Probes and Sensors: The Design of Feedback Loops for Usability Improvements

Luke Church
Computer Laboratory
University of Cambridge
luke@church.name

Emma Söderberg
Department of Computer Science
Lund University
emma.soderberg@cs.lth.se

Abstract

The importance of user-centric design methods in the design of programming tools is now well accepted. These methods depend on creating a feedback loop between the designers and their users, providing data about developers, their needs and behaviour gathered through various means. These include controlled experiments, field observations, as well as analytical frameworks. However, whilst there have been a number of experiments detailed, quantitative data is rarely used as part of the design process. Part of the reason for this might be that such feedback loops are hard to design and use in practice. Still, we believe there is potential in this approach and opportunities in gathering this kind of 'big data'. In this paper, we sketch a framework for reasoning about these feedback loops - when data gathering may make sense and for how to incorporate the results of such data gathering into the programming tool design process. We illustrate how to use the framework on two case studies and outline some of the challenges in instrumentation and in knowing when and how to act on signals.

Keywords: user-centred design, feedback, instrumentation, adaptive systems

1. Introduction

Programming changes over the lifetime of a system or language. At PPIG in 2018 we described early work in the diachronic study of notations (Church et al, 2018), especially of programming; that is, looking at how notations evolve overtime. However, it's not only the notations that change, the interfaces used for programming change more rapidly, and the APIs and frameworks even more rapidly still.

It's no longer necessary within most of the programming tooling community to argue for a user centric approach to these changes. Thanks to advocacy, including from the PPIG community, we know that we need to understand developers in order to motivate improvements to languages and their associated tooling.

But where does this understanding come from? Some of it comes from controlled experiments e.g. (Sime, Green, & Guest, 1973), some from field observations e.g. (Nardi, 1993), some of it comes from analytical frameworks e.g. (Blackwell, 2002). Despite a number of efforts, such as (Brown, Kölling, McCall, & Utting, 2014), comparatively little of it comes from empirical data from instrumentation in tooling.

This seems strange - it is, if anything, the era of 'big data', or as the advocates of the application of statistical techniques to improvement development call it 'big code'. In the past we've written about possible strategies for applying machine learning to development (Church, 2005), (Church, Söderberg, Bracha, & Tanimoto, 2016). These techniques relied on tools for gathering substantial amounts of data about developers. If we can build systems that use statistical aggregations of data to create adaptivity it should be possible to use data to inform design work on the iteration of our tools? We continue to see this as an opportunity.

This challenge of making this work effective, was succinctly summarized in (Lewis, 2016), where Clayton suggests that ‘measurement is considered hopeless’ due to the varied and intersectional nature of the design of programming context. Here we explore an alternative hypothesis - that it isn’t always hopeless, but it works in some places and not in others. In this paper we aim to describe some of those conditions.

We sketch a framework and use it to characterise a number of previous instrumentation systems built to support design feedback. We then use this framework to exemplify where instrumentation works and were it doesn’t and to share the lessons we’ve learnt along the way.

We’ll start by identifying the decisions that need to be made, both in the instrumentation system and in the larger feedback loop.

2. Framework: Sensors, Probes, and Feedback

2.1 What to Measure

In this paper we will separate the discussion into the instrumentation system, the technical component that senses information and records it in some manner, and the broader socio-technical feedback system that it is embedded within that includes decision making processes based on the feedback.

These two components have a chicken-and-egg relationship with each other. The design of the instrumentation system is affected by the types of feedback that can be operationalised, and the types of feedback that can be operationalised are determined by what can be measured.

There are two broad strategies that get taken: record everything in anticipation of unknown questions, and try and make sense of it later, and record information about specific questions in the knowledge that you will only want to answer that question.

These strategies also determine how much interaction we expect to have with users. Some data-capture systems operate completely autonomously, not interacting with users during the capturing of data, others require specific interaction from their users. Broadly we refer to these instruments as sensors (autonomous, no interaction with the user), and probes (have specific interactions). We’ll elaborate on these categories more shortly. We are not suggesting that either of these strategies is ‘correct’ - that depends on the circumstances.

2.2 Lengths of the Feedback Loop

There are a number of different things that can be done with measurements. At the tightest interaction loop, they could provide data that drives a self-adaptive system (Brun et al., 2009), responding immediately to the sensed data to adjust the interface and change what the next keystroke will do, for example like the smart text entry systems on mobile phones.

At completely the opposite end of the reactivity spectrum, the sensors might provide data that inform the next generation of programming language, often a matter of years after the data has been collected. The data can also be used summatively to assess how well something worked (did users accept the quick fixes the IDE offered?), or formatively to guide the design of the next generation of a system (Where do the majority of errors in code come from?).

2.3 Known and Unknown Problems

The final inter-tangled axis is whether the endeavour is to address a known problem, or an unknown one. For example, a known problem - and one we’ll discuss shortly - is that static analysis systems have false positive rates. An unknown problem might be closer to ‘improve the syntax of this language’. Of course this is not really a dichotomy, but rather a spectrum of how well pre-characterised the problem is.

2.4 How to Measure?

There are different places in which we could do the measurement. We characterise this by considering the 'distance' from the user. We could measure something about the user themselves, about the users' interaction with the computer, or about just the computer. The latter are studied by the systems communities, with distributed logging tools such as Kafka[1] or commercial products like StackDriver[2]. In this paper we'll focus on the first two cases, measuring programmers and their interaction.

