# CS 4500
# Software Development

## Design by Contract

Ferdinand Vesely

October 22, 2019

```
def f(x):
  ...
```

```
def f(x):
  ...
```

What can I pass as an argument?
What can I expect as output?

```
def f(x):
  ...
```

What can I pass as an argument?
What can I expect as output?

```
f(20)
f(-12)
f(3.14)
f("Hello World")
f([1, 30, 3.14,])
```

```
def f(x):
  return g(x) + 1
```

What can I pass as an argument? What can I expect as output?

```
f(20)
f(-12)
f(3.14)
f("Hello World")
f([1, 30, 3.14,])
```

# Comments?

```
def f(x):
  """...Expects a number, returns a number..."""
  return g(x) + 1
```

What can I pass as an argument? What can I expect as output?

```
f(20)
f(-12)
f(3.14)
f("Hello World")
f([1, 30, 3.14,])
```

# Comments?

```
def f(x):
  """...Expects a number, returns a number..."""
  return g(x) + 1
```

What can I pass as an argument? What can I expect as output?

```
f(20)
f(-12)
f(3.14)
f("Hello World")
f([1, 30, 3.14,])
```

- Unchecked

# Types!

```
def f(x: float) -> float:
  return g(x) + 1
```

What can I pass as an argument? What can I expect as output?

```
f(20)
f(-12)
f(3.14)
f("Hello World")
f([1, 30, 3.14,])
```

```
def g(x):
    if x < len(sqrt_table):
        return sqrt_table[x]
    else:
        return math.ceil(math.sqrt(x))
```

```
def f(x: float) -> float:
  return g(x) + 1
```

What can I pass as an argument? What can I expect as output?

```
f(20)
f(-12)
f(3.14)
f("Hello World")
f([1, 30, 3.14,])
```

More accurate types:

```
def g(x: int) -> int:
    if x < len(sqrt_table):
        return sqrt_table[x]
    else:
        return math.ceil(math.sqrt(x))
```

```
def f(x: int) -> int:
  return g(x) + 1
```

What can I pass as an argument? What can I expect as output?

```
f(20)
f(-12)
f(3.14)
f("Hello World")
f([1, 30, 3.14,])
```

## Contracts:

```
@contract
def g(x 'int, >=0') -> 'int, >=0':
    if x < len(sqrt_table):
        return sqrt_table[x]
    else:
        return math.ceil(math.sqrt(x))
```

```
@contract
def f(x: 'int, >=0') -> 'int, >= 1':
  return g(x) + 1
```

What can I pass as an argument? What can I expect as output?

```
f(20)
f(-12)
f(3.14)
f("Hello World")
f([1, 30, 3.14,])
```

Contracts:

```
@contract
def f(x: 'int, >=0') -> 'int, >= 1':
  return g(x) + 1
```

```
f(20)
f(-12)
```

```
...ContractNotRespected: Breach for argument 'x' to f().
Condition -12 >= 0 not respected
checking: >=0       for value: Instance of <class 'int'>: -12
checking: int,>=0   for value: Instance of <class 'int'>: -12
```

# Contracts

- Between the caller and the callee
- Serve as documentation
- Enforceable
- Design principle: design by contract
- Originated in Eiffel

# In Eiffel

```
set_second (s: INTEGER)
    require
        valid_argument_for_second: 0 <= s and s <= 59
    do
        second := s
    ensure
        second_set: second = s
    end
```

- Class invariants:

```
invariant
    hour_valid: 0 <= hour and hour <= 23
    minute_valid: 0 <= minute and minute <= 59
    second_valid: 0 <= second and second <= 59
```

# Contracts

## Preconditions

- The routine's requirements:
  *What must be true for the routine to be called.*

# Contracts

**Preconditions**

- The routine's requirements:
  *What must be true for the routine to be called.*

**Postconditions**

- What the routine is guaranteed to do
- The state of the world after it's done

# Contracts

**Preconditions**

- The routine's requirements:
  *What must be true for the routine to be called.*

**Postconditions**

- What the routine is guaranteed to do
- The state of the world after it's done

**Class Invariants**

- Always true from the perspective of a caller
- Might not be true during internal processing of a routine
- After routine finishes and returns control, must be true

# Contracts

- If either party breaks the contract – remedy: exception, program terminates, etc.
- Always means a failure, bug
- Thus: Should not be used for, e.g., validating user input

# Preconditions & Postconditions

If precondition is true when a routine is called, then the routine **will terminate** and the postcondition (and class invariant) will be true when it returns

If precondition is false when a routine is called, then the routine may do anything (including not terminate)

# Responsibilities and Rewards

**Caller**

RESPONSIBILITY: Ensure that the argument is in the domain of the callee

- Usually has more domain/higher-level knowledge – more options to handle invalid arguments
- Not callee's problem

REWARD: May assume the postcondition is true when callee returns

# Responsibilities and Rewards

**Callee**

RESPONSIBILITY: Ensure postconditions (and invariant) are true when input is in the domain

REWARD: May assume the precondition is true when called

# Contracts: Principles

- Preconditions: Strict in what you accept
- Postconditions: Promise as little as possible
- Write *before* implementation – like signatures
- Can guide unit tests
- Crash early

