COMP 490/L Lecture 1

Kyle Dewey

Disclaimer

- This is a **research-oriented** senior design course
- If you are looking for the **traditional industry-oriented** senior design course, you are in the wrong class
  - Same time: Prof. Wiegley in JD 2213

About Me

- My research: automated test case generation and CS education
- This is my third semester at CSUN
- First time teaching this course

About this Class

- First time this version of the class is taught
- See something wrong? Want something improved? Email me about it! (kyle.dewey@csun.edu)
- I generally operate based on feedback

Bad Feedback

- This guy sucks.
- This class is boring.
- This material is useless.

–I can't do anything in response to this

Good Feedback

- This guy sucks, *I can't read his writing.*
- This class is boring, *it's way too slow.*
- This material is useless, *I don't see how it relates to anything in reality.*

- I can't fix anything if I don't know what's wrong

-I can actually do something about this!

Target Audience

-Maybe you're interested in graduate school...

Target Audience

-Maybe you're interested in academia...

Target Audience

Fuzzing the Rust Typechecker Using CLP

Kyle Dewey Jared Roesch Ben Hardekopf
University of California, Santa Barbara
{kyledewey, jroesch, benh}@cs.ucsb.edu

***Abstract*—Language fuzzing is a bug-finding technique for testing compilers and interpreters; its effectiveness depends upon the ability to automatically generate valid programs in the language under test. Despite the proven success of language fuzzing, there is a severe lack of tool support for fuzzing statically-typed languages with advanced type systems because existing fuzzing techniques cannot effectively and automatically generate well-typed programs that use sophisticated types. In this work we describe how to automatically generate well-typed programs that use sophisticated type systems by phrasing the problem of well-typed program generation in terms of Constraint Logic Programming (CLP). In addition, we describe how to specifically target the typechecker implementation for testing, unlike all existing work which ignores the typechecker. We focus on typechecker precision bugs, soundness bugs, and consistency bugs. We apply our techniques to Rust, a complex, industrial-strength language with a sophisticated type system.**

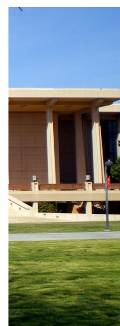
I. INTRODUCTION

The central idea of a language fuzzer is to automatically generate valid programs in a given language, which are then fed to a language implementation under test in order to check for crashes or miscompilations. This idea is well-established as a confidence-building and bug-finding technique for com-

a logical proposition, we can straightforwardly encode types and type systems using CLP. Because programs are proofs, querying the CLP engine whether a type is "true" corresponds to generating a well-typed program. The nondeterminism inherent in CLP languages means that when there are multiple possible proofs (i.e., multiple well-typed programs) the CLP engine can easily generate all possible solutions—that is, it can output as many well-typed programs as we desire. This method for automated program generation allows us to take advantage of long-standing existing implementations of CLP [7], [8] and community wisdom about effectively using CLP [9].

Our second advance describes techniques for specifically testing typechecker implementations. The three main kinds of typechecker bugs we target are (1) **precision bugs**, where the typechecker conservatively rejects well-behaved programs it should accept; (2) **soundness bugs**, where the typechecker optimistically accepts potentially ill-behaved programs it should reject; and (3) **consistency bugs**, where the typechecker treats a set of equivalent programs (in terms of being well- or ill-typed) inconsistently, accepting some while rejecting others.

Testing for precision bugs requires only that we generate well-typed programs as described previously and then



-But most of all, you're interested in publishing papers
-Papers are a gateway into graduate school and academia, and represent a significant portion (and often the most difficult part) of either one of them

Target Audience

Fuzzing the Rust Typechecker Using CLP

Kyle Dewey Jared Roesch Ben Hardekopf

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

Abstract testing com the ability language u fuzzing, th statically-ty existing fu generate w this work v programs t problem of Logic Prog specifically unlike all e on typeche bugs. We a strength la

The ce generate v fed to a la for crashe as a confi

Abstract

Compilers should be correct. To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input. In this paper we present our compiler-testing tool and the results of our bug-hunting study. Our first contribution is to advance the state of the art in compiler testing. Unlike previous tools, Csmith generates programs that cover a large subset of C while avoiding the undefined and unspecified behaviors that would destroy its ability to automatically find wrong-code bugs. Our second contribution is a collection of qualitative and quantitative results about the bugs we have found in open-source C compilers.

