benefit. The client is entitled to receive a certain result.
    - It protects the supplier by specifying how little is acceptable. The supplier must not be liable for failing to carry out tasks outside of the specified scope.

- If a party fulfills its obligations it is entitled to its benefits
    - No Hidden Clauses Rule: no requirement other than the obligations written in the contract can be imposed on a party to obtain the benefits

What is this all about?

- You are not in law school, so what are we talking about?
- Here is the basic idea
  - One can think of pre and post conditions of a procedure as obligations and benefits of a contract between the client (the caller) and the supplier (the called procedure)
- Design by contract promotes using pre and post-conditions (written as assertions) as a part of module design
- Eiffel is an object oriented programming language that supports design by contract
  - In Eiffel the pre and post-conditions are written using require and ensure constructs, respectively

Design by Contract in Eiffel

In Eiffel procedures are written is in the following form:

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

Design by Contract in Eiffel

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

- 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)

The put_child Contract

- The put_child contract in English would be something like the table below.
  - Eiffel language enables the software developer to write this contract formally using require and ensure constructs

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

Contracts

- The pre and postconditions are assertions, i.e., they are expressions which evaluate to true or false
  - The precondition expresses the requirements that any call must satisfy
  - The postcondition expresses the properties that are ensured at the end of the procedure execution

- 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)

Assertion Violations

- What happens if a precondition or a postcondition fails (i.e., evaluates to false)
  - The assertions can be checked (i.e., monitored) dynamically at run-time to debug the software
  - A ***precondition violation*** would indicate a bug at the ***caller***
  - A ***postcondition violation*** would indicate a bug at the ***callee***

- Our goal is to prevent assertion violations from happening
  - The pre and postconditions are not supposed to fail if the software is correct
    - hence, they differ from exceptions and exception handling
  - By writing the contracts explicitly, we are trying to avoid contract violations, (i.e, failed pre and postconditions)

Assertion Violations

- In the example below, if `new_child = Void` then precondition fails
- The procedure body is not supposed to handle the case where `new_child = Void`, that is the responsibility of the caller

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

Defensive Programming vs. Design by Contract

- Defensive programming is an approach that promotes putting checks in every module to detect unexpected situations

- This results in redundant checks (for example, both caller and callee may check the same condition)
  - A lot of checks makes the software more complex and harder to maintain

- In Design by Contract the responsibility assignment is clear and it is part of the module interface
  - prevents redundant checks
  - easier to maintain
  - provides a (partial) specification of functionality

Class Invariants

- A class invariant is an assertion that holds for all instances (objects) of the class
  - A class invariant must be satisfied after creation of every instance of the class
  - The invariant must be preserved by every method of the class, i.e., if we assume that the invariant holds at the method entry it should hold at the method exit
  - We can think of the class invariant as conjunction added to the precondition and postcondition of each method in the class

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

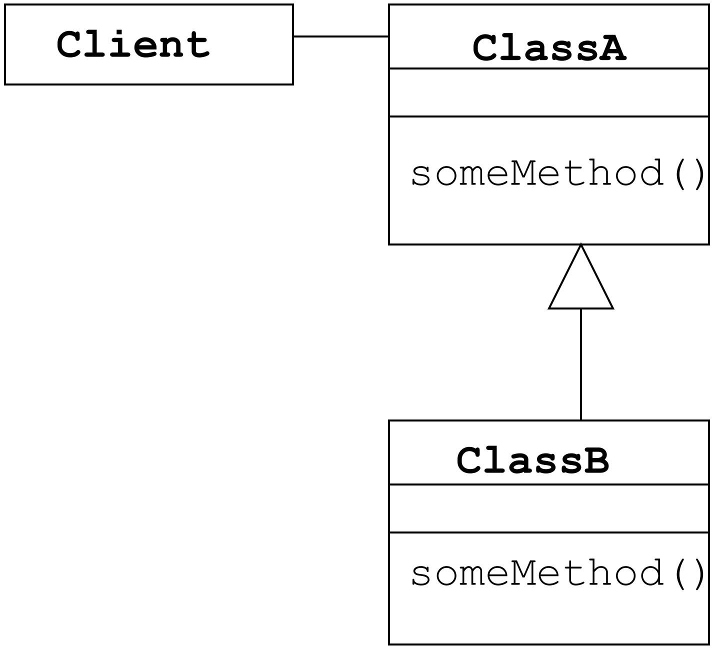
Design by Contract and Inheritance

- Inheritance enables declaration of subclasses which can redeclare 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 can be extended to handle these concepts?

