Principles of Software Construction: Objects, Design, and Concurrency (Part 2: Designing (Sub-)Systems)

Design for Robustness

Christian Kästner  Bogdan Vasilescu
Administrativa

• Midterm 1: Thursday here
• Practice midterm on Piazza
• Review session tomorrow, 6:30pm GHC4401

• HW 2 grades
• HW 4 out, Milestone A due Feb 23
  – Do not underestimate design
Problem Space
Domain Model

inspires objects and names

PineTree
age
tsize
harvest()

Solution Space
Object Model

RangerAgent
sanitation(Forest)
salvage(Forest)
Design principle for reuse: low coupling
Design principle for reuse: *low coupling*

• Each component should depend on as few other components as possible

• Benefits of low coupling:
  – Enhances understandability
  – Reduces cost of change
  – Eases reuse
Design principle for reuse: *high cohesion*
Design principle for reuse: *high cohesion*

- Each component should have a small set of closely-related responsibilities

- Benefits:
  - Facilitates understandability
  - Facilitates reuse
  - Eases maintenance
Information Expert (GRASP Pattern/Design Heuristic)

• **Heuristic:** *Assign a responsibility to the class that has the information necessary to fulfill the responsibility*

• Start assigning responsibilities by clearly stating responsibilities!

• Typically follows common intuition

• Software classes instead of Domain Model classes
  – If software classes do not yet exist, look in Domain Model for fitting abstractions (-> correspondence)
Creator
(Grasp Pattern/Design Heuristic)

• Problem: Who creates an A?
• Solution: Assign class responsibility of creating instance of class A to B if
  – B aggregates A objects
  – B contains A objects
  – B records instances of A objects
  – B closely uses A objects
  – B has the initializing data for creating A objects
• the more the better; where there is a choice, prefer
  – B aggregates or contains A objects
• Key idea: Creator needs to keep reference anyway and will frequently use the created object
Learning Goals

• Use exceptions to write robust programs
• Make error handling explicit in interfaces and contracts
• Isolate errors modularly
• Test complex interactions locally
• Test for error conditions
Design Goals, Principles, and Patterns

• Design Goals
  – Design for robustness

• Design Principle
  – Modular protection
  – Explicit interfaces

• Supporting Language Features
  – Exceptions
EXCEPTION HANDLING
What does this code do?

```java
FileInputStream fIn = new FileInputStream(filename);
if (fIn == null) {
    switch (errno) {
        case _ENOFILE:
            System.err.println("File not found: " + ...);
            return -1;
        default:
            System.err.println("Something else bad happened: " + ...);
            return -1;
    }
}
DataInput dataInput = new DataInputStream(fIn);
if (dataInput == null) {
    System.err.println("Unknown internal error.");
    return -1;  // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
    System.err.println("Error reading binary data from file");
    return -1;
}  // The slide lacks space to close the file. Oh well.
return i;
```
Compare to:

```java
try {
    FileInputStream fileInput = new FileInputStream(filename);
    DataInput dataInput = new DataInputStream(fileInput);
    int i = dataInput.readInt();
    fileInput.close();
    return i;
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + filename);
    return -1;
} catch (IOException e) {
    System.out.println("Error reading binary data from file " + filename);
    return -1;
}
```
Exceptions

- Exceptions notify the caller of an exceptional circumstance (usually operation failure)

Semantics
- An exception propagates up the function-call stack until main() is reached (terminates program) or until the exception is caught

Sources of exceptions:
- Programmatically throwing an exception
- Exceptions thrown by the Java Virtual Machine
public static void test() {
    try {
        System.out.println("Top");
        int[] a = new int[10];
        a[42] = 42;
        System.out.println("Bottom");
    } catch (NegativeArraySizeException e) {
        System.out.println("Caught negative array size");
    }
}

public static void main(String[] args) {
    try {
        test();
    } catch (IndexOutOfBoundsException e) {
        System.out.println("Caught index out of bounds");
    }
}
Java: The **finally** keyword

- The `finally` block always runs after `try/catch`:

```java
try {
    System.out.println("Top");
    int[] a = new int[10];
    a[2] = 2;
    System.out.println("Bottom");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Caught index out of bounds");
} finally {
    System.out.println("Finally got here");
}
```
The exception hierarchy in Java

