Learning to Listen for Design

Elisa Baniassad, Ivan Beschastnikh, Reid Holmes, Gregor Kiczales, Meghan Allen
ebani@cs.ubc.ca,bestchai@cs.ubc.ca,rthomes@cs.ubc.ca,gregor@cs.ubc.ca,meghana@cs.ubc.ca
Department of Computer Science
University of British Columbia
Vancouver, Canada

Abstract

In his essay, *Designed as Designer*, Richard Gabriel suggests that artifacts are agents of their own design. Building on Gabriel's position, this essay makes three observations (1) Code "speaks" to the programmer through code smells, and it talks about the shape it wants to take by signalling design principle violations. By "listening" to code, even a novice programmer can let the code itself signal its own emergent natural structure. (2) Seasoned programmers listen for code smells, but they hear in the language of design principles (3) Design patterns are emergent structures that naturally arise from designers listening to what the code is signaling and then responding to these signals through refactoring transformations. Rather than seeing design patterns as an educational destination, we see them as a vehicle for teaching the skill of listening. By showing novices the stories of listening to code and unfolding design patterns (starting from code smells, through refactorings, to arrive at principled structure), we can open up the possibility of listening for emergent design.

ACM Reference Format:
Elisa Baniassad, Ivan Beschastnikh, Reid Holmes, Gregor Kiczales, Meghan Allen. 2019. Learning to Listen for Design. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '19), October 23–24, 2019, Athens, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3359591.3359738

1 Voice

In his essay, *Designed as Designer* [5], computer scientist and poet Richard Gabriel lays out the idea that an artifact being designed gives rise to its own design, and that if someone is good enough at "listening" to the thing they are designing, they will be guided by the artifact itself to its own natural design[1]. In all of Gabriel's examples, he points to aesthetic structural imbalances that need to be fixed, rhymes that need to be repaired.

The emergence of refactoring [4] as a primary design method tells us that strong programmers intuitively listen to their code to arrive at good, evolvable, design. The skill of listening is what separates the advanced, wise, programmer, from a technician. Gabriel implies that code speaks to the programmer the same way a poem speaks to the poet. Just as artists look for asymmetry, imbalance, or lack of rhythm, developers look for deficiencies in their code as they take the step of attaining good design.

Gabriel does not lay out explicitly how developers might hear code, and how code, line by line, provides its signals to the programmer. So we began to explore this idea. We thought about how code speaks to the programmer when they are making a code change to their system, and what the programmer does in response. Maybe code is sending us signals that describe missing abstractions – all those abstractions that would have made the programmers' life easier for making this change. Missing abstraction are manifested in code constructs that violate some of the foundational principles of software design, for example (from the Pragmatic Programmer's Quick Reference):

> ***Eliminate Effects Between Unrelated Things.***
> *Design components that are: self-contained, independent, and have a single, well-defined purpose.*

Looking at the finest granule of a missing abstraction, and how it is signalled, we begin to see that *code smells* are a possible *voice of the code*: Code smells are the code-analog to the missing balance of a painting, or the too-short phrase in a poem, or the discord in a song. They are small things that stand out and speak volumes. Almost all code smells signal missing abstractions. A *magic number* is asking to be abstracted into a symbolic constant; a *data clump* is asking to be abstracted into a data structure; a *type switch* is often asking to be abstracted into a type hierarchy; *divergent changes* ask for affected code to be abstracted into a new class; an *inline comment* is asking for its code to be abstracted into a method; *one class doing the work of two* is asking for the "extra" behaviour to be abstracted into a class.

Onward! '19, October 23–24, 2019, Athens, Greece

ACM ISBN 978-1-4503-6995-4/19/10…$15.00
https://doi.org/10.1145/3359591.3359738

[1] Editors, commentors, and reviewers play the role of an extra set of ears, to listen to the artifact and interpret what it is saying.

