The Impact of "Cosmetic" Changes on the Usability of Error Messages

Tao Dong
Google, Inc.
Mountain View, CA, USA
dongtao@acm.org

Kandarp Khandwala[*]
The Design Lab, UC San Diego
La Jolla, CA, USA
kandarpksk@gmail.com

ABSTRACT

Programmatic errors are often difficult to resolve due to poor usability of error messages. Applying theories of visual perception and techniques in visual design, we created three visual variants of a representative error message in a modern UI framework. In an online experiment, we found that the visual variants led to substantial improvements over the original error message in both error comprehension and resolution. Our results demonstrate that seemingly cosmetic changes to the presentation of an error message can have an oversized impact on its usability.

KEYWORDS

Error message; programming; software engineering; presentation technique; experiment; developer experience

[*] Kandarp Khandwala worked on this study when he was an intern at Google.

```dart
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  final _controller = new TextEditingController();

  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Flutter Demo',
      home: new Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          new TextField(
            controller: _controller,
            decoration: new InputDecoration(
              hintText: 'Type something',
            ),
          ),
        ],
      ),
    );
  }
}
```

```
┤ EXCEPTION CAUGHT BY WIDGETS LIBRARY ├
The following assertion was thrown building TextField(controller:
TextEditingController#335e3(TextEditingValue(text: ┊├, selection: TextSelection(baseOffset: -1,
extentOffset: -1, affinity: TextAffinity.downstream, isDirectional: false), composing:
TextRange(start: -1, end: -1))), decoration: InputDecoration(hintText: "Type something"),
autocorrect: true, max length enforced, dirty, state: _TextFieldState#549bb):
TextField widgets require a Material widget ancestor, but we couldn't find any.
In material design, most widgets are conceptually "printed" on a sheet of material. In Flutter's
material library, this material is represented by the Material widget. It is the Material widget
that renders ink splashes, for instance. Because of this, many material library widgets require that
there be a Material widget in the tree above them.
To introduce a Material widget, you can either directly include one, or use a widget that contains
Material itself, such as a Card, Dialog, Drawer, or Scaffold.
The specific widget that could not find a Material ancestor was:
  TextField(controller: TextEditingController#cd3e1(TextEditingValue(text: ┊├, selection:
  TextSelection(baseOffset: -1, extentOffset: -1, affinity: TextAffinity.downstream, isDirectional:
  false), composing: TextRange(start: -1, end: -1))), decoration: InputDecoration(hintText: "Type
  something"), autocorrect: true, max length enforced)
The ancestors of this widget were:
  Padding(padding: EdgeInsets.all(50.0))
  Column(direction: vertical, mainAxisAlignment: center, crossAxisAlignment: center)
  ExampleWidget
  Semantics(container: false, properties: SemanticsProperties, label: null, value: null, hint: null)
  Builder
  RepaintBoundary-[GlobalKey#fbb10]
  IgnorePointer(ignoring: false)
  FadeTransition(opacity: AlwaysStoppedAnimation<double>#1adfb(▶ 1.0; paused))
  FractionalTranslation
  SlideTransition(animation: AnimationController#30593(⏸ 1.000; paused; for
  ...

When the exception was thrown, this was the stack:
#0    debugCheckHasMaterial.<anonymous closure> (package:flutter/src/material/debug.dart:97:7)
#1    debugCheckHasMaterial (package:flutter/src/material/debug.dart:100:4)
#2    _TextFieldState.build (package:flutter/src/material/text_field.dart:456:12)
#3    StatefulElement.build (package:flutter/src/widgets/framework.dart:3737:27)
#4    ComponentElement.performRebuild (package:flutter/src/widgets/framework.dart:3646:15)
#5    Element.rebuild (package:flutter/src/widgets/framework.dart:3495:5)
#6    ComponentElement._firstBuild (package:flutter/src/widgets/framework.dart:3626:5)
#7    StatefulElement._firstBuild (package:flutter/src/widgets/framework.dart:3776:11)
#8    ComponentElement.mount (package:flutter/src/widgets/framework.dart:3621:5)
#9    Element.inflateWidget (package:flutter/src/widgets/framework.dart:2907:14)
#10   Element.updateChild (package:flutter/src/widgets/framework.dart:2723:12)
...

Another exception was thrown: A RenderFlex overflowed by 99361 pixels on the bottom.
```

Figure 1: An example code snippet (upper) and the error message (lower) resulted from running it in a program written in Flutter, a UI framework.