2.4.1 Biometrics: Measuring the Programmer

Biometric sensors offer an opportunity to capture signals enabling detection of, for instance, task difficulty (Fritz, Begel, Müller, Yigit-Elliott, & Züger, 2014), programmer proficiency (Busjahn et al., 2015), and developer frustration (Müller & Fritz, 2015). There are a number of devices measuring biometric signals available, for instance, eye-trackers, electroencephalography (EGG) headbands, and wristbands measuring properties like temperature and blood volume. Data from such devices can be used on their own or in combination with each other to enable more complex detection.

In this setting, we are considering biometric sensors as passive collectors of information and not as devices for control. Especially eye-trackers are being explored as control devices in systems like EyeDE - eye-tracker controlled actions in an IDE (Glücker, Raab, Echtler, & Wolff, 2014), EyeNav - code navigation with eye movement (Radevski, Hata, & Matsumoto, 2016), and CodeGazer - eye-tracker controlled code navigation (Shakil, Lutteroth, & Weber, 2019). Sensors being used in this way have other, but sometimes related, kinds of challenges. For instance, eye-trackers have to deal with the so called "midas touch" issue, when gaze-based functionality is triggered without user consent.

Even though these sensors do not actively prompt a user for information they do incur other user costs; a user may have to wear the measuring device, or calibrate a device before it can be used. The user cost in this regard has been reduced in recent years with less intrusive free-standing eye-trackers[3], and non-sticky EEG measuring devices in the form of headbands[4]. Still, the user needs to commit to an additional device in their environment or on their person. The device needs to be considered useful enough to overcome that cost.

2.4.2 Instrumentation: Measuring the Interaction

Behavioural instrumentation systems are pieces of software running on the computer that a developer is using. The software observes one or more activities that the developer performs, and reports it to a server for archiving and analysis.

For example we have previously used this to study code completion behaviours (Mărășoiu, Church, & Blackwell, 2015). Others have aimed to build general purpose instrumentations similar to the platform used in Dynamo above. BlackBox (Brown et al., 2014) is an example of one such system used in the educational platform BlueJ.

Sensors like this offer the opportunity of capturing a very wide range of data for subsequent analysis. As demonstrated in the Dynamo case, they can be useful for determining behaviours which are not known at the time when the system is being originally designed.

A full discussion of the technical implementation strategy is beyond the scope of this paper, however we have learnt several lessons by building and operating a number of these tools, two of which are important for the rest of the discussion.

---

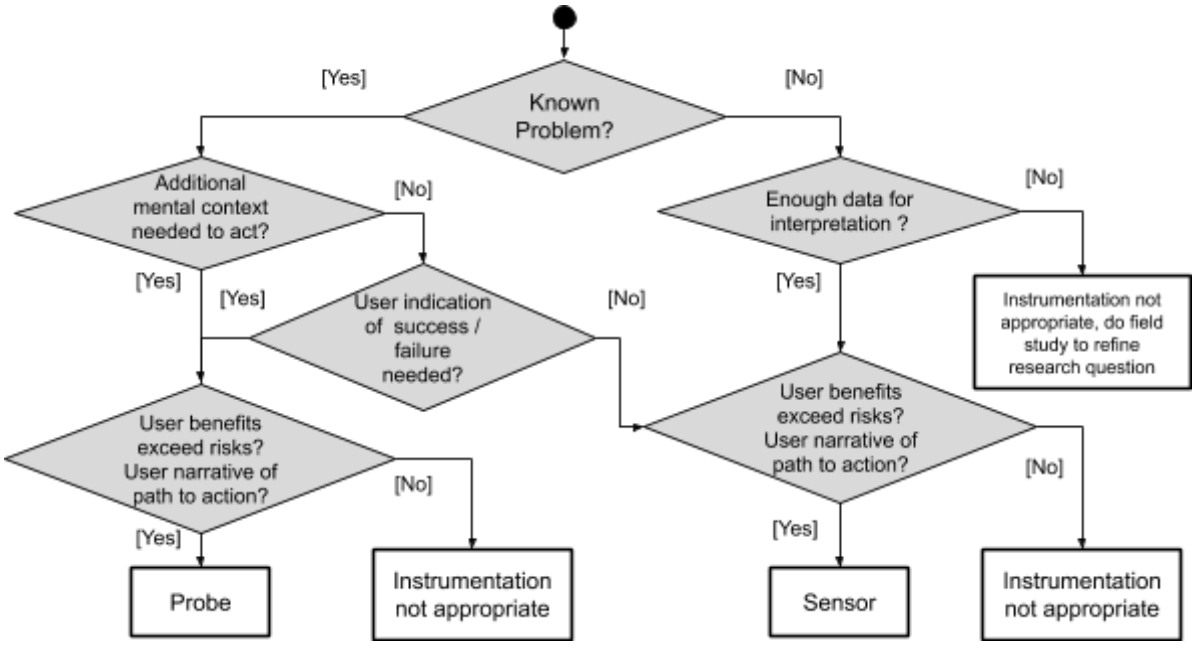
[1] https://kafka.apache.org/
[2] https://cloud.google.com/stackdriver/
[3] https://www.tobii.com/
[4] https://brainbit.com/

Firstly: It is economically feasible to build instrumentation systems that respond to user actions, but often not to system actions. For example, the first author built a system that recorded code completion requests within Eclipse (triggered by a user pressing '.'), however when the system was extended to record internal exceptions thrown by Eclipse (triggered by a system event), the data volume overwhelmed the infrastructure. This limits what can be effectively recorded.