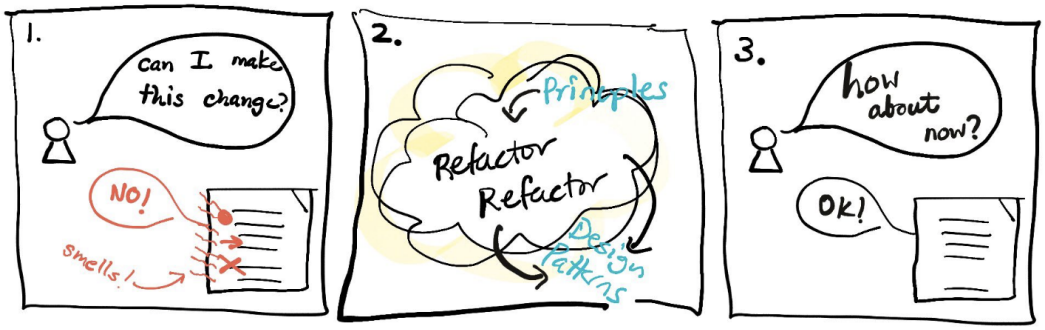


Figure 1. Seasoned programmer's conversation with code.

So if code speaks through smells which themselves are often concrete manifestations of design principle violations, then programmers respond with refactorings. By refactoring, a programmer introduces those abstractions that are missing – providing those missing hooks – to make the change that the code smells are screaming out for. Programmers refactor and refactor until they have reached a good design: a design that can accommodate their change (Figure 1). And in this way the code has itself 'spoken' to the programmer by signaling about problems with its current concrete form that can be changed to emerge at a better design.

But programmers aren't constantly refactoring, so when do they listen? Well, code that is not being changed is not that interesting to listen to. If we are not trying to alter code, there is no reason to talk to it. It is like iterating over a painting you are not going to look at again – why search for imbalance in something that is never going to be viewed? Change is certainly not the *only* time code talks to a programmer, but it is certainly a time when it screams in protest. When trying to make a change to code (especially a change that was unplanned, or not *designed for up front*) the code will yell back: *"noooooooo!!!!!!"*. This aligns with the agile approach: programmers do not just refactor all the time for kicks, and they do not fix every code smell. Wise programmers refactor when they hear the outcry from the code in response to it being asked to make a change. We refactor so that the change will result in less screaming and ultimately a more evolvable design.

2 Hearing

Good programmers listen to their code with wisdom. They see the signals, and hear the discord, and contextualise it in a depth of expertise: they filter what the code is saying through years of contemplation of the principles of design and expert solutions that have worked in the past. They bring with them a wealth of perspective and an arsenal of tools.

And with this wisdom and arsenal, programmers listen to the code, and they interpret what the code is saying by abstracting its 'words' into phrases and concepts which are, in fact, design principle violations. Programmers do not just look at code and in a rote way say "this class is doing the work of two classes." Programmers make interpretations, and so abstract this into "this is a violation of the single responsibility principle." They do not just look at a switch on type: they think "oh no, I am relying too deeply on an implementation here – this is a dependency inversion violation." They listen to code smells, but they *hear* in principles.

With each refactoring, the programmer is easing their way to a more principled design.

3 Emerging

With some program semantics, the code may, smell by smell, principle by principle, refactoring by refactoring, talk itself into an arrangement we have identified as a Design Pattern [6]. Design patterns emerged, and were devised in just this way: by wise programmers, listening to their code, hearing principles, and arriving at these well-formed solutions. Exploiting refactoring support, a wisely-listening programmer can arrive at a well-formed design, and indeed even a named and catalogued design pattern serendipitously. They can approach it, incrementally, by listening to the code, and hearing where the principles are crying out for repair.

Patterns as named constellations have value as a language in and of themselves. The provide us with a way to communicate about our programs without remaining mired in their details. But it is fine and even natural that they are names and constellations that emerge from reworking the code, rather than names that have been imposed upon it. One of the cautions with patterns is applying the wrong variant, perhaps one that contains too much heavy abstraction for the problem at hand. Thinking about patterns first runs this risk. And coming back to Gabriel's essay, if the writer/painter/programmer imposes their will too strongly on the artifact, its voice will be muted. The wise programmer

listens for patterns, instead of forcing them into code where they might not be appropriate.

Patterns as emergent structures, as opposed to prescriptive structures, agrees with Gabriel's view of emergent design: He points out that if the artist is too dominating in their approach, or too fixed in their design mindset, the artifact will be silenced, and will not emerge to a good design. This suggests that the programmer working on an artifact must develop code listening skills as a primary capability, without which they will not achieve design success.

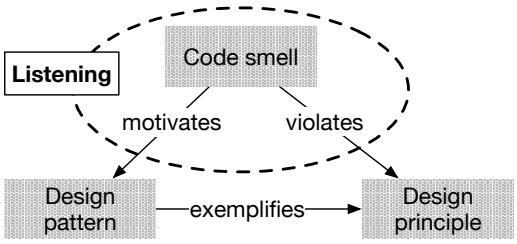


Figure 2. Integration of code smells, design patterns, and design principles into a single conceptual structure.

