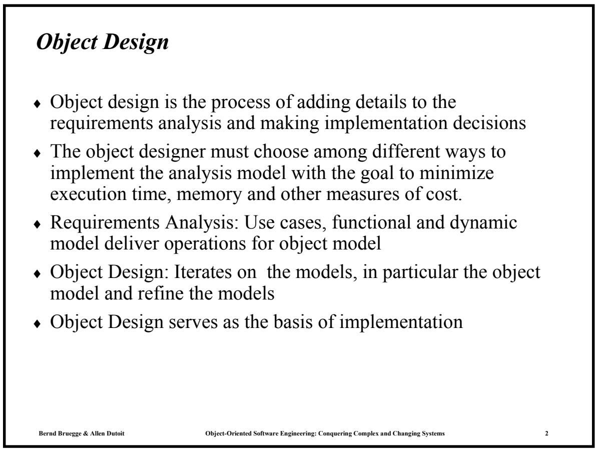
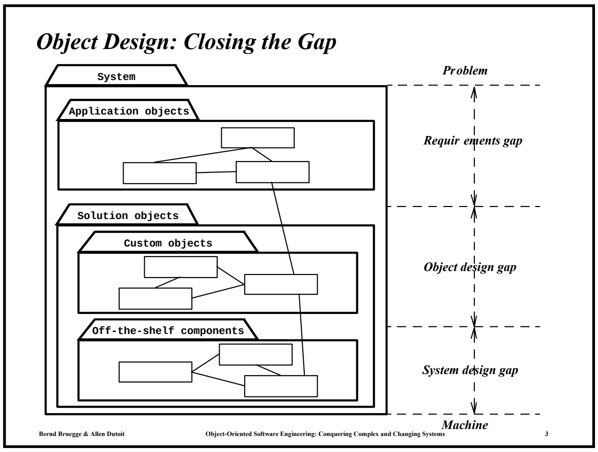
Object-Oriented Software Engineering
Using UML, Patterns, and Java

Chapter 8, Object Design: Reuse and Patterns I



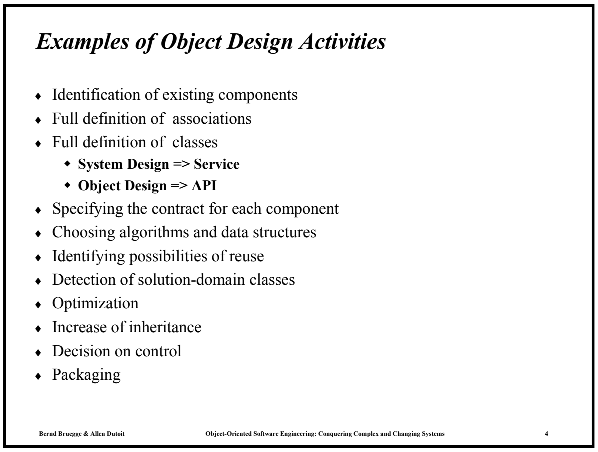
Object Design

- Object design is the process of adding details to the requirements analysis and making implementation decisions
- The object designer must choose among different ways to implement the analysis model with the goal to minimize execution time, memory and other measures of cost.
- Requirements Analysis: Use cases, functional and dynamic model deliver operations for object model
- Object Design: Iterates on the models, in particular the object model and refine the models
- Object Design serves as the basis of implementation



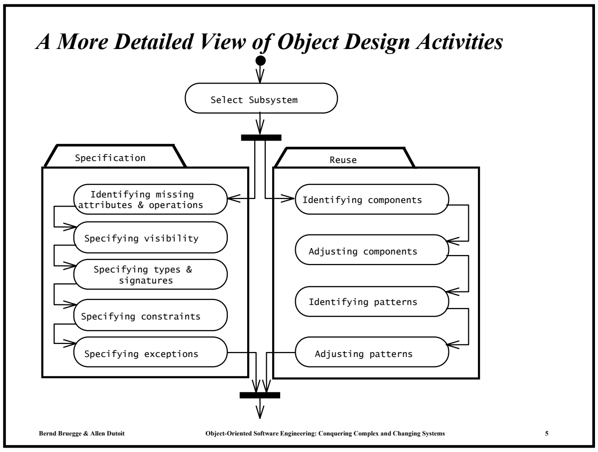
Object Design: Closing the Gap

System
Application objects
Solution objects
Custom objects
Off-the-shelf components
Problem
Requir ements gap
Object design gap
System design gap
Machine



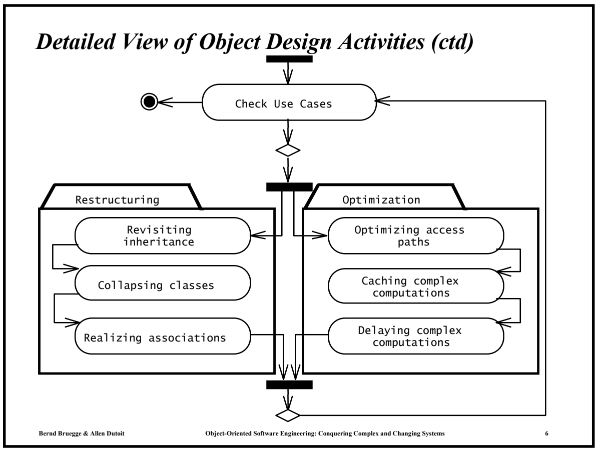
Examples of Object Design Activities

- Identification of existing components
- Full definition of associations
- Full definition of classes
  - **System Design => Service**
  - **Object Design => API**
- Specifying the contract for each component
- Choosing algorithms and data structures
- Identifying possibilities of reuse
- Detection of solution-domain classes
- Optimization
- Increase of inheritance
- Decision on control
- Packaging



A More Detailed View of Object Design Activities

Select Subsystem
Specification
Identifying missing attributes & operations
Specifying visibility
Specifying types & signatures
Specifying constraints
Specifying exceptions
Reuse
Identifying components
Adjusting components
Identifying patterns
Adjusting patterns



Detailed View of Object Design Activities (ctd)

Check Use Cases
Restructuring
Revisiting inheritance
Collapsing classes
Realizing associations
Optimization
Optimizing access paths
Caching complex computations
Delaying complex computations

A Little Bit of Terminology: Activities

- Object-Oriented methodologies use these terms:
  - **System Design Activity**
    - Decomposition into subsystems
  - **Object Design Activity**
    - Implementation language chosen
    - Data structures and algorithms chosen
- Structured analysis/structured design uses these terms:
  - **Preliminary Design Activity**
    - Decomposition into subsystems
    - Data structures are chosen
  - **Detailed Design Activity**
    - Algorithms are chosen
    - Data structures are refined
    - Implementation language is chosen
    - Typically in parallel with preliminary design, not a separate activity

