SOFTWARE ENGINEERING

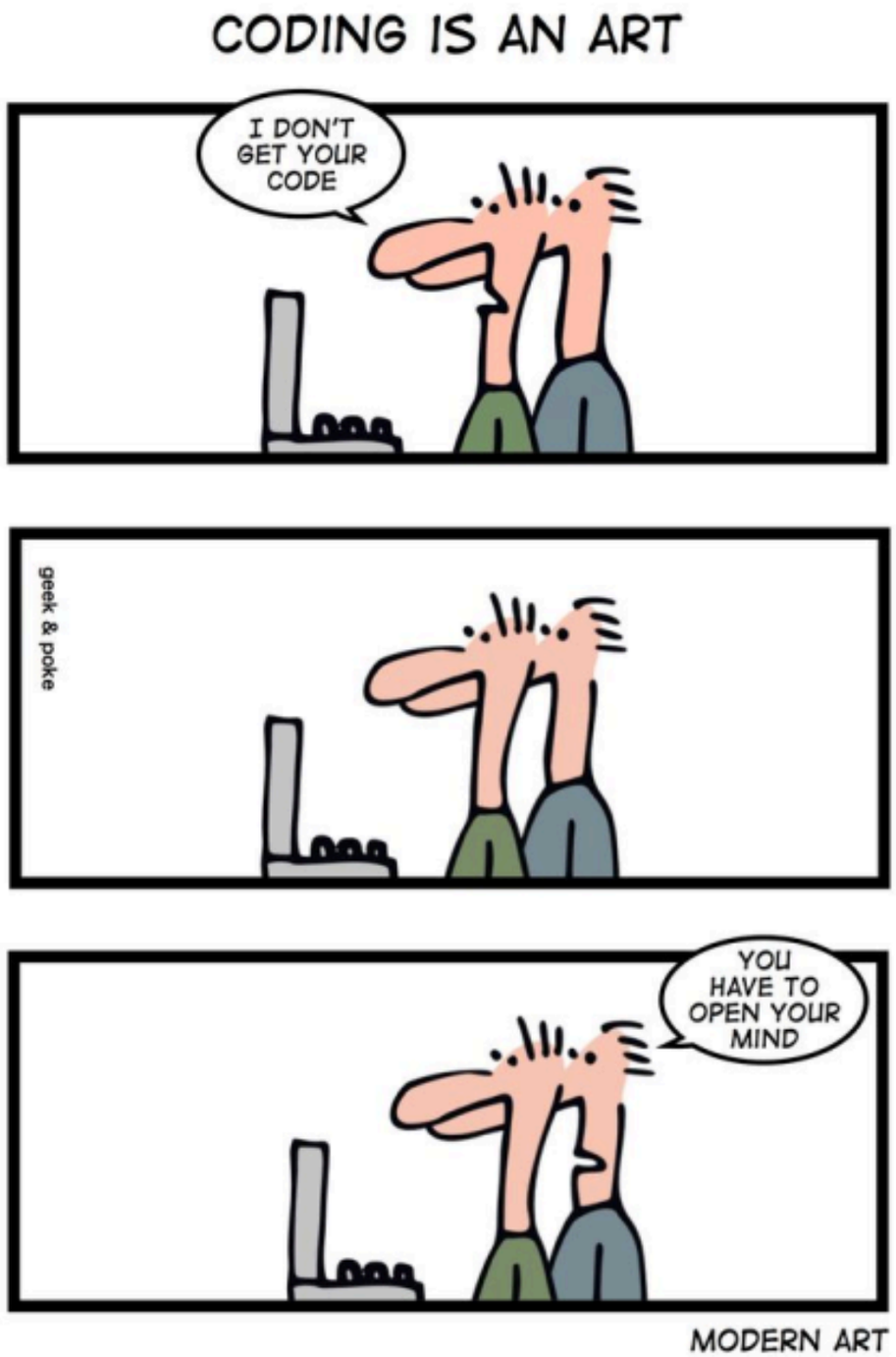
Class 11

amrut@iiit.ac.in

CODE READING

CODE READING

Code reading skills are important



CODE READING

Code reading skills are important

- Code is read more often than it's written.
- Code reviews require code reading skills.
- Debugging requires code reading skills.
- New feature additions require code reading skills.

