Design Patterns

Sungeun Byun and Derek Choi

13 Apr 2006

EE 382V - Software Architecture and Design Rationale

Outline

- Design Patterns
  - Using catalogued operational semantics of object interaction as an implicit representation of design intent
- Introduction
- Metaphor and Metonymy in Object-Oriented Design Patterns
- Augmenting Design Patterns with Design Rationale
- Design Patterns as Language Constructs
- Industrial Experience with Design Patterns
- Conclusion
  - Analysis/Commentary/Questions/Discussion

Introduction

- Strong tendency to reuse designs
- More experience -> more proficient
- Restricted to personal experience
- Little sharing of design knowledge
- Design pattern is a particular form of recording design information such that designs which have worked well in particular situations can be applied again in similar situations in the future by others
- Ward Cunningham and Kent Beck developed a set of patterns for developing user interfaces in Smalltalk
- Jim Coplien was developing a catalog of language-specific C++ patterns called idioms

Introduction

- Erich Gamma recognized the value of explicitly recording recurring design structures while working on his doctoral dissertation on object-oriented software development
- These people and others met and intensified their discussions on patterns
- Influenced by the works of Christopher Alexander
- Pattern : to encode knowledge of the design and construction of communities and buildings
- More meaning than the usual dictionary definition
- Description of a recurring pattern of architectural elements and a rule for how and when to create that pattern

Introduction

- Recurring decisions made by experts, written so that those less skilled can use them
- Describe more of the "why" of design than a simple description of a set of relationships between objects
- “Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” Gamma et al.
- Presented a catalogue of 23 design patterns, organized in three categories depending on the pattern’s purpose
- Creational patterns : concerned with object creation
- Structural patterns : the composition of classes and objects
- Behavioral patterns : concerned with the ways in which classes or objects interact and distribute responsibility

Introduction

Creational Patterns	
Abstract Factory	creates related objects
Builder	creates complex objects
Factory Method	creates subclasses
Prototype	exemplary object
Singleton	single instance
Structural Patterns	
Adapter	converts interfaces
Bridge	decouple abstraction and implementation
Composite	model recursive tree structure
Decorator	add responsibilities to objects
Facade	interface for a subsystem
Flyweight	save memory of similar objects
Proxy	surrogate for access control
Behavioural Patterns	
Chain of Responsibility	handle requests
Command	request as an object
Interpreter	interpret a language
Iterator	iteration cursor
Mediator	encapsulate interactions
Memento	snapshot of objects’ state
Observer	update dependents
State	change behaviour
Strategy	vary algorithms
Template Method	subclasses change algorithms
Visitor	represent traversal operations

Classification of design patterns by Gamma et al.

Introduction

- The software engineer makes use of a paradigm, i.e. a set of related concepts such as object, class, method, inheritance, etc.
- Inexperienced engineer : a small concept set, consisting of the concepts represented in the used programming language
- Experienced software engineers : access to larger concept sets, (typical data structures and algorithms course)
- Only way to get access to more advanced concepts is through personal, hands-on experience
- Ability to capture such implicit experience and make it shared by the (object-oriented) software engineering community
- Paradigm : to make a complex work understandable by organizing it into separable parts
- Patterns : try to describe relationships within and between the parts, not just the parts themselves
- Patterns are truly an architectural technique, not just a divide-and-conquer technique

Metaphor and Metonymy in Object-Oriented Design Patterns

James Noble, Robert Biddle, and Ewan Tempero

Victoria University of Wellington 2001

Overview

- Why do we need this classification?
- What is the Metaphor and Metonymy?
- What is the Direct Metaphorical Design?
- What is the Indirect Metaphorical Design?
- What is the Programmatic Patterns?
- Result of classification of patterns

Why do we need this classification?

- "A program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world" Lehrmann Madsen et. al.
- Core principle : an object-oriented program simulates the world
- Object-oriented is the "natural" way to program or to design
- Some kind of external "reality" to which the program can refer
- Many programs have no reference to objective reality
  -> modeling an "imagined reality" that does not exist, but still constitutes an external referent for the program
- Alternatively, perhaps we create that world in the process of modeling it
- There are no reality, even imaginary ones, until they are modeled in the process of systems development and the process of systems development calls them into being

Why do we need this classification?

A Picture of farm animals, and the corresponding UML

- Pigs, sheep, and cattle live in fields-> Grazing-Area class, one instance per field, and associate each Farm-Animal with one Grazing-Area, and so on
- Diagram shows how a farm system can be seen as a model for the real world

Why do we need this classification?