Outline of the Lecture

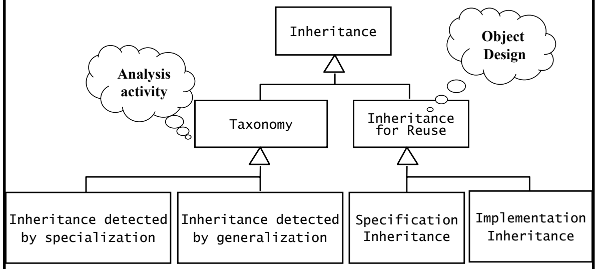
- Design Patterns
  - **Usefulness of design patterns**
  - **Design Pattern Categories**
- Patterns covered in this lecture
  - **Composite: Model dynamic aggregates**
  - **Facade: Interfacing to subsystems**
  - **Adapter: Interfacing to existing systems (legacy systems)**
  - **Bridge: Interfacing to existing and future systems**
- More patterns:
  - **Abstract Factory: Provide manufacturer independence**
  - **Builder: Hide a complex creation process**
  - **Proxy: Provide Location transparency**
  - **Command: Encapsulate control flow**
  - **Observer: Provide publisher/subscribe mechanism**
  - **Strategy: Support family of algorithms, separate of policy and mechanism**

The use of inheritance

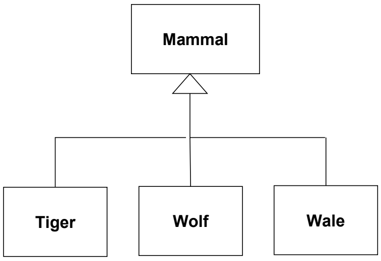
- Inheritance is used to achieve two different goals
  - **Description of Taxonomies**
  - **Interface Specification**
- Identification of taxonomies
  - **Used during requirements analysis.**
  - **Activity: identify application domain objects that are hierarchically related**
  - **Goal: make the analysis model more understandable**
- Service specification
  - **Used during object design**
  - **Activity:**
  - **Goal: increase reusability, enhance modifiability and extensibility**
- Inheritance is found either by specialization or generalization

Metamodel for Inheritance

- Inheritance is used during analysis and object design

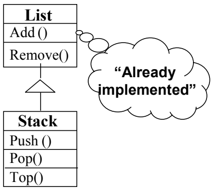

Inheritance
Analysis activity
Object Design
Taxonomy
Inheritance for Reuse
Inheritance detected by specialization
Inheritance detected by generalization
Specification Inheritance
Implementation Inheritance

Taxonomy Example



Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation.
  - Example: I have a **List** class, I need a **Stack** class. How about subclassing the **Stack** class from the **List** class and providing three methods, **Push()** and **Pop(), Top()**?



- Problem with implementation inheritance:

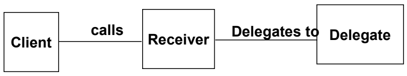
  Some of the inherited operations might exhibit unwanted behavior. What happens if the Stack user calls Remove() instead of Pop()?

Implementation Inheritance vs Interface Inheritance

- Implementation inheritance
  - **Also called class inheritance**
  - **Goal: Extend an applications' functionality by reusing functionality in parent class**
  - **Inherit from an existing class with some or all operations already implemented**
- Interface inheritance
  - **Also called subtyping**
  - **Inherit from an abstract class with all operations specified, but not yet implemented**

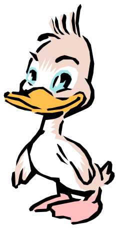
Delegation as alternative to Implementation Inheritance

- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation two objects are involved in handling a request
  - **A receiving object delegates operations to its delegate.**
  - **The developer can make sure that the receiving object does not allow the client to misuse the delegate object**



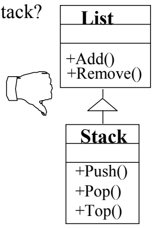
Duck: Delegation vs. Inheritance

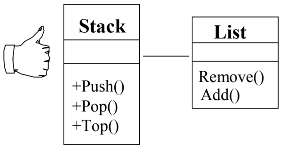
- Description: Decide whether to use delegation or inheritance for designing the following classes. Specify the attributes and methods for each class. Draw the UML diagram for the whole thing.
  - Array
  - Queue
  - Stack
  - Tree
  - Linked list
- Process:
  - Work in pairs
  - You have about 10 minutes.



Delegation instead of Implementation Inheritance

- **Inheritance**: Extending a Base class by a new operation or overwriting an operation.
- **Delegation**: Catching an operation and sending it to another object.
- Which of the following models is better for implementing a stack?



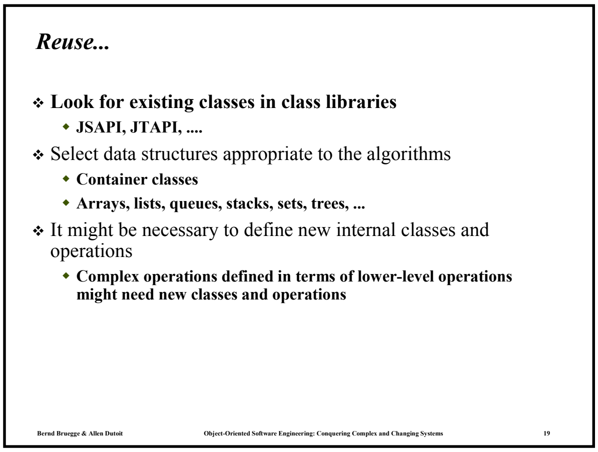


Comparison: Delegation vs Implementation Inheritance

- Delegation
  - **Pro:**
    - **Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)**
  - **Con:**
    - **Inefficiency: Objects are encapsulated.**
- Inheritance
  - **Pro:**
    - **Straightforward to use**
    - **Supported by many programming languages**
    - **Easy to implement new functionality**
  - **Con:**
    - **Inheritance exposes a subclass to the details of its parent class**
    - **Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)**

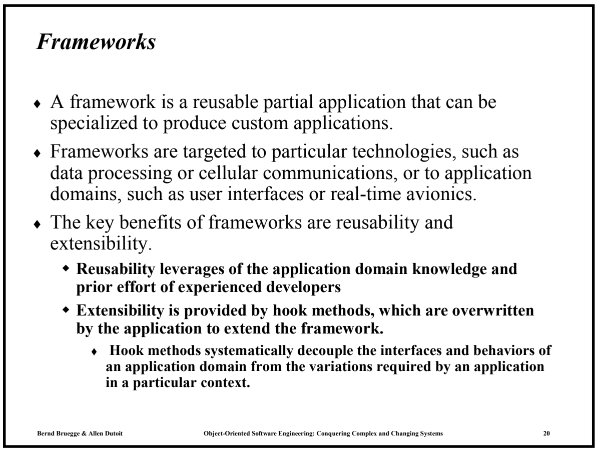
Component Selection

- Select existing
  - **off-the-shelf class libraries**
  - **frameworks or**
  - **components**
- Adjust the class libraries, framework or components
  - **Change the API if you have the source code.**
  - **Use the adapter or bridge pattern if you don't have access**
- Architecture Driven Design



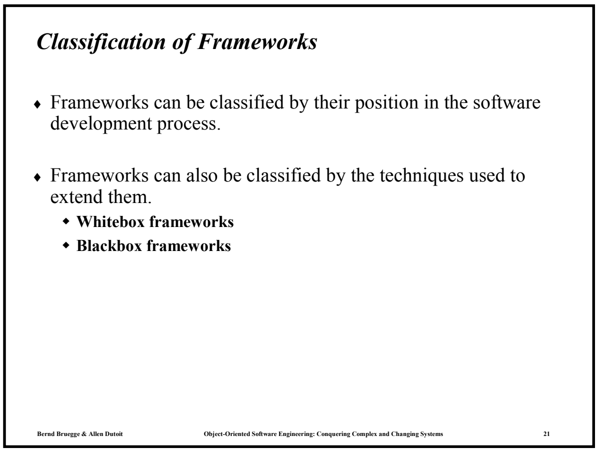
Reuse...

