UNIVERSITY OF ICELAND

HBV401G SOFTWARE DEVELOPMENT

12. Software Testing

Matthias Book

Spring 2022

FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL ENGINEERING AND COMPUTER SCIENCE

In-Class Quiz #11 Solution: Software Testing Concepts

- **Indicate which concepts are described by the following definitions – Test Cases, Test Fixtures, Test Oracles, Test Subjects or Mock Objects?**

a) Test Oracle: An entity determining what a test subject's expected "correct" output for a particular input is supposed to be.

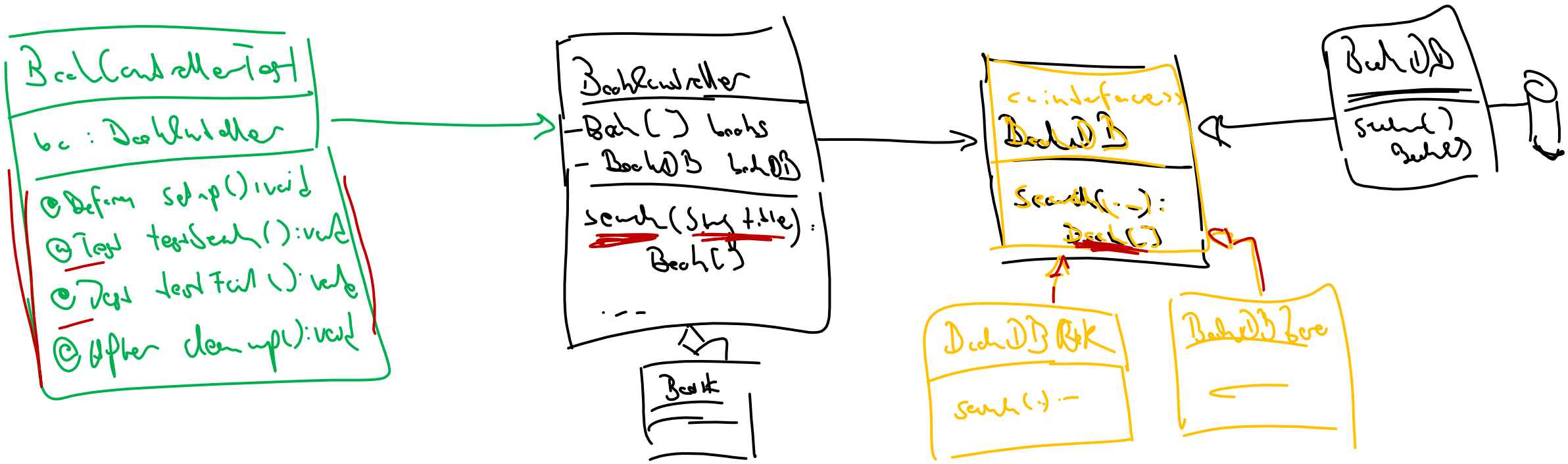
b) Test Fixture: A piece of code exposing a test subject to a particular test case.

c) Mock Object: A piece of code that simulates certain behavior of other, not yet implemented code that a test subject is relying on.