Secondly: It is often very useful in analysis to be able to, as much as possible, recreate the context that the user was in when they performed an action. For example, what source code where they looking at? In order to build systems that are practically resilient to data loss problems such as network failure, it helps to send either all the information with each request, or to send periodic 'keyframes' from which

2.4 The Framework

With this structure, of known and unknown problems, opportunities for measurement, risks and returns and probes and sensors, we can sketch a decision process for deciding when and in what form to use instrumentation.



We've outlined a framework, now we're going to show you how to use it by applying it to two examples: Tricorder and Dynamo.

4 Case Studies

4.1 Tricorder

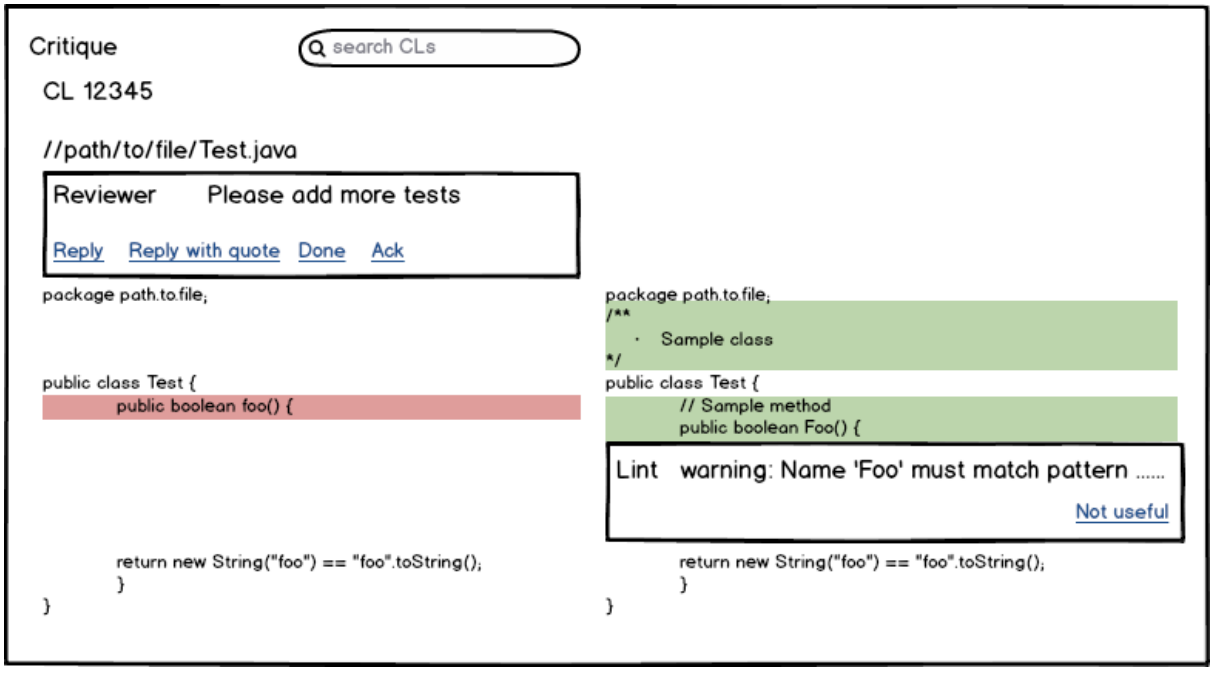


Figure 1: Example of a robot comment displayed on a line on a code review diff. The 'not useful' probe is integrated as a link in the bottom right corner.

We describe this case in terms of the system, the known problem, the probe, and the collected signal:

The System: As an example, we will consider Tricorder (Sadowski et al. 2015), a system running internally at Google which integrates results from program analyzers into tools used by software developers. The most prominent integration point is the code review tool Critique, where the system presents analysis results in the form of robot comments on code under review. That is, next to comments made by humans on diffs of code, comments from program analyzers are presented using a different color - grey is used for robot comments and yellow for human comments (as shown in Figure 1).

The Known Problem: False Positives: Use of program analysis tools is plagued with a history of problems with false positives (Johnson, Song, Murphy-Hill, & Bowdidge, 2013) causing users to not trust the results and eventually ignore them (Imtiaz, Rahman, Farhana, & Williams, 2019). An awareness of this problem has been central to the shaping of the philosophy behind the Tricorder system (Sadowski, van Gogh, Jaspan, Söderberg, & Winter, 2015), which emphasizes data-driven usability improvements and the collection of 'not useful' feedback for analysis results.

The Probe: 'Not useful' Links on Robot Comments: When users are presented with program analysis results as robot comments during code review, they are presented with the option of clicking a link named 'not useful', placed on each individual robot comment, shown in Figure 1. If a user decides to click the link, a non-obstructing, e.g., not in the middle of their vision, notification pops up asking if they would like to provide more details. If a user decides to do so, they click a link and are then directed to a partially filled in bug template on a bug filing page.

