Software Testing, Quality Assurance & Maintenance—Lecture 6

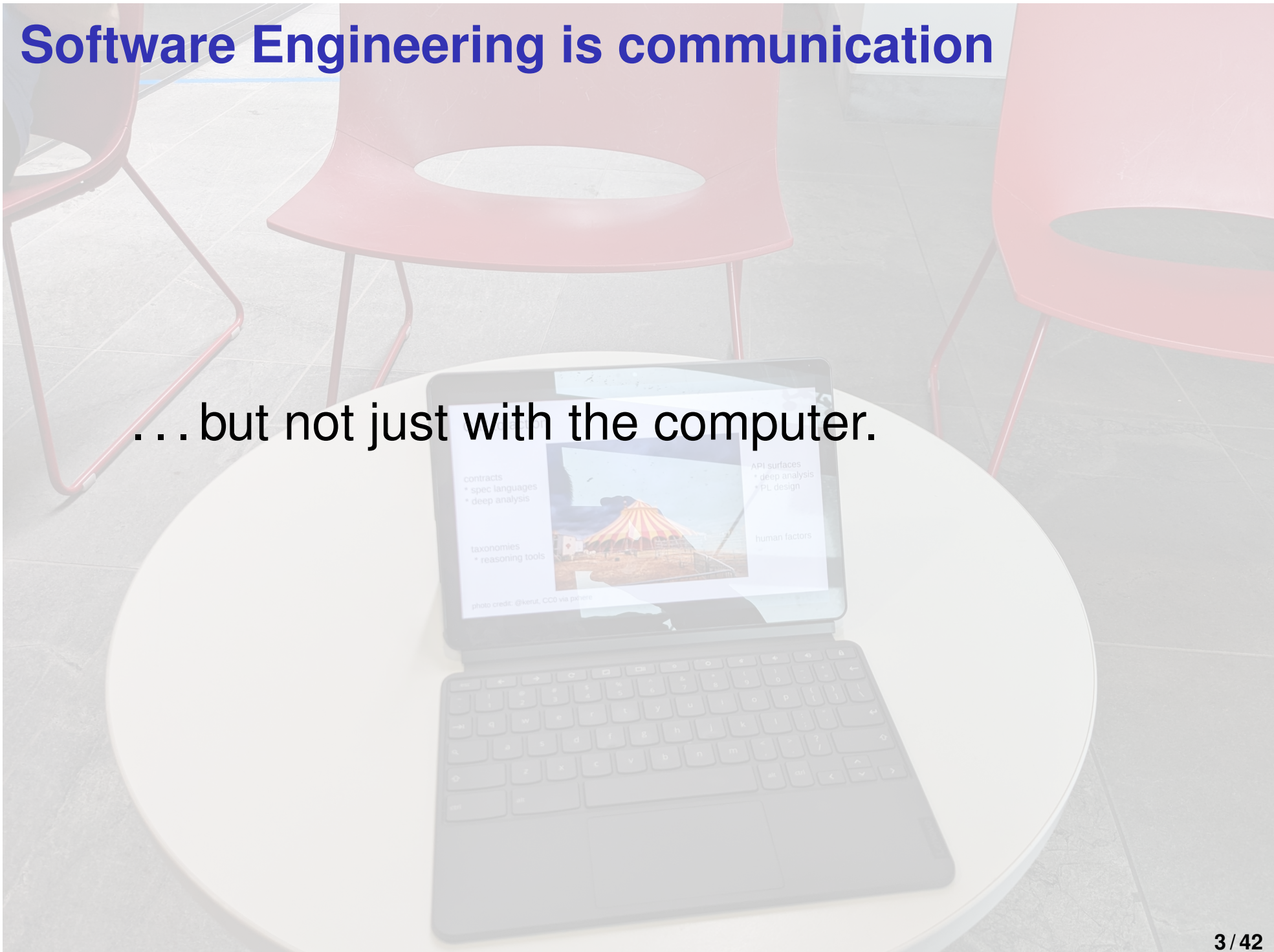
Patrick Lam
University of Waterloo

January 23, 2026

Part I

Code Review

Software Engineering is communication



. . . but not just with the computer.

Teamwork is key

What Code Review Is

Reviewer must:

- read someone else's code;
- communicate suggestions to the code author.

Almost always in the context of a proposed code change.

