COMP 401
ASSERTIONS

Instructor: Prasun Dewan
PREREQUISITE

- Documentation Assertions
- Composite Visitors
PREVENTING INVALID BMI
public class ABMISpreadsheet {
    double height, weight;

    public ABMISpreadsheet(
            double theInitialHeight, double theInitialWeight) {
        setHeight (theInitialHeight);
        setWeight (theInitialWeight);
    }

    public double getHeight() { return height; }
    public void setHeight(double newHeight) { height = newHeight; }
    public double getWeight() { return weight; }
    public void setWeight(double newWeight) { weight = newWeight; }
    public double getBMI() { return weight/(height*height); }
}
public class ABMISpreadsheet {
    double height, weight;

    public ABMISpreadsheet(
            double theInitialHeight, double theInitialWeight) {
        setHeight (theInitialHeight);
        setWeight(theInitialWeight);
    }
    ...
    public boolean preGetBMI() {
        return weight > 0 && height > 0;
    }

    public double getBMI() {
        assert preGetBMI();
        return weight/(height*height);
    }
}
**JAVA ASSERTIONS/Pre(Post)Conditions**

- `assert <Boolean Expression>`
- `assert <Boolean Expression>: <Value>`

Statement can be inserted anywhere to state that some condition should be true.

If condition is false, Java throws `AssertionError`, which may be caught by programmer code.

If uncaught, depending on which `assert` used:
  - generic message saying assertion failed printed
  - `<Value>.toString()` printed

An assertion made at the beginning/end of a statement block (method, loop, if ..) is called its precondition/postcondition.
ASSERTIONS

- Declare some property of the program
  - Before getBMI() is called, height and weight should be greater than 0
Compile time vs. runtime properties

- Some assertions are language-supported
  - Compile time
    - String s = nextElement()
    - @Override
  - Runtime
    - ((String) nextElement())
    - @util.annotations.ObserverRegisterer(util.annotations.ObserverTypes.VECTOR_LISTENER)
      - addVectorListener(VectorListener)

- We will consider runtime properties.
- Casting is application-independent.
APPLICATION-INDEPENDENT VS. DEPENDENT

- Language can provide us with fixed number of application-independent assertions.
- Cannot handle
  - First character of String is a letter.
  - Letter concept not burnt into language.
    - Class Character defines it
  - Innumerable assertions about letters possible
    - Second elements of string is letter.
    - Third element of string is letter.
- Need mechanism to express arbitrary assertions.
- Originally Java had no assertions.
- In 1.4, assertions were added
**WHY LANGUAGE SUPPORT**

- Can always define a library with `assert(<Boolean Expression>)` method that throws a special exception denoting assertion error.

- Assertions can be dynamically turned on or off for package or Class
  - `java -ea assignment11.MainClass -da bus.uigen...`

```java
class MainClass {
    public void myAssert (boolean boolExp, String message) throws AssertionError {
        if (!boolExp) throw new AssertionError (message);
    }
}
```
**Error vs. Exception**

- Java assertion failure results in `AssertionError`
- Subclass of `Error` rather than `RunTimeException`
- **Reasoning:**
  - Convention dictates that `Exception` should be caught
  - Should “discourage programmers from attempting to recover from assertion failures.”
    - Might do custom reporting, mail error report etc.
  - `AssertionError` is a subclass of `Throwable` and can indeed be caught
  - Decision was controversial
Assertion Uses

- Potentially useful for
  - specification
  - testing
  - formal correctness
  - documentation
  - user-interface automation
public class ABMISpreadsheet {
    double height, weight;

    public ABMISpreadsheet(
            double theInitialHeight, double theInitialWeight) {
        setHeight (theInitialHeight);  
        setWeight (theInitialWeight);
    }

    ...  

    public boolean preGetBMI() {
        return weight > 0 && height > 0;
    }

    public double getBMI() {
        assert (preGetBMI());
        return weight/(height*height);
    }
}
OBJECTEDITOR USES PRECONDITIONS

Parameters of Create ABMISpreadsheet

- File
- Parameter 1: double 1.77
- Parameter 2: double 75
- Create ABMISpreadsheet

[ABMISpreadsheet]

- File  Edit  View  Customize  ABMISpreadsheet
- Height: 1.77
- Weight: 75.0
- BMI: 23.93948099205209
OBJECTEDITOR USES PRECONDITION
The menu item for a method is disabled when its precondition not met.
public class ABMISpreadsheet {
    double height, weight;
    double initialHeight, initialWeight;

    public ABMISpreadsheet(
            double theInitialHeight, double theInitialWeight) {
        setHeight (theInitialHeight);
        setWeight(theInitialWeight);
        initialHeight = theInitialHeight;
        initialWeight = theInitialWeight;
    }