The Signal: The What and Why of a 'Not useful' Analysis Result: In collection of 'not useful' clicks, information of the analyzer category not found useful is collected, and if the user decides to

provide more information, details of the position of the analysis result in the code under review is collected and inserted into the bug template presented to the user.

The ‘what’ of the signal is thus collected in two steps, first just the analyzer category, for instance, a specific bug pattern detected by ErrorProne (“Error Prone,” n.d.), and second details of the code context in which the analyzer presented a result, that is, the version of the file under review and the position of the comment plus the message provided to the user. The ‘why’ of the signal is potentially answered in the second step where users provide a comment in the bug template to explain why the result was not found useful to them.

The second step also provides a forum to have a discussion about the reasoning behind the bug report to further investigate why the report was filed. In the Tricorder system this second step has been very useful in understanding why results are not found useful. An analyzer category may, for instance, get a lot of bugs filed against it due to incomprehensible messages in the analyzer result (Sadowski et al., 2015).

With numerous analyzer improvements due to the ‘not useful’ signal in Tricorder, it appears to be a well-chosen and successful signal. What can we learn from this example? Why does this signal work?

4.1.1 Discussion

Analyzing the ‘not useful’ signal with the properties from Section 2 in mind, we hypothesize as follows:

Activity/Elements That the cost of collecting feedback is lowered by using common at hand elements in the domain. In the context of code review, a domain where adding, reading, and replying to comments is the main activity and using mouse clicks to navigate, at least among comments, the collection cost is lowered by using elements like comments and clickable links. As a comparison, imagine a scenario where the user would be encouraged to leave feedback, but would be sent to a draft of an email program or a spreadsheet instead to manually enter ‘not useful’ click feedback.

Activity/Mindset That the mindset of the reviewer and reviewee in code review present especially fertile ground for explicit and detailed signal collection. Users are in a mode of inspection with the goal of scrutinising and improving the code under review (Sadowski, Söderberg, Church, Sipko, & Bacchelli, 2018). In this state of mind, with fresh mental models of the code in their mind ready for inspection, the effort for providing detailed feedback to an analysis result perceived as not useful is lower. For comparison, imagine being presented with a snippet of code outside code review together with an analysis result and then having to load in the mental models of the code to perceive the usefulness of the analysis results and to potentially provide an explanation as to why it is not useful.

Organization/Culture That the culture among the users of Tricorder, that is, among Google software engineers, encourages feedback (“If something is broken, fix it”) (“Alphabet Investor Relations,” n.d.). A cultural motivation may provide a constant micro encouragement to react to any annoyance and clink the link. As opposed to a culture where filing a bug would be frowned upon or even considered offensive.

System/Trust That users experience that the effort of filing a bug matters (return on investment, effort matters). The system needs to be responsive and act on the feedback. In the case of Tricorder, responsiveness is part of the philosophy in the analyzer contract, problems need to be addressed or an analyzer may be turned off. Unresponsive behavior for filed bugs would be one way of not following this usability contract. In addition to trust in the system, the system administrators can assume that there is no malicious intent from users within the same organization. In contrast, imagine having a system open to all users of the Internet.

Probe/Granularity That the probe is at the right granularity in terms of precision. The ‘not useful’ signal could probably be divided into different kinds of not usefulness, for instance, ‘I don’t understand’ or ‘this is incorrect’. However, we argue that this would require the user to make another

decision, that is, the user just decide to give feedback and now she also needs to decide which kind of feedback to give. The cost now becomes higher. To allow users to quickly provide a ‘not useful’ click, even if it means more than one thing, is a good trade off. This quick and perhaps coarse activity acts as a funnel, indicating usability problems and potentially also providing some answers to why for the cases where users decide to take the extra cost of filing a bug.

4.2 Dynamo

The System: Dynamo (“Dynamo BIM,” n.d.) is an open source hybrid textual/visual programming language, often used by architects in the design of buildings.

