Self Testing

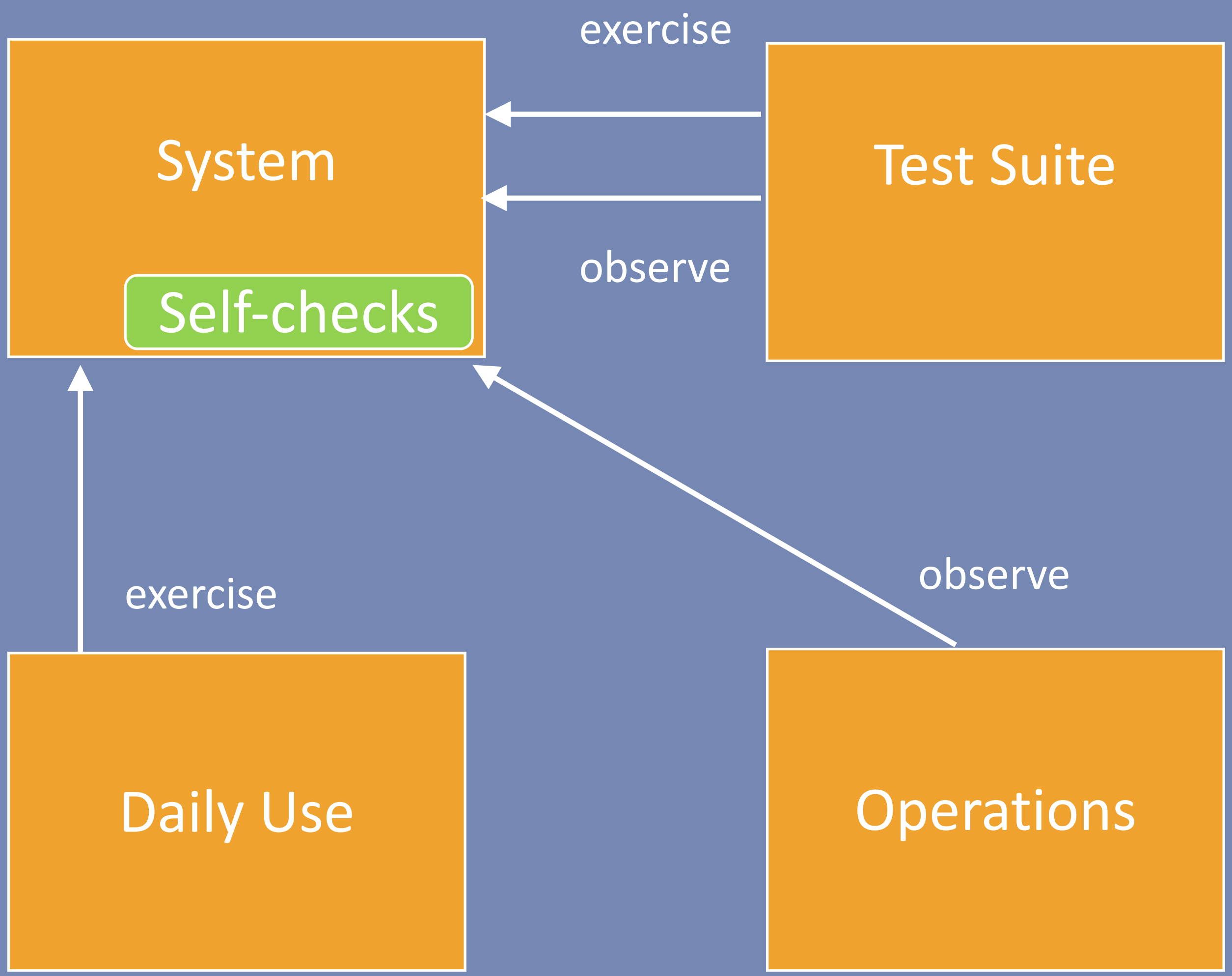
CSE1110

Software Testing & Quality Engineering

Arie van Deursen

June 7, 2019

Self Testing



The Java (C, C++, Python, …) assert Statement

```
"assert" boolean-expression [":" string ]
```

If *boolean-expression* is true, do nothing.

If it is false, raise an `AssertionError`, with the string as message

Assert by Example

```
public class MyStack() {

  public Element pop() {
    assert count() > 0;
    ....
    // real method body here
    ....
    assert count() == oldCount - 1;
  }
}
```

Assertion Checking can be Enabled or Disabled

- Enabling assertions = running application in “self-aware” mode.

- java –enableassertions
- java –ea
- Eclipse: add jvm argument to run configuration
- Maven/IntelliJ: *enabled by default when executing tests*

- *Program must run correctly independent of assertion status!*

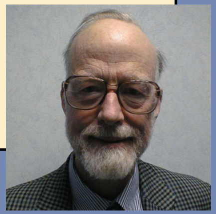
Assertions Defined

An assertion is a Boolean expression
at a specific point in a program
which will be true
unless there is a <u>bug</u> in the program.

http://wiki.c2.com/?WhatAreAssertions

Pre- and Postconditions

```
{ P }  A  { Q }
```



Tony Hoare

- Any execution of A,
  - starting in a state where P holds
  - will terminate in a state where Q holds

{ *preconds* } **Method** { *postconds* }

Preconditions

Can you think of some pre-conditions?

```
public class FavoriteBooks() {
 List<Book> favorites;
 ...
 public void merge(List<Book> books) {




   favorites.addAll(books);
   pushNotifications.booksAdded(books);
  }
}
```

Preconditions

```
public class FavoriteBooks() {
 List<Book> favorites;
  ...
 public void merge(List<Book> books) {
  assert books != null;




  favorites.addAll(books);
  pushNotifications.booksAdded(books);
 }
}
```

Preconditions

```
public class FavoriteBooks() {
 List<Book> favorites;
 ...
 public void merge(List<Book> books) {
  assert books != null;
  assert !books.isEmpty();



  favorites.addAll(books);
  pushNotifications.booksAdded(books);
 }
}
```

Preconditions

```
public class FavoriteBooks() {
 List<Book> favorites;
 ...
 public void merge(List<Book> books) {
  assert books != null;
  assert !books.isEmpty();
  assert favorites != null;


  favorites.addAll(books);
  pushNotifications.booksAdded(books);
 }
}
```

Preconditions

```
public class FavoriteBooks() {
 List<Book> favorites;
 ...
 public void merge(List<Book> books) {
  assert books != null;
  assert !books.isEmpty();
  assert favorites != null;
  assert !favorites.containsAll(books);
  ...
  favorites.addAll(books);
  pushNotifications.booksAdded(books);
 }
}
```

Weakening Preconditions

```
public class FavoriteBooks() {
 List<Book> favorites;
 ...
 public void merge(List<Book> books) {
  assert books != null;
  assert favorites != null;
  assert !favorites.containsAll(books);
  ...
  if (!books.isEmpty()) {
   favorites.addAll(books);
   pushNotifications.booksAdded(books);
}}}
```

Weakening Preconditions

```
public class FavoriteBooks() {
 List<Book> favorites;
 ...
 public void merge(List<Book> books) {
  assert books != null;
  assert favorites != null;

  // logic to find new books only
  List<Book> newBooks = ... (books);

  if (!newBooks.isEmpty()) {
   favorites.addAll(newBooks);
   pushNotifications.booksAdded(...);
}}}
```

Precondition Design

- “Strength” of preconditions is a design choice.
- The *weaker* your precondition
  - The more situations your method needs to handle
  - The less thinking the client needs to do
- However, with weak preconditions:
  - The method will *always* have to do the checking
  - Checks even done if we’re sure they’ll pass.