Object
  ├── Throwable
  │   ├── Exception
  │   │   ├── RuntimeException
  │   │   ├── IOException
  │   │   ├── EOFException
  │   │   └── FileNotFoundException
  │   ├── NullPointerException
  │   ├── IndexOutOfBoundsException
  │   └── ... (other exceptions)

...
Design choice: Checked and unchecked exceptions and return values

• *Unchecked exception*: any subclass of RuntimeException
  – Indicates an error which is highly unlikely and/or typically unrecoverable

• *Checked exception*: any subclass of Exception that is not a subclass of RuntimeException
  – Indicates an error that every caller should be aware of and explicitly decide to handle or pass on

• Return values (boolean, empty lists, null, etc): If failure is common and expected possibility

Design Principle: Explicit Interfaces (contracts)
Creating and throwing your own exceptions

• Methods must declare any checked exceptions they might throw
• If your class extends java.lang.Throwable you can throw it:
  – if (someErrorBlahBlahBlahBlah) {
    – throw new MyCustomException(“Blah blah blah”);
  – }
Benefits of exceptions

• Provide high-level summary of error and stack trace
  – Compare: core dumped in C
• Can’t forget to handle common failure modes
  – Compare: using a flag or special return value
• Can optionally recover from failure
  – Compare: calling System.exit()
• Improve code structure
  – Separate routine operations from error-handling (see Cohesion)
• Allow consistent clean-up in both normal and exceptional operation
Guidelines for using exceptions

• Catch and handle all checked exceptions
  – Unless there is no good way to do so...
• Use runtime exceptions for programming errors

• Other good practices
  – Do not catch an exception without (at least somewhat) handling the error
  – When you throw an exception, describe the error
  – If you re-throw an exception, always include the original exception as the cause
Testing for presence of an exception

```java
import org.junit.*;
import static org.junit.Assert.fail;

public class Tests {

    @Test
    public void testSanityTest(){
        try {
            openNonexistingFile();
            fail("Expected exception");
        } catch(IOException e) { }
    }

    @Test(expected = IOException.class)
    public void testSanityTestAlternative() {
        openNonexistingFile();
    }
}
```
DESIGN PRINCIPLE: MODULAR PROTECTION
Modular Protection

• Errors and bugs unavoidable, but exceptions should not leak across modules (methods, classes), if possible
• Good modules handle exceptional conditions locally
  – Local input validation and local exception handling where possible
  – Explicit interfaces with clear pre/post conditions
  – Explicitly documented and checked exceptions where exceptional conditions may propagate between modules
  – Information hiding/encapsulation of critical code (likely bugs, likely exceptions)
Examples

• Printer crash should not corrupt entire system
  – E.g., printer problem handled locally, logged, user informed

• Exception/infinite loop in Pine Simulation should not freeze GUI
  – E.g., decouple simulation from UI

• Error in shortest-path algorithm should not corrupt graph
  – E.g., computation on immutable data structure
TESTING WITH COMPLEX ENVIRONMENTS
Problems when testing (sub-)systems

• User-facing applications
  – Users click, drag, etc., and interpret output
  – Timing issues

• Testing against big infrastructure
  – Databases, web services, etc.

• Real world effects
  – Printing, mailing documents, etc.

• Collectively comprise *the test environment*
Example – Tiramisu app

- Mobile route planning app
- Android UI
- Back end uses live PAT data
Another example

- 3rd party Facebook apps
- Android user interface
- Backend uses Facebook data
Testing in real environments

void buttonClicked() {
    render(getFriends());
}

List<Friend> getFriends() {
    Connection c = http.getConnection();
    FacebookAPI api = new FacebookAPI(c);
    List<Node> persons = api.getFriends("john");
    for (Node person1 : persons) {
        for (Node person2 : persons) {
            ...
        }
    }
    return result;
}
Eliminating Android dependency

```java
@Test void testGetFriends() {
    assert getFriends() == ...;
}

List<Friend> getFriends() {
    Connection c = http.getConnection();
    FacebookAPI api = new FacebookAPI(c);
    List<Node> persons = api.getFriends("john");
    for (Node person1 : persons) {
        for (Node person2 : persons) {
            ...
        }
    }
    return result;
}
```
That won’t quite work

