Object Oriented Programming System

“No technique/technology or system is a silver bullet to enable failsafe automation solution. They are all enablers of a kind. The key to the best solution or the program is the development team’s overall ability”
Contents

Note: The audience should PAIR out now to do it the real way.. XP Style.. As we run through the lec-dem and the various excercises..

Demo of CAD Graphics: Procedural, Blocks and finally Object Oriented to visually display the simile in transition from procedural to object oriented and beyond..

• Fun with Objects if you understand them well..
• What is OOPS and what is not OOPS..
• Specifications and structure of OOPS: a lil bit of history..
• Evolving OOPS for your needs:
  – Trapping requirements
  – Designing with OOPS
    • Visualizing and representing OOPS
  – Writing the code
  – Debugging and testing OOP code
  – Delivering OOP code..
• Practical OOPS tips.. The best way to be a Zen master at OOPS..
  – All that you should do
  – All that you should avoid...
  – Somethings you can get away with.. Occassionally..
• Is OOPS the cure-all methodology.. Alternatives and their terrains..
  – The way ahead to succeed with OOPS.. And meld it with other techniques..
Questions

The 15 min run through on your baseline on OOPS:

• Quick one sentence definition of OOPS
• OOPS features
• All the languages that are OOP oriented
• Benefits of OOPS
• Where and when not to use OOPS..
Fun with Objects..

- "Have you heard about the **object-oriented way to become wealthy**?"
  - "No..."
  - "Inheritance."

- *In the case of a Australian application for military simulation the kangaro takes off like a helicopter as someone had accidentally termed kangaro’s class as inherited from helicopter’s...!!*

- *There is a point in your life when you realize that you have written enough destructors, and have spent enough time tracking down a memory leak, and you have spend enough time tracking down memory corruption, and you have spent enough time using low-level insecure functions, and you have implemented way too many linked lists.*
  -- Miguel de Icaza

- *C++ : an octopus made by nailing extra legs onto a dog.*
  -- off smalltalk.org

- *C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, you blow your whole leg off.*
  -- Bjarne Stroustrup

- Most programmers have probably never even heard of any languages other than C, C++, Java, C#, Visual Basic, Fortran, and (maybe) Perl. That's too bad, because there isn't a great language in that entire list. What this shows is that people who develop and use good languages have got to get better at getting the word out. For my part, I would recommend that anyone reading this go out and learn Ocaml (especially!), Lisp and/or Scheme, Python, Smalltalk, and Eiffel, especially if they think that "all languages are the same".

- *Plus ça change, plus c'est la même chose.*
  -- Alphonse Karr

- *The crash of the European Space Agency's Ariane 5 launcher in June 1996-$500 million up in smoke in a few seconds-was entirely due to the reuse of a routine without the proper assertions (the proper contract)*

- *my initial "larval" phase of clumsy class hierarchies*
Specifications and Structure of OOPS:

• **Fortran**
  – Pascal 1970’s

• **Simula: 1960’s**.
  – Smalltalk 1970’s.
    • Python 1990’s
    • Ruby 1990’s
  – C++ 1980’s
    • Java 1990’s
    • C# 2000...

Variables are pointers to The Objects.

Function Stacks.. Of Procedural Programs..

ClassDescription

Behavior

Class description contains the methods And class hierarchy..

Objects contain The variables/instance data And pointer to its class
Smalltalk.. Can be Java/ C# too

An applications development platform:

Objects: Objects (class) description
- Object header and data
- Object state
  (includes contexts as objects)

• Compiler
• Interpreter

• Memory handling aka memory reclamation
• Primitives

• Numbered <primitive: 123>
For user extensions use:
• DLLCC  <c: void func( int ...)> new
• Named  <primitive: ‘a’ module: ‘b’>
Oops ptr ( int*)
Class oop
Size, hash, gcflags..
Indirection pointer
Inst var1
Inst var2
Inst var3...
Class oop
Size, hash, ...
Indirection pointer
Small Integer oop
Stores the integer value with 1’s in the 2 low order bit

32 bit integer pointer

Small Integers ...
integerValueOf
integerObjectOf

Byte1, 2, 3, 4
5, 6, 7, 8
.......
What..?

• What is OOPS theoretically, is understood by all..