Postconditions

```
public class FavoriteBooks() {
 List<Book> favorites;
  ...
 public void merge(List<Book> books) {
  // assert four preconditions
  ...
  // the method body
  ...
  // the postcondition.
  ...??
}}
```

Postconditions

```
public class FavoriteBooks() {
 List<Book> favorites;
  ...
 public void merge(List<Book> books) {
  // assert four preconditions
   ...
  // the method body
   ...
  // the postcondition.
  assert favorites.containsAll(books);
}}
```

Composite Postconditions

Multiple exit paths?

Overall postcondition = Disjunction (OR'ed) of multiple smaller post-conditions

```
if (A) {
  ...
  if (B) {
    ...
    assert PC1
    return ...;
  } else {
    ...
    assert PC2
    return ...;
  }
}
...
assert PC3
return ...;
```

A && B ? PC1

A && !B ? PC2

!A ? PC3

Postcondition Design

- Postcondition holds *after* method execution
  - Represents (part of) the desirable effect the method should have
- Method guarantees its postcondition
  - as long as caller meets its preconditions.
- With weak preconditions?
  - Multiple postconditions guarded by conditions over the inputs or program state.

Invariants

Invariant:

A condition that holds throughout the lifetime of a system, an object, or a data-structure.

Two Invariant Idioms for Checking Representations

- Implementing a non-trivial data structure?

- Create a representation checker (“checkRep”)
  - that traverses the entire structure
  - and asserts everything it can.

- Alternative: offer Boolean method (“repOK”)
  - Return a single value indicating whether the data structure is in a consistent state

checkRep by Example: Red-Black Tree Consistency

Parents of children of a node are the same as that node

Nodes are well sorted

```
/* the tree order must be respected */
/* parents and children must point to each other */
if (node->left != t->nil) {
  int tmp = t->Compare (node->key, node->left->key);
  assert (tmp==0 || tmp==1);
  assert (node->left->parent == node);
}
if (node->right != t->nil) {
  int tmp = t->Compare (node->key, node->right->key);
  assert (tmp==0 || tmp==-1);
  assert (node->right->parent == node);
}
if (node->left != t->nil && node->right != t->nil) {
  int tmp = t->Compare (node->left->key, node->right->key);
  assert (tmp==0 || tmp==-1);
}
```


https://blog.regehr.org/archives/1091

Class Invariant Rule

Assertion *P* is a class invariant for class *C* if:

- *All public methods and constructors of C,*
- *when applied to arguments satisfying the methods precondition,*
- *yield a state satisfying P.*

Asserting Class Invariants

- “repOK” idiom at class level.
- Boolean “invariant()” method
  - Assert after constructor
  - Assert at start and end of any public method

Defining Invariants

```
public class FavoriteBooks() {
 List<Book> favorites;
 List<Listeners> pushNotifications;

 protected boolean invariant() {
  return
   favorites != null &&
   pushNotifications != null
 }

 ...
}
```

Invariant at Constructor End

```
public class FavoriteBooks() {
 ...

 protected boolean invariant() { ... }

 public FavoriteBooks(...) {
  favorites = ...
  pushNotifications = ...
  ...
  assert invariant();
 }
 ...
}
```

Invariant at Method Start and End

Some pre-conditions now have moved to invariant

Invariant = pre- and postcondition shared by all methods

```
public class FavoriteBooks() {
 ...
 protected boolean invariant() { ... }

 public merge(List<Book> books) {
  assert invariant();
  // assert remaining pre-conditions
  ...
  // assert remaining post-conditions
  assert invariant();
 }
}
```

Tree Invariants

For your own classes: Learn to think in terms of invariants!

```
public class Node() {
 Node left;
 Node right;
 Node parent;
 ...
 protected boolean invariant() {
  return parentsOK() && orderingOK();
 }

 private boolean parentsOK() {
  return
   (left == null || left.parent == this) &&
   (right == null || right.parent == this)
 }
}
```

Intermezzo: @NotNull

@NotNull

The `@NotNull` annotation is, actually, an explicit contract declaring that:

- A method should not return null
- Variables (fields, local variables, and parameters) cannot hold a null value

IntelliJ IDEA warns you if these contracts are violated.

Intermezzo: @Nullable

@Nullable

The `@Nullable` annotation helps you detect:

- Method calls that can return null
- Variables (fields, local variables, and parameters), that can be null

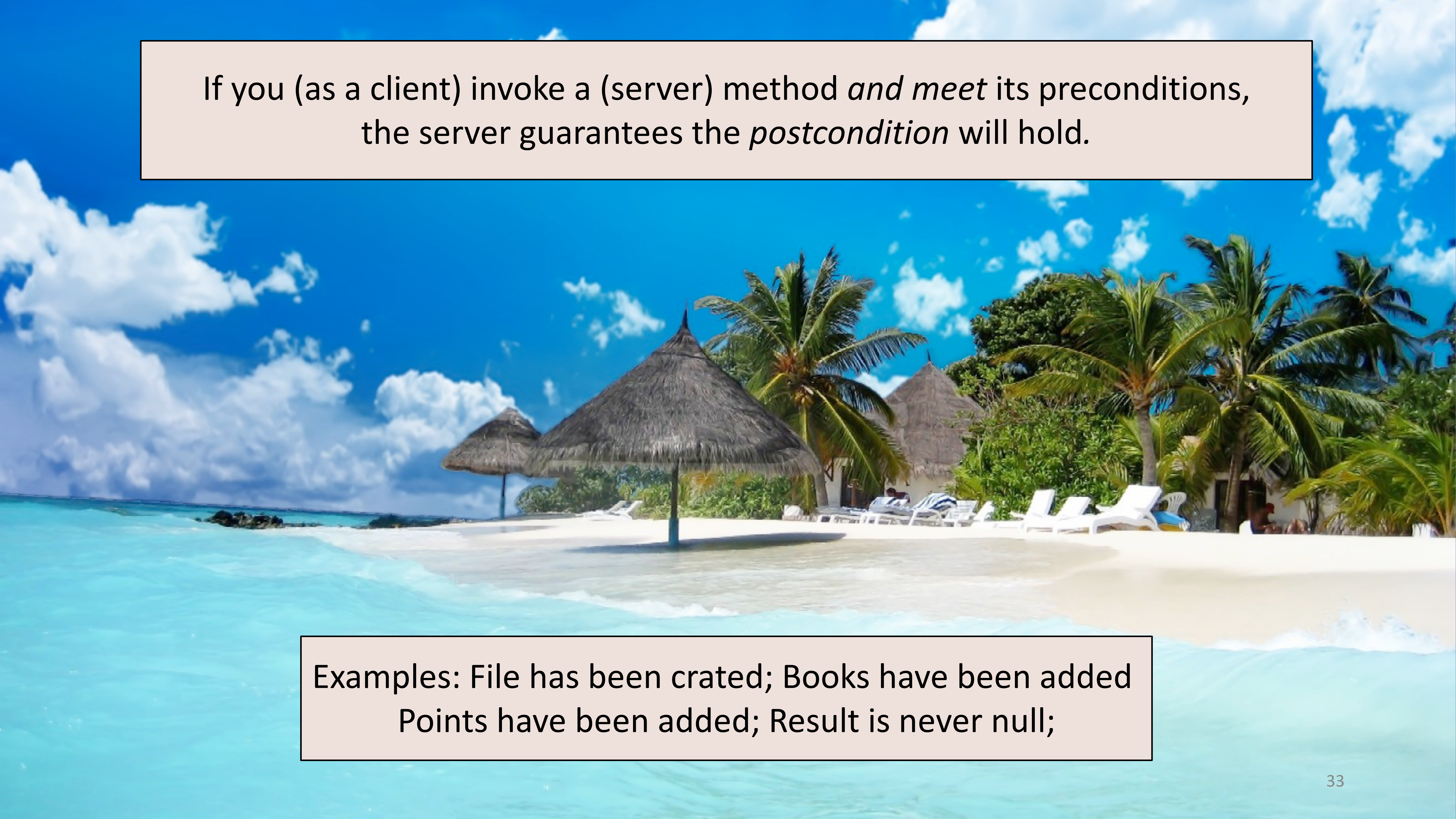
Replacing Null-Checking Preconditions with @NotNull

```
public class FavoriteBooks() {
 @NotNull List<Book> favorites = ...
 ...
 public void merge(@NotNull List<Book> books) {
  assert !books.isEmpty();
  assert !favorites.containsAll(books);
  ...
  favorites.addAll(books);
  pushNotifications.booksAdded(books);
 }
}
```