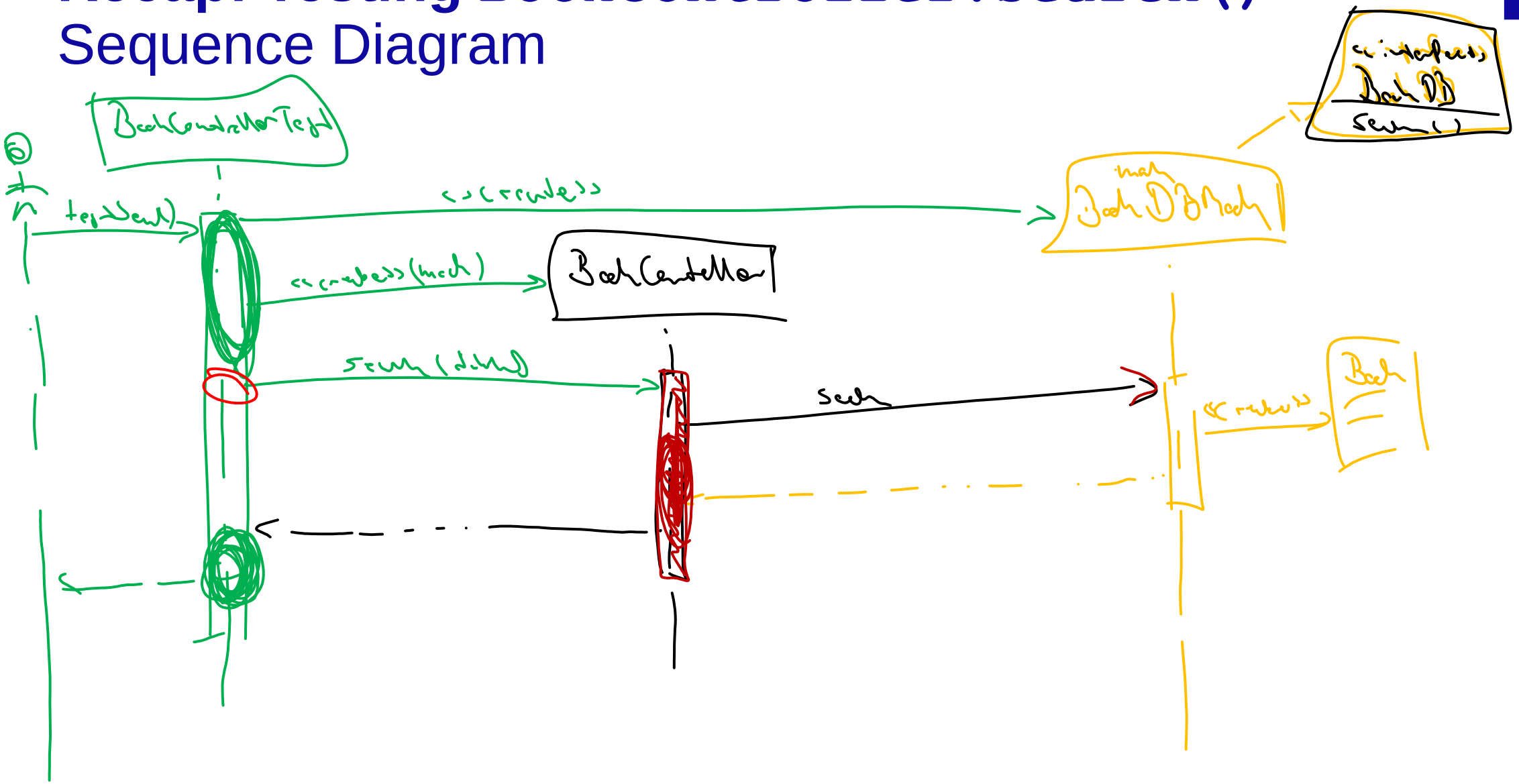
d) Test Subject: A piece of code whose correctness / agreement with specifications is being tested.

e) Test Case: A specific output/result that is expected to be produced by a test subject when exposed to a specific input.

Recap: Testing `BookController.search()`
Class Diagram



Recap: Testing `BookController.search()`
Sequence Diagram



Testing Strategies

Testing a Method that Returns Nothing (e.g. a Constructor or `void` Return Type)

- A method returning nothing should still have some side effect (otherwise it's pointless!)

➢ Test for the presence of the side effect!

- Example:

```
public class A {
   private int x;
   public void setX(int x) {
      this.x = x;
   }
   public int getX() {
      return x;
   }
}
```

```
@Test
public void testSetX() {
   a.setX(42);
   assertEquals(42, a.getX());
}
```

Testing for `setX`'s effect on the attribute `x`

Should We Test Getters and Setters?

- If their implementation is trivial, you typically don't need to:

For a trivial implementation such as…

```
public class A {
  private int x;
  public void setX(int x) {
    this.x = x;
  }
  public int getX() {
    return x;
  }
}
```

```
@Test
public void testGetSetX() {
  a.setX(42);
  assertEquals(42, a.getX());
}
```

…is pretty much the same as…

```
@Test
public void testGetSetX() {
  assertEquals(42, 42);
}
```

- But if they perform any input validation/output transformation, you likely want to.

Testing if Exceptions Are Raised

- Suppose a constructor is expected to raise an **`IllegalArgumentException`** if it receives an empty currency string:

```
@Test(expected=IllegalArgumentException.class)
public void testIllegalConstructorArgument() {
  Money undef500 = new Money(500, "");
}
```

Will fail if the exception is not raised in this situation.

- Note we need another test to ensure the exception is only thrown with reason!

```
@Test
public void testLegalConstructorArgument()
    throws IllegalArgumentException {
  Money isk500 = new Money(500, "ISK");
}
```

Will fail if the exception is raised in this situation.

Note: Only both tests together ensure correct exception behavior!
One test alone could be satisfied by a method that always/never throws an exception, even though that would not be our desired behavior.