We shall not talk of these.. But presume fully comprehended by all... :
  – Encapsulation, inheritance, polymorphism, Abstraction
  – Decomposition and Aggregation
  – Language inclusion of Classes, Methods/Functions or Messages, attributes or variables
  – Access specifiers: private, public and protected
  – Pure OOPS languages vs semi OOPS
  – Mutable vs Immutable Objects/ Classes..
  – Properties
  – Constructors and Destructors
  – Operator Overloading

If you have any doubts on any of these.. Google or clarify now.. Each of these encompasses years of refinement and understanding that is not easy to realize.. Unless you have done it a 100 to a 1000 times over, it is still the scratch of the surface. So clarify in depth all of these till you have exhausted all aspects of each of them..
The Simplest OOPs Example

• **A car:**
  
  **Attributes** of size, color, price, speed et als... these are stored as variables
  
  **Operations**: carDrive, turn, accelerate are the function calls(methods).

If modelled appropriately this car Object gives all the information required of distance covered, petrol consumed, time taken, max speed, etc...

**Composition** of the Car from various other objects with their own attributes, methods say the tyres, engine, etc.... thus change the type of one object (say the engine) and the attribute/operations of the car object simulated will change.

**Note:**

• *If you have not noticed this does not take away the rigours of the mathematical modelling that will have to be done, but simplifies expression and development of the program to do the simulation which is rather more arduous in procedural code. The documentation is rather less tedious and facilitates easier maintenance of the application on the long run.*
A Larger Example

- **Military Simulation package JWARS for US Defence Dept.**
  Battlefield objects: the tank, the army personnel, the helicopters, the missiles, the ships, just pick all entities and they are termed as object classes.
  Now the simulation exercise is in modeling what if scenario’s based on empirical statistics. Simulate the movement of men and resources to various battle spots, obtain the feedback on the relative strengths of the opposing armies, the potency of ballistics from their position and its impact.
  Also this gives a fairly good picture of costs, time and obstacles if any that can be foreseen given known parameters.

**Note:**

- *Suddenly you can notice OOPS simplifies even understanding the concept of a large application, not that building it is simplified that very much, as the same pitfalls exist as before, viz: a poor attribute definition or method (action) description can lead to hilarious situation like the missile hits its target and comes back to its silo..!!.*
Break Out..

- Do a quick OOPS example on a paper at hand.. Pair programming.. XP Style 15 min..
  - Capital Markets: Buying and selling Shares
    - Consider the simplest scenario and problem domain
      - Stock Exchange: As the listing agency for all Cos
      - Stock Broker: As the agent for share trade of individual, sub-broker and FI types
      - Customers: Of individual and corporate types
      - Listed Companies: Types of Companies, attributes appropriate to SE/Stock trade
    Shares of the simplest types.. Let us not confuse with other types and possibilities currently.. But let the boundary be diffuse in design to account for them in future..
    Actions: Customers Buy, Sell shares
    Shares listing at any instance and at Start of Day/ EOD
    Broker forwards/Stock Exchange executes trades
    Stock Exchange owns and maintains the backend persistent data storage..

Review on the Board..
The Exercise in detail: Understanding OOPS in action

- All the UML: Use case/Class diagrams
- Workflow, Sequence, State Chart Diagrams
- Typical Code layout.. In Modules/Classes and methods..

The diagram images are just spoofs and the exercise is to be done live on the board during the lecture..
Before OOPS

OOPS borrows from the evolution path..

• Procedural Programming..
  – Procedures, modules, structs, ADT....

• Function Interfaces..
  – DLLs and COM world..
  – RPC (and now SOA.. WebServices..)
Pitfalls of OOP

• If you are wanting the bare metal performance: The Graphics, network operations et als.. Choose the procedural path.. C is the best bet.. But you may need to do so for barely 10% or 20% of your code.. Not the entire application per se if it’s a business application requiring such an interface
• If it’s a one off application and needs no scalability, flexibility etc built in but pure performance and you are great in Haskell, Eiffel or Lisp or whatever you are good at.. Give it a shot and do it.. In that language, don’t bother getting bogged down by OOPS if it does not suit your taste.
• the fragile base-class problem
  [real object collections aren't always hierarchical.
  the piece of paper on my desk doesn't have discrete methods. If I decide for example, to burn it for fuel, or fold it into a paper airplane, does that mean that there is a "burn" or "fly" operation that's somehow built into the paper, and that it inherits these operations from a superclass of "flat things"? There are an almost infinite number of things I can do with a simple piece of paper, none of which may have been anticipated by the creator of that paper.
• I've seen many cases where a programmer will call a particular function pervasively throughout their code, thinking that the routine is fast, when in fact it is slow.
• Confusion of is-a, has-a, and is-implemented-using relationships;
  – Often, inheritance is misapplied when the designer really means "has-a" or "is-implemented-using-a"..
• Confusion of interface inheritance with implementation inheritance;
• Use of inheritance to violate encapsulation;
• Use of Multiple inheritance in most circumstances. ]

OOPS Thoughts..