- A common fallacy : the "objects" in the world is the same kind of objects as the objects used in object-oriented programming
- Do all objects in the real world have encapsulation?
- Do they have interfaces?
- Polymorphism?, Constructors?
- What about "objects" in the real world such as "sunshine", "credit ratings", or "gravity" which do not seem to be the same kinds of objects as "cows" or "sheep"
- How can we characterize the relationship between the classes and objects in the program and the classes and objects in the (real or imaginary) external world?
- This is the reason why we need the Metaphor / Metonymy classification in Object-Oriented Design Patterns

What is the Metaphor and Metonymy?

signifier	referent	signified
"lion"	person	brave person
"lamb"	person	docile person

Metaphor

- Metaphor : Greek word for "transfer"
- Transfer meaning from one thing to another
- The Figure shows how metaphor functions in speech. In phrases such as "he's a lion!" or "she's a lamb!" we use words (signifiers) "lion" or "lamb" as metaphors for a person to signify that that person is brave or docile

What is the Metaphor and Metonymy?

signifier	referent	signified
"crown"	person	king
"law"	person	police officer

Metonymy

- "Metaphor is a figure of speech based on similarity, whereas metonymy is based on contiguity. In metaphor you substitute something like the thing you mean for the thing itself, whereas in metonymy you substitute some attribute or cause or effect of the thing for the thing itself" (Martin Secker & Waxburg)
- The signifiers "crown" or "law" applied to a person to signify "king" and "police officer" respectively

What is the Direct Metaphorical Design?

- Objects in a program can model objects in the real world. It is important to realize what "modeled by" means
- Clearly it does not mean the Bovine objects in the program physically eat grass, produce cowpats into which one can step, and contribute large volumes of methane and other gases to warming the biosphere
- The traditional way to describe this relationship is to say that the objects in the program are "abstractions" of the real objects
- For example, a stack is an abstraction that might be implemented by an array, a pointer, and some executable code
  - The stack is an abstraction because it elides many of the details of actual implementation
- Is Bovine object in the program an "abstraction" of a real cow?
- Is the object in the program "implemented" by a cow in reality?
- Are the objects in the program special kinds of cows which do not eat, excrete, or expire?

What is the Direct Metaphorical Design?

signifier	referent	signified
"lion"	person	brave person
"lamb"	person	docile person
Bovine object	cow	cow-object
Ovine object	sheep	sheep-object

Metaphor between programmatic object and external object

- Different description of the relationship between programmatic object and external object
- This kind of relationship can be seen as metaphor
- An object in the program is a metaphor for an object in reality, and part of the meaning of the object in reality is transferred to the object in the program
- Thus, while our Bovine objects do not eat grass, they may have an identity number, age, and weight attributes that model some features of the corresponding real cow
- Metaphorical because it is based on metaphor, and direct because the metaphors are being represented directly, using the constructs from most object-oriented programming languages or modeling notations

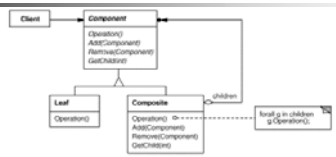
What is the Direct Metaphorical Design?

- Direct metaphorical design is the most common and most basic kind of object-oriented design
- Accounting system : one object per account
- Game : one object per demon, dragon
- Where direct metaphorical designs work, they are easy to produce, implement, understand, and modify
- One simply identifies the objects of interest in the real world and creates corresponding objects in the design model

What is the Indirect Metaphorical Design?

- Objects that are common in the worlds we wish to model, but that cannot be translated directly into object-oriented designs
- Objects in the real world are often created recursively from parts and wholes
- Large organizations are made up of smaller organizational units, and these units are composed of smaller units, in turn composed of still smaller ones
- Most programming languages and modeling notations don't support this kind of composite object, so these structures need to be encoded using the features of the available languages and notations
- Designers incorporate patterns into their program to address general problems in the structure of their programs' designs, in a similar way that algorithms or data structures are incorporated into programs to solve particular problems
- These patterns are often not intuitive to novice designers and programmers, although experienced ones may find them quite obvious

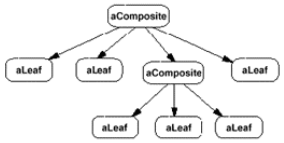
What is the Indirect Metaphorical Design?



