Practical Prototyping for Programming Tools

Andrew Head, Postdoctoral scholar, UC Berkeley

Happy 2069th birthday, Lucan!

Happy 2069th birthday, Lucan!

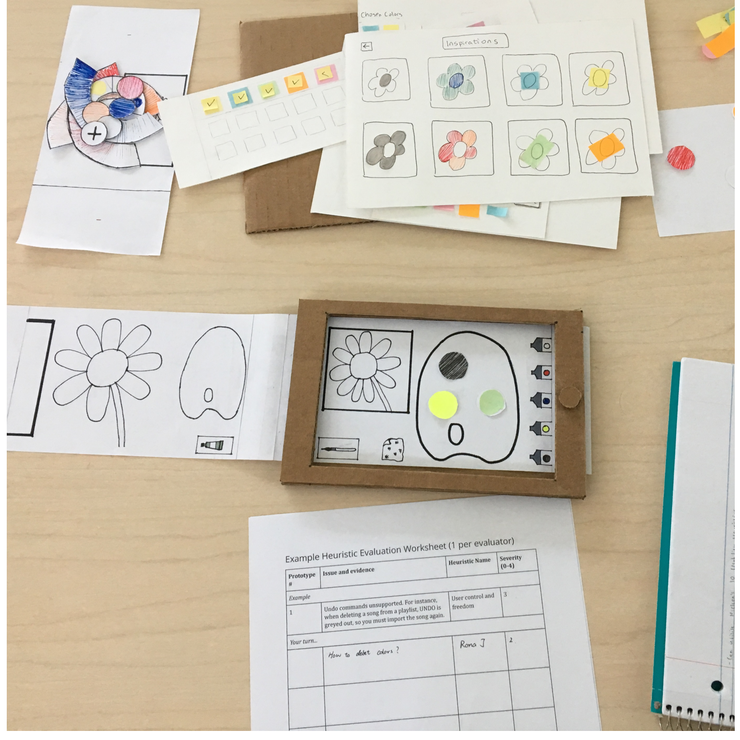
Happy 2069th birthday, Lucan!

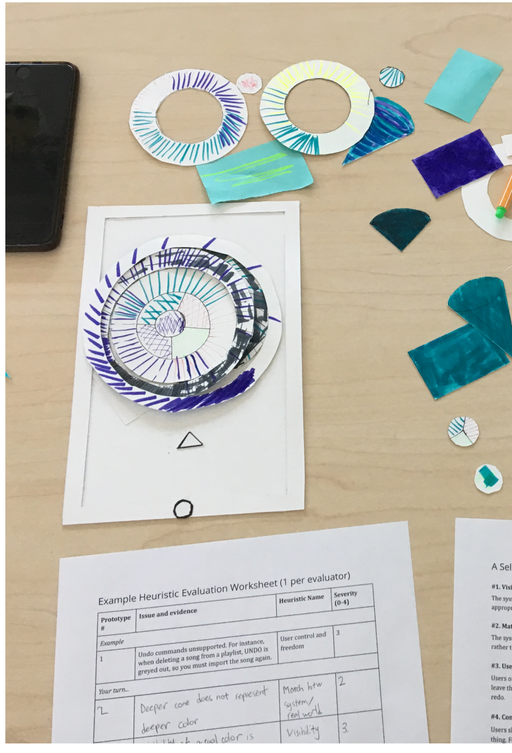
"Let the mind of people be blind to design problems; they fear, but leave them hope."

Objectives

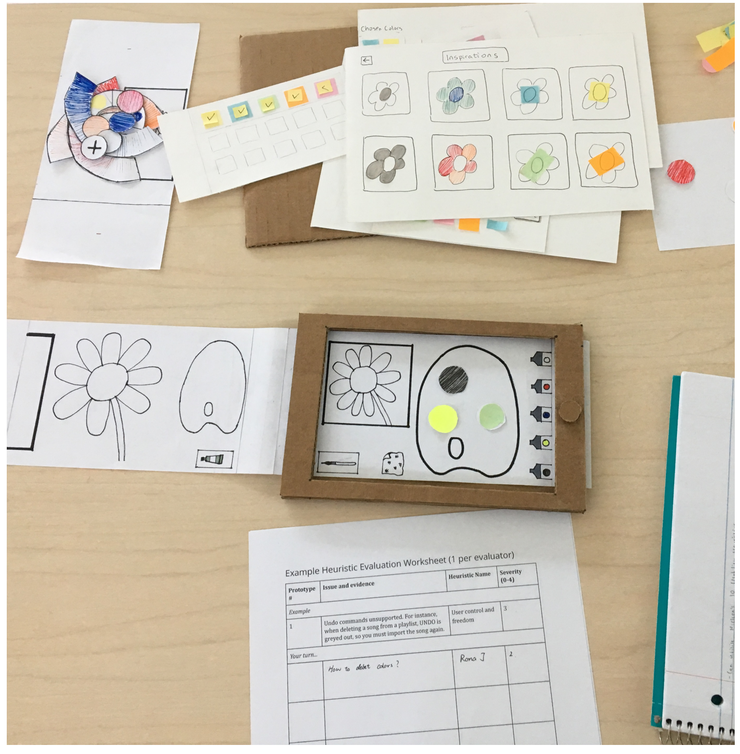
- What prototypes should I make to help me find a good design?
- How should I collect feedback to improve my design?

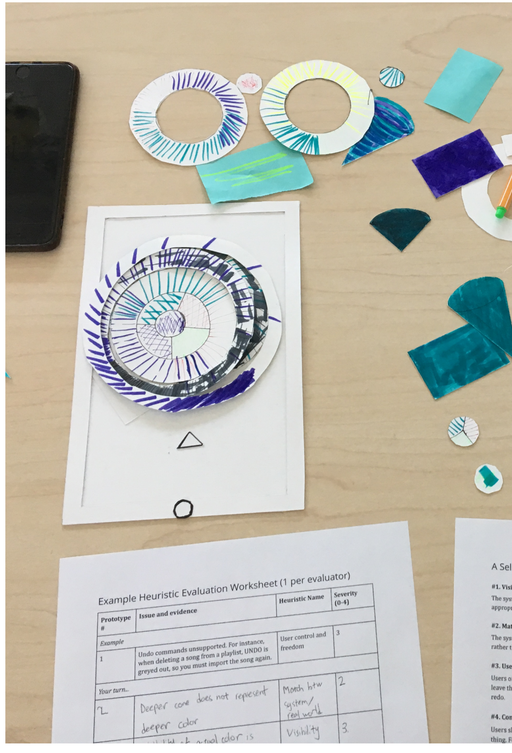

Who is this guy?


Example Heuristic Evaluation Worksheet (1 per evaluator)


Example Heuristic Evaluation Worksheet (1 per evaluator)


Who is this guy?


Example Heuristic Evaluation Worksheet (1 per evaluator)


Example Heuristic Evaluation Worksheet (1 per evaluator)

Who is this guy?

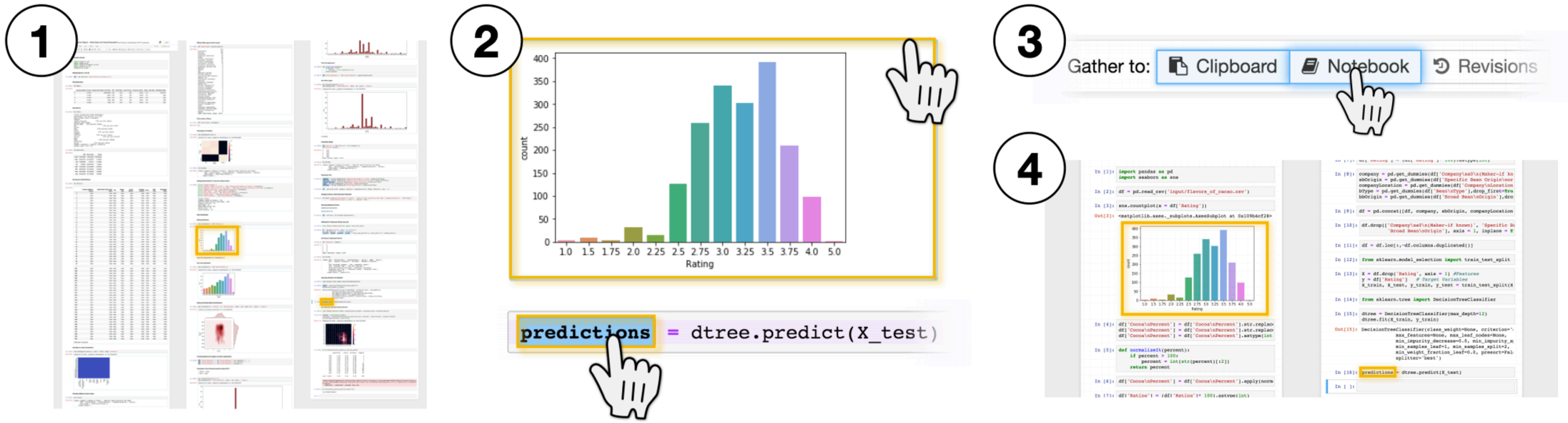


Figure 3: Cleaning a notebook with code gathering tools. Over the course of a long analysis, a notebook will become cluttered and inconsistent (1). With code gathering tools, an analyst can select results (e.g., charts, tables, variable definitions, and any other code output) (2) and click "Gather to Notebook" (3) to obtain a minimal, complete, ordered slice that replicates the selected results (4).

Managing Messes in Computational Notebooks, CHI '18

Who is this guy?

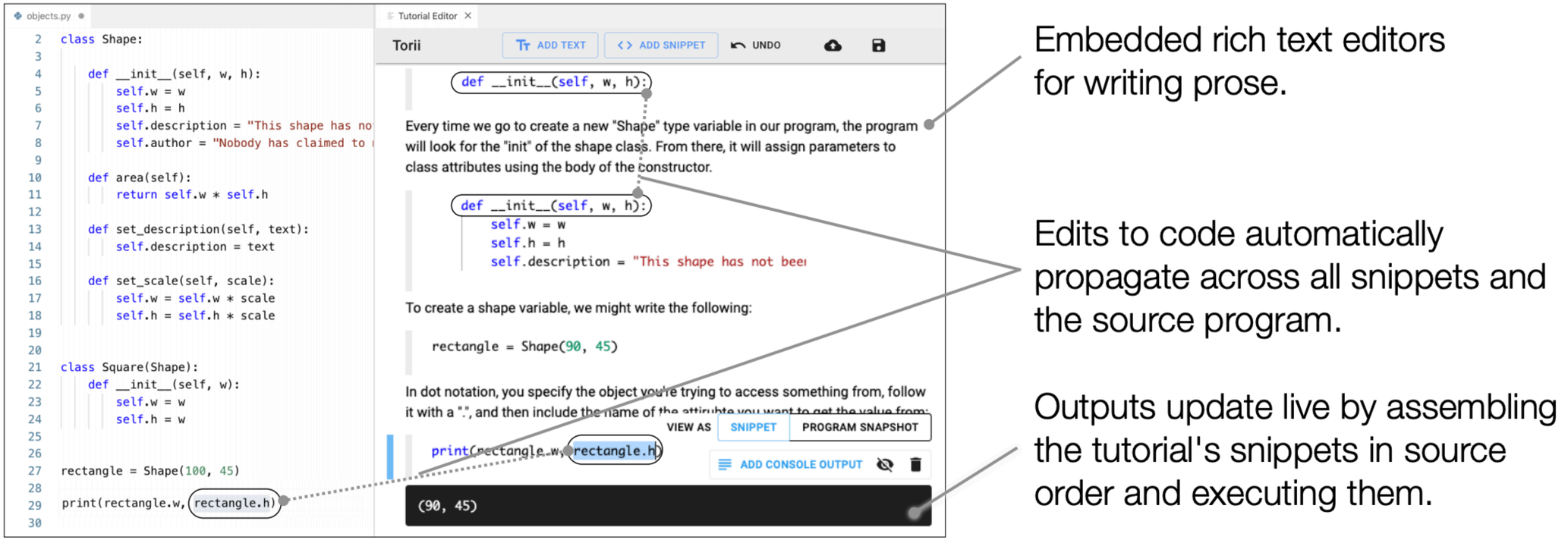


Figure 4. Writing tutorials with Torii. Torii helps authors write tutorials by keeping source programs, snippets, and outputs consistent with each other, while still letting authors organize the code in the tutorial flexibly. An edit to code anywhere in the tutorial workspace automatically triggers an update to clones of that code in the source program and snippets, and to all outputs generated from that code.

Composing Flexibly-Organized Step-by-Step Tutorials from Linked Source Code, Snippets, and Output, CHI '18

Who is this guy?

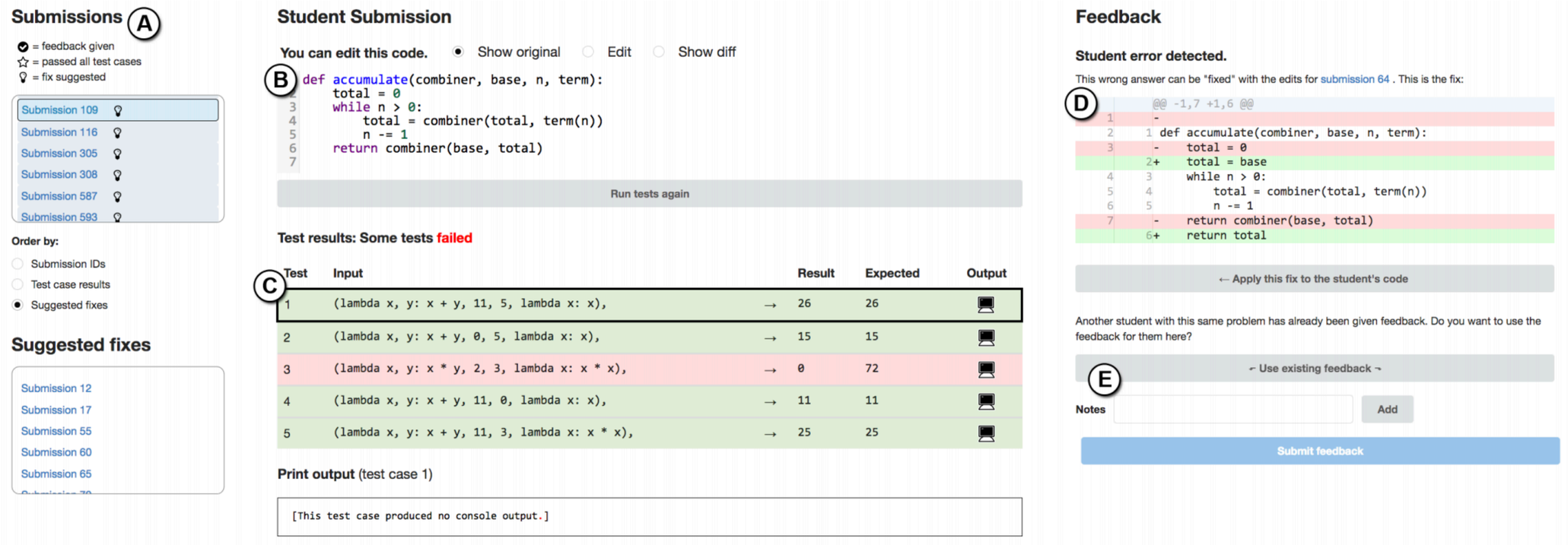


Figure 4. FIXPROPAGATOR interface: The left panel shows all of the incorrect submissions (A). When the teacher selects one, the submission is loaded into the Python code editor in the center of the interface (B). Then the teacher can edit the code, re-run tests, and inspect results. The bottom of the center panel shows the list of tests and console output (C). Once the teacher has fixed the submission, they add some hint that will be shown to current and future students fixed by the same transformation. The bottom of the left panel shows submissions for which the system is suggesting a fix. When the teacher selects a suggested fix, it is shown as a diff in the right panel (D). The teacher can reuse the previously written hint or create a new one (E).

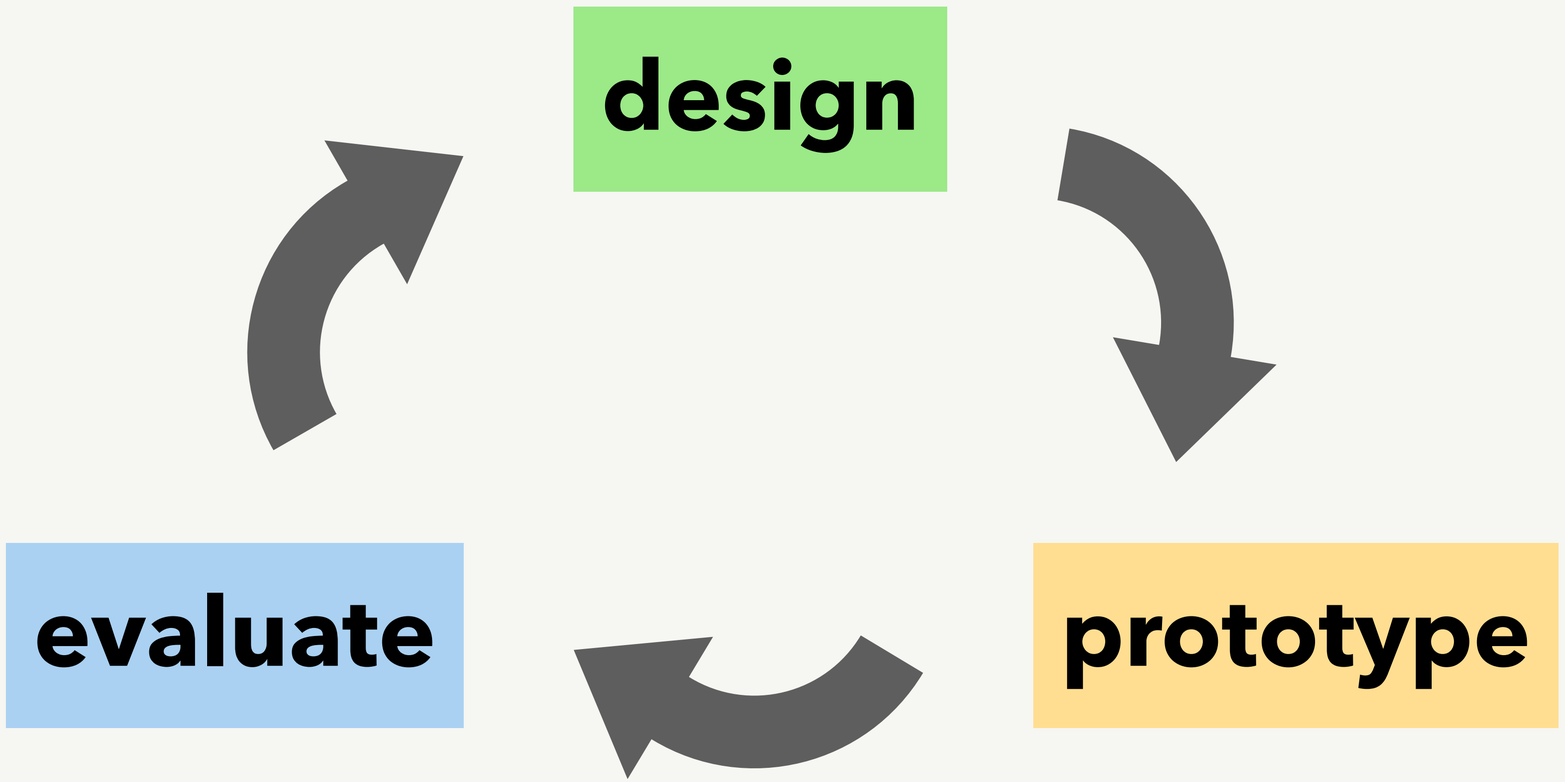
Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis, CHI '18