- **Look for existing classes in class libraries**
  - **JSAPI, JTAPI, ...**
- Select data structures appropriate to the algorithms
  - **Container classes**
  - **Arrays, lists, queues, stacks, sets, trees, ...**
- It might be necessary to define new internal classes and operations
  - **Complex operations defined in terms of lower-level operations might need new classes and operations**



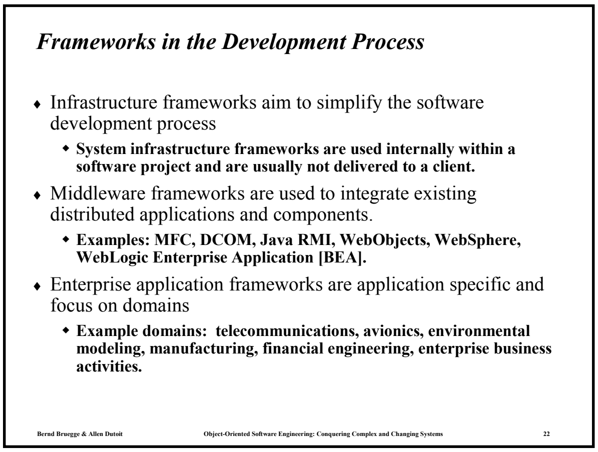
Frameworks

- A framework is a reusable partial application that can be specialized to produce custom applications.
- Frameworks are targeted to particular technologies, such as data processing or cellular communications, or to application domains, such as user interfaces or real-time avionics.
- The key benefits of frameworks are reusability and extensibility.
  - **Reusability leverages of the application domain knowledge and prior effort of experienced developers**
  - **Extensibility is provided by hook methods, which are overwritten by the application to extend the framework.**
    - **Hook methods systematically decouple the interfaces and behaviors of an application domain from the variations required by an application in a particular context.**



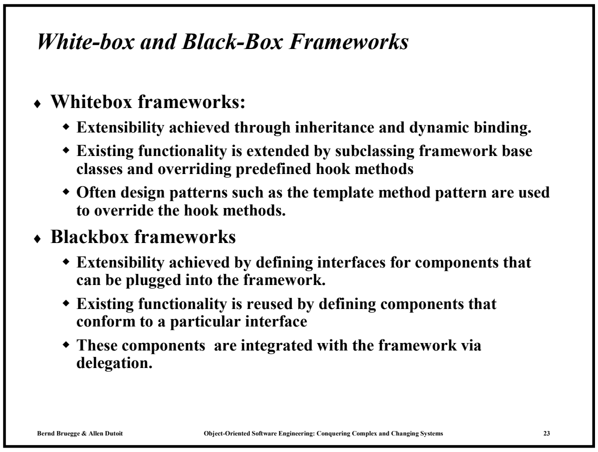
Classification of Frameworks

- Frameworks can be classified by their position in the software development process.
- Frameworks can also be classified by the techniques used to extend them.
  - **Whitebox frameworks**
  - **Blackbox frameworks**



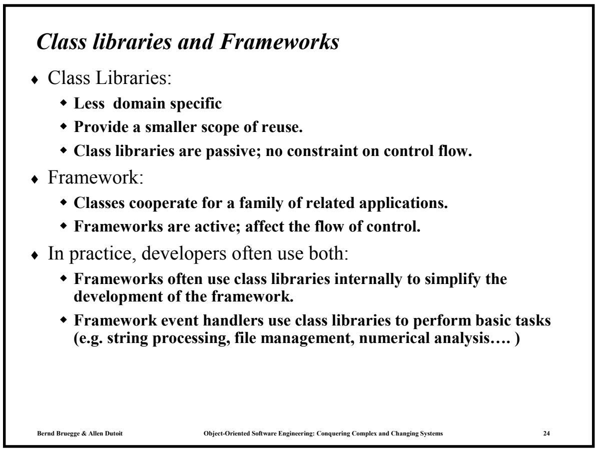
Frameworks in the Development Process

- Infrastructure frameworks aim to simplify the software development process
  - **System infrastructure frameworks are used internally within a software project and are usually not delivered to a client.**
- Middleware frameworks are used to integrate existing distributed applications and components.
  - **Examples: MFC, DCOM, Java RMI, WebObjects, WebSphere, WebLogic Enterprise Application [BEA].**
- Enterprise application frameworks are application specific and focus on domains
  - **Example domains: telecommunications, avionics, environmental modeling, manufacturing, financial engineering, enterprise business activities.**



White-box and Black-Box Frameworks

- **Whitebox frameworks:**
  - **Extensibility achieved through inheritance and dynamic binding.**
  - **Existing functionality is extended by subclassing framework base classes and overriding predefined hook methods**
  - **Often design patterns such as the template method pattern are used to override the hook methods.**
- **Blackbox frameworks**
  - **Extensibility achieved by defining interfaces for components that can be plugged into the framework.**
  - **Existing functionality is reused by defining components that conform to a particular interface**
  - **These components are integrated with the framework via delegation.**



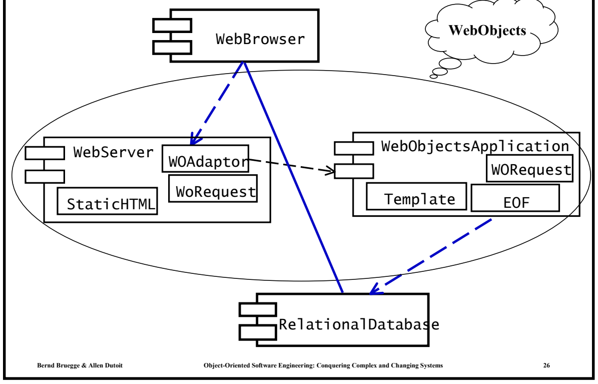
Class libraries and Frameworks

- Class Libraries:
  - **Less domain specific**
  - **Provide a smaller scope of reuse.**
  - **Class libraries are passive; no constraint on control flow.**
- Framework:
  - **Classes cooperate for a family of related applications.**
  - **Frameworks are active; affect the flow of control.**
- In practice, developers often use both:
  - **Frameworks often use class libraries internally to simplify the development of the framework.**
  - **Framework event handlers use class libraries to perform basic tasks (e.g. string processing, file management, numerical analysis….)**

Components and Frameworks

- Components
  - Self-contained instances of classes
  - Plugged together to form complete applications.
  - Blackbox that defines a cohesive set of operations,
  - Can be used based on the syntax and semantics of the interface.
  - Components can even be reused on the binary code level.
    - The advantage is that applications do not always have to be recompiled when components change.
- Frameworks:
  - Often used to develop components
  - Components are often plugged into blackbox frameworks.