- Composition is more important than inheritance. You need inheritance to support some things you do with composition, but it's composition that's the key concept. Keep your inheritance hierarchies as flat as possible.
- Interfaces are important. Implementation inheritance is useful and handy at times, but it's all about sharing interfaces. I don't even mean explicit interfaces here, see my comment about dynamic typing later.
- OO doesn't really shine in C++. No, I take that back. OO really doesn't shine in C++. C++ is too complex. Only use C++ when you really really need it; i.e. you want the raw speed and the hardware access while still being able to use some OO abstraction techniques. You need that far less than you may think. C++ gives people the wrong idea about OO. I know, as C++ was my first OO language too.
- OO really shines in a dynamically typed language. I myself like Python, but the classical example is Smalltalk. In a statically typed language it's usually much harder to make or change a class; you have to work a lot to get the interface right, it takes lots of time to change an interface, and so on. In a more dynamic language you can do rapid prototyping. For instance, you can easily change an interface or create a new one; you don't have to make interfaces explicit from the beginning. Making a new class shouldn't be that much harder than making a new function.
- OO libraries are often better when they're thin layers above some procedural subsystem than when they're a thick layer. That's because it's hard to make a good generic OO library, and bad OO libraries probably hurt more than bad procedural libraries.
- Collections are good. Use them. That is, lists and dictionaries/hash tables. Collections are even better in dynamically typed languages; you basically get the STL for free without any of the complexity. Generic code without templates or explicit interface constructs or multiple inheritance.

Ref: [http://www.advogato.org/article/83.html ]
OOPS Thoughts..

- **Axiom: Modelling and design are hard**
  True enough. It is not easy to design and implement systems that are general, flexible, fast, extensible and accommodating of a broad range of unknown future requirements. This is hard with or without OOP.

- **Corollary 1: It's hard to live off inheritance alone**
  Design is indeed even harder if one constrains oneself to using a single approach for all modelling - not everything fits in a strict inheritance hierarchy. While inheritance is often overemphasised in introductory OO texts, it is not (nor is it intended to be) a design panacea. It is also not the one thing that defines an OO design - the author mentions composition and other ways to achieve delegation - these are perfectly valid OO techniques and there is no OO law that states reuse through inheritance or design by inheritance are holy grails to be pursued at all costs.

- **Corollary 2: Achieving durable abstraction is hard**
  The benefits of modularity (encapsulation, data hiding, yyy) and the complexities of realizing them are fundamental issues that have been discussed since well before OOP became popular. See, for instance, the works of Parnas and Dijkstra. OOP attempts to provide some basic tools to express and enforce encapsulation - coming up with a good decomposition is still the (hard) job of the programmer.

- **Corollary 3: C++ is hard**
  C++ is a large, complex, powerful language. Among its design goals is providing OO facilities while maintaining the performance and down-to-the-metal freedom of C. This is a tall order and while C++ delivers, it does so at a price - effective use of the language requires what amounts to 40 years of ferocious training in the Arctic with Doc Savage. This is necessary if you want to or have to get the most out of C++ - but C++ is not the only road to OOP.

- **But it ain't real**
  Reality might be spikes in my visual cortex, it might be break-dancing wave-functions. While OOP's roots are indeed in simulation, the approach has shown itself to be useful in modelling a variety of systems. There is no strict requirement to always directly map classes to concepts. It would be naive and restrictive to interpret and evaluate OOP on the quality of its representation of human perception. It doesn't have to be real - it just has to be useful.

Ref: [http://www.advogato.org/article/83.html ]
Benefits of OOPS

- **Faster Development**: OOD has long been touted as leading to faster development. Many of the claims of potentially reduced development time are correct in principle, if a bit overstated.

- **Reuse of Previous work**: This is the benefit cited most commonly in literature, particularly in business periodicals. OOD produces software modules that can be plugged into one another, which allows creation of new programs. However, such reuse does not come easily. It takes planning and investment.

- **Increased Quality**: Increases in quality are largely a by-product of this program reuse. If 90% of a new application consists of proven, existing components, then only the remaining 10% of the code has to be tested from scratch. That observation implies an order-of-magnitude reduction in defects.