Design methods

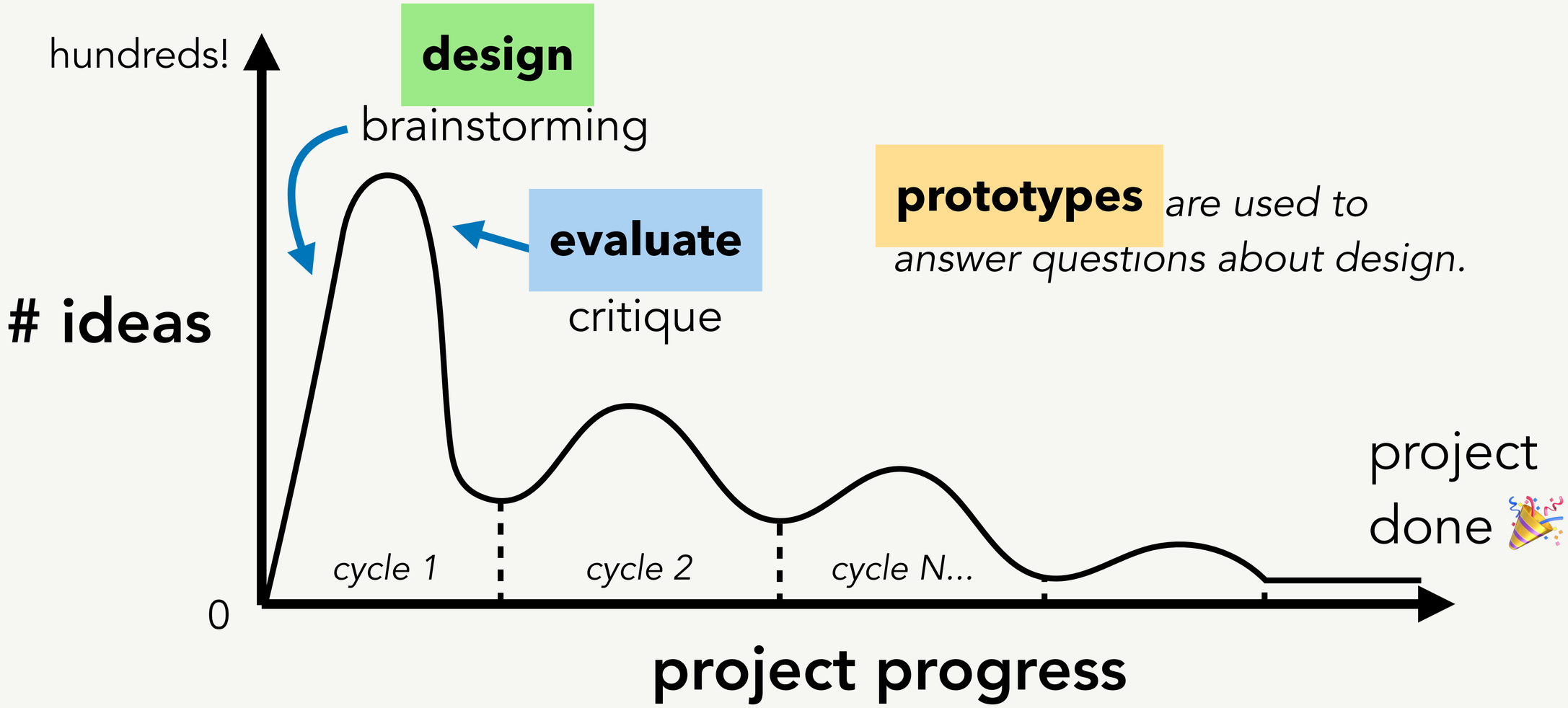
Design methods

Design methods for programming tools

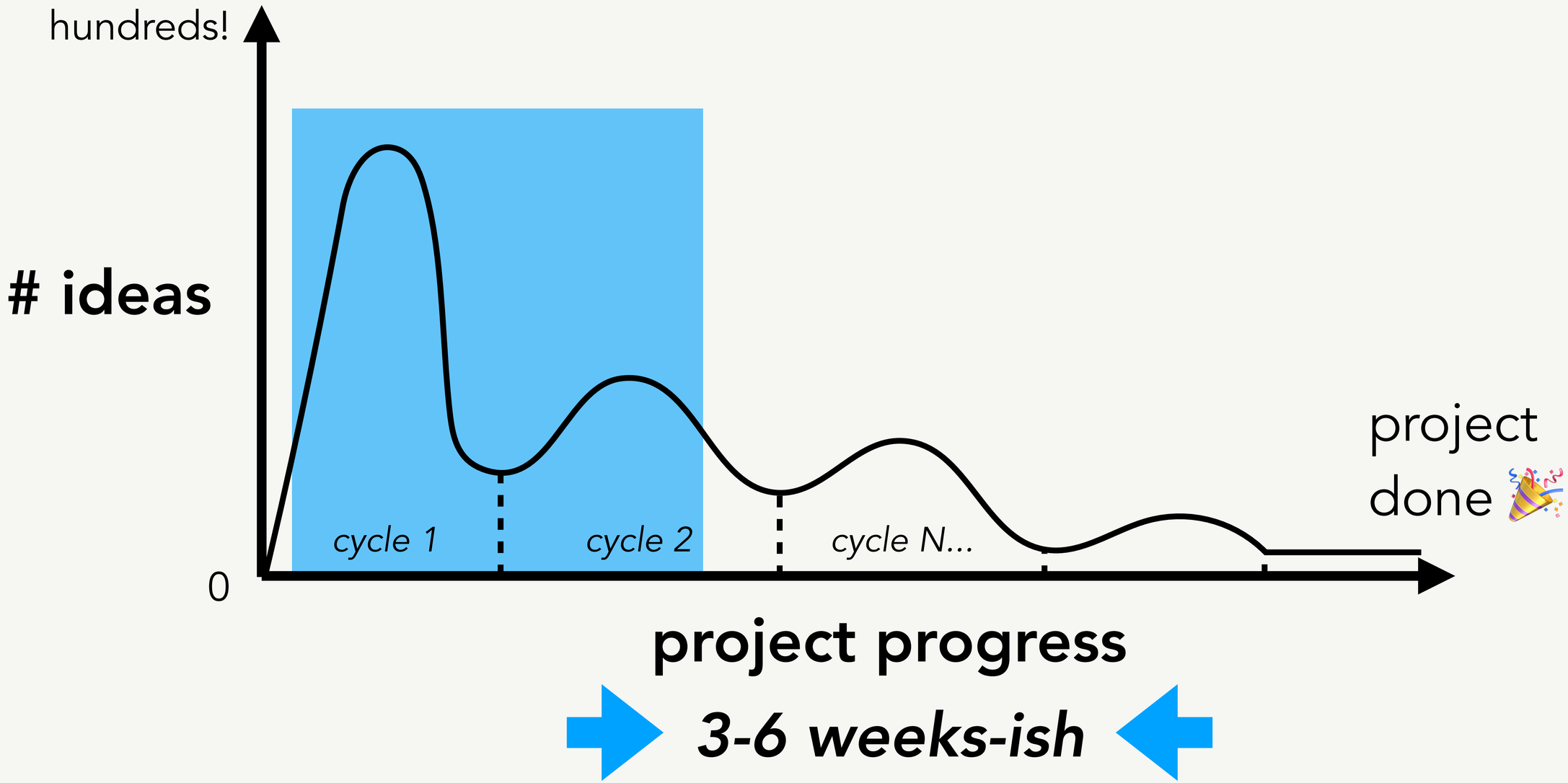
THE DESIGN CYCLE



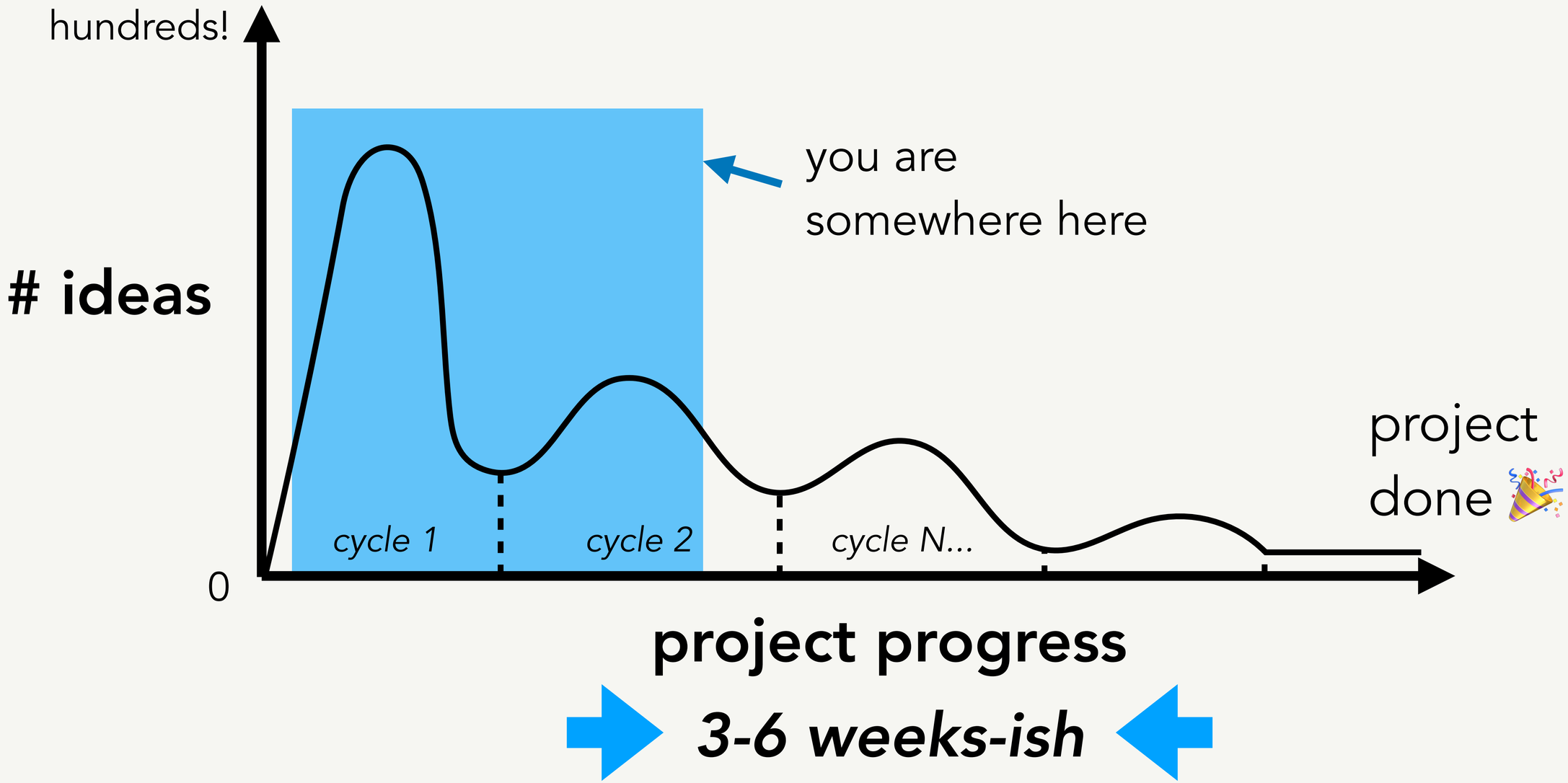
DESIGN IDEAS DIVERGE AND CONVERGE



DESIGN IDEAS DIVERGE AND CONVERGE

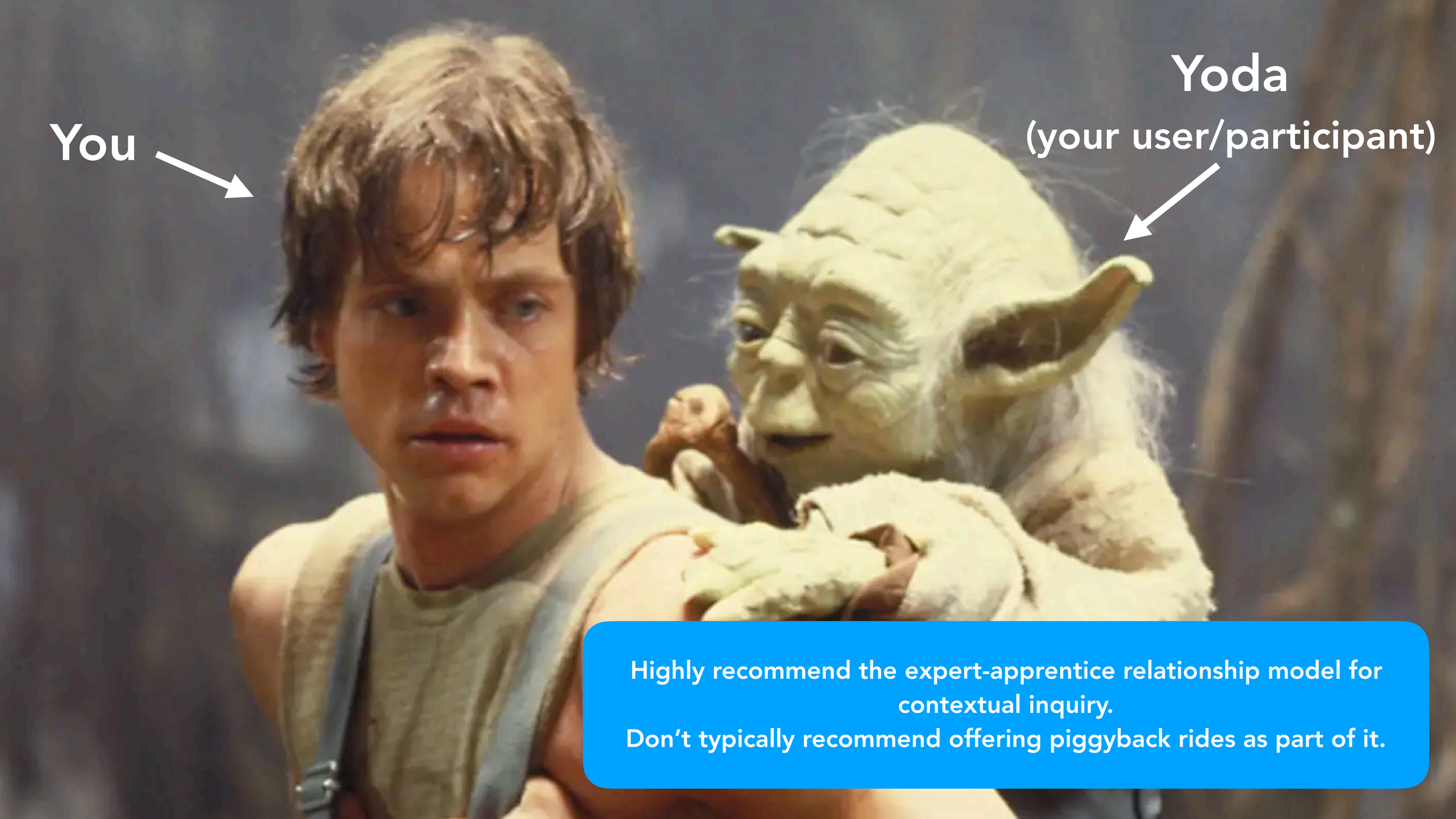


DESIGN IDEAS DIVERGE AND CONVERGE



Objectives

- What prototypes should I make to help me find a good design?
- How should I collect feedback to improve my design?


You
Yoda
(your user/participant)
Highly recommend the expert-apprentice relationship model for contextual inquiry.
Don’t typically recommend offering piggyback rides as part of it.

Don't look at me!

Discussion time

Think of an idea you had for a programming sometime in the past that you were *really* excited to work on.

What convincing evidence did you have that it was a good idea?

Don't look at me!

Discussion time

Think of an idea you had for a programming sometime in the past that you were *really* excited to work on.

What convincing evidence did you have that it was a good idea?

Brainstorming

1. Defer judgement
2. Encourage wild ideas
3. Build on the ideas of others
4. Stay focused on the topic
5. One conversation at a time
6. Be visual
7. Go for quantity

How do you know these ideas are any good?

From *IDEO Design Kit: Brainstorm Rules*

PRAGMATIC PROTOTYPING

FIDELITY

LOW FIDELITY

Many details missing.

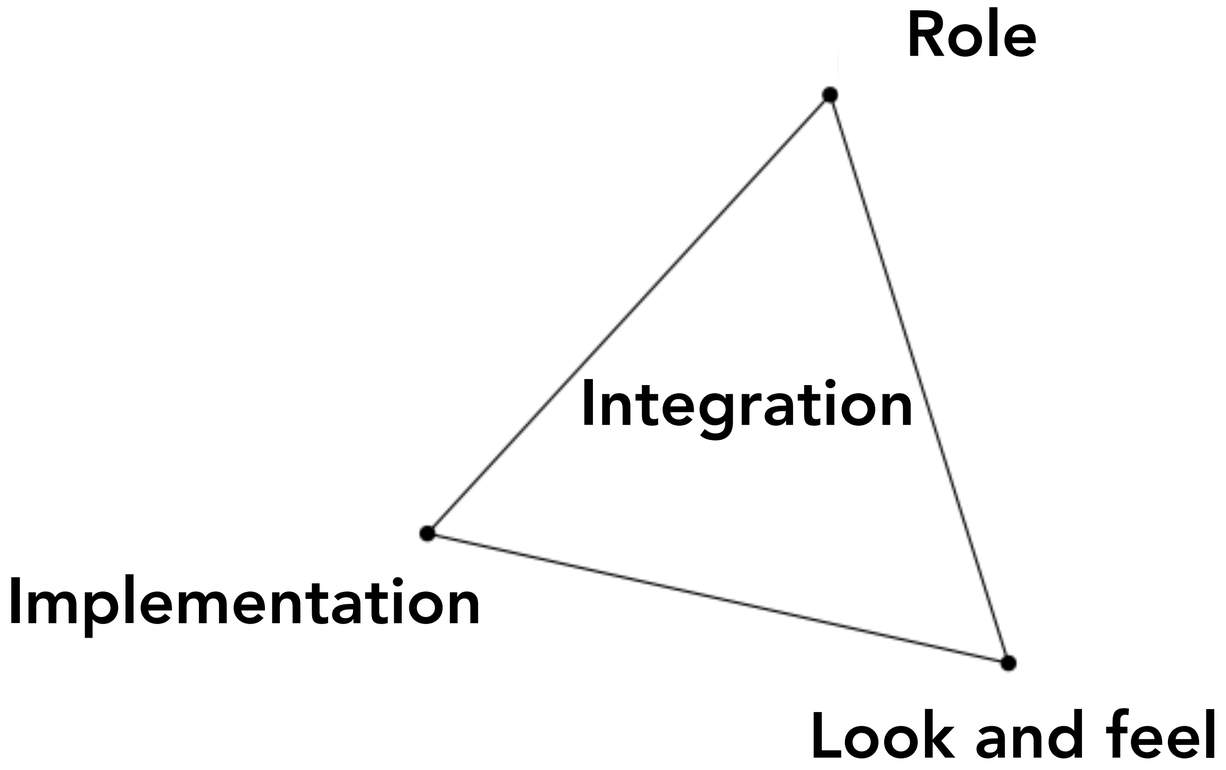
HIGH FIDELITY

Looks like final product.

#1 RULE OF PROTOTYPING

Make prototypes with a well-defined **purpose** and **scope**. Adjust the **fidelity** of your prototype to match the purpose and scope.

SCOPE: WHAT DOES YOUR PROTOTYPE PROTOTYPE?



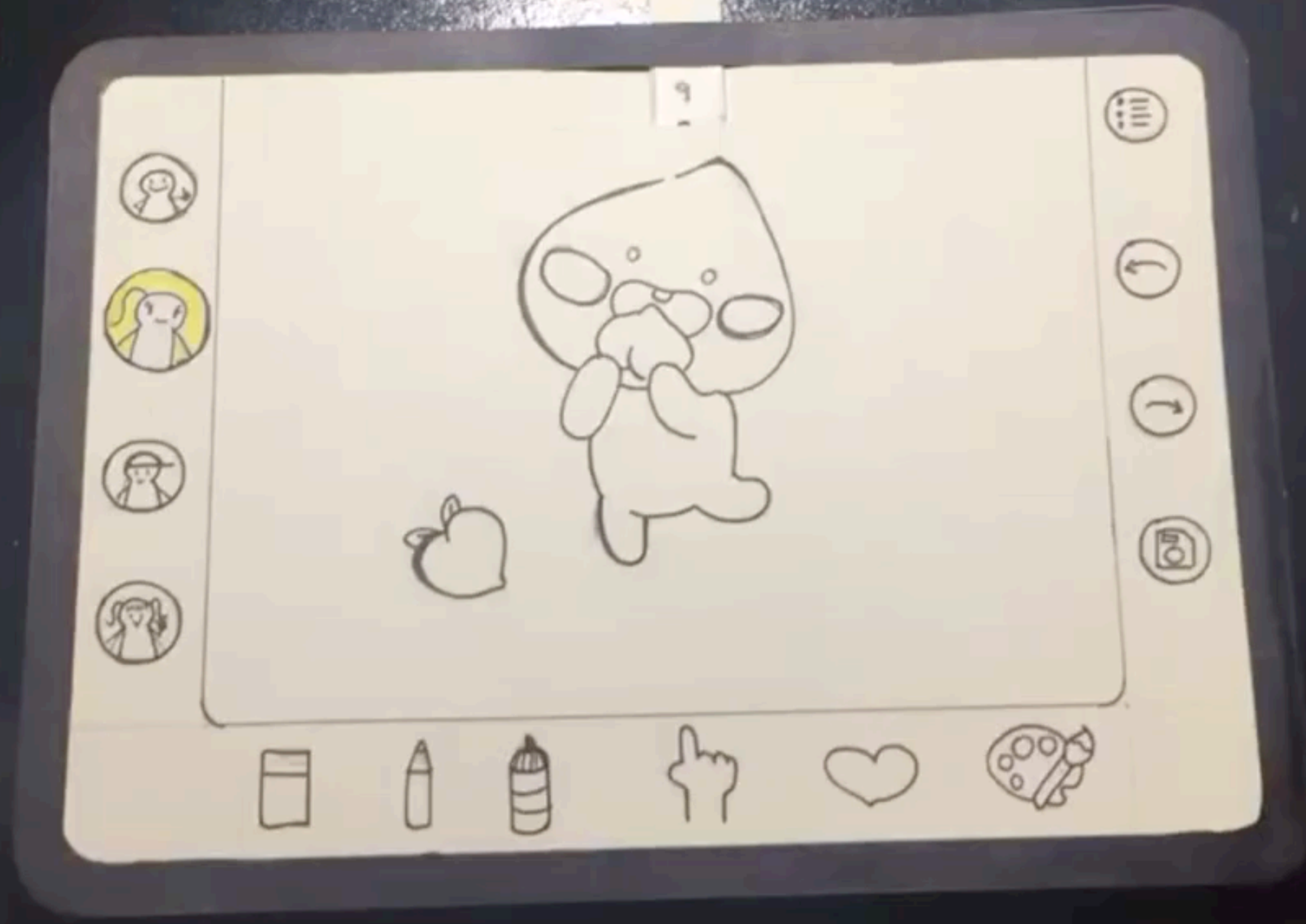
Role: function, fit

Look and feel: appearance, sensory experience

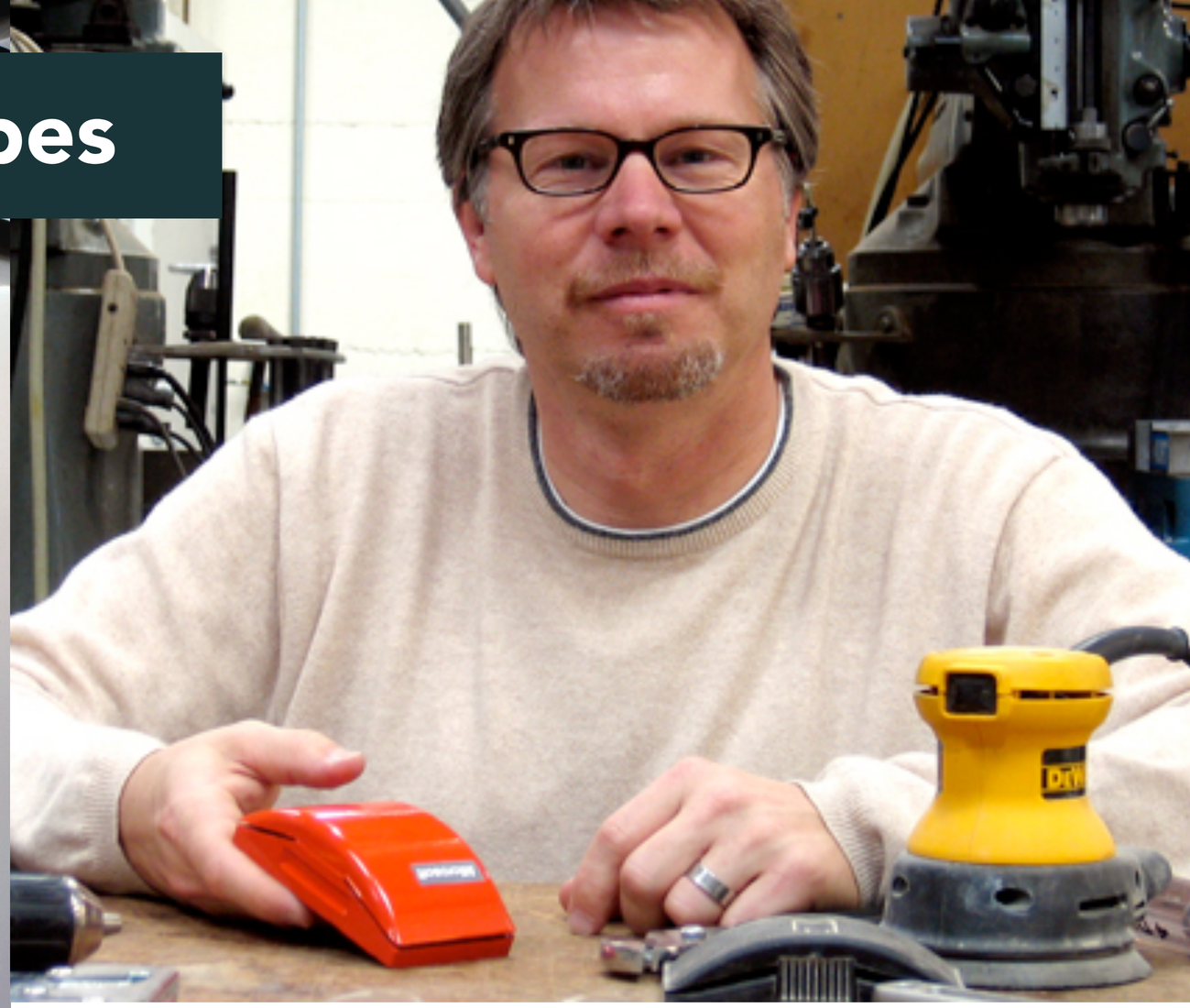
Implementation: algorithms, engineering, code