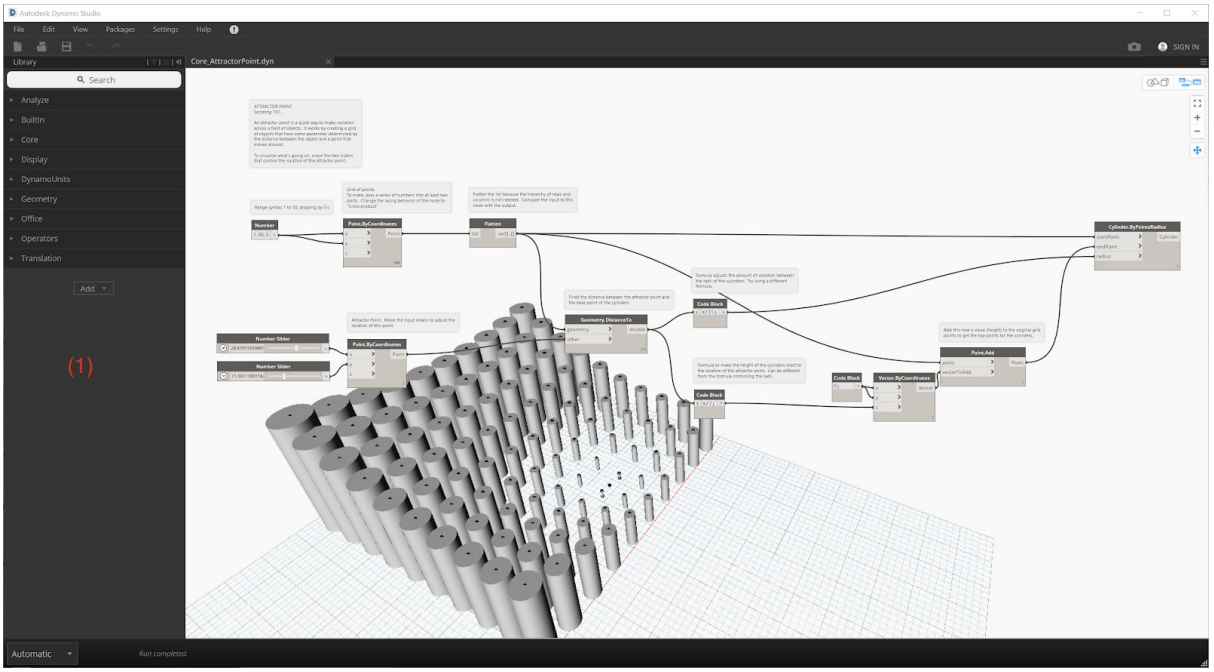


Figure 2: Dynamo in use create a cut out that might be used to pattern a facade on the side of a building.

The main Dynamo user interface, shown in Figure 2, consists of a canvas where ‘nodes’ are placed and linked together to make a graph. These nodes either directly represent expressions and function calls, or they contain short blocks of scripts where multiple expressions and function calls can be written together. Together these nodes make up a ‘workspace’. This workspace is the program that is executed. The result of executing that program is typically geometric and appears underneath the nodes.

Nodes get added to the canvas by the bar on the left labelled (1). This bar contains a hierarchical categorisation of the nodes by group. For example nodes to do with geometry are grouped together. Clicking on an element in the bar then adds it to the canvas. It also includes a mechanism for searching within the menu. For example typing in ‘point’ searches for items related to point, resulting in Figure 3:

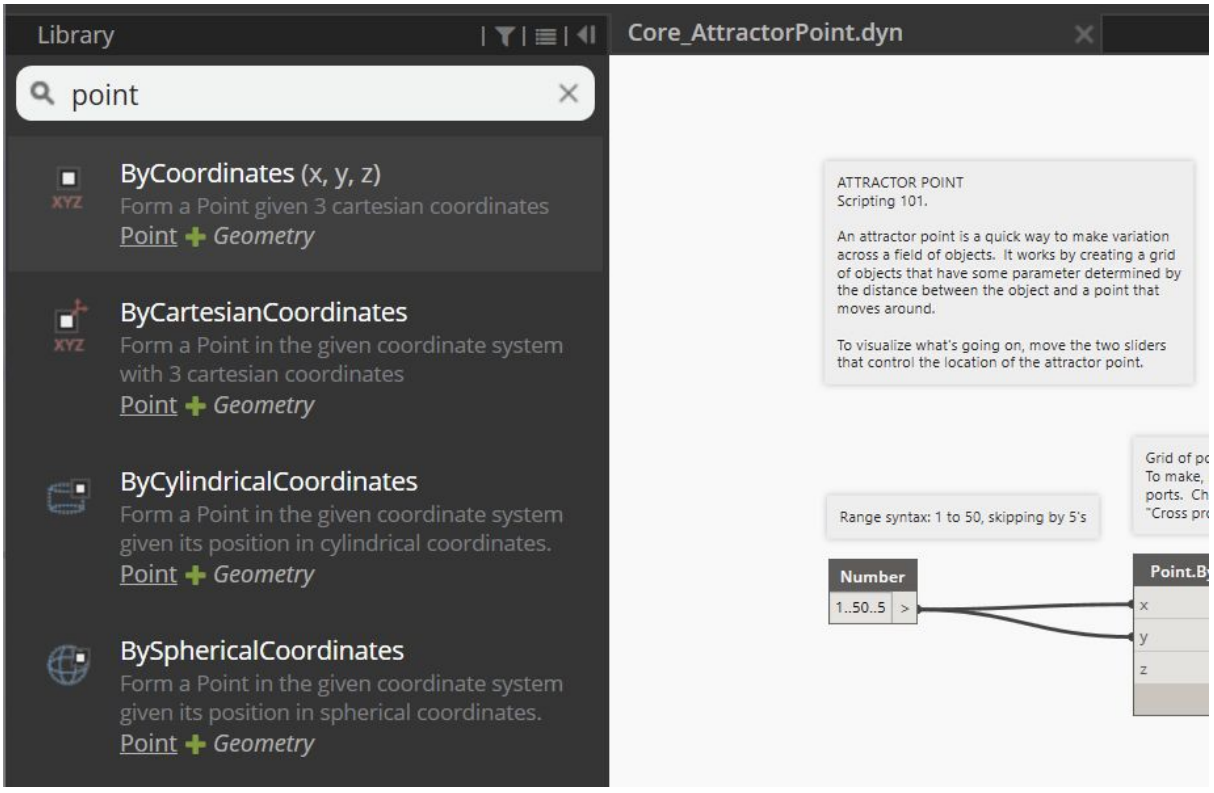


Figure 3: Dynamo's search window after searching for 'Point'

The Known Problem: **Searching for APIs** The design of the API for a system like Dynamo is hard. In the history of the Dynamo project a number of attempts had been to standardise the vocabulary in use in the API, however there are still a large number of cases where synonyms remain. For example, a user might think of making 'a cube by extruding a rectangle', or they might think of 'creating a solid from points'. They will need to use the search tool, in connection with other tools like documentation to find the API they want to use. This relates closely to the problem of using code completion functionality in a traditional IDE (Mărăşoiu et al., 2015)

The Sensor: Search Results: To address the search problem a sensor was added to the search mechanism. It records what users [who have opted-in] type into the search bar, associated timing information, and whether or not they add a node to the canvas from the search window.