    ...}

    public boolean preGetBMI() { return weight > 0 && height > 0; }
    public double getBMI() {
        assert preGetBMI(); return weight/(height*height); }

    public boolean preRestoreHeightAndWeight() {
        return height != initialHeight || weight != initialWeight; }
    public void restoreHeightAndWeight() {
        assert preRestoreHeightAndWeight();
        height = initialHeight;
        weight = initialWeight;
    }
}
PRECONDITIONS OF OTHER METHODS

```java
public class ABMISpreadsheet {

    ... 

    public double getWeight() {
        return weight;
    }

    public void setWeight(double newWeight) {
        weight = newWeight;
    }

    ...
}
```
Preconditions of Other Methods

```
public class ABMISpreadsheet {
    ...
    public double preGetWeight() {return weight > 0;}
    public double getWeight() {
        assert preGetWeight();
        return weight;
    }
    public boolean preSetWeight(double newWeight) {
        return newWeight > 0;
    }
    public void setWeight(double newWeight) {
        assert preSetWeight(newWeight);
        weight = newWeight;
    }
    ...
}
```

Prevention of getter not needed if setter and constructor prevent assignment of illegal values
public class ABMISpreadsheet {
    ... 
    public double getWeight() {
        return weight;
    }
    public boolean preSetWeight(double newWeight) {
        return newWeight > 0;
    }
    public void setWeight(double newWeight) {
        assert preSetWeight(newWeight);
        weight = newWeight;
    }
    ...
} 

Prevention of getter not needed if setter and constructor prevent assignment of illegal values
PRECONDITION STYLE RULE

- If there are constraints on the input of a method M(...) that may not be met, write a precondition boolean method, preM(...) for it.
- Call the precondition method in an assert statement as the first statement of M(..)
- To keep examples short, preconditions will not be shown in future examples.
public class ABankAccount implements BankAccount {
    int currentBalance = 0;
    public static final int MIN_BALANCE = 100;
    public ABankAccount (int initialBalance) {
        currentBalance = initialBalance;
    }
    public int getCurrentBalance () {return currentBalance;}
    public void deposit (int amount) {currentBalance += amount;}
    public boolean withdraw (int amount) {
        int minNecessaryBalance = MIN_BALANCE + amount;
        if (minNecessaryBalance <= currentBalance) {
            currentBalance -= amount;
            return true;
        } else return false;
    }
}
The Importance of Being Earnest
THE IMPORTANCE OF BEING EARNEST
THE IMPORTANCE OF BEING EARNEST

File Edit View Customize ABankAccount

Current Balance
100

Withdraw...
Safe Withdraw...
Deposit

Parameters of Withdraw

File

Parameter 1: int 2147483647
Withdraw

Integer.MAX_INT

[Boolean]

File Edit View Customize Boolean Comparable

✓
THE IMPORTANCE OF BEING EARNEST
The Importance of Being Earnest

Most significant bit of positive (negative) numbers is 0(1)
THE IMPORTANCE OF BEING EARNEST

```java
public boolean safeWithdraw (int amount) {
    assert amount > 0: "amount < 0";
    boolean retVal = withdraw(amount);
    assert currentBalance >= MIN_BALANCE: "currentBalance < MIN_BALANCE";
    return retVal;
}
```
THE IMPORTANCE OF BEING EARNEST
THE IMPORTANCE OF BEING EARNEST

Security Issues - Gain
Description, integer overflow in memory allocation results in heap overflow.
By passing the size variable as -1, integer overflows to 0 when 1 is added in...
gain.sourceforge.net/security/?id=2 - 7k - Cached - Similar pages

Gentoo Linux Documentation -- Sambar Integer overflow
Impact information, 3. Resolution information, 4. References ...

The KOffice Project - XPDF Integer Overflow 2
KOffice 1.3 (including betas) to 1.3.5 have an integer overflow vulnerability in
KWord's PDF. ... References, the corresponding security advisory for KOE: ...
www.koffice.org/security/2004_xpdf_integer_overflow_2.php - 9k - Cached - Similar pages

Microsoft Windows LoadImage API Function Integer Overflow ...
Microsoft Windows is reported prone to a remote integer overflow vulnerability ...
... Microsoft Upgrade Security Update for Windows NT Server 4.0 (KB91711) ...
securityresponse.symantec.com/avcenter/security/Content/12095.html - 26k - Cached - Similar pages

CERT Advisory CA-2002-25 Integer Overflow In XDR Library
There is an integer overflow present in the xdr_array() function ... CERT publications
and other security information are available from our web site ...
www.cert.org/advisories/CA-2002-25.html - 26k - Cached - Similar pages

[LSS] Security [Exposed by LSS] [Details]
mod_auth_radius remote integer overflow. Advisory ID: LSS-2005-01-02 ...
security.lss.hr/fox/index.php?page=details&id=LSS-2005-01-02 - 16k - Cached - Similar pages

Network Security, Vulnerability Assessment, Intrusion Prevention
eEye - Network security & vulnerability management software including ...
Although the copy length is similarly subject to an integer overflow, ...

