Best practices

CSE P 504

How to avoid bugs in your code?

How to avoid bugs in your code?

It's super simple...don't introduce them during coding!

"Everybody makes mistakes except for me…

But then, there is just one of me."

How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
2. Correctness: get it right the first time
3. Bug visibility

How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
    a. Programming language
        i. Ever had a use-after free bug in a garbage-collected language?
        ii. Ever had an assignment bug (String to Integer) in a statically typed language? (Even stronger guarantees with custom types and pluggable type systems.)
    b. Libraries and protocols
        i. TCP vs. UDP
        ii. No overflows in BigInteger

How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
   a. Programming language
   b. Libraries and protocols
2. Correctness: get it right the first time
   a. A program without a spec is bug free
   b. Keep it simple, modular, and testable
   c. Defensive programming and conventions (discipline)

How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
   a. Programming language
   b. Libraries and protocols
2. Correctness: get it right the first time
   a. A program without a spec is bug free
   b. Keep it simple, modular, and testable
   c. Defensive programming and conventions (discipline)
3. Bug visibility
   a. Assertions (pre/post conditions)
   b. (Regression) testing
   c. Fail fast

QUIZ
NIGHT

Quiz: setup and goals

- 3-4 students per team
- 4 code snippets
- 2 rounds
  - **First round**
    - For each code snippet, decide whether it represents good or bad practice.
    - **Goal:** discuss and reach consensus on good or bad practice.
  - **Second round** (known “solutions”)
    - For each code snippet, try to understand why it is good or bad practice.
    - **Goal:** come up with an explanation or a counter argument.

Round 1: good or bad?

Snippet 1: good or bad?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

Snippet 2: good or bad?

```
public enum PaymentType {DEBIT, CREDIT}

public void doTransaction(double amount, PaymentType payType) {
  switch (payType) {
    case DEBIT:
       … // process debit card
       break;
    case CREDIT:
       … // process credit card
       break;
    default:
       throw new IllegalArgumentException("Unexpected payment type");
   }
}
```

Snippet 3: good or bad?

```
public class ArrayList<E> {
    public E remove(int index) {
        …
    }
    public boolean remove(Object o) {
        …
    }
    …
}
```

Snippet 4: good or bad?

```
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
}
```

Round 2: why is it good or bad?

My take on this

- Snippet 1: bad

- Snippet 2: good

- Snippet 3: bad

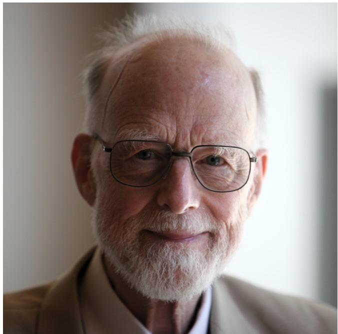
- Snippet 4: good

Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```



Null references...the billion dollar mistake.

Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

```
File[] files = getAllLogs();
for (File f : files) {
    …
}
```

Don't return null; return an empty array instead.

Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

No diagnostic information.

Snippet 2: this is good, but why?

```
public enum PaymentType {DEBIT, CREDIT}

public void doTransaction(double amount, PaymentType payType) {
  switch (payType) {
    case DEBIT:
       … // process debit card
       break;
    case CREDIT:
       … // process credit card
       break;
    default:
       throw new IllegalArgumentException("Unexpected payment type");
   }
}
```

Snippet 2: this is good, but why?

```
public enum PaymentType {DEBIT, CREDIT}
public void doTransaction(double amount, PaymentType payType) {
  switch (payType) {
    case DEBIT:
      … // process debit card
      break;
    case CREDIT:
      … // process credit card
      break;
    default:
      throw new IllegalArgumentException("Unexpected payment type");
   }
}
```

Type safety using an enum; throws an exception for unexpected cases (e.g., future extensions of PaymentType).

Snippet 3: Java API, but still bad! why?

```
public class ArrayList<E> {
    public E remove(int index) {
        …
    }
    public boolean remove(Object o) {
        …
    }
    …
}
```

Snippet 3: Java API, but still bad! why?

```
public class ArrayList<E> {
    public E remove(int index) {
        …
    }
    public boolean remove(Object o) {
        …
    }
    …
}
```

```
ArrayList<String> l = new ArrayList<>();
Integer index = Integer.valueOf(1);
l.add("Hello");
l.add("World");
l.remove(index);
```

What does the last call return?

Snippet 3: Java API, but still bad! why?

```
public class ArrayList<E> {
    public E remove(int index) {
        …
    }
    public boolean remove(Object o) {
        …
    }
    …
}
```

```
ArrayList<String> l = new ArrayList<>();
Integer index = Integer.valueOf(1);
l.add("Hello");
l.add("World");
l.remove(index);
```

Avoid method overloading, which is statically resolved. Autoboxing/unboxing adds additional confusion.

Snippet 4: this is good, but why?

```
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
}
```

Snippet 4: this is good, but why?

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
}
```

Good encapsulation; immutable object.