Testing for `null` References

- Should we test for proper treatment of **`null`** references in these methods?

```
public int convertTo(String toCurrency) {
  return converter.convert(amount, currency, toCurrency);
}
```

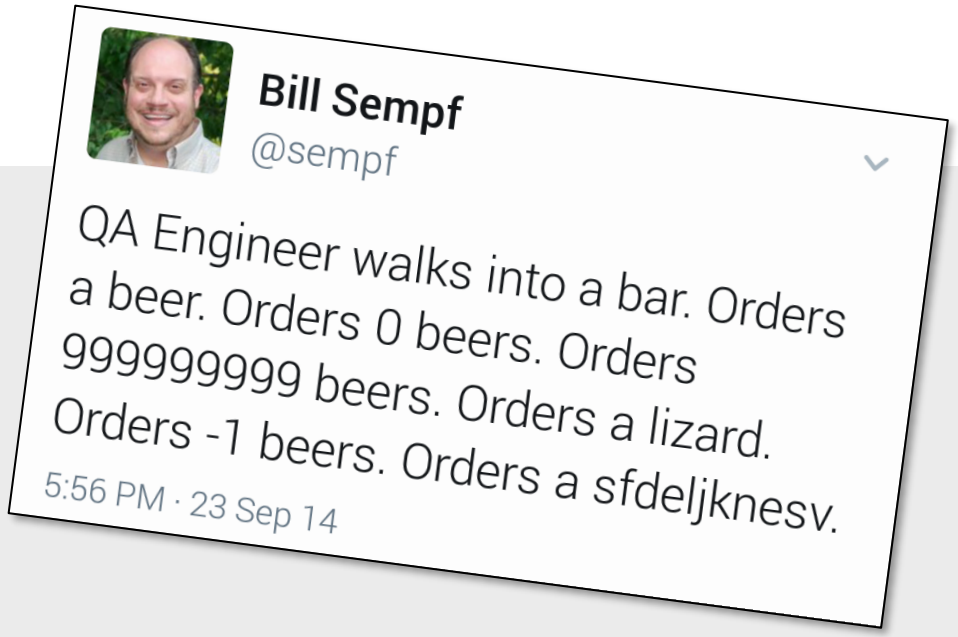
- No – making sure we have a reference to a **`converter`** object is the job of the constructor (or whoever provides the **`converter`**), and should be tested for there.
  - This method should be able to rely on that, and does not need to do any exception handling.

```
public Money add(Money summand) {
  return new Money(amount + summand.getAmount(), currency);
}
```

- No – if the **`summand`** is **`null`**, the JVM's default behavior of throwing a **`NullPointer-Exception`** is the only sensible reaction.
  - As Java's default behavior, no explicit implementation or test is necessary at this point.
  - We may want to test that no **`null`** references can be produced at the **`summand`**'s origin though.