The structure of the Composite pattern

- Composite pattern supports the organization of objects into part-whole hierarchies
- The resulting objects present a uniform interface to clients, whether the object is an individual object or a composition of objects
- Classes representing the whole composition are subclasses of classes representing the part ("Component")
- The most important class in this structure is the class representing the part, not the whole
- Recursive relationship appears as a many-to-one aggregation from a subclass Composite to a superclass Component
- To novices, this link appears backwards, going up the tree rather than down, from whole to parts, and the aggregation within an inheritance hierarchy appears completely arbitrary

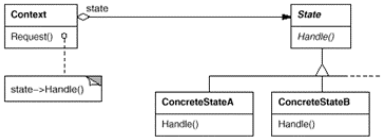
What is the Indirect Metaphorical Design?



The objects created by the Composite pattern

- Although the class diagram looks strange, the result structure of the objects is straightforward
- Indirect metaphorical designs. metaphorical because the relationship between the resulting objects in the program and the objects in the world is metaphorical for objects in the world, indirect because the program or modeling language (class) structures are not obvious
- The difference between direct and indirect metaphorical designs lies in the features of the languages used to express them
- Some languages lack sufficient features to express required metaphors directly, so they must be encoded indirectly, such as by a design pattern
- Translating between languages can change an indirect metaphoric design into a direct metaphoric design, and vice versa

What is the Metonymic Design?



The structure of the State pattern

- Some design patterns don't make sense considered as metaphors. Let's consider the State pattern, one of the simple patterns
- The State pattern is used in situations where the behavior of the object depends on the internal state of the object
- Thus, state pattern allows an object (the Context) to alter its behavior when its internal state changes, causing the Context object to appear to change its class

What is the Metonymic Design?

- The State pattern introduces an internal state object aggregated inside the context, and delegates some requests to it
- The internal state object is an instance of a ConcreteState class (where the ConcreteState classes all inherit from a common abstract State class)
- The behavior which the context object receives when delegating requests to the state object will change according to the ConcreteState object that is installed at any time, so by changing state objects dynamically the whole context object can provide different behavior
- The State pattern to record the changing state of a sheep from newborn lamb, dipping, crutching, dagging, hogget, breeding ram, and finally to mutton

What is the Metonymic Design?

- The implementation of the State pattern is quite straightforward, Just add an extra object and class hierarchy to design, and then change internal state objects to change context objects' behavior
- The State pattern raises an important question regarding the design or analysis of the program
- What object in reality does the state object represent?
- In the farm example, the state object certainly doesn't model a subordinate physical object that is attached to a sheep and that changes throughout the sheep's lifecycle
- There may be no physical change at all between a sheep considered a newborn lamb one day, a yearling the next, and a prime export candidate the day after
- State objects do not represent objects from the real world. They are not metaphorical, either directly or indirectly
- Rather, these objects and design patterns are exemplars of metonymic designs

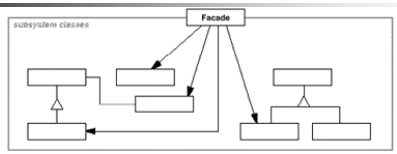
What is the Metonymic Design?

signifier	referent	signified
"crown"	person	king
"law"	person	police officer
SheepState	sheep	state of the sheep

Metonymy between programmatic object and external object

- The states of the sheep are not metaphors for "real" objects, they signify attributes of sheep
- Figure shows how an object in a program can be a signifier for some referent in the world
- One intuitive way to determine whether a pattern is metaphor or metonymy is to ask how hard the pattern is to explain
- Easy patterns that involve just one object tend to be metaphor (this is a composite object, this is a prototype which can be cloned)
- While the complex patterns involving multiple objects tend to be metonymy (this is part of the internal state of another object)

What is the Programmatic Patterns?



The Facade pattern

- There are some patterns that are neither metaphor nor metonymy
- The Facade design pattern is used to provide a single, integrated interface to a set of interfaces in a subsystem
- Facade defines a higher-level interface that simplifies the use of the subsystem, inserts an extra interface into a program to encapsulate a set of objects forming a subsystem
- Extra interface is typically nothing to do with any external reality, rather, it is purely about the internal structure of the software
- These kinds of patterns are programmatic, because they are about programs' internal structure rather than their relation to an external reality