- GUI applications process *thousands* of events
- Solution: automated GUI testing frameworks
  - Allow streams of GUI events to be captured, replayed
- These tools are sometimes called *robots*
Eliminating Facebook dependency

```
@Test void testGetFriends() {
    assert getFriends() == ...;
}

List<Friend> getFriends() {
    Connection c = http.getConnection();
    FacebookAPI api = new MockFacebook(c);
    List<Node> persons = api.getFriends("john");
    for (Node person1 : persons) {
        for (Node person2 : persons) {
            ...
        }
    }
    return result;
}
```

class MockFacebook implements FacebookInterface {
    void connect() {}
    List<Node> getFriends(String name) {
        if ("john".equals(name)) {
            List<Node> result=new List();
            result.add(...);
            return result;
        }
    }
}
That won’t quite work!

• Changing production code for testing unacceptable
• Problem caused by constructor in code
• Use tools to facilitate this sort of testing
  – Dependency injection tools, e.g., Dagger, Guice
  – Mock object frameworks such as Mockito
Fault injection

- Mocks can emulate failures such as timeouts
- Allows you to verify the robustness of system
Advantages of using mocks

- Test code locally without large environment
- Enable deterministic tests
- Enable fault injection
- Can speed up test execution
  - e.g., avoid slow database access
- Can simulate functionality not yet implemented
- Enable test automation
Design Implications

- Think about testability when writing code
- When a mock may be appropriate, design for it
- Hide subsystems behind an interface
- Use factories, not constructors to instantiate
- Use appropriate tools
  - Dependency injection or mocking frameworks
More Testing in 15-313

Foundations of Software Engineering

• Manual testing
• Security testing, penetration testing
• Fuzz testing for reliability
• Usability testing
• GUI/Web testing
• Regression testing
• Differential testing
• Stress/soak testing
DESIGN PATTERN: PROXY
Proxy Design Pattern

Applicability
• Whenever you need a more sophisticated obj reference than a simple pointer
• Local representative for remote obj.
• Create/load expensive obj on demand
• Control access to an object
• Extra error handling, failover
• Caching
• Reference count an object

Consequences
• Introduces a level of indirection
• Hides distribution from client
• Hides optimizations from client
• Adds housekeeping tasks
Example: Caching

interface FacebookAPI {
    List<Node> getFriends(String name);
}
class FacebookProxy implements FacebookAPI {
    FacebookAPI api;
    HashMap<String,List<Node>> cache = new HashMap... 
    FacebookProxy(FacebookAPI api) { this.api=api;}

    List<Node> getFriends(String name) {
        result = cache.get(name);
        if (result == null) {
            result = api.getFriends(name);
            cache.put(name, result);
        }
        return result;
    }
}
Example: Caching and Failover

```java
interface FacebookAPI {
    List<Node> getFriends(String name);
}
class FacebookProxy implements FacebookAPI {
    FacebookAPI api;
    HashMap<String, List<Node>> cache = new HashMap...
    FacebookProxy(FacebookAPI api) { this.api=api; }

    List<Node> getFriends(String name) {
        try {
            result = api.getFriends(name);
            cache.put(name, result);
            return result;
        } catch (ConnectionException c) {
            return cache.get(name);
        }
    }
}
```
Example: Redirect to Local Service

```java
interface FacebookAPI {
    List<Node> getFriends(String name);
}

class FacebookProxy implements FacebookAPI {
    FacebookAPI api;
    FacebookAPI fallbackApi;
    FacebookProxy(FacebookAPI api, FacebookAPI f) {
        this.api = api; fallbackApi = f;
    }

    List<Node> getFriends(String name) {
        try {
            return api.getFriends(name);
        } catch (ConnectionException c) {
            return fallbackApi.getFriends(name);
        }
    }
}

Further alternatives: other error handling, redirect to other/local service, default values, etc
```
Summary

• Design for Robustness as Design Goal
• Explicit Interfaces as Design Principle
  – Error handling explicit in interfaces (declared exceptions, return types)
  – Exceptions in Java as supporting language mechanism
• Modular Protection as Design Principle
  – Handle Exceptions Locally
• Local testing with stubs and drivers
• Proxy design pattern for separate error handling
ASSERTIONS
What is an assertion?

