



Object Oriented Architecture: Patterns, Technologies, Implementations

Lecture 1

Introduction to Architectural and Design Patterns

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 all about the creation of a language, and nothing more, and those who succeed will understand this. To talk about a class is to talk about a new 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, identify patterns within this language, modify it, work within its confines when possible and extend it when necessary. It is this creation of a language, this language instinct, that defines us as human, and it is the ability to create a language and actually use it that separates great OO designers from mediocre ones.

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.
- 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

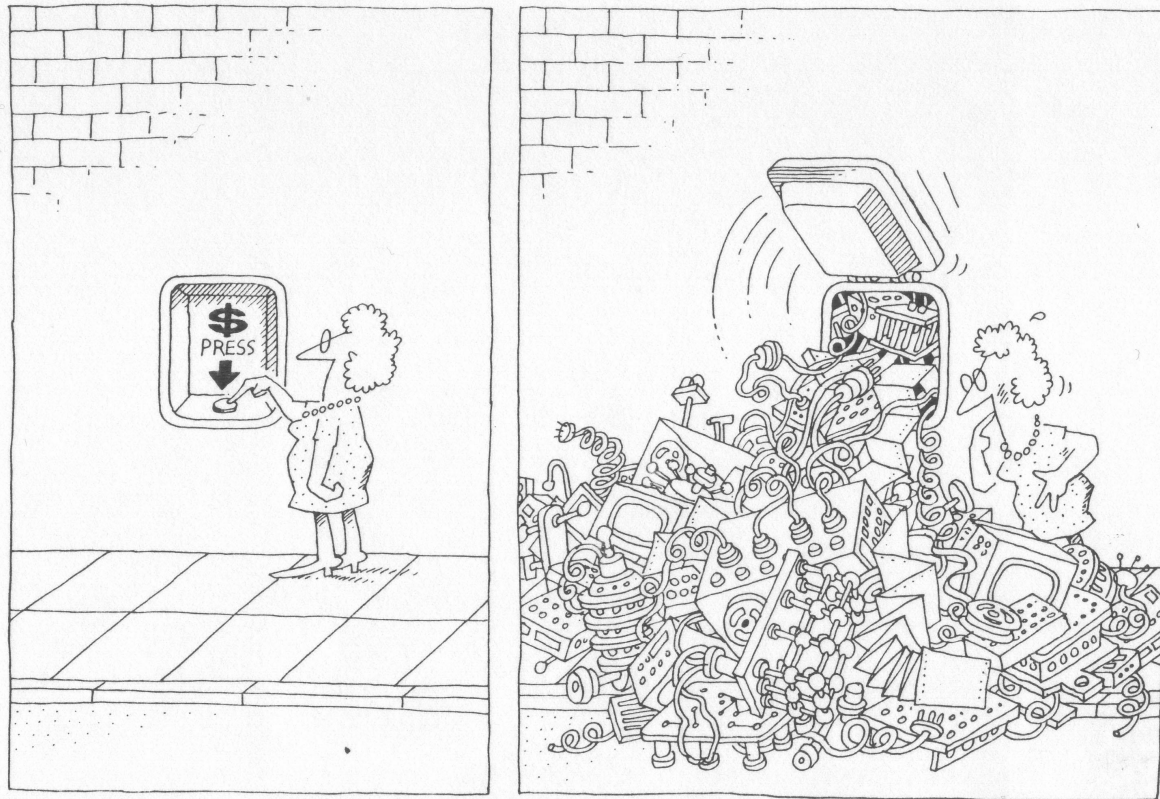


Why is Software Development So Hard?

