

Contracts Reloaded

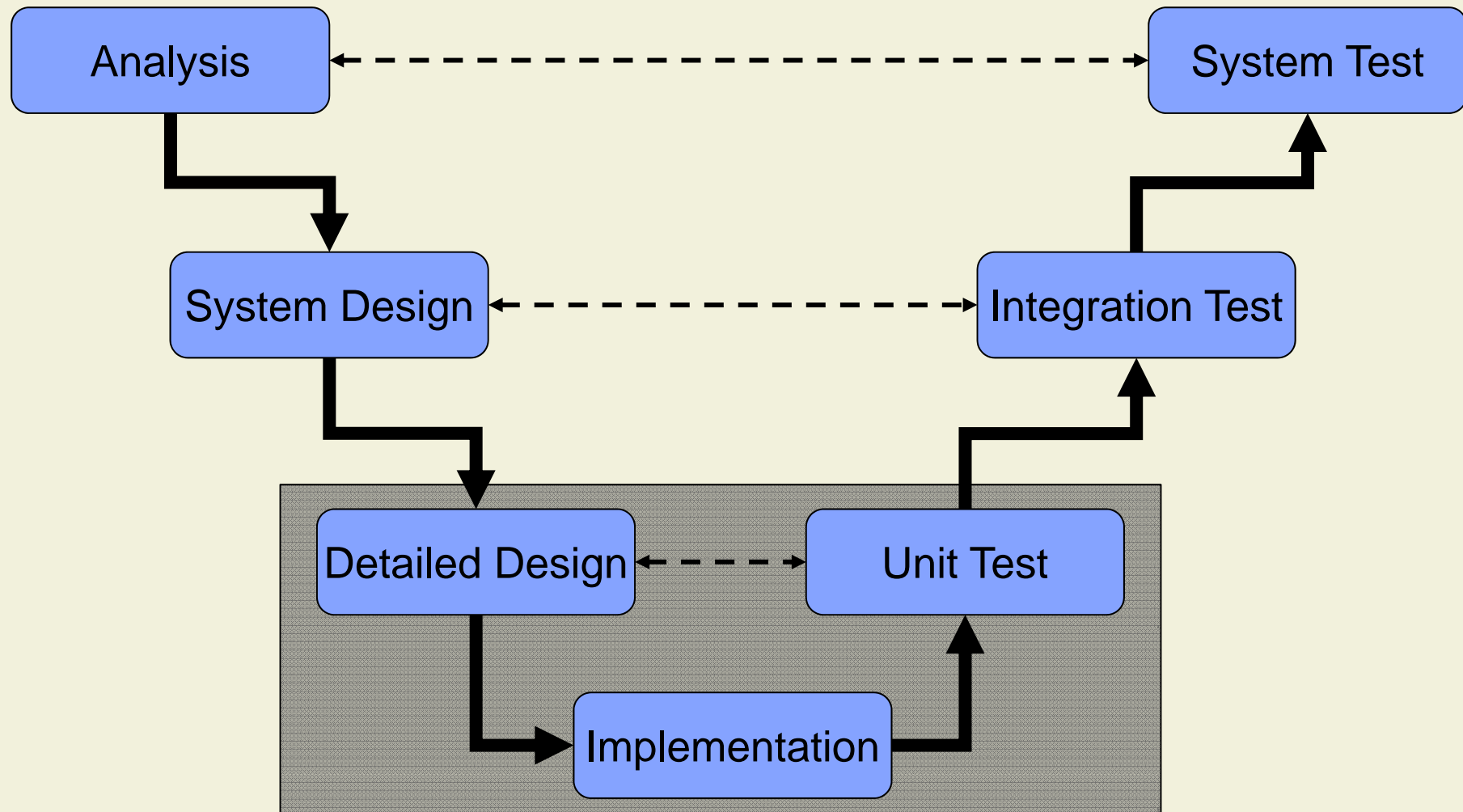
Peter Müller

Chair of Programming Methodology

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Test Stages



Unit Test Example

```
public class AbsoluteValue {  
    public static int Abs( int x ) {  
        if ( x < 0 )  
            return -x;  
        else  
            return x;  
    }  
}
```

```
[ TestMethod() ]  
public void AbsTest( ) {  
    int x = -12;  
    int expected = 12;  
    int actual;  
    actual = AbsoluteValue.Abs( x );  
    Assert.AreEqual( expected, actual );  
}
```