Categories and Subject Descriptors D.2.5 [*Software Engineer-*

```
1   int foo (void) {
2     signed char x = 1;
3     unsigned char y = 255;
4     return x > y;
5   }
```

Figure 1. We found a bug in the version of GCC that shipped with Ubuntu Linux 8.04.1 for x86. At all optimization levels it compiles this function to return 1; the correct result is 0. The Ubuntu compiler was heavily patched; the base version of GCC did not have this bug.

We created Csmith, a randomized test-case generator that supports compiler bug-hunting using differential testing. Csmith generates a C program; a test harness then compiles the program using several compilers, runs the executables, and compares the out-

-But most of all, you're interested in publishing papers
-Papers are a gateway into graduate school and academia, and represent a significant portion (and often the most difficult part) of either one of them

Target Audience

Fuzzing the Rust Typechecker Using CLP

Kyle Dewey   Jared Roesch   Ben Hardekopf

Finding and Understanding Bugs in C Compilers

Xuejun Yang   Yang Chen   Eric Eide   John Regehr

Korat: Automated Testing Based on Java Predicates

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov
MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
{chandra,khurshid,marinov}@lcs.mit.edu

ABSTRACT

This paper presents Korat, a novel framework for automated testing of Java programs. Given a formal specification for a method, Korat uses the method precondition to automatically generate all (nonisomorphic) test cases up to a given small size. Korat then executes the method on each test case, and uses the method postcondition as a test oracle to check the correctness of each output.

cate (i.e., a method that returns a boolean) from the method's precondition. One of the key contributions of Korat is a technique for automatic test case generation: given a predicate, and a bound on the size of its inputs, Korat generates all nonisomorphic inputs for which the predicate returns `true`. Korat uses backtracking to systematically explore the bounded input space of the predicate. Korat generates *candidate* inputs and checks their validity by invoking the predicate on them. Korat monitors accesses that the predicate

-But most of all, you're interested in publishing papers
-Papers are a gateway into graduate school and academia, and represent a significant portion (and often the most difficult part) of either one of them

Target Audience

Fuzzing the Rust Typechecker Using CLP

Kyle Dewey   Jared Roesch   Ben Hardekopf

Finding and Understanding Bugs in C Compilers

Xuejun Yang   Yang Chen   Eric Eide   John Regehr

Korat: Automated Testing Based on Java Predicates

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov

DART: Directed Automated Random Testing

Patrice Godefroid   Nils Klarlund
Bell Laboratories, Lucent Technologies
{god,klarlund}@bell-labs.com

Koushik Sen
Computer Science Department
University of Illinois at Urbana-Champaign
ksen@cs.uiuc.edu

Abstract

We present a new tool, named DART, for automatically testing software that combines three main techniques: (1) *automated* extrac-

unit testing is so hard and expensive to perform that it is rarely
properly. Indeed, in order to be able to execute and test a comp
in isolation, one needs to write test driver/harness code to sin

-But most of all, you're interested in publishing papers
-Papers are a gateway into graduate school and academia, and represent a significant portion (and often the most difficult part) of either one of them

Target Audience

Fuzzing the Rust Typechecker Using CLP

Kyle Dewey Jared Roesch Ben Hardekopf

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr

Korat: Automated Testing Based on Java Predicates

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov

DART: Directed Automated Random Testing

Patrice Godefroid Nils Klarlund Koushik Sen

Fuzzing with Code Fragments

Christian Holler, *Mozilla Corporation**, *choller@mozilla.com*
Kim Herzig, *Saarland University*, *herzig@cs.uni-saarland.de*
Andreas Zeller, *Saarland University*, *zeller@cs.uni-saarland.de*

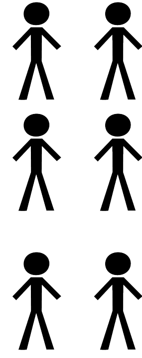
-But most of all, you're interested in publishing papers
-Papers are a gateway into graduate school and academia, and represent a significant portion (and often the most difficult part) of either one of them

Course Design

Traditional Senior Design

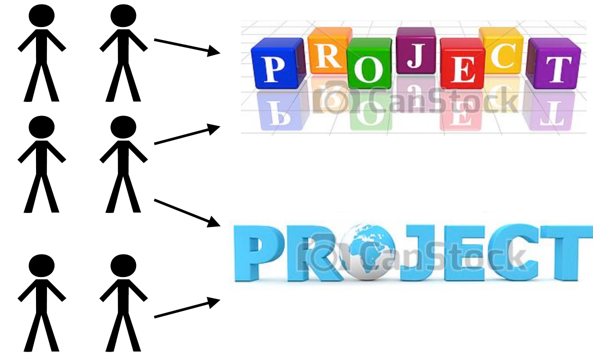
-To get a better sense of what this class is, and how it compares to the usual senior design, let's first explain what the usual senior design process looks like