CODE READING

Questions to ask when confronted with new class

1. What is the big picture context for the class? What are the responsibilities?
2. What is type of this class?
3. Does it have any state? Is it mutable?
4. What are class dependencies? Is it using any structural design patterns?
5. How is class instantiated? Is it using any creational design patterns?
6. What public interface does it expose? Do methods use any behavioral design patterns?
7. Is the code “clean”?
8. Is it following design principles correctly?
9. How can I make it better?

CODE READING

https://github.com/google/guava/blob/master/guava/src/com/google/common/base/Optional.java

or

https://tinyurl.com/vm5tlcm

CODE READING

Question 1:

What is the big picture context for the class? What are the responsibilities?

CODE READING

Question 1:

What is the big picture context for the class? What are the responsibilities?

- Read JavaDoc if available
- Google if this is public library code
- Look at the tests for use-cases
- Talk to people

CODE READING

Question 1:

What is the big picture context for the class? What are the responsibilities?

Problem:

- Most of languages have concept of “null”
- Null is used in multiple situations.
- Nulls make code difficult to read.
- Nulls do not fail fast.

CODE READING

Question 1:

What is the big picture context for the class? What are the responsibilities?

Solution:

Create Optional data type that allows two states, empty and non-empty which gives methods ability to clearly mention that they can return the request object or empty.

CODE READING

Question 1:

What is the big picture context for the class? What are the responsibilities?

```
private void func1() {
    Integer i = func2(2);
    System.out.println(i < 0);
}
private Integer func2(Integer i) {
    return func3(i);
}
private Integer func3(Integer i) {
    return i == 0 ? i : null;
}
```

CODE READING

Question 1:

What is the big picture context for the class? What are the responsibilities?

```
private void func1() {
    Integer i = func2(2);
    System.out.println(i < 0);
}
private Integer func2(Integer i) {
    return func3(i);
}
private Integer func3(Integer i) {
    return i == 0 ? i : null;
}
```

NullPointerException in func1

CODE READING

Question 1:

What is the big picture context for the class? What are the responsibilities?

```
private void func1() {
    Integer i = func2(2);
    System.out.println(i < 0);
}
private Integer func2(Integer i) {
    return func3(i).get();
}
private Optional<Integer> func3(Integer i) {
    return i == 0 ? Optional.of(i) : Optional.absent();
}
```

CODE READING

Question 1:

What is the big picture context for the class? What are the responsibilities?

```
private void func1() {
    Integer i = func2(2);
    System.out.println("");
}
private Integer func2(Integer i) {
    return func3(i).get();
}
private Optional<Integer> func3(Integer i) {
    return i == 0 ? Optional.of(i) : Optional.empty();
}
```

IllegalStateException in func2

"Optional.get() cannot be called on an absent value"

CODE READING

Question 2: What is type of this class?

CODE READING

Question 2: What is type of this class?

```
Line 85: public abstract class Optional<T> …
… •
```

CODE READING

Question 2: What is type of this class?

```
Line 85: public abstract class Optional<T> …
…
```

.. means this cannot be instantiated with “new”

.. means there must be some other classes that extend this

.. all these subclasses can share some implementation

CODE READING

Question 3: Does it have any state? Is it mutable?

CODE READING

Question 3: Does it have any state? Is it mutable?

- Find fields
- Find setters/getters and other methods that change these fields.

CODE READING

Question 3: Does it have any state? Is it mutable?

No

CODE READING

Question 3: Does it have any state? Is it mutable?

No

.. means that this class is thread-safe.