Implement
test driver

Create
test data

Implement
test oracle

The Code Contracts Approach

- **Specification**: document intended behavior of method using contracts

```
Contract.Requires( true );  
Contract.Ensures( 0 <= Contract.Result<int>() );  
  
if ( x < 0 )  
    return -x;  
else  
    return x;
```

Any input
is allowed

Result must be
non-negative

- **Testing**: generate unit tests automatically from contracts
- **Static checking**: try to find as many errors as possible at compile time

Overview

1. Code Contracts
2. Unit Testing with Pex
3. Static Checking with Clousot

Design by Contract

- Code Contracts are based on «**Design by Contract**», pioneered by Bertrand Meyer in the Eiffel language
- Basic idea: a method and its callers agree on a contract, expressed via a precondition and a postcondition



	Obligation	Benefit
Method	Establish postcondition for all calls that satisfy precondition	May assume precondition upon method entry
Caller	Establish precondition of called method	May assume postcondition upon return from call

Benefits over Assertions

- Documentation
 - Contracts are visible to caller
 - Clear intention
- Conciseness
 - Contracts avoid code duplication
- Inheritance
 - Contracts relate super- and subclasses
- Abstract methods
 - Contracts can be used in interfaces and abstract methods

```
if ( x < 0 ) {  
    int res = -x;  
    Debug.Assert( 0 <= res );  
    return res;  
} else {  
    Debug.Assert( 0 <= x );  
    return x;  
}
```

```
Contract.Requires( true );  
Contract.Ensures( 0 <=  
    Contract.Result<int>() );  
  
if ( x < 0 )  
    return -x;  
else  
    return x;
```

Code Contracts

- Idea: Use code to specify code

```
public static int Abs( int x ) {  
    Contract.Requires( true );  
    Contract.Ensures( 0 <= Contract.Result<int>() );  
  
    if ( x < 0 )  
        return -x;  
    else  
        return x;  
}
```

- Contracts are static method calls
 - Class Contract in System.Diagnostics.Contracts
- Conditions are boolean expressions

Quantifiers

- Expressing properties of arrays and collections often requires quantification

```
public static bool Contains( ICollection<string> students, string name ) {  
    Contract.Requires( students != null && name != null );  
    Contract.Requires( Contract.ForAll<string>(students, s => s != null) );  
    Contract.Ensures(  
        Contract.Result<bool>() ==  
        Contract.Exists<string>( students, s => s.Equals(name) )  
    );  
    foreach ( string s in students )  
        if ( s.Equals(name) )  
            return true;  
    return false;  
}
```

Collection

Predicate

Overview

1. Code Contracts
2. Unit Testing with Pex
3. Static Checking with Clousot

Unit Testing Revisited

```
public void Withdraw( int amount ) {
```

```
    balance -= amount;  
}
```

Implement
test driver

```
[ TestMethod() ]
```

```
public void WithdrawTest() {
```

```
    SavingsAccount target = new SavingsAccount();
```

```
    target.Deposit( 300 );
```

```
    int amount = 100;
```

```
    target.Withdraw( amount );
```

```
    Assert.AreEqual( target.Balance, 200 );
```

```
}
```

Create
test data

Implement
test oracle

Unit Testing Revisited

```
public void Withdraw( int amount ) {  
    Contract.Requires( 0 <= amount );  
    Contract.Requires( amount <= Balance );  
    Contract.Ensures( Balance == Contract.OldValue( Balance ) - amount );  
  
    balance -= amount;  
}
```

```
[ TestMethod() ]  
public void WithdrawTest() {  
    SavingsAccount target = new SavingsAccount();  
    target.Deposit( 300 );  
    int amount = 100;  
    target.Withdraw( amount );  
    Assert.AreEqual( target.Balance, 200 );  
}
```

Can we use
precondition to
generate test data?