The Signal: Search Effectiveness By using a combination of character-to-character editing models, and timing analysis, we can determine which queries lead to the successful addition of a node into the canvas, and what synonyms the users were trying on their way to getting there. From this we can define 'problem nodes' - that is nodes which users need to perform multiple searches to get to, and 'problem queries' - that is queries that don't usually result in a node being added. These can then be mapped by occurrence, so we can produce characterisation of severity and frequency of issues. The qualitative nature of the search queries helps in interpretation of what developers were looking for.

The [Mostly] Unknown Problem: **Visual Languages and Usability** As well as this specific problem of searching for APIs there is the more general concern. Visual programming languages are known to have significant usability difficulties (Green, Petre, & Bellamy, 1991). These issues were very well known to the team who built Dynamo, including the first author. We wanted to be able to empirically investigate the significance of these usability problems during the lifetime of the Dynamo project. This is of course a rather vague characterisation of a problem - but often one that design teams at the start of a practical project will be faced with.

The Sensor: Workspace Sampling To address the problem of the unknown usability improvement needs, Dynamo also contains a recording mechanism that takes a snapshot of the nodes and connectivity - but not the data flowing through them - in the workspace for users that have opted in. This snapshot is taken once per minute by using a mechanism that is derived from the file serialisation format.

The Signal: Dychronic Graphs The signal from the workspace sensor is a sample of the evolution of graphs over time. These can then be used to analyse the usage either synchronically, for example: 'what is the most commonly used node?' 'in graphs which use nodes from Revit, are nodes from Dynamo's native geometry library also used?', or dycronically, for example "how is usage of Code Block Nodes changing?"

However, as is typical in purely sensor based applications - there is a lack of context associated with the data that can result in it being challenging to interpret. We know a lot about what users of Dynamo were doing, but can only infer why they were trying to do it from their actions. Because the information about what they were trying to do is only in their minds, there wouldn't be any way of gathering this using a sensor - it would have needed a probe, and a fairly demanding one in terms of user time: explicitly asking the user what they were doing.

4.1.1 Discussion

Activity/Mindset As suggested above, the two cases here - search and workspace sampling have very different activities - and understanding of what is going on. The activity in search is fairly specific, the user is searching for a node to add to the canvas, whereas in general development it is unknown. This directly affects the utility of the analysis that is possible.

Utility: Both signals, of search effectiveness and graphs, have proven practically useful in providing a feedback mechanism from the users of Dynamo to the designers. The search effectiveness analysis has been used to both select synonyms to add to the search system as well as to evaluate proposed changes to the algorithm.

The workspace sampling has been used to evaluate the success of features (e.g. investigating how much impact the List at Level feature[5] has had on the usage of an alternative feature that known usability difficulties (partial function application).

Interpretation: The lack of an explicit probe mechanism by which a user of Dynamo could indicate whether the behavior they have just experienced was one they had desired makes the data from both of these sensors more difficult to interpret. However, it does carry a minimum cost to the users.

The data from the search sensor has been considerably easier to interpret in practice, requiring only a few days of data scientist time to analyse into results that have been used to directly influence the design. The data from the workspace sampling system has been more difficult to interpret. The primary reason for this was the difficulty of understanding from the graphs what the developer was trying to do, and then turning that understanding into a query that could be applied automatically to a large number of graphs.

Missing Measurements: Despite the generality of the sensor for recording the workspaces, some data was missed, creating further challenges in the analysis. For example, a workspace can reference functions defined in other workspaces. If these were not loaded into the editor, they would not have been sampled by the sensor.

Sampling Bias Caused by Opt-in: The data that is recorded by the sensors is clearly sensitive, and requires explicit consent from the users. This is provided by the user checking an extra box agreeing to send the data. This process may result in a bias caused by, for example, users working on commercial projects not opting in to instrumentation.

[5] https://dynamobim.org/introducing-listlevel-working-with-lists-made-easier/

5 Discussion

What can we learn from these two examples? The addition of empirical feedback loops has been useful in both cases, with the purpose ranging from disabling naughty analysers that cry wolf too often, to fixing discoverability issues in locating some nodes.

However, we can also see that the viability of a particular solution is dependent on the circumstance. The use of Critique is part of a highly normalised workflow, which means that it is possible to automatically store and reproduce all the context of a code review. This helps for looking at being able to empathise with the user of the tool. Usage of the Dynamo canvas is much less normalised, as it is used by a wide variety of different organisations for different tasks, so it's harder to know the context just from the data that has been recorded.

