Design-by-Contract
Systems Thinking

• A **system** is any part of anything that you want to think about as an indivisible unit.

• An **interface** is a description of the “boundary” between a system and everything else, that also describes how to think about that system as a unit.

• A **subsystem (component)** is a system that is used inside, i.e., as a part of, another system — a relative notion!
Example: Ice/Water Dispenser

Select water, crushed ice, or cubed ice. Place a glass against the pad and push.
People’s Roles wrt Systems

• A *client* is a person (or a role played by some agent) viewing a system “from the outside” as an indivisible unit.

• An *implementer* is a person (or a role played by some agent) viewing a system “from the inside” as an assembly of subsystems/components.
Describing Behavior: Part 1

• One side of the coin: *information hiding* is a technique for describing system behavior in which you *intentionally leave out* “internal implementation details” of the system.
Describing Behavior: Part 2

- Other side of the coin (and a necessary consequence of information hiding): **abstraction** is a technique in which you create a *valid cover story* to counteract the effects of hiding some internal implementation details.
  - Presumably the hidden information is relevant to the system behavior, so even if you hide it you still need to account for its presence!
Overview of Design-by-Contract

• Also known as \textit{programming-to-the-interface}
• Articulated clearly only in the 1980s
• Design-by-contract has become \textit{the standard policy} governing “separation of concerns” across modern software engineering
• This is how software components are really used…
Recall: Mathematical Models

• Each variable in the program has a type
  – Examples: `int`, `double`, ...

• Each program type has a mathematical type that models it: you should think of any variable of that program type as having a value from its mathematical model’s mathematical space-domain
  – Examples (respectively): `integer`, `real`, ...
Informal Models

• Models are not always *formal* mathematical models like integers, real numbers, etc., but can be based on *informal* concepts from other situations

• Example of an *anthropomorphic* description of behavior:
  – “This TV *remembers* the last channel you watched.”

• More examples to come…
Structure of a Method Contract

• Each method has:
  – A precondition (requires clause) that characterizes the responsibility of the program that calls (uses) that method (client code)
  – A postcondition (ensures clause) that characterizes the responsibility of the program that implements that method (implementation code in the method body)
Meaning of a Method Contract

• If its precondition is true when a method is called, then the method will *terminate* — return to the calling program — and the postcondition will be true when it does return.

• If its precondition is not true when a method is called, then the method may do anything (including not terminate)
Responsibilities and Rewards

• Responsibility: Making sure the **precondition** is true when a method is called is the responsibility of the **client**

• Reward: The client may assume the postcondition is true when the method returns
Responsibilities and Rewards

• Responsibility: Making sure the postcondition is true when a method returns is the responsibility of the implementer

• Reward: The implementer may assume the precondition is true when the method is called
Recall: Static (Class) Methods

• A *static method* (class method) is one that:
  – Has zero or more *formal parameters* of various types — placeholders for the *arguments* that appear in the call between (…)
  – Returns a value of a particular *return type* to the calling program; or, returns nothing, denoted by a return type of *void*
• Example of a call and its *arguments*:
  ```
double a, b;
...
double c = sqrt (a*a + b*b, 0.001);
```
Recall: Static (Class) Methods

- A **static method** (class method) is one that:
  - Has zero or more **formal parameters** of various types — placeholders for the arguments that appear in the call between (...).
  - Returns a value of a particular **return type** to the calling program; or, returns nothing, denoted by a return type of **void**.

- Example of a call and its arguments:

  ```java
  double a, b;
  ...
  double c = sqrt (a*a + b*b, 0.001);
  ```

What does this method do? How do you know?
Example of a Contract

/**
 * ...  
 * @param x number to take the square root of 
 * @param epsilon allowed relative error 
 * @return the approximate square root of x
 * @requires 
 * x > 0 and epsilon > 0
 * @ensures <pre>
 * sqrt >= 0 and 
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 * </pre>
 */

private static double sqrt(double x,  
 double epsilon)
Example of a Contract

A Java comment that starts with the symbols /** is called a Javadoc comment; it goes before the method header.

```java
/**
 * ...
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @requires
 * x > 0 and epsilon > 0
 * @ensures <pre>
 * sqrt >= 0 and
 * [sqrt is within relative error epsilon of the actual square root of x]
 * </pre>
 */

private static double sqrt(double x, double epsilon)
```
Javadoc