4 Teaching

How do we teach novices to become wise programmers? Time is, of course, a key ingredient, and certainly the traditional assumption is that it takes years to learn the skill of wise, principled listening. But can we, as educators, help?

Teaching computer science students does not begin with instruction into code smells, obviously. The first step is necessarily making code in the first place: seeding the starting artifact. In How to Design Programs (HtDP), Felleisen et al. put forward a systematic approach for obtaining a well-formed seed [3]. We use this approach in our introductory programming courses and build on this foundation through our software engineering curriculum. Via HtDP, students learn how to design functions through data-driven templates. Once they have an understanding of how the structure of data can shape the structure of a function that operates on that data, they are introduced to new techniques for function design. They learn how to combine design strategies, which results in functions that use and combine recognizable patterns. Our students are still novice programmers at this stage but are learning to recognize patterns and can articulate why each piece of code was included. The ability to reason about design strategies and patterns lays the foundation for our introductory students because they begin to think about their code at a granularity that is finer than a function and coarser than a single line. This foundation prepares students to listen to code smells as they progress through our software engineering curriculum.

HtDP has systematised the approach for forming that initial artifact. So can we systematise the emergent/listening design approach by grounding it in the teaching of code smells, principles, and design patterns in follow-on software engineering courses?

It is absolutely the case that these three elements (principles, patterns, and smells) do, in the minds of experts, feed into one another, in terms of practice, intuition, and form through refactorings and agile methodologies. But in education, these three concepts are often taught separately, in isolation from one another. Code smells are taught by example, with examples of refactorings that can heal them. Principles are taught with counter-examples, and with the code that addresses the concern. Patterns are traditionally taught in the Gang of Four patterns style: by outlining the intent behind the design pattern, and then showing how the prescriptive structure satisfies that intent.

What if we tell a more interwoven story of code smells, principles, and patterns? For example, by choosing a principle like SOLID [8] and integrating it with relevant smells and design patterns. In doing this, our aim is to help students learn and eventually internalise a powerful conceptual framework (Figure 2) that relates the three concepts. We can link an intended change through to problematic code smells, which violate basic design principles. Then, we can apply a refactoring to address the code smell, the choice of which is motivated by some design principle violation. By applying these refactorings, we can demonstrate that design patterns (and good designs generally) emerge from code by resolving design principle violations.

This has a satisfying endpoint: by telling a story of a design pattern as emerging from code, we use design patterns not as an end unto themselves, but instead as examples of principled design. We can tell story after story of code evolving to encode design patterns by refactoring code smells that would scream back if you tried to make that particular change without them, and show the principles that are violated that cause the cacophony.

The Composite Pattern Story. Imagine the toy-version of the listening-based story we might tell about the composite pattern. We start with the fairly nice, readable piece of code in Figure 3. We might then explain that if we wanted to add another kind of element (video, for instance) to the hierarchy, we would have to make a change to this code in three places: the declarations (a new list), the print method (a new loop), and the display method (another new loop). This change is shown in Figure 4. The code is now starting to talk back to us. It is trying to say that this change is delocalised because of a lack of abstraction. The for-loops are the young beginnings of a smelly *de facto* switch on type. And through our wise programmer lens, we can see that we have a lack of obliviousness that feels like a dependency inversion violation. The canonical response to the switch on type code smell is to introduce a hierarchy. We would introduce a type that served as an abstraction over `Description`, `Topic` and

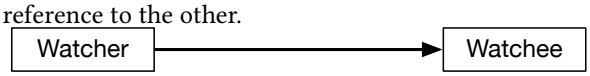
```
class Topic {
    ArrayList<Description> descriptions...
    ArrayList<Topic> subtopics...
    ...
    public void print(){
        for (Description d : descriptions){
            d.print();
        }
        for (Topic t : subtopics){
            t.print();
        }
    }

    public void display(){
        // same pair of for-loops,
       // except calling display instead of print
    }
}
```

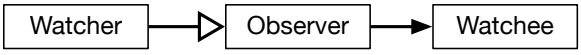
Figure 3. Small program iterating over a topic hierarchy.

`Video`. Adding this abstraction, shown in the code in Figure 5, would solve the dependency inversion problem, the switch on type code smell (presenting as multiple for-loops), and graduate us to the Composite pattern.

