CSE P 590
Beyond Coverage: Modern Testing and Debugging
Spring 2019

Best practices and Version control

April 09, 2019
Recap: what is program analysis?

- (Automatically) analyze the behavior of a program
  - optimize the program or
  - check program’s behavior (against its specification)
- Concerned with properties such as
  - Correctness
  - Safety
  - Liveness
  - Performance
- Can be static or dynamic or a combination of both
Recap: static vs. dynamic analysis

**Static analysis**
- Build an abstraction of run-time states (and prove a property of the program)
- Reason about the program without executing it

**Dynamic analysis**
- Execute the program with some inputs
- Observe behavior
Today

- Best practices (or how to avoid bugs)
- Version control with git
- Class projects
- In-class exercise 1
Logistics

- Course material, schedule, etc. on website: https://homes.cs.washington.edu/~rjust/courses/2019Spring/CSEP590
- Submission of assignments via Canvas: https://canvas.uw.edu
- Discussions on Piazza: piazza.com/washington/spring2019/csep590

Having trouble accessing any of those -- let us know!
How to avoid bugs in your code?
How to avoid bugs in your code?

It’s super simple...don’t introduce them during coding!
How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
2. Correctness: get it right the first time
3. Bug visibility
How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
   a. Programming language
      i. Ever had a use-after free bug in a Java program?
      ii. Ever had an assignment bug (String to Integer) in a statically typed language?
   b. Libraries and protocols
      i. TCP vs. UDP
      ii. No overflows in BigInteger
How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
   a. Programming language
   b. Libraries and protocols

2. Correctness: get it right the first time
   a. A program without a spec is bug free
   b. Keep it simple, modular, and testable
   c. Defensive programming and conventions (discipline)
How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
   a. Programming language
   b. Libraries and protocols

2. Correctness: get it right the first time
   a. A program without a spec is bug free
   b. Keep it simple, modular, and testable
   c. Defensive programming and conventions (discipline)

3. Bug visibility
   a. Assertions (pre/post conditions)
   b. (Regression) testing
   c. Fail fast
Testing vs. Verification

What’s the difference between verification and validation?

\[
\Gamma, x : \tau \vdash x : \tau \\
\Gamma, x : \tau_p \vdash t : \tau_r \\
\Gamma \vdash (\lambda x : \tau_p.t) : \tau_p \to \tau_r \\
\Gamma \vdash t_f : \tau_p \to \tau_r \\
\Gamma \vdash t_p : \tau_p \\
\Gamma \vdash t_f \; t_p : \tau_r
\]
Quiz: setup and goals

- 2-3 students per team
- 6 code snippets
- 2 rounds
  - **First round**
    - For each code snippet, decide whether it represents good or bad practice.
    - **Goal:** discuss and reach consensus on good or bad practice.
  - **Second round** (known “solutions”)
    - For each code snippet, try to understand why it is good or bad practice.
    - **Goal:** come up with an explanation or a counter argument.
Round 1: good or bad?
Snippet 1: good or bad?

```java
public File[] getAllLogs(Directory dir) {
    if (dir == null || !dir.exists() || dir.isEmpty()) {
        return null;
    } else {
        int numLogs = ... // determine number of log files
        File[] allLogs = new File[numLogs];
        for (int i=0; i<numLogs; ++i) {
            allLogs[i] = ... // populate the array
        }
        return allLogs;
    }
}
```
public void addStudent(Student student, String course) {
    if (course.equals("CSEP590")) {
        p590Students.add(student);
    }
    allStudents.add(student)
}
Snippet 3: good or bad?

```java
public enum PaymentType {DEBIT, CREDIT}

public void doTransaction(double amount, PaymentType payType) {
    switch (payType) {
        case DEBIT:
            ... // process debit card
            break;
        case CREDIT:
            ... // process credit card
            break;
        default:
            throw new IllegalArgumentException("Unexpected payment type");
    }
}
```
public int getAbsMax(int x, int y) {
    if (x<0) {
        x = -x;
    }
    if (y<0) {
        y = -y;
    }
    return Math.max(x, y);
}
public class ArrayList<E> {
    public E remove(int index) {
        ...
    }
    public boolean remove(Object o) {
        ...
    }
    ...
}
Snippet 6: good or bad?

```java
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }
}
```
Round 2: why is it good or bad?
My take on this