1 INTRODUCTION

Programmatic errors are errors encountered by users of programming tools in the process of writing code. When a programmatic error occurs, the programmer can usually expect a message from the tool describing what has gone wrong and try to debug from there. Nonetheless, research has shown that programmers, especially novices, struggle to make sense of error messages, not to mention take action on them [9].

The work described in this paper aims to improve the usability of error messages in programming by enhancing their presentation. Specifically, we designed and evaluated three visual variants of a representative error message in an open-source UI framework, by applying visual perception theories and visual design techniques. In an experiment, we found that all three variants resulted in much higher error comprehension and error resolution rates than the original presentation of the error message, though the enhancements are seemingly "cosmetic."

Our design exploration and experimental results make two contributions: 1) validating specific presentation techniques for improving the usability of error messages in programming, and 2) linking the design space of error messages to HCI theories and techniques about visual perception.

2 RELATED WORK

2.1 The Usability of Error Messages

Unintuitive error messages can be frustrating to novice programmers and a significant barrier in learning programming [9]. Even to professional developers, resolving errors in programs can be time consuming and a source of productivity loss [7]. To make error messages easier to understand and act on, prior work has examined several aspects of the problem. Here, we describe a few representative works due to the space limit.

To address the *style* of error messages, Nielsen made a number of recommendations, such as human-readable language, polite phrasing, and precise descriptions [5]. Inspired by models in argumentation theory, Barik et al. [1] examined the *structure* of error messages, and argued that an effective error message should follow a simple argument layout consisting of a claim, grounds, and a warrant, with the possibility to include additional elements. To enrich the *content* of error messages, researchers have proposed systems such as HelpMeOut, which presents examples of how other programmers corrected similar errors, in order to help novices [4].

The *presentation* of error messages, however, is an under-examined aspect in the literature and hence the focus of our study. The most relevant work is Barik et al.'s survey of how program analysis tools present their output [2]. Yet, there is a crucial difference between the error messages programmers run into during regular execution of their code and the output from conducting a deliberate program analysis, often using a separate tool. In addition, Barik et al. acknowledged that the prior work they surveyed lacked user evaluations and connections to HCI research, two gaps we intend to bridge.

```
┤ EXCEPTION CAUGHT BY WIDGETS LIBRARY ├

The following assertion was thrown building TextField(controller:
TextEditingController#335e3(TextEditingValue(text: ⹂, selection: TextSelection(baseOffset: -1,
extentOffset: -1, affinity: TextAffinity.downstream, isDirectional: false), composing:
TextRange(start: -1, end: -1))), decoration: InputDecoration(hintText: "Type something"),
autocorrect: true, max length enforced, dirty, state: _TextFieldState#549bb)):

TextField widgets require a Material widget ancestor, but we couldn't find any.

Explanation
In material design, most widgets are conceptually "printed" on a sheet of material. In Flutter's
material library, that material is represented by the Material widget. It is the Material widget
that renders ink splashes, for instance. Because of this, many material library widgets require that
there be a Material widget in the tree above them.

Potential Fix
To introduce a Material widget, you can either directly include one, or use a widget that contains
Material itself, such as a Card, Dialog, Drawer, or Scaffold.

The specific widget that could not find a Material ancestor was:
    TextField(controller: TextEditingController#cd3e1(TextEditingValue(text: ⹂, selection:
    TextSelection(baseOffset: -1, extentOffset: -1, affinity: TextAffinity.downstream, isDirectional:
    false), composing: TextRange(start: -1, end: -1))), decoration: InputDecoration(hintText: "Type
    something"), autocorrect: true, max length enforced)

The ancestors of this widget were:
    Padding(padding: EdgeInsets.all(50.0))
    Column(direction: vertical, mainAxisAlignment: center, crossAxisAlignment: center)
    ExampleWidget
    Semantics(container: false, properties: SemanticsProperties, label: null, value: null, hint: null)
    Builder
    RepaintBoundary-[GlobalKey#fbb10]
    IgnorePointer(ignoring: false)
    FadeTransition(opacity: AlwaysStoppedAnimation<double>#1adfb(▶ 1.0; paused))
    FractionalTranslation
    SlideTransition(animation: AnimationController#30593(⬜ 1.000; paused; for
    ...

When the exception was thrown, this was the stack:
#0    debugCheckHasMaterial.<anonymous closure> (package:flutter/src/material/debug.dart:97:7)
#1    debugCheckHasMaterial (package:flutter/src/material/debug.dart:100:4)
#2    _TextFieldState.build (package:flutter/src/material/text_field.dart:456:12)
#3    StatefulElement.build (package:flutter/src/widgets/framework.dart:3737:27)
#4    ComponentElement.performRebuild (package:flutter/src/widgets/framework.dart:3646:15)
#5    Element.rebuild (package:flutter/src/widgets/framework.dart:3495:5)
#6    ComponentElement._firstBuild (package:flutter/src/widgets/framework.dart:3626:5)
#7    StatefulElement._firstBuild (package:flutter/src/widgets/framework.dart:3776:11)
#8    ComponentElement.mount (package:flutter/src/widgets/framework.dart:3621:5)
#9    Element.inflateWidget (package:flutter/src/widgets/framework.dart:2907:14)
#10   Element.updateChild (package:flutter/src/widgets/framework.dart:2723:12)
    ...

Another exception was thrown: A RenderFlex overflowed by 99361 pixels on the bottom.
```