Postcondition
replaces test
oracle

Parameterized Unit Tests

- Parameterized test methods take arguments for test data
 - Decouple test driver (logic) from test data

```
[ Test ]  
public void WithdrawTest( int balance, int amount ) {  
    SavingsAccount target = new SavingsAccount();  
    target.Deposit( balance );  
    target.Withdraw( amount );  
}
```

- Traditionally, tester provides possible values
 - Goal: generate test data automatically

Test Data: Code Coverage

- Standard quality criterion for unit tests is **coverage**
 - Path coverage:
Execute each possible path (not practical)
 - Branch coverage:
test each possible outcome from a condition

```
public static string ParseLines( string[ ] lines ) {  
    for ( int i = 0; i < lines.Length; i++ ) {  
        string line = lines[ i ];  
        int index = line.IndexOf( '=' );  
        string key = line.Substring( 0, index );  
        if( key.Equals( "Foo" ) )  
            return line.Substring( index+1 );  
    }  
    return "??";  
}
```

```
public void ParseTest1() {  
    string[ ] l = { "Foo=Y" };  
    string res = ParseLines( l );  
    Assert.AreEqual( res, "Y" );  
}
```

```
public void ParseTest2() {  
    string[ ] l = { "Bar=X" };  
    string res = ParseLines( l );  
    Assert.AreEqual( res, "??" );  
}
```

Test Data: Assertion Coverage

- Code coverage does not ensure that errors are revealed

```
public static string ParseLines( string[] lines ) {  
    for ( int i = 0; i < lines.Length; i++ ) {  
        string line = lines[ i ];  
        int index = line.IndexOf( '=' );  
        string key = line.Substring( 0, index );  
        if( key.Equals( "Foo" ) )  
            return line.Substring( index+1 );  
    }  
    return "??";  
}
```

Null pointer?

Precondition?

Null pointer?

Precondition?

- We also need to cover as many assertions as possible (explicit and implicit)

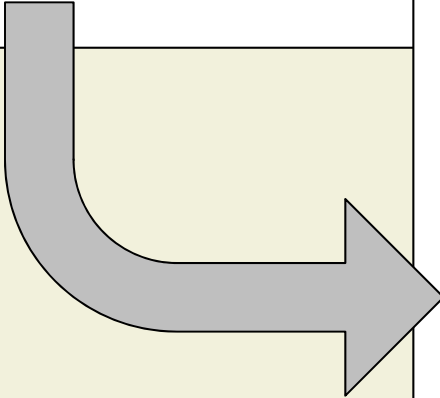
Concolic Testing

- Concolic testing combines **concrete** program executions with **symbolic** reasoning
- **Compute constraints** on input data that force method to take a particular branch
- Let **constraint solver** provide concrete input data that satisfies the constraints
- Pex is a concolic testing tool for .NET
 - Pex uses (but does not require) Code Contracts



Step 1: Make Assertions Explicit

```
public static int Abs( int x ) {  
    Contract.Ensures( 0 <= Contract.Result<int>() );  
  
    if ( x < 0 )  
        return -x;  
    else  
        return x;  
}
```



```
public static int Abs( int x ) {  
    if ( x < 0 )  
        if ( 0 <= -x )  
            return -x;  
        else  
            throw new ContractException( ... );  
    else  
        if ( 0 <= x )  
            return x;  
        else  
            throw new ContractException( ... );  
}
```