- Snippet 1: bad
- Snippet 2: bad
- Snippet 3: good
- Snippet 4: bad
- Snippet 5: bad
- Snippet 6: good
Snippet 1: this is bad! why?

```java
public File[] getAllLogs(Directory dir) {
    if (dir == null || !dir.exists() || dir.isEmpty()) {
        return null;
    } else {
        int numLogs = ... // determine number of log files
        File[] allLogs = new File[numLogs];
        for (int i=0; i<numLogs; ++i) {
            allLogs[i] = ... // populate the array
        }
        return allLogs;
    }
}
```
Snippet 1: this is bad! why?

```java
public File[] getAllLogs(Directory dir) {
    if (dir == null || !dir.exists() || dir.isEmpty()) {
        return null;
    } else {
        int numLogs = ... // determine number of log files
        File[] allLogs = new File[numLogs];
        for (int i=0; i<numLogs; ++i) {
            allLogs[i] = ... // populate the array
        }
        return allLogs;
    }
}
```

Null references...the billion dollar mistake.
Snippet 1: this is bad! why?

```java
public File[] getAllLogs(Directory dir) {
    if (dir == null || !dir.exists() || dir.isEmpty()) {
        return null;
    } else {
        int numLogs = ... // determine number of log files
        File[] allLogs = new File[numLogs];
        for (int i=0; i<numLogs; ++i) {
            allLogs[i] = ... // populate the array
        }
        return allLogs;
    }
}

File[] files = getAllLogs();
for (File f : files) {
    ...
}
```

Don’t return null; return an empty array instead.
Snippet 1: this is bad! why?

```java
public File[] getAllLogs(Directory dir) {
    if (dir == null || !dir.exists() || dir.isEmpty()) {
        return null;
    } else {
        int numLogs = ... // determine number of log files
        File[] allLogs = new File[numLogs];
        for (int i=0; i<numLogs; ++i) {
            allLogs[i] = ... // populate the array
        }
        return allLogs;
    }
}
```

No diagnostic information.
Snippet 2: short but bad! why?

```java
public void addStudent(Student student, String course) {
    if (course.equals("CSEP590")) {
        p590Students.add(student);
    }
    allStudents.add(student)
}
```
Snippet 2: short but bad! why?

```java
public void addStudent(Student student, String course) {
    if (course.equals("CSEP590")) {
        p590Students.add(student);
    }
    allStudents.add(student)
}
```

Defensive programming: add an explicit assertion (or write the literal first).
Snippet 3: this is pretty good, but why?

```java
public enum PaymentType {DEBIT, CREDIT}

public void doTransaction(double amount, PaymentType payType) {
    switch (payType) {
    case DEBIT:
        ... // process debit card
        break;
    case CREDIT:
        ... // process credit card
        break;
    default:
        throw new IllegalArgumentException("Unexpected payment type");
    }
}
```
Snippet 3: this is pretty good, but why?

```java
public enum PaymentType {DEBIT, CREDIT}

public void doTransaction(double amount, PaymentType payType) {
    switch (payType) {
        case DEBIT:
            ... // process debit card
            break;
        case CREDIT:
            ... // process credit card
            break;
        default:
            throw new IllegalArgumentException("Unexpected payment type");
    }
}
```

Type safety using an enum; throws an exception for unexpected cases (e.g., future extensions of PaymentType).
Snippet 4: also bad! huh?

```java
public int getAbsMax(int x, int y) {
    if (x<0) {
        x = -x;
    }
    if (y<0) {
        y = -y;
    }
    return Math.max(x, y);
}
```
Snippet 4: also bad! huh?

```java
public int getAbsMax(int x, int y) {
    if (x<0) {
        x = -x;
    }
    if (y<0) {
        y = -y;
    }
    return Math.max(x, y);
}
```