Interfaces as Contracts

- A client and a server are bound by a contract
- The server promises to do its job
  - Defined by the postconditions
- As long as the client uses the server correctly
  - Defined by the pre-conditions

Bertrand Meyer
Design by Contract



If you (as a client) invoke a (server) method *and meet* its preconditions, the server guarantees the *postcondition* will hold.

Examples: File has been crated; Books have been added Points have been added; Result is never null;

If you (as a client) invoke a (server) method <u>without</u> meeting its preconditions, *anything can happen.*

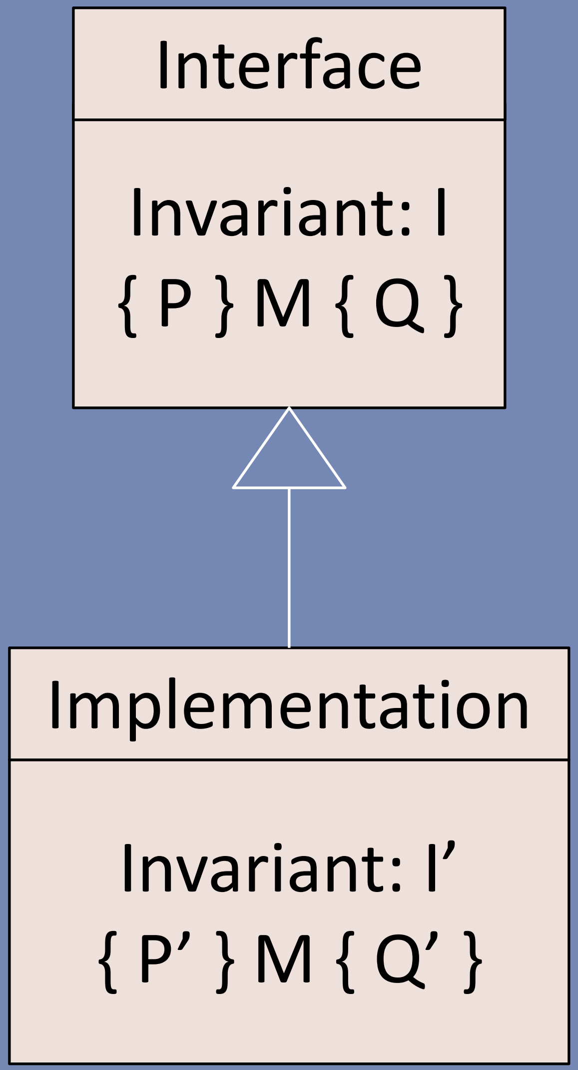
E.g.: Null pointer exception

Proposition *Strength*

- P is stronger than Q

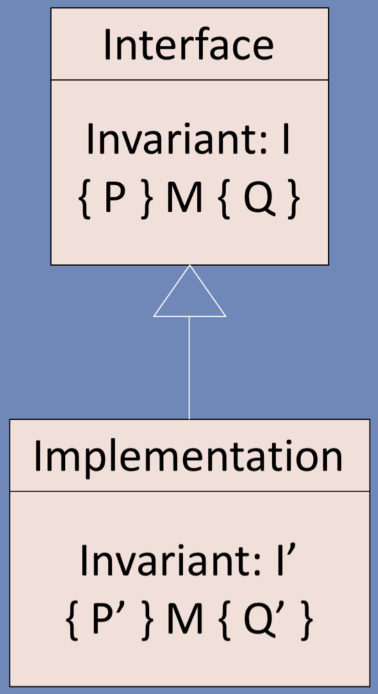
- P implies Q

Subcontracting



Subcontracting dictates *relative strength* of P/P’, I/I’, Q/Q’

- Postcondition Q’
  - **Stronger** than Q.
  - *Ensure no less*
- Precondition P’
  - **Weaker** than P
  - *Require no more*
- Invariant I’
  - **Stronger** than I