Experience Report: Vadim Kravcenko

At a startup, early in his career:

> *The real problem was that no one could understand what the hell anyone else had written; we had duplicate logic in many places and different code styles in our modules. It was really bad.*

`https://vadimkravcenko.com/shorts/code-reviews/`

Code Review: Purpose

An old comment on Hacker News says (`https://news.ycombinator.com/item?id=8862602`):

> From my experience doing reviews and having my code reviewed the most important thing is to understand:
>
> - What the intention of change is (what is this trying to achieve)
> - What is the code actually doing

Code Review: Levels

and it continues:

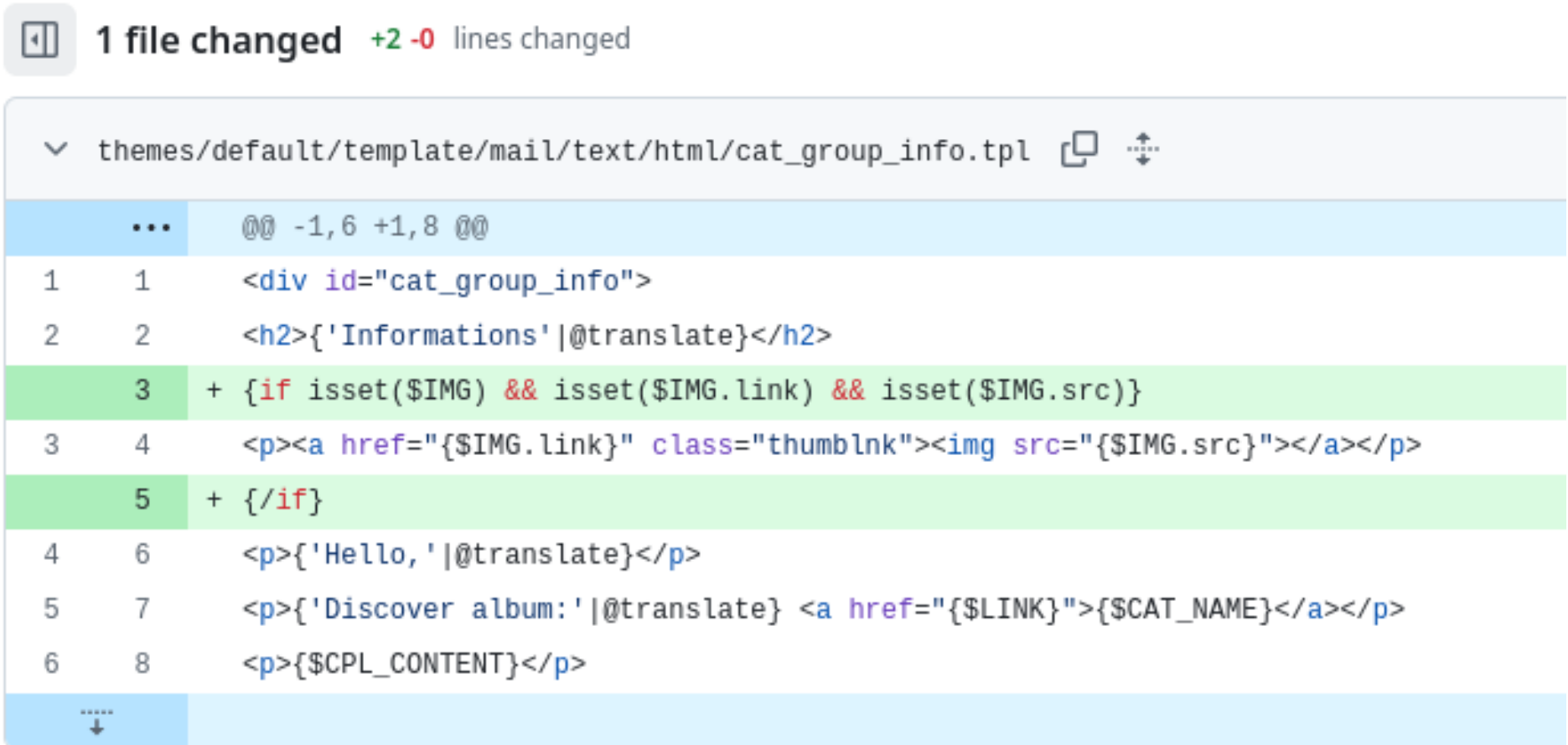
> There are three levels of review I typically observe:
>
> 1. Skim, find one or two comments to leave. LGTM!
> 2. Review each line or method in isolation. Recognize style and formatting errors. Identify a couple local bugs.
> 3. Understand the purpose and the code.

