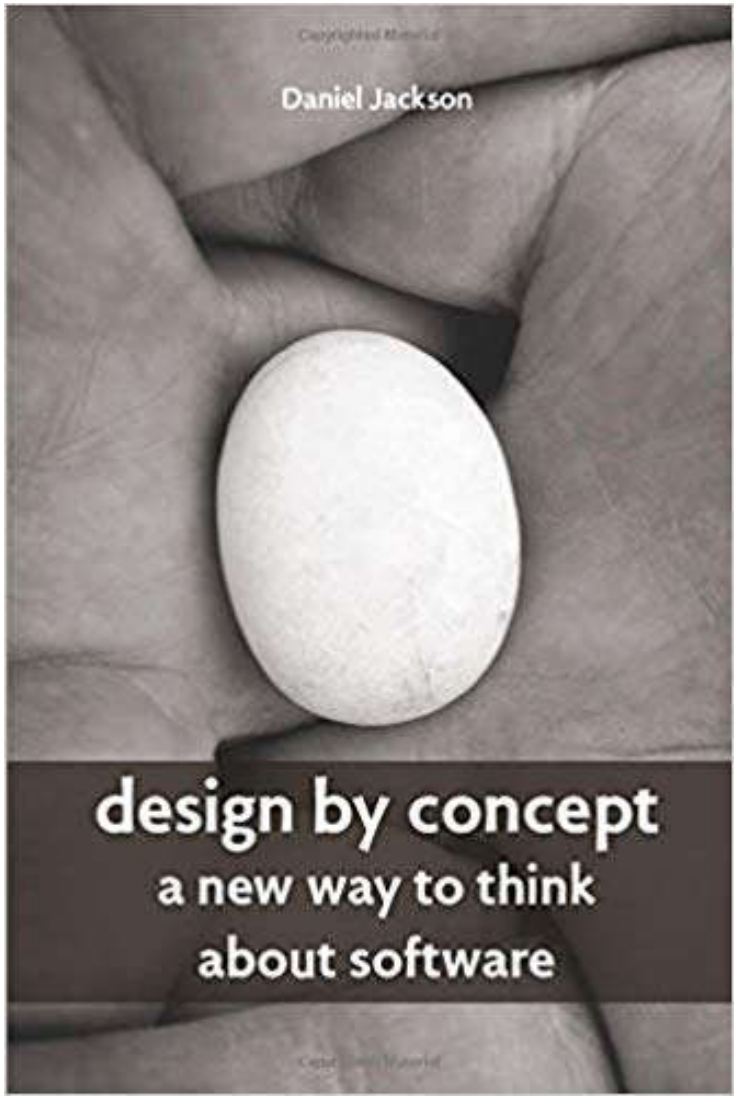
Lecture 2: Concepts for Language Design

Jonathan Aldrich

17-396/17-696/17-960 Language Design and Prototyping

Concepts Are a Tool for Conceptual Design

- Approach to capture and analyze the **conceptual design** of software
- Example: Concepts in Facebook
  - **Posts** – to share information
  - **Friends** – to decide who sees posts
  - **Likes** – to rank posts
- **Concept:** an abstract and elemental **mechanism** that is designed to fulfill a **purpose** by **interacting** with the world around it.



