# **Finding Classes**

© Gunnar Gotshalks

## Background

- Looking for good and useful data abstractions
  » Basis is good ADTs
- As in all design work, need creativity and experience
- We'll look at some
  - » good ideas
  - » precedents
  - » reuse
  - » some known pitfalls
- Need to read other designs

## Nouns & Verbs

- Some approaches suggest
  - » "Take your requirements document, and underline all the nouns and all the verbs. Your nouns will correspond to classes, and your verbs will correspond to methods of classes"
- Example
  - » "A database record must be created every time the elevator moves, from one floor to another"
  - » Suggests
    - > Classes: DATABASE\_RECORD, ELEVATOR, FLOOR
    - > Methods: create, move

## Nouns & Verbs – 2

- On the other hand the phrase may have been
  - » "A database record must be created for every move of the elevator from one floor to another
- Now move is noun and suggest it should be a class
- One major problem is
  - » All nouns can be verbed and many verbs can be nouns
- Selecting classes cannot rely on the style or word choice of a requirements document

## **Raw Data**

- Creating a comprehensive list of
  - » nouns potential entity and entity types
  - » verbs potential events
- Is at best a means to get started
- Also need a comprehensive
  - » lists of questions and potential questions the model is going to answer

> M is a model of a system S, if M can be used to answer questions about S with accuracy A

- Point is to be all inclusive
  - » It is easier to throw away later, then add later

#### **Process the Raw Data**

- Organize into potential ADT's
  - » Associate verbs with nouns
  - » Verbs correspond to events in the real world
    - > Messages that cross the system boundary
  - » Reject those that do not fit within the model boundary
  - » Reject those that are not related to the questions and answers associated with the model
  - » Reject those that are subsumed by other ADTs
  - » Be prepared to change names, look for abstractions, generalities

## Example

• Had the following earlier

> Classes: DATABASE\_RECORD, ELEVATOR, FLOOR

> Methods: create, move

Consider FLOOR

» What actions can it engage in?

- » What methods might be applicable?
  - > Can't find any, then this is rejected as a class
- » Maybe floor is an attribute of elevator
  - > An elevator can be at a floor
  - > When an elevator moves it changes floors

#### **Potentially Bad Choices of Classes**

- A class that performs something, that has an imperative name PARSE, PRINT, etc.
  - » Parses the input
    - > On the other hand we saw sort algorithms be objects
    - > Perhaps a parser is an object that we want to pass around
- It is never clear cut
  - » But a useful heuristic is to reject such classes until a need for them arises in later development
- Change the viewpoint
  - **»** Parse methods are operations on the object INPUT

## Bad Choices of Classes – 2

- A class that has a single routine that is exported
  - **»** Perhaps the method should be in another class
- A class that exists purely for classification
  - » Early class selection should not worry about inheritance structure
  - **»** First define the ADT's of interest
  - » Then look for abstractions, generalizations, taxonomy
    - > Danger is early taxonomy may bias ADT's in directions that do not correspond to the model

## Bad Choices of Classes – 3

- A class with no methods
  - » Only a record structure
  - » Rarely is such a class useful
    - > Exceptions global constants
- A class that refers to multiple abstractions
  - » A class should be an expert on one thing and one thing only
    - > Merging properties of STRING and EDITABLE\_LINE
  - » Class becomes too large and cumbersome

## The Ideal Class

- There is a clearly associated data abstraction (ADT)
- The class name is a noun or an adjective that characterizes the abstraction
- The class represents a collection of possible objects
- Several functions are available
- Several procedures are available
- Abstract properties can be stated formally
  - » class invariants
  - » requires and ensures clauses
- These are goals, not all properties necessarily hold

#### **General Heuristics for Finding Classes**

- Class categories
  - » Analysis classes
    - > from real world (as opposed to the model world)
      - plane, paragraph, course
  - » Design classes
    - > Architectural choices belonging to solution space
      - Command, State inheritance case study
  - » Implementation classes
    - > Data abstractions for internal needs of software
      - Linked list, array

### **Analysis & Implementation Classes**

- Analysis classes
  - » Based on the abstract concepts of the problem domain
    - > CAR, SENIORITY\_RULE, MARKET\_TENDENCY
  - » Characterized through visible features

> Chosen because of lasting value

- Implementation classes
  - » Used to make the system run on a computer
    - > Heavy on reuse
    - > Don't reinvent the wheel

## **Design Classes**

• Represent the abstractions that help produce extendible software structures

> STATE, COMMAND, APPLICATION

- » Design classes have been devised by others
   reuse is possible
  - > Read books, articles that describe designs
- » Design patterns capture proven design techniques > See some later
- » Describe abstractions that can be better understood as computational machines rather than objects

## **Other Class Sources**

- Previous developments
  - » May need to rework existing classes
- Adaptation through Inheritance
  - » This may provide sufficient adaptability
- Criticism & rework
  - » Study the data flow and modularization of your class structure
  - » Information needs to be known in too many places indicates missing abstractions

## Other Class Sources – 2

- Hints from other approaches
  - » Non OO systems are frequently good designs and may give ideas for classes in an OO approach
  - » Non OO methods give suggestions for finding abstractions
    - > Entity-Event approach from JSD
    - > JSP input and output structures, structure clashes and communicating sequential processes – repack problem
    - > Structured Design Constantine, et. al.
  - » Experienced developers are a source of suggestions

## Other Class Sources – 3

- Files
  - » System's file structure suggests objects and their class abstractions

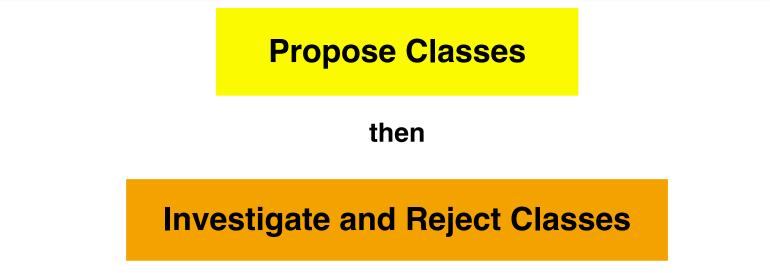
> ADT's for a system's input and output

- Uses cases
  - » Study scenarios of how a system is being used
  - » Problem is bias towards specific sequential processing too early in the design
  - » Do not provide abstractions that make a wider range of scenarios possible
  - » Useful for validating a system

### **Other Class Sources – 4**

- Reuse
  - » The best one of all
  - » Look at what is available
  - » Adapt to needs





- Do not be afraid to get it wrong at first
- Do not worry about a "main" program or a "user interface" first
  - » That's top down