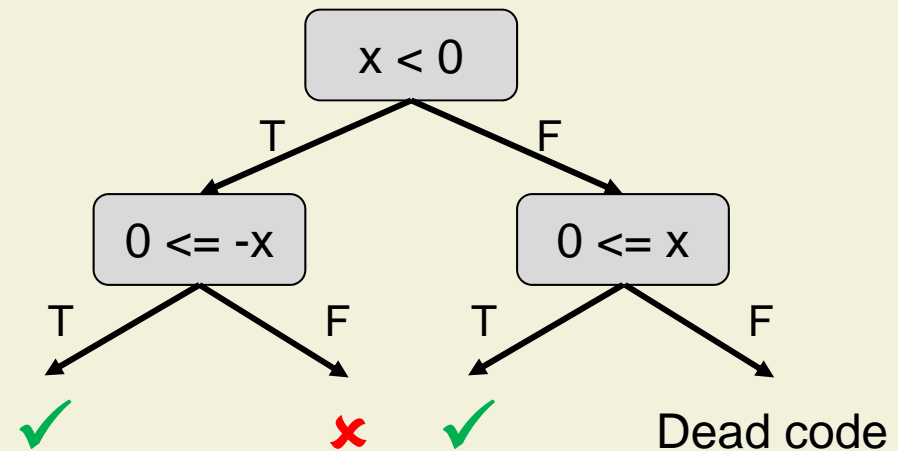
Step 2: Constraint Generation

```

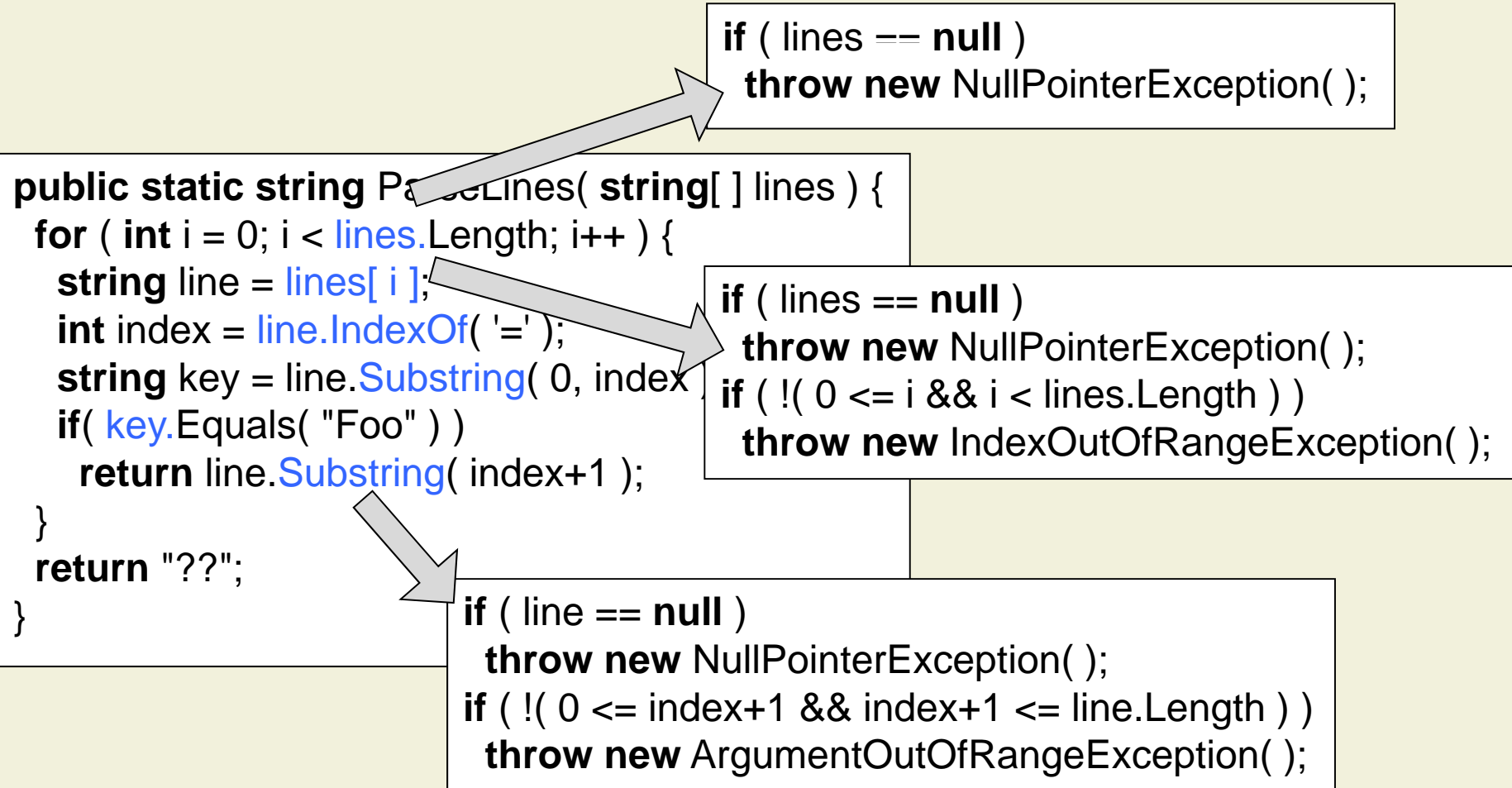
public static int Abs( int x ) {
  if ( x < 0 )
    if ( 0 <= -x )
      return -x;
    else
      throw new ContractException( ... );
  else
    if ( 0 <= x )
      return x;
    else
      throw new ContractException( ... );
}