Traditional Senior Design



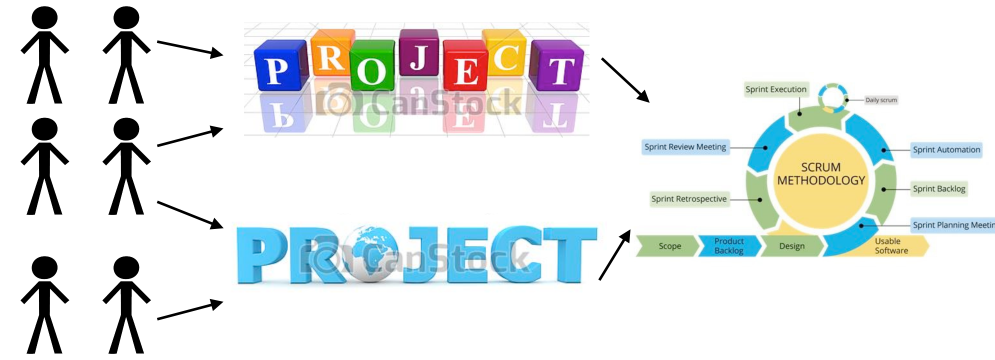
-We have students in the class...

Traditional Senior Design



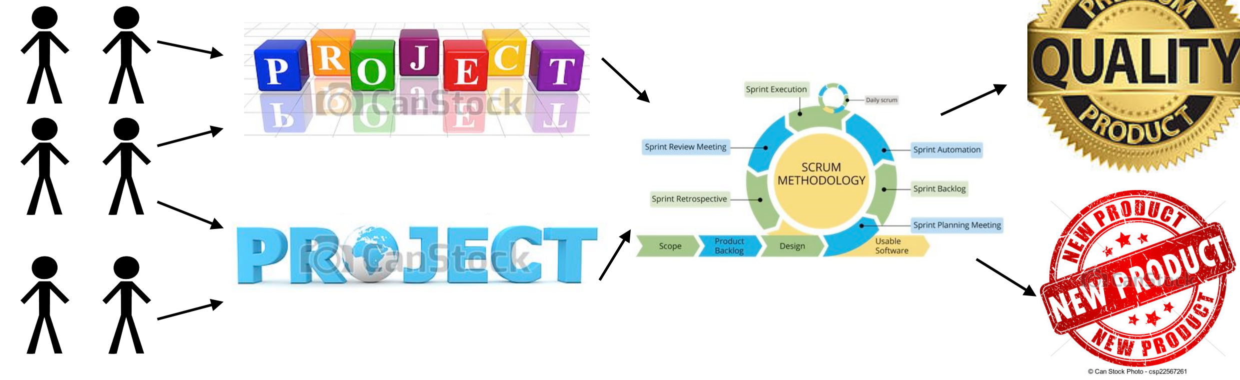
-...and these students pitch / select projects to work on.

Traditional Senior Design



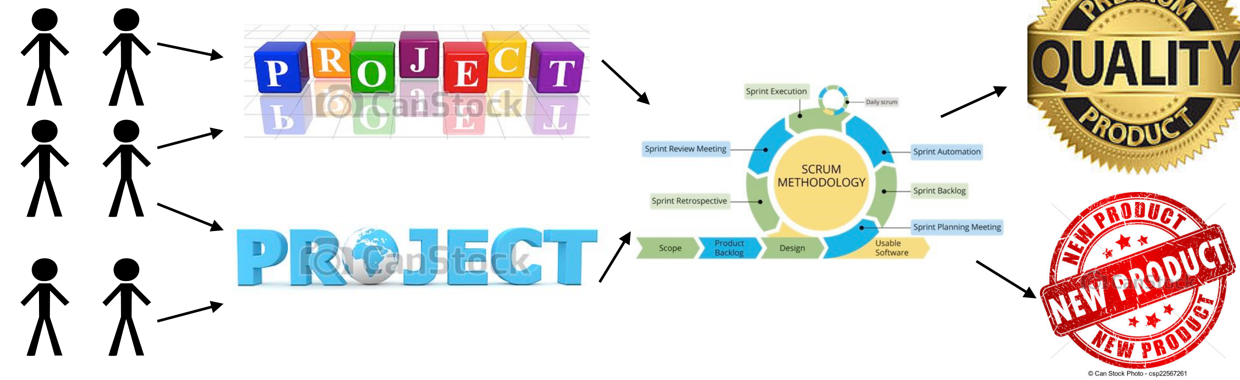
-Students then use industry-standard techniques (learned in the course)...