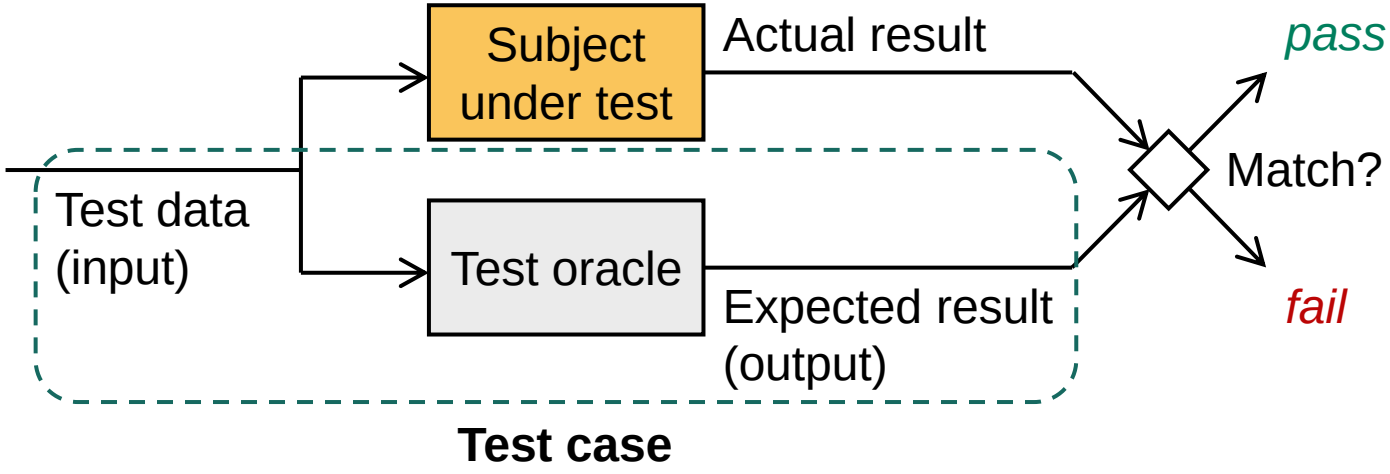
UNIVERSITY OF ICELAND

Test Coverage:
Equivalence Partitioning, Boundary Value Analysis



Choosing Appropriate Test Data

- The test subject is exposed to the test data, and the actual result is compared to the expected result.



- Question: How do we choose appropriate test data? – i.e:
  - Test data that covers all possible scenarios that the test subject could experience
    - Black-box view: Test cases designed to cover required behaviors
  - Test data that covers all possible ways in which the test subject can behave
    - White-box view: Test cases designed to cover implemented execution paths
- Problem: It is usually impractical/impossible to achieve 100% coverage.
  - ➢ What is the most economic subset to choose?
    - i.e. **how can we find the most defects with the least effort?**

e.g. equivalence partitioning, boundary analysis *(up next)*

various coverage criteria *(beyond scope of this course)*

Equivalence Partitioning

Note: These are not object-oriented classes, but mathematical sets of input values

- **Problem:** Typically, it is impractical/impossible to expose the test subject to all possible combinations of input values.
- **Idea:** Partition the set of all possible input values into “equivalence classes”, i.e. subsets for which the test subject’s behavior will be the same.
  - i.e. we want to identify subsets of input values for which the test subject will behave the same (e.g. a given test will pass for any member of the subset, or fail for any member).
  - Once we have identified these equivalence classes *for a particular test*, we only need to test one representative of each equivalence class instead of the whole set of input values.

- **Testing procedure**
  1. Determine conditions imposed on the input data (should be found in specification docs)
  2. Partition overall input value range into equivalence classes
  3. Pick a representative from each equivalence class to serve as test input

Examples for Determining Equivalence Classes

- For a given test, we want to find [combinations of] input values for which the test yields the same result (i.e. passes or fails).
- Various strategies can be used, depending on the kind of input value range:

- **Ordered input value range**
  - The range of acceptable values forms a “valid input” class
  - The ranges of values below and above the acceptable values form “invalid input” classes
  - Example:
    - Complete value range: $i \in \mathbb{Z}$, acceptable values: $1 \leq i \leq 10$
    - “valid input” equivalence class: $1 \leq i \leq 10$
    - “invalid input” equivalence classes:
      - $i < 1$
      - $i > 10$

Examples for Determining Equivalence Classes

- **Collections of objects**
  - Example: A list may contain between 1 and 25 elements
  - “valid input” equivalence class: all lists containing between 1 and 25 elements
  - “invalid input” equivalence classes:
    - an empty list
    - all lists containing over 25 elements
    - a `null` reference
- **Distinction of individual input values**
  - Example: Same behavior for inputs ‘a’, ‘b’ and ‘c’, but different behavior for ‘x’ and ‘y’
  - “valid input” equivalence classes:
    - {‘a’, ‘b’, ‘c’}
    - {‘x’, ‘y’}
  - “invalid input” equivalence class: all other inputs

Constructing Test Input Data

- Pick / construct a combination of test input data that covers *as many* **“valid input” equivalence classes** *as possible*
  - (i.e. the [components of the] test case’s input data should be representative[s] of as many “valid input” equivalence classes as possible)
  - Repeat this as long as there are still “valid input” equivalence classes from which no representative has been used yet

- Pick / construct a combination of test input data that covers *one* **“invalid input” equivalence class**
  - Repeat this as long as there are still “invalid input” equivalence classes from which no representative has been used yet

Example: Equivalence Partitioning
Steps 1 & 2: Construct Equivalence Classes

- We want to test the method `String read(String filename, int n)` that is supposed to return the first `n` lines of the file with the given `filename`.
- **Step 1: Determine conditions imposed on input data:** According to specs,
  - the filename should consist of 1 to 6 characters (numbers or letters)
  - the first character of the filename should be a letter
  - the number of requested lines should be greater than 0 and less than 1000

- **Step 2: Partition equivalence classes**

Input condition	"Valid input" equiv. classes	"Invalid input " equiv. classes
Filename length	1. 1 to 6 characters	a) Empty filename b) More than 6 characters
Filename characters	2. Numbers or letters	a) Other characters
1st filename character	3. Letter	a) No letter
Number of lines	4. > 0 and < 1000	a) ≤ 0 b) ≥ 1000

