

# Lecture 1

## Introduction to Object Oriented Architecture, Design, and Patterns

“Every message is, in one sense or another, a simulation of some idea”—Alan Kay and Adele Goldberg, creators of the Smalltalk Language

# Prolegomena to Any Future Discussion

- Lasciate ogne speranza, voi ch'intrate. –Dante
- ἄρμονίαν ἀφανῆς φανερῆς κρείττων. –Heraclitus
- When we start cataloguing the gains in tools sitting on a computer, the benefits of software are amazing. But, if the benefits of software are so great, why do we worry about making it easier--don't the ends pay for the means? We worry because making such software is extraordinarily hard and almost no one can do it--the detail is exhausting, the creativity required is extreme, the hours of failure upon failure requiring patience and persistence would tax anyone claiming to be sane. Yet we require that people with such characteristics be found readily and employed cheaply. - Richard Gabriel, paraphrased

# Prolegomena to Any Future Discussion

- Why has the object-oriented style become so popular? Certainly no small part has been played by the tendency of programmers to jump on to the latest “fad” language. However there is real substance behind the reasons for the increasing use of object-oriented languages. There seem to be clear advantages for the object-oriented style in organizing and reusing software components. . . . However, in many ways the quality of object-oriented programming languages falls short of existing procedural and functional languages—Kim Bruce, *Foundations of Object-Oriented Languages*

“Is there one personality type that is better suited to object technology than all the others?”

“This question is relevant to your project because it appears the answer may be yes...The ideal object designer/programmer thinks *abstractly*, deals well with uncertainty, and communicates reasonably, in contrast to the quiet, detail-oriented programmer stereotype. Good designers rely on the ability to look at things in the world, at *words* and *concepts*, and to extract from them properties that they share, that can be *named*, and that can be *classified* into a hierarchy. An object called a “strategy” is by no means easy to think up, and yet it becomes one of the most powerful elements in the system. What are the properties that all your user-interface screens have in common? The quality of the answer depends on the ability to identify commonalities and *imagine* various futures. Thus, the quality of OO design hinges on *abstract* thinking.”

—Alistair Cockburn, *Surviving Object-Oriented Projects* (*italics mine*)

# OO in a Nutshell

Understanding Object technology is not about strange and impressive words like classification, encapsulation, polymorphism, inheritance—these are but means to an end. Object Oriented thinking is about the creation of a language, and those who succeed will understand this. To talk about a class is to talk about a concept, to create a new word, and to define how that word relates to the rest of the language. Those who succeed will be able to speak this new language, modify it, work within its confines when possible and extend it when necessary. It is this ability to create a language and actually use it that separates great OO designers from mediocre ones. And the successful communication of, and facility in, this new language will separate successful projects from failures.

# First, the Bad News

- In 1995, 40% of projects initiated *failed* to deliver and were cancelled or shelved
- Again in 1995, an additional 33% of projects were delivered either late and/or over budget
- For every single minute [www.schwab.com](http://www.schwab.com) is down, the estimated cost is \$1M
- A single software bug in American Airline's scheduling system cost \$50M in lost revenue
- Projects that “succeed” and are actually delivered are delivered with only about 40% of the originally specified requirements
  - Housing Analogy
  - Bill Gates Vignette

# First, the Bad News

- Lest you think this is just old data (and agile programming has saved the day):
  - In 2011, 37% of projects initiated were identified as *at risk* of failure
  - 62% failed to meet their delivery schedule
  - 49% suffered budget overruns
  - 47% had higher-than-expected maintenance costs
  - 41% failed to deliver the expected business value and ROI

# More Bad News

- Between 1985 and 1987, a software-controlled accelerator massively overdosed 6 individuals with 10-25K RADs (normal therapeutic range is around 200-500 RADs), with 4 consequent fatalities (<http://sunnyday.mit.edu/papers/therac.pdf>)
- On February 25, 1991 a Patriot missile failed to track a scud missile due to a code bug that had been “fixed” in “some” parts of the code but not in others, resulting in 28 deaths (<http://www.ima.umn.edu/~arnold/disasters/patriot.html>)
- Mars Orbiter: “The 'root cause' of the loss of the spacecraft was the failed translation of English units into metric units in a segment of ground-based, navigation-related mission software” (<http://mars.jpl.nasa.gov/msp98/news/mco991110.html>)

# IF ([.NOT.]WORKING) THEN ... CALL SELFDESTRUCT

- In 1994, Fidelity Investments was unable to calculate the NAV for 2/3s of its mutual funds because a software bug had overwritten most every stock in its database with the digit '9'.
- In 1989, a police computer in Paris mistakenly sent out letters charging over 40,000 *traffic offenders* with crimes ranging from murder, drug trafficking, extortion, to prostitution.
- An early F16 autopilot would flip the jet upside down whenever it crossed the equator
- The Mariner 1 launch to Venus failed due to a period instead of a comma in its FORTRAN program's DO statement
- A bug in GE Energy's XA/21 system caused a blackout in the Northeastern United States in 2003

