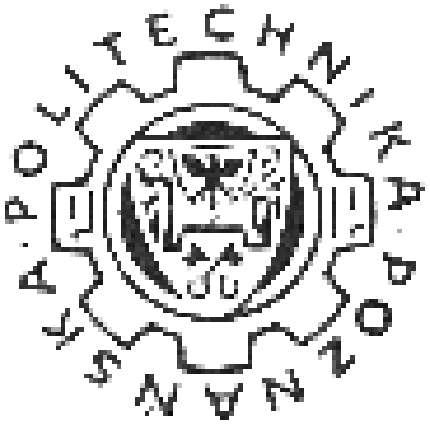


Introduction to Software Testing

Błażej Pietrzak
blazej.pietrzak@cs.put.poznan.pl

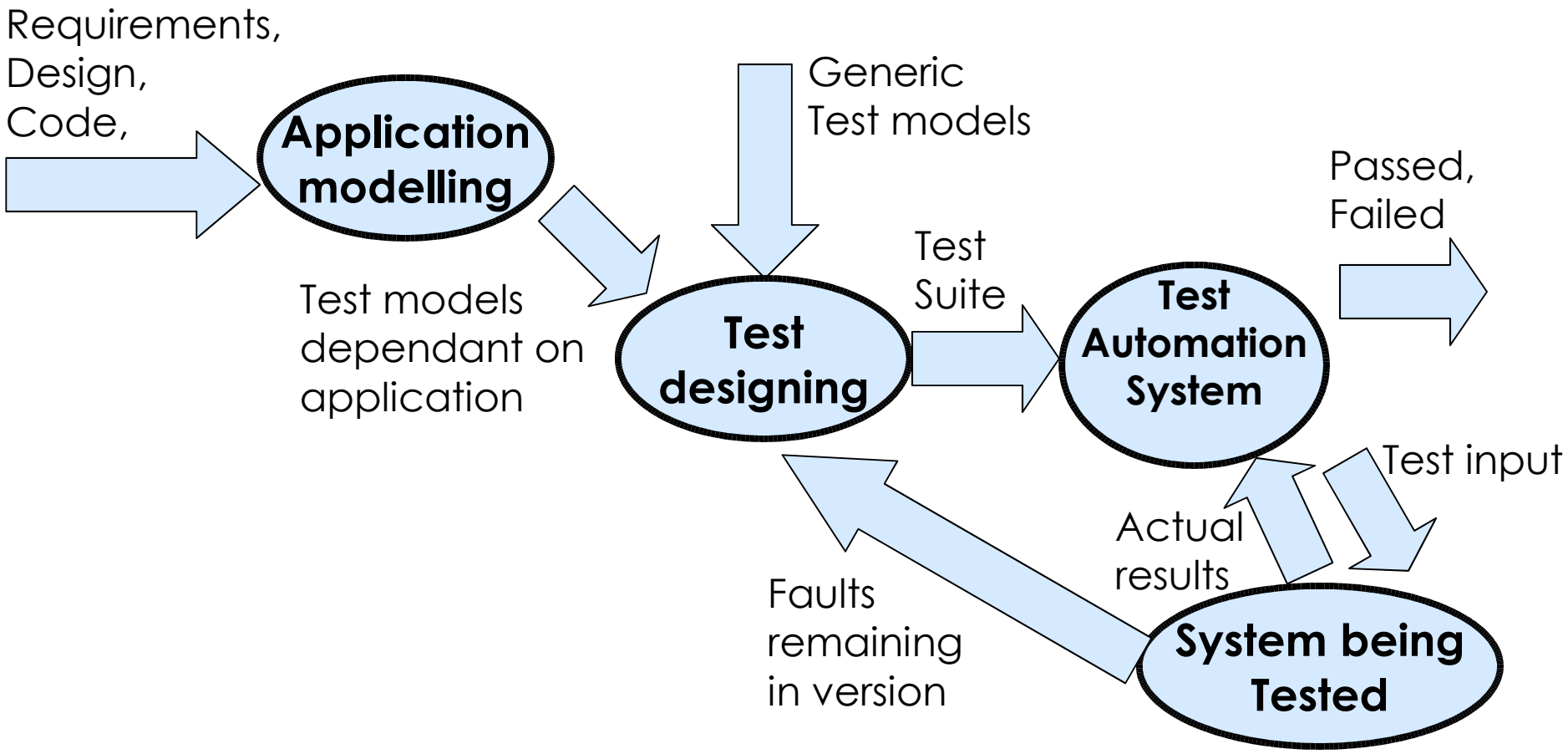
Organisation

- ✓ Teachers
  - Maciej Gabor mgabor@bestcom.com.pl
  - Grzegorz Jachimko grzegorz.jachimko@otp.pl
  - Błażej Pietrzak blazej.pietrzak@cs.put.poznan.pl
- ✓ Lectures
  - Tuesdays, 8:00 AM, G-1
  - Duty Hours: Mondays 3:15-4:45 PM confirmed by e-mail
  - Method of Verification: Test (*pol. kolokwium*) ☺
- ✓ Laboratory classes
  - Mondays, 4:50 PM, 6:30 PM, 427
  - Method of Verification: Grade Average of all given excersises

Lecture proposals

- ✓ Introduction to Software Testing (1 lecture) ☺
- ✓ Test models: Combinational Models (1 lecture)
- ✓ Test models: State Machines (2 lectures)
- ? A Tester's Guide to UML (1 lecture)
- ? Patterns (3 lectures)
- ? Test Driven Development (1 lecture)
- ? Building Maintainable Tests (1-2 lectures)
- ? Test process improvement TPI, TMM (1 lecture)
- ? Reliability measures – IEEE 982-1 1988 (1 lecture)

What is Software Testing?



Software Testing from System Engineering point of view

What is Software Testing? – cont.

Test designing

2. Separate, model and analyze responsibilities of the system being tested.
3. Design **test cases** that meet the above perspective.
4. Add test cases araising from code analysis, assumptions and heuristics.
5. For each test case define **expected results** or select the approach which makes possible to evaluate if the test case passes or fails.

What is Software Testing? – cont.

Test execution:

2. Bring the tested implementation to minimal efficiency by testing interfaces between its parts.
3. Execute **test suite**. Each test is evaluted: passed or failed.
4. Use **coverage** tools to evaluate the coverage of tests.
5. When necessary, add more test cases to test the not covered code
6. Stop testing, when coverage goal is met and all tests passed.

What is Software Testing? – cont.

Software testing is designing, executing and evaluating the test cases.

Software testing is running the code for combinations of states and input data in order to detect faults.

The Limits of Testing

Exhaustive testing is usually impossible (intractable)!

```
for (int i = 0; i < n; i++) {
    if (a.get(i) == b.get(i))
        x[i] = x[i] + 100;
    else
        x[i] = x[i] / 2;
}
```

n	Path No.
1	3
2	5
10	1025
60	1 152 921 504 606 847 200

The Limits of Testing

Fault Sensitivity – the ability to hide faults from test suite

Coincidental Correctness – the code produces correct results for some input data

```
short scale(short j) {
    j = j - 1;      //should be j = j + 1
    j = j / 30000;
    return j;
}
```

Friedman M. A., Voas J.M.: Software Assessment: reliability, safety, testability. New York, John Wiley & Sons 1995

Only -30001, -30000, -1, 0, 29999, 30000 generate incorrect results.

For 99,9908% of input values the code generates correct results.

Fault Model

Why are these properties tested with this technique?

Answer can be based on common sense, experience, assumption, analysis or experiment.

Fault Model shows associations and components of the system being tested that have highest probability for faults to occur.