Result of classification of patterns

- Design Patterns by Gamma et al. categorized patterns into three types
- Creational patterns are about creating objects, structural patterns about program structure, and behavioral patterns about program behavior
- This categorization seems orthogonal to our classification of patterns according to metaphor and metonymy
- Object-oriented design is primarily metaphorical
- Metaphorical designs that cannot be implemented directly in a programming or design language give rise to patterns (such as Composite) corresponding to indirect implementations of these metaphorical designs
- Modeling attributes, causes, and effects (rather than real world objects) produces metonymic designs and more advanced patterns
- Designs that improve the internal structure of the program without reference to an external reality give rise to programmatic patterns

Result of classification of patterns

Creational Patterns		
Abstract Factory	creates related objects	metonymy
Builder	creates complex objects	metonymy
Factory Method	creates subclasses	metonymy
Prototype	exemplary object	metaphor
Singleton	single instance	metaphor
Structural Patterns		
Adapter	converts interfaces	metaphor
Bridge	decouple abstraction and implementation	programmatic
Composite	model recursive tree structure	metaphor
Decorator	add responsibilities to objects	metonymy
Facade	interface for a subsystem	programmatic
Flyweight	save memory of similar objects	programmatic
Proxy	surrogate for access control	metaphor
Behavioural Patterns		
Chain of Responsibility	handle requests	metonymy
Command	request as an object	metonymy
Interpreter	interpret a language	programmatic
Iterator	iteration cursor	metonymy
Mediator	encapsulate interactions	metonymy
Memento	snapshot of objects' state	metaphor
Observer	update dependents	metonymy
State	change behaviour	metonymy
Strategy	vary algorithms	metonymy
Template Method	subclasses change algorithms	programmatic
Visitor	represent traversal operations	metonymy

Classification of Patterns by Gamma et al. and James Noble et al.

Augmenting Design Patterns with Design Rationale

Feniosky Peña-Mora
and Sanjeev Vadhavkar
MIT 1997

Software Engineering

- Requirements
  - Operating Platforms
    - Heterogeneous Hardware/Software Systems
  - Topology
    - Distributed Systems
  - Evolutionary
    - Rapidly Changing Constraints

Software Engineering (cont'd)

- Code Reuse
  - Domain/Context Knowledge
  - Development Experience
  - Design Decisions
  - Design History
  - Code and Documentation
- Time and Cost Savings

Motivation for Design Patterns

- Another Tool to Speed System Development
- Design Pattern:
  - Architecture: Communicating Objects and Classes (Components/Connectors)
  - Customization: Solve a General Design Problem in a Particular Context

Motivation for Design Rationale

- Software Development
  - 70% Life Cycle Costs in Maintenance
  - Up to Half of System Maintainers' Resources Are for Reverse Engineering to Make Changes
- Only Implicit Design Decisions Kept
  - Obscure Design Notebooks
  - Minutes of Design Reviews
  - Designers' Memory

Combining Patterns and Rationale

- Synthesizing Reusable Code via Design Patterns
  - Easy to Browse, Well-Cataloged Convenient Software Components
- Creating Models that Capture and Retrieve Relevant Design Rationale
  - Facilitate Making Changes and Combining Parts of Code to Form Libraries

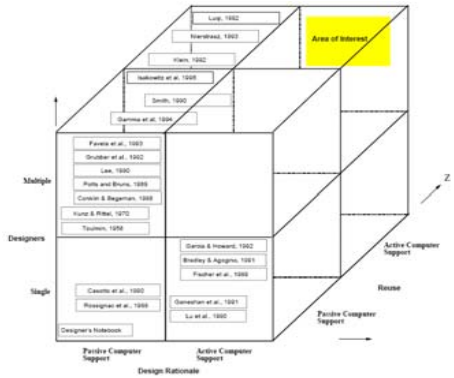
Research Goal

- Explore Role of Design Rationale in Intelligent Software Classification and Retrieval for Reuse Purpose
- To Use an Object Model Integrating Reusable Software Libraries With Explicit Schemes of Design Rationale Capture and Retrieval
- To Develop Prototype Using That Object Model as Its Base
- Test in an Industrial Setting for Use as an Integrated Design Tool for Software Developers Working in Reusable Software Engineering

Case Study

- Integrating Design Rationale and Reusability Requires Information Record
  - Why Design Decisions Are Made
  - Why Particular Solutions Were Not Undertaken
  - Why Some Solutions Are Accepted Given Certain Constraints
- Case Studies to Examine This Capture