Example: Framework for Building Web Applications



Finding Objects

- The hardest problems in object-oriented system development are:
  - Identifying objects
  - Decomposing the system into objects
- Requirements Analysis focuses on application domain:
  - Object identification
- System Design addresses both, application and implementation domain:
  - Subsystem Identification
- Object Design focuses on implementation domain:
  - Additional solution objects

Techniques for Finding Objects

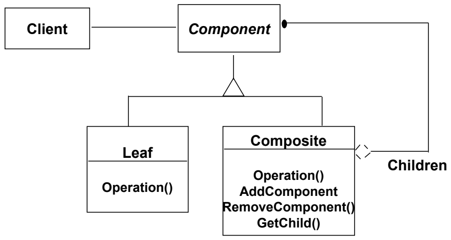
- Requirements Analysis
  - Start with Use Cases. Identify participating objects
  - Textual analysis of flow of events (find nouns, verbs, ...)
  - Extract application domain objects by interviewing client (application domain knowledge)
  - Find objects by using general knowledge
- System Design
  - Subsystem decomposition
  - Try to identify layers and partitions
- Object Design
  - Find additional objects by applying implementation domain knowledge

Another Source for Finding Objects : Design Patterns

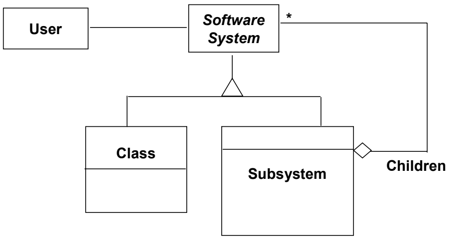
- What are Design Patterns?
  - A design pattern describes a problem which occurs over and over again in our environment
  - Then it describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same twice

Introducing the Composite Pattern

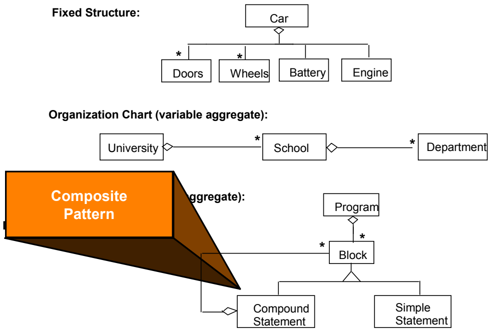
- Models tree structures that represent part-whole hierarchies with arbitrary depth and width.
- The Composite Pattern lets client treat individual objects and compositions of these objects uniformly



Modeling a Software System with a Composite Pattern

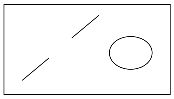


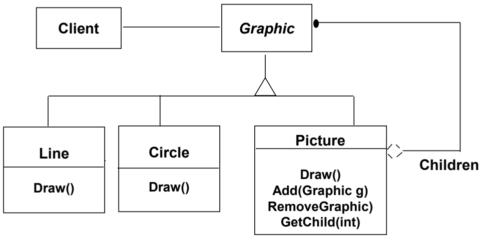
The Composite Patterns models dynamic aggregates



Graphic Applications also use Composite Patterns

- The *Graphic* Class represents both primitives (Line, Circle) and their containers (Picture)



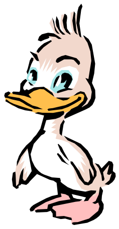


Design Patterns reduce the Complexity of Models

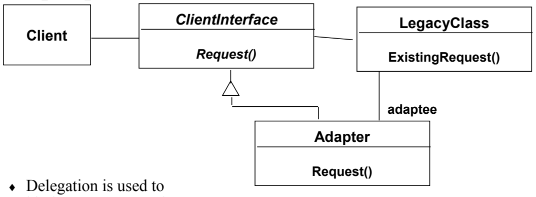
- To communicate a complex model we use navigation and reduction of complexity
  - **We do not simply use a picture from the CASE tool and dump it in front of the user**
  - **The key is navigate through the model so the user can follow it.**
- We start with a very simple model and then decorate it incrementally
  - **Start with key abstractions (use animation)**
  - **Then decorate the model with the additional classes**
- To reduce the complexity of the model even further, we
  - **Apply the use of inheritance (for taxonomies, and for design patterns)**
    - **If the model is still too complex, we show the subclasses on a separate slide**
  - **Then identify (or introduced) patterns in the model**
    - **We make sure to use the name of the patterns**

Duck: Studying your object design

- Description:
  - **Review your current object design.**
  - **Identify any objects that are missing.**
  - **Does the composite pattern fit any part of your design?**
  - **Review all the attributes and methods, including their types and visibility, of your objects. Fill in the missing attributes and methods.**
- Process:
  - Work in teams
  - You have about 10 minutes.



Adapter pattern



- Delegation is used to bind an **Adapter** and an **Adaptee**
- Interface inheritance is use to specify the interface of the **Adapter** class.
- ***Target*** and **Adaptee** (usually called legacy system) pre-exist the **Adapter.**
- **Target** may be realized as an interface in Java.

Adapter Pattern

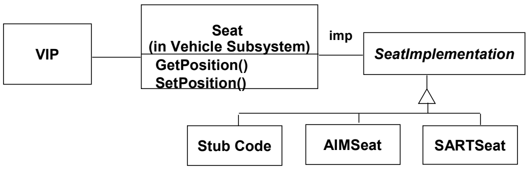
- "Convert the interface of a class into another interface clients expect."
- The adapter pattern lets classes work together that couldn't otherwise because of incompatible interfaces
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Also known as a wrapper
- Two adapter patterns:
  - **Class adapter:**
    - **Uses multiple inheritance to adapt one interface to another**
  - **Object adapter:**
    - **Uses single inheritance and delegation**
- Object adapters are much more frequent. We will only cover object adapters (and call them therefore simply adapters)

Bridge Pattern

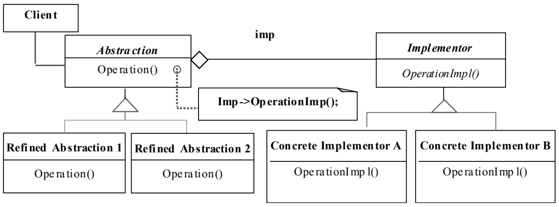
- Use a bridge to "decouple an abstraction from its implementation so that the two can vary independently". (From [Gamma et al 1995])
- Also know as a Handle/Body pattern.
- Allows different implementations of an interface to be decided upon dynamically.

Using a Bridge

- The bridge pattern is used to provide multiple implementations under the same interface.
- Examples: Interface to a component that is incomplete, not yet known or unavailable during testing
- JAMES Project: if seat data is required to be read, but the seat is not yet implemented, known, or only available by a simulation, provide a bridge:



Bridge Pattern

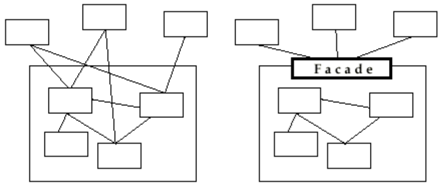


Adapter vs Bridge

- Similarities:
  - **Both are used to hide the details of the underlying implementation.**
- Difference:
  - **The adapter pattern is geared towards making unrelated components work together**
    - **Applied to systems after they're designed (reengineering, interface engineering).**
  - **A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.**
    - **Green field engineering of an "extensible system"**
    - **New "beasts" can be added to the "object zoo", even if these are not known at analysis or system design time.**

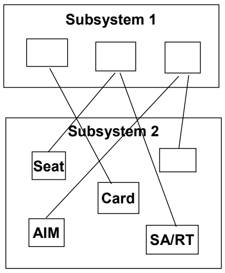
Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Facades allow us to provide a closed architecture