Code Review: Benefits

Our HN commenter observes:

> #3 takes time. It's what it takes however to find the issues where the implementation doesn't actually accomplish what was set out to do (at least in all cases). To recognize where the code is going to break when integrated with other modules. To suggest big simplifications.
>
> #3 is also where you get one of the biggest review benefits—shared understanding of the code base.

Review all the things at #3?



1 file changed +2 -0 lines changed

themes/default/template/mail/text/html/cat_group_info.tpl

```
@@ -1,6 +1,8 @@
<div id="cat_group_info">
<h2>{'Informations'|@translate}</h2>
+ {if isset($IMG) && isset($IMG.link) && isset($IMG.src)}
<p><a href="{$IMG.link}" class="thumblnk"><img src="{$IMG.src}"></a></p>
+ {/if}
<p>{'Hello,'|@translate}</p>
<p>{'Discover album:'|@translate} <a href="{$LINK}">{$CAT_NAME}</a></p>
<p>{$CPL_CONTENT}</p>
```

LGTM is probably fine here.

On formatting and style

Use a tool to enforce consistent standards, e.g.

- positioning of { }
- spaces vs tabs

Receiving feedback

Usually not useful: getting into an argument.

Let them finish, thank them for their input.

Consider whether you agree with it or not.

In the context of code review

Some misconceptions Kravcenko points out:

- “So if the code is bad = they are bad”
- “US VS THEM”

Let's review this code

```
public static int dayOfYear(int month, int dayOfMonth,
                            int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    }
    // ... through month == 12
    return dayOfMonth;
}
```

Code Smells

- don’t repeat yourself
- fail fast
- avoid magic numbers
- one purpose per variable

What to do with dayOfYear?

- frame proposed changes positively
- don’t provide overwhelming amounts of feedback at once
- I’d probably propose a pair programming session

Good documentation example

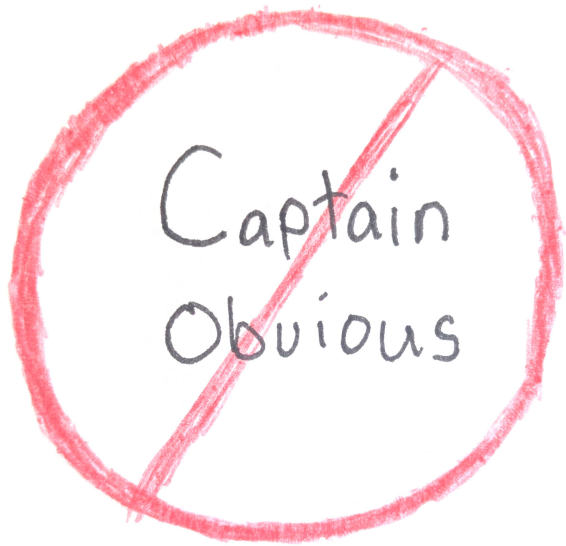
```
/**
 * Compute the hailstone sequence.
 * See http://en.wikipedia.org/wiki/Collatz_conjecture
 * @param n starting number of sequence; requires n > 0.
 * @return the hailstone sequence starting at n
 *         and ending with 1.
 *         For example, hailstone(3)=[3,10,5,16,8,4,2,1].
 */
public static List<Integer> hailstoneSequence(int n) {
    ...
}
```

Citing your sources

```
// adapted from Eli Bendersky's Lexer:
//    http://eli.thegreenplace.net/2013/07/16/hand-written
// public domain according to author
// modifications by Patrick Lam
```

Writing Comments

Explain why, not what.



Nonviolent code review

- it's about the code, not the coder
- no personal feelings (on both sides)
- write positive comments, aiming to improve the code

Use I-messages

× “You write code like a toddler.”

✓ “I’m finding it hard to understand what’s happening here.”

Prefer to ask (non-“why”) questions.

How long per review?