Network Security, Vulnerability Assessment, Intrusion Prevention
For the purpose of signature development and further security research, a sample
**Expressing Assertions**

- **Natural language**
  - No collection element is null.
  - All collection elements are not odd.
  - All collection elements are either odd or positive.
  - Easy to read but ambiguous.

- **Programming language**
  - Library or language constructs
  - Executable, unambiguous but language-dependent and awkward
    - Useful for debugging
    - Specification cannot be done before language decided.

- **Mathematical language**
  - Unambiguous, time tested, convenient but not executable
  - $\forall j: 0 \leq j < b\text{.size()} : b\text{.get}(j) \neq \text{null}$
**Propositional Calculus**

- **Logic operators**
  - not, and, or
  - We will use Java syntax.

- **Quantifiers**
  - Universal ($\forall$)
  - Existential ($\exists$)

- **Propositional variables**
  - Program
  - Others: Recording, Quantifier

- **Propositions**
  - Boolean expressions involving operators, variables, and quantifiers

- **Simple/quantified propositions**
  - Do not use/use quantifiers

\[
\forall j: 0 \leq j < b\text{.size()} : \\
b\text{.get}(j) \neq \text{null} \land \\
b\text{.get}(j) \neq a\text{.get}(0)
\]
Propositional Algebra

- Calculus based on algebra
- Algebra defines
  - Arithmetic operations
  - Relations operations
  - We will use Java syntax

\[ \forall j : 0 \leq j < b.\text{size()} : \]
\[ b.\text{get}(j) \neq \text{null} \ \&\& \]
\[ b.\text{get}(j) == a.\text{get}(0) \]
**Example Propositions**

- Simple propositions
  - True
  - False
  - weight > 0
  - (weight > 0) && (height > 0)

- Quantified
  - $\forall j: 0 \leq j < b.\text{size}(): b.\text{get}(j) \neq \text{null}$
    - All elements of B are not null
  - $\exists j: 0 \leq j < b.\text{size}(): b.\text{get}(j) \neq \text{null}$
    - At least one element of B is not null.
Quantified Propositions

- **Quantified**
  - \( \forall j: 0 \leq j < b.\text{size}(): \)
    - \( b.\text{get}(j) \neq \text{null} \)
  - \( \exists j: 0 \leq j < b.\text{size}(): \)
    - \( b.\text{get}(j) \neq \text{null} \)

- **General form:**
  - \( Qx:D(x):P(x) \)
  - \( Q \) is either \( \forall \) or \( \exists \) quantifier
  - \( X \) is quantified variable
  - \( D(x) \) is domain description
  - \( P(x) \) is sub-proposition

- **Sub-proposition**
  - Simple or quantified proposition in terms of quantifier

- **Domain**
  - A collection of values used in sub-proposition evaluation
  - \( b.\text{get}(0), \ldots, b.\text{get}(b.\text{size}() - 1) \)

- **Domain description**
  - Describes domain using quantified variable
Quantified Assertions

Syntax

- $Qx:D(x):P(x)$
- $\forall j: 0 \leq j < b.size(): b.get(j) \neq \text{null}$
- $\exists j: 0 \leq j < b.size(): b.get(j) \neq \text{null}$

Goal:

- Write general boolean functions that take as arguments encoding of the elements of domain and return true iff proposition is true
**Expressing Quantified Assertions**

```java
public interface Asserter {
    public boolean checkQuantified (String assertion);
}
```

Cannot pass expression string as variables (such as b) in our scope have no meaning to library

Library must do parsing.

Qx:D(x):P(x)
∀j: 0 <= j < b.size(): b.get(j) != null

**assert** asserter.checkQuantified("∀j: 0 <= j < b.size(): b.get(j) != null"): “some element of b is null” ;
Separate the Three Components

```java
public interface Asserter {
    public boolean checkUniversal (...,...);
    public boolean checkExistential (...,...);
}
```

Separate functions for two quantifiers

One argument for domain and one for predicate

Assume domain is some collection

Qx: D(x): P(x)  
∀j: 0 <= j < b.size(): b.get(j) != null

assert asserter.checkUniversal(...,...): “some element of b is null”
public interface Asserter<ElementType> {
    public boolean checkUniversal (List<ElementType> domain, ...);
    public boolean checkExistential (List<ElementType> domain, ...);
}

Assume domain implements List

Want to support histories, sets, databases, streams...