Capturing Rationale for Reusability



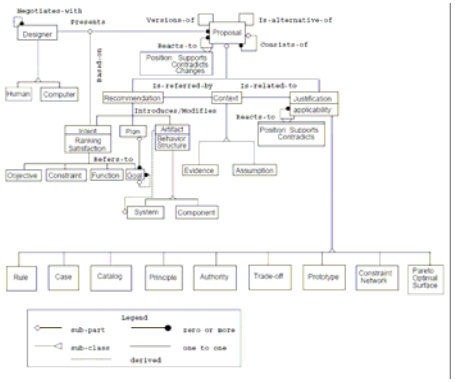
Areas of Study

- Single Designer – Passive Computer Support for Design Rationale
  - Designer Notebook, etc.
- Multiple Designers – Passive Computer Support for Design Rationale
  - IBIS, gIBIS, CAD
- Multiple Designer – Passive Computer Support for Design Rationale – Passive Computer Support for Design Reuse
  - KIDS, ORCA, AMHYRST
- Multiple Designers – Passive Design Rationale Capture – Active Computer Support for Reuse
  - CAPS
- Multiple Designers – Active Design Rationale Capture – Active Computer Support for Reuse
  - ***

Overview of Building Objects

- Design Recommendation and Intent Model
  - Allows Design Rationale from Multiple Designers to be Partially Generated, Stored, and Later Retrieved by a Computer System
- To Capture Design Rationale, It Uses:
  - Domain Knowledge
  - Design Experiences from Past
  - Interaction With Designers

DRIM

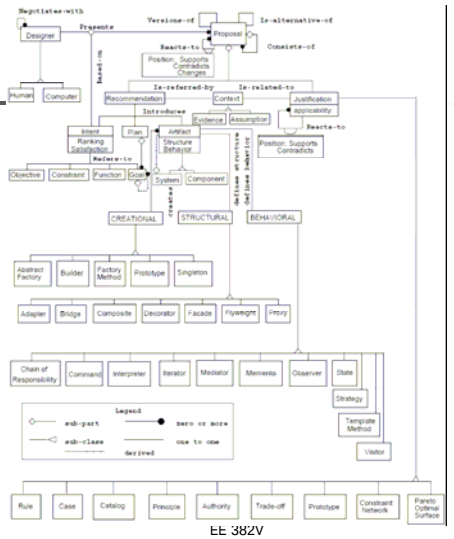


Overview of Design Patterns

- 1. Identify Good Design That Maps Solution to Implementation
- 2. Explicitly Specify How Reusable Classes Relate to the Design
- 3. Define Context in Which Design Patterns Are Valid
- 4. Explicitly Specify Key Issues, Assumptions, Constraints, and Dependencies in Prior Designs

Using DRIM for Software Reusability

- Design Recommendation and Intent Model Extended to Reusability (DRIMER)
  - Combined Approach -> “Patterns-by-Intent”
    - Software Designer
    - Proposal = Recommendation/Justification
    - Components
    - Design Patterns (Creational/Structural/Behavioral Patterns)

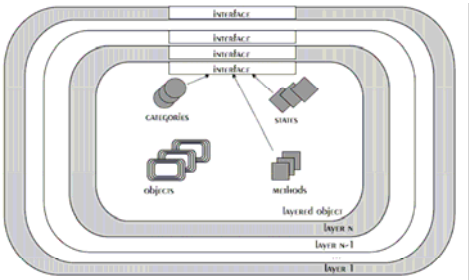


Design Patterns as Language Constructs

Jan Bosch

Sweden 1998

Layered Object Model



Types of Layers

- Structural Relation Layers
  - Define Structure of an Application
  - Extends Class Behavior
- Behavioral Relation Layers
  - Relation Types Between Objects and Its Clients
  - Constrain Access of Clients and Behavior of Object
- Application Domain Relations
  - E.g. Controls

Types of Design Patterns

- Structural Design Patterns
  - Adapter – convert interfaces
  - Bridge – decouples abstraction/implementation
  - Composite – supports part-whole hierarchies
  - Façade – single, integrated interface
- Behavioral Design Patterns
  - State – object behavior depends on internal state
  - Observer – objects depend on state changes
  - Strategy – models algorithm/behavior as object
  - Mediator – encapsulates interaction of objects

Industrial Experience with Design Patterns

Kent Beck, James Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulish, and John Vlissides