Fig. 2: The "Spaces" variant of the error message.

2.2 Presentation Theories and Techniques

To improve the presentation of error messages, we wanted to achieve three specific goals: 1) making key information from a long message stand out, 2) communicating the relative importance of different elements within a message, and 3) making it easier to scan and skim the error output. To this end, we drew from a number of theories and techniques in HCI.

First, we draw from a visual perception theory called Semiology of Graphics (SoG), which Conversy applied to examining strengths and weaknesses of different code representations [3]. According to SoG, different visual variables of 2D marks, such as color, shape, position, size, etc., have different effects on visual perception. For example, color makes it easier to selectively focus one's attention on particular marks quickly, while differences in luminosity allows the user to rank marks from light to dark.

Second, we employed the law of proximity in Gestalt Principles [8]. It shows that humans perceive connections between visual elements when they are closer to one another, and separation when they are relatively far apart. One practical application of the law of proximity is to use whitespace to group visual elements into meaningful chunks.

Last, we applied the Progressive Disclosure technique widely used in GUI design [6]. According to Nielsen, "Progressive disclosure defers advanced or rarely used features to a secondary screen, making applications easier to learn and less error-prone." We believe modern IDEs (Integrated Development Environments) have the capabilities to enable progressive disclosure of information within an error message.

3 REDESIGNING AN ERROR MESSAGE

We applied the presentation theories and techniques described above to a representative runtime error message in Flutter, an open-source, multi-platform UI framework that was gaining popularity. The error, shown in Fig. 1 was run into by a user in a real-world situation and the message emitted was not helpful, to the extent that the user asked for help on Stack Overflow, an online Q&A forum for programmers. Using the snippet included in the user's post[‡], we reproduced the error and modified the resulting message into three variants described below.

3.1 The "Spaces" Variant

Applying the law of proximity [8] to the original error message led to this variant we called "Spaces" (see Fig. 2). Specifically, we made two modifications. First, we added an empty line both above and below the summary of the error "TextField widgets require a Material widget ancestor, but we couldn't find any." The extra whitespace makes the summary stand out. Second, we added subheadings such as "Explanation" and "Potential Fix", and additional line breaks to break up the long message into smaller, more skimmable sections.

Fig. 3: The "Colors" variant of the error message.

Fig. 4: The "Ellipses" variant of the error message.

3.2 The "Colors" Variant

To create the "Colors" variant (see Fig. 3), we analyze the relative importance of different pieces of information in the message and came up with a simple color coding scheme:

- Display the error summary in red.
- Display the object which the error is associated with in blue.
- Display detail that is usually not needed in gray. As Fig. 3 shows, "TextField" is displayed in blue, while its constructor's parameters are downplayed in gray.
- Display the rest of the message in the standard color (usually black).
- In addition, use boldface sparingly to emphasize certain information at the message author's discretion.

The goal of such color treatments is to make the most important information visually salient, so the user can selectively pay attention to useful parts of the message.