```

Constraint to solve	Concrete data	Observed constraint
	0	$!x < 0 \ \&\& \ 0 \leq x$
$!x < 0 \ \&\& \ !0 \leq x$	No solution	
$x < 0$	-1	$x < 0 \ \&\& \ 0 \leq -x$
$x < 0 \ \&\& \ !0 \leq -x$	Int32.MinValue	



Preventing Runtime Errors



Step 3: Test Data Generation

- Turning solutions for constraints into concrete data is relatively simple for integers, arrays, and strings
- Limitations
 - Small number of loop iterations
 - Theories not understood by constraint solver (for instance, float)
 - Instrumentation of called methods

```
public static int Power( int n ) {  
    Contract.Requires( 0 <= n );  
    Contract.Ensures(  
        Contract.Result<int>() != 256 );  
  
    int res = 1;  
    for ( int i = 0; i < n; i++ ) res = res * 2;  
    return res;  
}
```

```
public static void Y2kParser( string s ) {  
    if ( DateTime.Parse(s).Year == 2000 )  
        throw new Exception( "found it" );  
}
```

Concolic Testing: Summary

Strengths

- High degree of **automation** (human effort limited to writing contracts and factories)
- Very **high code and assertion coverage**
- Effective testing of parameter validation

Weaknesses

- Limited by constraint solver (e.g., floats)
- Expressing tests via contracts is sometimes difficult

Overview

1. Code Contracts
2. Unit Testing with Pex
3. Static Checking with Clousot

Motivation

- Move runtime errors to compile time
- Classical example: Static type checking
 - Python and Smalltalk throw “message not understood” errors when an object does not have the called method
 - .NET’s type system prevents this error statically
- Goal: Extend static checking to other errors
 - InvalidCast, NullPointer, IndexOutOfBounds, etc.
 - Assertions, postconditions, etc.

Abstract Interpretation

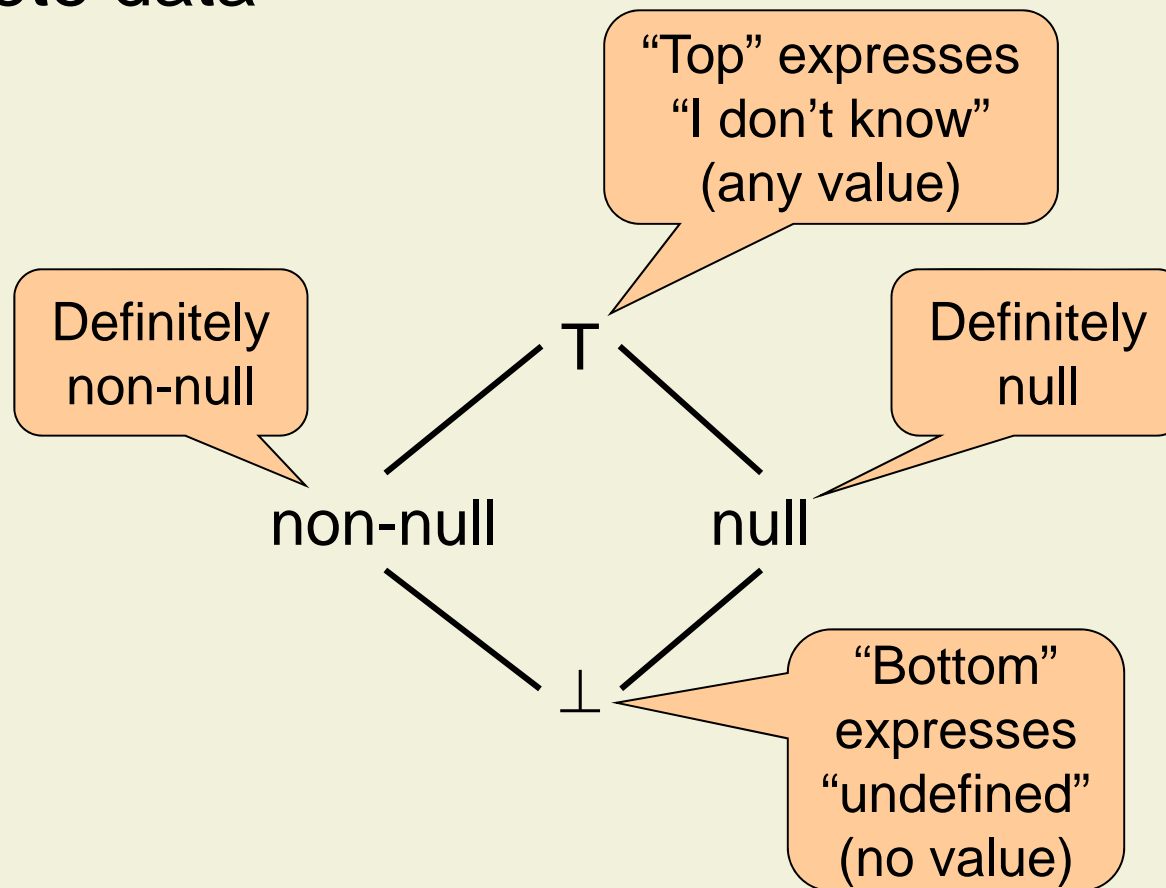
- Methods have too many possible executions to check each individually

```
public static void MyAssert ( bool cond, string msg ) {  
    string log;  
    if ( cond )  
        log = "OK";  
    else  
        log = msg;  
  
    log = log.Trim();  
    Console.WriteLine( log );  
}
```

