SOFTWARE MAINTENANCE AND EVOLUTION

--- REFACTORING FOR ASYNC ---

CS563
WEEK 3 - THU
Danny Dig
Course Objectives: Project

Practice a research or novel-industrial project through all its stages:
- formulate problem,
- find the current state of the practice/research (i.e., related work)
- design and implement solution
- evaluate the solution empirically
- write about it
- present progress, receive feedback
Mobile Devices are Everywhere

300B apps downloaded annually [Gartner]

Will be the main control agents for IoT

Most end-users spend most of their time on mobile devices
Slow Operations Make Mobile Apps Unresponsive

300B apps downloaded annually [Gartner]

Slow operations freeze the UI and frustrate users
- 75% of performance bugs in Android [Li et al., ICSE’14]
Why are Apps Unresponsive?

- **Event Queue**
  - **onStart**
  - **onKeyUp**
  - **onClick**

- **App Process**
- **UI thread**
How frequently have you encountered unresponsive apps?

- Never
- Rarely
- Occasionally
- A moderate amount
- A great deal
The Perils of Asynchronous Programming

Different programming model (call-backs invert flow of control)

Data races between async call and main thread (e.g., 77 data races on 13 top apps)
Modernize legacy async code from call-backs to async/await

On some platforms, Microsoft no longer supports legacy async

Non-trivial changes: preserve program behavior and even the order of exceptions
The Perils of Asynchronous Programming

Misused async constructs hurt performance, cause deadlocks

4% of async code runs synchronously

Async code that constantly badgers UI kills responsiveness by “a thousand paper cuts”
The Perils of Asynchronous Programming

Short-running async constructs can lead memory leaks, lost results, and wasted energy.

Requires changing stateful into stateless code: from shared memory communication to distributed style.
Overview of Our Transformations for Asynchronous Programming

sync → async

async → modern async
async → performant async
async → long-running async
A STUDY AND TOOLKIT FOR ASYNCHRONOUS PROGRAMMING IN C#

Semih Okur
David Hartveld
Danny Dig
Arie van Deursen
private void SnapAndPost ()
{
    Busy = true;
    UpdateUIStatus ("Taking a picture");
    var picker = new Xamarin.Media.MediaPicker ();
    var picTask = picker.TakePhotoAsync (new Xamarin.Media.StoreCameraMediaOptions ());
    picTask.ContinueWith ((picRetTask) => {
        InvokeOnMainThread (() => {
            if (picRetTask.IsCanceled) {
                Busy = false;
                UpdateUIStatus ("Canceled");
            } else {
                var tagsCtrl = new GetTagsUIViewController (picRetTask.Result.GetStream ());
                PresentViewController (tagsCtrl, true, () => {
                    UpdateUIStatus ("Submitting picture to server");
                    var uploadTask = new Task (() => {
                        return PostPicToService (picRetTask.Result.GetStream (), tagsCtrl.Tags);
                    });
                    uploadTask.ContinueWith ((uploadRetTask) => {
                        InvokeOnMainThread (() => {
                            Busy = false;
                        });
                    });
                    uploadTask.Start ();
                });
        });
    });
}
async/await keywords

Special language constructs introduced in C# 5 (2012)

Make asynchronous programming a first-class citizen!

Possible to express efficient asynchronous code in a familiar direct style

async: method identifier
await: used to define pausing points
our study

(i) We do not know how developers use asynchronous programming in practice.
(ii) Asynchronous programming is hard to use and developers lack tool support for it.

Conducted a first large-scale formative study about asynchronous programming to answer two RQs:

Q1) How do developers use asynchronous programming?
   Built Asyncifier, a refactoring tool.

Q2) Do developers misuse async/await?
   Built Corrector, a bug fixing tool.
Analyzed 1378 open source Windows Phone apps, comprising 12M SLOC, produced by 3376 developers.

Built a static analysis tool with Microsoft’s Roslyn (C# compiler APIs: syntax, symbol, binding)
Developers need a refactoring tool: ASYNCIFIER

Developers heavily use callback-based operations

Microsoft officially no longer recommends these operations

Refactoring from callbacks to async/await is non-trivial

We built ASYNCIFIER:
a refactoring tool to upgrade callback code to async/await
Developers need a bug fixing tool: 
Corrector

Developers frequently misuse async/await keywords

We built CORRECTOR: a transformation tool to find and correct the misused async/await

Batch mode to fix existing mistakes
Quick fix (VS) mode prevents user from introducing mistakes
Empirical evaluation

Used the same code corpus from the Formative Empirical Study: 1378 WP apps, 12M SLOC

(EQ1) Are they applicable?
    A: Yes, applied successfully on thousands of instances

(EQ2) What is the impact on productivity?
    A: Each refactoring changes 29 LOC, non-trivial

(EQ3) What is the run-time performance?
    A: < 2sec, suitable for interactive use

(EQ4) Do developers find the transformations useful?
Developers find Asyncifier & Corrector useful

Chose 10 most recently updated apps that have callbacks. Applied Asyncifier ourselves and offered patches. 9 out of 10 apps responded and accepted 28 refactorings.

“That was pretty good, look forward to the release of refactoring tool, it seems to be really useful.” Ocell app

Chose 20 most recently updated apps (where we found async err)
All apps accepted all our 288 instances of CORRECTOR transformations

“What you have pointed out is correct. This time, performance has been improved by 2x.” Softbuild.Data app

CORRECTOR ships now with official release of
Paper conclusions

First large-scale formative study on the usage of asynchronous programming

A toolkit inspired by our empirical study:
(1) refactoring, (2) bug fixing tools

Evaluation: highly applicable, efficient, useful

Architects of C# and F# confirmed that our findings will influence the language design

Parts of CORRECTOR already ship with official Visual Studio 2014
Discussion

What is the most unexpected result from the paper?

How to make impact on the practice of SE?

What is a different workflow of using these tools?

What did the authors do right for an award paper?

What could have been improved?

New research ideas?
Lesson: The Killer Offer

Failure - The Backdoor to Success

Establish the value of the paper, than go the extra mile

Killer Feature #1: large-scale formative study for mapping the landscape where you plan to create a solution

Killer Feature #2: large-scale empirical evaluation (e.g., 2972 applied refactorings in LambdaFicator – FSE’13)

Killer Feature #3: show you add value in the real world (e.g., developers accepted 288 of our patches)
Discussion

New research ideas?
- Rama, Mihai, Sam: cross platform, cross-language refactoring
- Alex, Ameya, Ujjival: Expanding Asyncifier with more styles of coding
- Nirvik, Ameya: further development to new APM constructs
- Ameya: to increase adoption, augment tool suggestions with ML
- Spencer, Ujjival: educate programmers through IDE, give live recommendations

Open questions:
- Rama, Mihai, Sam: Why focus on C# and Windows apps?
- Mihai: ConfigureAwait(false)
- Nirvik: who to broaden preconditions to accept more apps
- Ameya, Spencer: false positives and false negatives
- Ujjival: how to get idioms?