• The standard documentation technique for Java is called **Javadoc**

• You place special **Javadoc comments** enclosed in /** ... */ in your code, and the **javadoc tool** generates nicely formatted web-based documentation from them
APIs

• The resulting documentation is known as the *API (application programming interface)* for the Java code to which the Javadoc tags are attached

• The API for the OSU CSE components is at:

  http://cse.osu.edu/software/common/doc/
APIs

- The resulting documentation is known as the **API (application programming interface)** for the Java code to which the Javadoc tags are attached.
- The API for the OSU CSE components is at: [http://cse.osu.edu](http://cse.osu.edu)

The word **interface** has two related but distinct meanings:
- A unit of Java code that contains Javadoc comments used to produce documentation.
- The resulting documentation.
Example of a Contract

```java
/**
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @requires
 * x > 0 and epsilon > 0
 * @ensures <pre>
 * sqrt >= 0 and
 * [sqrt is within relative error epsilon of the actual square root of x]
 * </pre>
*/

private static double sqrt(double x, double epsilon)
```

The **Javadoc tag** `@param` is needed for each formal parameter; you describe the parameter’s role in the method.
Example of a Contract

/**
 * ...
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @requires
 * x > 0 and epsilon > 0
 * @ensures <pre>
 * sqrt >= 0 and
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 * </pre>
 */

private static double sqrt(double x,
                          double epsilon)
Example of a Contract

```java
/**
 * ...*
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @requires
 * x > 0 and epsilon > 0
 * @ensures <pre>
 * sqrt >= 0 and
 * [sqrt is within relative error epsilon
 *  of the actual square root of x]
 * </pre>
 */

private static double sqrt(double x, 
   double epsilon)
```

The Javadoc tag `@requires` introduces the precondition for the method.
Example of a Contract

```java
/**
 * ...*
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @requires*
 * x > 0 and epsilon > 0
 * @ensures <pre>
 * sqrt >= 0 and
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 * </pre>
 */

private static double sqrt(double x,
   double epsilon)
```

The **Javadoc tag** @ensures introduces the postcondition for the sqrt method.
Example of Javadoc comments:

```java
/**
 * ...*
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @requires
 * x > 0 and epsilon > 0
 * @ensures <pre>
 * sqrt >= 0 and
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 * </pre>
 */

private static double sqrt(double x, double epsilon)
```

Javadoc comments may contain HTML-like tags; e.g., `<pre> ... </pre>` means spacing and line-breaks are retained in generated documentation.
Abbreviated Javadoc

• For this course:
  – Any actual code you see in *.java files will have the full Javadoc comments, as above
  – Some code you see in these slides will not have the Javadoc tags @param, @return, and formatting tags <pre>; plus, “keywords” in the Javadoc and mathematics will be bold-faced for easy reading
    • This allows you to focus on the contract content: the requires and ensures clauses themselves
Example Contract (Abbreviated)

/**
 * ...*
 * @requires
 * x > 0  and  epsilon > 0
 * @ensures
 * sqrt >= 0  and
 * [sqrt is within relative error epsilon
 *  of the actual square root of x]
 */

private static double sqrt(double x,
   double epsilon)
Example Contract (Abbreviated)

/**
 * ...
 * @requires
 * x > 0 and epsilon > 0
 * @ensures
 * sqrt >= 0 and [sqrt is within relative error epsilon of the actual square root of x]
 */

private static double sqrt(double x, double epsilon)

This is the precondition, indicating that the arguments passed in for the formal parameters x and epsilon both must be positive before a client may call sqrt.
Example Contract (Abbreviated)

```java
/**
 * ...*
 * @requires 
 * x > 0 and epsilon > 0 
 * @ensures 
 * sqrt >= 0 
 * [sqrt is within relative error epsilon 
 * of the actual square root of x]
 */

private static double
    sqrt(
        double x,
        double epsilon)
```