Method parameters should be final; use local variables to sanitize inputs.
Snippet 5: Java API, but still bad! why?

```java
public class ArrayList<E> {
    public E remove(int index) {
        ...
    }
    public boolean remove(Object o) {
        ...
    }
    ...
}
```
Snippet 5: Java API, but still bad! why?

```java
public class ArrayList<E> {
    public E remove(int index) {
        ...
    }
    public boolean remove(Object o) {
        ...
    }
    ...
}
```

```java
ArrayList<String> l = new ArrayList<>();
Integer index = Integer.valueOf(1);
l.add("Hello");
l.add("World");
l.remove(index);
```

What does the last call return?
Snippet 5: Java API, but still bad! why?

public class ArrayList<E> {
    public E remove(int index) {
        ...
    }
    public boolean remove(Object o) {
        ...
    }
}

ArrayList<String> l = new ArrayList<>();
Integer index = Integer.valueOf(1);
l.add("Hello");
l.add("World");
l.remove(index);

Avoid method overloading, which is statically resolved.
Autoboxing/unboxing adds additional confusion.
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }
}
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
}

Good encapsulation; immutable object.
Version control
What is version control?

- Version control records changes to a set of files over time, making it easy to review or obtain specific versions later.

- Simple Example
  - Alice writes a paper, using version control: v1.0
  - Alice corrects grammatical mistakes: v1.1
  - Alice discovers new findings and rewrites her paper: v1.2
  - Alice realizes her findings are wrong: restore v1.1

- Version control is used by
  - Developers (obviously)
  - Researchers
  - Applications (e.g., (cloud-based) word processors)
Why use version control?
Why use version control?

More likely the case...
Why use version control?

Just kidding … this is more realistic
Centralized version control

- There exists **one “central” copy** of a project where all developers commit their changes to.
- Each developer has a working copy. As soon as she/he commits, the repository gets updated.
- Examples: Subversion, CVS.
Distributed version control

- Multiple **clones** of a repository exist.
- Each developer has access to a local (private) repository.
- All committed changes remain local unless **pushed** to another repository.
- No external changes are visible unless **pulled** from another repository.
- Examples: Git, Hg (Mercurial).
Conflicts

• Conflicts occur when two users make a simultaneous change to the same line of a file.
• When conflicts arise, the last committer needs to choose which line(s) to keep.

Are conflicts more likely in (de)centralized version control?
Branches

- One **main** development branch (**master**, trunk, etc.).
- To add a new feature, it’s useful to create a new **branch** -- an independent line of development.
Merge

C0 → C1 → C2 → C3 → experiment

C3 → master
Merge
Rebase: changing the commit history

- Rebasing lets us change and sequentialize our commit history
- A powerful tool, but … think about pros and cons
Rebase
Rebase
Version control with Git

- “Because my hatred of CVS has meant that I see Subversion as being the most pointless project ever started, because the whole slogan for the Subversion for a while was 'CVS done right' or something like that. And if you start with that kind of slogan, there is nowhere you can go.”

- "‘what would CVS never ever do’-kind of approach”
A Motivating Example: What is this Git command?

NAME
   git-______ - ______ file contents to the index

SYNOPSIS

DESCRIPTION
This command updates the index using the current content found in the working tree, to prepare the content staged for the next commit. It typically ______s the current content of existing paths as a whole, but with some options it can also be used to ______ content with only part of the changes made to the working tree files applied, or remove paths that do not exist in the working tree anymore.
A Motivating Example: What is this Git command?

NAME
   git-add - Adds file contents to the index

SYNOPSIS

DESCRIPTION
This command updates the index using the current content found in the working
tree, to prepare the content staged for the next commit. It typically adds the
current content of existing paths as a whole, but with some options it can also
be used to add content with only part of the changes made to the working tree
files applied, or remove paths that do not exist in the working tree anymore.
A Motivating Example: What is this Git command?