Design Example

- Subsystem 1 can look into the Subsystem 2 (vehicle subsystem) and call on any component or class operation at will.
- This is “Ravioli Design”
- Why is this good?
  - **Efficiency**
- Why is this bad?
  - **Can’t expect the caller to understand how the subsystem works or the complex relationships within the subsystem.**
  - **We can be assured that the subsystem will be misused, leading to non-portable code**

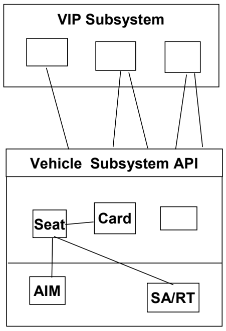


Subsystem Design with Façade, Adapter, Bridge

- The ideal structure of a subsystem consists of
  - **an interface object**
  - **a set of application domain objects (entity objects) modeling real entities or existing systems**
    - **Some of the application domain objects are interfaces to existing systems**
  - **one or more control objects**
- We can use design patterns to realize this subsystem structure
- Realization of the Interface Object: Facade
  - **Provides the interface to the subsystem**
- Interface to existing systems: Adapter or Bridge
  - **Provides the interface to existing system (legacy system)**
  - **The existing system is not necessarily object-oriented!**

Realizing an Opaque Architecture with a Facade

- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers
- If a façade is used the subsystem can be used in an early integration test
  - **We need to write only a driver**



Design Patterns encourage reusable Designs

- A facade pattern should be used by all subsystems in a software system. The façade defines all the services of the subsystem.
  - **The facade will delegate requests to the appropriate components within the subsystem. Most of the time the façade does not need to be changed, when the component is changed,**
- Adapters should be used to interface to existing components.
  - **For example, a smart card software system should provide an adapter for a particular smart card reader and other hardware that it controls and queries.**
- Bridges should be used to interface to a set of objects
  - **where the full set is not completely known at analysis or design time.**
  - **when the subsystem must be extended later after the system has been deployed and client programs are in the field(dynamic extension).**
- Model/View/Controller should be used
  - **when the interface changes much more rapidly than the application domain.**

Review: Design pattern

A design pattern is…

…a template solution to a recurring design problem

- **Look before re-inventing the wheel just one more time**

…reusable design knowledge

- **Higher level than classes or datastructures (link lists,binary trees...)**
- **Lower level than application frameworks**

…an example of *modifiable* design

- **Learning to design starts by studying other designs**

Why are modifiable designs important?

A modifiable design enables…

…an iterative and incremental development cycle

- **concurrent development**
- **risk management**
- **flexibility to change**

…to minimize the introduction of new problems when fixing old ones

…to deliver more functionality after initial delivery

What makes a design modifiable?

- Low coupling and high cohesion
- Clear dependencies
- Explicit assumptions

How do design patterns help?

- They are generalized from existing systems
- They provide a shared vocabulary to designers
- They provide examples of modifiable designs
  - **Abstract classes**
  - **Delegation**

On to More Patterns!

- Structural pattern
  - **Proxy**
- Creational Patterns
  - **Abstract Factory**
  - **Builder**
- Behavioral pattern
  - **Command**
  - **Observer**
  - **Strategy**

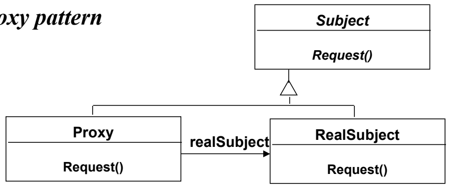
Proxy Pattern: Motivation

- It is 15:00pm. I am sitting at my 14.4 baud modem connection and retrieve a fancy web site from the US, This is prime web time all over the US. So I am getting 10 bits/sec.
- What can I do?

Proxy Pattern

- What is expensive?
  - **Object Creation**
  - **Object Initialization**
- Defer object creation and object initialization to the time you need the object
- Proxy pattern:
  - **Reduces the cost of accessing objects**
  - **Uses another object ("the proxy") that acts as a stand-in for the real object**
  - **The proxy creates the real object only if the user asks for it**

Proxy pattern



- Interface inheritance is used to specify the interface shared by **Proxy** and **RealSubject.**
- Delegation is used to catch and forward any accesses to the **RealSubject** (if desired)
- Proxy patterns can be used for lazy evaluation and for remote invocation.
- Proxy patterns can be implemented with a Java interface.

Proxy Applicability

- Remote Proxy
  - **Local representative for an object in a different address space**
  - **Caching of information: Good if information does not change too often**
- Virtual Proxy
  - **Object is too expensive to create or too expensive to download**
  - **Proxy is a stand-in**
- Protection Proxy
  - **Proxy provides access control to the real object**
  - **Useful when different objects should have different access and viewing rights for the same document.**
  - **Example: Grade information for a student shared by administrators, teachers and students.**

Virtual Proxy example

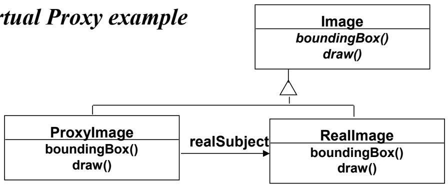
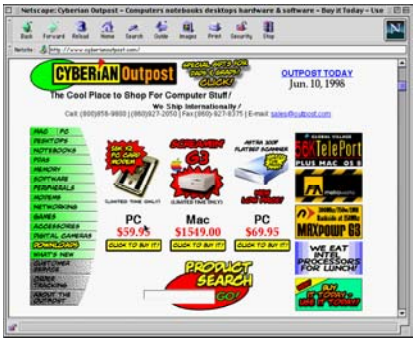


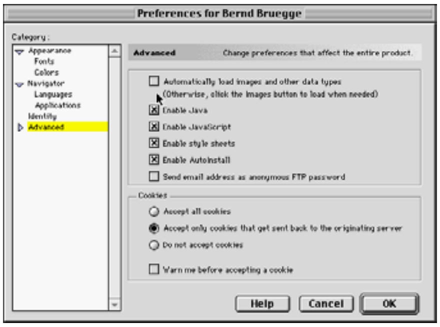
Image
boundingBox()
draw()
ProxyImage
boundingBox()
draw()
realSubject
RealImage
boundingBox()
draw()

- **Images** are stored and loaded separately from text
- If a **RealImage** is not loaded a **ProxyImage** displays a grey rectangle in place of the image
- The client cannot tell that it is dealing with a **ProxyImage** instead of a **RealImage**
- A proxy pattern can be easily combined with a **Bridge**

Before



Controlling Access


Preferences for Bernd Bruegge

Category:
- Appearance
  - Fonts
  - Colors
- Navigator
  - Languages
  - Applications
- Identity
- Advanced

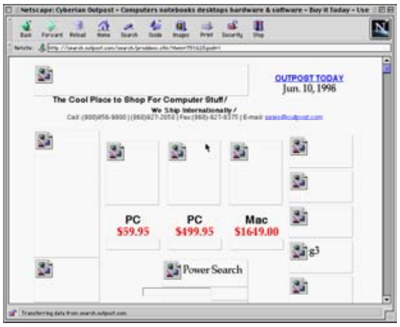
Advanced Change preferences that affect the entire product.