The precondition is a statement about the models of the arguments; here, it is a formal mathematical statement about mathematical reals.
Example Contract (Abbreviated)

/**
 * ...*
 * @requires
 * x > 0 and
 * @ensures
 * sqrt >= 0 and
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 */

private static double sqrt(double x, double epsilon)
Example Contract (Abbreviated)

/**
 * ... *
 * @requires
 *  x > 0  and
 * @ensures
 *  sqrt >= 0  and
 *  [sqrt is within relative error epsilon
 *   of the actual square root of x]
 */

private static double sqrt(double x, double epsilon)
Using a Method Contract

• A static method’s contract refers to its formal parameters, and (only if it returns a value, not \texttt{void}) to the name of the method (which stands for the return value)

• To determine whether the precondition and postcondition are true for a particular client call:
  – The model values of the \textit{arguments} are substituted for the respective formal parameters
  – The model value of the \textit{result returned by the method} is substituted for the method name
## Reasoning: Tracing Tables

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$y = 76.9$</td>
</tr>
<tr>
<td>$y = \sqrt{4.0, 0.01}$;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$y = 2.0$</td>
</tr>
</tbody>
</table>
## Reasoning: Tracing Tables

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$y = 76.9$</td>
</tr>
<tr>
<td></td>
<td>$z = 4.0$</td>
</tr>
<tr>
<td>$y = \sqrt{z, 0.01}$;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$y = 2.0$</td>
</tr>
<tr>
<td></td>
<td>$z = 4.0$</td>
</tr>
</tbody>
</table>
From the contract of `sqrt`, do we know that
\[ y = 2.0 \]
instead of
\[ y = -2.0 \]?
From the contract of `sqrt`, do we know that
\[ y = 2.0 \]
instead of
\[ y = 1.9996 \]?
A Partly Informal Contract

/**
 * ... 
 * @requires
 * x > 0 and epsilon > 0
 * @ensures
 * sqrt >= 0 and
 * [sqrt is within relative error epsilon of the actual square root of x]
 */

private static double sqrt(double x, double epsilon)
A Formal Contract

/**
 * ... *
 * @requires
 * x > 0 and epsilon > 0
 * @ensures
 * sqrt >= 0 and
 * |sqrt - x^(1/2)| / x^(1/2) <= epsilon
 */

private static double sqrt(double x, double epsilon)
A Formal Contract

/**
 * ...*
 * @requires
 * x > 0 and epsilon > 0
 * @ensures
 * sqrt >= 0 and
 * |sqrt - x^(1/2)| / x^(1/2) <= epsilon
 */

private static double sqrt(double x, double epsilon)
private static double sqrt(double x, double epsilon) {
    assert x > 0.0 :
        "Violation of: x > 0";
    assert epsilon > 0.0 :
        "Violation of: epsilon > 0";
    // rest of body: compute the square root
}
A Method Body

```java
private static double sqrt(double x,
    double epsilon) {
    assert x > 0.0 :
        "Violation of: x > 0";
    assert epsilon > 0.0 :
        "Violation of: epsilon > 0";
    // rest of body
}
```

The `assert` statement in Java checks whether a condition (an assertion) is true; if it is not, it stops execution and reports the message after the colon.
A Method Body

private static double sqrt(double x, double epsilon) {
    assert x > 0.0 : "Violation of: x > 0";
    assert epsilon > 0.0 : "Violation of: epsilon > 0";
    // rest of body
}

But why are there assert statements in this method body to check what the implementer is supposed to assume?
Checking a Precondition

• During *software development*, it is a **best practice** to check assumptions with **assert** when it is easy to do so
  – This checking can be turned on and off (on by using the “-ea” argument to the JVM)
  – When turned off, **assert** is documentation

• Preconditions generally are easy to check; postconditions generally are not easy to check
A Misconception

• A common misconception is that using `assert` statements to check preconditions contradicts design-by-contract principles.

• It does not, because the advice is not to `deliver` software with assertion-checking turned on, but rather to `develop` software with assertion-checking turned on — to help catch your mistakes, not the client’s!
Resources

- Wikipedia: Design by Contract