- Lack of domain knowledge
- Users often don't know what they want until they see it
- Lack of expertise in languages (UML, Java, C++, Smalltalk)
- Lack of expertise in technologies (CORBA, J2EE, etc.)
- The Customer may not want the system
- Designing from an already broken business model
- Lack of OO design experience on developer's part
- Insufficient or non-existent risk management
- Politics (corporate, personal, interdepartmental, technological, etc.)
- "This prototype looks good. When's it going into production?"
- Complexity

Complexity

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)
 - "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)

3 Areas of Complexity

- Problem Domain Knowledge and Communication
- Development Process and Management
 - scalability (hardware, software, management)
- Software Infinity in Discrete Systems
 - Infinite interpretations
 - Physical laws do not apply in software: In the physical world, a tossed ball will always fall.
 - Unpredictability



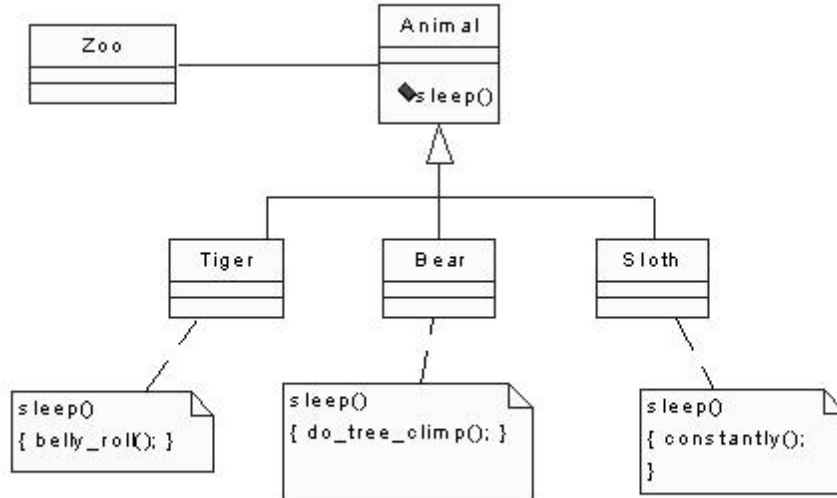
Controlling Complexity

- Modeling
- Modular Design
- Abstraction and Classification

So What is Inheritance?

- Inheritance implements an “is-a” or “is a type of” relationship:
 - A man or woman is an animal
 - A bubble sort is a type of sorting algorithm
 - A driver’s license is a type of official documentation
 - A salaried employee is a type of Employee
 - A manager is a type of salaried employee
- Inheritance is a way of reusing code and/or interfaces of a class
- Inheritance can be either singular or multiple:
 - A professor is an employee, a student is a person, and a teaching assistant is a type of professor *as well as* a student. A teaching assistant *is both* a professor and a student at the same time.
- Back to simulation and imitation: inheritance allows a design to mimic the real world

Simple Inheritance



- A Zoo has animals
- Individual Animals know how to sleep
- Abstract sleep into base class, so you can command *any Animal* to sleep
- *Implement* `sleep()` in concrete derivatives so each individual animal knows how to sleep

Inheritance continued

- Inheritance is also often called a “Generalization/Specialization” relationship (Coad)
- base class: generalization
- derived class: specialization (it specializes or extends the base class)
- Canine (generalization)
 - Collie (specialization)
 - Shetland Sheepdog (specialization)
 - Terrier (generalization and specialization)
 - Boston, Bull, Fox, Scottish, Clydesdale, Dandie Dinmont