Qx:D(x):P(x)
∀j: 0 <= j < b.size(): b.get(j) != null

assert asserter.checkUniversal(b, ....): “some element of b is null” ;
public interface Asserter<ElementType> {
    public boolean checkUniversal (Iterator<ElementType> domain, ...);
    public boolean checkExistential (Iterator<ElementType> domain, ...);
}

Qx:D(x):P(x)
∀j: 0 <= j < b.size(): b.get(j) != null

assert asserter.checkUniversal(b.iterator(), ....): “some element of b is null” }
package util.assertions;
import java.util.Iterator;

public class AnAsserter<ElementType> implements Asserter<ElementType> {
  public boolean checkUniversal (Iterator<ElementType> elements, ...) {
    while (elements.hasNext())
      if (!...) return false;
    return true;
  }

  public void checkExistential (Iterator<ElementType> elements, ...) {
    while (elements.hasNext())
      if (...) return true;
    return false;
  }
}
Describing the Domain Using Method Parameters

```java
public boolean checkUniversal (Iterator<ElementType> elements, (<ElementType→ boolean)elementChecker) {
    while (elements.hasNext())
        if (!elementChecker(elements.next())) return false;
    return true;
}
```

Method must be invoked on some object

Passing method and object separately raises typing issues

```java
public boolean nonNullChecker(Object element) {
    return element != null;
}
```

```java
assert asserter.checkUniversal(b.iterator(), nonNullChecker, "some element of b is null")
```
Describing the Domain Using Method Parameters

```java
public boolean checkUniversal (Iterator<ElementType> elements, ElementChecker elementChecker) {
    while (elements.hasNext())
        if (!elementChecker.check(elements.next()))
            return false;
    return true;
}
```

Pass an object with a check method that takes argument of type `<ElementType>`

Qx:D(x):P(x)
∀j: 0 <= j < b.size(): b.get(j) != null

```
assert asserter.checkUniversal(b.iterator(), new ANonNullChecker()): “some element of b is null”;
```
**Subproposition Visitor Objects**

```java
package util.assertions;
public interface ElementChecker<ElementType> {
    public boolean check (ElementType element);
}

import util.assertions.ElementChecker;
public class ANonNullChecker implements ElementChecker<Object> {
    public boolean check(Object element) {
        return element != null;
    }
}

public boolean checkUniversal (Iterator<ElementType> elements, ElementChecker elementChecker) {
    while (elements.hasNext())
        if (!elementChecker.check(elements.next()))
            return false;
    return true;
}

assert asserter.checkUniversal(b.iterator(), new ANonNullNullChecker()): “some element of b is null”;
package util.assertions;

public interface ElementChecker<ElementType> {
    public boolean check (ElementType element);
}

import util.assertions.ElementChecker;

public class ANonNullChecker implements ElementChecker<Object> {
    public boolean check(Object element) {
        return element != null;
    }
}

∀j: 0 <= j < b.size(): b.get(j) != a.get(0)

assert asserter.checkUniversal(b.iterator(), new ANonNullChecker()): “some element of b is null” ;
import util.assertions.ElementChecker;

public class AnInequalityChecker implements ElementChecker<String> {
    String testObject;
    public AnInequalityChecker(String theTestObject) {
        testObject = theTestObject;
    }
    public boolean check(String element) {
        return !element.equals(testObject);
    }
}

Each external var becomes constructor parameter and checker instance variable

∀j: 0 <= j < b.size(): b.get(j) != a.get(0)

assert asserter.checkUniversal(b.iterator(), new AnInequalityChecker(a.get(0))): “some element of b == a.get(0)” ;
**ActionObject**

Action Object = Embedded Operation

Provides an execute method to perform some embedded operation.

The execute operation takes the object on which the embedded operation is to be invoked and an array of parameters of the target method.

execute (targetObject, params)

getWeightMethod.invoke (bmi, nullParams);
**COMMAND OBJECT**

Command Object = Embedded Operation + Target + Parameters

- Provides a execute operation to perform some embedded operation.
- The execute operation takes no arguments.

Constructor (targetObject, params)

execute()

setWeightCommand.execute();

- Constructor takes parameters of operation as arguments.
- Action is an operation that can be invoked on many different arguments.
- A command is a specific action invocation.
**Visitor Object**

**Visitor Object** = Embedded Operation + Parameters

Provides a execute operation to perform some embedded operation.

The execute operation takes target object as argument.

**execute (targetObject)**

**Constructor (params)**

**elementChecker.check (elements.next())**

**Constructor in (a) command takes target and params, (b) action takes no params, and (c) visitor takes params as arguments.**

**Execute in (a) command takes no params, (b) action takes target object and params, (c) visitor takes target object.**