□ Automatically load images and other data types
(Otherwise, click the Images button to load when needed)
☒ Enable Java
☒ Enable JavaScript
☒ Enable style sheets
☒ Enable AutoInstall
□ Send email address as anonymous FTP password

Cookies
○ Accept all cookies
● Accept only cookies that get sent back to the originating server
○ Do not accept cookies

□ Warn me before accepting a cookie

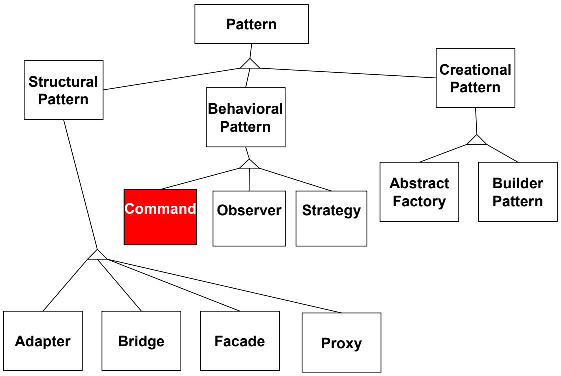
After


Netscape: Cyberian Outpost - Computers notebooks desktops hardware & software - Buy it Today - Use
Back
Forward
Reload
Home
Search
Guide
Images
Print
Security
Stop
OUTPOST TODAY
Jun. 10, 1998
The Cool Place to Shop For Computer Stuff!
We Ship Internationally!
PC
$59.95
PC
$499.95
Mac
$1649.00
g3
Power Search
Transferring data from search.outpost.com

Towards a Pattern Taxonomy

- Structural Patterns
  - **Adapters, Bridges, Facades, and Proxies are variations on a single theme:**
    - **They reduce the coupling between two or more classes**
    - **They introduce an abstract class to enable future extensions**
    - **They encapsulate complex structures**
- Behavioral Patterns
  - **Here we are concerned with algorithms and the assignment of responsibilities between objects: Who does what?**
  - **Behavioral patterns allow us to characterize complex control flows that are difficult to follow at runtime.**
- Creational Patterns
  - **Here our goal is to provide a simple abstraction for a complex instantiation process.**
  - **We want to make the system independent from the way its objects are created, composed and represented.**

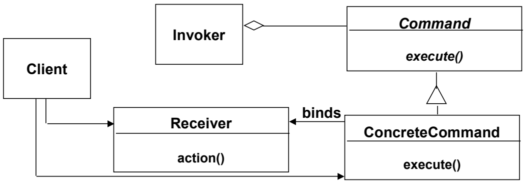
A Pattern Taxonomy


Pattern
Structural Pattern
Behavioral Pattern
Creational Pattern
Command
Observer
Strategy
Abstract Factory
Builder Pattern
Adapter
Bridge
Facade
Proxy

Command Pattern: Motivation

- You want to build a user interface
- You want to provide menus
- You want to make the user interface reusable across many applications
  - **You cannot hardcode the meanings of the menus for the various applications**
  - **The applications only know what has to be done when a menu is selected.**
- Such a menu can easily be implemented with the Command Pattern

Command pattern



- **Client** creates a **ConcreteCommand** and binds it with a **Receiver.**
- **Client** hands the **ConcreteCommand** over to the **Invoker** which stores it.
- The **Invoker** has the responsibility to do the command ("execute" or "undo")**.**

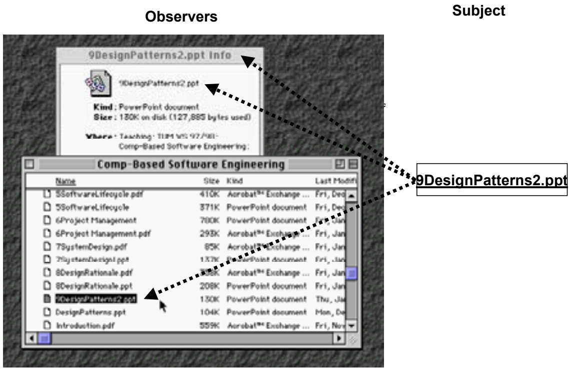
Command pattern Applicability

- "Encapsulate a request as an object, thereby letting you
  - **parameterize clients with different requests,**
  - **queue or log requests, and**
  - **support undoable operations."**
- Uses:
  - **Undo queues**
  - **Database transaction buffering**

Observer pattern

- "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."
- Also called "Publish and Subscribe"
- Uses:
  - **Maintaining consistency across redundant state**
  - **Optimizing batch changes to maintain consistency**

Observer pattern (continued)



9DesignPatterns2.ppt Info

9DesignPatterns2.ppt

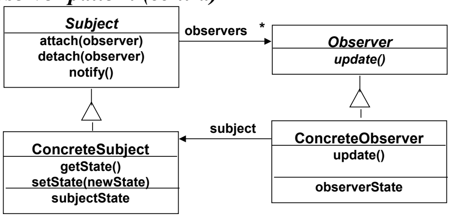
Kind : PowerPoint document
Size : 130K on disk (127,885 bytes used)

Where : Teaching : TUM WS 97/98 :
Comp-Based Software Engineering :

Comp-Based Software Engineering

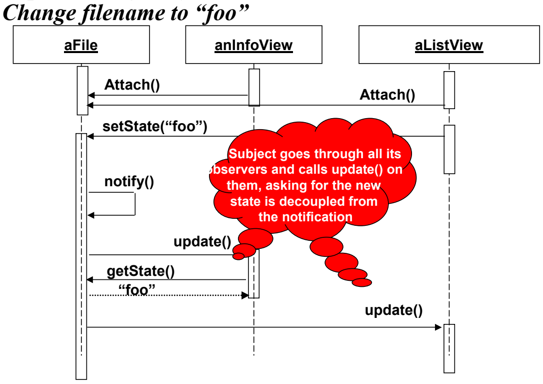
Name	Size	Kind	Last Modifi
5SoftwareLifecycle.pdf	410K	Acrobat™ Exchange ...	Fri, Dec
5SoftwareLifecycle	371K	PowerPoint document	Fri, Dec
6Project Management	700K	PowerPoint document	Fri, Jan
6Project Management.pdf	293K	Acrobat™ Exchange ...	Fri, Jan
7SystemDesign.pdf	85K	Acrobat™ Exchange ...	Fri, Jan
7SystemDesign1.ppt	137K	PowerPoint document	Fri, Jan
8DesignRationale.pdf	358K	Acrobat™ Exchange ...	Fri, Jan
8DesignRationale.ppt	208K	PowerPoint document	Fri, Jan
9DesignPatterns2.ppt	130K	PowerPoint document	Thu, Jan
DesignPatterns.ppt	104K	PowerPoint document	Mon, De
Introduction.pdf	559K	Acrobat™ Exchange ...	Fri, Nov

Observer pattern (cont'd)



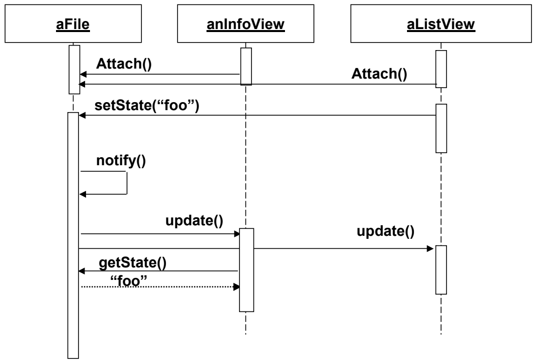
- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can be implemented as a Java interface.
- **Subject** is a super class (needs to store the observers vector) ***not*** an interface.