- Idea: Abstract concrete data and check property on the abstraction

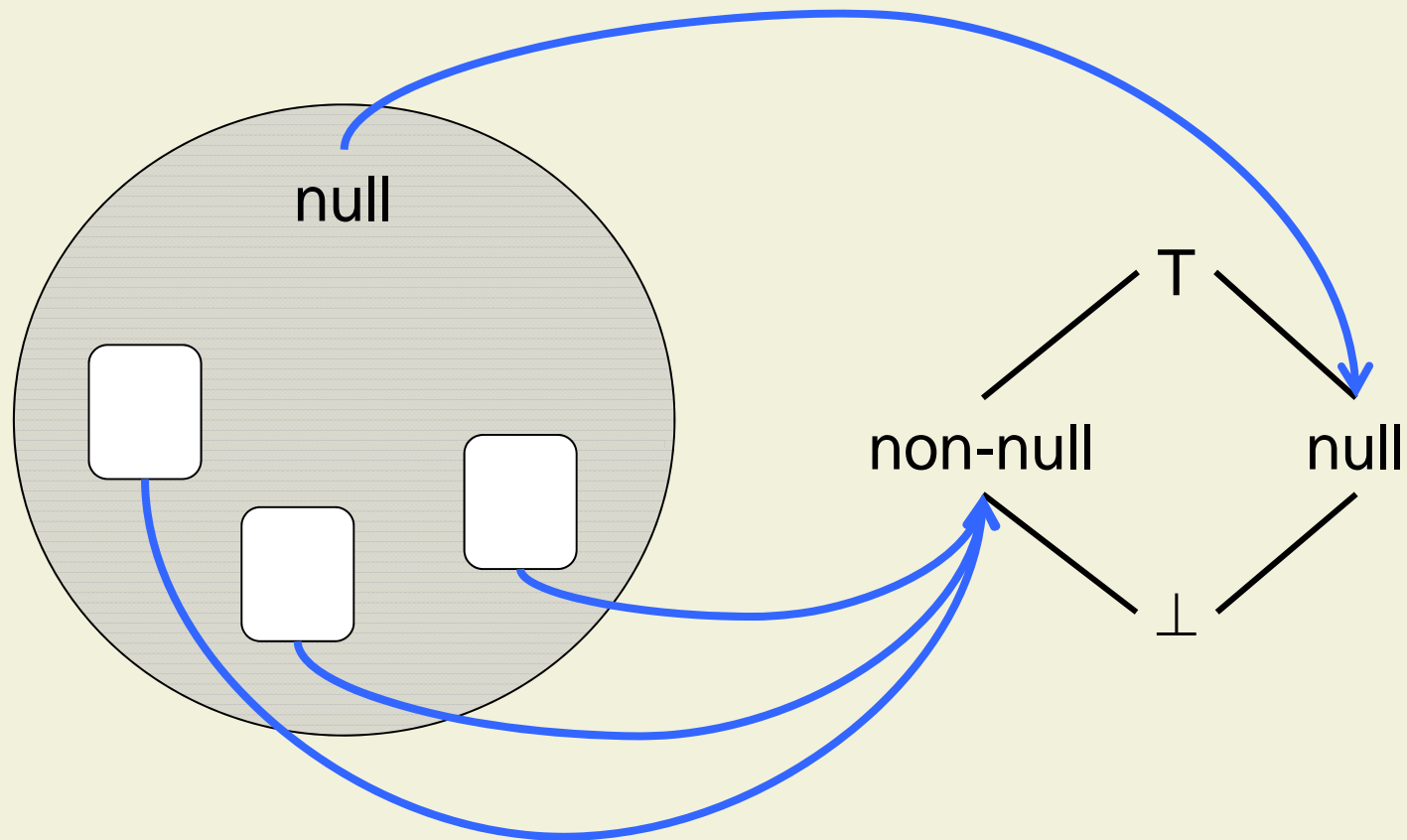
Abstract Domain

- Abstract domains capture some abstraction of concrete data



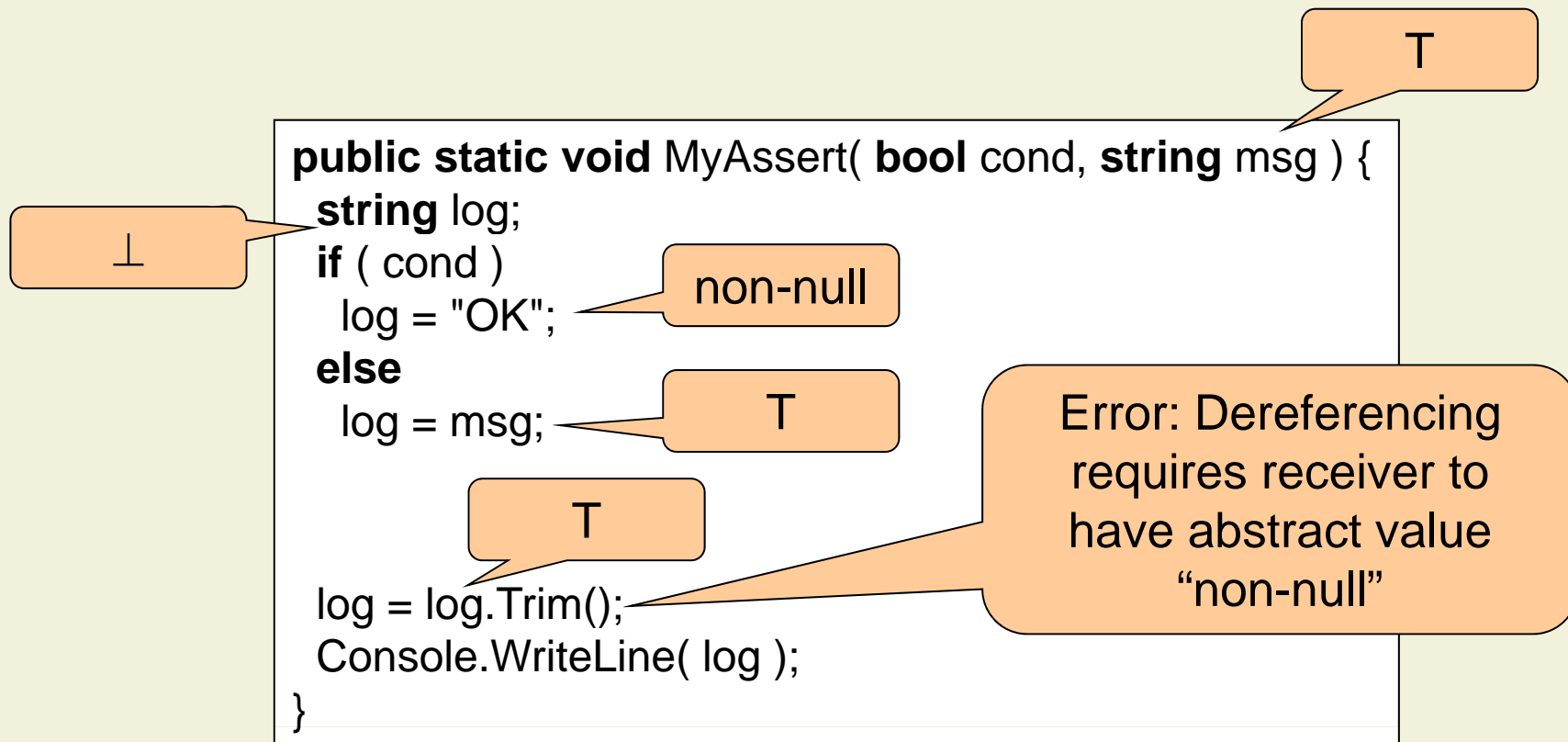
Abstraction

- Each concrete value can be mapped to an abstract value



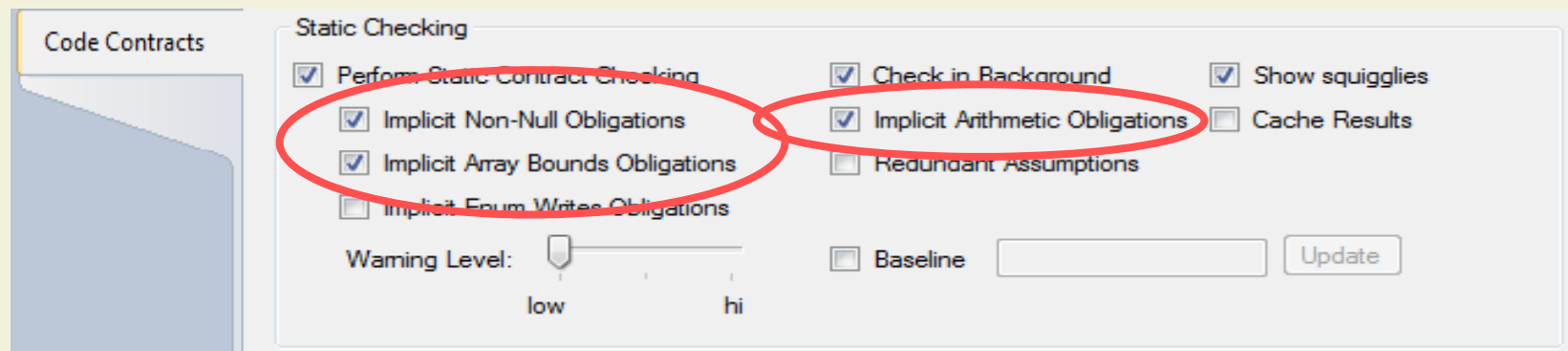
Abstract Semantics

- Abstract semantics describes the execution of the program in terms of abstract values



Static Checks

- Clousot tracks information about
 - Non-nullness of references
 - Numerical values
(intervals, linear equalities and inequalities)
 - Array and collection contents
- It attempts to check runtime errors as well as contract violations (similar to Pex)



Example: Contract Validation

```
public static string ParseLines( string[ ] lines ) {  
    Contract.Ensures( Contract.Result<string>() != null );  
  