The Liskov Substitution Principle

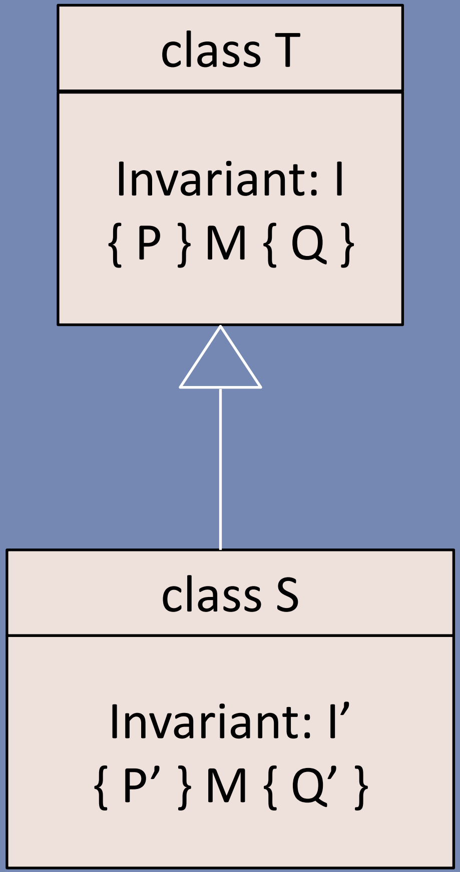
If you use a class T,

you should be allowed

to substitute T

by *any* subclass of S of T

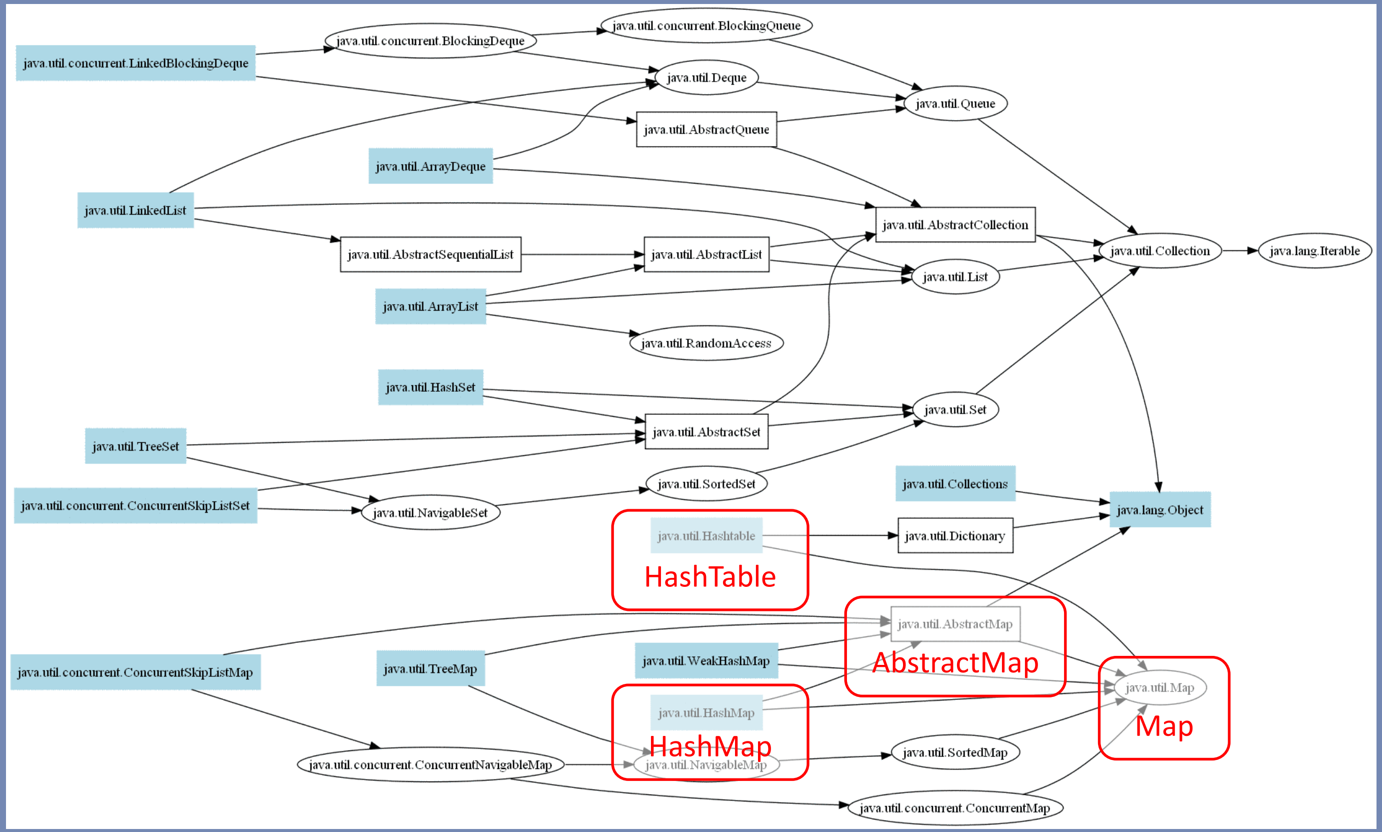
Sub-contracting formalizes this principle



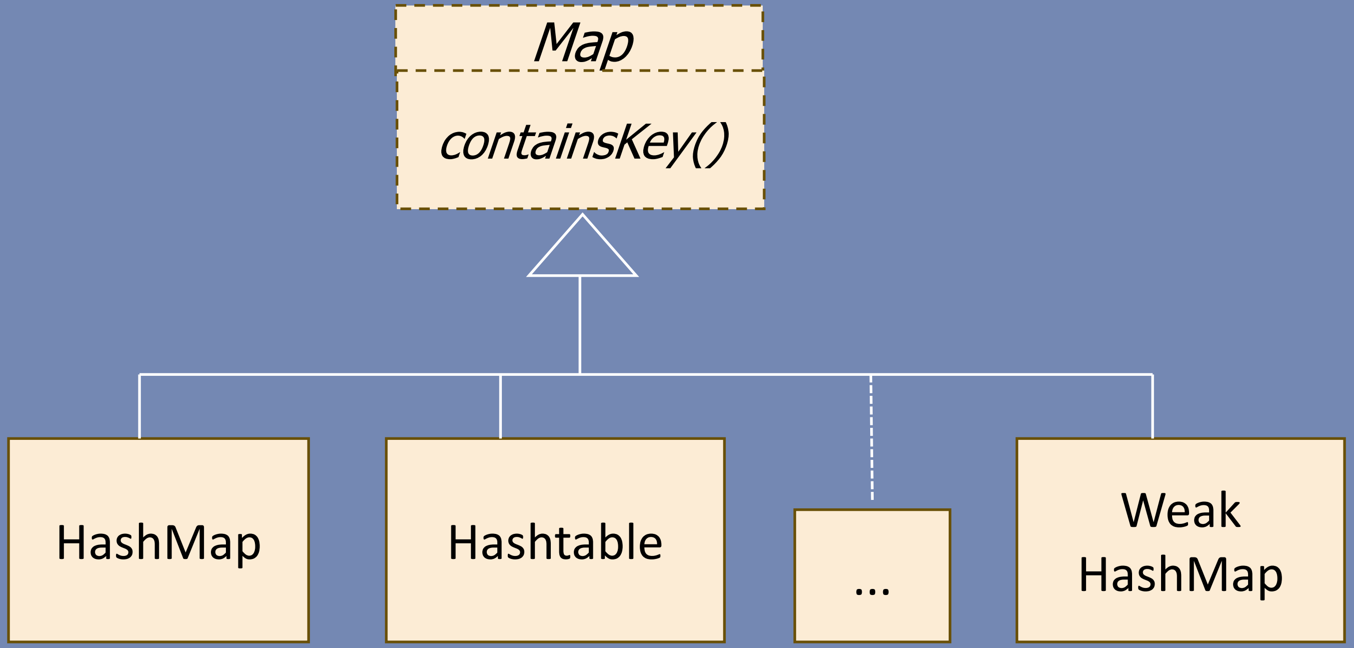
Design By Contract

- Interface is a contract
  - Ensures (promises) certain effects will happen
  - Provided certain assumptions are true
- Its implementation is a subcontract
  - Promises at least the same effects
  - Under at most the same assumptions
  - “Require no more; Ensure no less”
- Formalize with assertions