Kravchenko suggests: 30–60 minutes

Part II

Static code analysis (via PMD)

Static and dynamic analysis

Static analysis:

look at the source code of a system & infer properties (including code smells and fault detection)

Dynamic analysis:

look at the executions of a system & infer properties (example: coverage)

Static analysis tools

We'll talk about PMD (`pmd.github.io`).

Other tools exist: SpotBugs, SAST (by gitlab), clippy (for Rust), etc.

We said: use tools to flag style issues.
PMD can do this.

Using PMD

Easiest: in an IDE, using built-in PMD rulesets.

PMD has rulesets for many languages, including C++, Scala, Java.

For Java: many rulesets, grouping related rules.

You can choose rules to apply for your project.

PMD Example: SimplifyConditional [design ruleset]

Detect redundant null checks.

```
class ConditionalToBeSimplified {
  void bar(Object x) {
    if (x != null && x instanceof Bar) {
      // just drop the "x != null" check
    }
  }
}
```

PMD Example: UseCollectionIsEmpty [design ruleset]

Better to use `c.isEmpty()` rather than `c.size() == 0`.

```
class UseIsEmpty {
    void good() {
        List foo = getList();
        if (foo.isEmpty()) { /* ... */ }
    }

    void bad() {
        List foo = getList();
        if (foo.size() == 0) { /* ... */ }
    }
}
```

PMD Example: MisplacedNullCheck [basic ruleset]

Don't check nullness after relying on non-nullness.

```
class RedundantNullCheck {
    void bar() {
      if (a.equals(baz) || a == null) {
        // don't need to check a == null
        /* ... */
      }
}
```

PMD Example: UseNotifyAllInsteadOfNotify [design ruleset]

Usually, `notifyAll()` is the right call to use, not `notify()`.
Unless you know what you're doing, using `notify()` is going to result in stuck threads, which is a bug.

```
class Notifier {
    void bar() {
      synchronized(this) {
        // should likely be .notifyAll()
        this.notify();
      }
    }
}
```

PMD

Find more about the above rules at
`https://pmd.github.io/pmd/pmd_rules_java.html`.

We saw rules from the design and basic rulesets.

Other rulesets: specifically for JUnit; detecting empty or otherwise useless code; naming conventions; etc.

Static analysis: strengths

Can find security issues, e.g. uses of unsafe APIs like `gets()`.

Can write custom checkers for new vulnerabilities as they come out.

Static analysis is exhaustive; good at finding missing error handling or edge cases.

Static analysis: limitations

Because these tools are exhaustive, they return false positives.

Interprocedural analysis is especially hard.

People ignore excessive false positives.

Tools don't substitute for human judgment in saying how important issues are.

Writing your own PMD rules

You can also write your own rules, via XPath queries or Java visitors.

SemGrep is supposedly easier to write rules for compared to PMD: the marketing copy says “Semgrep rules look like the code you already write”.

On XPath

Write selectors to find nodes.

Example following: a simple XML document.

Other applications of XPath: web programming (Document Object Model); and, also, Java code (PMD).

Example XML file

```
<?xml version="1.0" encoding="UTF-8"?>

<bookstore>
 <book>
  <title lang="fr">Harry Potter</title>
  <price>29.99</price>
 </book>
 <book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
 </book>
</bookstore>
```

(Play along at `https://www.w3schools.com/xml/tryit.asp?filename=try_xpath_select_cdnodes`.)

Three Queries

XPath expression: `//price`
Result: set of `<price>` nodes with data `29.99`, `39.95`.

`//` means any descendants (including self) of the context node (= root node, here).

XPath expression: `count(//price)`
Result: 2.

XPath expression: `/bookstore/book[1]/title`
Result: `Harry Potter`.
If we omitted `[1]`, then we'd get all of the titles.