.. means that the class state is always consistent.

.. means that objects can be safely passed.

CODE READING

Question 4: What are class dependencies? Is it using any structural design patterns?

CODE READING

Question 4: What are class dependencies? Is it using any structural design patterns?

- Inspect imports
- Search for “new"
- Inspect constructors

CODE READING

Question 4: What are class dependencies? Is it using any structural design patterns?

```
Line 22: import java.io.Serializable;
```

CODE READING

Question 4: What are class dependencies? Is it using any structural design patterns?

```
Line 105: return new Present<T>(checkNotNull(reference));
Line 116: return (nullableReference == null)
              ? Optional.<T>absent()
              : new Present<T>(nullableReference);
```

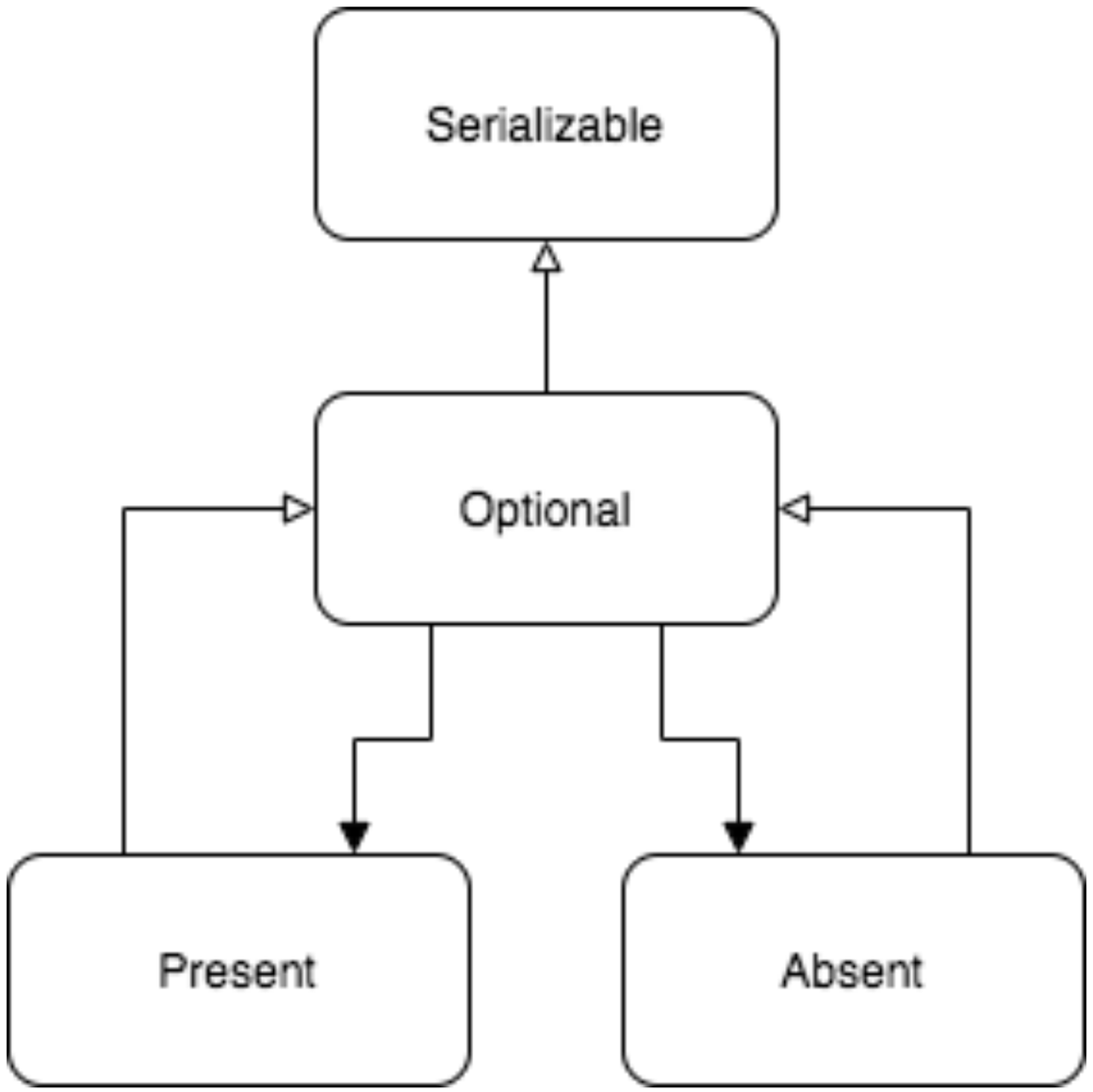
CODE READING

Question 4: What are class dependencies? Is it using any structural design patterns?

```
Line 93: return Absent.withType();
```

CODE READING

Question 4: What are class dependencies? Is it using any structural design patterns?



CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

- Look for constructors
- Look for static methods

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 161:  Optional() {}
```

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 161:  Optional() {}
```

.. means default public constructor is suppressed.

.. means this can be subclassed only from this package.

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 95: public static <T> Optional<T> absent()…
```

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 95: public static <T> Optional<T> absent()…
```

.. uses Static Factory Method Pattern, not to be confused with factory method pattern.

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 104: public static <T> Optional<T> of(T reference) …
```

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 115: public static <T> Optional<T>
                fromNullable(@Nullable T nullableReference) …
```

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 34: private Absent() {}
```

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 34: private Absent() {}
```

.. mean this cannot be instantiated with “new"

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 27: static final Absent<Object> INSTANCE = new Absent<>();
```

```
Lines 30-32:
  static <T> Optional<T> withType() {
