# Abstraction

# Reading Reflection

Discuss in groups

- Are there constructs you take for granted that Guarino describes as abstractions?
- How much of our abstraction stack do you think is a product of quirks of history?
- Pick any language element—GOTO, the null pointer, the loop, message passing, first-class functions.  Someone was the first person to build that on top of machine instructions!  What do you suppose they were thinking when they did that?
- After this reading, do you think of our collective programming language culture as more or less malleable?

# Reading Reflection

Let’s share!

# Reading Reflection

Let’s go a little deeper…

- This reading is weird because it isn’t a design reference material like many of our other readings, but it can change our points of view as designers.  Did anyone observe a shift themselves?
- Do you think any of Guarino’s observations don’t stand the test of time?

# Some Clarity on the Current Assignment

- Explicitly allows abstractions and not just new language constructs
- For this assignment we're looking for designs that *change the program text* (or change the AST if you're in a non-textual situation! :) )
  - New language features, new libraries, new APIs
  - Tweaks/changes to languages, libraries, and APIs

# How to Design Languages

Reading some of these is a good way to figure out that designing languages is hard.  And it they give you a sense of why various language families came about.  Do they tell you how to design more?

- Hints on Programming Language Design, Hoare
- Why Functional Programming Matters, Hughes
- Lisp: Good News, Bad News, and How to Win Big, Gabriel
- Confessions of a Used Programming Language Salesman, Meijer
- How Enterprises Use Functional Languages, and Why They Don’t, Wadler

# How to Design Languages

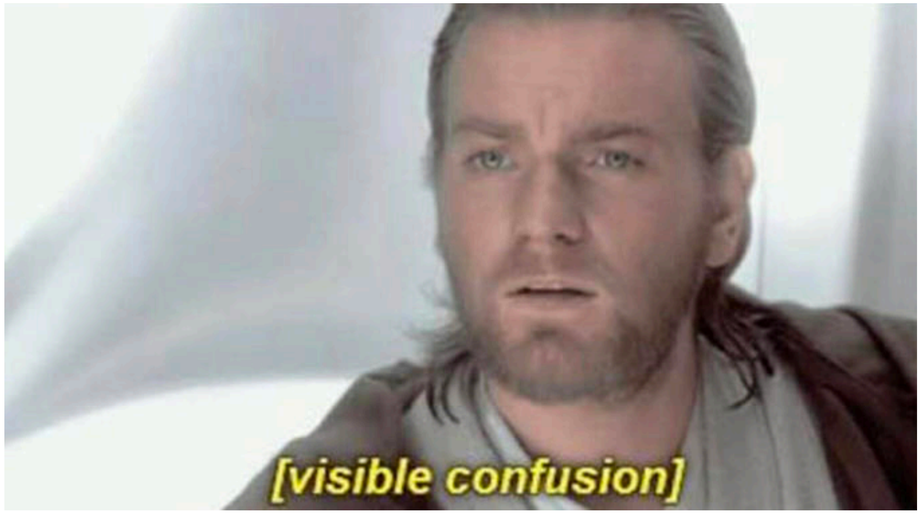
## Some guides are more limited in scope but more concrete…