Example: Equivalence Partitioning
Step 3: Construct Test Cases

Test case covering all “valid input” equivalence classes:

- `read("abc1", 42);` (1)–(4)

Test cases covering individual “invalid input” equivalence classes:

- `read("", 42);` (1a)
- `read("abc1234", 42);` (1b)
- `read("a+", 42);` (2a)
- `read("21a", 42);` (3a)
- `read("abc1", -21);` (4a)
- `read("abc1", 2100);` (4b)

Input condition	“Valid input” equiv. classes	“Invalid input ” equiv. classes
Filename length	1. 1 to 6 characters	a) Empty filename b) More than 6 characters
Filename characters	2. Numbers or letters	a) Other characters
1st filename character	3. Letter	a) No letter
Number of lines	4. > 0 and < 1000	a) ≤ 0 b) ≥ 1000

How Representative are the Representatives?

- Mathematically, it seems sufficient to pick just one representative from each equivalence class:
  - If our code should accept $1 \leq x \leq 100$, isn't it sufficient to test just e.g. –50, 50, 150?
  - If our implementation is correct, these values are indeed representative of their eqv. classes

- However, if our implementation is defective, it might actually imply different equivalence classes!
  - If a defective implementation rejects the input 42 for some reason, testing with the input values –50, 50 and 150 does not detect this discontinuity!

➢ Equivalence partitioning cannot guarantee the absence of defects.
  - But (testing economics): Putting such a discontinuity into the code by accident is rare, so we typically can have reasonable confidence in the result.

Boundary Value Analysis

- Still, we should be aware of parts of our code that could introduce such discontinuities, or that imply different equivalence classes than we intended, and test if we treated them correctly.
- Such defects can hide anywhere, but are most easily introduced inadvertently at the boundaries of our intended equivalence classes. Typical examples:
  - Loop counters
  - Array boundaries
  - Empty strings
  - Empty collections
  - Last days of a month
  - Overflows
  - `null` pointers
- Testing economics: Usually worth testing if boundaries are treated correctly!
  - So for the requirement $1 \leq x \leq 100$, we would e.g. test –50, 0, 1, 2, 50, 99, 100, 101, 150

Quiz #12: Equivalence Classes & Boundary Values

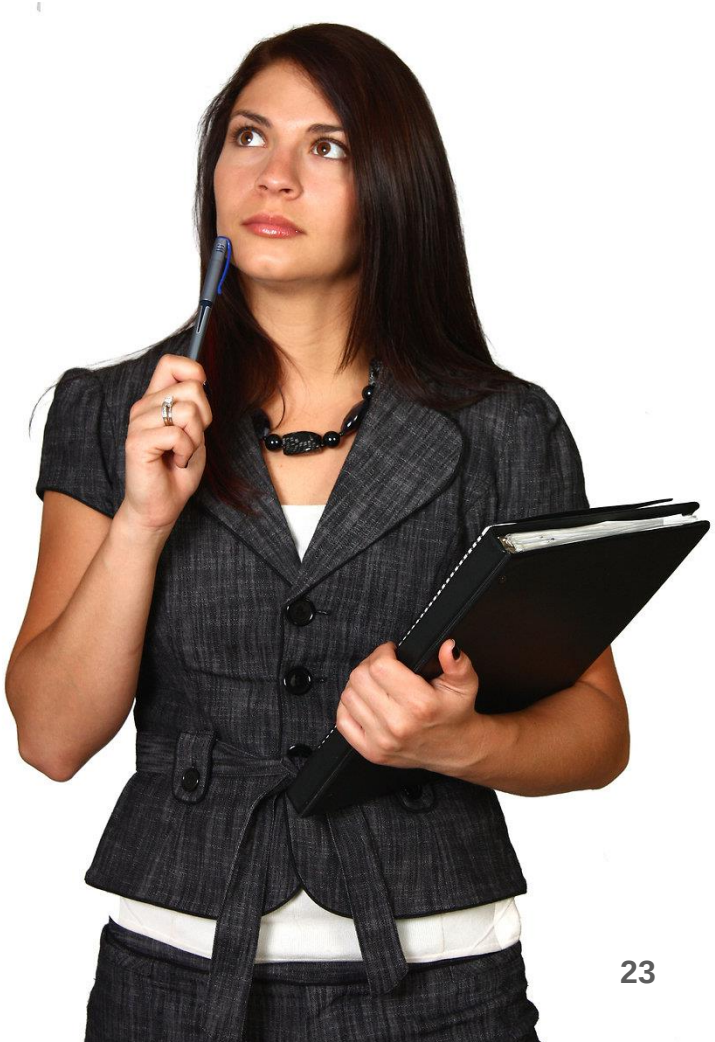
- We want to test the method `String read(String filename, int n)`, where
  - the filename should consist of 1 to 6 characters (numbers or letters)
  - the number of requested lines should be greater than 0 and less than 1000
- Indicate which statements are well-constructed test cases for the “valid input” or “invalid input” equivalence classes, and which are poorly constructed test cases:

```
a) String s = read("file", -1);
b) String s = read("file", 0);
c) String s = read("file", 1);
d) String s = read("filename", 999);
e) String s = read("filename", 1000);
f) String s = read("filename", 1001);
```