Traditional Senior Design



–...to turn the project into a product

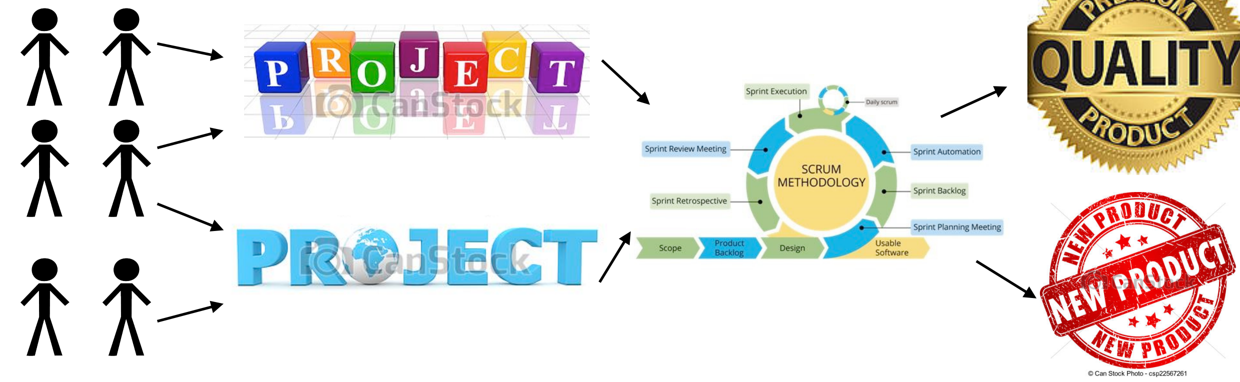
Traditional Senior Design



Research-based Senior Design (Expected Typical)

-The research-based senior design looks a little different
-This is showing how I'm expecting things to go for most students (variations are possible; students can pitch their own projects to me, but talk to me first)

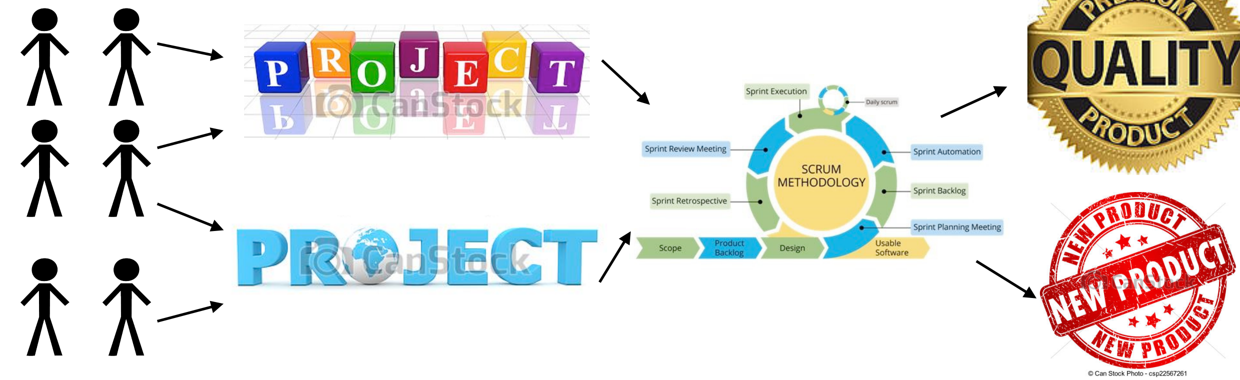
Traditional Senior Design



Research-based Senior Design (Expected Typical)