```java
public boolean check(String element) {
    return !element.equals(testObject);
}
```
Example of Visitor Pattern

- **Iterator**<T>

  - **Component**
    - T
    - T

  - **ElementChecker**
    - **implements**
      - ANonNull Checker
      - AnInequality Checker

  - **uses**
    - checkUniversal() (method)
    - checkExistential() (method)
Visitor Pattern

Collection Interface C with elements of type T

- element1: T
  - component
  - implements

Visitor Interface

- implements

Visitor Class 1

Visitor Class 2

uses

- Traverser 1
- Traverser 2
Everyday Visitor Objects

- Compiler visitors for:
  - Formatting program elements
  - Refactoring program elements
  - Compiling program elements.

- ObjectEditor visitors for:
  - Attaching widgets to object components.
  - Registering listeners of object components.
  - Printing a textual representation of object components.
VISITOR PATTERN

- Some collection C of elements of type T
- Visitor interface
  - `public interface ElementChecker { public boolean visit (T element); }`
  - `public interface V { public T2 m (T p); }
- One or more traverser methods that use collection and visitor interface to pass one or more collection elements to the method.
  - `public static boolean checkUniversal (Iterator<ElementChecker> domain, ElementChecker subProposition)
    { ...subProposition.visit(domain.next());
    traverser1 (C c, V v) { ...v.m(element of C)... }
- Implementation of interface whose constructors take as arguments external variables that need to be accessed by the visitor method
  - `public class AnInequalityChecker implements ElementChecker {
    Object testObject;
    public AnInequalityChecker(Object theTestObject) {...}
    public boolean visit(Object element) {...};
  }
  public class AV1 implements V {
    public AV1 (T1 p1, ... Tn pN) { ...}
    public T2 m (T p) { ... }
  }
- Client passes traverser visitor implementation and collection
  - `asserter.checkUniversalb.iterate(), new AnInequalityChecker(a.get(0));`
  - `traverser1(c, new AV1(a1,.. aN));`
switch c {  
case 'a': ...
  case 'b': ...
  default: assert false
}

Unreachable statement
Nested Assertions

public class AListChecker implements ElementChecker<List> {
    public boolean check(List element) {
        Iterator children = element.iterate();
        return asserter.checkUniversal(children, new ANonNullChecker());
    }
}

∀j: 0 <= j < b.size(): ∀k: 0 <= k < b.get(j).size(): b.get(j).get(k) != null

assert asserter.checkUniversal(b.iterator(), new AListChecker(), “some nested element of b is null”)
package util.assertions;
import java.util.Iterator;
public class AnAsserter<ElementType> implements Asserter<ElementType> {
    public void assertUniversal (Iterator<ElementType> elements, ElementChecker elementChecker, String message) {
        while (elements.hasNext()) {
            if (!elementChecker.check(elements.next())) throw new AssertionError (message);
        }
    }
    public void assertExistential (Iterator<ElementType> elements, ElementChecker elementChecker, String message) {
        while (elements.hasNext()) {
            if (elementChecker.check(elements.next())) return;
            throw new AssertionError (message);
        }
    }
}

asserter.assertUniversal(b.iterator(), nonNullChecker, “some element of b is null”);
**Java Assertions**

- `assert <Boolean Expression>`
- `assert <Boolean Expression>: <Value>`

Statement can be inserted anywhere to state that some condition should be true.

- If condition is false, Java throws `AssertionError`, which may be caught by programmer code.
- If uncaught, generic message saying assertion failed is printed.

- An assertion made at the beginning/end of a statement block (method, loop, if ..) is called its precondition/postcondition.
WHY LANGUAGE SUPPORT

- Can always define a library with assert(<Boolean Expression>) method that throws a special exception denoting assertion error.

- Assertions can be dynamically turned on or off for package or Class
  - `java -ea assignment11.MainClass -da bus.uigen...`