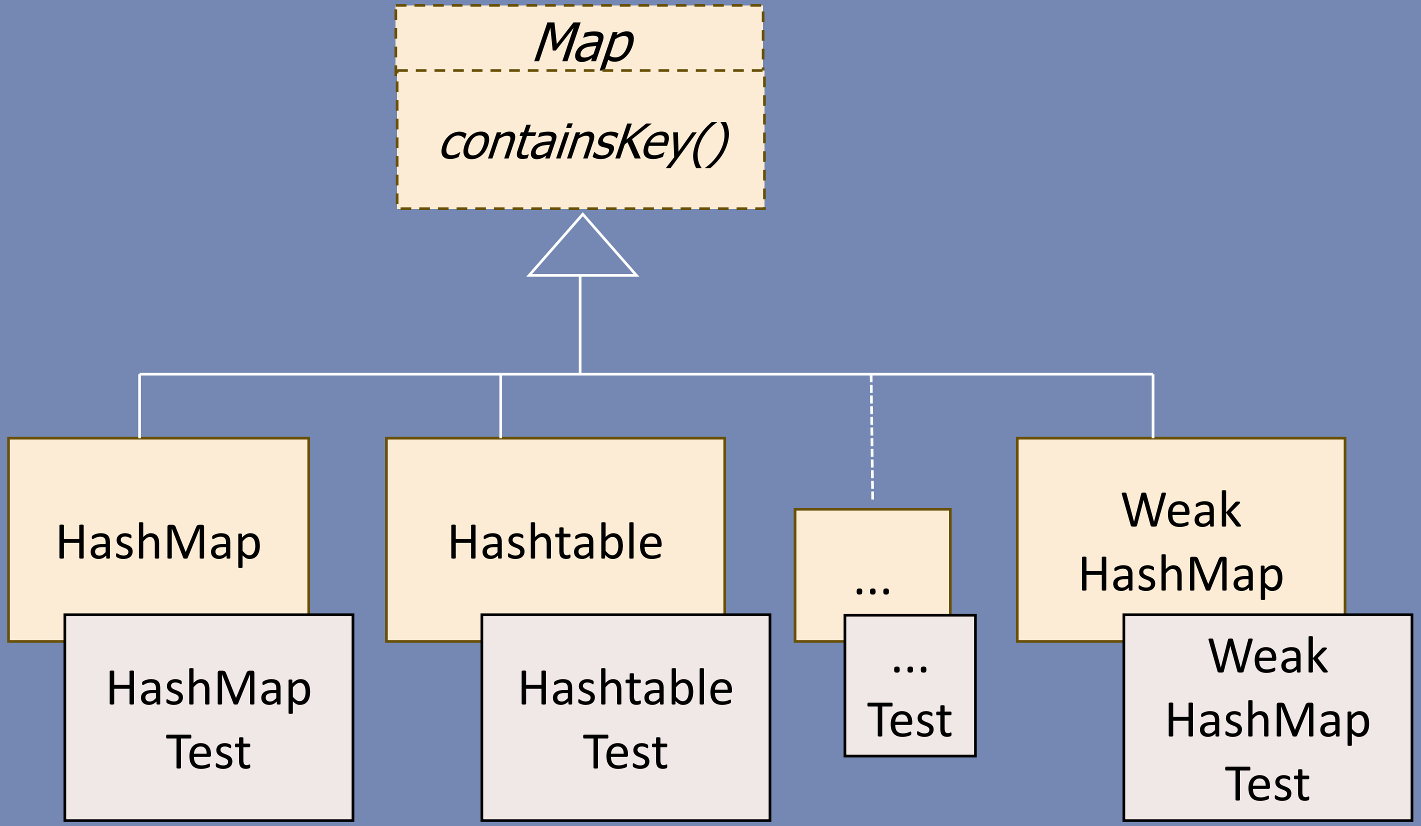
Testing for LSP Compliance



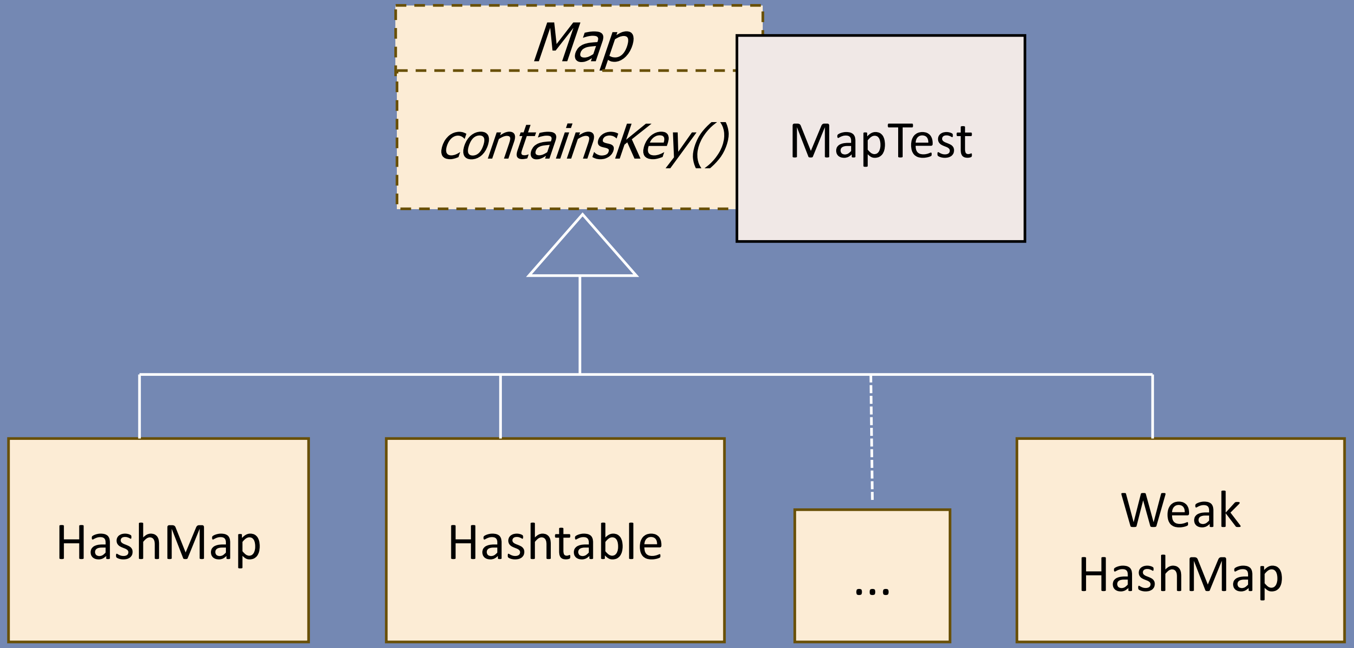
Example Class Hierarchy



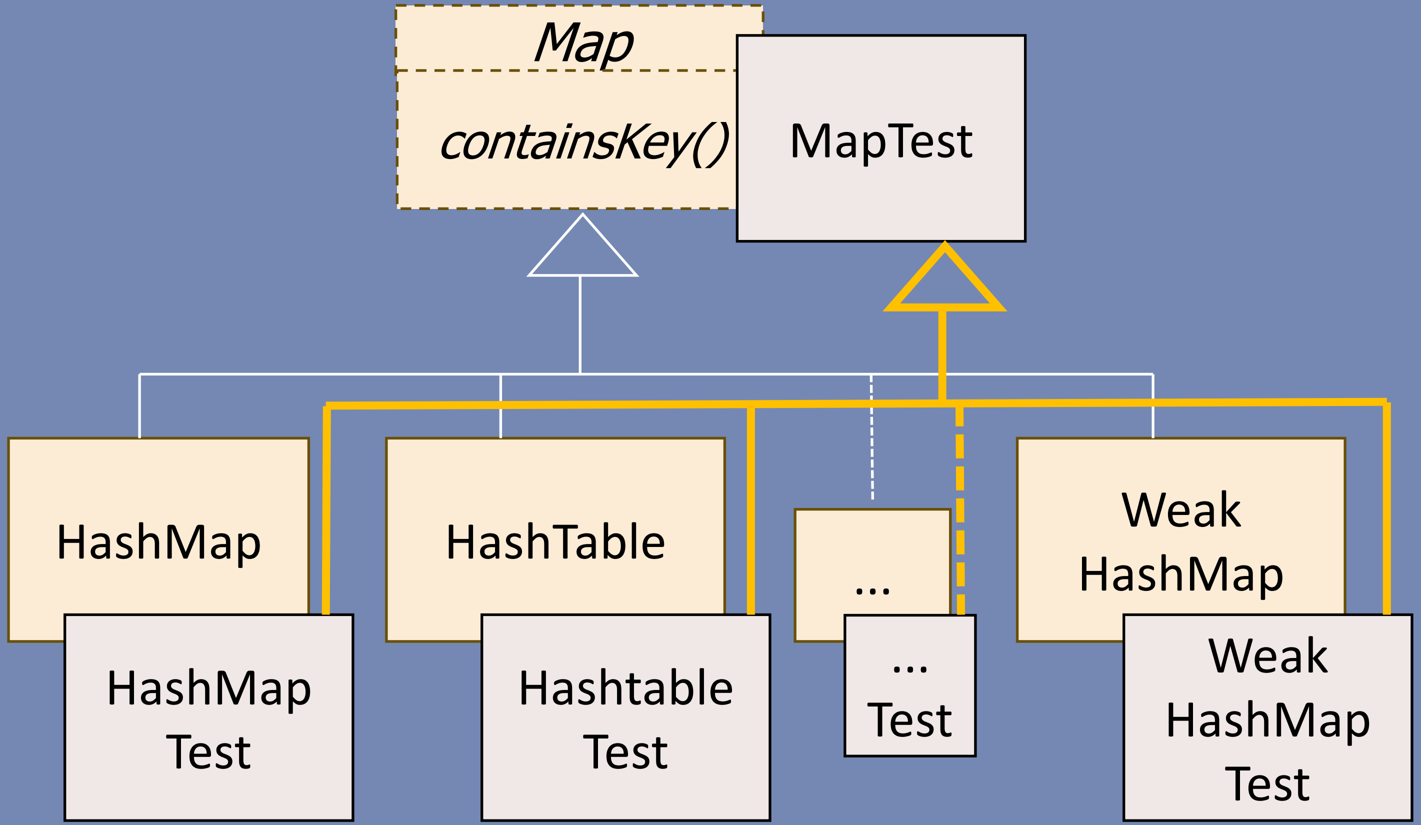