-In this senior design, we have faculty sponsors

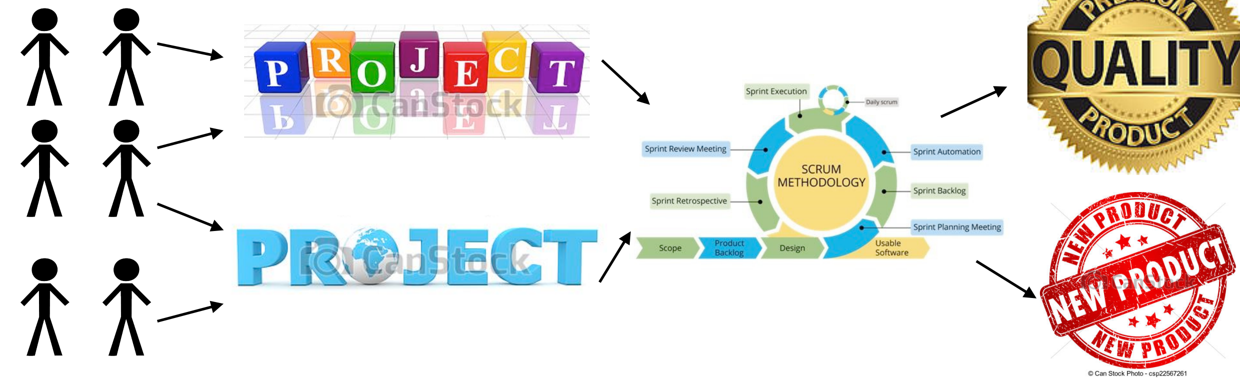
Traditional Senior Design



Research-based Senior Design (Expected Typical)

-Faculty sponsors pitch projects
-Reasoning: it's unexpected that students would know coming in what makes a viable research project
-Similarly, it's practically necessary to have a technical expert in the area in order to make progress when you're first starting out

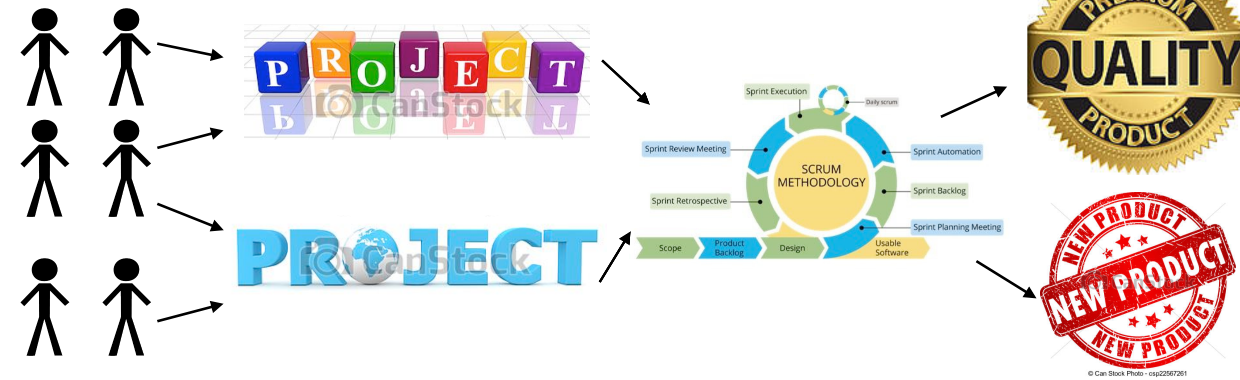
Traditional Senior Design



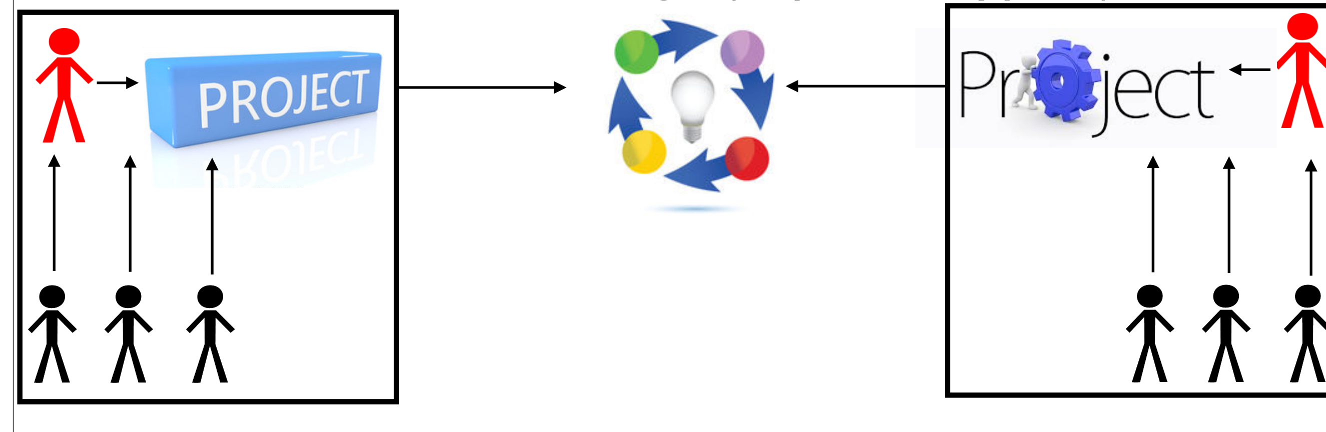
Research-based Senior Design (Expected Typical)