The change in the interpretability of the context is not the only thing that is contingent on the circumstances; so is the motivation to engage with the system probe or sensor in the first place. The code review system rests on the users taking the time to interact with the system, in the knowledge that they are doing so in a context where the results of their interaction will be looked at, and so a broader community will benefit. As a single organisation, that social context can be explicitly built by Google (see for example the code of conduct ("Alphabet Investor Relations," n.d.), a similar community spirit exists within users of Dynamo and constructing it has been an important part of design discussion (Kron, Personal communications)

Contextualising Trust

However as we can see this trust is contextual - the Tricorder system and its integration with the Critique code review system are both built and used by Google software engineers - consequently the developers using it feel assured that the incentives align effectively to making sure that their data will be held securely, as the same organisation that builds the system would be directly harmed by any problems that arose. The Dynamo case is considerably more complex, the majority of its users will not be employed by the same organisation that provided the infrastructure. Whilst there have been no security incidents associated with the system, the trust relationship is considerably different.

In the Tricorder/Critique context, the user trusts that their time is not being wasted; the data they provide will in fact be used to benefit themselves and a broader community. In other words, they trust that the system will behave as intended and will do what the user has requested. In the Dynamo case, the user doesn't pay any time cost after they have opted-in, however they trust that the data will be held and used responsibly.

Meaningful engagement with the system comes down to a couple of properties, firstly trust that the data will be held and used appropriately, however more subtly a trust that the data will be actioned so that the risk (of IP exposure, abuse use of data etc. in sensors, or of the above and time wasting for probes) is worth the engagement.

The key questions for each case then is how to interpret the information to support a feedback loop, how to take action on the basis of that information.

Interpretation and Action

Understanding the signal from a sensor or a probe is not usually trivial. The data in itself often feels impoverished to look at, as it's missing the extra context that you would have in an in-person study - notably asking the user what it was they were doing!

In the absence of these direct observations, it's tempting to try and use metrics to determine use. For example, it's common practice in technology development organisations to measure feature usages, similar to the usage signal from Dynamo above.

However even with a simple metric like usage, there are multiple possible causes that could result in an increased usage. It could be that the feature is serving user needs effectively, so people are choosing to use it more. Alternately it could be that the feature is very difficult to use, so users are needing to interact with it more in order to achieve the effect they need.

Another option that has an initial appeal is visualisation. If we could visualise the patterns of interaction, that might provide a way of understanding what was happening. We have used this technique before in (Mărăşoiu et al., 2015). However it doesn't solve the problem, whilst visualisation provides a mechanism for seeing highlighting distributional patterns, the structure of the visualisation still determines what can be seen and what can't.

As a heuristic, we have found that quantitative methods can be used effectively in cases where either (1) they are used in a mixed-methods approach to determine how much a phenomenon that has been observed generalises, or (2) where they are coupled to a specific action that the designer can take on the basis of the feedback, such as disabling a misbehaving analyser.

Self-adaptive Systems

One fairly extreme design strategy is to build *self-adaptive systems*, rather than waiting for the socio-technical processes to take place, where a researcher looks at the results and makes a design, the system adapts by itself. Systems that otherwise would be open-loop (involving a person in the design loop) are designed to be adaptive by incorporating closed-loop feedback to enable automatic adaptation (Brun et al., 2009) without a person involved. The general reasons for development of self-adaptive systems is to make it manageable to deal with a changing environment, changing requirements, and uncertainty. The design of the feedback loop which is central to the design of the system.

This centrality of the feedback mechanism is shared with the framework outlined in this paper. In the framework, there should be a narrative to the user connecting the action that may be the result of feedback they provide, and in an self-adaptive system the feedback loop, or control loop, should be known to the user (Brun et al., 2009).

The majority of self-adaptive systems are not end-user applications, but rather systems controlling, for instance, the numbers of VMs spun up for a cloud application to handle different kinds of load. There has been some early exploration of self-adaptive interfaces (Innocent, 1982) and examples can be found in, for instance, adaptive text input applications in mobile phones.

We're not aware of examples of systems of closed-loop self-adaptive systems for use by developers. Many years ago we experimented with using the Dasher interface to build a self-adaptive system based on a statistical model for text entry (Church, 2005). It had an awkward tendency to go into what would now be described as 'extreme over-fitting', paying rather too much attention to what the developer did last and repeating that as suggested code.

Whilst the system was unbearable to develop with, it at least suggested that there is an alternative counterpoint; a feedback cycle to the developer that was too close and too obvious!

6 Conclusions

We started this paper discussing Clayton's worry that "measurement was hopeless". Much of his concern was that the variation in the users, and the reasons why they are programming, create variation that swamps the quantitative measurements that can be taken. This, combined with the problem that the quantitative methods don't give data that can be used to guide formative processes makes measurements in controlled experiments a very difficult approach to take to build design.

This paper represents a slightly alternative perspective. Whilst we agree with many of Clayton's critiques in the context of controlled experiments, we suggest that automatic measurement (sensors)

and semi-automatic measurement (probes) can indeed be turned into useful feedback mechanisms to drive design.

They offer the ability to measure users in the wild, as they go about doing their wide range of varied tasks, so have strong external validity. These tools can scale to measure entire populations - making substantial inroads to the problems of tracking variation. By careful design and analysis we've shown examples where measurements have been practically useful.