Break

Defensive Programming

see also: J. Bloch, Effective Java



Why Programs Fail – And How to Prevent It

- Localized bugs in some algorithm, e.g.
  - mistyped operators
  - boundary overruns in loops, arrays etc.
- Violations of consistency assumptions in data structures, often due to
  - lax visibility rules
  - lack of enforcing invariants
- Side effects from using methods in ways they were not intended to, often due to
  - violations of implied protocols in object collaboration
  - overriding an inherited method with behavior that clashes with expectations of superclass
- Unforeseen system conditions, e.g.
  - Failure/unavailability of components
  - Corruption of data
- Unforeseen user activities, e.g.
  - Entering invalid data
  - Performing malicious activities

Can be caught early by test-driven development

Can be prevented by defensive programming

Can be handled with exceptions

Can be identified through testing (more or less easily)

And of course (remember first lectures):
Not satisfying user requirements
(Can be minimized through constant feedback)

Enforcing Consistency Through Encapsulation

- **An object is responsible itself for always being in a valid state.**
  - i.e. its methods must guarantee that its attributes can't contain invalid [combinations of] data
- Use constructors to initialize attributes validly right from the creation of an object.
- Use setters to ensure that attributes are set within valid range.
- Make sure that the class' other methods maintain attributes within valid range.
- Check if incoming parameters are valid (within reason – "know who to trust")
- Throw an exception if an operation would put your object in an invalid state.
- Use the `final` modifier on any attribute, parameter or variable that you don't intend to change after its initialization.
  - This is not just for constants, but can actually be applied to many parameters and variables.
  - Note that for reference types, `final` just means the object reference in the variable can't be changed anymore – the attributes within the object can still be changed.

Enforcing Consistency Through Encapsulation

- **Prefer working with objects over working with primitive data types.**
- Don’t rebuild the reference (pointer) mechanism.
  - Don’t give objects unique IDs and refer to them by ID – use the object references instead.
    - When dealing with objects coming from databases, you might still need the primary key to retrieve them – but consider whether there’s a natural way to keep the object in memory between uses.
- Consider if it is more natural…
  - …to pull lots of primitive data from object A and feed it to object B for processing
  - …or to give object B a reference to object A and let it ask for the data it needs itself
- Within an object, store data in a way that balances…
  - …the real-world structure of the application domain
    - This will keep your structure domain-oriented, technology-independent, and more maintainable
  - …and the technical structures that are most efficient for working with the data
    - This will make data access more straightforward – but don’t get too hung up on optimization!
- Use getters to transform data from your attributes as needed by other classes.
- Return empty collections instead of `null` references when a collection is expected.

Enforcing Consistency Through Encapsulation

- **“Use the scope, Luke.” (Put any variable in the narrowest possible scope.)**
- Method variables should be used for any data that is only relevant within a method.
  - They should be declared in the block where they are required – this way they are destroyed after the block and can’t live on to create side effects
    - e.g. don’t declare a loop’s count variable at the beginning of the method, but in the loop head
- Instance variables (attributes) should be used only for data that is shared among the methods of one class, and that constitutes the state of the object.
  - Use `private` visibility to encapsulate attribute values within the object.
- Class variables (`static` attributes) are intended only for data that is shared among all instances of a class, or that shall be globally visible in the program.
  - This is very rarely necessary! Avoid unless you have a good reason.
    - Typically used e.g. for singletons or publicly visible constants
  - If you do this only because you don’t know how else to access some data from somewhere, rethink your design model!