Sequence diagram for scenario: Change filename to “foo”

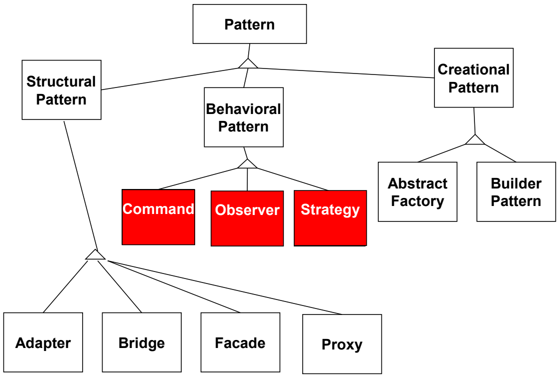


aFile
anInfoView
aListView
Attach()
Attach()
setState(“foo”)
notify()
update()
getState()
“foo”
update()
Subject goes through all its bservers and calls update() on them, asking for the new state is decoupled from the notification

Animated Sequence diagram



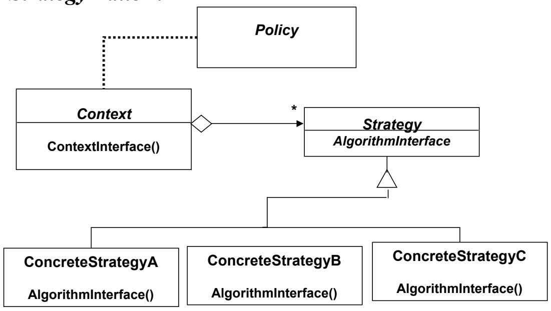
A Pattern Taxonomy



Strategy Pattern

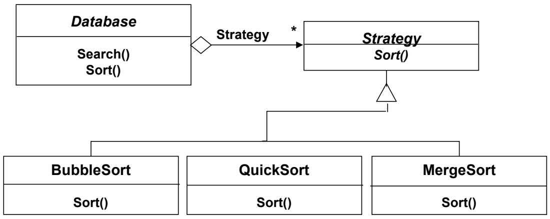
- Many different algorithms exists for the same task
- Examples:
  - **Breaking a stream of text into lines**
  - **Parsing a set of tokens into an abstract syntax tree**
  - **Sorting a list of customers**
- The different algorithms will be appropriate at different times
  - **Rapid prototyping vs delivery of final product**
- We don’t want to support all the algorithms if we don’t need them
- If we need a new algorithm, we want to add it easily without disturbing the application using the algorithm

Strategy Pattern



Policy decides which Strategy is best given the current Context

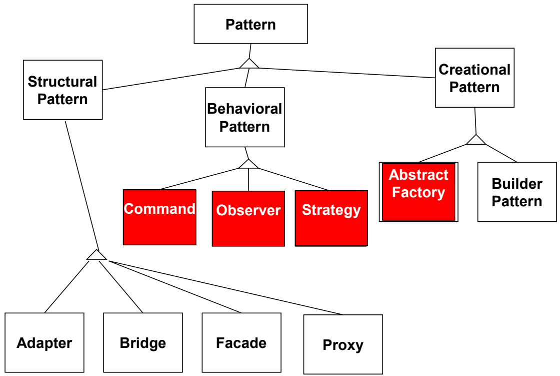
Applying a Strategy Pattern in a Database Application



Applicability of Strategy Pattern

- Many related classes differ only in their behavior. Strategy allows to configure a single class with one of many behaviors
- Different variants of an algorithm are needed that trade-off space against time. All these variants can be implemented as a class hierarchy of algorithms

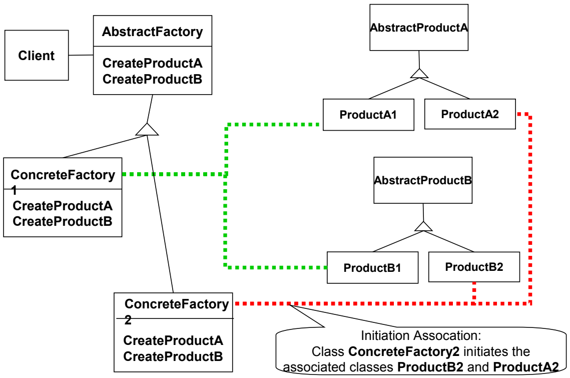
A Pattern Taxonomy



Abstract Factory Motivation

- 2 Examples
- Consider a user interface toolkit that supports multiple looks and feel standards such as Motif, Windows 95 or the finder in MacOS.
  - **How can you write a single user interface and make it portable across the different look and feel standards for these window managers?**
- Consider a facility management system for an intelligent house that supports different control systems such as Siemens' Instabus, Johnson & Control Metasys or Zumtobe's proprietary standard.
  - **How can you write a single control system that is independent from the manufacturer?**

Abstract Factory

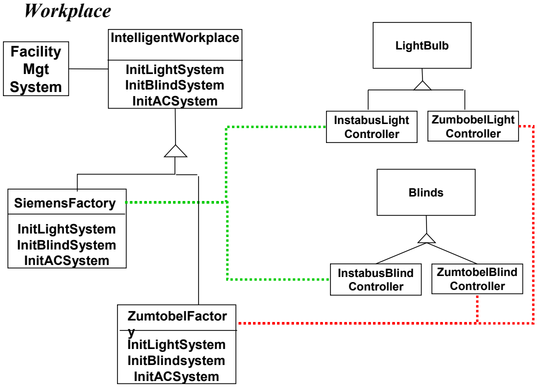


Initiation Assocation:
Class **ConcreteFactory2** initiates the associated classes **ProductB2** and **ProductA2**

Applicability for Abstract Factory Pattern

- Independence from Initialization or Representation:
  - **The system should be independent of how its products are created, composed or represented**
- Manufacturer Independence:
  - **A system should be configured with one family of products, where one has a choice from many different families.**
  - **You want to provide a class library for a customer ("facility management library"), but you don't want to reveal what particular product you are using.**
- Constraints on related products
  - **A family of related products is designed to be used together and you need to enforce this constraint**
- Cope with upcoming change:
  - **You use one particular product family, but you expect that the underlying technology is changing very soon, and new products will appear on the market.**

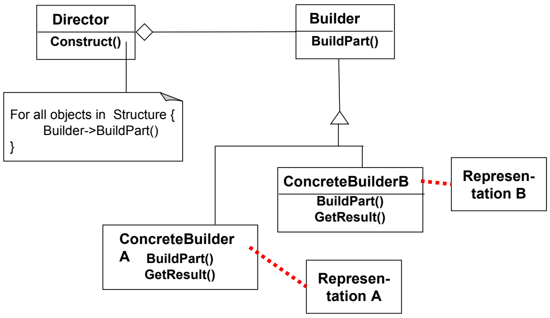
Example: A Facility Management System for the Intelligent Workplace