From Houde and Hill – *What do Prototypes Prototype?*

Role Prototypes

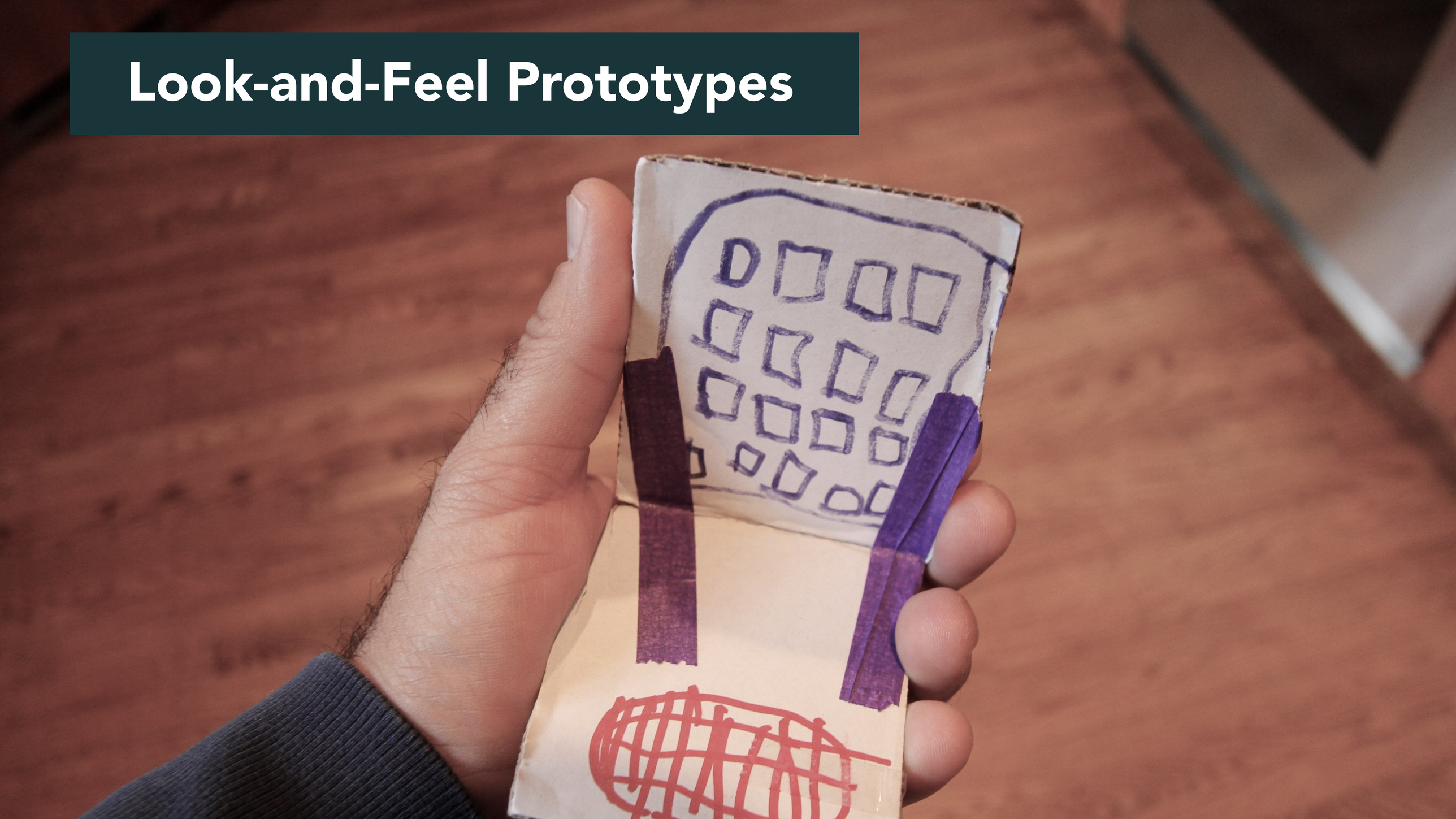


Look-and-Feel Prototypes

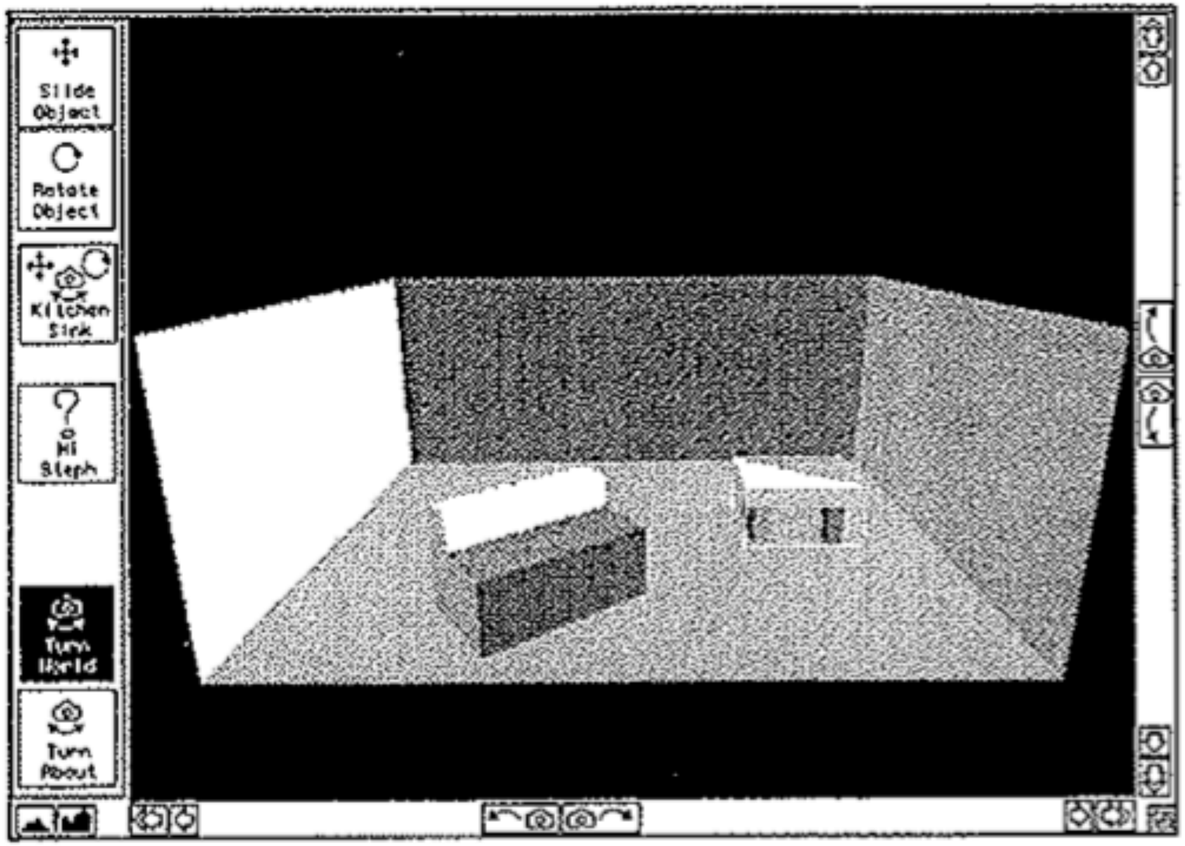


Prototypes for the Microsoft mouse

Look-and-Feel Prototypes



Implementation Prototypes



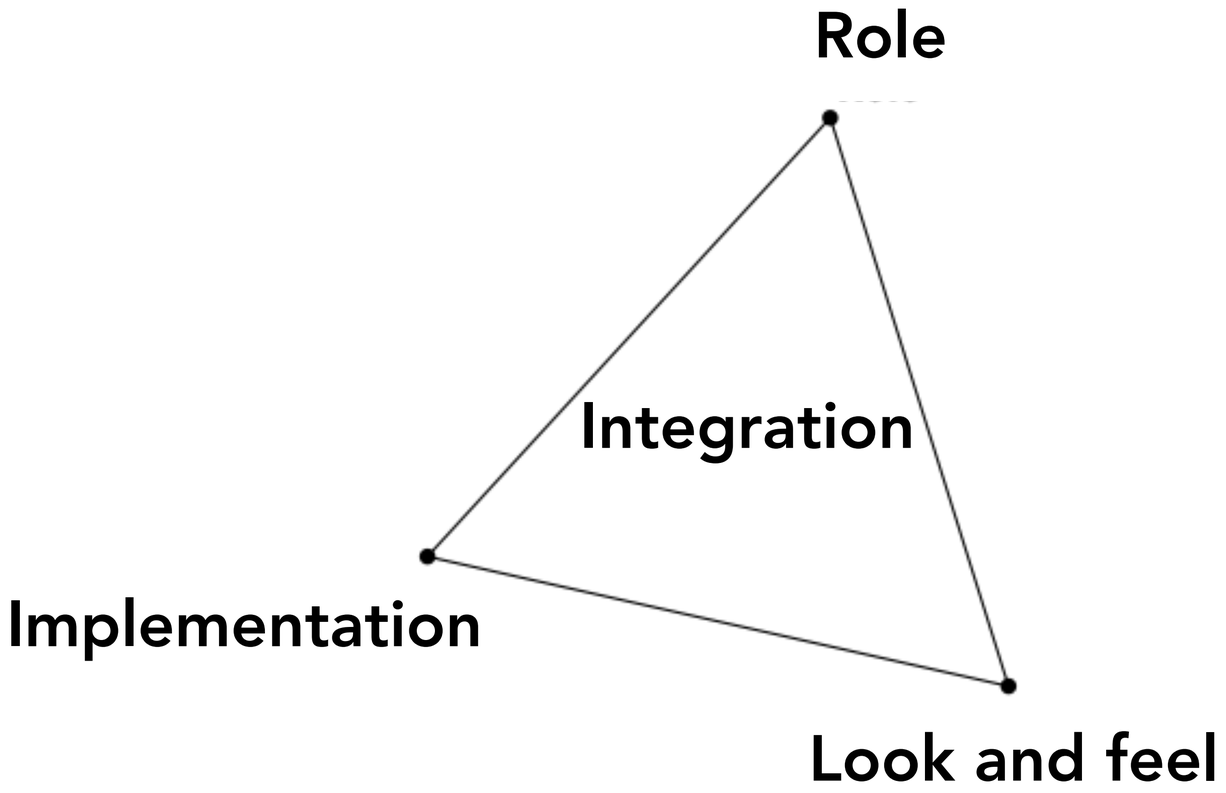
Example 3. Implementation prototypes for 3D space-planning application [E3: Chen 1990].

Implementation Prototypes

```
IntList& IntList::operator=(const IntList& oldList)
{
register long n = oldList.size;
  if (n != size) setSize(n);
  register int* newPtr = &values[n];
  register int* oldPtr = &oldList.values[n];
  while (n--) *--newPtr = *--oldPtr;
  return *this;
}
```

Example 12. C++ program sample from a fluid dynamics simulation system [E12: Hill, 1993].

SCOPE: WHAT DOES YOUR PROTOTYPE PROTOTYPE?



Why are the types of prototypes corners of a triangle? What does this mean for scoping your prototypes?

From Houde and Hill – *What do Prototypes Prototype?*

Prototyping Programming Tools

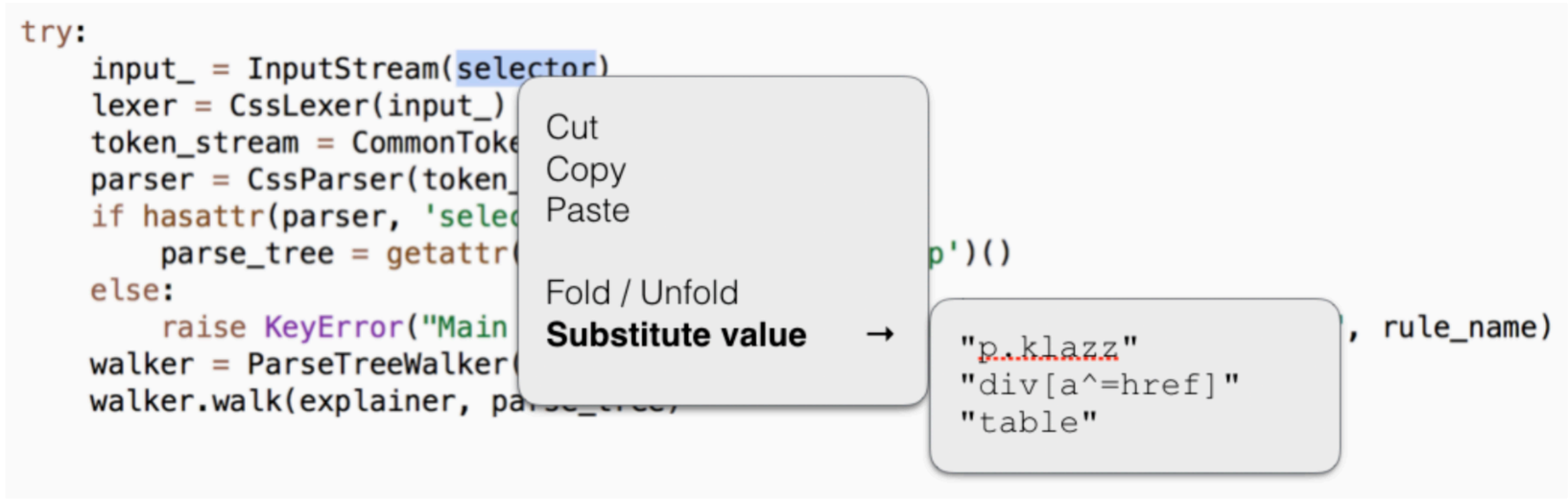
Why prototype?

- Full implementations take a long amount of time
- At least in research, development teams are only 1 or 2 people
- Solutions need to merge into workspaces that are already complex

Role Prototypes

Narrative scenarios

After expanding the code some more, it should let me substitute in realistic input values. *These could be captured from the runtime data of my program.* Or maybe they're inferred from typical values an API is called with, mined from open source code online.

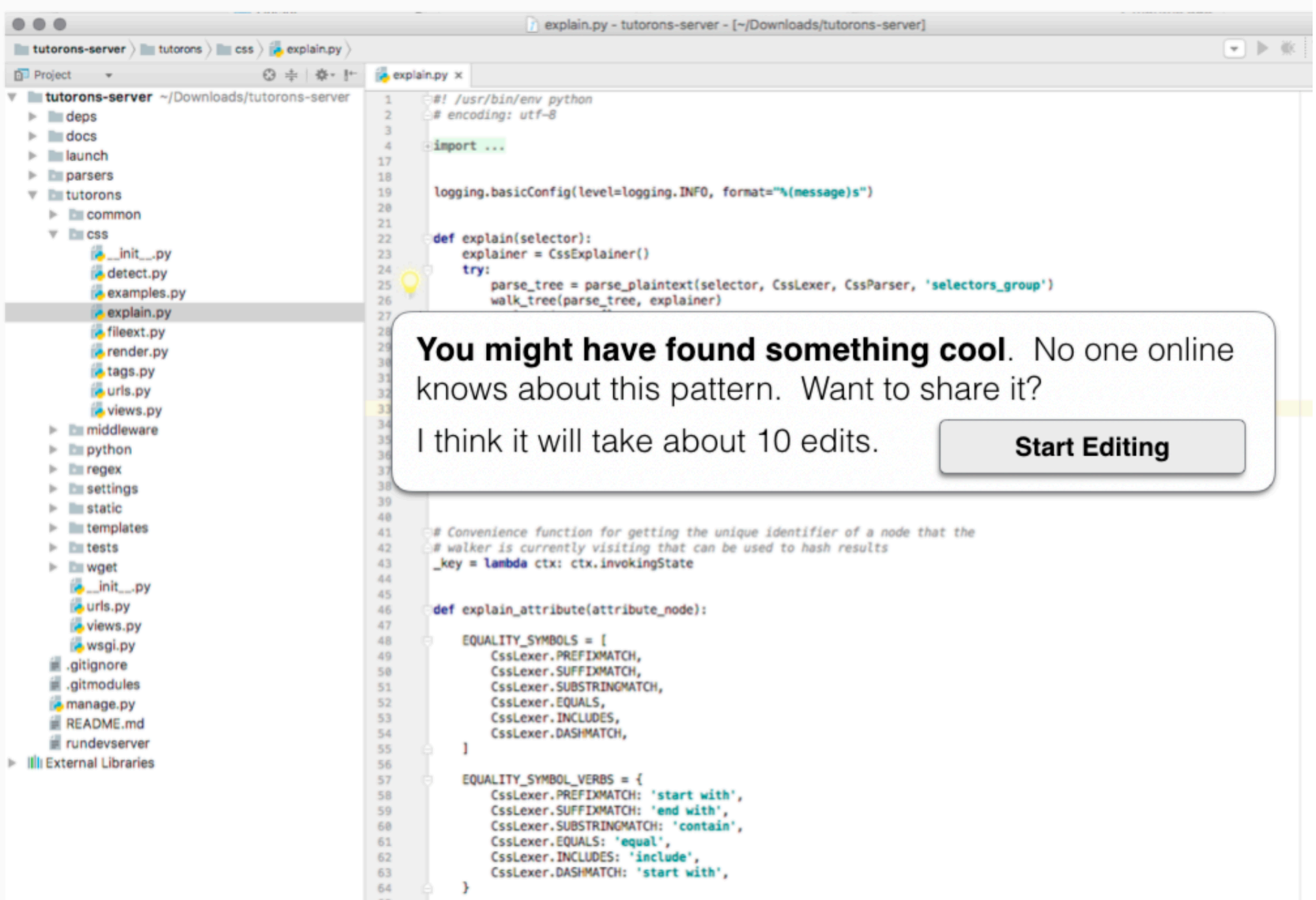


Now I've still got some try-catch blocks and if-else statements to remove. When I remove these, I want to make sure the code still runs fine. Others should be able to copy, paste, and run this code, without bugs I've accidentally introduced. So there should be an output pane like this:

```
try:
    input_ = InputStream("p.klazz")
    lexer = CssLexer(input_)
    token_stream = CommonTokenStream(lexer)
    parser = CssParser(token_stream)
    if hasattr(parser, 'selectors_group'):
        parse_tree = getattr(parser, 'selectors_group')()
    else:
```

Look-and-Feel Prototypes

IDE mockups



Implementation Prototypes

Assignment 7 - Program Slicing

Submission details: Please submit a .py file. Submit via GradeScope. If you have questions on this process, get in touch via the Slack or via email.
Due: 10/19/20

In class, we worked with a program that generates a control flow graph (CFG) for a limited subset of Python. For this assignment, transform that program into a program slicer.

Required: handle straight-line programs
Strongly encouraged: handle the if then statements we added during class
Extra super awesome: handle loops

Please support this usage:

```
python program_slicing.py filename line_number variable_name
```

FORMATIVE USER RESEARCH

So many methods!

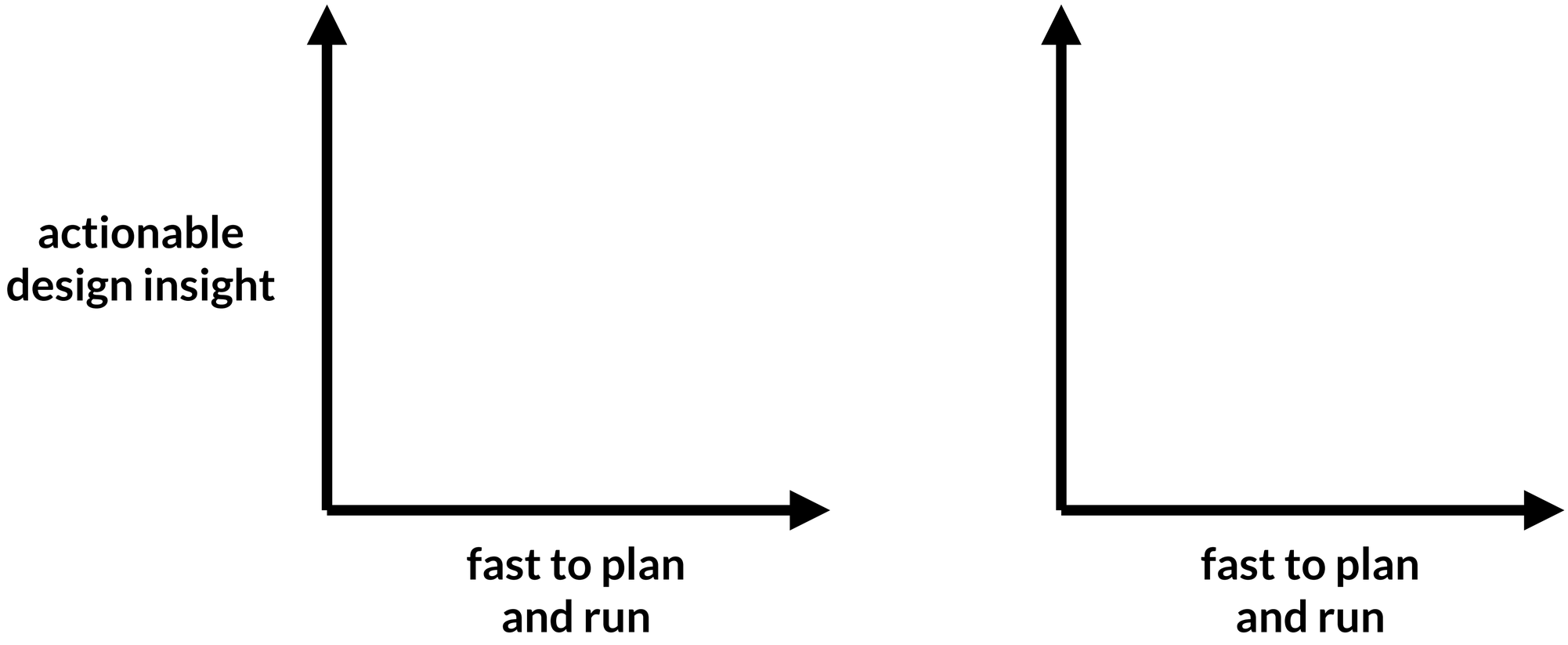
Method	Tool development activities supported	Key benefits
Contextual inquiry	Requirements and problem analysis	» Experimenters gain insight into day-to-day activities and challenges. » Experimenters gain high-quality data on the developer's intent.
Exploratory lab studies	Requirements and problem analysis	» Focusing on the activity of interest is easier. » Experimenters can compare participants doing the same tasks. » Experimenters gain data on the developer's intent.
Surveys	» Requirements and problem analysis » Evaluation and testing	» Surveys provide quantitative data. » There are many participants. » Surveys are (relatively) fast.
Data mining (including corpus studies and log analysis)	» Requirements and problem analysis » Evaluation and testing	» Data mining provides large quantities of data. » Experimenters can see patterns that emerge only with large corpuses.
Natural-programming elicitation	» Requirements and problem analysis » Design	Experimenters gain insight into developer expectations.
Rapid prototyping	Design	Experimenters can gather feedback at low cost before committing to high-cost development.
Heuristic evaluations	» Requirements and problem analysis » Design » Evaluation and testing	» Evaluations are fast. » They do not require participants.
Cognitive walkthroughs	» Design » Evaluation and testing	» Walkthroughs are fast. » They do not require participants.
Think-aloud usability evaluations	» Requirements and problem analysis » Design » Evaluation and testing	Evaluations reveal usability problems and the developer's intent.
A/B testing	Evaluation and testing	» Testing provides direct evidence that a new tool or technique benefits developers. » It provides objective numbers.

