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

Design for Robustness

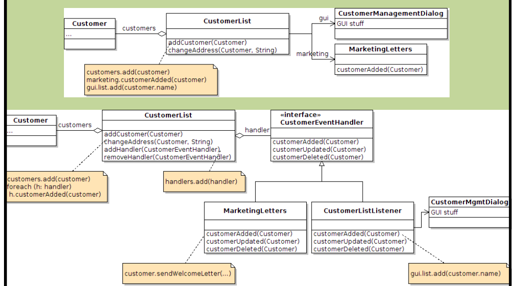
Christian Kästner Charlie Garrod

Administrativa

- Midterm 1: Thursday here
- Practice midterm on Piazza
- Review session tomorrow, 7pm HBH 100

- HW 2 grades
- HW 4 out, Milestone A due Feb 19
  - Do not underestimate design

Which design is better? Argue with design goals, principles, heuristics, and patterns that you know



 * old midterm question

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

Additional Readings

- Textbook Chapter 36 (handling failure, exceptions, proxy)

EXCEPTION HANDLING

What does this code do?

```
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:

```
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

Exceptional control-flow in Java

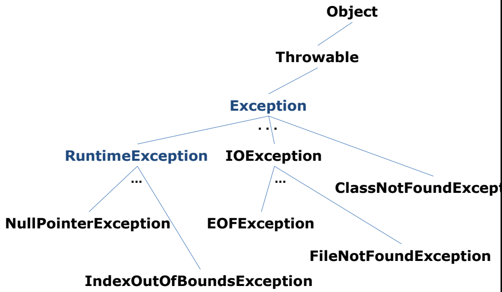
```
public static void test() {
    try {
        System.out.println("Top");
        int[] a = new int[10];
        a[42] = 42;
        System.out.println("Bottom");
    } catch (NegativeArraySizeException e) {
        System.out.println("Caught negative array size");
    }
}

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:

```
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



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 (someErrorBlahBlahBlah) {
  - throw new MyCustomException(“Blah blah blah”);
  - }

Benefits of exceptions

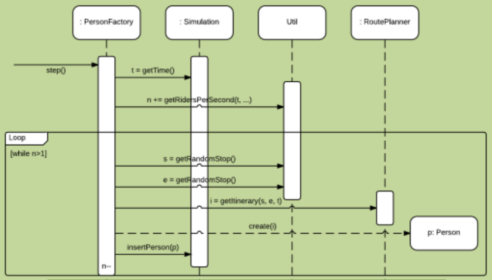
- Provide high-level summary of error and stack trace
  - Compare: core dumped in C
- Can’t forget to handle common failure modes
  - Compare: using a flag or special return value
- Can optionally recover from failure
  - Compare: calling System.exit()
- Improve code structure
  - Separate routine operations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

Guidelines for using exceptions

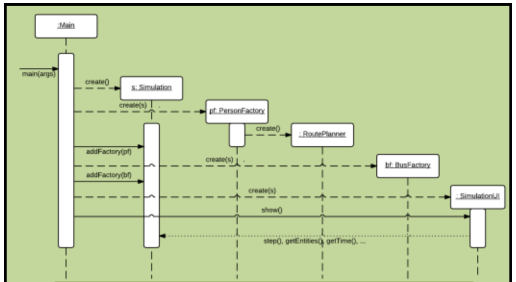
- Catch and handle all checked exceptions
  - Unless there is no good way to do so…
- Use runtime exceptions for programming errors
- Other good practices
  - Do not catch an exception without (at least somewhat) handling the error
  - When you throw an exception, describe the error
  - If you re-throw an exception, always include the original exception as the cause

Testing for presence of an exception

```
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();
        }
}
```

How to test failure to compute a route?



How to test behavior for missing routes file?

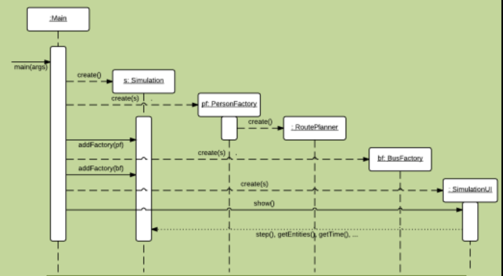
DESIGN PRINCIPLE: MODULAR PROTECTION

Modular Protection

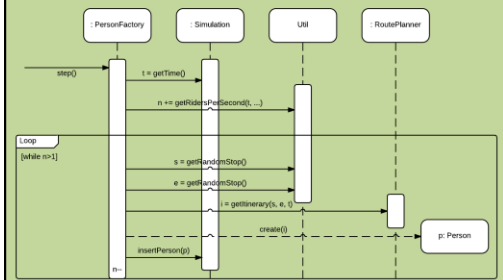
- Errors and bugs unavoidable, but exceptions should not leak across modules (methods, classes), if possible
- Good modules handle exceptional conditions locally
  - Local input validation and local exception handling where possible
  - Explicit interfaces with clear pre/post conditions
  - Explicitly 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



What happens with incorrect entries in routes file?
What happens if the routes file missing?
What happens when getEntities returns null?



Where and how to handle invalid routes?