-Students select faculty sponsors and corresponding projects
-The same faculty sponsor may sponsor multiple projects and multiple students, but students will only work on one project

Traditional Senior Design

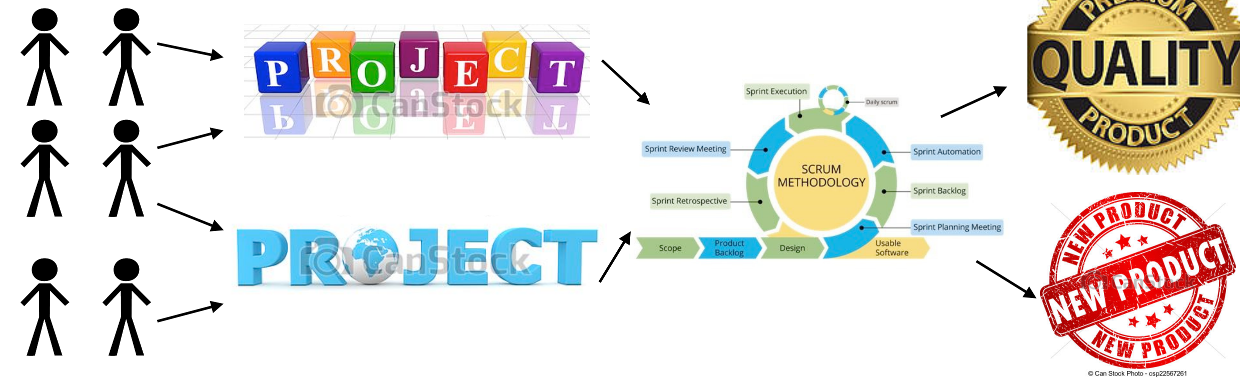


Research-based Senior Design (Expected Typical)

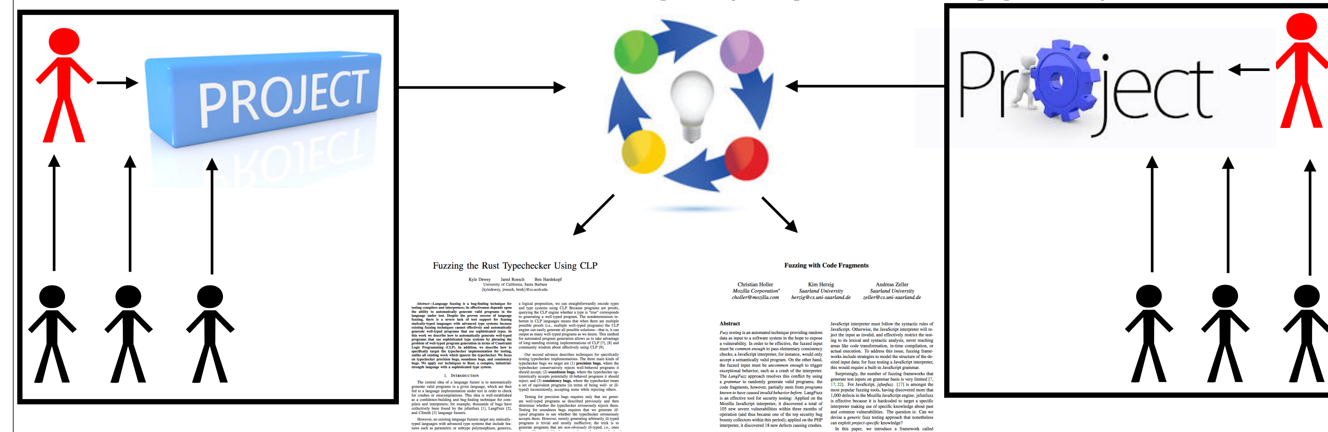


-Faculty sponsor/student groups will then work with an iterative process to progress on the research
-This iterative process is partially defined by the class itself, and partially defined by the faculty sponsor
-The class itself will focus on more mechanical, generic aspects of research (e.g., technical writing and presentation skills), whereas individual faculty sponsors will get into domain-specific things

