SOFTWARE MAINTENANCE AND EVOLUTION

--- REFACTORING FOR ASYNC ---

CS563
WEEK 3 - TUE
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?

Activity Browser (in application Browser) is not responding.

onStart
onKeyUp
onClick

App Process
UI thread

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

Semih Okur
David Hartveld
Danny Dig
Arie van Deursen

36th International Conference on Software Engineering

ICSE 2014
Hyderabad, India · May 31 - June 7, 2014
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
Corrector developers need a bug fixing tool: CORRECTOR.

We built CORRECTOR: a transformation tool to find and correct the misused async/await keywords. Developers frequently misuse async/await keywords.

Quick fix (VS) mode prevents user from introducing mistakes. Batch mode to fix existing 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 Visual Studio
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
- Braden, Deval, Dan: analyze async anti-patterns in other paradigms and languages
- Malinda:
  - extend the ideas to replace long-running UI operations with async code
  - Language independent analysis platforms for async code
- Vijay: implement detection with ML

Open questions:
- Miguel: How to identify mapping from blocking to non-blocking?
- Braden: why choose C# over other platforms?
  - The future of async programming?
- Deval: WP7 vs WP8
- Malinda: instantiate async from inside async
- Vijay: what are the best practices, what about other platforms?
Converting Sync into Async Code via AsyncTask

Yu Lin, Cosmin Radoi, Danny Dig
A Formative Study of AsyncTask

100 most popular Android projects from Github
  – 1.34M SLOC by 1139 developers

Find the commits where AsyncTask is introduced for the first time
  – Compare code before/after the commit
Use, Misuse, and Underuse of \textit{AsyncTask}

**Use**
- half of apps use \textit{AsyncTask}
  - 46% manually refactored, 54% introduced new

  Performance: 4% of \textit{AsyncTask} run synchronously

**Misuse**
- Correctness: 13 apps manually refactored had 77 races

**Underuse**
- 251 places in 51 apps execute long-running in UI
Asynchronizer, our refactoring tool

Code Transformation
  – Extract selected code into background thread by using AsyncTask

Safety Check
  – Check data races introduced by code transformation

Empirical evaluation
void onCreate() {
    ListView lv = getListView();
    final String query = "select ...";
    lv.addFooterView(mCsr.get(0));
}

class ProcessRoutes extends AsyncTask{
    ListView lv;
    ProcessRoutes(ListView lv){
        this.lv = lv;
    }
    Cursor doInBackground(String... args){
        String query = args[0];
        return mCsr;
    }
    void onPostExecute(Cursor mCsr) {
    }
}
What is the continuation?

- Naive approach
  - move all statements after the selected code

Our approach considers
- Control-flow, i.e., return statements are not moved
- Data-flow: statements that are data dependent need to stay together
Safety Check

_IteRace [Radoi and Dig., ISSTA’13]_

– Analyzing data races in parallel loops statically
– Specializations: thread-structure, summarizes at
  library/application membrane, dynamic context-sensitivity

Two challenges to address event-driven model

– Statically determining what instructions may execute in
  parallel with doInBackground
– Constructing call graph
class RouteselectActivity extends Activity {
    public void onCreate() {
        button.setOnClickListener(
            new OnClickListener(){
                public void onClick() { ... }
            }
        );

        new AsyncTask() {
            void doInBackground(...){
                ... mCsr = db.rawQuery(query);
            }
            }.execute();

        ... if (mCsr.getCount() == 0) return;
    }
}
Evaluation

RQ1: Applicability
   – How applicable is the code transformation?

RQ2: Accuracy
   – How accurate is the code transformation?

RQ3: Effort
   – How much effort can be potentially saved?

RQ4: Safety
   – Is the automated refactoring safer than manual refactoring?

RQ5: Value
   – Do developers accept our refactorings?
Experimental Methodology

Reenact existing refactoring scenarios

– Retain all projects with at least two manual refactorings
– Apply refactorings to the same code as the manual refactorings did
– Answer RQs: Applicability, Accuracy, Effort, Safety

Apply refactorings in new scenarios

– Start from 19 projects where other researchers detected potential responsiveness issues
– Apply refactorings to long-running operations in UI thread, and submit patches to the developers
– Answer RQs: Value
Asynchronizer is applicable, accurate, and saves human effort

RQ1: Applicability
- applied 135 refactorings in 19 apps. 95% pass preconditions

RQ2: Accuracy
- 99% of changes are similar with the ones applied by developers

RQ3: Effort
- Asynchronizer changed 2394 LOC in 62 files
- on average, 31 non-trivial LOC changed, in less than 10 sec
Asynchronizer finds real races

- Races fixed in a later version
  - On average, 193 days elapsed to fix them
Asynchronizer is Valuable

62 refactoring patches in 7 apps

Accepted: 38 in 6 apps

“Thanks for your contribution”

Rejected: 24 in 4 apps

VLC app: “I think that most of the calls you put AsyncTask are not very costly”
Paper Discussion

New research ideas

Miguel, Braden: automated recommendation for refactoring opportunities (e.g., via profiling)

Deval, Dan: support other platforms (iOS) or other languages (JavaScript)

Dan: tools to help me learn how concurrency in Android works
Open questions?

Miguel: which code to move in the onPostExecute in order to improve UI responsiveness?

Deval: how do you define a popular OSS?

Braden: how can we better educate programmers?
implications

Developers

LearnAsync.NET: hundreds of real-world positive and negative examples of asynchronous idioms
(EQ1) our tools are highly applicable

54% of 1245 callbacks passed the preconditions.
ASYNCIFIER refactored 54% of callbacks.
We manually verified that 10% of the instances are correct.

The reasons for un-refactored 46% of 1245 callbacks:
(i) Algorithm’s preconditions
(ii) Tool limitations

CORRECTOR fixed 4 main types of anti-patterns (2209 instances)

Our tools can be applied to any platform written in C# (not only Windows Phone)
(EQ2) Automation is needed

ASYNCIFIER changes 28.9 lines on average per refactoring!

(EQ3) Our tools are efficient

ASYNCIFIER takes 508ms on average per refactoring

CORRECTOR takes 47ms per transformation

suitable for an interactive usage in an IDE