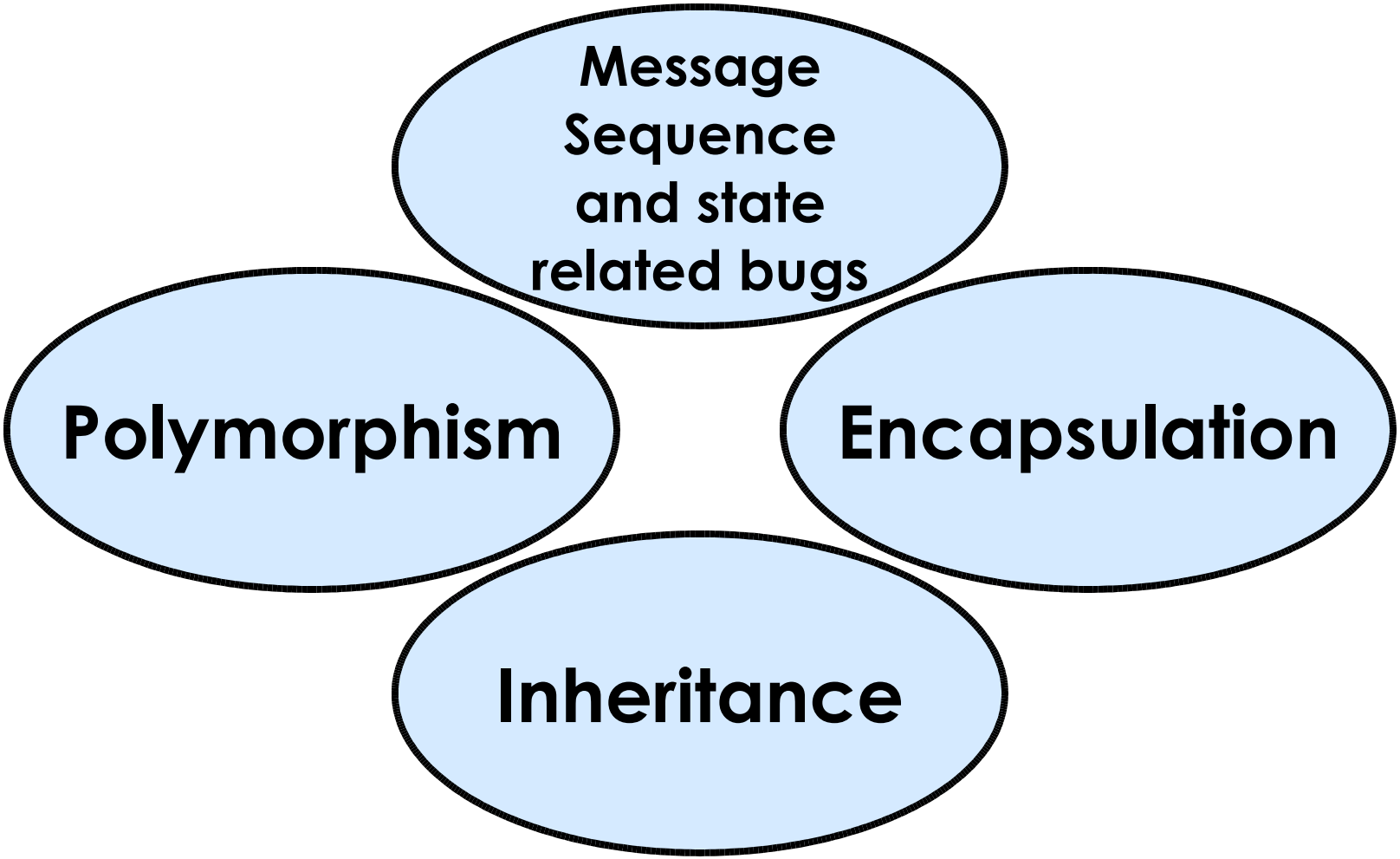
Bug hazard is a circumstance that increase the chance of a bug.