Inheritance: Preconditions

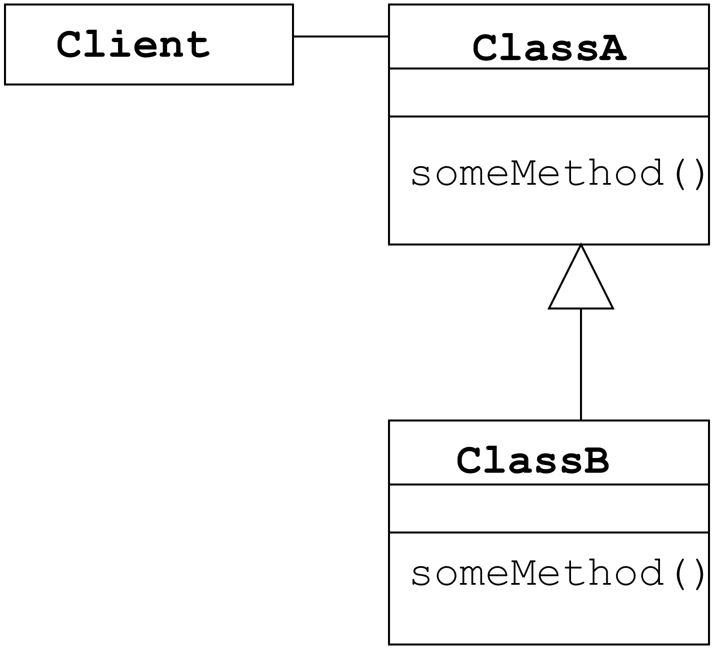
- If the precondition of the `ClassB.someMethod` is stronger than the precondition of the `ClassA.someMethod`, then this is not fair to the `Client`
- The code for `ClassB` may have been written after `Client` was written, so `Client` has no way of knowing its contractual requirements for `ClassB`



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 `Client` may not have known about `ClassB`, it could have relied on the stronger guarantees provided by the `ClassA.someMethod`



Inheritance

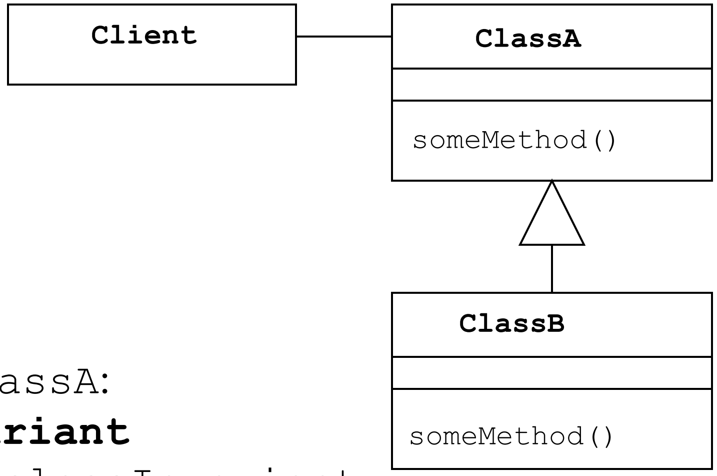
- Eiffel enforces the following
  - the precondition of a derived method to be weaker
  - the postcondition of a derived method to be stronger

- In Eiffel when a method overwrites another method the new declared precondition is combined with previous precondition using disjunction

- When a method overwrites another method the new declared postcondition is combined with previous postcondition using conjunction

- Also, the invariants of the parent class are passed to the derived classes
  - invariants are combined using conjunction

Inheritance - Example



In ClassA:

```
invariant
    classInvariant
someMethod() is
require
    Precondition
do
    Procedure body
ensure
    Postcondition
end
```

In ClassB which is derived from ClassA:

```
invariant
    newClassInvariant
someMethod() is
require
    newPrecondition
do
    Procedure body
ensure
    newPostcondition
end
```

Inheritance - Example

The precondition of `ClassB.aMethod` is defined as:
`newPrecondition` **or** `Precondition`

The postcondition of `ClassB.aMethod` is defined as:
`newPostcondition` **and** `Postcondition`

The invariant of `ClassB` is
`classInvariant` **and** `newClassInvariant`

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 server is to blame
  - Generate an error message blaming the server (callee)

Writing Contracts in JML

- Java Modeling Language (JML) is an annotation language for Java that enables specification of contracts for Java classes as annotations.
  http://www.cs.iastate.edu/~leavens/JML/

- JML can be used to write pre, post-conditions and invariants for Java classes

- There are dynamic contract checking tools based on JML (jmlc) that compile the source code and insert assertion checks for dynamic contract checking

- There are static contract checking tools (such as ESC Java) which check JML contracts statically
  - Static verification tools can check contracts at compile-time rather than runtime. JContractor checks contracts at runtime.

jContractor: Design-by-Contract for Java

- jContractor is a design by contract tool for Java
  - http://jcontractor.sourceforge.net/
  - Developed here at UCSB by Murat Karaorman

- References:
  - "jContractor Crash Course", Parker Abercrombie, http://jcontractor.sourceforge.net/doc/crashcourse.html
  - jContractor: Bytecode instrumentation techniques for implementing design by contract in Java." In Proceedings of Second Workshop on Runtime Verification, RV 02. Copenhagen, Denmark. July 26, 2002.

