ADVANCED SOFTWARE ENGINEERING

CS561
TUE/THU @4PM,

Danny Dig
QUIZ #1

- Write down your name

- Grad Program (e.g., EECS PhD, EECS MS) and year of study

- What are your expectations from the class?
Software Engineering as Done Today

Change is the heart of software development:
- add features, fix bugs, support new hardware/UI/OS

Today’s program development is very crude
- change carried manually, through low-level edits
- changes are almost never reused
- versioning tools focus on changes to lines of code

Change is too ad-hoc making software development error-prone, time-consuming, and $$$
- 2/3rd of software costs due to software evolution, some industrial surveys claiming 90%
My View of Tomorrow: Programming is Program Transformation

Change needs to move to a higher-level of abstraction

Program transformations as first-class:
- most changes carried through automated program transformations
- even manual edits become transformations
- programs as sequence of program transformations

Q1: Analyze what software changes occur in practice?
Q2: How can we automate them?
Q3: Can we represent programs as transformations? Archive, retrieve, and visualize them?
Q4: Can we infer higher-level transformations?
Overview of my Approach to Software Evolution Research

- Inferring
  - ECOOP13, ASE09a, ECOOP06
- Archiving & Retrieving
  - TSE08, ICSE07
- Program Transformations (Refactorings)
- Understanding
  - ICST13, ICSM12, ICSE11, SOFTW11, OOPSLA09, ICSE09, ICSE08
- Testing
  - ICST13, TSE10, ASE09b, ISSTA08, FSE07
- Automating
  - ECOOP12, FSE12, IWMSE11, JSME06, ICSM05
Our Refactorings for Parallelism

Refactorings for thread-safety
- make class immutable [ICSE'11]
- convert to Atomic* classes [ICSE'09]
- use concurrent collections [ICSE'09]
- infer region annotations [ASE'09]
- atomic check-then-act operations [ICST’13]

Refactorings for throughput
- parallel recursive divide-and-conquer [ICSE'09]
- loop parallelism via ParallelArray [OOPSLA’10:demo]
- loop parallelism via lambda-enabled functional operators [FSE’13]

Refactorings for scalability
- Atomic*, concurrent collections [ICSE'09]
An immutable object is one whose state can not be changed after it has been constructed.

The transitive state of a deeply immutable object can not be mutated.
Motivation: immutable classes make applications simpler

Simpler parallel applications
• No interference with other threads through immutable objects
• No races ➔ no locks ➔ no deadlocks
• Embarrassingly thread-safe

Simpler sequential applications
• No side effects from method calls
• Efficient comparison based on identity
• Reduced memory footprint through interning
• Secure applications

Simpler distributed applications
• Reduced need for complex proxy update protocols
Mutable classes can be refactored to immutable classes in two steps

1. Generate Immutable Class
2. Adapt Client Code

I will talk about Immutator, a technique and a tool that *Generates Immutable Classes*
Challenges: finding mutator methods and entering/escaping objects

```java
public class Circle {
    private Point center = new Point(0, 0);
    private int radius = 1;

    public void setRadius(int r) {
        radius = r;
    }

    public void moveTo(Point c) {
        center = c;
    }

    public void moveBy(int dx, int dy) {
        Point c = new Point(center.x + dx, center.y + dy);
        moveTo(c);
    }

    public void moveTo(int x, int y) {
        center.setLocation(x, y);
    }

    public Point getCenter() { return center; }
}
```
Find mutator methods that change the state of the instances

```java
public class Circle {
    private Point center = new Point(0, 0);
    private int radius = 1;

    public void setRadius(int r) {
        radius = r;
    }

    public void moveTo(Point c) {
        center = c;
    }

    public void moveBy(int dx, int dy) {
        Point c = new Point(center.x + dx, center.y + dy);
        moveTo(c);
    }

    public void moveTo(int x, int y) {
        center.setLocation(x, y);
    }

    public Point getCenter() {
        return center;
    }
}
```
Find transitive mutators that change the state of objects reachable through fields

```java
public class Circle {
    private Point center = new Point(0, 0);
    private int radius = 1;

    public void setRadius(int r) {
        radius = r;
    }

    public void moveTo(Point c) {
        center = c;
    }

    public void moveBy(int dx, int dy) {
        Point c = new Point(center.x + dx, center.y + dy);
        moveTo(c);
    }

    public void moveTo(int x, int y) {
        center.setLocation(x, y);
    }

    public Point getCenter() { return center; }
}
```

library code, aliases, long call chains, polymorphism
Detect objects that enter or escape. These can be mutated by client code.

```java
public class Circle {
    private Point center = new Point(0, 0);
    private int radius = 1;

    public void setRadius(int r) {
        radius = r;
    }

    public void moveTo(Point c) {
        center = c;
    }

    public void moveTo(int x, int y) {
        center.setLocation(x, y);
    }

    public Point getCenter() {
        return center;
    }

    Enter

    Client Code
    Point point = new Point(2, 2);
    circle.moveTo(point);
    point.setLocation(3, 3);

    Escape
}
```
Analysis and Transformations