- How to Design a Good API and Why it Matters, Bloch

**All programmers are API designers.** Good programs are modular, and intermodular boundaries define APIs. Good modules get reused.
**APIs can be among your greatest assets or liabilities.** Good APIs create long-term customers; bad ones create long-term support nightmares.
**Public APIs, like diamonds, are forever.** You have one chance to get it right so give it your best.
**APIs should be easy to use and hard to misuse.** It should be easy to do simple things; possible to do complex things; and impossible, or at least difficult, to do wrong things.
**APIs should be self-documenting:** It should rarely require documentation to read code written to a good API. In fact, it should rarely require documentation to write it.
**When designing an API, first gather requirements—with a healthy degree of skepticism.** People often provide solutions; it's your job to ferret out the underlying problems and find the best solutions.
**Structure requirements as use-cases:** they are the yardstick against which you'll measure your API.
**Early drafts of APIs should be short,** typically one page with class and method signatures and one-line descriptions. This makes it easy to restructure the API when you don't get it right the first time.
**Code the use-cases against your API before you implement it,** even before you specify it properly. This will save you from implementing, or even specifying, a fundamentally broken API.
**Maintain the code for uses-cases as the API evolves.** Not only will this protect you from rude surprises, but the resulting code will become the examples for the API, the basis for tutorials and tests.
**Example code should be exemplary.** If an API is used widely, its examples will be the archetypes for thousands of programs. Any mistakes will come back to haunt you a thousand fold.
**You can't please everyone so aim to displease everyone equally.** Most APIs are overconstrained.
**Expect API-design mistakes due to failures of imagination.** You can't reasonably hope to imagine everything that everyone will do with an API, or how it will interact with every other part of a system.
**API design is not a solitary activity.** Show your design to as many people as you can, and take their feedback seriously. Possibilities that elude your imagination may be clear to others.
**Avoid fixed limits on input sizes.** They limit usefulness and hasten obsolescence.
**Names matter.** Strive for intelligibility, consistency, and symmetry. Every API is a little language, and people must learn to read and write it. If you get an API right, code will read like prose.
**If it's hard to find good names, go back to the drawing board.** Don't be afraid to split or merge an API, or embed it in a more general setting. If names start falling into place, you're on the right track.
**When in doubt, leave it out.** If there is a fundamental theorem of API design, this is it. It applies equally to functionality, classes, methods, and parameters. Every facet of an API should be as small as possible, but no smaller. You can always add things later, but you can't take them away. Minimizing conceptual weight is more important than class- or method-count.
**Keep APIs free of implementations details.** They confuse users and inhibit the flexibility to evolve. It isn't always obvious what's an implementation detail: **Be wary of overspecification.**
**Minimize mutability.** Immutable objects are simple, thread-safe, and freely sharable.
**Documentation matters.** No matter how good an API, it won't get used without good documentation. Document every exported API element: every class, method, field, and parameter.
**Consider the performance consequences of API design decisions, but don't warp an API to achieve performance gains.** Luckily, good APIs typically lend themselves to fast implementations.
**When in Rome, do as the Romans do**. APIs must coexist peacefully with the platform, so do what is customary. It is almost always wrong to transliterate an API from one platform to another.
**Minimize accessibility;** when in doubt, make it private. This simplifies APIs and reduces coupling.
**Subclass only if you can say with a straight face that every instance of the subclass is an instance of the superclass.** Exposed classes should never subclass just to reuse implementation code.
**Design and document for inheritance or else prohibit it.** This documentation takes the form of selfuse patterns: how methods in a class use one another. Without it, safe subclassing is impossible.
**Don't make the client do anything the library could do.** Violating this rule leads to boilerplate code in the client, which is annoying and error-prone.
**Obey the principle of least astonishment.** Every method should do the least surprising thing it could, given its name. If a method doesn't do what users think it will, bugs will result.
**Fail fast.** The sooner you report a bug, the less damage it will do. Compile-time is best. If you must fail at run-time, do it as soon as possible.
**Provide programmatic access to all data available in string form.** Otherwise, programmers will be forced to parse strings, which is painful. Worse, the string forms will turn into de facto APIs.
**Overload with care.** If the behaviors of two methods differ, it's better to give them different names.
**Use the right data type for the job.** For example, don't use string if there is a more appropriate type.
**Use consistent parameter ordering across methods.** Otherwise, programmers will get it backwards.
**Avoid long parameter lists,** especially those with multiple consecutive parameters of the same type.
**Avoid return values that demand exceptional processing.** Clients will forget to write the specialcase code, leading to bugs. For example, return zero-length arrays or collections rather than nulls.
**Throw exceptions only to indicate exceptional conditions.** Otherwise, clients will be forced to use exceptions for normal flow control, leading to programs that are hard to read, buggy, or slow.
**Throw unchecked exceptions unless clients can realistically recover from the failure.**
**API design is an art, not a science.** Strive for beauty, and trust your gut. Do not adhere slavishly to the above heuristics, but violate them only infrequently and with good reason.

# How to Design Languages

- ????



Sarah's semi-facetious rules for

# How to Design Languages

- Figure out what users need.
- Do what makes them successful.