Single vs. Multiple Inheritance

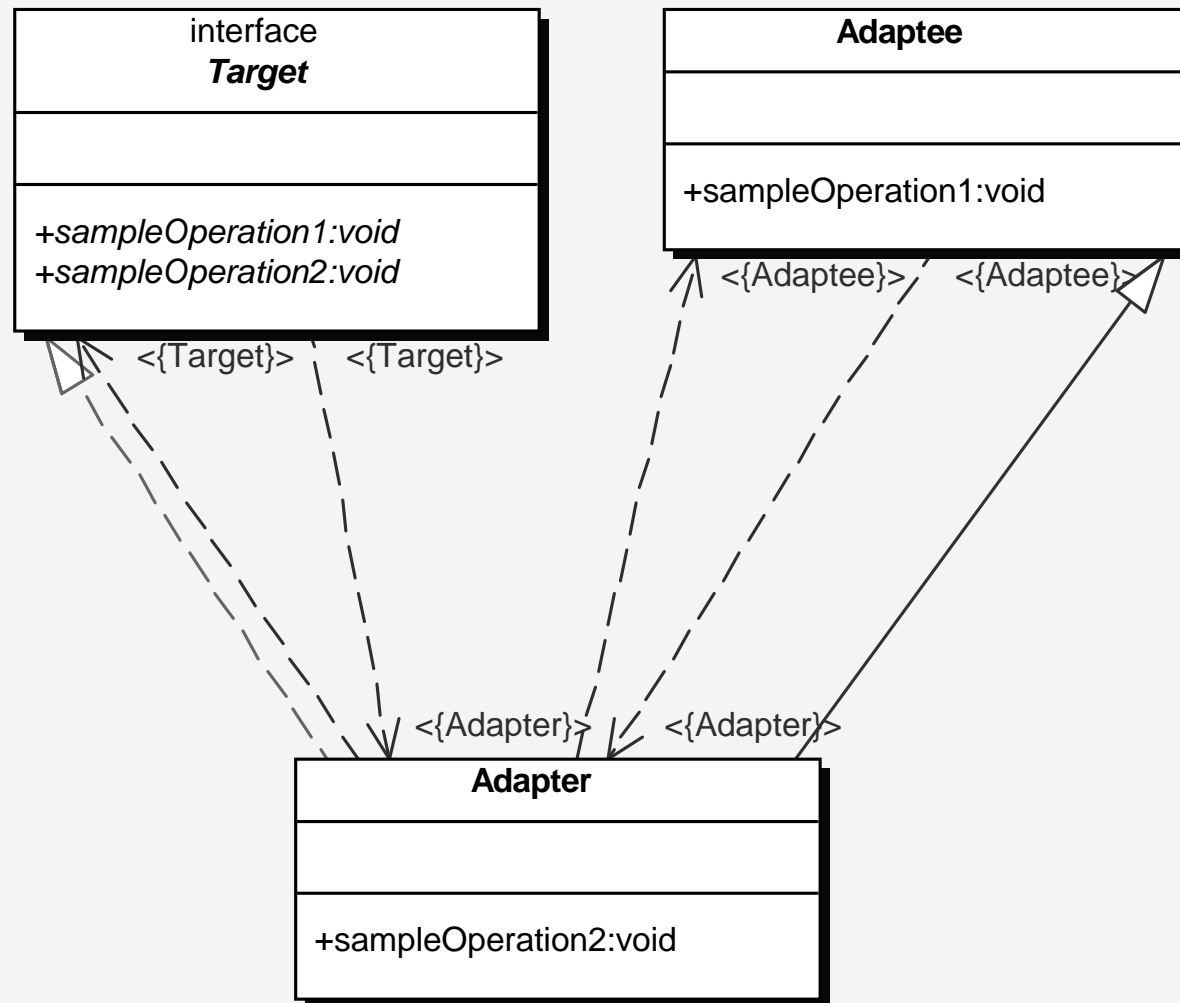
- Single: Java, Smalltalk
- Multiple: C++, Eiffel, CLOS
 - Base A: Car
 - Base B: Person
 - Derived C: Car_Owner (derived from both Car and Person): is this legitimate?
- Other examples:
 - Airplane + Corporate Asset <-- Company_Plane
 - Boat + Plane <-- Hydroplane
 - Vehicle + House <-- Mobile_Home
 - Tax Exemption + Infant <-- Well_Loved_Baby