• Statement containing boolean expression that programmer believes to be true:
  
  ```
  assert speed <= SPEED_OF_LIGHT;
  ```

• Evaluated at run time – throws Error if false

• Disabled by default - no performance effect

• Typically enabled during development

• Can enable in the field when problems occur!
Syntax

AssertStatement:

assert Expression$_1$ ;
assert(Expression$_1$, Expression$_2$) ;

• Expression$_1$ - asserted condition (boolean)
• Expression$_2$ - detail message of AssertionError
Why use assertions?

• Document & test programmer's assumptions
  – e.g., class invariants
• Verify programmer’s’s understanding
• Quickly uncover bugs
• Increase confidence that program is bug-free
Look for “assertive comments”

```java
int remainder = i % 3;
if (remainder == 0) {
    ...
} else if (remainder == 1) {
    ...
} else { // (remainder == 2)
    ...
}
```
int remainder = i % 3;
if (remainder == 0) {
    ...
}
else if (remainder == 1) {
    ...
}
else {
    assert remainder == 2;
    ...
}
Use second argument for *failure capture*

```java
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert (i % 3 == 2, i);
    ...
}
```
Look for switch with no default

```
switch(flavor) {
    case VANILLA:
        ...
        break;
    case CHOCOLATE:
        ...
        break;
    case STRAWBERRY:
        ...
}
```
Add an “assertive default”

```
switch(flavor) {
    case VANILLA:
        ...
        break;
    case CHOCOLATE:
        ...
        break;
    case STRAWBERRY:
        ...
        break;
    default:
        assert (false, flavor);
}
```
Do not use assertions for *public* preconditions

/**
 * Sets the refresh rate.
 *  
 * @param rate refresh rate, in frames per second.
 * @throws IllegalArgumentException if rate <= 0
 * or rate > MAX_REFRESH_RATE.
 */

public void setRefreshRate(int rate) {
    if (rate <= 0 || rate > MAX_REFRESH_RATE)
        throw new IllegalArgumentException(...);
    setRefreshInterval(1000 / rate);
}

Do use assertions for non-public preconditions

/**
 * Sets the refresh interval (which must correspond to a legal frame rate).
 *
 * @param interval refresh interval in ms
 */

private void setRefreshInterval(int interval) {
    assert interval > 0 && interval <= 1000, interval;
    ... // Set the refresh interval
}
/**
 * Returns BigInteger whose value is \( (\text{this}^{-1} \mod m) \).
 * @throws ArithmeticException if \( m \leq 0 \), or this
 *     BigInteger is not relatively prime to \( m \).
 */
public BigInteger modInverse(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException(m + " \leq 0");
    ... // Do the computation
    assert this.multiply(result).mod(m).equals(ONE);
    return result;
}
Complex postconditions

```c
void foo(int[] a) {
    // Manipulate contents of array
    ...
    // Array will appear unchanged
}
```
void foo(final int[] a) {
    class DataCopy {
        private int[] aCopy;
        DataCopy() { aCopy = (int[]) a.clone(); }
        boolean isConsistent() {
            return Arrays.equals(a, aCopy);
        }
    }
    DataCopy copy = null;
    assert (copy = new DataCopy()) != null;
    ... // Manipulate contents of array
    assert copy.isConsistent();
}
Caveat – asserts must not have *side effects* visible outside other asserts

Do this:

```java
boolean modified = set.remove(elt);
assert modified;
```

Not this:

```java
assert set.remove(elt);  // Bug!
```
Sermon: accept assertions into your life

• Programmer’s interior monologue:
  – “Now at this point, we know...”
• During, not after, development
• Quickly becomes second nature
• Pays big code-quality dividends
IN-CLASS EXERCISE
// The actual checkin is not shown so you don't need to model that part of the interaction.
return item.getFine(today);

return rate * (checkinDate - dueDate);

return base + rate * (checkinDate - dueDate);
UML interaction diagrams

1. Using the UML class diagram for reference, sketch a **UML sequence diagram** for:

   ```java
   public class Client {
       public static void main(String[] args) {
           Library library = new Library();
           Item book = new Book("Alice...", "Lives...");
           library.checkin(book);
       }
   }
   ```

2. Name any design patterns in the UML class diagram.
Sample solution
Sample solution, version 2