# A pragmatic guide to abstraction design

If you're in this class, here's hoping you're willing to take the time to visit with users, *but* if for some reason you can't…

- Write out >20 programs that could benefit from the category of abstraction you're considering
- Brainstorm abstraction alternatives
- Write out all 20+ programs with each of the alternatives

Is this a substitute for user input? No. But it's better than doing literally nothing to assess design! :)

# A pragmatic guide to abstraction design

If you're in this class, here's hoping you're willing to take the time to visit with users, *but* if for some reason you can't…

- Write out >20 programs that could benefit from the category of abstraction you're considering
- Brainstorm abstraction alternatives
- Or do cognitive dimensions analysis for each of the alternatives!

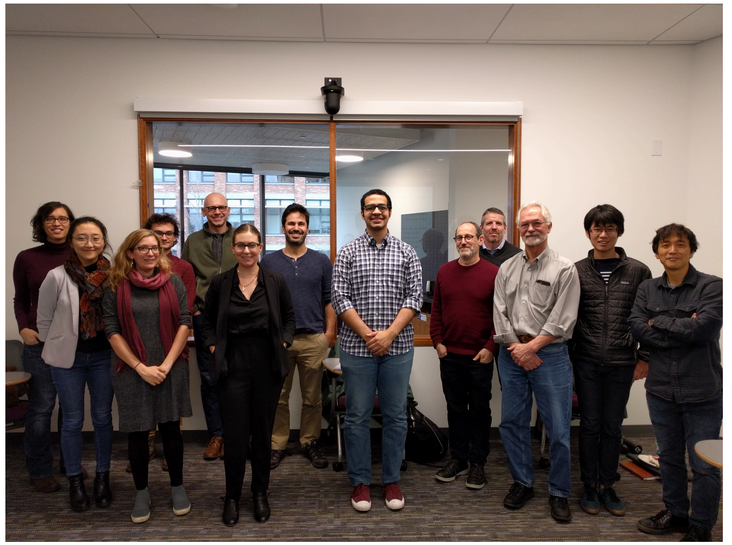
Is this a substitute for user input? No. But it's better than doing literally nothing to assess design! :)

# A better guide to abstraction design

- **Learn what tasks are hard**
  - What trips users up in their current language jungle (the set of languages, libraries, APIs they use regularly)?
- **Learn what tasks are easy**
  - What are the programming tasks that they do every day, that they could do in their sleep, that they execute or talk about as though it's nothing?
- **Learn what *natural language* is easy**
  - How do your users talk about their domain?

# Story Time

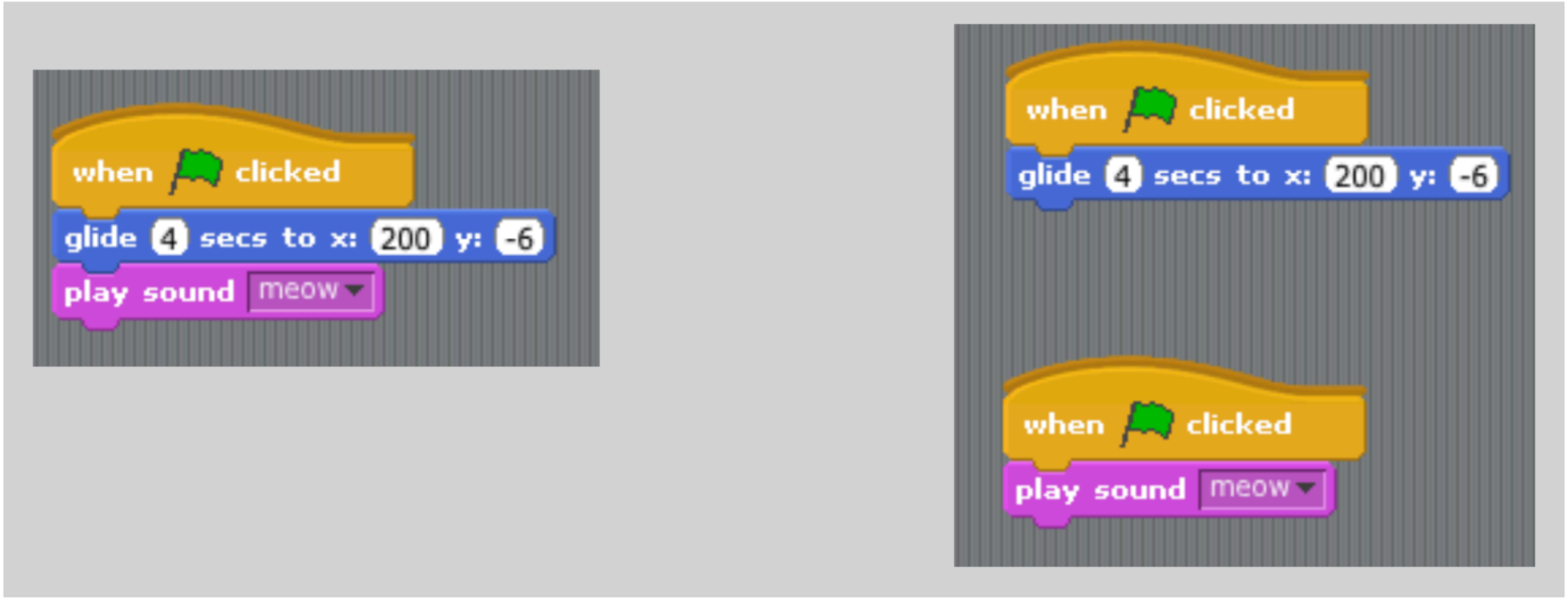
- Learn what tasks are hard
- Learn what tasks are easy
- Learn what *natural language* is easy