Myers, Ko, LaToza, and Yoon "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools." *Computer.*

When to use a design method

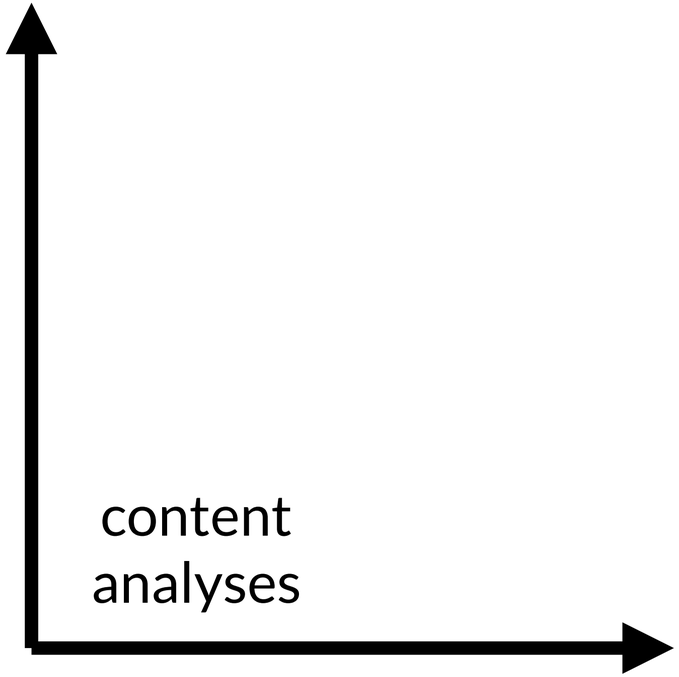
I need to understand the problem

I need to evaluate the solution

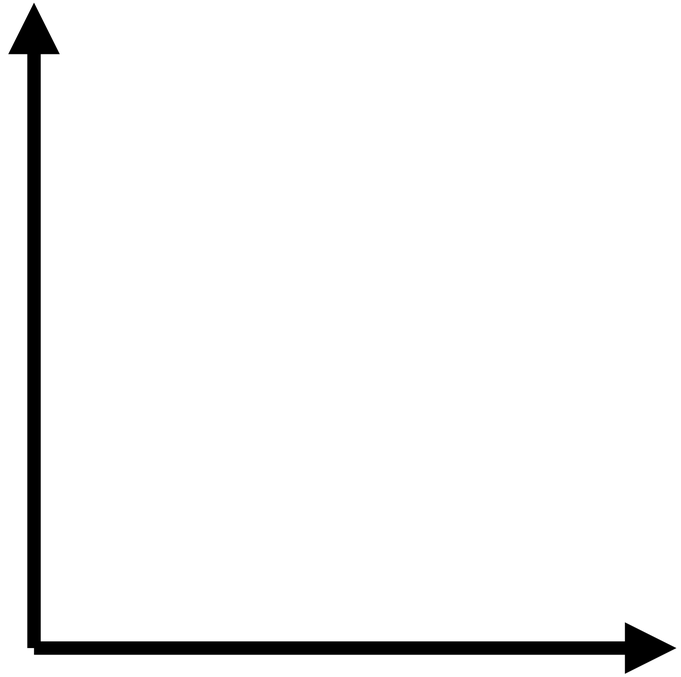


When to use a design method

I need to understand the problem



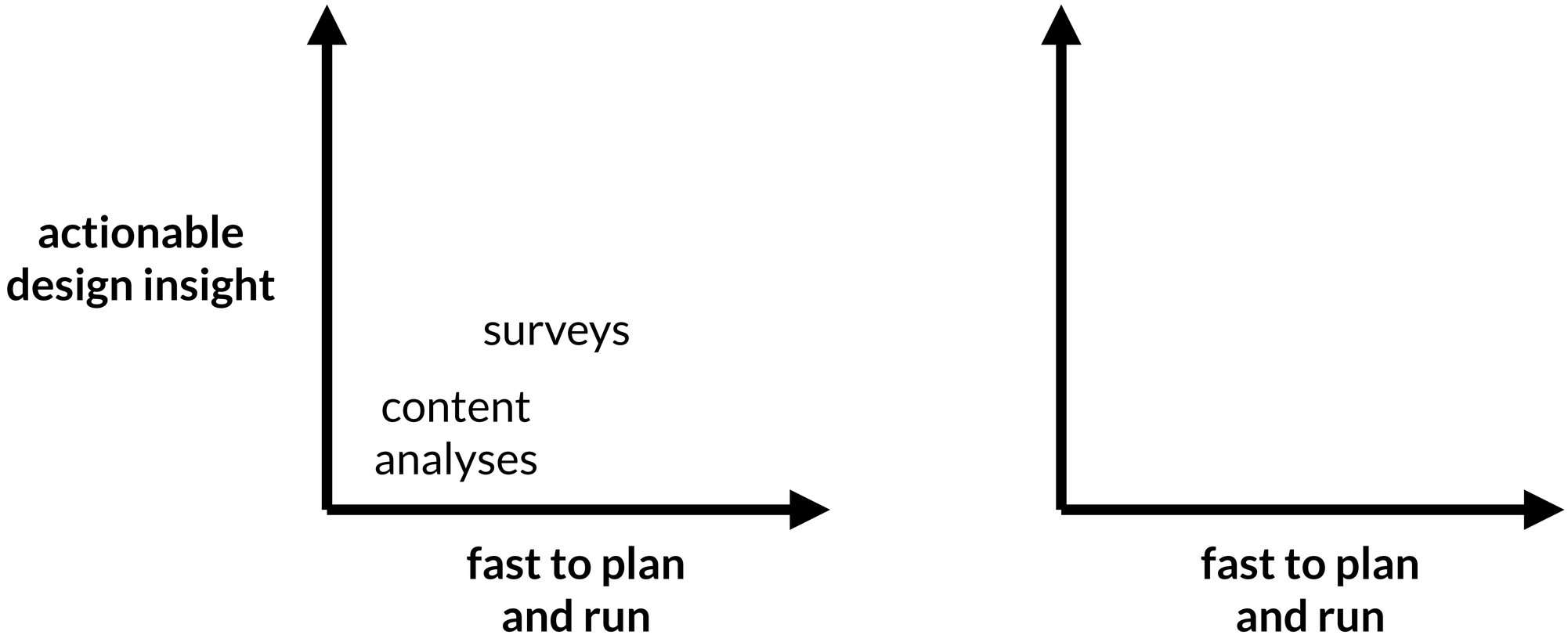
I need to evaluate the solution



When to use a design method

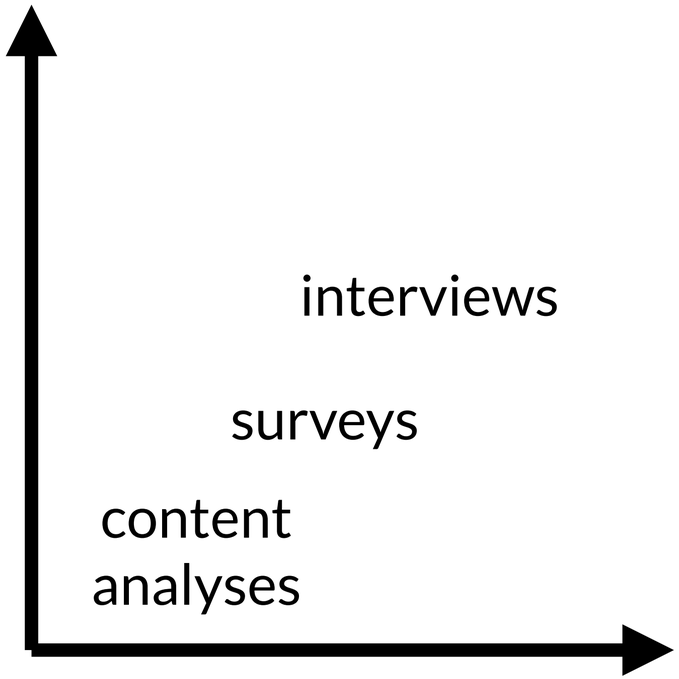
I need to understand the problem

I need to evaluate the solution

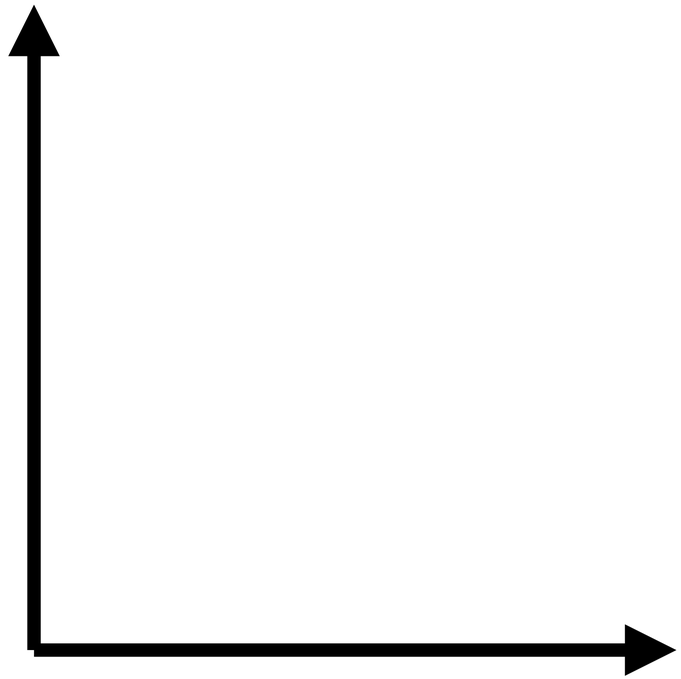


When to use a design method

I need to understand the problem



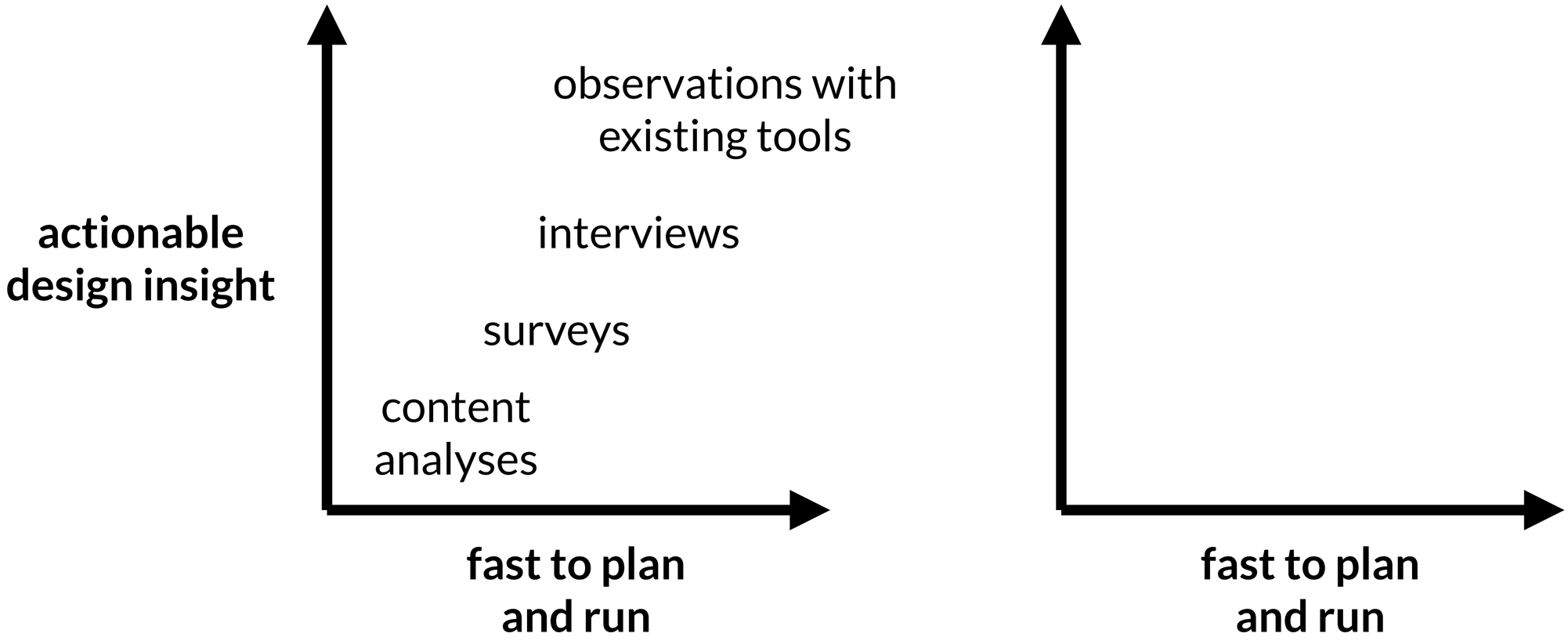
I need to evaluate the solution



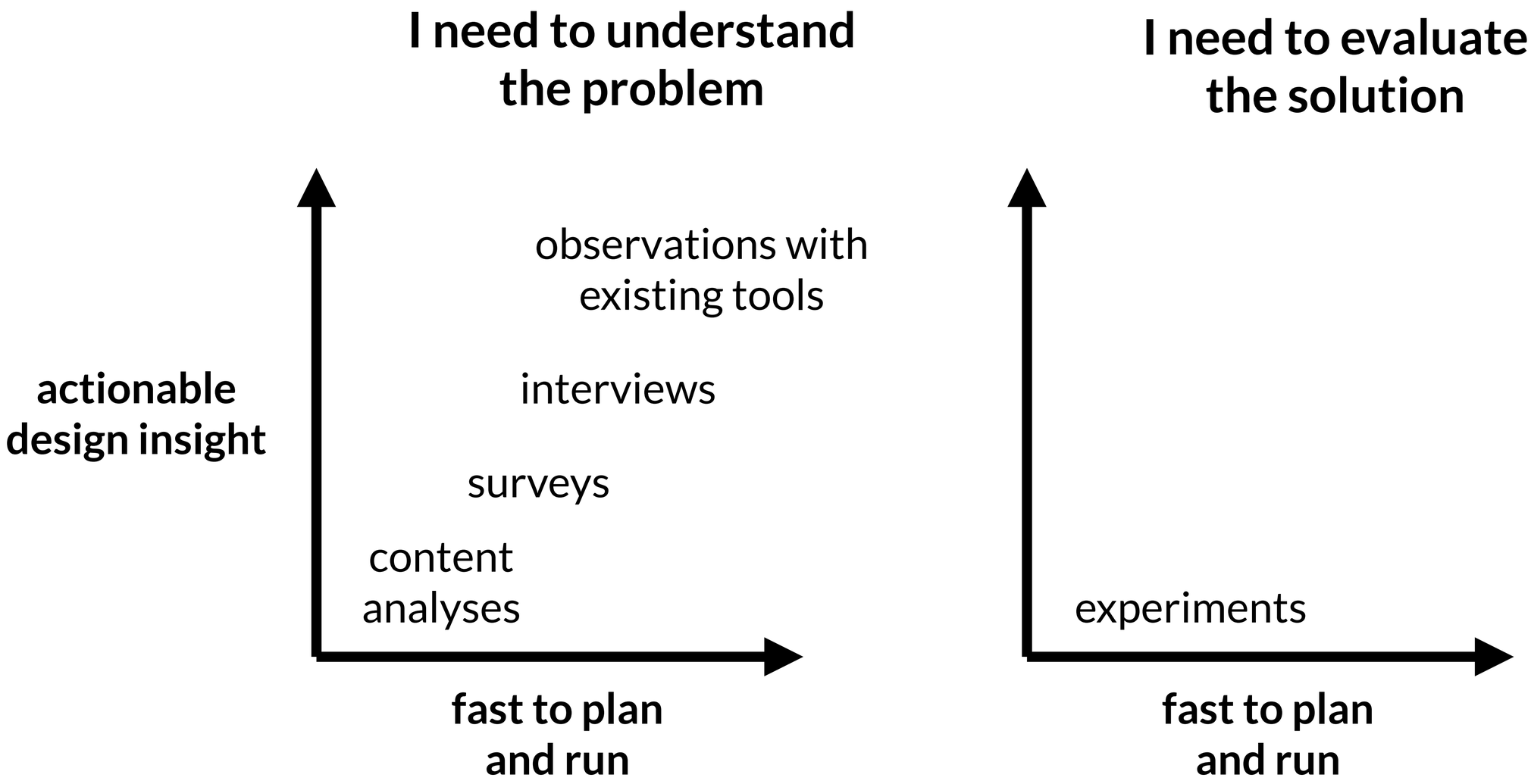
When to use a design method

I need to understand the problem

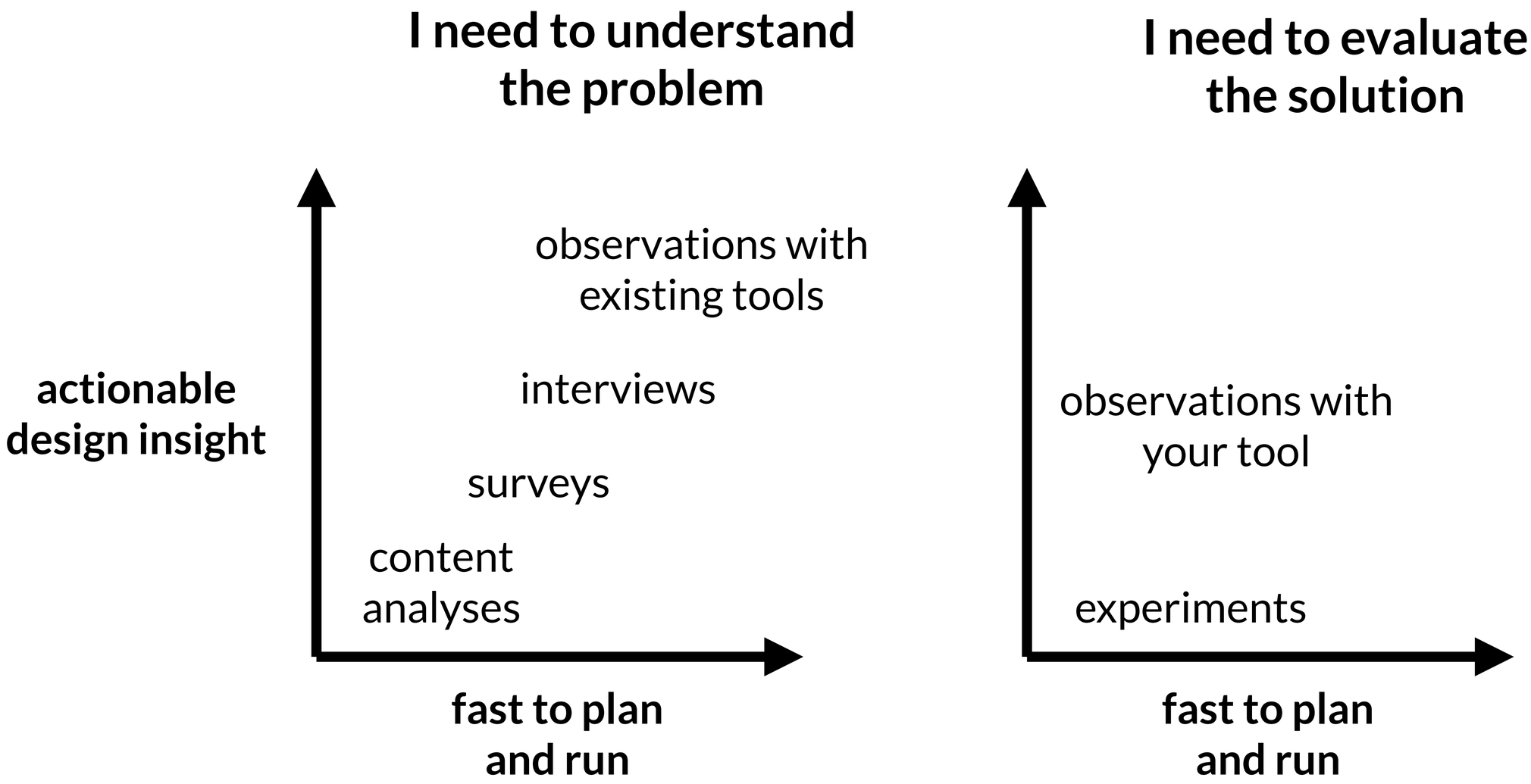
I need to evaluate the solution



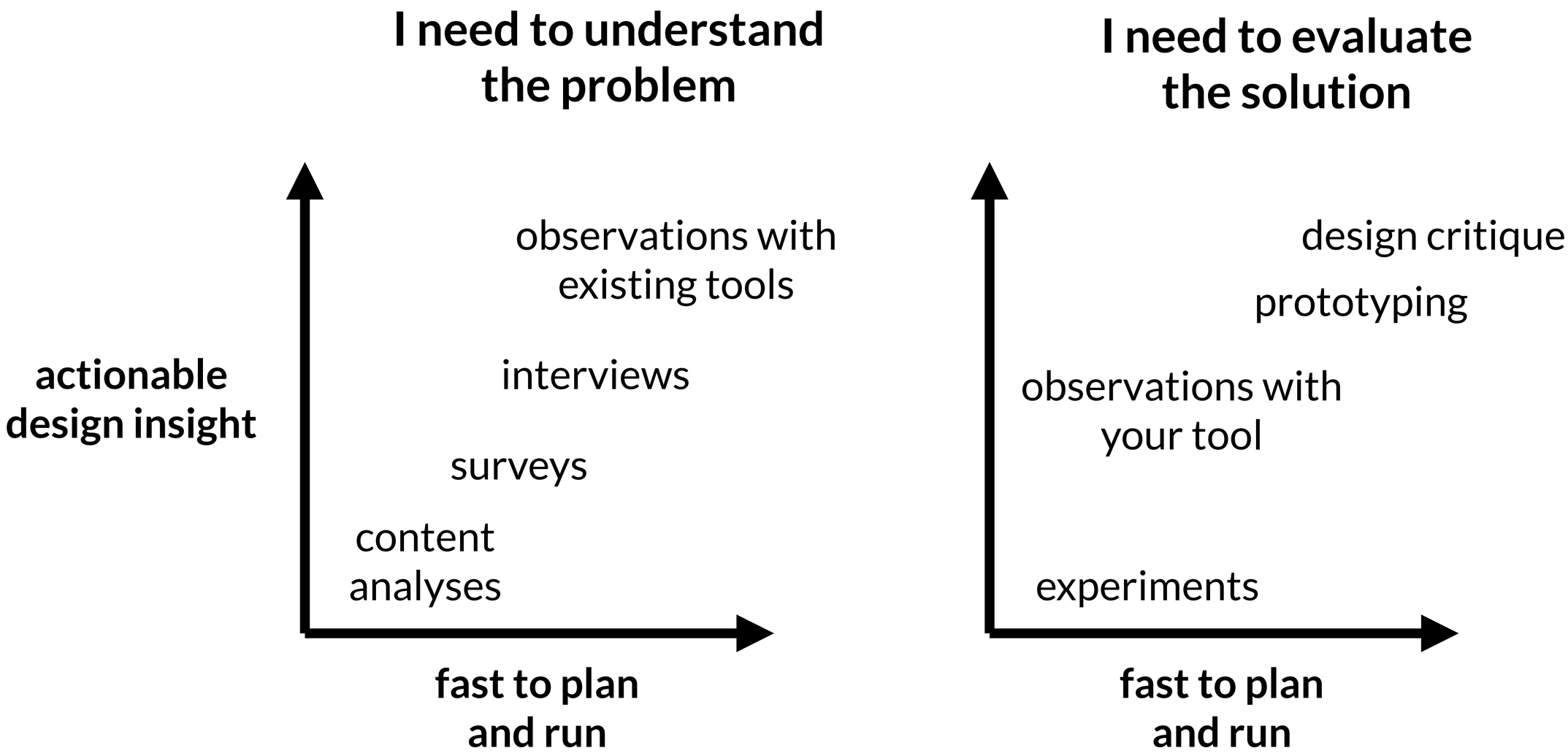
When to use a design method



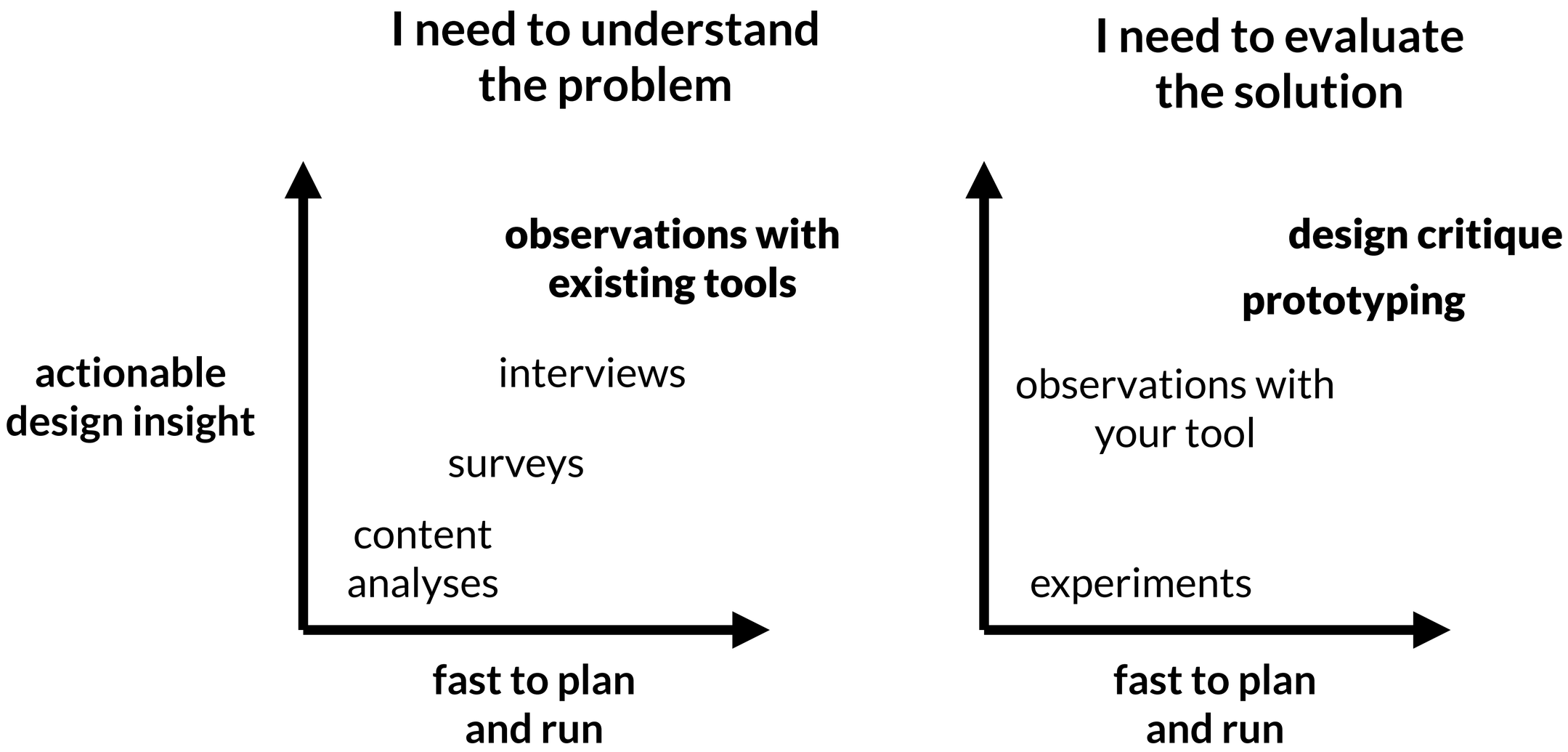
When to use a design method



When to use a design method



When to use a design method



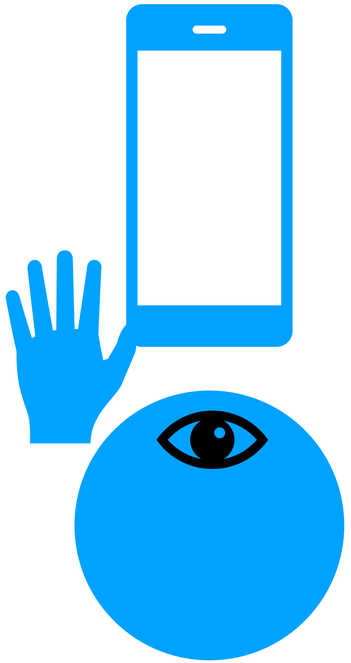
Understanding Problems in a Time Crunch: Observations

Answers the questions,

(1) "Did I pick an **actual** problem?"

(2) "What **issues** can a tool help fix?"

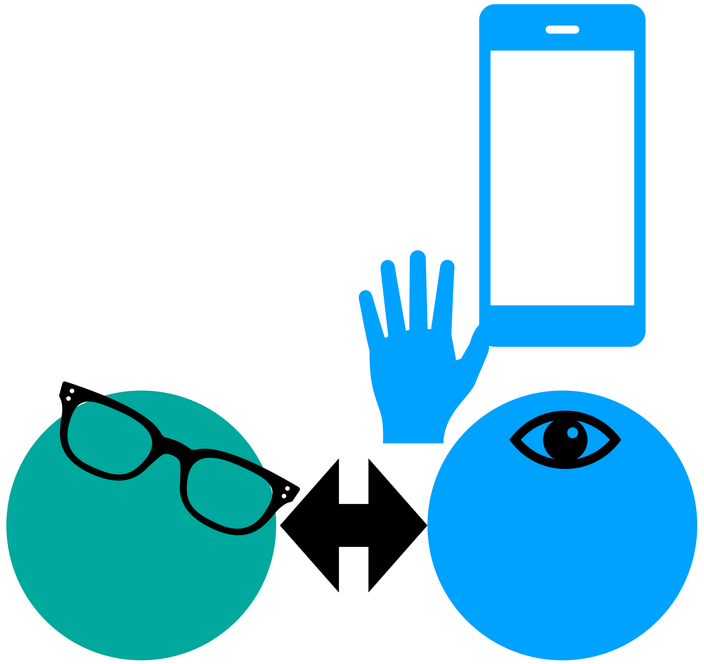
Observations



designed thing

user

Observations

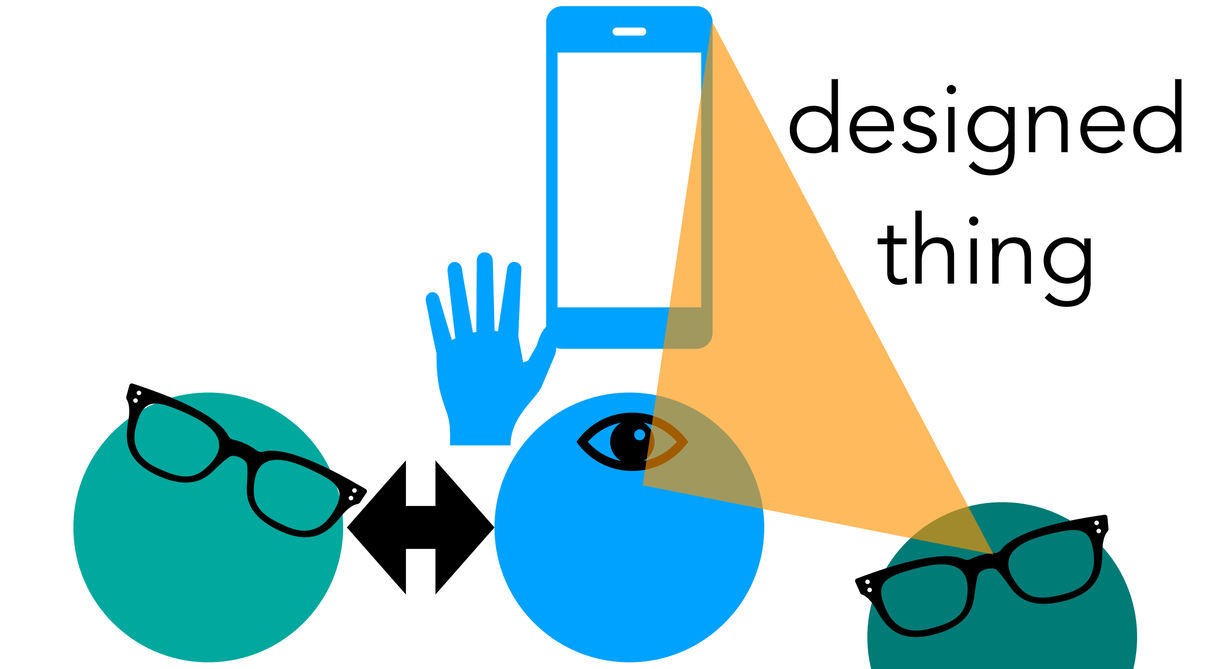


designed thing

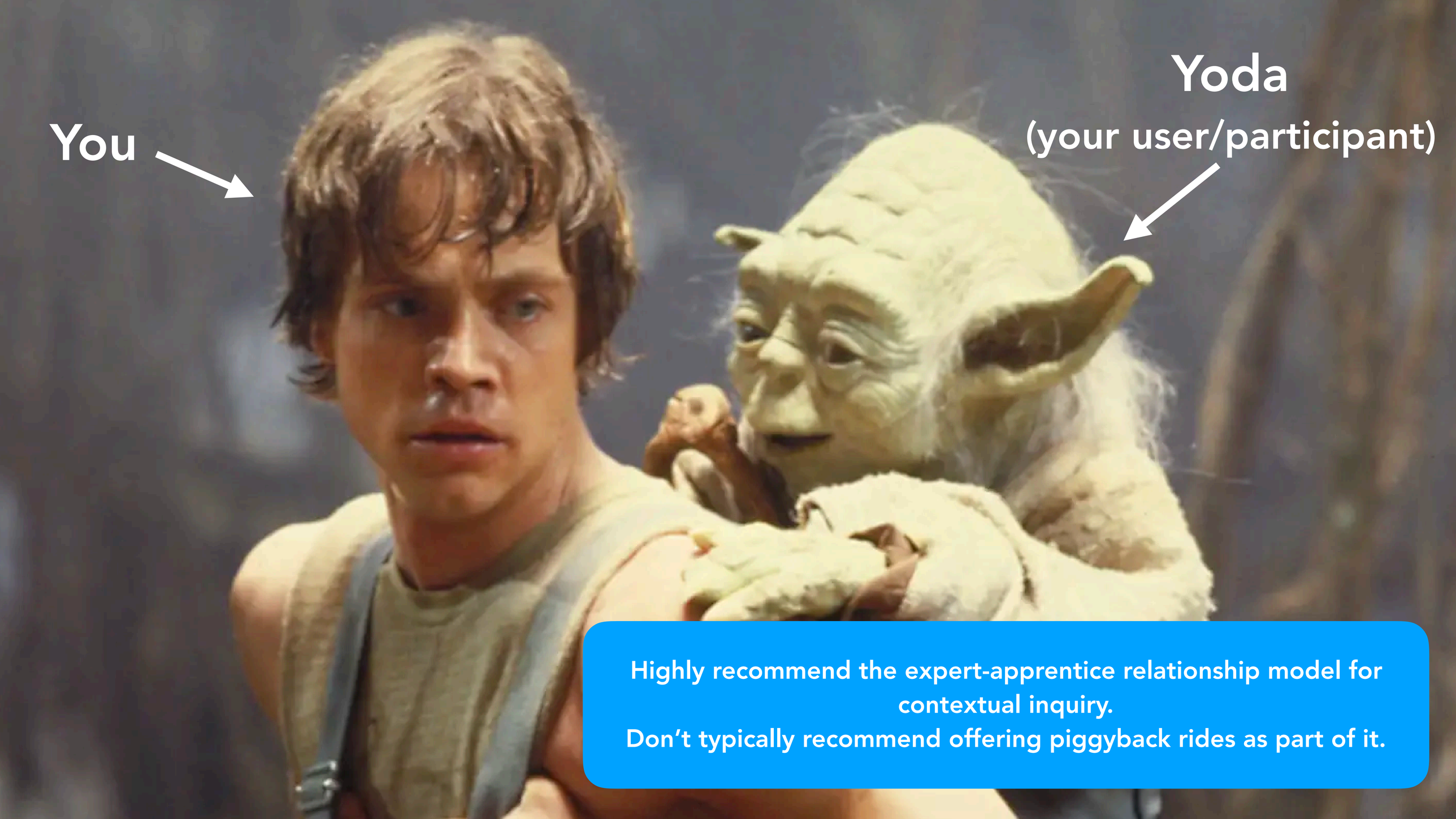
facilitator user

greets user, gives tutorial,
asks and answers questions

Observations



takes focused, complete notes


You
Yoda
(your user/participant)
Highly recommend the expert-apprentice relationship model for contextual inquiry.
Don’t typically recommend offering piggyback rides as part of it.

FORMATIVE STUDY