# Recent Personal Favorite

- ATM, Süddeutschen Zeitung, 09.01.2002:  
Glückspilz mit Geldsegen ohne Geheimzahl  
(  
[http://www5.in.tum.de/~huckle/  
bugse.html#atm](http://www5.in.tum.de/~huckle/bugse.html#atm))
- For even more, see:  
<http://www5.in.tum.de/~huckle/bugse.html>

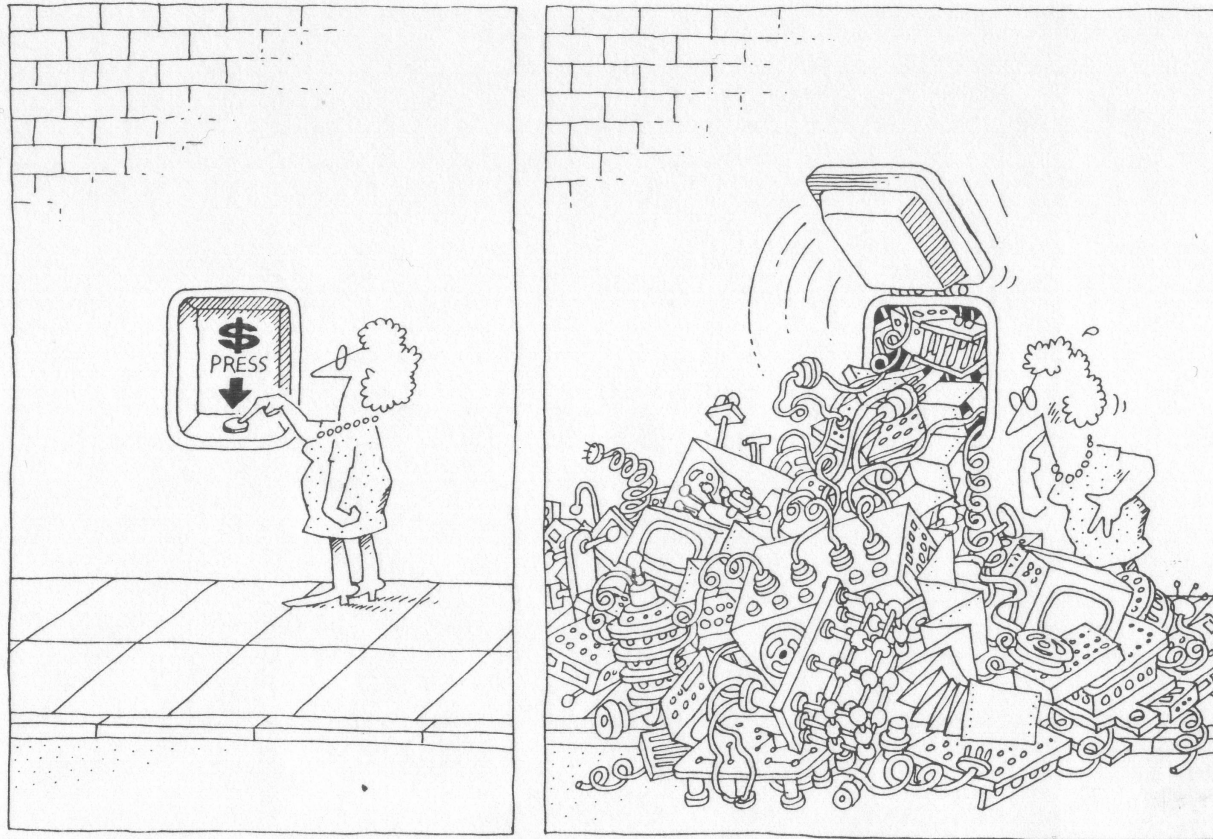
# Why is this stuff so hard?

- Lack of Domain Knowledge on the part of “the computer people”
- Users don't know what they want (until they see it)
  - Corollary: “Details hurt my head”
- Lack of expertise in languages (UML, C++, etc.)
- Users may not *want* the system
- Designing from an already broken business model
- Lack of design experience on developer's part
- Lack of adequate risk management
- Politics
- Culture
- Complexity

# Complexity

(from Booch, Object Oriented Analysis and Design, Benjamin Cummings, 1994)

*The First Section: Concepts*



The task of the software development team is to engineer the illusion of simplicity.