    for ( int i = 0; i < lines.Length; i++ ) {  
        string line = lines[ i ];  
        int index = line.IndexOf( '=' );  
        string key = line.Substring( 0, index );  
        if( key.Equals( "Foo" ) )  
            return line.Substring( index+1 );  
    }  
    return null;  
}
```

Check
precondition

Check
postcondition

Understanding Warnings

- Clousot produces **false positives**
 - When Clousot **does not track the information** required for the check (e.g., string contents)
 - When the **abstraction is too coarse** to check the property (e.g., disjunctions, loops)

- Clousot produces **false negatives**
 - Heap abstraction
 - Object invariants
 - Unchecked exceptional paths

Static Contract Checking: Summary

Strengths

- Finds many program errors at compile time
 - Especially common bugs like null-pointer-dereferencing and array-index-out-of-bounds
- Checks methods for all possible inputs
- High degree of automation

Weaknesses

- Analysis may produce spurious errors
- Analysis may miss errors
- Unit testing is still necessary
 - To detect errors missed by static checker
 - To check functional correctness

Summary

- Code Contracts bring **Design-by-Contract** to mainstream programming languages
 - **Library solution** enables smooth integration
 - **Runtime checking** is useful for testing, including **integration testing**
- Pex automates unit testing
 - Automatic generation of **parameterized unit tests**
 - Automatic generation of **test data** (concolic testing)
- Clousot finds errors **at compile time**
 - Checks for runtime errors and contracts

Kompaktkurs
**Qualitätssicherung in .NET
mit Code Contracts**

20. Januar 2012

www.pm.inf.ethz.ch/education/compact