Ensuring Independence Through Low Coupling

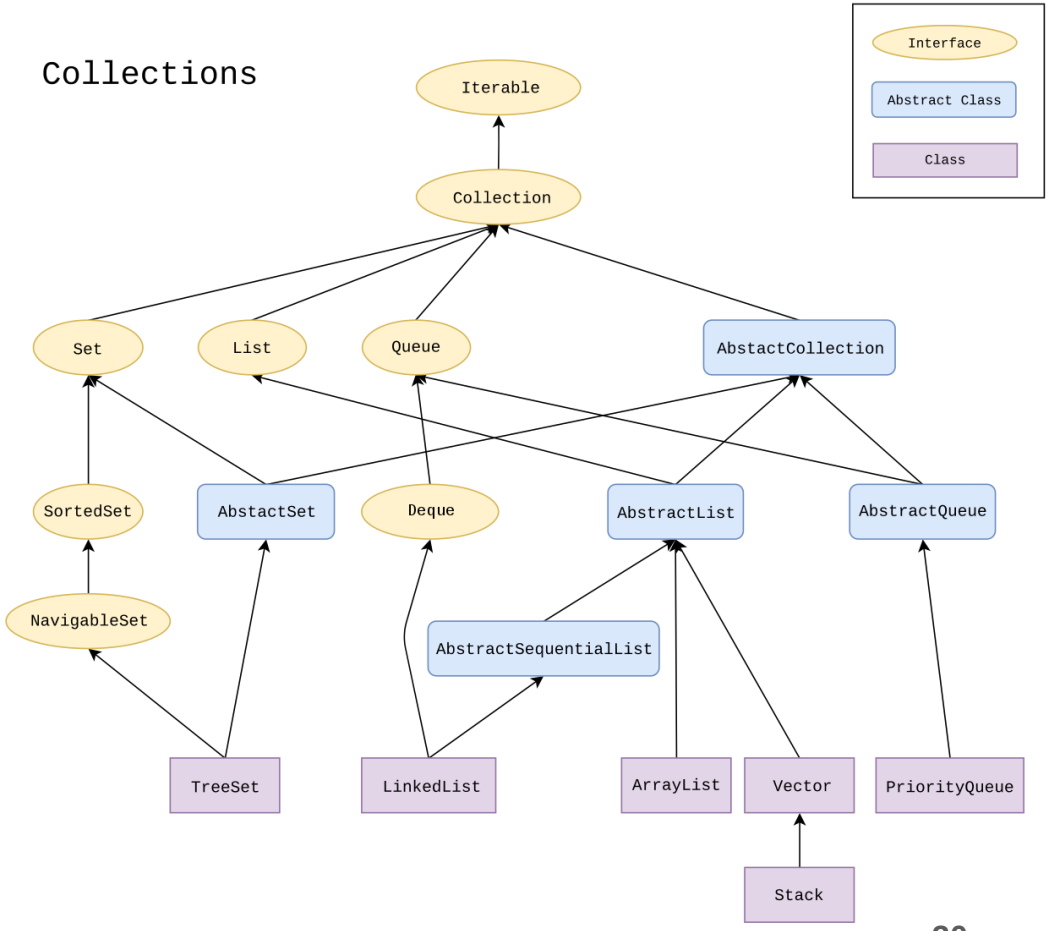
- **Avoid any protocol beyond the method signature.**
- Other classes should normally not have to execute a particular set of your methods in a particular order to accomplish certain functionality.
  - Consider if you can do part of this automatically (e.g. by calling private methods).
- Don’t create a control language.
  - Don’t let methods react significantly differently to various input strings or input constellations.
    - Use overloaded methods instead; possibly rely on polymorphism.
  - Avoid signaling special conditions with special input/output parameters tucked into corners of the regular value range, e.g.
    - Special “optional” input parameters – consider overloaded methods, model “mode” explicitly
    - Special output values to signal error conditions – throw exceptions instead
- Your method signatures should be…
  - generic enough to give you the freedom of changing your internal implementation
  - close enough to your internal implementation to avoid awkward conversions
  - naturally representing the concepts of your application domain
- If you need to implement the same operation in similar ways, use an `interface` to describe the behavior, and implement the variants in several classes derived from it

Ensuring Independence Through Low Coupling

- **Use methods in the context they are intended for.**
- Use visibility modifiers to make methods accessible only to intended audience.
- When overriding a method, ensure the new implementation satisfies all expectations the original implementation satisfied (Liskov Substitution Principle), i.e. behaves as expected…
  - when called from other methods that were inherited from the same superclass
  - when called by other classes who assume that this is an instance of the superclass
  - when calling other methods that the superclass' original method used to call
- Use `final` modifier to prevent a method from being overridden in subclasses.
  - Such methods can still be inherited, but not specialized anymore in subclasses.
- Consider using delegation / composition instead of inheritance to avoid some of the above access issues
  - Especially when the classes don't really have an "is-a" relationship, but just share code
  - Use inheritance when you want to make use of polymorphism

