

Core Object-Oriented Concepts

Object Oriented Analysis and Design

Aron Trauring

T++ Technical Skills Training Program

CUNY Institute for Software Design & Development (CISDD)

New York Software Industry Association (NYSIA)

December 1st, 2004

The Fundamental Challenge of Software Development

- Change is the only constant
- How do we deal with change?

It Starts With Requirements

- Requirements are incomplete
- Requirements are usually wrong
- Requirements (and users) are misleading
- Requirements do not tell the whole story

Why Requirements Change

- The users' view of their needs change as a result of their discussions with developers and from seeing new possibilities for the software
- The developers' view of the users' problem domain changes as they develop software to automate it and thus become more familiar with it
- The environment in which the software is being developed changes

Authoritarian Model

- Change is a necessary evil
- Manage change by controlling it
- Top-down approaches - Structured Analysis/Design/Programming
- SDLC, UP
- Ada School of OO — OO is about Information Hiding

Anarchist Model

- Embrace change
- Allow maximum flexibility to both encourage and manage change
- Bottom-up approaches
- Agilists, XP
- Smalltalk school of OO — OO is about “Those Who Know, Decide”

Functional Decomposition

- Break down (decompose) problem into functional steps
- Repeat and rinse
- Recipe model

Mini-Exercise 1

“Access a set of shapes stored in a database and then display them”

Problem with Structured Programming

- One main controller program
- “With great power comes great responsibility”
- Too much responsibility — code gets complicated
- Lesson 1: need to delegate responsibility
- All changes impact main module
- Focusing on too many things in one place means more error prone
- Lesson 2: Keep things simple

- Modularity can't deal with variable and changing data structure for shapes
- Lesson 3: Those who know should deal

Cohesion and Coupling

- Cohesion — how strongly the internal contents of a routine are related to each other
- Coupling — how strongly a routine is related to other routines
- Functional decomposition leads to weak cohesion and strong coupling
- Unwanted Side Effects — change in one place effects another unknown place
- Most of the time in debugging is spent on finding unwanted side effects or understanding how to avoid them
- Goal: strong cohesion (internal integrity) and loose coupling (small, direct and visible relations)

Mini exercise 2

“You are an instructor at a conference. People in your class had another class to attend following yours, but didn't know where it was located. One of your responsibilities is to make sure everyone knows how to get to their next class.”

Difference Between SP and OO

- SP — giving directions to everyone — you have to pay close attention to too many details. No one but you is responsible. You go crazy!
- OO – you give general instructions and then assume each person will figure out how to the task individually.
- Shift responsibility to those who know.

Mini exercise 3

“Need to give special instructions to graduate students who are assisting at the conference. Perhaps they need to collect course evaluations and take them to the conference office before they can go to the next class.”

Accommodating Change

- SP: every type of new student requires a change to controller program
- OO: new categories of students don't effect the main program; students them-self responsible

Three Perspectives on Software Systems

- Conceptual — “What am I responsible for?”
- Protocol (Interface) — “How am I used?”
- Implementation — “How do I fulfill my responsibilities?”

Three Sources of Flexibility

1. Conceptual — Students responsible for own behavior
2. Protocol (Interface) — Control program can talk to different students as if they were the same
3. Implementation — Control program doesn't need to be aware of the internal behaviors that lead to desired outcome

Abstraction — A Human-Centric Need

- In the beginning Binary machine code: 10010101010100.....
- Hex: 0A DE 01.....
- Assembler:

```
MOV DPTR,#CNT
```

- Third generation:

```
Do  
Portc = Inp(pdc)   ' Reads port C  
Out Pda , Portc   ' Writes it to port A  
Loop
```

- OOPL:

```
SJ = RegularStudent('Joe')  
SJ.GoToNextClass()
```

What is an Object?

- Conceptual — set of responsibilities — **Student**
- Protocol (Interface) — set of operations (behaviors) that can be invoked by other objects or itself — **GoToNextClass**
- Implementation — the specific data structures and related methods that implement the behaviors — **RegularStudent.GoToNextClass**

Classes and Instances

- Abstract Class — Highest level of an object — **Student**
- Concrete Class — *Subclasses* — more specific — **RegularStudent, GraduateStudent**
- Instance — Concrete items in a programs domain — *Instantiation* of a Class — **SJ = RegularStudent('Joe')**

Abstraction Benefit — Model Objects at Three Levels

- Analysis — Requirements of System under Design — Domain Model
- Design — Structure of software system — Domain Layer
- Implementation — Working Code — Language specific
- Goal: Small gap as possible between all three levels

Inheritance Benefit of Objects

- Set of attributes passed to descendants
- Conceptual — Is-A Relationship — Defines sets of responsibilities down the tree — **GoToNextClass**
- Protocol (Interface) — Customization / Specialization — Sub-classes can replace, provide or extend behaviors of parent classes — **RegularStudent.GoToNextClass, GraduateStudent.GoToNextClass**
- Implementation — Code reuse

Polymorphism Benefit of Objects

- “Many forms”
- Conceptual — Delegation of responsibility down the tree
- Protocol (Interface) — “many forms” of methods for same operation (depends on instance) — **RegularStudent.GoToNextClass**, **GraduateStudent.GoToNextClass**
- Implementation — overloading operators or operations — “+”, “print”, “__init__”

Encapsulation Benefits

- Hiding
- Conceptual — Composition — Has-a-Relationship — Classes
- Protocol (Interface) — hiding implementation details — **def GoToNextClass:**
- Implementation — properties hide accessor attributes — **Grade = property(getGrade, setGrade)**