The Observer Pattern Story. The story to get to the Observer pattern has more twists and turns, because it is a story of two halves (the Observed and the Observer). The story again begins with code riddled with smells. In this case we have a watcher and a watchee, each of which is maintaining a reference to the other.



There are several ways that trouble can start, and the journey is what guides us to the variant of our solution. If we wanted to add another kind of watcher, then we would see duplication between the two watchers, and would see a switch on type in the watchee, when trying to update each of the kinds of observers. So we would solve that by introducing a higher-level type to both pull up the duplication at the point of observation, and provide an abstraction over which the watchee could loop obliviously. At this point we have reached a very minimal version of the pattern, with abstraction only present on the Observer side.



The revelatory process has a positive outcome: Rather than prescriptively applying some specially chosen variant of a pattern, the developer lets the code talk them through to a minimal yet solid (pun intended) set of design cues. If there was another kind of Subject, meaning there was duplication between Watchee types, we would have pulled up that duplication into a higher level type, and then would have arrived at the four collaborators of the classic version of the pattern. By promoting listening and responding, we can systematise a concept that is hard for novice developers to internalise: to employ "good design" but not to "over design".

Educators could use new visualisations of the stories of patterns that depict the issues in code as giving rise to new abstractions (such as that sketched in Figure 6).

```
class Topic {
    ArrayList<Description> descriptions...
    ArrayList<Topic> subtopics...
    ArrayList<Video> videos...
    ...
    public void print(){
        for (Description d : descriptions){
            d.print();
        }
        for (Topic t : subtopics){
            t.print();
        }
        for (Video v : videos){
            v.print();
        }
    }

    public void display(){
        // same three for-loops,
       // except calling display instead of print
    }
}
```

Figure 4. Now with videos!

```
interface CourseContent {
    public void print();
    public void display();
}
class Topic implements CourseContent{...}
class Video implements CourseContent{...}
class Description implements CourseContent{
    ArrayList<CourseContent> elements...
    ...
    public void print(){
        for (CourseContent e : elements){
            e.print();
        }
    }

    public void display(){...}
}
```

Figure 5. Refactored code from Figure 4 after introducing a hierarchy.