# Origin / Foundations: Hoare Logic

- Hoare triples:

$$\{ P \} S \{ Q \}$$

- $P$ – precondition
- $S$ – program
- $Q$ – postcondition
- if we start in a state satisfying $P$, then, after executing $S$, we end up in a state satisfying $Q$
- proof calculus

# Hoare Triple Examples

- $\{\texttt{true}\}\ \texttt{x = 5}\ \{x = 5\}$
- $\{x = y\}\ \texttt{x = x + 3}\ \{x = y + 3\}$
- $\{x > 0\}\ \texttt{x = x * 2}\ \{x \geqslant 2\}$
- $\{x = a\}\ \texttt{if (x < 0): x = -x}\ \{x = |a|\}$

# Contracts and Inheritance

- With a proper OO implementation, contracts get inherited
- Inheritance / overriding – how?

# Contracts and Inheritance

- With a proper OO implementation, contracts get inherited
- Inheritance / overriding – how?
- Inherit contracts by default
- Usually:
  - weaken preconditions
  - strengthen postconditions

# Language Support

…varies

**Native:**

- Eiffel
- Clojure
- Kotlin (?)
- Racket
- D
- Scala
- …

# Language Support: Clojure

```
(defn limited-sqrt [x]
    {:pre  [(pos? x)]
     :post [(>= % 0), (< % 10)]}
    (Math/sqrt x))

(limited-sqrt 9)   ;; 3.0
(limited-sqrt -9)  ;; AssertionError Assert failed: (pos? x)
(limited-sqrt 144) ;; AssertionError Assert failed: (< % 10)
```

# Language Support: Scala

```scala
def addNaturals(nats: List[Int]): Int = {
  require(nats forall (_ >= 0),
          "List contains negative numbers")
  nats.foldLeft(0)(_ + _)
} ensuring(_ >= 0)
```

# Language Support

**Library-based:**

- Java
- Ruby
- Javascript
- Rust
- Python
- …

# Language Support: Java Modeling Language

```
public class Date {
    int /*@spec_public@*/ day;
    int /*@spec_public@*/ hour;

    /*@invariant 1 <= day && day <= 31; @*/
    /*@invariant 0 <= hour && hour < 24; @*/

    /*@
      @requires 1 <= d && d <= 31;
      @ensures day == d;
      @*/
    public void setDay(int d) {
        day = d;
    }
    ...
```

```
    /*@
```

# Alternatives

## Comments

# Alternatives

**Comments**

- DBC intended as a *design* principle
- Still provides value: planning, thinking before coding
- Connection to actual code not enforced, unchecked

# Alternatives

**Assertions**

- Common
- Many languages some form of `assert`
- Compiler switch to ignore asserts in production code
- Preconditions: first lines of the function/method body
- Postconditions + class invariants: last lines just before return

# Contracts via Assertions

```
def g(x):
    assert x >= 0           # precondition
    if x < len(sqrt_table):
        result = sqrt_table[x]
    else:
        result = math.ceil(math.sqrt(x))
    assert result >= 0      # postcondition
    return result

def f(x):
    assert x >= 0       # precondition
    result = g(x) + 1
    assert result >= 1  # postcondition
    return result
```

# Contracts via Assertions

```
def g(x):
    assert x >= 0          # precondition
    if x < len(sqrt_table):
        result = sqrt_table[x]
    else:
        result = math.ceil(math.sqrt(x))
    assert result >= 0     # postcondition
    return result

def f(x):
    assert x >= 0       # precondition
    result = g(x) + 1
    assert result >= 1  # postcondition
    return result
```

What about types?

# Contracts via Assertions

```
def g(x):
    assert type(x) is int and x >= 0       # precondition
    if x < len(sqrt_table):
        result = sqrt_table[x]
    else:
        result = math.ceil(math.sqrt(x))

    assert type(x) is int and result >= 0  # postcondition
    return result

def f(x):
    assert type(x) is int and x >= 0       # precondition
    result = g(x) + 1
    assert type(x) is int and result >= 1  # postcondition
    return result
```

# Contracts via Assertions

What's missing?

# Limitations of Assertions for Contracts

- Inheritance
- Class invariants: call before exiting every method
- No support for original (“old”) values in postconditions – need to save explicitly
- Readability

# Assertions – in General

- To “prevent the impossible”
- Check for things that should never happen
- Never: in place of error handling
- Useful for debugging
- Careful about side effects:

```
while (iter.hasNext()) {
  assert(iter.next() != null);
  Object obj = iter.next();
  ...
```

# Types vs. Contracts

## Type checking

- Usually static
  - Compile-time, or
  - External tools – e.g., MyPy for Python
- Somewhat limited level of detail
- Refinement types

# Types vs. Contracts

## Type checking

- Usually static
  - Compile-time, or
  - External tools – e.g., MyPy for Python
- Somewhat limited level of detail
- Refinement types

## Contracts

- Richer property language
- Many implementations: dynamic
- Some tools allow static checking
- Generate proof obligations, use a constraint solver
- e.g., the Java Modeling Language, Ada (SPARK)
- Limitations?