Testing Subclasses



Testing The Superclass

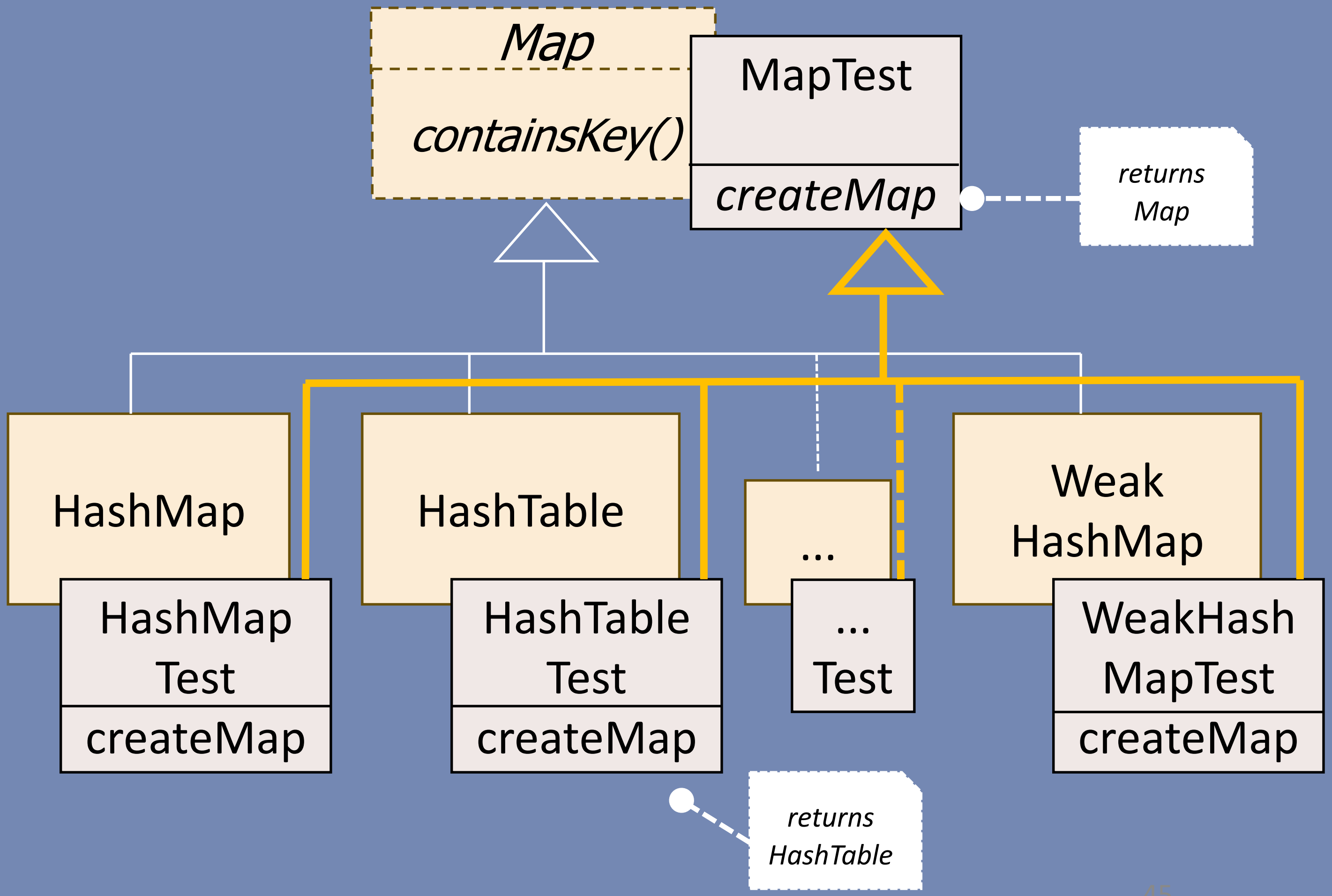


A Parallel Hierarchy for Testing



John McGregor: A Parallel Architecture for Class Testing (PACT)