We conducted a formative study to understand the process that programmers follow when creating executable code examples from their own code, and the obstacles they encounter along the way. We observed 12 programmers as they created example code. Participants were recruited from our professional networks, local MeetUps, and computer science researchers from a local university.

This study and a review of literature on code examples led to design recommendations for improving the user experience of extracting code examples from existing code (Figure 2). We refer the reader to Section A1 of the auxiliary material for protocol details and observations from the formative study.

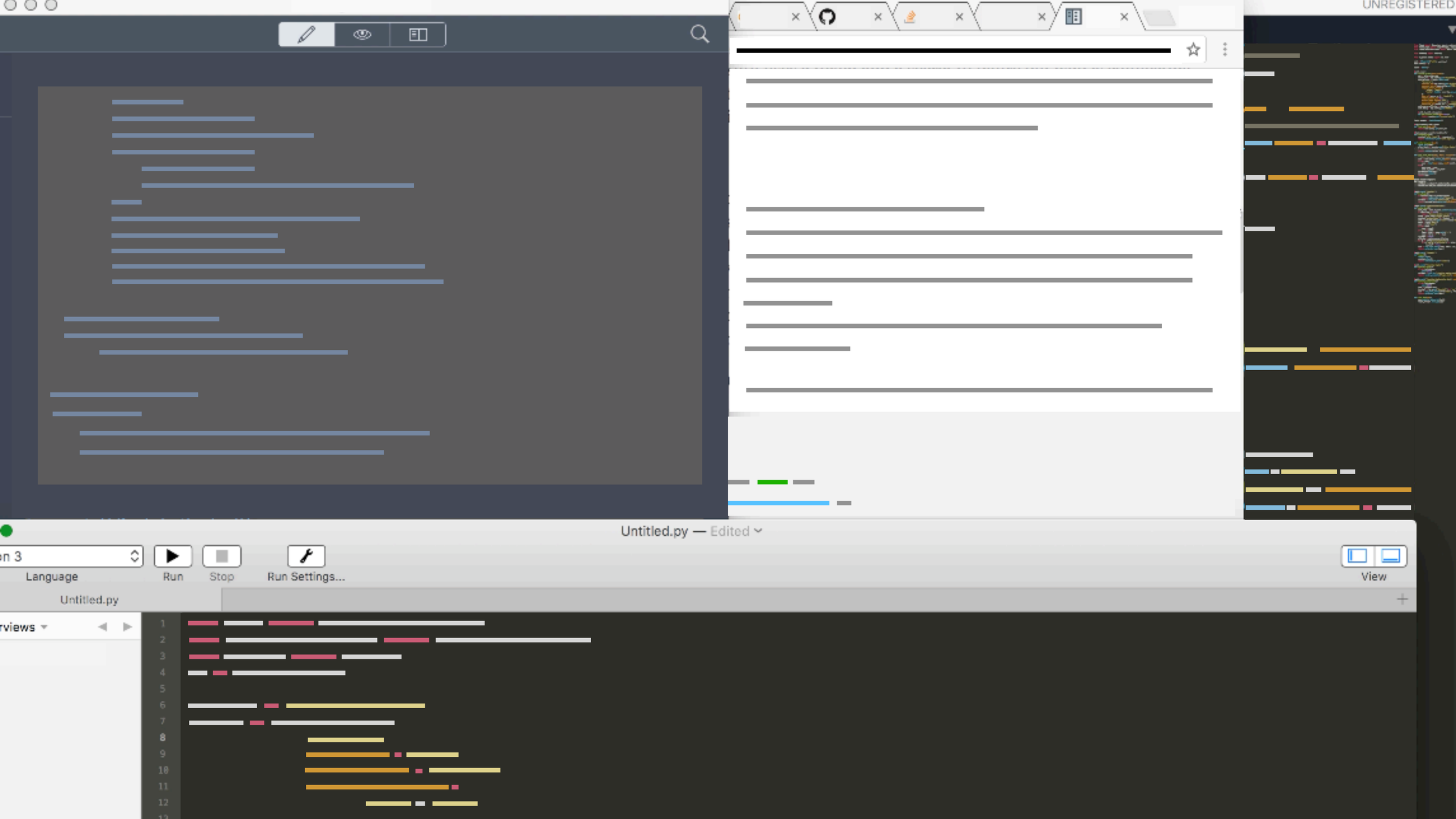
Authors made examples by...	**Tools should help authors...**
Copying the original code and pasting into example editor	• Create examples from text selections • Add lines from original code at any time
Replacing variables with meaningful literal values	• Review and insert literal values that preserve program behavior
Tweaking comments and code format for readability	• Directly edit code to add comments, group lines, and add `print` statements
Making examples could be time-consuming because...	**Better tools could...**
Authors left out code	• Suggest lines of code that the current example needs to run • Add missing code automatically when it's the only sensible fix
Authors introduced errors via transcription or edits	• Constrain manual code edits • Enable early and frequent testing
It took time to remove irrelevant code	• Start from a blank file • Omit code except for explicit code selections and necessary fixes

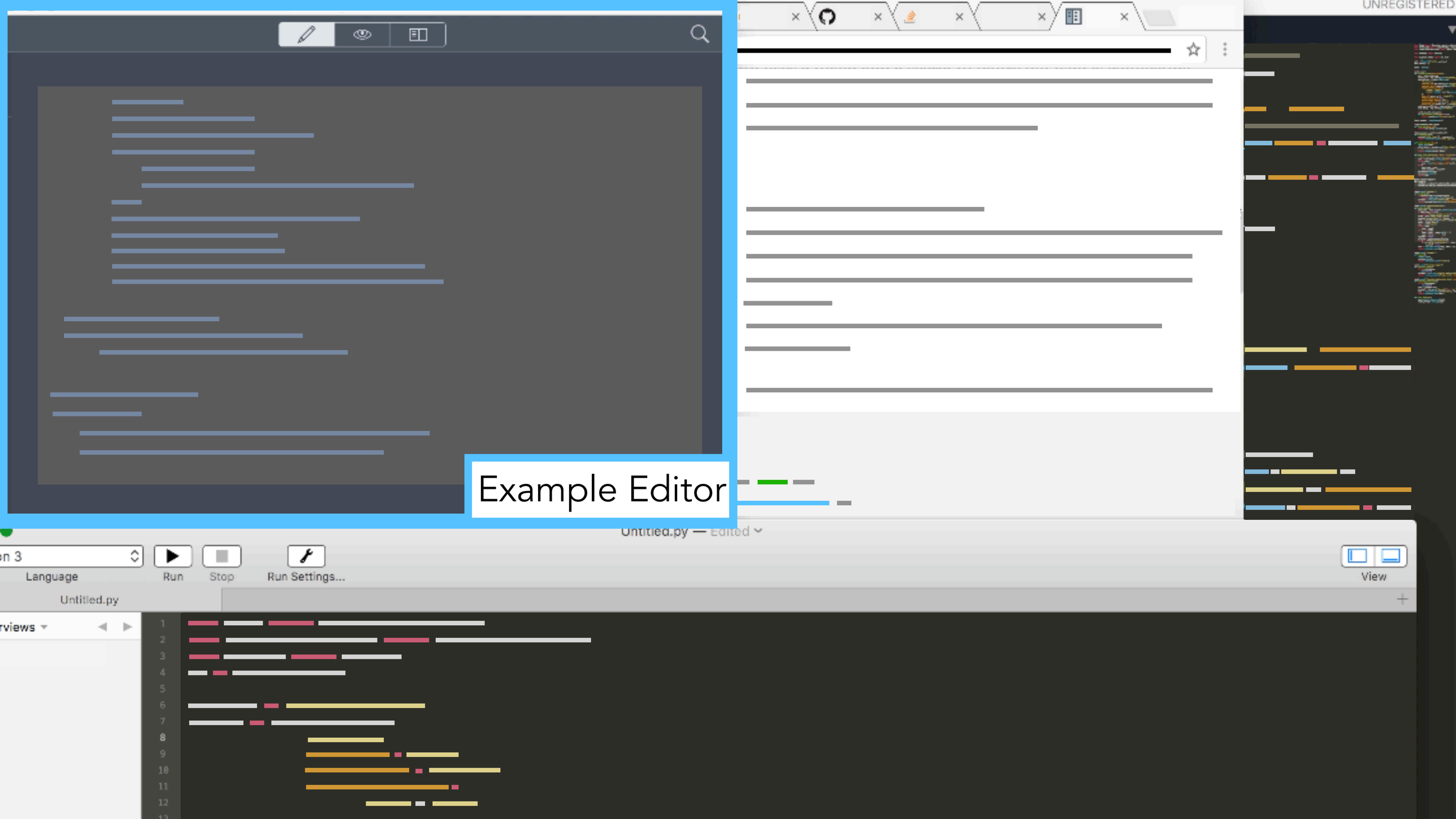
FORMATIVE STUDY

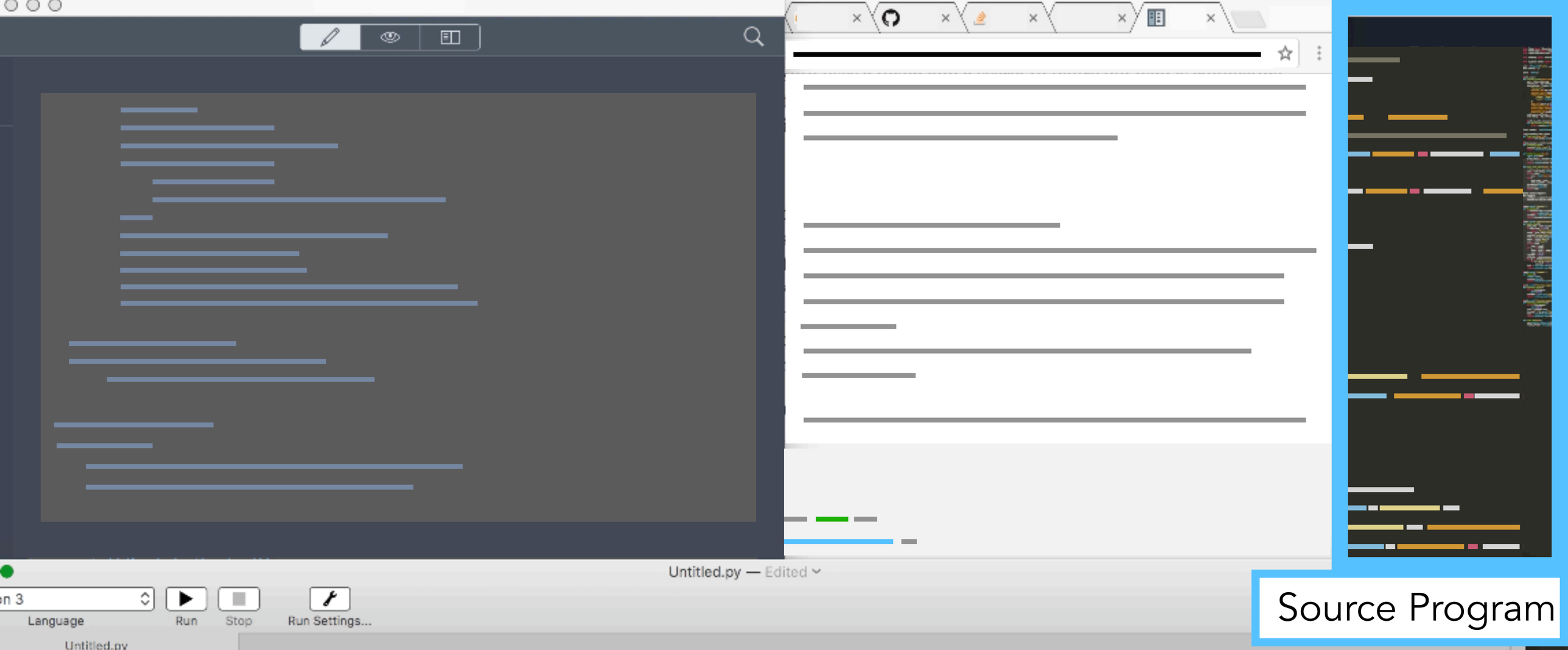
We conducted a formative study to understand the process that programmers follow when creating executable code examples from their own code, and the obstacles they encounter along the way. We observed 12 programmers as they created example code. Participants were recruited from our professional networks, local MeetUps, and computer science researchers from a local university.