```java
public void myAssert (boolean boolExp, String message) throws AssertionError {
    if (boolExp) throw new AssertionError (message);
}
```
**ERROR VS. EXCEPTION**

- Java assertion failure results in `AssertionError`
- Subclass of `Error` rather than `RunTimeException`
- Reasoning:
  - Convention dictates that `Exception` should be caught
  - Should “discourage programmers from attempting to recover from assertion failures.”
    - Might do custom reporting, mail error report etc.
  - `AssertionError` is a subclass of `Throwable` and can indeed be caught
  - Decision was controversial
public class ABankAccount implements BankAccount {
    int currentBalance = 0;
    public static final int MIN_BALANCE = 100;
    public ABankAccount (int initialBalance) {
        currentBalance = initialBalance;
    }
    public int getCurrentBalance () {return currentBalance;}
    public void deposit (int amount) {currentBalance += amount;}
    public boolean withdraw (int amount) {
        int minNecessaryBalance = MIN_BALANCE + amount;
        if (minNecessaryBalance <= currentBalance) {
            currentBalance -= amount;
            return true;
        } else return false;
    }
}
public boolean safeWithdraw (int amount) {
    assert amount > 0: "amount < 0";
    boolean retVal = withdraw(amount);
    assert currentBalance >= MIN_BALANCE: "currentBalance < MIN_BALANCE"
    return retVal;
}
THE IMPORTANCE OF BEING Earnest
The Importance of Being Earnest
The Importance of Being Earnest

integer.MAX_INT
THE IMPORTANCE OF BEING EARNEST
The Importance of Being Earnest

Most significant but of positive (negative) numbers is 0(1)
THE IMPORTANCE OF BEING EARNEST
The Importance of Being Earnest

Security Issues - Gain
Description, integer overflow in memory allocation results in heap overflow. By passing the size variable as -0, integer overflows to 0 when 1 is added in ...
gain.sourceforge.net/security/?id=2 - 7k - Cached - Similar pages

Gentoo Linux Documentation -- Samba Integer overflow
Samba: Integer overflow. Content: 1. Gentoo Linux Security Advisory. 2. Impact information. 3. Resolution information. 4. References ...

The KOffice Project - XPDF Integer Overflow 2
KOffice 1.3 (including betas) to 1.3.5 have an integer overflow vulnerability in KWord's PDF ... References, the corresponding security advisory for KOE ...
www.koffice.org/security/2004_xpdf_integer_overflow_2.php - 9k - Cached - Similar pages

Microsoft Windows LoadImage API Function Integer Overflow ...
Microsoft Windows is reported prone to a remote integer overflow vulnerability ...
Microsoft Upgrade Security Update for Windows NT Server 4.0 (KB91711) ...
securityresponse.symantec.com/arcenter/security/content/12095.html - 26k - Cached - Similar pages

CERT Advisory CA-2002-25 Integer Overflow In XDR Library
There is an integer overflow present in the xdr_array() function ... CERT publications and other security information are available from our web site ...
www.cert.org/advisories/CA-2002-25.html - 26k - Cached - Similar pages

[LSS | Security | Exposed by LSS | Detail1]
security.lss.hr/index.php?page=details&id=LSS-2005-01-02 - 16k - Cached - Similar pages

Network Security, Vulnerability Assessment, Intrusion Prevention
eEye - Network security & vulnerability management software including ...
Although the copy length is similarly subject to an integer overflow ...

Network Security, Vulnerability Assessment, Intrusion Prevention
For the purpose of signature development and further security research, a sample
EXTRA SLIDES
**Problem with inaccessible variables**

- **Syntax**
  - $Q_x : D(x) : P(x)$
  - $\forall j: 0 \leq j < \text{b.size}(): \text{b.get}(j) \neq \text{null}$
  - $\exists j: 0 \leq j < \text{b.size}(): \text{b.get}(j) \neq \text{null}$

- **How to describe $D(x)$ and $P(x)$?**
  - Cannot pass expression string as variables involved have no meaning to library
  - Will pass one argument describing the domain
    - Collection of elements
  - Another argument describing the subproposition to be evaluated for each domain element.
package util.assertions;
import java.util.Iterator;
public class AnAsserter<ElementType> implements Asserter<ElementType> {
  public void assertUniversal (Iterator<ElementType> elements, ElementChecker elementChecker, String message) {
    while (elements.hasNext())
      if (!elementChecker.check(elements.next())) throw new AssertionError (message);
  }
  public void assertExistential (Iterator<ElementType> elements, ElementChecker elementChecker, String message) {
    while (elements.hasNext())
      if (elementChecker.check(elements.next())) return;
    throw new AssertionError (message);
  }
}
package util.assertions;
import java.util.Iterator;
public class AnAsserter<ElementType> implements Asserter<ElementType> {
    public void assertUniversal (Iterator<ElementType> elements, ..., String message) {
        assert checkUniversal(elements, ..., message);
    }
    public void assertExistential (Iterator<ElementType> elements, ElementChecker elementChecker, String message) {
        assert !checkExistential(elements, ..., message);
    }
    public boolean checkUniversal (Iterator<ElementType> elements, ...) {
        while (elements.hasNext())
            if (...) return false;
        return true;
    }
    public void checkExistential (Iterator<ElementType> elements, ...) {
        while (elements.hasNext())
            if (...) return true;
        return false;
    }
}
package util.assertions;
import java.util.Iterator;
public interface Asserter<ElementType> {
    public void assertUniversal (Iterator<ElementType> enumParam, ElementChecker elementChecker, String message);
    public void assertExistential (Iterator<ElementType> enumParam, ElementChecker elementChecker, String message);
}
package util.assertions;
public interface Asserter<ElementType> {
    public void assertUniversal (Iterator<ElementType> enumParam, ElementChecker elementChecker, String message);
    public void assertExistential (Iterator<ElementType> enumParam, ElementChecker elementChecker, String message);
}
**Problem with inaccessible variables**

- **Syntax**
  - $Q_x:D(x):P(x)$
  - $\forall j: 0 \leq j < b.size(): b.get(j) \neq \text{null}$
  - $\exists j: 0 \leq j < b.size(): b.get(j) \neq \text{null}$

- **How to describe $D(x)$ and $P(x)$?**
  - Cannot pass expression string as variables involved have no meaning to library
  - Will pass one argument describing the domain
    - Collection of elements
  - Another argument describing the subproposition to be evaluated for each domain element.
How to Describe Domain?

- Syntax
  - Qx:D(x):P(x)
  - $\forall j: 0 \leq j < b.size(): b.get(j) \neq \text{null}$
  - $\exists j: 0 \leq j < b.size(): b.get(j) \neq \text{null}$

- Domain can be
  - Array, Vector, StringHistory, ...

- Need a common interface to describe elements
  - java.util.Iterator
Describing the Domain