Composition

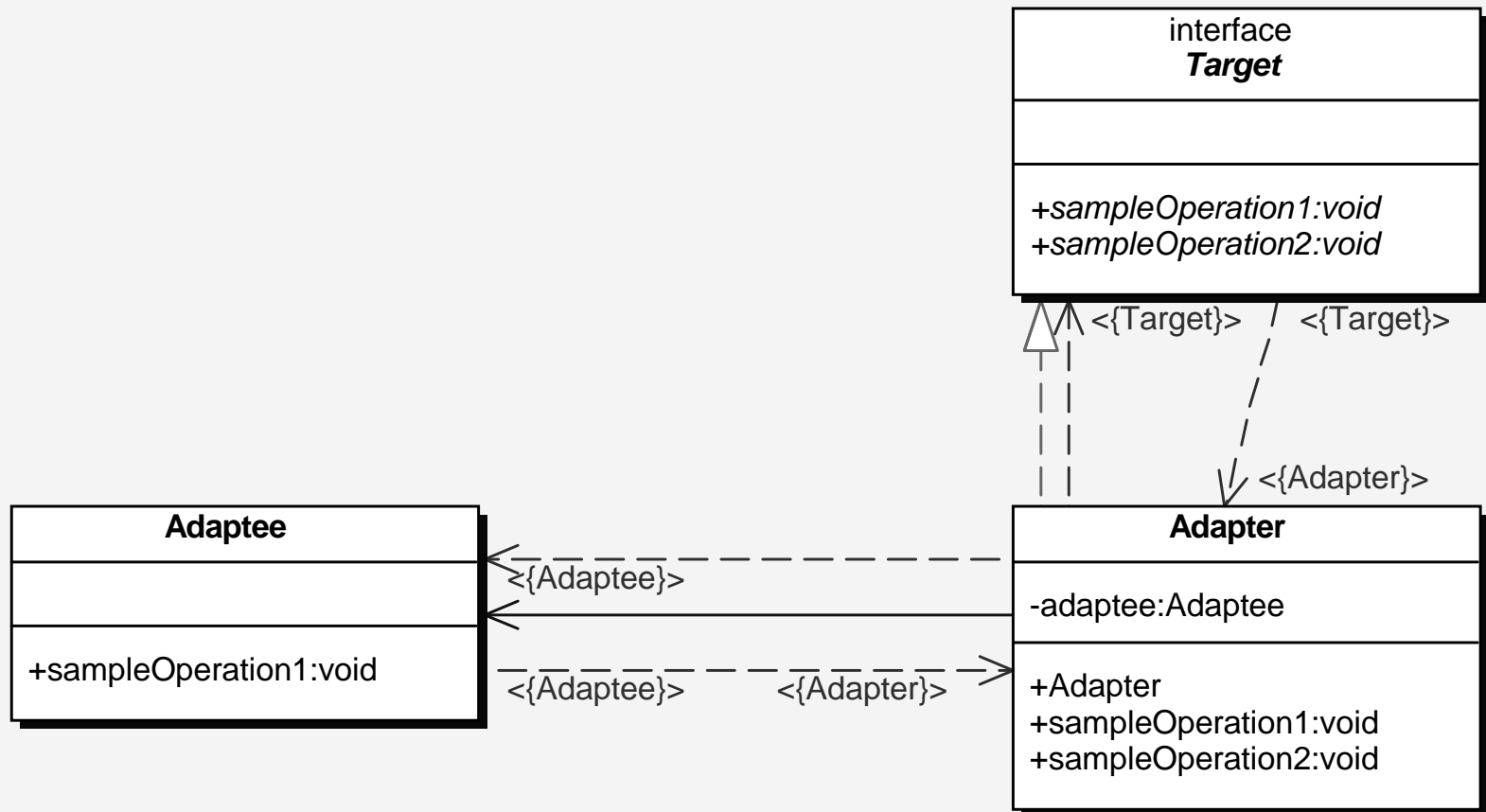
- A stronger form of aggregation, which “implies”:
 - that the lifetimes of the whole and part are simultaneous (implicit construction/destruction of parts along with whole)
 - the nonshareability of parts belonging to a single whole (No my dear, that’s my hand you’re holding!)
- Longest lifespan relationship



UML (Class Adapter)



UML (Object Adapter)



Motivation

- An Adapter is used when existing client calls on one interface need to be redirected to a new interface, without changing the client code
- An Adapter facilitates this by imposing a middleman to accept the old call interface, while delegating out to some new object for the implementation

Benefits

- Using an Adapter allows the substitution of one implementation of an interface for another without affecting existing client code
- Using an Adapter allows one to reuse legacy systems that depend on otherwise outdated or absent libraries or systems

Gang of Four Composition vs. Inheritance

- Recommend Composition over Inheritance
 - Composition: strong form of aggregation implying simultaneous lifetimes
 - Square and its four sides
 - Circle and its radius
 - hand and its fingers (robotics)
 - Strong Composition implies:
 - a part cannot belong to more than one whole
 - concurrent instantiation and destruction with whole