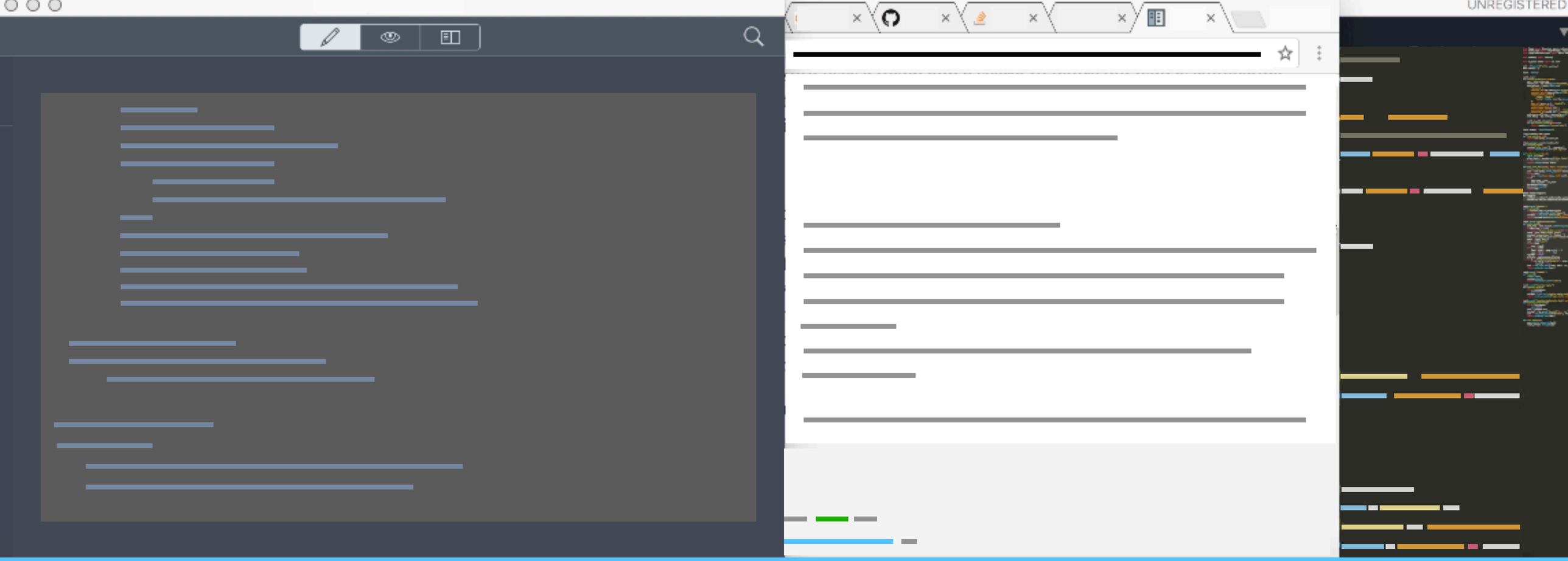
This study and a review of literature on code examples led to design recommendations for improving the user experience of extracting code examples from existing code (Figure 2). We refer the reader to Section A1 of the auxiliary material for protocol details and observations from the formative study.

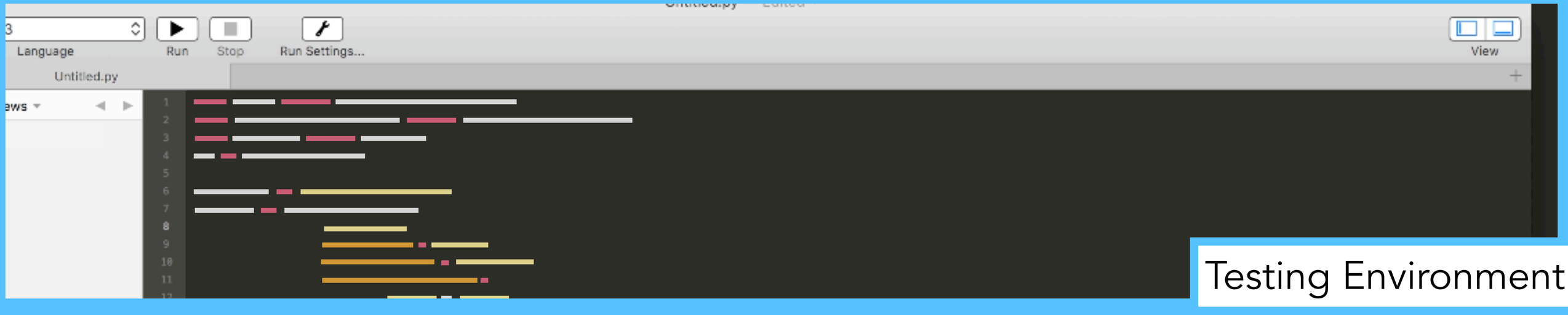
Authors made examples by...	**Tools should help authors...**
Copying the original code and pasting into example editor	• Create examples from text selections • Add lines from original code at any time
Replacing variables with meaningful literal values	• Review and insert literal values that preserve program behavior
Tweaking comments and code format for readability	• Directly edit code to add comments, group lines, and add `print` statements
Making examples could be time-consuming because...	**Better tools could...**
Authors left out code	• Suggest lines of code that the current example needs to run • Add missing code automatically when it's the only sensible fix
Authors introduced errors via transcription or edits	• Constrain manual code edits • Enable early and frequent testing
It took time to remove irrelevant code	• Start from a blank file • Omit code except for explicit code selections and necessary fixes


Untitled.py — Edited
Language
Run
Stop
Run Settings...
View
Untitled.py

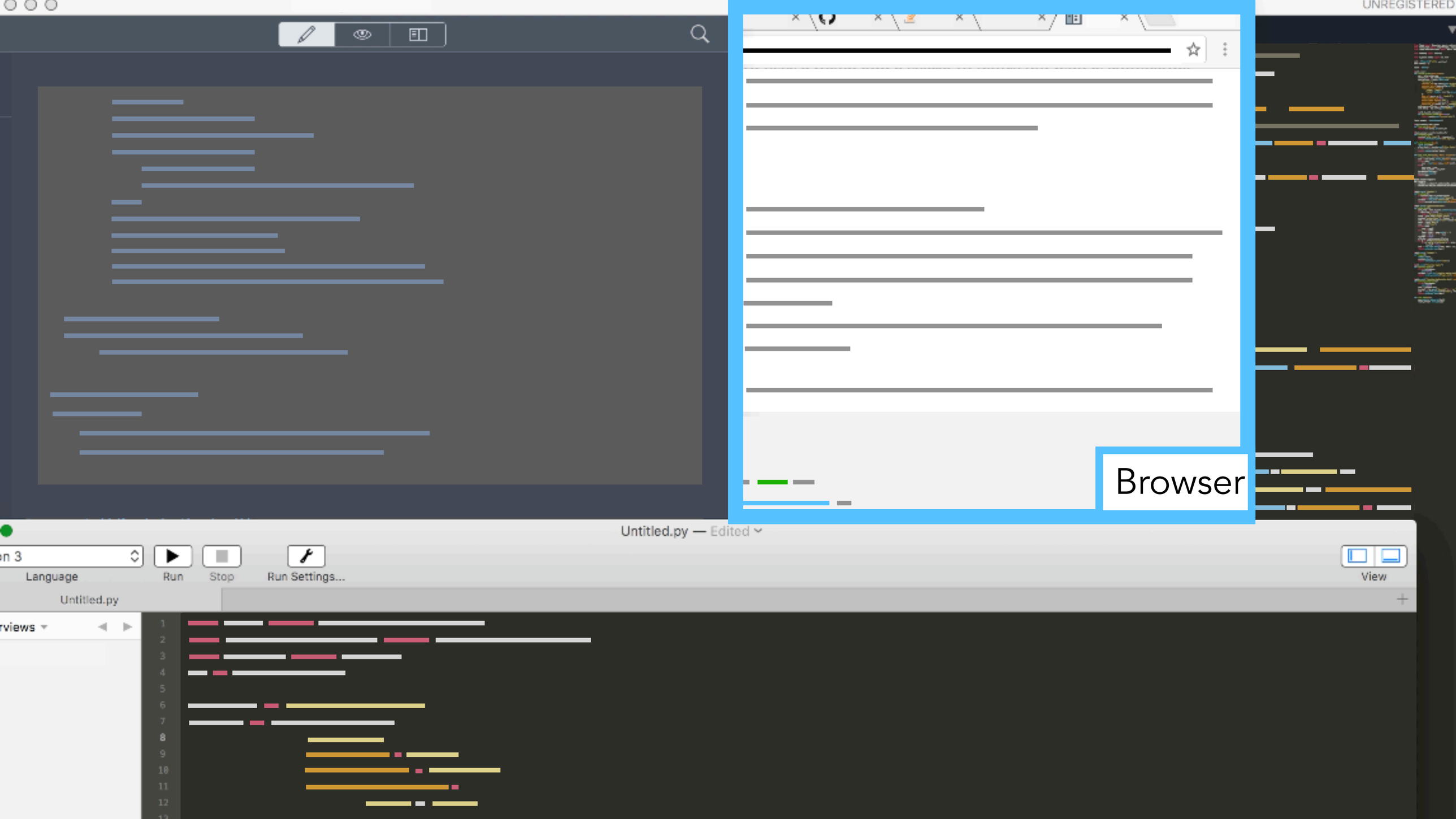

Example Editor
Untitled.py — Edited
Language
Run
Stop
Run Settings...
View
Untitled.py

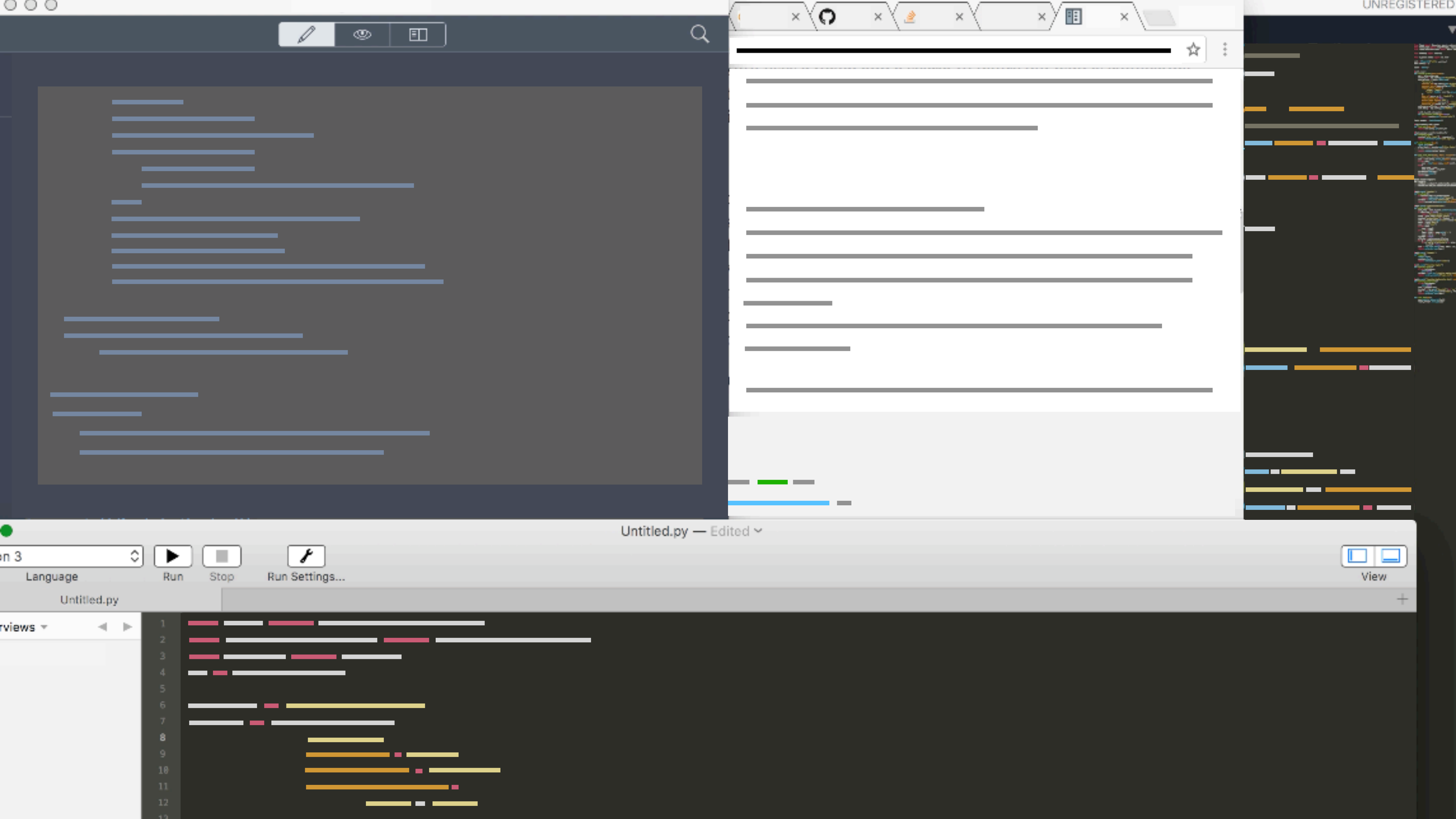

Untitled.py — Edited
Language
Run
Stop
Run Settings...
Untitled.py
Source Program