Predicates

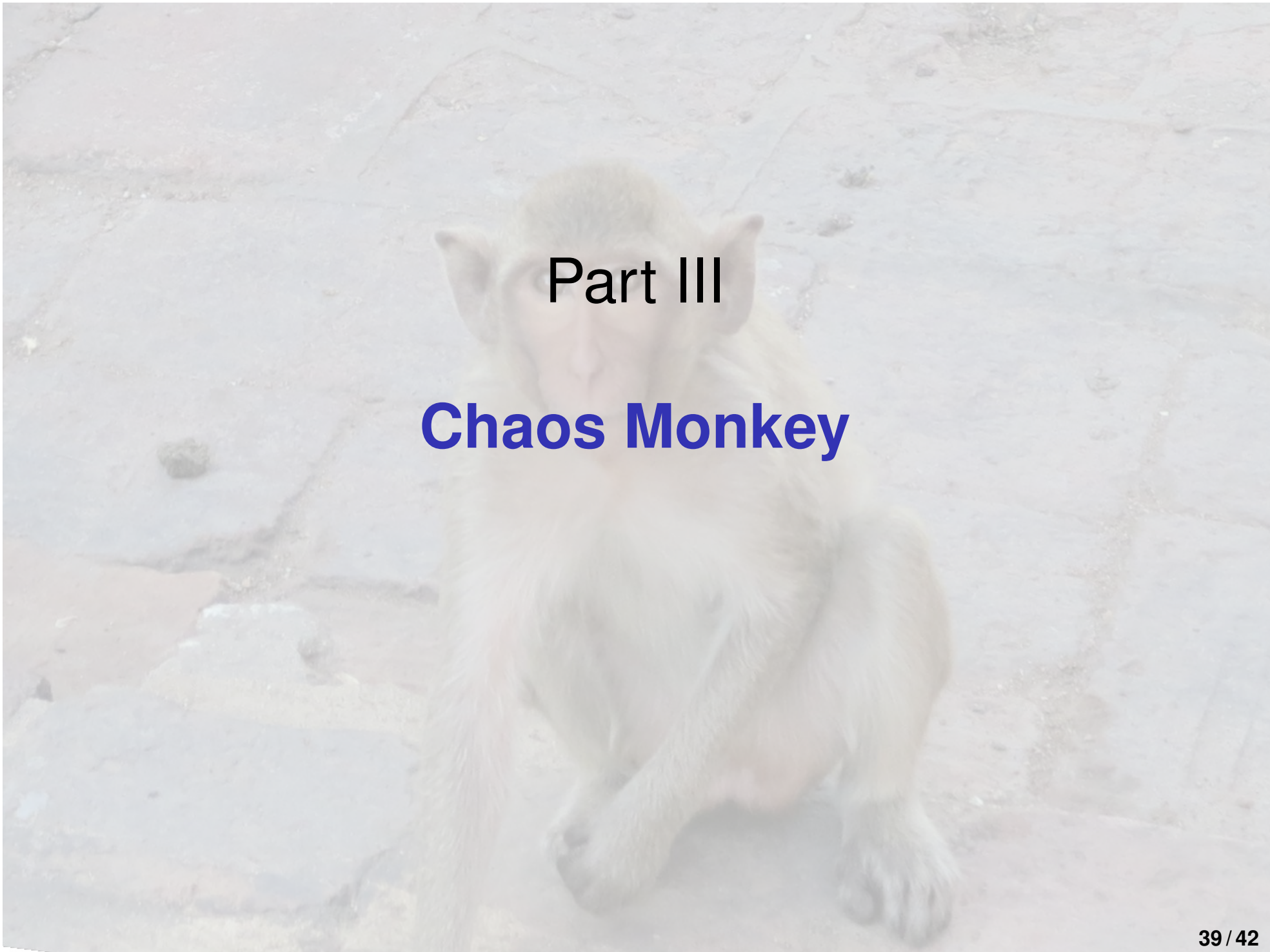
XPath: `/bookstore/book[price>35]/title`
Result: titles of books with price greater than 35.

Can put arbitrary tree queries in []s.

XPath: `//price[../title[text()="Learning XML"]]`.
XPath: `//title[../price < 35 or @lang="en"]`.

`*` works as you might expect.

`//` includes descendants and self.
For descendants excluding self, write e.g.
`descendant::book`.