Example: Java Collections Framework

- Check out the documentation and API specification of the Java Collections Framework for good examples on how to encapsulate functionality, and how to design and document method signatures:
  - https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html
- The Collections Framework consists of
  - Collection interfaces
  - Collection implementations
    - General-purpose, wrapper, adapter, convenience, legacy, special-purpose, concurrent, abstract
  - Algorithms
  - Infrastructure
  - Utilities



Cluster Assignment #5: Final Product
(Presentations & Submissions for Clusters with a T Team)

- On **Wed 20 Apr**, present **your cluster's integrated product** to your classmates
  1. A demonstration of your cluster's integrated product (live, ~5 min)
  2. An overview of your cluster's system architecture (slides, ~5 min)
  3. A retrospective on your cluster's project work (slides, ~5 min)
  4. Q&A (~5 min)

- By **Sun 24 Apr**, submit in Canvas:
  1. A ZIP archive with the code of your cluster's integrated product
  2. A PDF document with the slides of your cluster's presentation

- Note the presentation comes before the submission for this assignment!
  - Don't fix/optimize your product after the presentation; your presented state counts!
  - ➢ Simply submit what you presented

Cluster Assignment #5: Final Product
(Presentations & Submissions for Clusters without a T Team)

- On **Wed 20 Apr**, present **your team's product** to your classmates
  1. A demonstration of your team's product (live, ~2 min)
  2. An overview of your team's system architecture (slides, ~2 min)
  3. A retrospective on your team's project work (slides, ~2 min)
  4. Joint Q&A with other teams in cluster (~2 min)

- By **Sun 24 Apr**, submit in Canvas:
  1. A ZIP archive with the code of your team's component
  2. A PDF document with the slides of your team's presentation

- Note the presentation comes before the submission for this assignment!
  - Don't fix/optimize your product after the presentation; your presented state counts!
  - ➢ Simply submit what you presented

Cluster Assignment #5: Grading

- **Grading criteria**
  - **Demo:** Key features (searching, booking, reduced availability) are functional and accessible via the UI (75%)
  - **Architecture:** Clear and competent description of the system's design (12.5%)
  - **Process:** Clear and critical retrospective on the development process, discussion of lessons learned (12.5%)

- **Grading policy:**
  - Visible part of presentation → Assignment 5 grade for whole cluster/team
  - Audible part of presentation
    - Given by one person → that person's Presentation Grade
    - Given by several people → contributes to Assignment 5 cluster/team grade

Cluster Assignment #5: Presentation Schedule

- Clusters present their products in the Zoom classroom at https://eu01web.zoom.us/j/62847273071 on **Wed 20 Apr**
  - 15:00-15:20 Cluster 1
  - 15:25-15:45 Cluster 2
  - 15:50-16:10 Cluster 3
  - 16:15-16:35 Cluster 4
  - 16:40-17:00 Cluster 5
  - 17:05-17:25 Cluster 6
  - 17:30-17:50 Cluster 7
  - 17:55-18:15 Cluster 8

- All teams are encouraged to attend presentations of other clusters as well
  - to learn from other teams' experiences
  - to give other teams an audience

Cluster Assignment #5: Presentation Protocol

- Have your camera and microphone off by default
- Select “Hide non-video participants” in Zoom to see only the team “on stage”
- Turn your camera on (if available) during your own cluster’s/team’s presentation
- Speaker turns microphone on and shares screen to show your demo and slides
- Turn your microphone on during your team’s Q&A

- The final presentations will **not** be recorded.

Remaining Class Schedule

- **Wed 30 Mar** Assignment #4 (Software Test) presentations
- **Mon 4 Apr** Final lecture with information on final exam
- **Wed 6 Apr** Assignment #5 (Final Product) draft consultations
- **Mon 11 Apr** Spare lecture slot – anything you'd like to recap?
- **Wed 13 Apr** Easter break – no consultations
- **Mon 18 Apr** Easter break – no lecture
- **Wed 20 Apr** Assignment #5 (Final Product) presentations
- **Sun 24 Apr** Assignment #5 (Final Product) and peer assessment submission
- **Tue 26 Apr** Final exam

Thank you!

book@hi.is