3.3 The "Ellipses" Variant

The "Ellipses" variant shows an application of the progressive disclosure technique in error messages (see Fig. 4). When we consider the capabilities of a modern IDE, we can add useful affordances within the error message to create layers of disclosure. For example, we collapsed the TextField widget's parameter list (line 2 of the error) and chain of ancestors beyond the immediate three (before the stack trace) to ellipses that can potentially expand to show the full text upon clicking. The resulting error message is considerably shorter than the original, increasing the likelihood of drawing the user's attention to higher-level elements in the message at a glance.

4 EXPERIMENTAL DESIGN

To compare the usability of the three variants with the original, we conducted an online experiment using a scenario-based questionnaire created in survey software Qualtrics. We recruited 52 mobile app developers from Flutter's user community in this study. All participants reported an experience level of novice or above with Flutter. Most of them were also actively using Flutter: 75% of participants reported using Flutter within the past week of the experiment.

The 52 participants were randomly assigned to one of four experimental groups. The original form of the error message was shown to the control group, while the variants were shown to the three treatment groups, respectively. Participants went through 6 main steps in the experiment outlined in Table 1.

Table 1: Experimental Procedure

Step	Participant Action
1	Read the code snippet in Figure 1 for up to 90 seconds.
2	Read one of the four versions of the error message for up to 30 seconds.
3	Describe what the error message was trying to say.
4	Propose a potential error fix. (The code snippet was shown again, for participants to use if they so desired.)
5	Choose whether to take another look at the error message for up to 15 seconds, and repeat steps 2, 3, and 4.
6	Compare a variant with the original error message and explain preference.
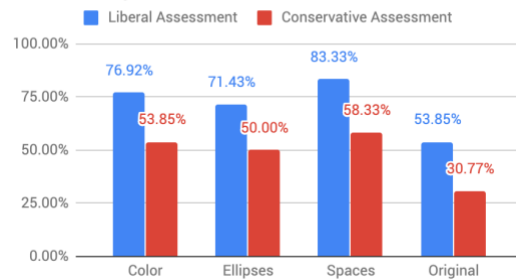
Figure 5: Error comprehension rates when up to 30 seconds were given to read the error message.

In the second step of the experiment, participants saw one of the four versions of the error message assigned to their group for up to 30 seconds. Why did we have this time limit? There were two reasons. First, this appeared to be a reasonable design goal, since a human expert was likely to take less than 30 seconds to explain this error based on the length of the accepted answer on Stack Overflow. Second, in a pilot study without any time limit, we found that some participants took an unrealistic amount of time to debug the error, perhaps debugging in an actual IDE.

We measured two outcome variables in the experiment: error comprehension and error resolution. The two authors independently coded participants' descriptions of what the error was about and their proposed modifications to the code into one of the three categories: "correct," "incorrect," or "uncertain." We assessed answers provided both within the 30-second limit and within the total limit of 45 seconds participants could use to examine the error message.

Sometimes, the participant's answer could be ambiguous or partially correct, which led to disagreements in our assessments. We resolved such differences in a mechanical way. When we consolidated our assessments of an answer "liberally", the final assessment was "correct" unless both authors considered the answer to be incorrect. In contrast, when we consolidated our assessments "conservatively," the final assessment was "incorrect" unless both authors considered the answer to be correct.

After consolidating two authors' assessments, we calculated the error comprehension rate for each experimental group as the number of participants with a correct explanation of the error divided by the total number of participants in that group. We calculated the error resolution rate in a similar manner using consolidated assessments of participants' proposed error fixes.

5 RESULTS

5.1 Error Comprehension

All variants outperformed the original in terms of error comprehension in both liberal and conservative assessments (see Fig. 5). In particular, the "Spaces" variant resulted in the highest accuracy rate, 29.48 percentage points (pp) higher than that of the original message in the liberal assessment when participants were given 30 seconds to read the error message. The advantage persisted when participants took another look at the error message for up to 15 seconds (see Fig. 6), after which the "Spaces" variant resulted in an impressive 91.67% error comprehension rate in the liberal assessment and 83.33% in the conservative assessment.