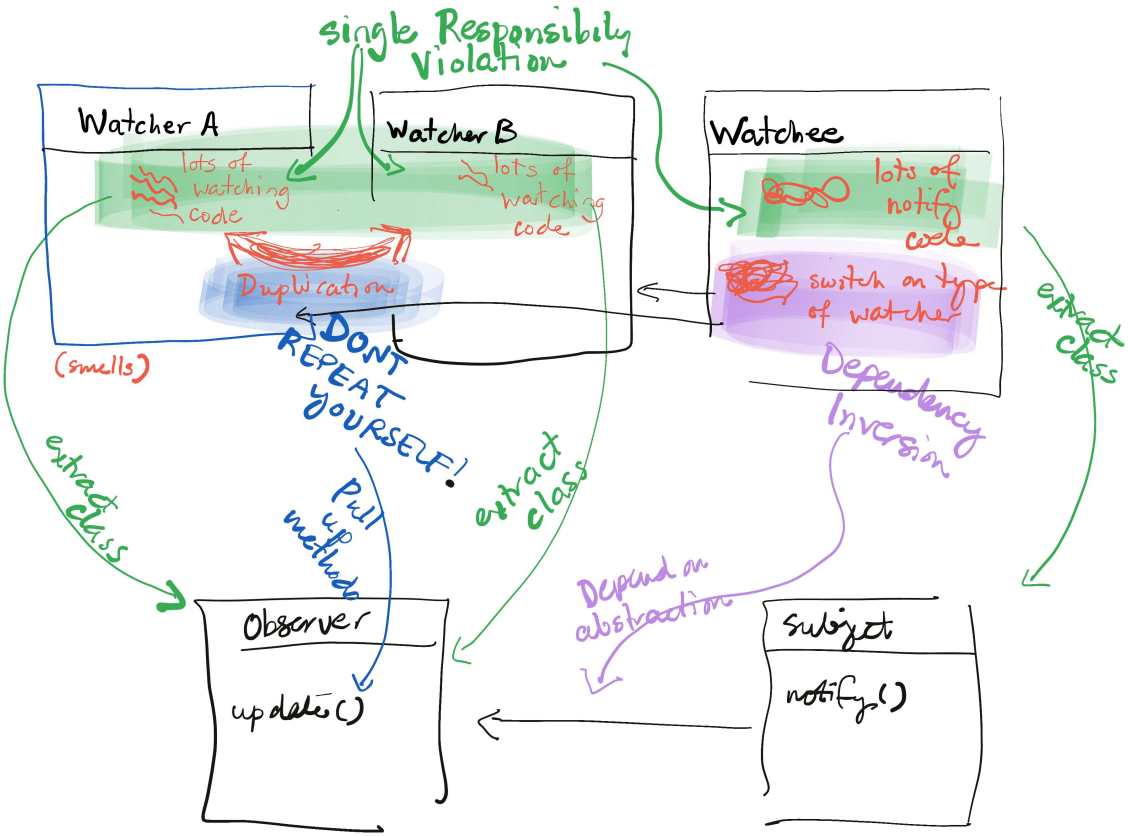


Figure 6. The emergent observer story sketch.

Learning Outcomes. Listening for design has some nice side effects, beyond just helping students hear what their code is saying. Here are a few:

The Listening perspective helps novices navigate the quagmire of heavily related patterns. Factory Method and Abstract Factory are great examples. One is just like the other, but with more abstraction. The descriptions of these two patterns are tricky to a newcomer, but the journeys to get there are straightforward to chart: when you hear a violation of the single responsibility principle, or you are experiencing shotgun surgery, abstract again.

While listening for design may help novices remember the structure of existing design patterns (by remembering how they are derived), it frees the educator from promoting rote memorization of a catalogue of design options. It allows students to uncover their own design patterns, maybe without even realising they have arrived at a canonical one. They no longer have to let the design patterns book become their menu for good design: instead they can listen to their code, and go where it beckons.

After tracing the stories of several design patterns, themes begin to emerge: duplication is spotted and pulled up, switches on type are spotted and higher types are introduced, responsibilities are pulled out of bloated classes. Students can begin to see that while the intents of the patterns differ dramatically, as do their program semantics, many of the smells are the same, and to some extent, the responses also look the same. Students can begin to appreciate the meta-patterns in design patterns, directing them straight to the core of object oriented design: pull up, abstract, override.

Assessing Listening. Our current approach involves evaluating students step by step: given this code, and this desired change, which code smells are problematic? Which principles are violated? What refactorings would you apply? What pattern would then emerge? At our scale, we do not have a lot of room for free exploration in our summative assessments, but at a smaller scale, or in a higher-level course, it may be tractable to give students code and see where they go. We can imagine a salon-style setting, in which, given a rich problem description, students can meander, explore, and debate their choices, without the right answer necessarily having been pre-established.

5 Contextualising

Design principles are typically associated with the dominant code use case: large industrial code that will be undergoing evolution. Not repeating yourself, for instance, is, in that context, almost always a good idea. It feels safe to say that if repetition becomes too insidious, then it should be stamped out through strategic application of abstraction.

However, digging a little deeper tells us a more nuanced story. That sometimes a code smell to one person, is a design principle to another. For instance: when a student is at the very beginning of learning the concept of sequential execution, it may be very helpful to see the same line of code repeated. This gives them a physical, and intentionally concrete representation of repetition in the code. This is necessary even before they know what a loop is. Showing a loop would be problematic if the student does not yet understand this control abstraction. Similarly, novice students tend to in-line behaviour so that they can follow, without jumping around, the details of the implementation of an algorithm. Hiding behaviour behind functional abstractions, especially if there is overriding involved, would confuse the student, and would be, quite rightly, considered poor pedagogical code design.

Open source development may implicate different principles from closed source development. Open source programmers likely prepare their code for public consumption to enhance the principle of readability, and may employ a more granulated style for the sake of facilitating unforeseen expansion and reuse.

That is all to say: different people will hear differently. And what they hear will depend on many factors, including the person's culture, the use cases that they are imagining, and their past experiences.

As a result of different contexts of hearing, the patterns that would emerge and the principles that would emerge would differ. This type of *design relativity* implies that patterns and principles are actually subjective and contextual.

When we teach students about listening to design, we need to build in awareness of which code use case we are asking them to occupy. In the later years of the software engineering stream, we ask students to mimic and acquire the industrial style of code listening. But in the earlier years, we may task them with attaining different design principles: we may impose certain coding conventions that facilitate grading, or code walk-throughs, or the derivation of clearer diagrams. We may tell them to be more or less aggressive in their use of inheritance. In How to Design Programs, for instance, we start them off using a recursive template to operate on a list, and must do so in a principled way, whereas later they are shown a more expert method (using abstract functions), which brings to bear new design principles, and concomitant code smells.