Using Factory Methods

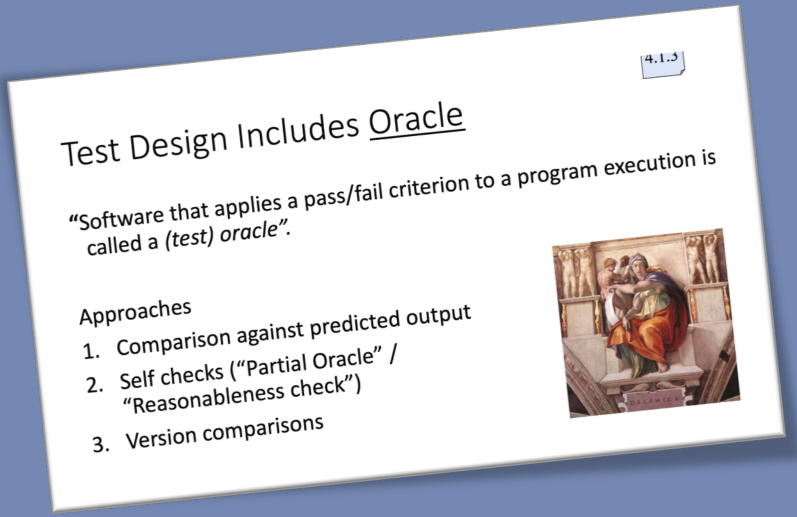


Testing for LSP Compliance

- Design test suite *T* at (top) *interface* level
- Reuse for all interface implementations
- Specific implementation may require additional tests, but should at least meet *T*.

Robert Binder (2000): “Polymorphic Server Test”

Test Oracles



“Software that applies a pass/fail criterion to a program execution is called a *(test) oracle*”.

Approaches

1. Value comparison
2. *Property checks*
3. Version comparisons

In Code Assertions as Oracles

- Enable run time assertion checking during testing
  - Post-conditions check method outcomes
  - Pre-conditions check correct method usage
  - Invariants check object health

- Run time assertions increase fault sensitivity
  - Increase likelihood program fails if there is a fault
  - Desirable during testing!

Assertions don’t Replace Testing

- Unit tests still needed to exercise methods
- In code assertions only check general properties
- In code assertions on top of asserts with concrete expected values in tests

Assertions Inspire Testing

- Test inputs should reach assertions
- Assertions may be disjunction P1 or P2
  - Test inputs should trigger both alternatives
- Assertions may contain boundaries
  - Test inputs should trigger those boundaries

Assertions Don’t Fail

- Test inputs can reach assertions
- Test inputs cannot make assertions fail
  - That would be a bug in the program!
- No need to write test cases that let pre-conditions fail
  - Method behavior undefined !

Property-Based Testing

- Think of ”properties” (assertions) for functions
- Let “generator” produce series of random input values for function
- For each random input check the assertions.

```java
import com.pholser.junit.quickcheck.Property;
import com.pholser.junit.quickcheck.runner.JUnitQuickcheck;
import org.junit.runner.RunWith;

import static org.junit.Assert.*;

@RunWith(JUnitQuickcheck.class)
public class StringPropertiesTest {

    @Property
    public void concatenationLength(String s1, String s2) {
        assertEquals(
                s1.length() + s2.length(),
                (s1 + s2).length());
    }
}
```

Quickcheck generates 100 random strings to check this property.

Property: length of concatenated strings equals sum of length of individual strings

```java
package com.pholser.junit.quickcheck.examples.crypto;

import ...

@RunWith(JUnitQuickcheck.class)
public class SymmetricKeyCryptoPropertiesTest {

    @Property
    public void decryptReversesEncrypt(
        @InCharset("UTF-8") String plaintext,
        Key key)
        throws Exception {

        SymmetricCrypto crypto = new SymmetricCrypto();

        EncryptionResult enciphered =
            crypto.encrypt(plaintext.getBytes("UTF-8"), key);

        assertEquals(
            plaintext,
            new String(crypto.decrypt(enciphered, key)));
    }
}
```

QuickCheck Ingredients

- Property specification language / library

- Data input generator for range of data types
- Mechanism to write your own data generators
- Mechanism to constrain data generated (junit assume)

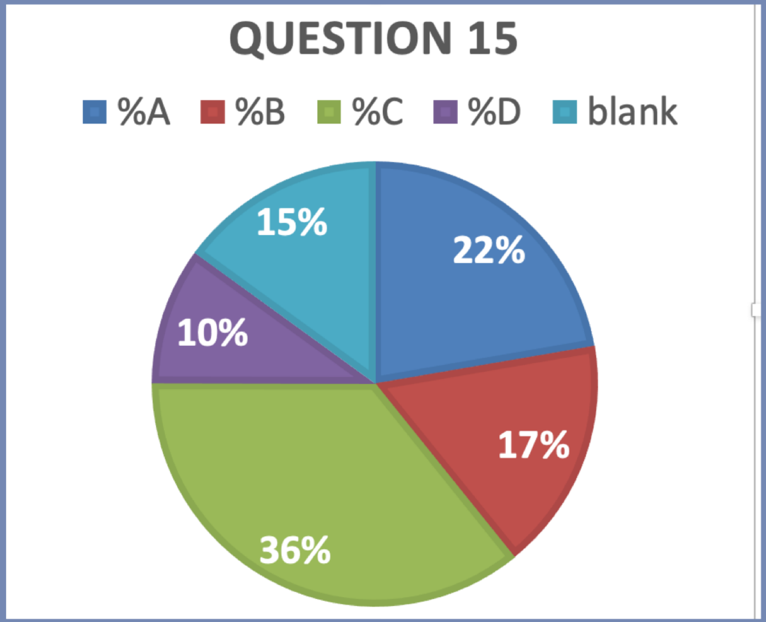
- Shrinking process to reduce inputs for failing tests to smallest data

Automated Self-Testing

- Random input generation:
  - Exercise system in variety of ways
  - Clever generators for specific data types
- Whole test suite perspective:
  - Maximize coverage achieved by inputs
  - Capture in fitness function
  - Evolutionary search for fittest test suite
- Properties, contracts, assertions:
  - The oracle distinguishing success from failure

Mid-Term Question 15

Which of the following statements is correct about the relationship between specification-based testing and structural testing?



A. Boundary analysis can only be done if testers have access to the source code, and thus, it should be considered a structural testing technique.