Various Large Industrial Complexes™ 1996

Industrial Experience with Design Patterns

- A design pattern is a particular prose form of recording design information such that designs which have worked well in the past can be applied again in similar situations in the future
- The availability of a collection of design patterns can help both the experienced and the novice designer recognize situations in which design reuse could or should occur
- In the industrial experience, design patterns provide
  - effective “shorthand” for communicating complex concepts effectively between designers
  - can be used to record and encourage the reuse of “best practices”
  - capture the essential parts of a design in compact form, e.g. for documentation of existing software architectures

AT&T

- AT&T has several core competencies that are fundamental to quality customer service
- High-availability system design and fault-tolerant software are among these core competencies
- Many of these core competencies can be captured as patterns since they solve a wide variety of reliability and availability problems that arise during architecture and design
- AT&T uses the fault-tolerance and high-availability patterns in architectural training
- Pattern training is largely for organizations that are "pattern consumers"
- These organizations are building new projects, using patterns as audits and drivers for design

AT&T

- Most of these courses are conducted as workshops that are highly participatory, with design exercises and pattern-writing exercises
- The training is effective on many levels
  - Attendees deepen their understanding of patterns in general and of specific core competency patterns
  - They deepen their appreciation for architecture and telecommunications foundations
- AT&T has patterns at all levels, from architectural frameworks down to design patterns and idioms
- The number of total patterns numbers in the hundreds
- Scale is a major obstacle to systematic and effective patterns usage
- AT&T is evaluating pattern organizing schemes, indexing schemes, and other attacks on the scale of the pattern knowledge base

Northern Telecom

- Northern Telecom has used the "pattern" and similar forms to capture project knowledge in a number of areas
- As part of developing a new architecture to allow rapid development and delivery of telecommunications services, Northern Telecom realized that service developers would require guidance in using the architecture and began to develop a "service design" methodology
- Many of these patterns were "prescriptive" in that they described how to get from one model to another
- As an example, a number of the patterns describe how to find and identify similar concepts in different requirements documents and capture the common concepts in the domain model of a service
- These patterns effectively are a "recipe" for doing abstraction for people to whom this does not come naturally

Northern Telecom

- Northern Telecom had discovered a number of recurring patterns in the design of telephone services
- Northern Telecom had coined terms for many of these, such as modifier service (service which observes another service and adds additional behavior at appropriate points)
- Northern Telecom quickly found themselves expressing their designs in terms of these patterns
  - They gave them a precise yet concise way of synchronizing their thoughts which saved a lot of effort
  - They no longer had to describe a key portion of the design since they had a common understanding of what was meant by "this object is using the Observer Pattern to monitor this other object"
- Northern Telecom has found patterns to be particularly useful for defining and describing software architectures
- Many patterns (Observer, Strategy, and Composite) are particularly useful when defining the architecture of a system because they encapsulate potential changes to the system

Comment from Northern Telecom

- "Using patterns written by others only takes an open mind. But, writing patterns takes a special mind! Most people whom we have exposed to the concept of patterns can quickly become proficient at using the common ones. But we have found that only a small percentage of people can write patterns. With respect to patterns, there are three kinds of people. Those who see patterns everywhere and can describe them, those who can recognize patterns but can not describe them easily, and those who are oblivious to the pattern surrounding them. This difference seems to stem from a basic orientation of people to focus on similarities as opposed to differences between things."
- "We have not attempted to measure the impact of patterns on productivity but we have noticed that communication between people with a "shared space" of patterns is quicker, more complete, and less likely to be misunderstood. At the programming level, we have had people design what might be rather complex designs much more quickly than expected by using one or more design patterns."

Conclusion

- Analysis and Observations
  - Metonymy
    - Justification for Reality ⇔ Objects Vague?
    - Abstract – Metaphor/Metonymy in Design Patterns/OOP => accurate, flexible, better understood designs?
  - Augmenting with Design Rationale
    - Emphasis on Reuse => Software Flexibility?
  - Language Constructs
    - Case Studies

Conclusion

- Engineers' Desire to Implement Reuse
- Software Engineering as Both Theoretical and Empirical
- Architecting vs. Designing
- Foundation and Models
- Types and Examples
- Industrial Applications

Further Reading

- "Software Design Patterns: Common Questions and Answers"
  - James Coplien
  - AT&T '94
- "Patterns Generate Architectures"
  - Kent Beck and Ralph Johnson
  - ECOOP '94
- "Design Patterns for Object-Oriented Software Development"
  - Wolfgang Pree and Hermann Sikora
  - ICSE '97

Questions?