TESTING WITH COMPLEX ENVIRONMENTS

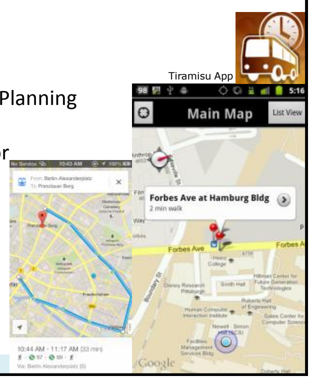
Problems when testing (sub-)systems

- User interfaces and user interactions
  - Users click buttons, interpret output
  - Waiting/timing issues
- Test data vs. real data
- Testing against big infrastructure (databases, web services, …)
- Testing with side effects (e.g., printing and mailing documents)
- Nondeterministic behavior
- Concurrency (more later in the semester)

-> the test environment

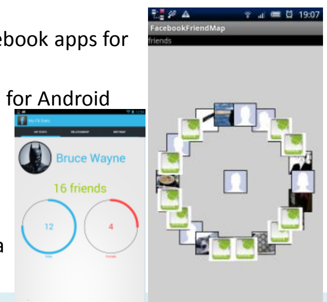
Example

- 3rd party Route Planning app for Android
- User interface for Android
- Internal computations ala HW2
- Backend with PAT live data



Example

- 3rd party Facebook apps for Android
- User interface for Android
- Internal computations ala HW1
- Backend with Facebook data



Testing in real environments

Facebook — Code — Android client

```
void buttonClicked() {
        render(getFriends());
}
Pair[] getFriends() {
        Connection c = http.getConnection();
        FacebookAPI api = new FacebookAPI(c);
        try {
                List<Node> persons = api.getFriends("john");
                for (Node personA: persons) {
                        for (Node personB: persons) {
                                ...
                }}
        } catch (...) { ... }
        return result;
}
```


Test drivers

Facebook — Code — Test driver (JUnit) / Android client

```
@Test void testGetFriends() {
        assert getFriends() == ...;
}
Pair[] getFriends() {
        Connection c = http.getConnection();
        FacebookAPI api = new FacebookAPI(c);
        try {
                List<Node> persons = api.getFriends("john");
                for (Node personA: persons) {
                        for (Node personB: persons) {
                                ...
                }}
        } catch (...) { ... }
        return result;
}
```


Stubs

Stub / Facebook — Facebook Interface — Code — Test driver (JUnit) / Android client

```
FacebookInterface fb;
@Before void init() {fb = new FacebookStub(); }

Pair[] getFriends() {
        try {

        } catch
        return r
}
```

```
class FacebookStub implements FacebookInterface {
        void connect() {}
        List<Node> getPersons(String name) {
                if ("john".equals(n)) {
                        List<Node> result=new List();
                        result.add(...);
                        return result;
                }
        }
}
```


Robustness test

Stub / Facebook / Connection Error — Facebook Interface — Code — Test driver (JUnit) / Android client

```
class ConnectionError implements FacebookInterface {
        List<Node> getPersons(String name) {
                throw new HttpConnectionException();
        }
}

@Test void testConnectionError() {
        assert getFriends(new ConnectionError()) == null;
}
```

Test for expected error conditions by introducing artificial errors through stubs

Testing in real environments

Server Backend (tbd.) — Code — Android client

- Separating code (with stubs) allows to test against functionality
  - provided by other teams
  - specified, but not yet implemented

Testing Strategies in Environments

- Separate business logic and data representation from GUI for testing (more later)
- Test algorithms locally without large environment using stubs
- Advantages of stubs
  - Create deterministic response
  - Can reliably simulate spurious states (e.g. network error)
  - Can speed up test execution (e.g. avoid slow database)
  - Can simulate functionality not yet implemented
- Automate, automate, automate

Design Implications

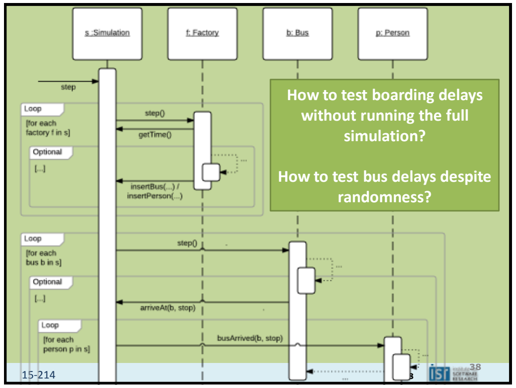
- Write testable code!
- When planning to test with a stub design for it! Abstract the actual subsystem behind and

```
int getFreeTime() {
      MySQLImpl db = new MySQLImpl("calendar.db");
      return db.execute("select …");
}
```

```
int getFreeTime() {
      DatabaseInterface db =
            databaseFactory.createDb("calendar.db");
      return db.execute("select …");
}
```

```
int getFreeTime(MySQLImpl db) {
      return db.execute("select …");
}
```

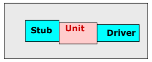
```
int getFreeTime(DatabaseInterface db) {
      return db.execute("select …");
}
```

How to test boarding delays without running the full simulation?

How to test bus delays despite randomness?

Scaffolding



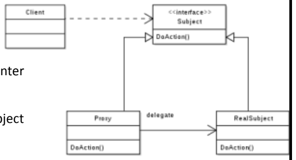
- Catch bugs early: Before client code or services are available
- Limit the scope of debugging: Localize errors
- Improve coverage
  - System-level tests may only cover 70% of code [Massol]
  - Simulate unusual error conditions – test internal robustness
- Validate internal interface/API designs
  - Simulate clients in advance of their development
  - Simulate services in advance of their development
- Capture developer intent (in the absence of specification documentation)
  - A test suite formally captures elements of design intent
  - Developer documentation
- Improve low-level design
  - Early attention to ability to test – "testability"

More Testing in 313

- 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 object reference than a simple pointer
  - Local representative for a remote object
  - Create or load expensive object 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);
		}
	}
}
```


Example: Caching and Failover

```
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

```
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