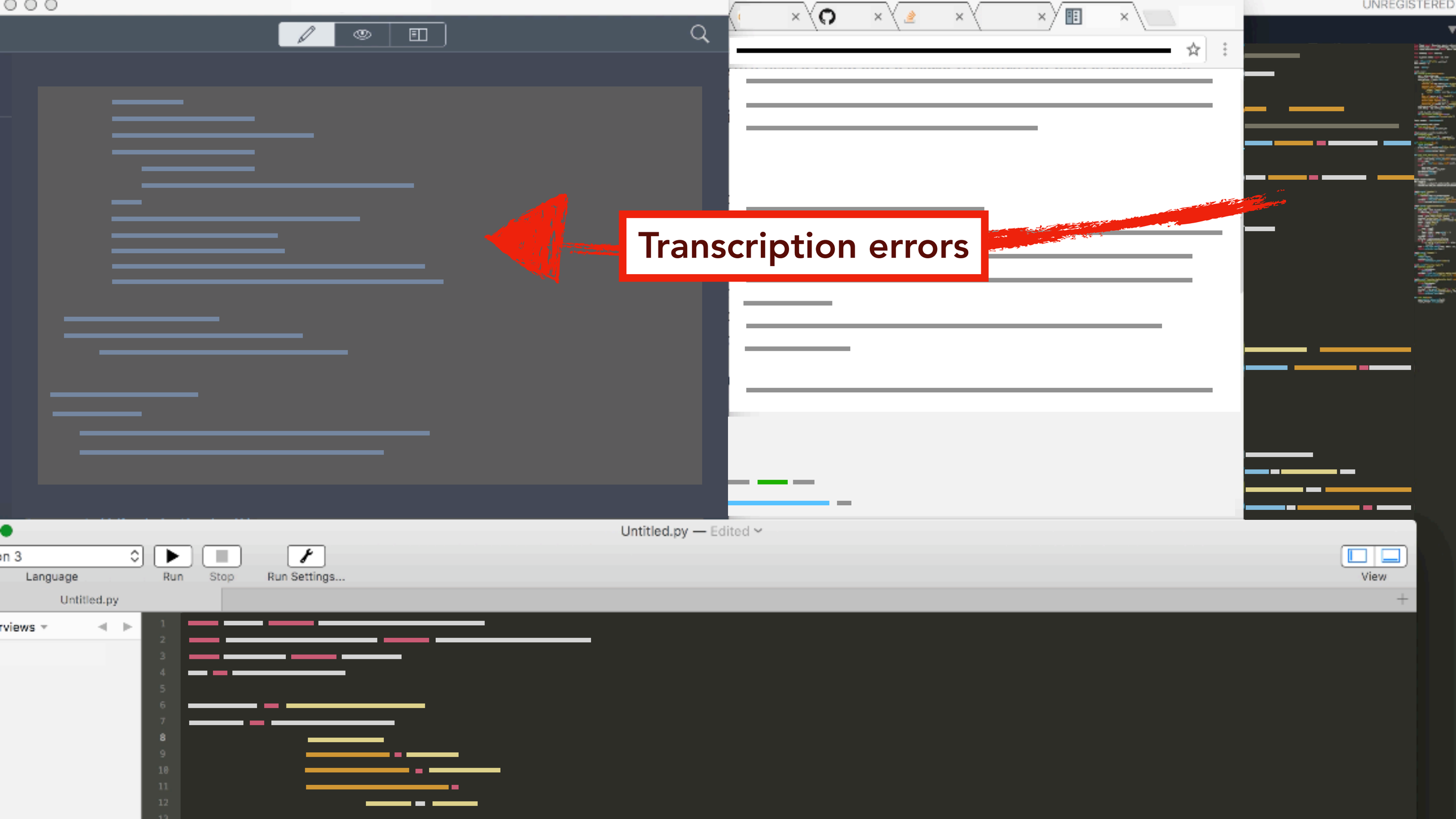


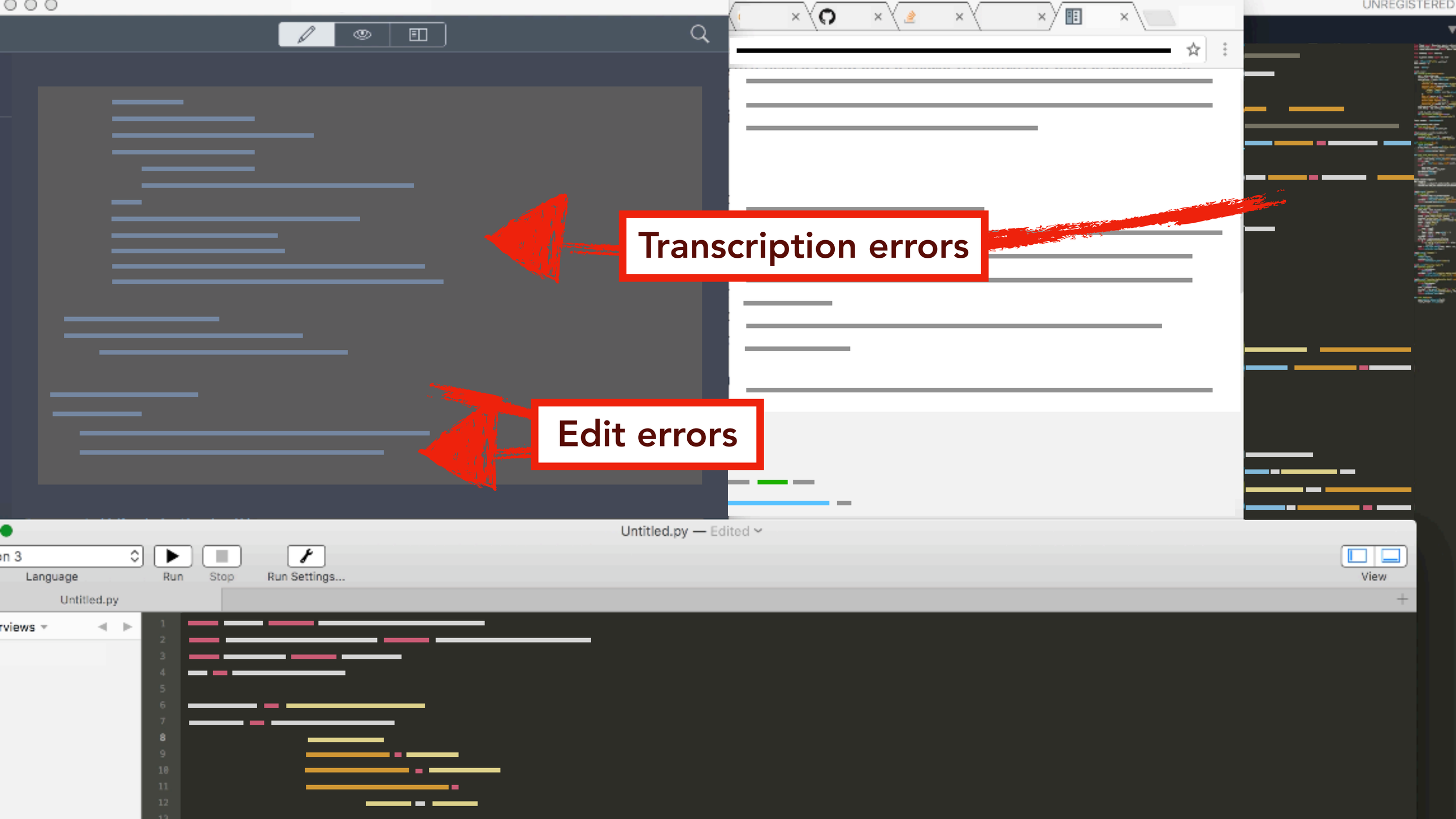


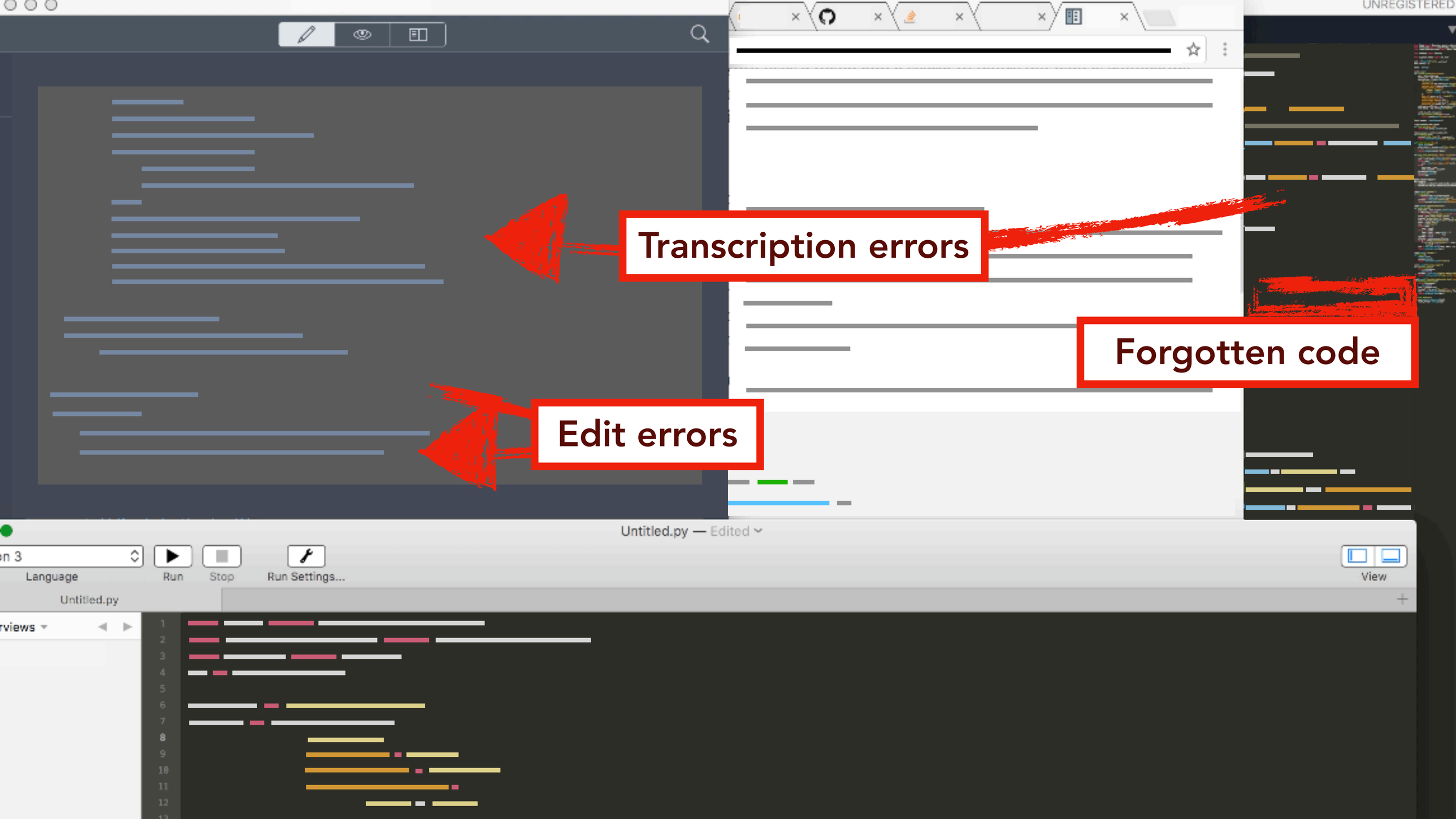
Testing Environment

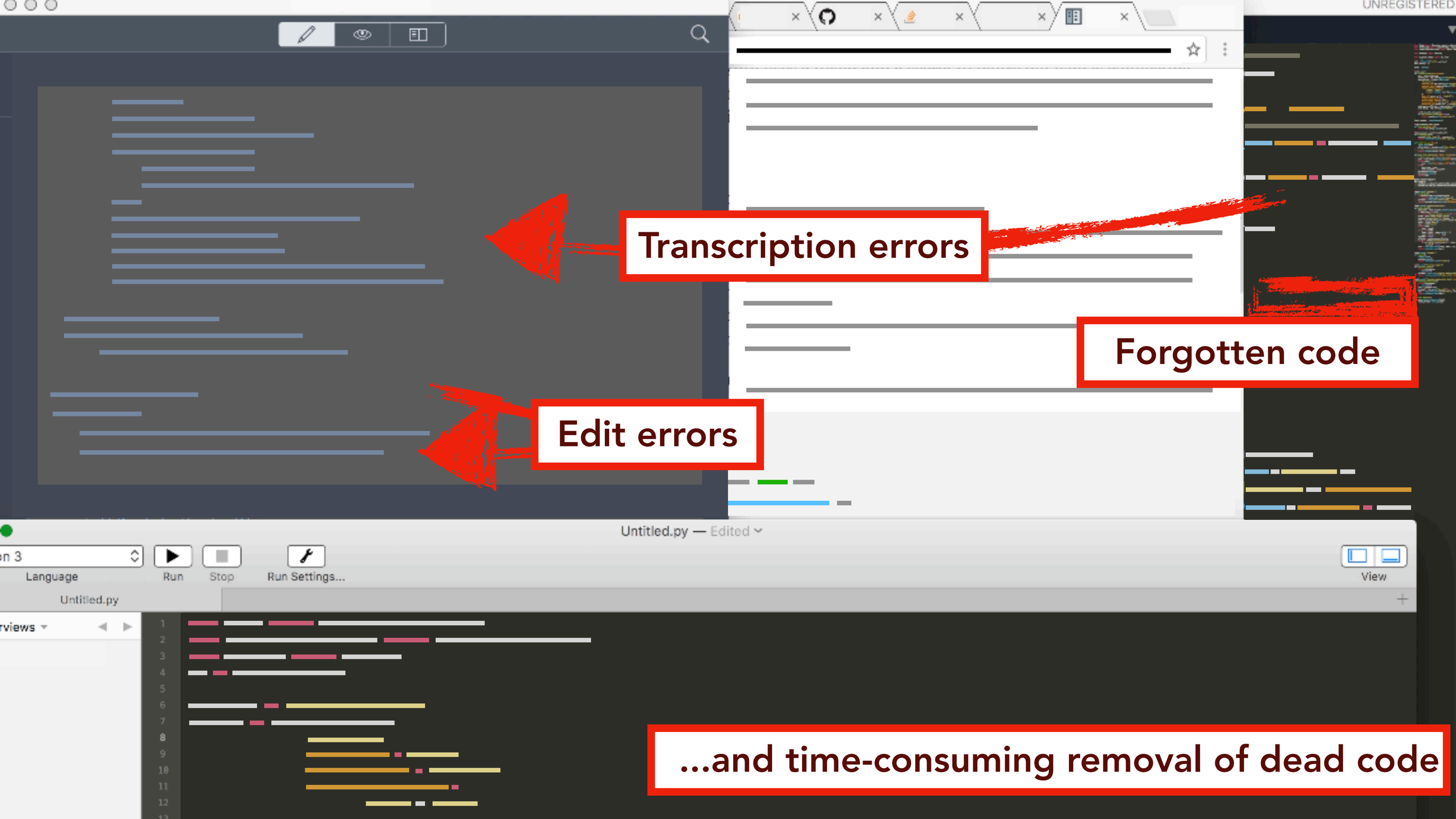

Browser
Untitled.py — Edited
Language
Run
Stop
Run Settings...
View
Untitled.py


Untitled.py — Edited
Language
Run
Stop
Run Settings...
View
Untitled.py


Transcription errors
Untitled.py — Edited
Language
Run
Stop
Run Settings...
View
Untitled.py


Transcription errors
Edit errors
Untitled.py — Edited
Language
Run
Stop
Run Settings...
View
Untitled.py


Transcription errors
Forgotten code
Edit errors
Untitled.py — Edited
Language
Run
Stop
Run Settings...
View
Untitled.py


Transcription errors
Forgotten code
Edit errors
Untitled.py — Edited
Language
Run
Stop
Run Settings...
View
Untitled.py
...and time-consuming removal of dead code

ANDREW'S MAXIMUM-FUN, MINIMUM-REGRET OBSERVATION TIPS

1. Keep It focused

1. Make your research questions **before the study**. Iterate. Keep the good ones.
2. Help **users understand what feedback is actionable** to you—and what's not
   a. Set the parameters of the conversation early
   b. Provide on-going guidance

ANDREW'S MAXIMUM-FUN, MINIMUM-REGRET OBSERVATION TIPS

2. Plan your notes for fast analysis

1. Take notes *and* record the conversation
2. **Structure your notes** document to make analysis easy and fast
3. Start synthesizing right after the study

TARGETED NOTES

Benefits and Challenges of Mixed-Initiative

When Guide Rails Are Helpful

Directing Focus to What Work Still Had to B[e] Done

- Participants generally reported that it was helpful t[o] and get suggestions of definitions to include (e.g.,
- "[the features this participant marked as most impo[rtant] task of making an example that worked rather than[] of which variables I needed to declare, etc." (N07)

Making Quick Work of Otherwise Tedious Trial and Error

- The value of small, automatic fixes
  - "although not necessarily hard to do, [all of the other featu[res] example a lot easier because I just had to look at the relev[ant] needed it or not instead of having to manually add them in[]"
  - "It fills in a lot of things that people usually don't really think about (exceptions, variables/constants) and saves a lot of time spent just searching and copy/pasting." (N04)
  - "Rocki[e] saved me the trouble of having to go through and find things like [u]ndeclared variables, missing import statements, and unchecked exceptions, which prevented my Sierra code from compiling." (N05)
- Some of the many small fixes CodeScoop made automatically, but that participants had to do manually in the baseline

A section for each research question (make before study)

interpretation (add in real time)

evidence (quotes, observations, add in real time)

user IDs

ANDREW'S MAXIMUM-FUN, MINIMUM-REGRET OBSERVATION TIPS

3. Develop rapport with users

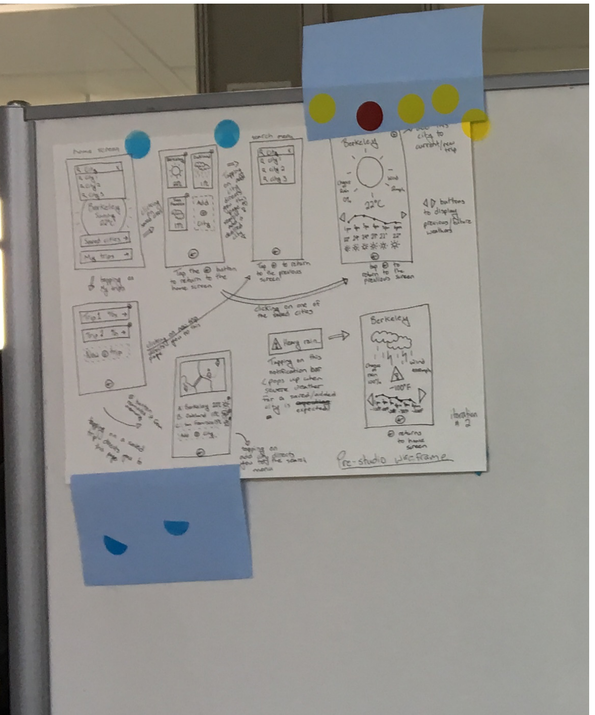
1. There's always time for a bit of small talk
   a. Make them feel **comfortable**
   b. Make them feel **appreciated** (they're doing you a huge favor!)
   c. Make them **want to help again**

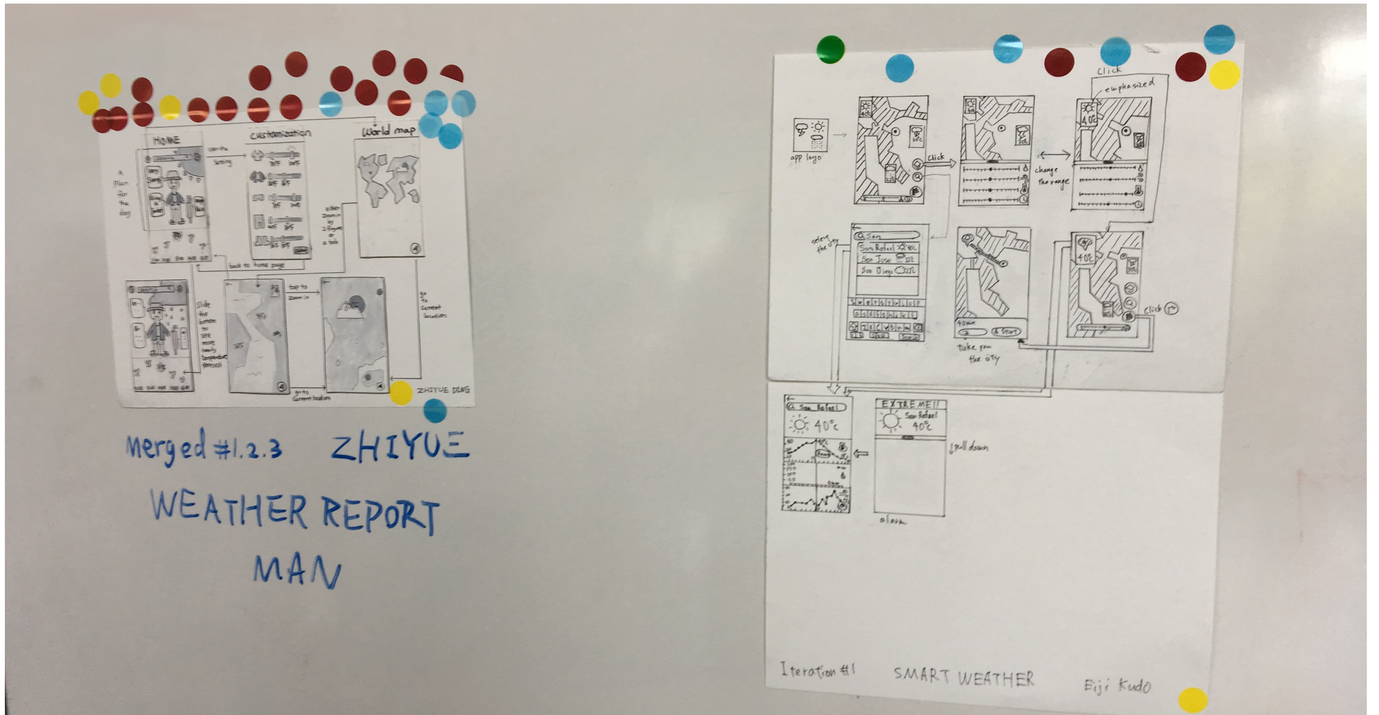
Understanding Solutions in a Time Crunch: Critiques

Answers the questions,

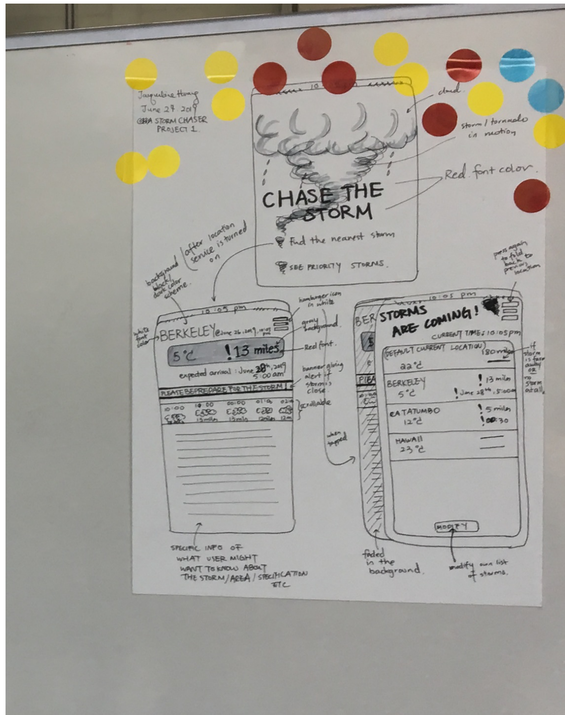
(1) "*Does this **solve** the problem?*"

(2) "*Is this something that users (and my peers) will get **excited** about?*


Pre-studio wireframe


HOME
customization
world map
Merged #1.2.3 ZHIYUE
WEATHER REPORT
MAN

Click
app logo
Iteration #1
SMART WEATHER
Eiji Kudo


CHASE THE STORM
Red font color
STORMS ARE COMING!
BERKELEY
5°C
13 miles
SPECIFIC INFO OF WHAT USER MIGHT WANT TO KNOW ABOUT THE STORM/AREA/SPECIFICATION ETC

Getting Feedback on Programming Tools Before They're Built

- Get feedback from ***multiple users***
- Get feedback from ***multiple tool builders***
- Present ***multiple ideas***, not just one
- Come up with ***concrete worked examples***
- Be open to new ideas

1. Get Feedback from Multiple Users

Programmers have diverse work styles and preferences. Here's one way of looking at differences in work styles.

- "*Opportunistic programmers* are more concerned with productivity than control or understanding."
- "*Pragmatic programmers* balance productivity with control and understanding."
- "*Systematic programmers* program defensively and these are the programmers for whom low-level APIs are targeted."

From Clarke, "Measuring API Usability", Dr. Dobb's
Elaborated on in Stylos and Clarke, "Usability Implications of Requiring Parameters in Objects' Constructors", ICSE '07

1. Get Feedback from Multiple Users

Abby

Pat

Tim

	Support ALL TYPES of users and their Cognitive Styles[1]
Motivations	People have different motivations for using technology: • **Abby** uses technology only as needed for his/her task. S/he prefers familiar features to keep focused on the task. • **Tim** likes using technology to learn what new features can help him/her accomplish. • **Pat** is like Abby in some situations and like Tim in others. **Make clear what a new feature does, and why someone would use it, but also keep familiar features available.**

GenderMag personas, gendermag.org

2. Get Feedback from *Tool Builders*

"When artists assessed one another's performances, they were about twice as accurate as managers and test audiences in predicting how often the videos would be shared. Compared to creators, managers and test audiences were 56 percent and 55 percent more prone to major false negatives, undervaluing a strong, novel performance by five ranks or more in the set of ten they viewed."

From Adam Grant, *Originals*, regarding Justin Berg's publication , "Balancing on the Creative Highwire: Forecasting the Success of Novel Ideas in Organizations"

3. Present Multiple Ideas, Not Just One

- Critics are more willing to give substantive feedback when there are several ideas in play
- Designs that evolve from parallel prototypes (rather than sequential prototypes)

Getting the Right Design and the Design Right: Testing Many Is Better Than One

Maryam Tohidi
University of Toronto
Toronto, Canada
mtohidi@dgp.toronto.edu

William Buxton
Microsoft Research
Toronto, Canada
bill@billbuxton.com

Ronald Baecker
University of Toronto
Toronto, Canada
rmb@kmdi.utoronto.ca

Abigail Sellen
Microsoft Research
Cambridge, UK
asellen@microsoft.com

ABSTRACT

We present a study comparing usability testing of a single interface versus three functionally equivalent but stylistically distinct designs. We found that when presented with a single design, users give significantly higher ratings and were more reluctant to criticize than when presented with the same design in a group of three. Our results imply that by presenting users with alternative design solutions, subjective ratings are less prone to inflation and give rise to more and stronger criticisms when appropriate. Contrary to our expectations, our results also suggest that usability testing by itself, even when multiple designs are presented, is not an effective vehicle for soliciting constructive suggestions about how to improve the design from end users. It is a means to identify problems, not provide solutions.

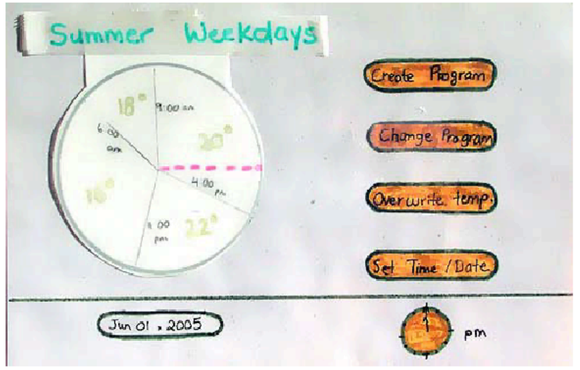


Figure 1. The "Circular" paper prototype

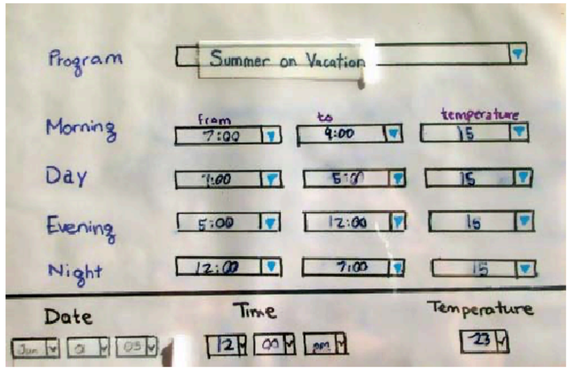


Figure 2. The "Tabular" paper prototype

Parallel Prototyping Leads to Better Design Results, More Divergence, and Increased Self-Efficacy

STEVEN P. DOW, ALANA GLASSCO, JONATHAN KASS, MELISSA SCHWARZ, DANIEL L. SCHWARTZ, and SCOTT R. KLEMMER
Stanford University

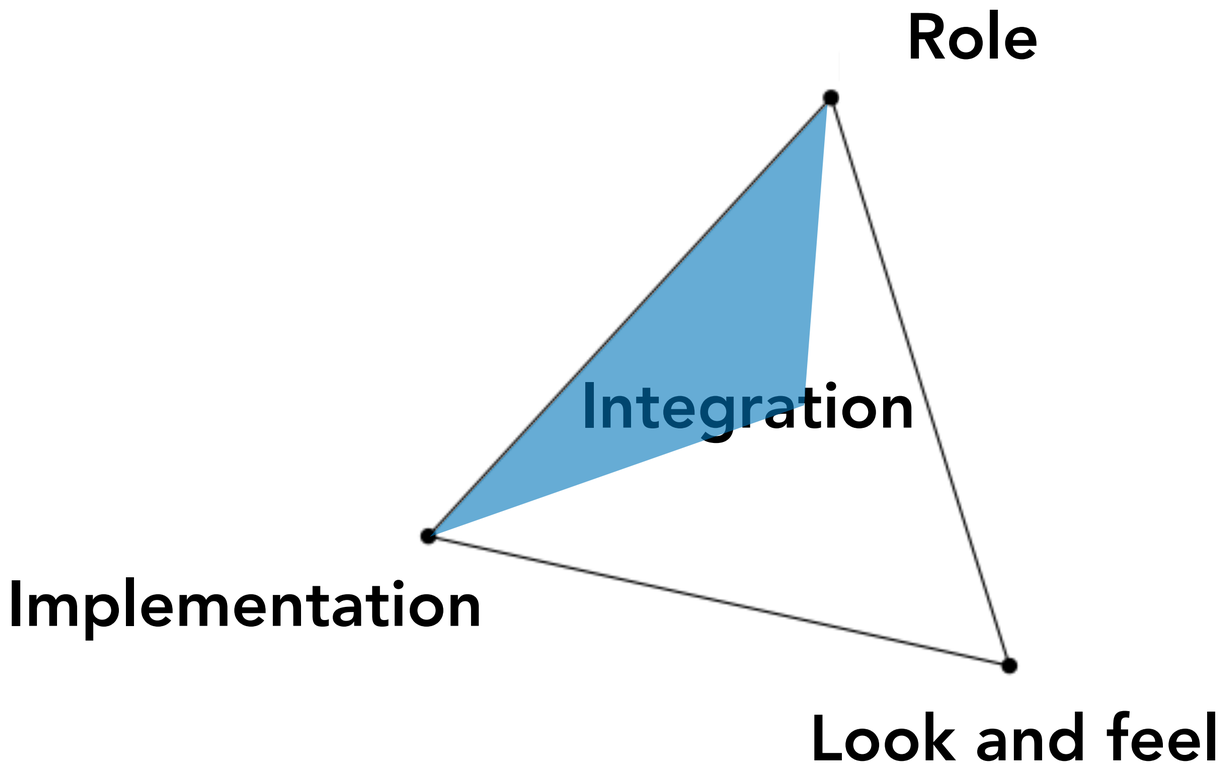
Iteration can help people improve ideas. It can also give rise to fixation, continuously refining one option without considering others. Does creating and receiving feedback on multiple prototypes in parallel, as opposed to serially, affect learning, self-efficacy, and design exploration? An experiment manipulated whether independent novice designers created graphic Web advertisements in parallel or in series. Serial participants received descriptive critique directly after each prototype. Parallel participants created multiple prototypes before receiving feedback. As measured by click-through data and expert ratings, ads created in the Parallel condition significantly outperformed those from the Serial condition. Moreover, independent raters found Parallel prototypes to be more diverse. Parallel participants also reported a larger increase in task-specific self-confidence. This article outlines a theoretical foundation for why parallel prototyping produces better design results and discusses the implications for design education.