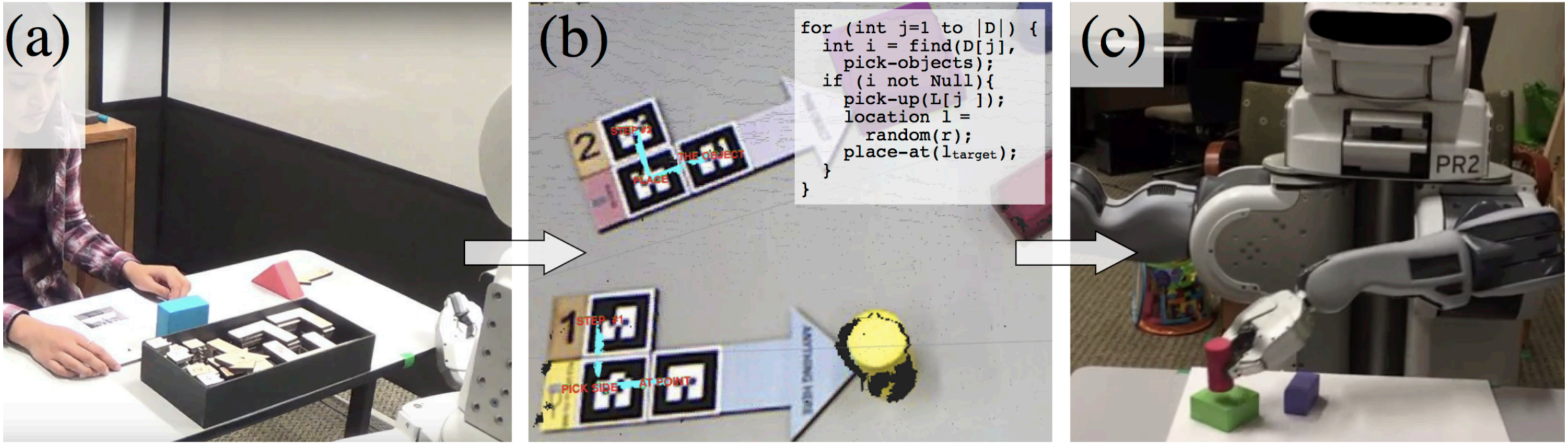
Workshop at UW to learn about social scientists' programming needs.

# You can get non-coders to do parallelism!

# You can get non-coders to do parallelism!




**Scratch Environment**
**http://coderdojodl.com/wp-content/uploads/2014/11/CoderDojoDL_Scratch-Session-3-Parallelism-Events.pdf**




Situated Tangible Robot Programming, Yasaman S. Sefidgar et al.

# Abstractions aren’t done evolving

- They weren’t handed down from the ancients in their current form
- They weren’t logically deduced or derived from first principles
- They aren’t fixed

# Assignment

- Questions before we begin?
- During today's work time, think about…
  - …times when your need finding participants wanted to write a program in a given way but stumbled.
  - …times when your need finding participants had to turn to resources outside their programming environment.  Why couldn't their code suggestion/autocomplete help them?
  - …how your need finding participants talked about their problem domain.  How can that shape your abstraction design?