- Entering Objects
- Transitive Mutations
- Indirect Mutations
- Mutations
- Escaping Objects
Immutator converts direct mutator methods to factory methods

```java
public void setRadius(int r) {
    radius = r;
}
```

```java
public Circle setRadius(int r) {
    return new Circle(this.center, r);
}
```
Immutator detects methods that are mutating the transitive state of the class

```java
public void moveTo(int x, int y) {
    this.center.setLocation(x, y);
}
```

```java
public Circle moveTo(int x, int y) {
    Circle _this = new Circle(center.clone(), radius);
    _this.center.setLocation(x, y);
    return _this;
}
```

Algorithm creates a summary of variables and fields in the class’ transitive state
Immutator detects objects that enter or escape from methods of the class

```java
public void moveTo(Point c) {
    this.center = c;
}
```

```java
public Circle moveTo(Point c) {
    return new Circle(c.clone(), this.r);
}
```
Static program analysis

Supporting data-structures:
- call graph, control-flow graph
- points-to graph (model objects as allocation sites)

Two novel interprocedural analyses determine safety w.r.t. to transitive state of target class:
1. purity analysis detects side effects
   - reachability problem: mutated state reachable from this
   - summarization algorithm propagates mutation in reversed topological order through call graph
2. class escape analysis detects entering/escaping objects
   - transitive closure of this and outside nodes
   - reachability between the in and out sets
Evaluation Setup

RQ1: Is Immutator safer than manual transformations?
RQ2: How applicable is Immutator?
RQ3: Does it make the programmer more productive?

CaseStudy1: Ran Immutator on all 346 classes from 3 open-source projects

CaseStudy2: Compared Immutator with 11 manual refactorings performed by developers from 6 open-source projects

Controlled Experiment: Asked 6 experienced programmers to refactor by hand
### Immutator is safer

<table>
<thead>
<tr>
<th>project (confirmed by developers)</th>
<th>immutable class</th>
<th>programmer errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mutator</td>
</tr>
<tr>
<td>JDigraph</td>
<td>ImmutableBag</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>HashDigraph</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ArrayGrid2D</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>MapGrid2D</td>
<td></td>
</tr>
<tr>
<td>WALA</td>
<td>ImmutableByteArray</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ImmutableStack</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ImmutableEntry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ImmutableEntry</td>
<td>-</td>
</tr>
<tr>
<td>Guava</td>
<td>ImmutableEntry</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ImmutableEntry</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ImmutableEntry</td>
<td>-</td>
</tr>
<tr>
<td>peaberry</td>
<td>ImmutableAttribute</td>
<td>-</td>
</tr>
<tr>
<td>Spring</td>
<td>ImmutableFlowMapper</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>ImmutableEntry</td>
<td>-</td>
</tr>
</tbody>
</table>

Developers of JDigraph applied our patch

Other developers updated the documentation with warnings about entering/escaping objects

Controlled experiment participants refactoring contained 6.3 errors/class

All 13,000 tests pass before/after our refactoring
Immutator dramatically improves productivity

Saves analyzing 57 methods/refactoring and rewriting 45 lines per refactored class

Immutator is fast: 2 sec/refactoring compared with 27 min/manual refactoring
Practical Impact of My Research

Inferring
- used at Google
- dozen labs

Automating
- ship
- ship/ongoing

Program Transformations (Refactoring)

Understanding
- influenced Java and .NET official concurrency libraries
- learnparallelism.net
30,000 visitors in 6 mo

Testing
- testing infra at Oracle

founded Workshop on Refactoring Tools, HotSwUp

first open-source refactoring
17,000 downloads
Course Administration

Check Wiki:
http://classes.engr.oregonstate.edu/eecs/fall2014/cs561/index.php

2 work items due today:
- sign up on Piazza (all communications through Piazza, no email)
- Read one paper about how to read Soft Eng papers

Prereqs: took at least two classes of undergrad coursework in Software Engineering (e.g., CS361/CS362)
Course Administration

Research-based course:
- Participate in class discussion and activities.
- Read two research papers for every class meeting (around 10 pages, double column => total of 800 research pages during the term)
- For each class meeting submit a one-page critique of one paper before class (at 12pm)
- Prepare and deliver presentations of the selected research papers

- Homework
- Complete a term project (teams of 2 students)
Projects Focus on Mobile & Cloud Applications

New converging forces that reshape computing
- end users spend most time on mobile apps
- by 2016, more than 300B applications downloaded [Gartner]

Technological shifts/opportunities:
- mobile devices are all going multicore
- constraints on memory/CPU/bandwidth/battery usage
- connectivity with the cloud

Encouraging good (old) software engineering practices
Transformations for Mobile Applications

What are the new transformations we need to automate?
- inspiration from explorative studies

Examples of transformations:
- adding concurrency in apps to improve responsiveness
- candidate programs with trade-offs between performance & power consumption
- adaptation to different display technologies
- split functionality between the device and cloud
“Change is the only guaranteed constant”

To foster a revolution in software technology, we need to raise the level of abstraction for changes.

Interactive, automated transformation more effective than manual.

Many of our tools ship with official release.
- YOU can make a difference too.

Today's brand new programs are tomorrow's legacy programs.
- software evolution becomes the primary paradigm of software development.