Concepts Are Mechanisms for a Purpose

Example from *Design by Concept* by Daniel Jackson

concept trash

purpose

to allow undo of deletions

structure

item: set items

deleted: set item

behavior

delete(x:item) : deleted := deleted + x

restore(x:item) : deleted := deleted – x, assuming x in deleted

empty() : items := items – deleted, deleted := none

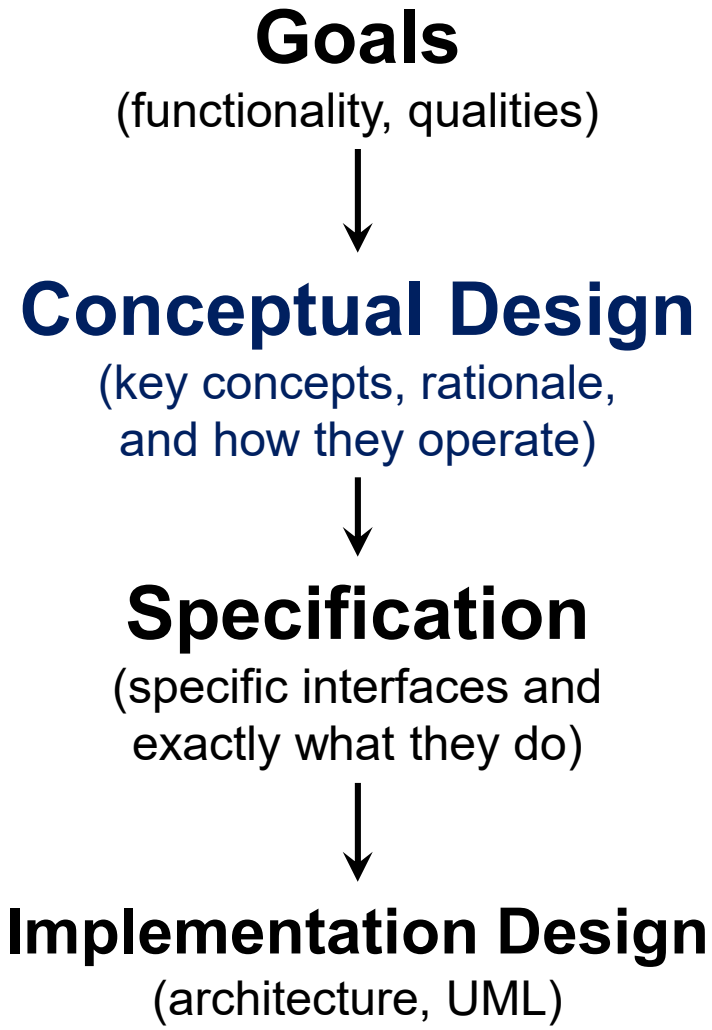
new():Item : items := items + result, assuming result not in items

tactic

if delete(x); not empty(); restore(x) then x not in deleted

if delete(x); empty() then x not in items

Concepts Sit Between Goals and Specification



- Relation to other design work
  - Vs. Goal modeling
    - Captures elements of the solution space
  - Vs. Specifications
    - Organized around ideas, not interface
    - Goal is facilitating design, not complete description
  - Vs. Implementation design
    - Focuses on interaction between system and the user or world
  - Vs. User-centered design
    - *cf.* The Design of Everyday Things – correspondence between need and interface
    - But software is different - concepts are *mechanisms*

Kinds of Analysis

- **Principles**
  - **One-to-One:** Concepts and purposes should be in a one-to-one correspondence.
  - **Uniformity:** A concept should apply generally and not be encumbered with special conditions.
  - **Genericity:** Prefer generic concepts to ones that are specific to a particular application.
  - **Integrity:** A concept's specification should continue to apply in the presence of other concepts, even if the concepts interact with one another

An Example PL Design Concept

concept everything is a value

purpose to allow programmers to uniformly manipulate the structure of programs at run time

structure every entity in the program that has run-time semantics is a first-class value

behavior any value can be stored in a variable or passed to a function

tactic passing a top-level defined function to a higher-order function

dynamically choosing what super-class to inherit from

dynamically choosing between implementation modules

The Genericity Principle: Prefer generic concepts to ones that are specific to a particular application.

In Smalltalk, Lisp, and Wyvern, everything is a value.

An Example Type System Concept

concept ownership

purpose to prevent sensitive operations made via two different aliases of an object from interfering

structure any variable may be marked owned

behavior invariant: at all execution points, for every object, at most one variable or field is the owner of that object

assigning one owned variable to another transfers ownership

tactic when a call is made from object A to object B, passing owned Money, then B is the owner of Money and can spend it; A can't spend it anymore.

The Uniformity Principle: A concept should apply generally and not be encumbered with special conditions.

My first "purpose" involved money, but I made it more generic to fit into broader uses of ownership.

An Overloaded Concept

concept untyped dynamic semantics

purpose

to make the semantics easy to understand

to allow developers to run code that doesn't typecheck

The One-to-One Principle: Concepts and purposes should be in a one-to-one correspondence.

Here we have 2 purposes for one concept! Maybe Jackson is wrong.

But, just for fun, let's try to follow the rule.

Can we unify the purposes? No.

Can we create two concepts?

Refactoring Into 2 Concepts

concept untyped dynamic semantics

purpose to make the semantics easy to understand, since you don't have to read the types to know what a program does

concept executing programs that don't typecheck

purpose to allow developers to get useful results or feedback from running code that doesn't typecheck

depends on untyped dynamic semantics

This version is more useful!