Traditional Senior Design



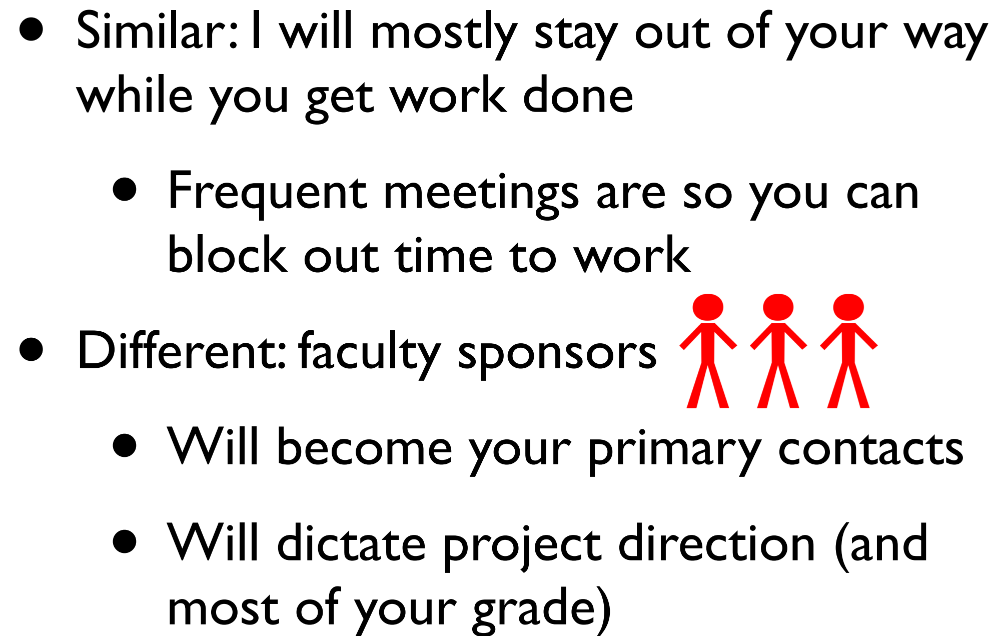
Research-based Senior Design (Expected Typical)



-End (somewhat stretch) goal: each project delivers a publishable unit at the end

-Exactly how publishable this unit will be largely depends on the projects themselves. Research is unpredictable by its very nature.

Comparing the Two

- Similar: I will mostly stay out of your way while you get work done
  - Frequent meetings are so you can block out time to work
- Different: faculty sponsors 
  - Will become your primary contacts
  - Will dictate project direction (and most of your grade)

Skills You Will Learn

- How to read papers
- How to maintain research notes
- How to write papers
- How to orally present research, especially to a general audience

Fair Warning: This is Hard

- Paper-reading instincts will probably fail you
  - Question: how would you read a paper?
- Tons of reading for even basic understanding
- Page **maximums** instead of minimums
  - English classes usually train bad habits

-I hand you a paper. How will you read it? You don't have to answer out loud.
-I read the first paper my advisor gave me over a dozen times over the course of a month
-I only understood about 10% of it at best at the time, and a full two years before I understood it about 70%
-For nearly two years, my adviser threw out everything I wrote as unsalvageably bad.

Syllabus

Building Up to Projects

- Making a well-informed decision about a project will require you to read papers
- Therefore, we need to go over how to best read papers before we can get into projects

Introduction to Reading Papers

Metaphor: an image coming into view

Inefficient Reading

Very carefully read from start to finish.

Inefficient Reading

Very carefully read from start to finish.

Inefficient Reading

Very carefully read from start to finish.

Inefficient Reading

Very carefully read from start to finish.

Inefficient Reading

Very carefully read from start to finish.

Inefficient Reading

Very carefully read from start to finish.

Inefficient Reading

Very carefully read from start to finish.

Inefficient Reading

Very carefully read from start to finish.

Inefficient?