```java
public interface Asserter<ElementType> {
    public void assertUniversal (Iterator<ElementType> domain, …, String message);
    public void assertExistential (Iterator<ElementType> domain, …, String message);
}

asserter.assertUniversal (B.iterate(), …, "Some element of B is null");
asserter.assertExistential(B.iterate(), …, "All elements of B are null");
```
Describing the Domain Using Method Parameters

```java
public interface Asserter<ElementType> {
    public void assertUniversal (Iterator<ElementType> domain, (<ElementType> boolean)elementChecker, String message);
    public void assertExistential (Iterator<ElementType> domain, (<ElementType> boolean)elementChecker, String message);
}

asserter.assertUniversal (B.iterate(), ..., "Some element of B is null");
asserter.assertExistential(B.iterate(), ..., "All elements of B are null");
```
public interface Asserter<ElementType> {
    public void assertUniversal (Iterator<ElementType> enumParam, ElementChecker elementChecker, String message);
    public void assertExistential (Iterator<ElementType> enumParam, ElementChecker elementChecker, String message);
}
Describing the Domain

```java
package assertions;
import java.util.Enumeration;
public class AQuantifier {
    public static boolean forAll ((Enumeration domain, ...) {
        while (domain.hasMoreElements())
            ...
    }
    public static boolean thereExists ((Enumeration domain, ...) {
        while (domain.hasMoreElements())
            ...
    }
}
```

AnAsserter.assert(AQuantifier.forAll(B.elements(), ..., "Some element of B is null");
AnAsserter.assert(AQuantifier.thereExists(b.elements(), ..., "All elements of B are null");
Need to fill ...
HOW TO DESCRIBE SUBPROPOSITION

- Syntax
  - $Qx:D(x):P(x)$
  - $\forall j: 0 <= j < b.size(): b.get(j) != null$
  - $\exists j: 0 <= j < b.size(): b.get(j) != null$

- Cannot pass expression string as variables involved have no meaning to library
- But can pass function that evaluates it.
  - `boolean isNotNull(Object element) {
    return element != null;
  }`
- Function will be evaluated for each domain element by our libraries
package assertions;
import java.util.Enumeration;
public class AQuantifier {
    public static boolean forAll (Enumeration domain, (object \rightarrow boolean) subProposition) {
        while (domain.hasMoreElements())
            if (!subProposition (domain.nextElement())) return false;
        return true;
    }
    public static boolean thereExists ((Enumeration domain, (object \rightarrow boolean) subProposition) {
        while (domain.hasMoreElements())
            if (!subProposition (domain.nextElement())) return true;
        return false;
    }
}
AnAsserter.assert(AQuantifier.forAll(B.elements(), isNotNull), "Some element of B is null");
AnAsserter.assert(AQuantifier.thereExists(b.elements(), isNotNull), "All elements of B are null");
**How to describe Subproposition**

- Can’t pass expression
- Can pass function that evaluates it.

```java
boolean isNotNull(Object element) {
    return element != null;
}
```

Java does not support function parameters but allows object parameters.

```
public class ANonNullChecker implements ElementChecker<Object> {
    public boolean check(String element) {
        return element != null;
    }
}
```

A subproposition object visits each element.

```java
import util.assertions;
public interface ElementChecker<ElementType> {
    public boolean check (ElementType element);
}
```

```java
public class ANonNullChecker implements ElementChecker<Object> {
    public boolean check(String element) {
        return element != null;
    }
}
```
package assertions;
import java.util.Enumeration;
public class AQuantifier {
    public static boolean forAll (Enumeration domain, ElementChecker subProposition) {
        while (domain.hasMoreElements())
            if (!subProposition.visit(domain.nextElement())) return false;
        return true;
    }
    public static boolean thereExists (Enumeration domain, ElementChecker subProposition) {
        while (domain.hasMoreElements())
            if (subProposition.visit(domain.nextElement())) return true;
        return false;
    }
}

AnAsserter.assert(AQuantifier.forAll(b.elements(), new ANonNullChecker()), "Some element of B is null");
AnAsserter.assert(AQuantifier.thereExists(b.elements(), new ANonNullChecker()), "All elements of B are null");
public interface Asserter<ElementType> {
    public void assertUniversal (Iterator<ElementType> enumParam, ElementChecker elementChecker, String message);
    public void assertExistential (Iterator<ElementType> enumParam, ElementChecker elementChecker, String message);
}
CALLS VS. CALLBACKS

- Calls
  - calls from reusing class to reused class

- Callbacks
  - calls from reused class to reusing class
  - not to implement a symbiotic relationship
  - done to service calls
SUBPROPOSITION accessing vars other than domain elements

- $\forall j: 0 \leq j < b.\text{size}(): b.\text{get}(j) \neq a.\text{get}(0)$
- $\exists j: 0 \leq j < b.\text{size}(): b.\text{get}(j) \neq a.\text{get}(0)$