Maintaining awareness of the contextual nature of design principles will likely enhance and accelerate students' wisdom in listening. It will show them that principles are present only *for a purpose*, and that when the domain of application changes, the principles and their priorities may also change.

6 Reflections

Hill Climbing. In teaching novices to listen and respond with refactorings, are we driving them towards a path of needless abstraction, needless refactoring, and even hill climbing? Will students arrive at design dead-ends with this approach? Perhaps… but teaching emergent design carries the same risk as when teaching refactoring: that any time an opportunity to refactor is identified, that the refactoring is carried out. The refactoring community has done an excellent job of counseling action only when needed, with the rule of three (*two instances of similar code don't require refactoring, but when similar code is used three times, it should be extracted into a new procedure*). Similar coaching goes along with the message of listening to code. Listen for protest to a change.

Hill climbing is less of a concern for code that is composed by a team. Group projects in software engineering courses teach students to respect what others on their team may be hearing the code say, and to work collaboratively. Different programmers will hear different smells and will bring individualised perspective and wisdom, rooted in principled understanding, to their interpretation of what the code is telling them. By listening in concert, a team can hear the code more clearly. As Gabriel writes, "Conceptual integrity arises not (simply) from one mind or from a small number of agreeing resonant minds, but from sometimes hidden co-authors and the thing designed itself."

Reality Check. The reality of writing code that runs in the wild is that it must solve a task and it must do so within constraints like cost, performance, and other non-functional requirements. Listening to code for emergent design, therefore, is only a part of the equation when building useful systems. Ultimately, code is not written to satisfy design principles, but to solve a problem. Code that solves the problem cheaper than a better-designed alternative may be more desirable. Our conceptual model does not aim to capture this complexity. And, we are interested in ideas for how to introduce these other concerns into the triad of code smells, design patterns, and design principles.

Programmer Relegation. Taken to its extreme, emergent design may seem to reduce the programmer to a functionary: a slave to their own creation. Does emergent design indeed mean that the programmer is passively listening and responding to what the artifact wants to be (with minimal volition), or is there a more active role that the programmer can play? Is the only true action the initial seed, while all else is a

response? This feels very unlikely, especially in the face of structurally contradictory changes. There will always be a place for wisdom and experience in design. Even in choices of technology, language, or the underlying framework. Listening is the first step, but the programmer's lexicon can expand out from principles and patterns into more sophisticated mechanisms. If this were not the case, then it's conceivable that we could write an automatic refactorer: one that does not just perform refactorings, but also imposes refactorings based on a change or augmentation to the system. We can intuitively see that this is reductionist and problematic: this refactorer's search space would be vastly inferior to our own ability to reason about the potential of our code. It's true that we need to listen to code to know what to do, but the response is wholly our domain.

When Abstractions Fail Us. What do we do in the face of truly contradictory changes? When listening to one message would necessitate ignoring another or causing more cries of protest, regardless of how much abstraction we employ? Design paradigms (Functional programming, Imperative programming, Object Orientation, RESTful design, AOP, etc.) arose because of a realisation that the available design paradigms, the available abstractions, were not up to the task of abstracting away the smells while satisfying principles.

In the early days of Aspect-orientation, Gregor Kiczales toyed with the term *emergent entity* as a way to identify an abstraction that emerged from the code, even when, on the surface, the program was well designed. The concept of an emergent entity was folded into a more actionable pair of terms: scattering and tangling, which capture the code smell that signalled the manifested entity. The two most prominent groups at the time looked at scattering and tangling, and each group chose one of those to individually address with language support: The team at TJ Watson [9] looked at tangling, providing support for specifying views of individual classes that would solve what the divergent changes code smell, and single responsibility principle violations. The team at PARC focused on scattering (while still addressing tangling), providing a language by which behaviour that was impossible to localise could be described in one place, and compiled (or close to it) into target locations, thereby solving shotgun surgery and attacking a particularly troublesome version of semantic coupling [7].