Fault Model

Generally there are two kinds of fault models:

- conformance – directed testing
  - to prove conformance with requirements or specificati
  - Tests are representative enough for essential features of the system being tested
  - Every fault makes the system not conformant (Nonspecific fault model)
- fault-directed testing
  - To prove implementation faults
  - It is based on observation that conformance can be proved for implementation containing bugs.
  - Specific fault model

Side effects of the Paradigm



Side effects of the Paradigm – Encapsulation

```
class Person {
    public String name;
}
```

Invalid

Do not create invalid objects

Encapsulate Field

Not structure equivalent

```
class Person {
    private String name;
    public Person(String name) {
        setName(name);
    }
    public String getName() {
        return name;
    }
    public String setName(String name) {
        this.name = name;
    }
    protected void walk() { ... }
}
```

Side effects of the Paradigm – Encapsulation cont.

- ✓ Fields can be accessed only via methods or by inheriting classes
- ✓ Information hiding –fields and code can be hidden (e.g. protected methods, private fields)
- ✓ Object contains also other methods than getter/setter methods(is not structure equivalent)
- ✓ Do not create invalid objects (with invalid state)

Side effects of the Paradigm – Encapsulation

- Fields can be accessed only via methods

  It is problematic for a test case to set or access appropriate state of the class

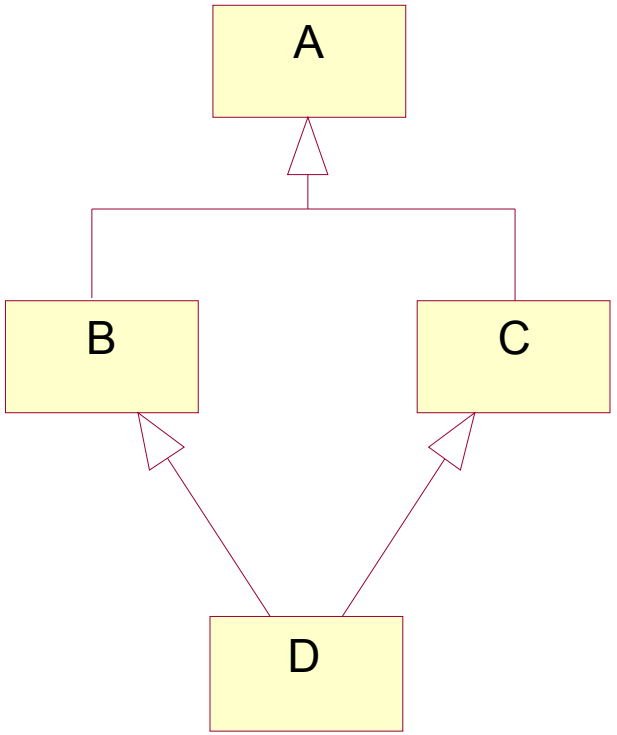
Side effects of the Paradigm – Inheritance

- Inheritance weakens encapsulation
  Inherited fields if not private can be modified by the subclass which can lead to unexpected results
- Coincidental inheritance
  The base class was not prepared for inheritance.