Categories and Subject Descriptors: H.1.m. [**Information Systems**]: Models and Principles

General Terms: Experimentation, Design

Fig. 1. The experiment manipulates when participants receive feedback during a design process: in serial after each design (top) versus in parallel on three, then two (bottom).

4. Come up with concrete worked examples



Worked examples, or scenarios of tool usage showing real programs.

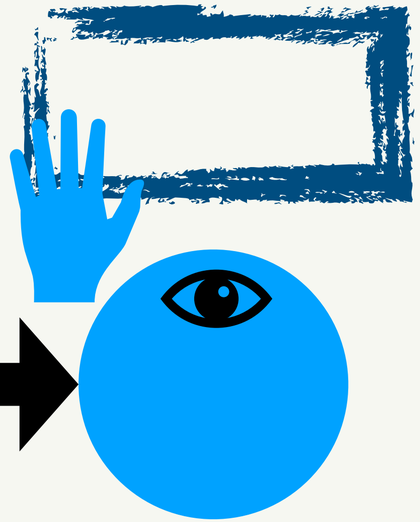
These let you simultaneously to start testing the *functionality and fit* of your idea while thinking about *implementation feasibility*.

WIZARD OF OZ STUDY

computer / "wizard"

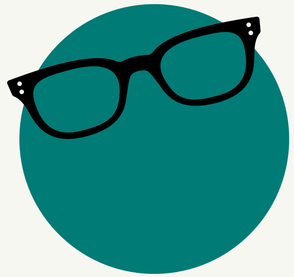
updates the prototype in response to user actions

prototyped thing

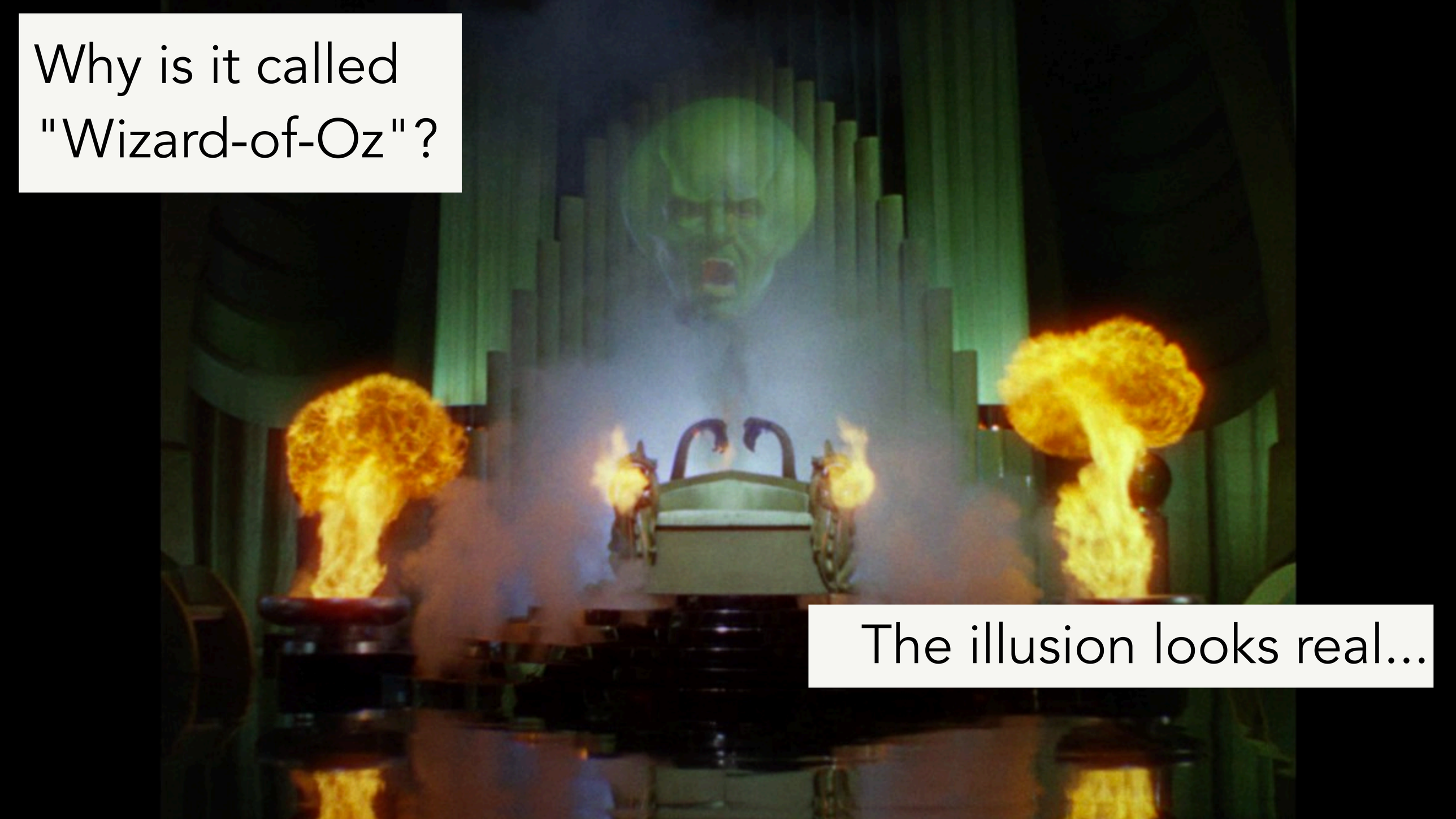


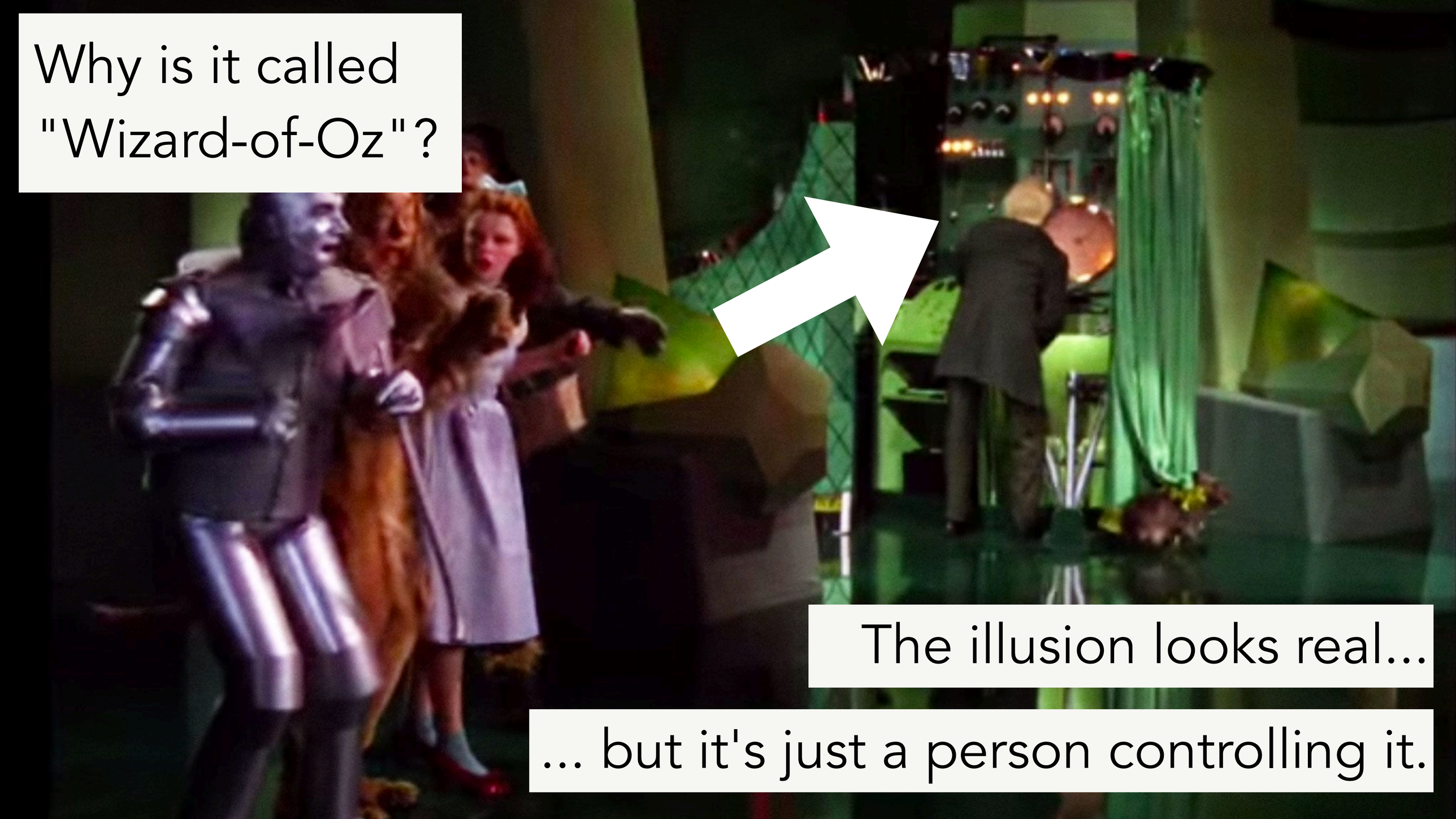
facilitator

user



observer


Why is it called "Wizard-of-Oz"?
The illusion looks real...


Why is it called "Wizard-of-Oz"?
The illusion looks real...
... but it's just a person controlling it.

A Discount Idea Evaluation Method

- Make a deck of slides
- Create a demo walkthrough of your 3 most exciting tool ideas
  - They show real programs, real text
  - They come with a problem description, solution description, and resolution
- Show this to 3 users, 3 tool builders. Ask them what they find most exciting and why.

3. Chop

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
        } else {
            policy = null;
        }

        final int offset = getEditor().getCaretModel().getOffset();
        PsiElement element = getFile().findElementAt(offset);
        element = PsiTreeUtil.getParentOfType(element, PsiCodeBlock.class, false);
        assertTrue("Selected element: " + element, element instanceof PsiCodeBlock);

        ControlFlow controlFlow = ControlFlowFactory.getInstance(getProject()).getControlFlow(element, policy);
        String result = controlFlow.toString().trim();

        final String expectedFullPath = StringUtil.trimEnd(file.getPath(), ".java") + ".txt";
        VirtualFile expectedFile = LocalFileSystem.getInstance().findFileByPath(expectedFullPath);
        String expected = LoadTextUtil.loadText(expectedFile).toString().trim();
        expected = expected.replaceAll("\r", "");
        assertEquals("Text mismatch (in file " + expectedFullPath + "):\n", expected, result);
    }

    // Not sure why this is failing on some simple tests (like flow3). It looks like the branching, reading, and
    // writing structure is correctly captured. So maybe we should just update the test output.
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
```

3. Chop

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
        } else {
            policy = null;
        }

        final int offset = getEditor().getCaretModel().getOffset();
        PsiElement element = getFile().findElementAt(offset);
        element = PsiTreeUtil.getParentOfType(element, PsiCodeBlock.class, false);
        assertTrue("Selected element: " + element, element instanceof PsiCodeBlock);

        ControlFlow controlFlow = ControlFlowFactory.getInstance(getProject()).getControlFlow(element, policy);
        String result = controlFlow.toString().trim();

        final String expectedFullPath = StringUtil.trimEnd(file.getPath(), ".java") + ".txt";
        VirtualFile expectedFile = LocalFileSystem.getInstance().findFileByPath(expectedFullPath);
        String expected = LoadTextUtil.loadText(expectedFile).toString().trim();
        expected = expected.replaceAll("\r", "");
        assertEquals("Text mismatch (in file " + expectedFullPath + "):\n", expected, result);
    }

    // Not sure why this is failing on some simple tests (like flow3). It looks like the branching, reading, and
    // writing structure is correctly captured. So maybe we should just update the test output.
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
```

3. Chop

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
        } else {
            policy = null;
        }

        final int offset = getEditor().getCaretModel().getOffset();
        PsiElement element = getFile().findElementAt(offset);
        element = PsiTreeUtil.getParentOfType(element, PsiCodeBlock.class, false);
        assertTrue("Selected element: " + element, element instanceof PsiCodeBlock);

        ControlFlow controlFlow = ControlFlowFactory.getInstance(getProject()).getControlFlow(element, policy);
        String result = controlFlow.toString().trim();

        final String expectedFullPath = StringUtil.trimEnd(file.getPath(), ".java") + ".txt";
        VirtualFile expectedFile = LocalFileSystem.getInstance().findFileByPath(expectedFullPath);
        String expected = LoadTextUtil.loadText(expectedFile).toString().trim();
        expected = expected.replaceAll("\r", "");
        assertEquals("Text mismatch (in file " + expectedFullPath + "):\n", expected, result);
    }

    // Not sure why this is failing on some simple tests (like flow3). It looks like the branching, reading, and
    // writing structure is correctly captured. So maybe we should just update the test output.
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
```

3. Chop

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
        } else {
            policy = null;
        }

        final int offset = getEditor().getCaretModel().getOffset();
        PsiElement element = getFile().findElementAt(offset);
        element = PsiTreeUtil.getParentOfType(element, PsiCodeBlock.class, false);
        assertTrue("Selected element: " + element, element instanceof PsiCodeBlock);

        ControlFlow controlFlow = ControlFlowFactory.getInstance(getProject()).getControlFlow(element, policy);
        String result = controlFlow.toString().trim();

        final String expectedFullPath = StringUtil.trimEnd(file.getPath(), ".java") + ".txt";
        VirtualFile expectedFile = LocalFileSystem.getInstance().findFileByPath(expectedFullPath);
        String expected = LoadTextUtil.loadText(expectedFile).toString().trim();
        expected = expected.replaceAll("\r", "");
        assertEquals("Text mismatch (in file " + expectedFullPath + "):\n", expected, result);
    }

    // Not sure why this is failing on some simple tests (like flow3). It looks like the branching, reading, and
    // writing structure is correctly captured. So maybe we should just update the test output.
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
```

3. Chop

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
        } else {
            policy = null;
        }

        final int offset = getEditor().getCaretModel().getOffset();
        PsiElement element = getFile().findElementAt(offset);
        element = PsiTreeUtil.getParentOfType(element, PsiCodeBlock.class, false);
        assertTrue("Selected element: " + element, element instanceof PsiCodeBlock);

        ControlFlow controlFlow = ControlFlowFactory.getInstance(getProject()).getControlFlow(element, policy);
        String result = controlFlow.toString().trim();

        final String expectedFullPath = StringUtil.trimEnd(file.getPath(), ".java") + ".txt";
        VirtualFile expectedFile = LocalFileSystem.getInstance().findFileByPath(expectedFullPath);
        String expected = LoadTextUtil.loadText(expectedFile).toString().trim();
        expected = expected.replaceAll("\r", "");
        assertEquals("Text mismatch (in file " + expectedFullPath + "):\n", expected, result);
    }

    // Not sure why this is failing on some simple tests (like flow3). It looks like the branching, reading, and
    // writing structure is correctly captured. So maybe we should just update the test output.
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
```

3. Chop

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
        } else {
            policy = null;
        }

        final int offset = getEditor().getCaretModel().getOffset();
        PsiElement element = getFile().findElementAt(offset);
        element = PsiTreeUtil.getParentOfType(element, PsiCodeBlock.class, false);
        assertTrue("Selected element: " + element, element instanceof PsiCodeBlock);

        ControlFlow controlFlow = ControlFlowFactory.getInstance(getProject()).getControlFlow(element, policy);
        String result = controlFlow.toString().trim();

        final String expectedFullPath = StringUtil.trimEnd(file.getPath(), ".java") + ".txt";
        VirtualFile expectedFile = LocalFileSystem.getInstance().findFileByPath(expectedFullPath);
        String expected = LoadTextUtil.loadText(expectedFile).toString().trim();
        expected = expected.replaceAll("\r", "");
        assertEquals("Text mismatch (in file " + expectedFullPath + "):\n", expected, result);
    }

    // Not sure why this is failing on some simple tests (like flow3). It looks like the branching, reading, and
    // writing structure is correctly captured. So maybe we should just update the test output.
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
}
```

3. Chop

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
        } else {
            policy = null;
        }

        final int offset = getEditor().getCaretModel().getOffset();
        PsiElement element = getFile().findElementAt(offset);
        element = PsiTreeUtil.getParentOfType(element, PsiCodeBlock.class, false);
        assertTrue("Selected element: " + element, element instanceof PsiCodeBlock);

        ControlFlow controlFlow = ControlFlowFactory.getInstance(getProject()).getControlFlow(element, policy);
        String result = controlFlow.toString().trim();

        final String expectedFullPath = StringUtil.trimEnd(file.getPath(), ".java") + ".txt";
        VirtualFile expectedFile = LocalFileSystem.getInstance().findFileByPath(expectedFullPath);
        String expected = LoadTextUtil.loadText(expectedFile).toString().trim();
        expected = expected.replaceAll("\r", "");
        assertEquals("Text mismatch (in file " + expectedFullPath + "):\n", expected, result);
    }

    // Not sure why this is failing on some simple tests (like flow3). It looks like the branching, reading, and
    // writing structure is correctly captured. So maybe we should just update the test output.
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
}
```

3. Chop

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
        } else {
            policy = null;
        }

        final int offset = getEditor().getCaretModel().getOffset();
        PsiElement element = getFile().findElementAt(offset);
        element = PsiTreeUtil.getParentOfType(element, PsiCodeBlock.class, false);
        assertNotNull("Not inside any PsiCodeBlock", element);

        ControlFlow flow = new ControlFlowAnalyzer(getProject()).getControlFlow(element, policy);
        String result = new InstructionTextPresentation(flow).getInstructionsText();

        final String expectedFullPath = FileUtil.changeExtension(file.getPath(), ".java") + ".txt";
        VirtualFile expectedFile = LocalFileSystem.getInstance().findFileByPath(expectedFullPath);
        String expected = VfsUtilCore.loadText(expectedFile).trim();
        expected = StringUtil.convertLineSeparators(expected);
        assertEquals("Wrong output for " + file.getPath() + "):\n", expected, result);
    }

    // Not sure why this is failing on some simple tests (like flow3). It looks like the branching, reading, and
    // writing structure is correctly captured. So maybe we should just update the test output.
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
}
```

Show input data for `element`:

- ☐ type: CodeBlock
- ☐ text: "{ i = 1; if (i == 1) return true; }"
- ☐ textOffset: 52
- ☐ firstChild: PsiElement →

3. Chop

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
        } else {
            policy = null;
        }

        final int offset = getEditor().getCaretModel().getOffset();
        PsiElement element = getFile().findElementAt(offset);
        element = PsiTreeUtil.getParentOfType(element, PsiCodeBlock.class, false);
        assertTru                                                    teof PsiCodeBlock);

        ControlFl                                                    etProject()).getControlFlow(element, policy);
        String re

        final Str                                                    tPath(), ".java") + ".txt";
        VirtualFi                                                    findFileByPath(expectedFullPath);
        String ex                                                    tring().trim();
        expected
        assertEqu                                                    + "):\n", expected, result);
    }

    // Not sure why this is failing on some simple tests (like flow3). It looks like the branching, reading, and
    // writing structure is correctly captured. So maybe we should just update the test output.
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
}
```

Show input data for `element`:

- ☑ type: CodeBlock
- ☐ text: "{ i = 1; if (i == 1) return true; }"
- ☐ textOffset: 52
- ☐ firstChild: PsiElement →

3. Chop

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
        } else {
            policy = null;
        }

        final int offset = getEditor().getCaretModel().getOffset();
        PsiElement element = getFile().findElementAt(offset);
        element = PsiTreeUtil.getParentOfType(element, PsiCodeBlock.class, false);
        assertTrue("Selected element: " + element, element instanceof PsiCodeBlock);

        ControlFlow controlFlow = Contro                    ()).getControlFlow(element, policy);
        String result = contr    low.toSt

        final String expectedFullPath =                     ".java") + ".txt";
        VirtualFile expectedFile = Local                    Path(expectedFullPath);
        String expected = LoadTextUtil.l                    im();
        expected = expected.replaceAll("
        assertEquals("Text mismatch (in                     expected, result);
    }

    // Not sure why this is failing on some simple tests (like flow3). It looks like the branching, reading, and
    // writing structure is correctly captured. So maybe we should just update the test output.
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
```

type: CodeBlock
text: "{ i = 1; if (i == 1) return true; }"

toString():
0: ReadVariable i
1: ConditionalGoTo [END] 2
...

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
        } else {
            policy = null;
        }

        final int offset = getEditor().getCaretModel().getOffset();
        PsiElement element = getFile().findElementAt(offset);
        element = PsiTreeUtil.getParentOfType(element, PsiCodeBlock.class, false);
        assertTrue("Selected element: " + element, element instanceof PsiCodeBlock);

        ControlFlow controlFlow = Contro                    ()).getControlFlow(element, policy);
        String result = controlFlow.toSt

        final String expectedFullPath =                     ".java") + ".txt";
        VirtualFile expectedFile = Local                    Path(expectedFullPath);
        String expected = LoadTextUtil.l                    im();
        expected = expected.replaceAll("
        assertEquals("Text mismatch (in                     expected, result);
    }

    // Not sure why this is failing on some simple tests (like flow3). It looks like the branching, reading, and
    // writing structure is correctly captured. So maybe we should just update the test output.
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
```

3. Chop

Make example

type: CodeBlock
text: "{ i = 1; if (i == 1) return true; }"

toString():
0: ReadVariable i
1: ConditionalGoTo [END] 2
...

```
public class ControlFlowTest extends LightCodeInsightTestCase {
    @NonNls
    private static final String BASE_PATH = "testData/psi/controlFlow";

    private static void doTestFor(final File file) throws Exception {
        String contents = StringUtil.convertLineSeparators(FileUtil.loadFile(file));
        configureFromFileText(file.getName(), contents);
        // extract factory policy class name
        Pattern pattern = Pattern.compile("^// (\\S*).*", Pattern.DOTALL);
        Matcher matcher = pattern.matcher(contents);
        assertTrue(matcher.matches());
        final String policyClassName = matcher.group(1);
        final ControlFlowPolicy policy;
        if ("LocalsOrMyInstanceFieldsControlFlowPolicy".equals(policyClassName)) {
            policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
```

3. Chop (Informal Everyday Sharing)

Make example

Result

Input:

element = PsiElement(type=CodeBlock, text="{i = 1, if(i == 1)...")

Snippet:

```
final ControlFlowPolicy policy = LocalsOrMyInstanceFieldsControlFlowPolicy.getInstance();
ControlFlow controlFlow = ControlFlowFactory.getInstance(getProject()).getControlFlow(element, policy);
```

Output:

```
controlFlow.toString() = "
0: ReadVariable i
1: ConditionalGoTo [END] 2
..."
```

```
    }

    //
    //
    private static void doAllTests() throws Exception {
        final String testDirPath = BASE_PATH;
        File testDir = new File(testDirPath);
        final File[] files = testDir.listFiles((dir, name) -> name.endsWith(".java"));
        for (int i = 0; i < files.length; i++) {
            File file = files[i];
            doTestFor(file);

            System.out.print((i + 1) + " ");
        }
    }
```

Objectives

- What prototypes should I make to help me find a good design?
- How should I collect feedback to improve my design?

(If time)

Pick two of the ideas you've been considering for your project?

Pair up. Make a pitch for these ideas to your partner. Find out which one most excites them.