Why use Composition over Inheritance

- Composition trades in reuse (through inheritance) for delegation (via reference), which allows for dynamic runtime configuration
- Inheritance generally ties you to a particular implementation
- This is limiting because:
 - you can't easily swap out that implementation for another at runtime
 - Inheritance can be seen to break encapsulation
 - Inheritance hinders your ability to “design for change”
 - Composition promotes delegation (distributed implementation)

Why Inheritance over Composition

- Inheritance makes global changes easier to make (change the base class, and eureka).
- Inheritance enforces type checking at compile time (in strongly typed languages)
- Delegation can complicate the reading of source code, especially in non-strongly typed languages (Smalltalk)

A graphic of a spiral-bound notebook with a grey cover and a white page. The spiral binding is on the left side. The word "Readings" is written in the center of the page in a black serif font.

Readings

Christopher Alexander & the Pattern Movement

- *The Timeless Way of Building & A Pattern Language*, 1979
- The "quality without a name"
 - aliveness
 - grounding
 - at-homeness (νοστος, νοσταλγια)
 - sense of belonging
 - comfortable
 - the whole is greater than the sum of the parts
 - whole
 - natural
 - discovery of truth (αληθεια)

What is a Design Pattern?

- A design pattern documents reusable models or architectures that can be applied to solve a particular fundamental and recurring class of problem.
- Patterns provide a succinct and 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 simultaneously independent of and dependent on any particular implementation language
- The pattern is one of the primary OO design units of reuse.

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
- "Patterns help you build on the collective experience of skilled software engineers. They capture existing, well-proven experience in software development and help to promote good design practise." –Buschmann et. al.

Design Patterns

- 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.
- Represent a micro architecture, representing a known solution to a recurring problem across a number of domains.



Benefits of Using Patterns

- Communication: Patterns promote a common vocabulary among designers
- Patterns provide reuse at the design level
- Community review: Pattern implementations tend to be standardized
- Patterns simplify documentation
- Patterns give beginners immediate “experience”
- Patterns help control “essential complexity”

Pattern Structure

- Pattern Structure (one example):
 - Pattern name: short noun or noun phrase
 - Problem statement: What problem needs solving?
 - Context: When to apply the pattern (languages?)
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?

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.

Caveats

- “Patterns don’t make the hard part go away. Instead, they will help people understand the complex problems, so they can be more easily tackled. They also free people from repeatedly solving the easy problems, allowing them to tackle the interesting hard problems.”
--Neil Harrison, “Potential Pattern Pitfalls, or How to Jump on the Patterns Bandwagon Without the Wheels Coming Off”

Pattern Languages

- “A pattern language is a system which allows its users to create an infinite variety of those three dimensional combinations of patterns which we call buildings, gardens, towns.” --Alexander
- Pattern languages are generative, they not only tell us the rules of arrangement, but show us how to construct arrangements which satisfy the rules.
- “Both ordinary languages and pattern languages are finite combinatory systems which allow us to create an infinite variety of unique combinations, appropriate to different circumstances, at will.” --
ibid.



Pattern Languages

- Each of these patterns is a field of relationships which can take an infinite variety of specific forms.
- Beautiful buildings are built from the patterns that exist in a common language, a language that everyone who participates in the building understands.
- “It is only because a person has a pattern language in his mind, that he can be creative when he builds.” --ibid.



Design Patterns

- Singleton
- Template Method
- Bridge
- Composite
- Visitor
- Iterator
- State

Architecture

- The “Architecture” category page at the WikiWeb says this:

“What’s to say?”

(that’s all it says)

Architectural Patterns

- Design Patterns are mid-level abstractions that generally focus on the *interaction* among various components or objects
- Architectural Patterns are abstracted another level up from design patterns, and often focus on *integration* of component tiers



Architectural Patterns

- Model-View-Architecture (borderline)
- Layers
- Pipes & Filters
- Message Queues
- Blackboard
- Microkernel and Reflection
- Broker and Publish-Subscribe