NAME
   git-______ - Forward-port local commits to the updated upstream head

SYNOPSIS
   git ______ [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
   [<upstream> [<branch>]]
   git ______ [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
            --root [<branch>]
   git ______ --continue | --skip | --abort | --edit-todo

DESCRIPTION
If <branch> is specified, git ______ will perform an automatic git checkout
<branch> before doing anything else. Otherwise it remains on the current branch.

If <upstream> is not specified, the upstream configured in branch.<name>.remote
and branch.<name>.merge options will be used (see git-config[1] for details) and
the --fork-point option is assumed. If you are currently not on any branch or if
the current branch does not have a configured upstream, the _____ will abort.
A Motivating Example: What is this Git command?

NAME

git-rebase - Forward-port local commits to the updated upstream head

SYNOPSIS

git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
  [<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
  --root [<branch>]
git rebase --continue | --skip | --abort | --edit-todo

DESCRIPTION

If <branch> is specified, git rebase will perform an automatic git checkout
<branch> before doing anything else. Otherwise it remains on the current branch.

If <upstream> is not specified, the upstream configured in branch.<name>.remote
and branch.<name>.merge options will be used (see git-config[1] for details) and
the --fork-point option is assumed. If you are currently not on any branch or if
the current branch does not have a configured upstream, the rebase will abort.
Git: concepts and terminology
Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, **not the file itself**
- tree: git's representation of a file system.
- working tree: tree representing what is currently checked out (what you see)
- staged: ready to be committed
- commit: a set of database entries detailing a snapshot of the working tree
- ref: pointer to a commit object
- branch: basically just a (special) ref; semantically: represents a *line of dev*
- HEAD: a ref pointing to the working tree
Git: understanding its manpages

SYNOPSIS

`git-diff-index` [-m] [--cached] [<common diff options>] <tree-ish> [<path>…]

DESCRIPTION

`git-diff-index` compares the content and mode of the blobs found in a tree object with the corresponding tracked files in the working tree, or with the corresponding paths in the index.

SYNOPSIS

`git-allocate-remote` [ --derive-head | --massage-link-head | --abduct-commit ]

DESCRIPTION

`git-allocate-remote` allocates various non-branched local remotes outside added logs, and the upstream to be packed can be supplied in several ways.

SYNOPSIS

`git-resign-index` [ --snap-file ] [ --direct-change ]

DESCRIPTION

`git-resign-index` resigns all non-stashed unstaged indices, and the --manipulate-submodule flag can be used to add a branch for the upstream that is counted by a temporary submodule.
Git: understanding its manpages

SYNOPSIS

```
git-diff-index [-m] [--cached] [<common diff options>] <tree-ish> [<path>…]
```

DESCRIPTION

git-diff-index compares the content and mode of the blobs found in a tree object with the corresponding tracked files in the working tree, or with the corresponding paths in the index.

SYNOPSIS

```
git-allocate-remote [ --derive-head | --massage-link-head | --abduct-commit ]
```

DESCRIPTION

git-allocate-remote allocates various non-branch local remotes outside added logs, and the upstream to be packed can be configured in several ways.

SYNOPSIS

```
git-resign-index [ --snap-file ] [ --direct-change ]
```

DESCRIPTION

git-resign-index resigns all non-stashed unstaged indices, and the --manipulate-submodule flag can be used to add a branch for the upstream that is counted by a temporary submodule.
Higher-level questions

- What’s the difference between reset, revert, and rebase?
- Why/when or why/when not use rebase? What are pros and cons?
Class projects

Project proposals

1. Code coverage
   a. A new code coverage tool for Java
   b. API for existing code coverage tools (Cobertura, JaCoCo)

2. Mutation testing
   a. Compiler-integrated mutator (compiler plugin)
   b. Mutation analyzer (standalone or IDE plugin)
   c. Visualization for mutation testing results

3. Fault database/benchmark (Defects4J)
   a. Build system inference
   b. Commit minimization

4. Static analysis: pluggable type checker

5. Additional proposals
   a. Duplicate code detection (C#)
   b. Fuzzing (Java)
   c. Implement a Clang checker (LLVM)
   d. Improved Checkstyle (already 5 team members)

Select a project on Canvas by 04/11!