- **Modular Architecture**: Object-oriented systems have a natural structure for modular design: objects, subsystems, framework, and so on. Thus, OOD systems are easier to modify. OOD systems can be altered in fundamental ways without ever breaking up since changes are neatly encapsulated. However, nothing in OOD guarantees or requires that the code produced will be modular. The same level of care in design and implementation is required to produce a modular structure in OOD, as it is for any form of software development.

- **Client/Server Applications**: By their very nature, client/server applications involve transmission of messages back and forth over a network, and the object-message paradigm of OOD meshes well with the physical and conceptual architecture of client/server applications.

- **Better Mapping to the Problem Domain**: This is a clear winner for OOD, particularly when the project maps to the real world. Whether objects represent customers, machinery, banks, sensors or pieces of paper, they can provide a clean, self-contained implication which fits naturally into human thought processes.

REF: [http://www.westga.edu/~bquest/1997/object.html]
Language Elements

• Modularization in OOPS..
  • Based on the functional description of the application
  • namespaces, packages..
• Interfaces.. Vs Abstract Classes..
  – Better implement contracts of operations..
• Delegates, Blocks, Events...
  – Functional Programming..
• Mutable vs Immutable Objects/ Classes..
• Aspect Oriented Programming
  – programming paradigm that increases modularity by allowing the separation of cross-cutting concerns, viz: Transactions, security, and logging etc..
Process and methods of OOPS

DIY and do it many times over. Probably 100 times to begin to comprehend the ease and complexity of OOPS. Start small and grow gradually. Consult often

- **Requirements Gathering.**
  - More Object oriented, talk of entities, nouns, operations/verbs etc. Capture more visually the various modules and their functionality
- **Design and Analysis: the grunt work**
  - Break it down visually: If it is not entirely visual it is not Object Oriented.
    - Modular decomposition
    - UML diagrams: Class diagrams, UML use case diagrams
    - Workflow, State Charts, Sequence diagrams
    - Objective point wise listing of requirements mapped to the design section
    - Interleave with prototypes.
- **Implementation:**
  - Code: Choose the right language
  - Translate the modules and UML/diagrams to code syntax
  - Prototype as much as required to finalize the appropriate D&A
  - Identify and leverage reuse
- **Testing**
  - Using Units testing and Testing team leverages the Object world from the design phase.
- **Delivery and Release**
  - Typically environment/platform dependent.
- **Change Management**
  - Object orientation should give the best leverage in change management
- **Reuse management**
  - Commit reusable components to a reuse library
Applying OOP in practice

• Development Projects: Ideal for the experienced..
  – From scratch
  – As a module or extension to existing application

• Maintenance Projects: ideal for the learner..
  – Projects in non OOP language: COBOL..??..C..
  – Projects in OOP:
    • Study and comprehensive understanding of the application should be a pre-
      req
    • Propose changes to the structure and implementation if you have mastered
      the application enough to do so..

• Testing Projects: Testers you too be Object Oriented
  • Review and understand the entire application module/sub systems in their object
    decomposition and aggregation to create appropriate regression/module test
cases

• OOPS in WebApplications, GUI Desktop Applications, Client-Server
  Application, Networking TCP-IP, Embedded Programming.... A
discussion..
Extensions to OOPS

• Virtual Machine: Garbage Control, Process Control, Graphics Control etc..
  – “Each Program is then an OS in itself..”

• Design Patterns..

• Reuse Library

• Frameworks.., Libraries...

Beyond OOPS... aspects of bare OS interactions, current genre of Workflow/Rules etc.. Just to position it in the future context...
Mandatory initial reading for all: Basics of OOPS:

- http://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info/ReadMe.html  ALL THE 7 CHAPTERS..

C#
- http://www.blackwasp.co.uk/CSharpObjectOriented.aspx

C++
- http://gd.tuwien.ac.at/languages/c/c++oop-pmueller/

Java:

Smalltalk:

Python:
- http://www.voidspace.org.uk/python/articles/OOP.shtml

Ruby:

Read this if you can:
- **Well even in C** as it is cooked up by this author.. very interesting and extremely informative if you follow ANSI-C:

Misc: Just to catch up and ensure you know all the jargon there is to anything related to OOPS..

History: If you are really deeply interested and want to engage in the philosophy and the arcane debates of CS:
- http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en
- http://dreamsongs.com/ObjectsHaveFailedNarrative.html