    return (Optional<T>) INSTANCE;
 }
```

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 27: static final Absent<Object> INSTANCE = new Absent<>();
```

```
Lines 30-32:
  static <T> Optional<T> withType() {
    return (Optional<T>) INSTANCE;
 }
```

.. uses Singleton Pattern.

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 29:  Present(T reference) {
```

CODE READING

Question 5: How is class instantiated? Is it using any creational design patterns?

```
Line 29:  Present(T reference) {
```

.. means can be instantiated with “new” but only from the same package.

CODE READING

Question 6: What public interface does it expose? Do methods use any behavioral design patterns?

CODE READING

Question 6: What public interface does it expose? Do methods use any behavioral design patterns?

```
Line 168: public abstract boolean isPresent();
```

```
Line 182: public abstract T get();
```

CODE READING

Question 6: What public interface does it expose? Do methods use any behavioral design patterns?

```
Lines 36-44:
public boolean isPresent() {
  return false;
}
public T get() {
  throw new IllegalStateException
    (“Optional.get() cannot be called on an absent value");
}
```

CODE READING

Question 6: What public interface does it expose? Do methods use any behavioral design patterns?

`Lines: 33-41`

```
public boolean isPresent() {
  return true;
}
public T get() {
  return reference;
}
```

CODE READING

Question 6: What public interface does it expose? Do methods use any behavioral design patterns?

.. uses State Pattern.

CODE READING

Question 7: Is the code “clean”?

CODE READING

Question 7: Is the code “clean”?

- Inspect class, method, variable names
- Look for bad comments
- Look for long methods
- Look for method with too many params
- Look for flag args
- Look for DRY violations
- …

CODE READING

Question 8: Is it following design principles correctly?

CODE READING

Question 8: Is it following design principles correctly?

- SRP
- Open/Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

CODE READING

Question 9: How can I make it better?

CODE READING

Question 9: How can I make it better?

- Can I make code more readable?
- Can I make the design better?
- Can I reduce complexity?
- …

CODE READING

Question 9: How can I make it better?

Boy Scout Rule: "Always leave the campground cleaner than you found it."

REFERENCES

- Null References: The Billion Dollar Mistake