# Complexity in Software Development

- “The complexity of software is an *essential* property, not an *accidental* one.” -- Frederick Brooks, 1986, “No Silver Bullet”
- Accidental versus Inherent Complexity (Aristotle)
  - essential complexity (inherent/conceptual/design/what)
    - "The essence of a software entity is a construct of interlocking concepts . . . This essence is abstract.”
    - The complexity of the design itself is essential. Everything else is accidental.
  - Accidental complexity (secondary/predicative/implementation-oriented/how)

# What is “Accidental Complexity?”

- Faster compilers, better debuggers
- Better and simpler languages
- Better integrated tools (IDEs, OS environments)
- Structured Design and Object-Oriented Design
- Better processes: Incremental, Iterative processes
- CASE tools

# Essential Complexity

- “The essence of a software entity is a construct of interlocking *concepts* . . . This essence is *abstract*.”
- Object Technology calls “for a different kind of software developer, one comfortable with *abstractions*, uncertainty, and communication.” – Alistair Cockburn
- Human communication is inherently complex and exposes the limited ability to visualize the problem, much less its solution
- Infinite states (at the microcosm level)
- “The part of software building I call essence is the mental crafting of the conceptual construct; the part I call accident is its implementation process.” – Frederick Brooks

# 3 Areas of Complexity

- Problem Domain Knowledge and Communication (Data Models, Business Models, etc.)
- Development Process and Management
  - Scalability (hardware, software, personnel)
- Software Infinity in Discrete Systems
  - Infinite interpretations
  - Physical laws do not apply in software: In the physical world, a tossed ball will always fall.
  - Building metaphor ultimately fails
  - Unpredictability

# Controlling Complexity

- “Hi-level” Programming Languages
- Modular Design
- Modeling
- Abstraction and Classification
- Patterns and Repetition

# Design Patterns

“Making abstractions which are powerful and deep is an art. It requires tremendous ability to go to the heart of things, and get at the really deep abstraction. No one can tell you how to do it in science. No one can tell you how to do it in design.”

—Christopher Alexander

# Christopher Alexander & the Pattern Movement

- *The Timeless Way of Building & A Pattern Language, 1979*
- The "quality without a name"
  - aliveness
  - grounding
  - at-homeness
  - sense of belonging
  - comfortable
  - whole
  - natural
  - discovery of truth

# The Discovery Process

- Patterns are discovered.
- Patterns are interrelated (they collaborate in design)
- the *invariant*: that nexus, the *point d'appui*, the centre point that does not change, the "key" the the pattern.
- A Pattern Language is a catalogue of known and documented patterns.

# Design Patterns

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." -Christopher Alexander
- Design Patterns are named architectural models
  - created by an assumed/presumed expert
  - that consist of a generic collaboration of classes
  - that apply across a broad range of problem domains,
  - that model a solution to some problem in one or more of those domains.
- A micro architecture, representing a known solution to a recurring problem across a number of domains.

# What is a Design Pattern?

- A design pattern is a reusable implementation model or architecture that can be applied to solve a particular recurring class of problem.
- Patterns provide a succinct and often descriptive vocabulary within which to convey design concepts.
- Think of a tailors pattern, where an expert tailor creates a design, and provides the blueprints for that design, so that less experienced seamstresses can copy that pattern and make a fairly decent dress, perhaps not Versace or Liz, but at least a wearable garment.
- Patterns represent a formal solution to a common problem based on expert OO know-how (Omar Sharif and Bridge, the chess column)
- Patterns are independent of any particular implementation language
- The pattern is one of the primary OO design units of reuse.

# Pattern Structure

- Pattern Structure (one example):
  - Pattern name: short noun or noun phrase
  - Problem statement: What problem needs solving? What are the forces at play in this problem?
  - Solution: A design which purports to resolve the forces at play in the problem statement, offering in the end a solution to the problem.
  - Consequences: Implementation trade-offs. What are the benefits of this pattern? What are its risks and complications?

# How Do Architectural Patterns Differ?

- Architectural Patterns:
  - Are designs that rise above the level of the algorithm or function, or even class
  - Focus on systems structures that affect the entire system or a substantial part of the system, offering mechanisms for system to system communication, synchronization, translation, data access
  - Are often delivered as a framework that provide system capabilities to all the classes in a system
  - A Solution to high-level problems at the system level, not at the class or object level

# Examples of Architectural Patterns

- Pipes & Filters
- Layers
- Message Queues
- Blackboard
- Broker
- Microkernel
- Reflection
- Model-View-Controller

# Great OO Myths

- We know C, so we' ll just write everything in objects using C++
- We need lower maintenance costs. Let' s do OO and reuse everything
  - Cockburn calls OO reuse a “diabolically difficult topic”: “Reuse’ is too simple a word...it should be unpronounceably difficult, to give the sense of how hard it is to achieve.”
- We can use our same process, and just do OO instead of procedural programming
- Our programmers are so smart we' ll be up to speed with objects in 3 months
  - Dave Thomas of Object Technology International calculates 9 months for every new hire.

# Great OO Myths

- OO is just another programming language
- What's all the fuss? Just model the real world
- Maintenance costs will go down because of the features of OO such as inheritance, polymorphism, etc.
- OO is nothing but a graft on current procedural technologies