B. Model-based testing is a structural testing technique.

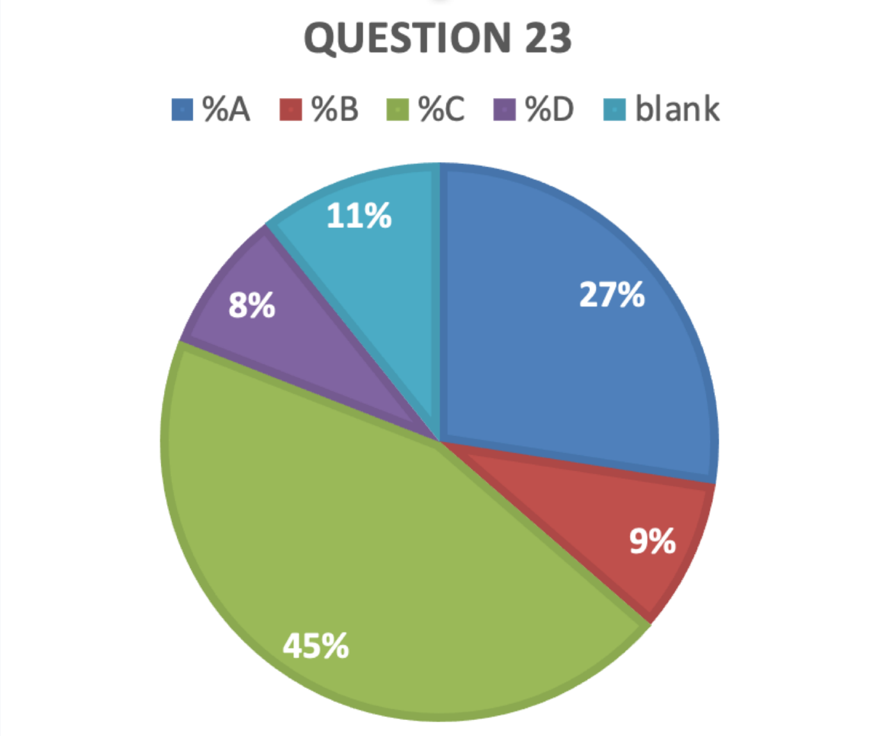
C. None of the other answers is true.

D. If we take costs into account, a testing process should then prioritize structural testing because it’s cheaper and yet highly effective.

Mid-Term Question 23

- You implement a decision table directly via if-then-else logic in your code. For which of the following decision table testing strategies are you guaranteed to achieve 100% branch coverage of the corresponding decision logic in your code?

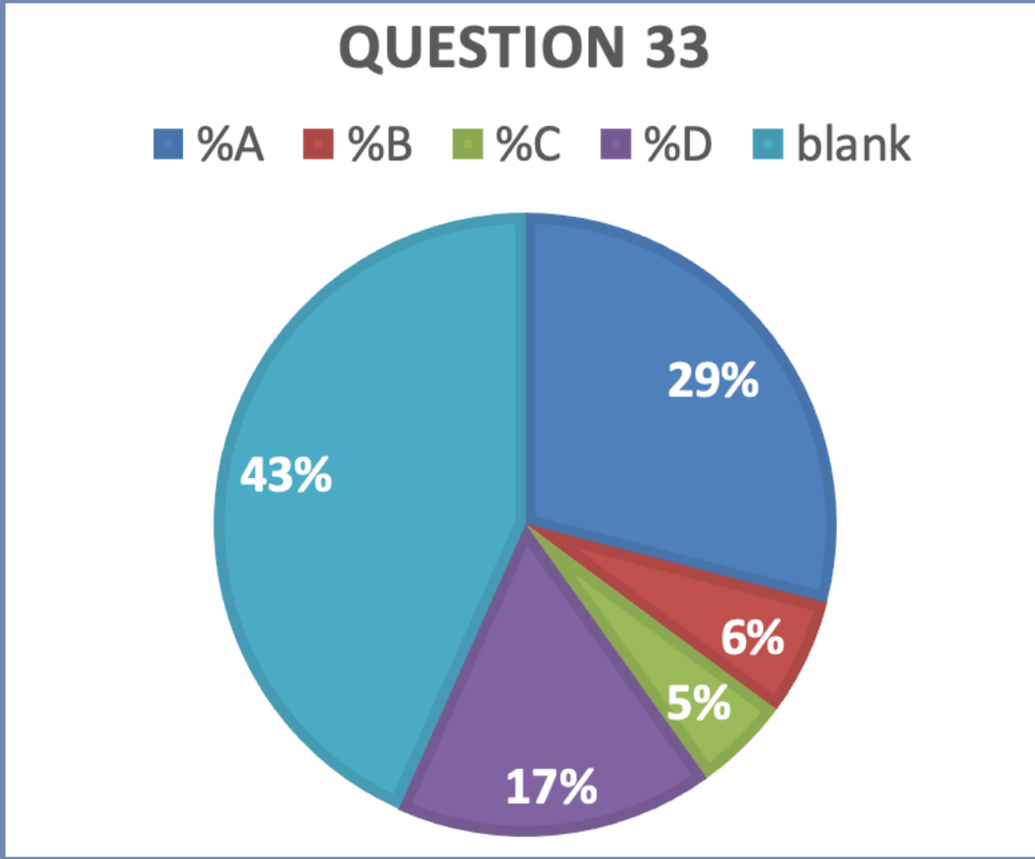
A. All possible variants
B. Each condition
C. MC/DC
D. All explicit variants



Mid-Term Question 33

A static analysis checking a *non-trivial property* is typically:

A. Sound but Imprecise
B. Unsound and Imprecise
C. Sound and Precise
D. Unsound but Precise



False Positive / Negative

- Many static analysis tools based on *heuristics*
  - Correct positive: Warning, and a problem (let's fix it! ☺)
  - Correct negative: No warning, no problem. (no need to act ☺)
  - False positive: Warning, but not a problem (annoying ☹)
  - False negative: Problem, but no warning (possibly dangerous ☹)

Poor Precision

Poor Recall

Static Analysis

- Soundness
  - No missed vulnerability (0 FNs)
  - No alarm → no vulnerability exists
- Completeness
  - No false alarms (0 FPs)
  - Raises an alarm → vulnerability found

Full Recall

Full Precision

- Ideally: ↑Soundness + ↑Completeness
- Reality: Compromise on FPs or FNs

73

Static Analysis Uses Terminology from Logic

- We want to prove that bad property X (e.g. injection attack) cannot occur
- Soundness:
  - A sound logic proves only true things
  - The conclusion “X cannot occur” can be trusted.
  - *No false negatives – full recall.*
- Completeness:
  - A complete logic proves all true things
  - The conclusion “X cannot occur” will be drawn for *all* programs for which this is true
    - In other words: The conclusion “X can occur” can be trusted (and should be acted upon).
  - *No false positives – full precision.*

https://courses.cs.washington.edu/courses/cse341/13wi/unit6notes.pdf

Question 40

QUESTION DISCARDED

Regarding the boundary analysis technique discussed in lecture, which of the following statements is true?

A. There can only be a single on-point which always m [illegible] on true.

B. There can be one or two off points which may or may [illegible] e the condition false.

C. There can be multiple on-points for a given condi [illegible] ich [illegible] may not make the condition true.

D. There can be multiple off-points for a given condition which always make the condition false.

- A problem is **decidable** if there exists an algorithm to solve it that is sound, complete, and terminating.
- Soundness means that the algorithm never returns “yes” when it shouldn’t
- Completeness means it always returns “yes” when it should.

Outlook

- Lectures:
  - Annibale Panichella (fuzzing, search-based testing)
  - Tim van der Lippe (open source)
  - Stéphane Nicoll (Pivotal, Spring)
- Reviewing part 2
- Labwork part 3
- Exam