They are by no means a panacea. We've shown that involvement in these systems from the users' perspective entails some risk - from having their time wasted, to security and perspective risks. It's easy to see how users can become rightly sceptical about doing this in exchange for platitudes about product improvement, they expect something concrete in return, or at the very least a concrete narrative about how the designer of the system is proposing to use the data to get there.

Looking ahead, these feedback loops in non self-adaptive systems aren't obvious. This is the case in product development, but is also the case in research. How will taking part in a study on C++ help make someone's future better?

As a community we've accepted that understanding users is needed, but there's future work to do in improving how we talk about how we'll use the feedback loops that we're creating both in our experiments and in our instrumentation systems to improve developers' experiences.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

Alphabet Investor Relations. (n.d.). Retrieved June 28, 2019, from Alphabet Investor Relations website: https://abc.xyz/investor/other/google-code-of-conduct

Blackwell, A. F. (2002). First steps in programming: a rationale for attention investment models. *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 2–10.

Brown, N. C. C., Kölling, M., McCall, D., & Utting, I. (2014). Blackbox: a large scale repository of novice programmers' activity. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 223–228. ACM.

Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., … Shaw, M. (2009). Engineering Self-Adaptive Systems through Feedback Loops. In B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, & J. Magee (Eds.), *Software Engineering for Self-Adaptive Systems* (pp. 48–70). Berlin, Heidelberg: Springer Berlin Heidelberg.

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., … Tamm, S. (2015). Eye Movements in Code Reading: Relaxing the Linear Order. *2015 IEEE 23rd International Conference on Program Comprehension*, 255–265.

Church, L. (2005). *Introducing #Dasher, A continuous gesture IDE, A work in progress paper*. Retrieved from http://www.ppig.org/sites/ppig.org/files/2005-PPIG-17th-church.pdf

Church, L. Marasoiu, M (2018): *A growing tip or a sprawling vine: How software grows.*

Church, L., Söderberg, E., Bracha, G., & Tanimoto, S. (2016). *Liveness becomes Entelechy - A scheme for L6*. Retrieved from https://ai.google/research/pubs/pub45596

Dynamo BIM. (n.d.). Retrieved June 28, 2019, from https://dynamobim.org

Error Prone. (n.d.). Retrieved June 28, 2019, from http://errorprone.info/

Fritz, T., Begel, A., Müller, S. C., Yigit-Elliott, S., & Züger, M. (2014). Using Psycho-physiological Measures to Assess Task Difficulty in Software Development. *Proceedings of the 36th International Conference on Software Engineering*, 402–413. New York, NY, USA: ACM.

Glücker, H., Raab, F., Echtler, F., & Wolff, C. (2014). EyeDE: gaze-enhanced software development environments. *Proceedings of the Extended Abstracts of the 32nd Annual ACM Conference on Human Factors in Computing Systems*, 1555–1560. ACM.

Green, T. R. G., Petre, M., & Bellamy, R. K. E. (1991). *Comprehensibility of visual and textual programs: A test of superlativism against the "match-mismatch" conjecture*. Retrieved from http://dx.doi.org/

Imtiaz, N., Rahman, A., Farhana, E., & Williams, L. (2019). Challenges with Responding to Static Analysis Tool Alerts. *Proceedings of the 16th International Conference on Mining Software Repositories*, 245–249. Piscataway, NJ, USA: IEEE Press.

Innocent, P. R. (1982). Towards self-adaptive interface systems. *International Journal of Man-Machine Studies*, *16*(3), 287–299.

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why Don'T Software Developers Use Static Analysis Tools to Find Bugs? *Proceedings of the 2013 International Conference on Software Engineering*, 672–681. Piscataway, NJ, USA: IEEE Press.

Lewis, C. (n.d.). *Methods in user oriented design of programming languages*. Retrieved from http://www.ppig.org/sites/ppig.org/files/2017-PPIG-28th-lewis.pdf

Mărășoiu, M., Church, L., & Blackwell, A. (2015). *An empirical investigation of code completion usage by professional software developers*. Retrieved from http://www.ppig.org/sites/default/files/2015-PPIG-26th-Marasoiu.pdf

Müller, S. C., & Fritz, T. (2015). Stuck and Frustrated or in Flow and Happy: Sensing Developers' Emotions and Progress. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, *1*, 688–699.

Nardi, B. A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA, USA: MIT Press.

Radevski, S., Hata, H., & Matsumoto, K. (2016, October 23). *EyeNav: Gaze-Based Code Navigation*. https://doi.org/10.1145/2971485.2996724

Sadowski, C., Söderberg, E., Church, L., Sipko, M., & Bacchelli, A. (2018). *Modern Code Review: A Case Study at Google*. Retrieved from https://ai.google/research/pubs/pub47025

Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., & Winter, C. (2015). Tricorder: Building a Program Analysis Ecosystem. *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, 598–608. Piscataway, NJ, USA: IEEE Press.

Shakil, A., Lutteroth, C., & Weber, G. (2019). CodeGazer: Making Code Navigation Easy and Natural With Gaze Input. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 76:1–76:12. New York, NY, USA: ACM.

Sime, M. E., Green, T. R. G., & Guest, D. J. (1973). Psychological evaluation of two conditional constructions used in computer languages. *International Journal of Man-Machine Studies*, *5*(1), 105–113.