Part III

Chaos Monkey

Motivation: Chaos Monkey

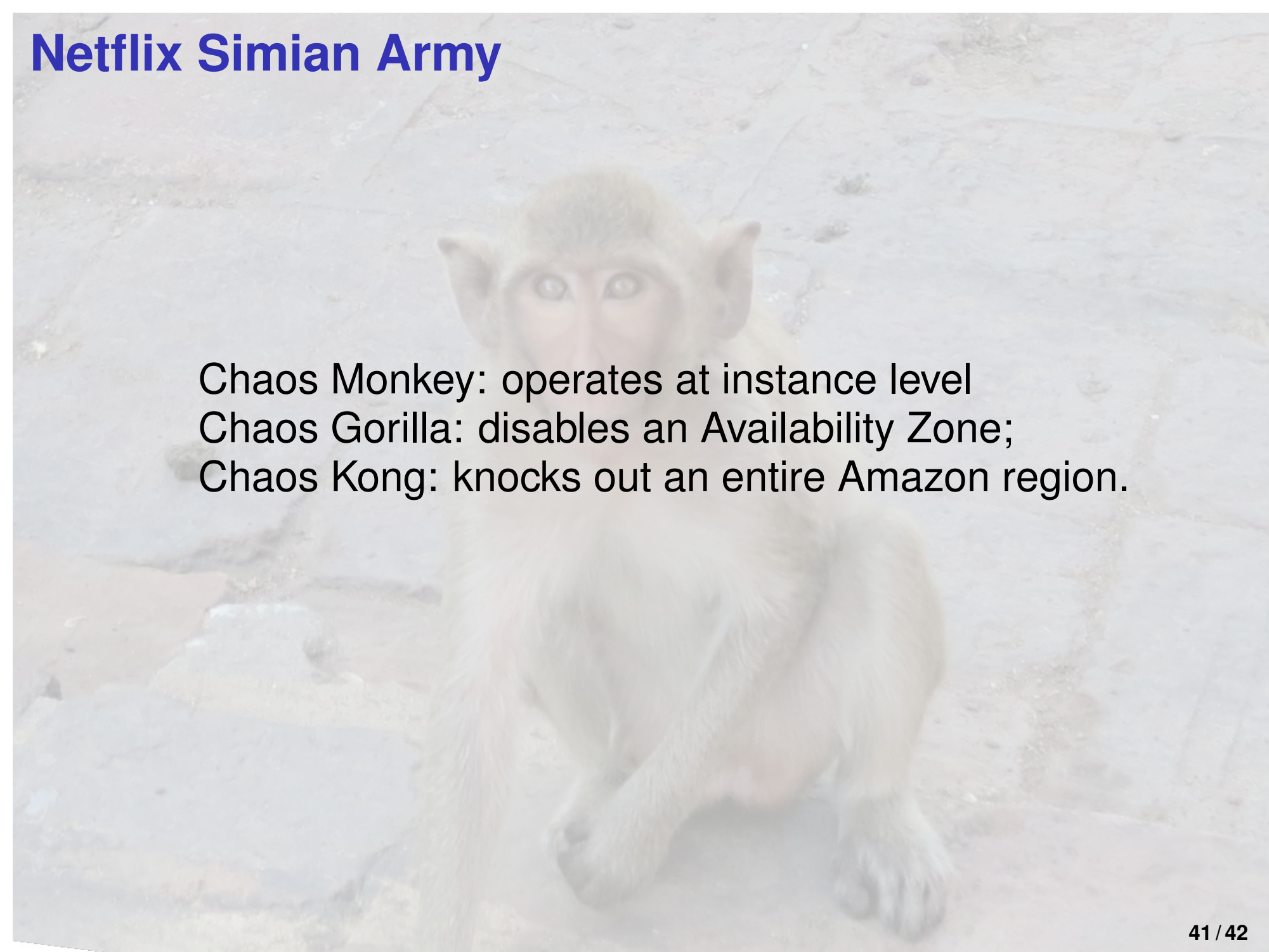
Instead of inputs, think distributed systems.

Some instances (components) randomly fail
(because of bogus inputs, or . . .).

Ideally: system continues to work.

Failures are inevitable, need a strategy to deal with,
better to encounter not-at-4am.

Netflix Simian Army



Chaos Monkey: operates at instance level
Chaos Gorilla: disables an Availability Zone;
Chaos Kong: knocks out an entire Amazon region.

Jeff Atwood Quotes

Why inflict such a system on yourself?
“Sometimes you don’t get a choice; the Chaos Monkey chooses you.”

Software engineering benefits:

> “Where we had one server performing an essential function, we switched to two.”
> “If we didn’t have a sensible fallback for something, we created one.”
> “We removed dependencies all over the place, paring down to the absolute minimum we required to run.”
> “We implemented workarounds to stay running at all times, even when services we previously considered essential were suddenly no longer available.”