Design (Yet Again)

With comments on the textbook chapters

Design Models

- How do you decide what models to build?
  - Sommerville's list: data flow models, entity-relation models, structural models, aggregation, inheritance, interaction models
- Of what use are they?

The Which and Why of Models

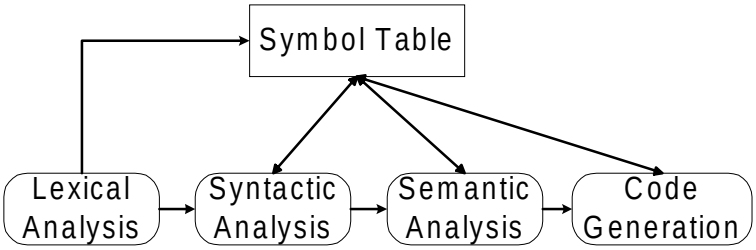
- A model exposes some properties and suppresses others
  - Ex.: a wind-tunnel model of an aircraft exposes aerodynamics and suppresses paint color and seating
- A model focuses attention on particular design decisions
  - For making and evaluating design decisions
    - which implies that we must be able to compare alternatives and evaluate their impacts
  - For understanding the system later

When Should You Use …

- Class hierarchy?
- E-R or aggregation hierarchy?
- State machine?
- Message sequence chart?
- BNF?

Functional & OO Design

- Can information hiding be applied in a data-flow oriented design?
  - Consider the standard compiler example:

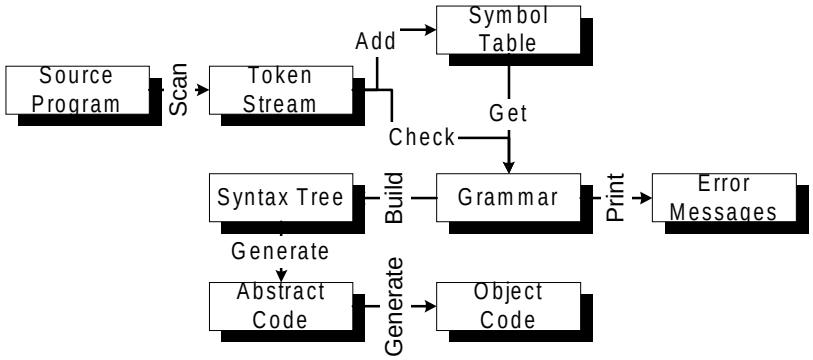


A Note on Diagrams

- Un-keyed, ambiguous diagrams are fine for brain-storming
- … but NOT for analyzing and documenting design
- Some of the textbook examples are not acceptable

OO compiler diagram (?)

- Sommerville's OO view, pg 216



OO Design (chap 14)

- "Objects are abstractions of real-world or *system entities*"
  - Doesn't rule out much …
  - Grammatical analysis for object identification?
    - But discovery is distinct from evaluation
- "Object-oriented systems are easier to maintain … "
  - Sometime

(Optional slides follow)

- I don't really expect to get to these on Tuesday, depending on how much we discuss assignment 1

Object Interface Design

- Page 248: "Shared data areas are eliminated. … There is no possibility of unexpected modification of shared data."
- Page 267: "Functions which should always be provided for each attribute include functions to store and retrieve information in that attribute"
- How can these be reconciled?

Objects and Abstraction

- "Attributes" are observable properties of idealized, abstract objects
  - Which may be completely different than internal structures
- A representation function maps concrete states to abstract states
  - Why not the other way? Consider a ring buffer.

Abstract Models and Interface Specs

- Abstract model helps simplify an interface specification and make it precise
  - Most useful when a complex state is maintained. May not be useful for simple transformations
- Model can come from existing domain (sets, integers, relations, …) or can be created (e.g. algebraic specs)
- Model operations may not be map one-to-one with interface operations
  - Consider stack, ring buffer, symbol table, …

Two-Level Specifications

- Abstract model
  - Algebraic spec
  - State machine
  - …
- Interface spec
  - Logical statements involving a representation function from program state to the model

Two-Level ADT Specs

- Abstract model
  - Algebraic, functional (clean and easy to reason about)
- Interface spec
  - Map program state to objects in the model

Stack: Algebraic Spec

Signature

Create: -> Stack;

Push: Stack, Item -> Stack

Pop: Stack -> Stack

Top: Stack -> Item

Axioms

for all S: Stack, I: Item

Top(Push(S,I)) = I;

Pop(Push(S)) = S;

- The abstract model is completely functional, simple and easy to reason about
  - But it doesn't correspond directly to a reasonable interface
- It says nothing about errors and other messy stuff
  - What is Pop(Create())? This spec just doesn't say.
  - Implementations are messy anyway; stick all the crud there.

Stack: Interface Spec

```
Class Stack { …
        Stack(); // ensures rep(self) = Create();
        push(int i); // ensures rep(self) = Push(rep('self), i)
        int pop(); // assumes rep(self) = Push(i,S);
                   // ensures rep(self) = S and returnval = i;
… }
```

Assertions describe program effects through a representation function

- rep maps from program states to objects in the abstract model

A single method call need not correspond to an abstract function

- S.pop() does not correspond to any single function