- Problem: no idea what the big picture is
  - Will not get an idea until you're done
- Rarely will you need to know every detail, but this guarantees you'll learn them all
  - This is wasted time

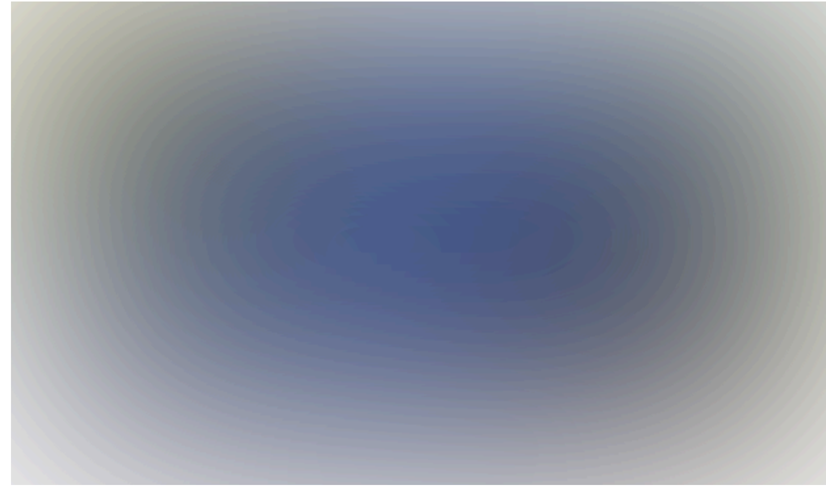
Efficient Reading

Multiple, shallow passes.

Efficient Reading

Multiple, shallow passes.

First pass

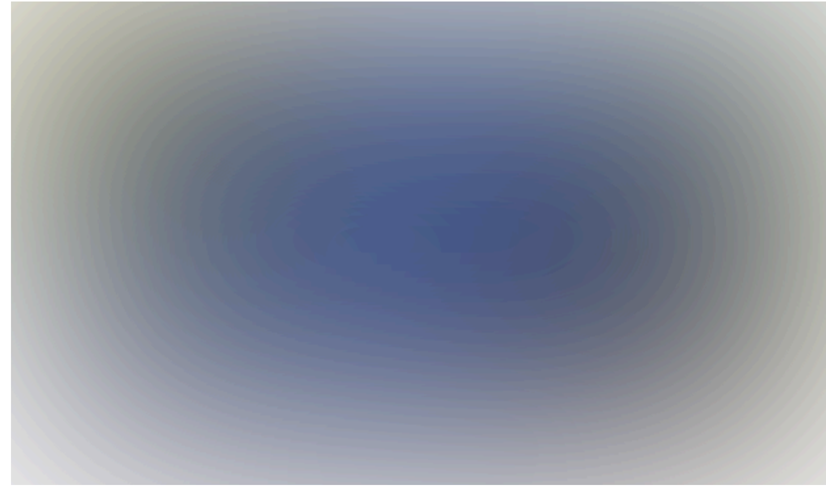


-Your first pass gives you something like this
-You can't tell what it is, but there is definitely a lot of blue.

Efficient Reading

Multiple, shallow passes.

First pass



Previous inefficient

-If you're doing this first pass right, you've spent less time than with the best of the previous method
-You are _already seeing a difference_ between the two. In fact, the first was misleading - there is no blue in it!

Efficient Reading

Multiple, shallow passes.

Second pass

-Not everything is blue, but most of it is

Efficient Reading

Multiple, shallow passes.

Third pass

-You can probably see enough details to make out that this is a car
-Depending on the reasons why you're reading the paper, this might be enough!

Efficient Reading

Multiple, shallow passes.

Fourth pass

-You can do more passes here, and each time it gets clearer
-Whether or not more passes is done all depends on what you need

Efficient Reading

Multiple, shallow passes.

Fifth pass

–Diminishing returns starts becoming apparent

Efficient Reading

Multiple, shallow passes.

Sixth pass

-Diminishing returns starts becoming apparent

Efficient Reading

Multiple, shallow passes.

Seventh pass

Reading Papers

- First question: do I have to read this paper?
- Generally good reading order: title, abstract, conclusions, figures with captions
- Then **skim** the paper
  - Get a general sense of what is going on
  - May need to repeat this
- Then in-depth reading

-With first question, the answer is often no. Usually you're looking for gems in a sea of information, so you're trying to get a "no" answer as quickly as possible.

Assignment: First Paper

- Read "The Structure of the 'THE'-Multiprogramming System", by Edsger W. Dijkstra
- **Take notes, and write a one-paragraph summary of the paper**
- We will discuss this in class on Wednesday