```java
public class ANonNullChecker implements ElementChecker<Object> {
    public boolean check(String element) {
        return element != null;
    }
}
```

AnAsserter.assert(AQuantifier.forAll(b.elements(), new ANonNullChecker()), "Some element of B is null");
Subproposition accessing vars other than domain elements

- \( \forall j: 0 \leq j < b.\text{size}() \): \( b.\text{get}(j) \neq a.\text{get}(0) \)
- \( \exists j: 0 \leq j < b.\text{size}() \): \( b.\text{get}(j) \neq a.\text{get}(0) \)

```java
package util.assertions;
import assertions.ElementChecker;
public class AnInequalityChecker implements ElementChecker<String> {
    String testObject;
    public AnInequalityChecker(String theTestObject) {
        testObject = theTestObject;
    }
    public boolean check(String element) {
        return !element.equals(testObject);
    }
}

AnAsserter.assert(AQuantifier.forAll(b.elements(), new AnInequalityChecker(a.get(0))), "Some element of b is equal to a.get(0)");
```

Each external var becomes constructor parameter
VISITOR PATTERN

- Some collection C of elements of type T
- Visitor interface
  - `public interface ElementChecker { public boolean visit (Object element);}`
  - `public interface V {public T2 m (T p);}
- One or more traverser methods that use collection and visitor interface to pass one or more collection elements to the method.
  - `public static boolean forAll (Enumeration domain, ElementChecker subProposition) {...subProposition.visit(domain.nextElement());
  - traverser1 (C c, V v) { ...v.m(element of C)...}
- Implementation of interface whose constructors take as arguments external variables that need to be accessed by the visitor method
  - `public class AnInequalityChecker implements ElementChecker {
    Object testObject;
    public AnInequalityChecker(Object theTestObject) {...}
    public boolean visit(Object element) {...};
    }
  - `public class AV1 implements V {
    public AV1 (T1 p1, ... Tn pN) {...}
    public T2 m (T p) { ... }
    }
- Client passes traverser visitor implementation and collection
  - `AQuantifier.forAll(b.elements(), new AnInequalityChecker(a.get(0)));`
**ACTIONOBJECT**

Action Object

Provides an execute operation to perform some action.

The execute operation takes the object on which the target operation is to invoked and an array of parameters of the target method.
**COMMAND OBJECT**

**Command Object**

Provides a execute operation to perform some action.

The execute operation takes no arguments.

Constructor:

Constructor (targetObject, params)

execute ()

Constructor takes parameters of operation as arguments.

Action is an operation that can be invoked on many different arguments.

A command is a specific action invocation.
**VISITOR OBJECT**

Visitor Object

- Provides a execute operation to perform some action.
- The execute operation takes target object as argument.

**execute (targetObject)**

**Constructor (params)**

- Constructor in (a) command takes target and params, (b) action takes no params, and (c) visitor takes oparams as arguments.
- Execute in (a) command takes no params, (b) action takes target object and params, (c) visitor takes target object.
**Visitor Pattern**

Collection Interface C with elements of type T

- component
- Traverser 1
- Traverser 2

Visitor Interface

- implements

Visitor Class 1

Visitor Class 2

uses

- element1: T
- element1: T
Example of Visitor Pattern

Iterator\(<T>\)

uses

assertUniversal()  
assertExistential()

ElementChecker

implements

ANonNull Checker  
AnInequality Checker
Nested Assertions

∀j: 0 ≤ j < b.size(): ∀k: 0 ≤ k < b.get(j).size(): b.get(j).get(k) != null

package visitors;
import assertions.ElementChecker;
public class ANonNullChecker implements ElementChecker {
    public boolean visit(Object element) {
        return (element != null);
    }
}

package visitors;
import assertions.ElementChecker;
public class AListChecker implements ElementChecker<List> {
    public boolean check(List element) {
        Iterator children = element.iterate();
        return asserter.assertUniversal(children, new ANonNullChecker());
    }
}

AnAsserter.assert(AQuantifier.forAll(b.elements(), new AForAllChecker(), "Some leaf-level element of b is null");
EXAMPLE OF PATTERN IN EVERYDAY APPLICATIONS

- Program tree
- A visitor for printing all nodes.
- Another for type checking.
- Yet another for generating code
- Do not want to put all of this code in tree class.
- In any case, printing should not be in tree.
**Subproposition accessing vars other than domain elements**

```java
public class ANonNullChecker implements ElementChecker<Object> {
    public boolean check(Object element) {
        return element != null;
    }
}
```

∀j: 0 <= j < b.size(): b.get(j) != a.get(0)

```java
assert checkUniversal(b.iterator(), new ANonNullChecker()):
    "some element of b is null"
```
public boolean nonNullChecker(Object element) {
    return element != null;
}

Check Method