5.2 Error Resolution

The visually-enhanced variants also improved error resolution. They all strongly outperformed the original in our experiment (see Fig. 7 and Fig. 8). For example, under the 45-second time limit, the variants had an advantage between 15.38 pp and 28.21 pp in conservative assessments and between 23.08 pp and 44.87 pp in liberal assessments. Within the variants, the "Spaces" variant again resulted in the best performance, achieving 83.33% error resolution rate in the liberal assessment and 75% in conservative assessment under the combined 45-second time limit.
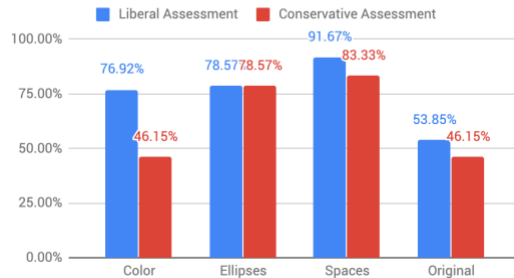
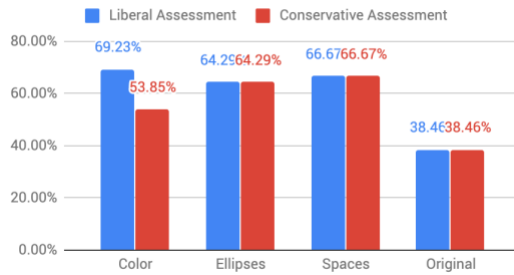**Figure 6: Error comprehension rates when up to 45 seconds were given to read the error message.**

Figure 7: Error resolution rates when up to 30 seconds were given to read the error message.
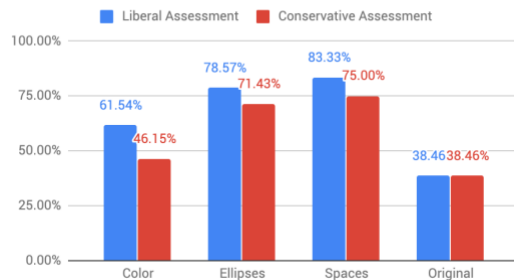

Figure 8: Error resolution rates when up to 45 seconds were given to read the error message.

5.3 User Preference

While most of our quantitative results did not achieve statistical significance due to small sample size, the risk of committing to type I error was greatly mitigated by self-reported preferences. In the experiment, all participants preferred to see one of the variants over the original. In open-ended comments, participants explained why they liked the variant better. Below are some examples (words in parentheses are added by the authors):

- "The error on the right (the 'colors' variant) is a huge improvement with the critical short message in red, the affected widget in blue."

- "Hiding the whole Widget object reduces clutter (in the 'ellipses' variant), makes it easy to find the reason the error occurred."

- "B (the 'spaces' variant) is much easier to parse. The sections are clearly broken up, so it is easy when skimming to know where to jump next."

6 CONCLUSIONS

Our experiment shows that small changes to the presentation of error messages can result in substantial improvements of error comprehension and resolution. Moreover, the findings suggest that visual perception techniques and theories can effectively guide innovations in error message usability.

REFERENCES

[1] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How Should Compilers Explain Problems to Developers? In *Proc. ESEC/FSE '18*, 633–643. https://doi.org/10.1145/3236024.3236040

[2] Titus Barik, Chris Parnin, and Emerson Murphy-Hill. 2017. One λ at a time: What do we know about presenting human-friendly output from program analysis tools? Workshop at *PLATEAU'17*.

[3] Stéphane Conversy. 2014. Unifying Textual and Visual: A Theoretical Account of the Visual Perception of Programming Languages. In *Proc. Onward! '14*, 201–212. https://doi.org/10.1145/2661136.2661138

[4] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proc. CHI '10*, 1019–1028. https://doi.org/10.1145/1753326.1753478

[5] Jakob Nielsen. 2001. Error Message Guidelines. *Nielsen Norman Group*. Retrieved January 1, 2019 from https://www.nngroup.com/articles/error-message-guidelines

[6] Jakob Nielsen. 2006. Progressive Disclosure. *Nielsen Norman Group*. Retrieved January 1, 2019 from https://www.nngroup.com/articles/progressive-disclosure

[7] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' Build Errors: A Case Study (at Google). In *Proc. ICSE '14*, 724–734. https://doi.org/10.1145/2568225.2568255

[8] Mads Soegaard. 2018. Laws of Proximity, Uniform Connectedness, and Continuation – Gestalt Principles (2). *The Interaction Design Foundation*. Retrieved January 1, 2019 from https://www.interaction-design.org/literature/article/laws-of-proximity-uniform-connectedness-and-continuation-gestalt-principles-2

[9] V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction* 2010: 1–26. https://doi.org/10.1155/2010/602570