- One feature can be adopted without the other
- Clarifies the justifications for each feature, enabling better design decision-making
- Discovered design nuance: the second concept makes the design more complex in some ways, because more kinds of errors can occur

Refactoring Into 2 Concepts

concept untyped dynamic semantics

purpose to make the semantics easy to understand, since you don't have to read the types to know what a program does

concept executing programs that don't typecheck

purpose to allow developers to get useful results or feedback from running code that doesn't typecheck

depends on untyped dynamic semantics

Method observation: these are more abstract than Jackson's concepts

- I was unable to specify structures / behavior for these without specifying the language (which would be too concrete, as these are general ideas)
- Rather, these put a *constraint* on the structure/behavior of the language
- Tactics can be specified, and they fulfill other aspects of Jackson's defn

Refactoring Into 2 Concepts

concept untyped dynamic semantics

purpose to make the semantics easy to understand, since you don't have to read the types to know what a program does

concept executing programs that don't typecheck

purpose to allow developers to get useful results or feedback from running code that doesn't typecheck

depends on untyped dynamic semantics

Observation: dependencies are important in this domain

- I tried to make these independent, thinking the one-to-one principle requires this, but was unable to do so
- I realized that there was a one-way dependency. This came up again, and appears to be common in the PL domain
- Jackson's original paper discusses dependencies, but the book de-emphasizes them

Integrity of Untyped Dynamic Semantics

concept untyped dynamic semantics

purpose to make the semantics easy to understand, since you don't have to read the types to know what a program does

structure language syntax has identified "value" parts

behavior dynamic semantics depends only on value parts of language

The Integrity Principle: A concept's specification should continue to apply in the presence of other concepts, even if the concepts interact with one another.

Wyvern's Type-Specific Language (TSL) mechanism breaks this specification. What to do?

Benefits from small, targeted specs

- When using concepts to capture the design of Wyvern, I identified several sources of unnecessary complexity
  - E.g. auto-import of types mentioned in module headers
- Any specification would have pointed this out
  - But in PL, people tend to specify core calculi, or *everything*
    - The particular features above weren't "core calculus" features
    - Specifying everything is too much work, and gets delayed
  - Small specs targeted at concepts are a chance to find these early

Benefits from focus on purpose

- Focus on purpose produces better designs
  - Every decision is motivated
  - Separating "why" from "how" is very clarifying.
    - It's easy/natural to get "how" pinned down.
    - "Why" is harder but doing it provides significant dividends by identifying problems.
  - Helps think about whether design is general enough to cover the entire purpose

- Purpose facilitates explanation of design in a paper
  - Helps come up with a coherent explanation for what each concept does, precisely, and how concepts fit together
  - Note: dependencies also super helpful – ordering presentation

Benefits from uniformity

- **Uniformity:** the concept should apply generally and not be encumbered with special conditions
- Indentation in Wyvern
  - No uniform way to describe how indentation works
    - Instead, many special cases
  - We are currently doing a minor redesign to correct this

Benefits of the One-to-One Principle

- **The One-to-One Principle:** Concepts and purposes should be in a one-to-one correspondence.
- OO Classes typically have 4 purposes
  - Specifying the type of a data structure
  - Creating instances of a data structure
  - A place to put module-local static state
  - A way to reuse code (via inheritance)
- Wyvern instead has 4 separate constructs
  - Types, object creation, modules, and delegation
  - Benefit: clearer conceptual model, more flexibility for programmer
  - If the flexibility benefit did not exist, we might not make these *constructs*, but we still would want separate *concepts*
- Wyvern will also eventually have classes
  - Purpose: to succinctly describe a data structure
  - Depends on all 4 concepts above
- Think about concepts separately, even if the syntax combines them

Summary Observations

- Design by Concept is not a panacea
  - Specifications don't fit together as well as standard type theory
  - Criteria like One-to-One, Integrity, etc. may not be achievable
    - But trying—and getting partway there—is still useful!
- Incomplete Design by Concept can still be useful
  - Pick the most important concepts
  - Focus on purpose and tactics; add structure/behavior where useful
- Design by Concept shares some advantages with other design techniques
  - Seems particularly well suited to PL (and software more generally?)

Summary: Concepts for PL Design

- Capture **essential ideas**
- **Organized conceptually**, rather than syntactically
- Capture **rationale**
  - both logical and by example
- Represent **dependencies**
  - Can't incorporate X without Y
  - Also helps to order a design presentation logically
- Describe **mechanism**: how the idea works
- **Evaluate design** at a conceptual level
  - Are concepts redundant?
  - Do concepts interfere?
  - Are concepts captured crisply?