Both teams identified a fundamental problem with object orientation, or really with any design paradigm: that of the *tyranny of the dominant decomposition*. That a programmer would need to make a choice when faced with structurally conflicting changes: to optimise for changeability in one way, or the other. When faced with a failure of available abstraction mechanisms, their response was to devise new abstraction mechanisms. Limited by their paradigm, they abstracted into a new paradigm. The solution could not be found within the code or in the language in which it was written. Change had to come from *without*: by introducing a new framework, new interpreter, or making changes to the compiler itself. The problematic and unresolvable contradictory code smells become motivation for new levels of previously unsupported abstraction (Aspects, explicit tests *a la* JUnit, Lambdas, etc). When abstractions fail us, we need to not just listen to the code, but to listen to the paradigm.

7 Unfolding

In his work *The Nature of Order, Volume 2* [1], Alexander described a four-principled approach to design, called *unfolding*. The principles were (1) Step-by-Step Adaptation, (2) Feedback, (3) Unpredictability, and (4) Awareness of the Whole. Echoes of this work can be found in today's Agile methodologies, but Alexander's ideas initially impacted software engineering through *patterns as language*. This notion was captured and explored by the Hillside Group (https://hillside.net/) and is precisely our starting point in this essay: as a method for listening to design, responding, and arriving at patterns. For the student, Alexander's third principle, unpredictability, is the least comfortable:

> *To make the adaptation successful, the process must be relaxed about the unpredictable character of where it goes. Unfolding cannot occur except in a framework which allows the whole to go where it must go. The dire modern passion for planning and advance control must be replaced by an attitude which recognizes that openness to the future, and lack of predictability, is a condition for success. It must be alright for the thing to become whatever it becomes, under the influence of adaptation and feedback, even though one does not know, in detail, what that thing is going to be.*
> – *The Nature of Order, Book 2*, Draft dated January 26, 1997, page 134; roughly the same content as *The Nature of Order, Book 2*, pp 236–241 along with all the material on wholeness in Part Two of that book, 2002.

Learning to live, in an agile way, with meandering and revelatory design, means practice at living with unpredictability. And it seems fitting to use design patterns as a vehicle for teaching students to listen and unfold.

Ultimately, we believe that *listening* to code allows code to guide the programmer to better design. Code smells are the voice of the code, and internalisation of design principles tunes the programmer into hearing and interpreting code's painful cries of resistance to change. Programmers respond by refactoring, incrementally emerging positive design. Listening, hearing, and responding, affords a highly skilled, revelatory process of design that sits in counterpoint with design planning.

Many applications of Alexander's principles 1, 2, and 3, can afford the developer a long enough design view that

they can arrive at Principle 4: Awareness of the Whole. The whole in a software project is so dynamic, so dependent on such a variety of influences including time, space, and changing needs, that such awareness comes at the hard won price of experience. Being able to maintain awareness of the whole means that the developer can see likely future developments, typical pitfalls, and common user-level requirements evolution [2].

Still, we believe this is not the domain, solely, of the expert. Just as students of writing, painting, architecture, and all arts, are taught to let their seeded artifacts demand their own refinement, we, too, can instill this skill in newcomers to our field. By telling story after story of revelatory design, perhaps mined from the world of Design Patterns, we can open up the world of listening so students can learn to unfold the abstract.

8 Acknowledgements

Sincere thanks to Richard Gabriel for in depth comments and insights, for chasing down the Unpredictability reference, and for lending us the jumping off point for this work.

References

[1] Christopher Alexander. 2002. *The Nature of Order, Volume 2.* Routledge.

[2] Kent Beck and Cynthia Andres. 2004. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional.

[3] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs: An Introduction to Programming and Computing.* MIT Press.

[4] Martin Fowler. 1999. *"Refactoring - Improving the Design of Existing Code"*. Addison-Wesley. http://martinfowler.com/books/refactoring.html

[5] Richard P. Gabriel. 2008. Designed as designer. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA).* 617–632. https://doi.org/10.1145/1449764.1449813

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.

[8] Robert C Martin. 2000. Design principles and design patterns. *Object Mentor* 1, 34 (2000), 597.

[9] Harold Ossher and Peri Tarr. 2001. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. *Commun. ACM* 44, 10 (Oct. 2001), 43–50. https://doi.org/10.1145/383845.383856

[2] It is possible that adopting a Kent Beck style systems metaphor [2] could accelerate the process of gathering the necessary experience to afford Principle 4. For instance, knowing that one is facing changes associated with a Blackboard, or a Shopping Cart, may help that zoom out/step back/gain perspective process that Alexander prescribes, and would foretell likely future changes. A full exploration of this could be the substance of an entirely different essay.