Preconditions

- Precondition of a method is written as a Boolean method and its name is the method name followed by "`_Precondition`"
- A method's precondition is checked when execution enters the method.
- Precondition methods return Boolean and take the same arguments as the non-contract method that they correspond to.

```
protected boolean push_Precondition (Object o) {
  return o != null;
}
private boolean searchStack_Precondition (Object o) {
  return o != null;
}
protected boolean Stack_Precondition (Object [] initialContents) {
  return (initialContents != null) && (initialContents.length > 0);
}
```

jContractor

- Contracts in jContractor are written as Java methods that follow a simple naming convention.
  - Assertions are written as Java methods that return a Boolean value
- jContractor provides runtime contract checking by instrumenting the bytecode of classes that define contracts.
- jContractor can
  - either add contract checking code to class files to be executed later,
  - or it can instrument classes at runtime as they are loaded.
- Contracts can be written in the class that they apply to, or in a separate contract class.

An Example Class

```
class Stack implements Cloneable {
  private Stack OLD;
  private Vector implementation;
  public Stack () { ... }
  public Stack (Object [] initialContents) { ... }
  public void push (Object o) { ... }
  public Object pop () { ... }
  public Object peek () { ... }
  public void clear () { ... }
  public int size () { ... }
  public Object clone () { ... }
  private int searchStack (Object o) { ... }
}
```

Postconditions

- Postcondition of a method is written as a Boolean method and its name is the method name followed by "`_Postcondition`"
- A method's postcondition is checked just before the method returns.
- Postcondition methods return Boolean and take the same arguments as the non-contract method, plus an additional argument of the method's return type. This argument (called `RESULT`) must be the last in the list, and holds the value returned by the method.

```
protected boolean push_Postcondition (Object o, Void RESULT) {
   return implementation.contains(o) && (size() == OLD.size() + 1);
}

protected boolean size_Postcondition (int RESULT) {
  return RESULT >= 0;
}

protected boolean Stack_Postcondition (Object [] initialContents,
Void RESULT) {
   return size() == initialContents.length;
}
```

Postconditions

- Postconditions may refer to the state of the object at method entry through the `OLD` instance variable.
- This variable must be declared private, and must have the same type as the class that contains it.
- If a class defines an `OLD` variable, it must also implement `Cloneable` and provide a `clone()` method.
- When execution enters a method, a clone of the object will be created and stored in `OLD`.

Invariants

- Class invariants are checked at the entry and exit of every public method in the class.
- The invariant is defined in a method called "`_Invariant`" that takes no arguments and returns a boolean.

```
protected boolean _Invariant () {
       return size() >= 0;
}
```

Assertion Evaluation Rule

- Since in the jContractor approach we use Java methods to write preconditions, postconditions and invariants, some problems may occur

- Consider the example below where invariant method makes a call to the method size().

- When size method is called the invariant is checked at the entry which makes a call to size(), and this results in an infinite recursion

```
class Stack  {
   ...
   public int size () { ... }
   protected boolean _Invariant () { return size() >= 0; }
}
```

Assertion Evaluation Rule

- To solve this problem, jContractor uses the following rule
  - **Assertion Evaluation Rule**: Only one contract can be checked at a time

- In the `Stack` example, the invariant will call `size()` and since there is already a contract check in progress the invariant will not be checked on `size()`

- jContractor implements Assertion Evaluation Rule by maintaining a shared hash table of threads that are actively checking contracts. Before a thread checks a contract it queries the table to see if it is already checking one.

Implementing OLD

- jContractor uses the OLD instance variable in postconditions

```
class Stack {
  private Stack OLD;
  private Vector implementation;
  ...
  public void push (Object o) { implementation.addElement(o); }
  protected boolean push_Precondition (Object o) {
     return o != null; }
  protected boolean push_Postcondition (Object o, Void RESULT) {
     return implementation.contains(o) &&
     (size() == OLD.size() + 1);
  }
}
```

Implementing OLD

- Simply storing the cloned state in the OLD instance variable is not sufficient.
  - The value needs to be saved at the entry point of every method that uses OLD in its postcondition

- Hence following instrumentation would not work

```
public void push (Object o) {
  OLD = (Stack) clone();
  // Check precondition and invariant
  // Method body
  // Check postcondition. OLD holds state at entry
  // Check invariant
}
```

Implementing OLD

- jContractor uses a stack to store OLD instances

```
public void push (Object o) {
  jContractorRunTime.pushState(clone());
  // check precondition and invariant
  // Method body
  // Check postcondition and invariant
}

protected boolean push_Postcondition (Object o, Void RESULT) {
   Stack $old = (Stack) jContractorRuntime.popState();
   return implementation.contains(o) && (size() == $old.size() + 1);
}
```

Writing Contacts in OCL

- Object Constraint Language (OCL) is a specification language that supports specification of contracts (i.e., pre, post conditions and invariants) in UML class diagrams.

- OCL constraints have formal syntax and semantics
  - their interpretation is unambiguous

- OCL can be used to add precision to UML diagrams

- There are tools which check OCL constraints.:
  - USE (A UML-based Specification Environment)
    http://www.db.informatik.uni-bremen.de/projects/USE/
  - Enables analysis of UML diagrams before implementation