Side effects of the Paradigm – Inheritance

Multiple inheritance

- Repeated inheritance can lead i.e. to name clashes



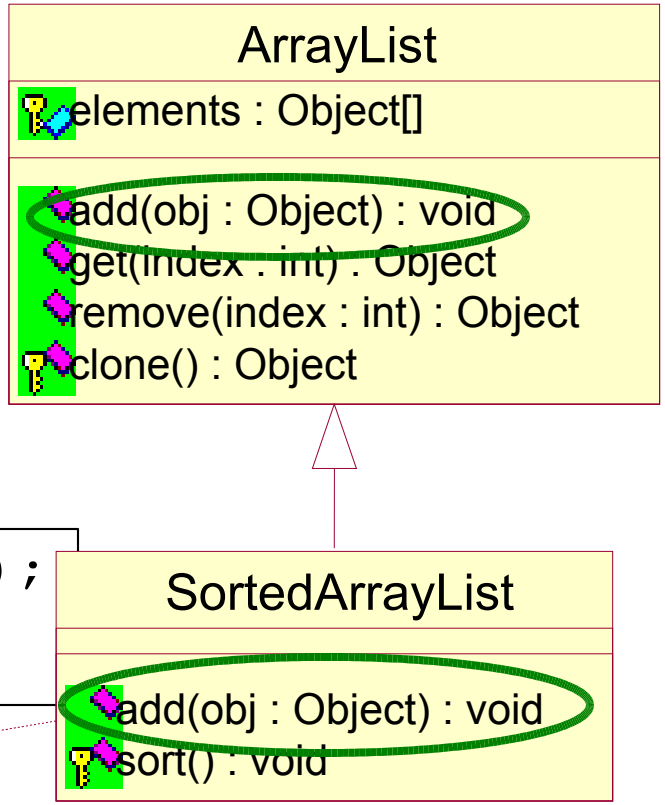
Side effects of the Paradigm – Inheritance cont.

- Abstract classes
  To test abstract class has to be specialized
- Generic classes
  „*Generic classes may never be fully tested*" – *Firesmith*
  Generic classes have to be realized in order to test them.

Side effects of the Paradigm – Polymorphism

Explicit Polymorphism

ability to accept the message by an appropriate class known at run-time




```
ArrayList list = new SortedArrayList();
list.add(new Object);
```

```
void add(Object obj) {
    super.add(obj);
    sort();
}
```

Side effects of the Paradigm – Polymorphism cont.

```
class Adder {
    public double add(double arg1, double arg2) {
        return arg1 + arg2;
    }
    public int add(int arg1, int arg2) {
        return arg1 + arg2;
    }
}
```

Implicit polymorphism

Ability to accept the appropriate message known at run-time.

Side effects of the Paradigm – Polymorphism

- The code can be difficult to understand and thus error-prone
- Even when the interface is correct changes to the polymorphic server can cause the client to fail
- Message can be associated with the wrong server if the client did even though minor mistake
- The Yo-Yo problem

Side effects of the Paradigm – Polymorphism

```
class Account {
    protected Date lastTransaction;
    protected Date today;

    public int calculateQuarters() {
        return 90 / getDays();
    }
    public int getDays() {
        return today.getDay()
            - lastTransaction.getDay() + 1;
    }
}
```

Side effects of the Paradigm – Polymorphism

```
class Deposit extends Account {
    public int getDays() {
        return today.getDay()
            - lastTransaction.getDay();
    }
}
```

Side effects of the Paradigm – Polymorphism

```
class Deposit extends Account {
    public int getDays() {
        return today.getDay()
            - lastTransaction.getDay();
    }
}
```

Incorrect - today's transactions have 0 days

Division by zero!

```
public int calculateQuarters() {
    return 90 / getDays();
}
```

Side effects of the Paradigm – Language-specific hazards

```
class BaseClass {
public:
~Base();
...
}

class SubClass : public BaseClass {
public:
    ~SubClass();
    ...
}
```

```
void doSth() {
    BaseClass *temp = new SubClass;
    delete temp;
}
```

Literatura

- Robert V. Binder: Testowanie systemów obiektowych. Modele wzorce i narzędzia, WNT 2003
- Friedman M. A., Voas J.M.: Software Assessment: reliability, safety, testability. New York, John Wiley & Sons 1995
- Green R.: Java gotchas. March 22, 1998; http://oberon.ark.com/~roedy/gotchas.-html

Quality Assessment

Thank You for your attention ☺

- What is your general impression (1-6)
- Was it too slow or too fast?
- What important did you learn during the lecture?
- What to improve and how?