Builder Pattern Motivation

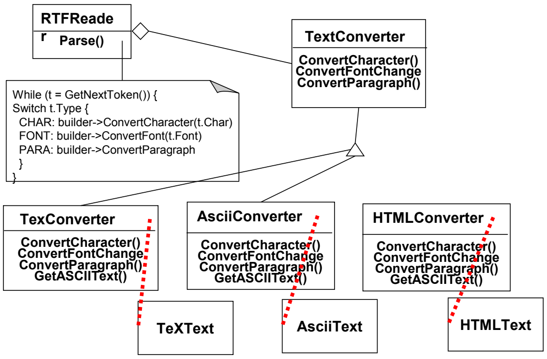
- Conversion of documents
- Software companies make their money by introducing new formats, forcing users to upgrades
  - **But you don't want to upgrade your software every time there is an update of the format for Word documents**
- Idea: A reader for RTF format
  - **Convert RTF to many text formats (EMACS, Framemaker 4.0, Framemaker 5.0, Framemaker 5.5, HTML, SGML, WordPerfect 3.5, WordPerfect 7.0, ….)**
    - ***Problem: The number of conversions is open-ended.***
- Solution
  - **Configure the RTF Reader with a "builder" object that specializes in conversions to any known format and can easily be extended to deal with any new format appearing on the market**

Builder Pattern



```
For all objects in  Structure {
     Builder->BuildPart()
}
```

Example



```
While (t = GetNextToken()) {
Switch t.Type {
  CHAR: builder->ConvertCharacter(t.Char)
  FONT: builder->ConvertFont(t.Font)
  PARA: builder->ConvertParagraph
  }
}
```

When do you use the Builder Pattern?

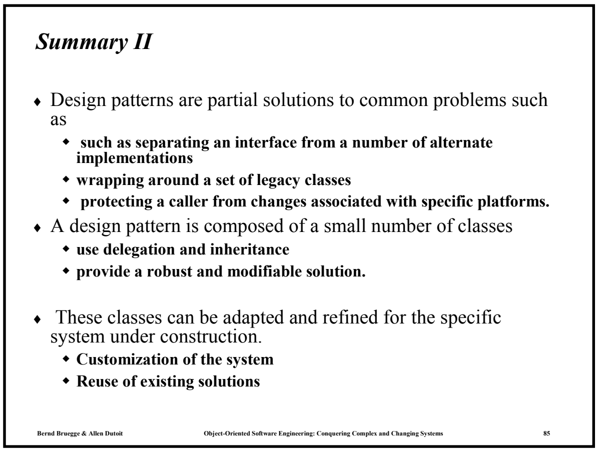
- The creation of a complex product must be independent of the particular parts that make up the product
  - **In particular, the creation process should not know about the assembly process (how the parts are put together to make up the product)**
- The creation process must allow different representations for the object that is constructed. Examples:
  - **A house with one floor, 3 rooms, 2 hallways, 1 garage and three doors.**
  - **A skyscraper with 50 floors, 15 offices and 5 hallways on each floor. The office layout varies for each floor.**

Comparison: Abstract Factory vs Builder

- Abstract Factory
  - **Focuses on product family**
    - **The products can be simple ("light bulb") or complex ("engine")**
  - **Does not hide the creation process**
    - **The product is immediately returned**
- Builder
  - **The underlying product needs to be constructed as part of the system, but the creation is very complex**
  - **The construction of the complex product changes from time to time**
  - **The builder patterns hides the creation process from the user:**
    - **The product is returned after creation as a final step**
- Abstract Factory and Builder work well together for a family of multiple complex products

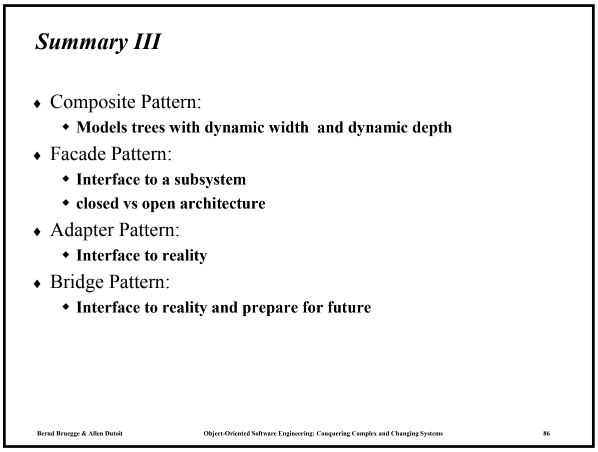
Summary I

- Object design closes the gap between the requirements and the machine.
- Object design is the process of adding details to the requirements analysis and making implementation decisions
- Object design activities include:
  - ✓ **Identification of Reuse**
  - ✓ **Identification of Inheritance and Delegation opportunities**
  - ✓ **Component selection**
- Object design is documented in the Object Design Document, which can be automatically generated from a specification using tools such as JavaDoc.



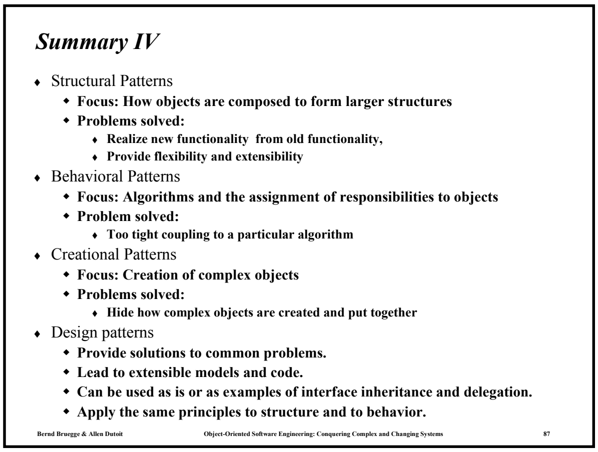
Summary II

- Design patterns are partial solutions to common problems such as
  - **such as separating an interface from a number of alternate implementations**
  - **wrapping around a set of legacy classes**
  - **protecting a caller from changes associated with specific platforms.**
- A design pattern is composed of a small number of classes
  - **use delegation and inheritance**
  - **provide a robust and modifiable solution.**
- These classes can be adapted and refined for the specific system under construction.
  - **Customization of the system**
  - **Reuse of existing solutions**



Summary III

- Composite Pattern:
  - **Models trees with dynamic width and dynamic depth**
- Facade Pattern:
  - **Interface to a subsystem**
  - **closed vs open architecture**
- Adapter Pattern:
  - **Interface to reality**
- Bridge Pattern:
  - **Interface to reality and prepare for future**



Summary IV

- Structural Patterns
  - **Focus: How objects are composed to form larger structures**
  - **Problems solved:**
    - **Realize new functionality from old functionality,**
    - **Provide flexibility and extensibility**
- Behavioral Patterns
  - **Focus: Algorithms and the assignment of responsibilities to objects**
  - **Problem solved:**
    - **Too tight coupling to a particular algorithm**
- Creational Patterns
  - **Focus: Creation of complex objects**
  - **Problems solved:**
    - **Hide how complex objects are created and put together**
- Design patterns
  - **Provide solutions to common problems.**
  - **Lead to extensible models and code.**
  - **Can be used as is or as examples of interface inheritance and delegation.**
  - **Apply the same principles to structure and to behavior.**