

Software Engineering


Session 8 – Main Theme
From Analysis and Design to
Software Architectures
(Part I)

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Presentation material partially based on textbook slides
Software Engineering: A Practitioner's Approach (7/e)
by Roger S. Pressman
Slides copyright © 1996, 2001, 2005, 2009

Agenda



- 1 Introduction**
- 2 Architectural Design**
- 3 Component-Level Design**
- 4 User Interface Design**
- 5 Pattern-Based Design**
- 6 Web Application Design**
- 7 Summary and Conclusion**

2

What is the class about?



▪ Course description and syllabus:

- » <http://www.nyu.edu/classes/jcf/g22.2440-001/>
- » <http://www.cs.nyu.edu/courses/spring10/G22.2440-001/>

▪ Textbooks:

- » **Software Engineering: A Practitioner's Approach**



Roger S. Pressman

McGraw-Hill Higher International

ISBN-10: 0-0712-6782-4, ISBN-13: 978-00711267823, 7th Edition (04/09)

- » http://highered.mcgraw-hill.com/sites/0073375977/information_center_view0/
- » http://highered.mcgraw-hill.com/sites/0073375977/information_center_view0/table_of_contents.html

3

Icons / Metaphors



Information



Common Realization



Knowledge/Competency Pattern



Governance



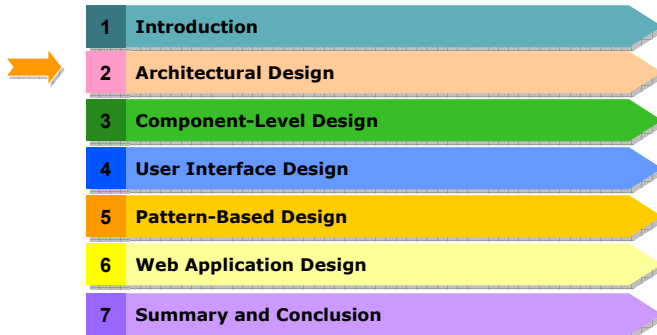
Alignment



Solution Approach

4

Agenda



5

Why Architecture?

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

- (1) **analyze the effectiveness of the design** in meeting its stated requirements,
- (2) **consider architectural alternatives** at a stage when making design changes is still relatively easy, and
- (3) **reduce the risks** associated with the construction of the software.

6

Why is Architecture Important?

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture “constitutes a relatively small, intellectually graspable mode of how the system is structured and how its components work together” [BAS03].

7

Architectural Descriptions

- The IEEE Computer Society has proposed IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive System*, [IEE00]
 - » to establish a conceptual framework and vocabulary for use during the design of software architecture,
 - » to provide detailed guidelines for representing an architectural description, and
 - » to encourage sound architectural design practices.
- The IEEE Standard defines an *architectural description* (AD) as a “a collection of products to document an architecture.”
 - » The description itself is represented using multiple views, where each *view* is “a representation of a whole system from the perspective of a related set of [stakeholder] concerns.”

8

Architectural Genres

- *Genre* implies a specific category within the overall software domain.
- Within each category, you encounter a number of subcategories.
 - » For example, within the genre of *buildings*, you would encounter the following general *styles*: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.
 - » Within each general style, more specific styles might apply. Each style would have a structure that can be described using a set of predictable patterns.

9

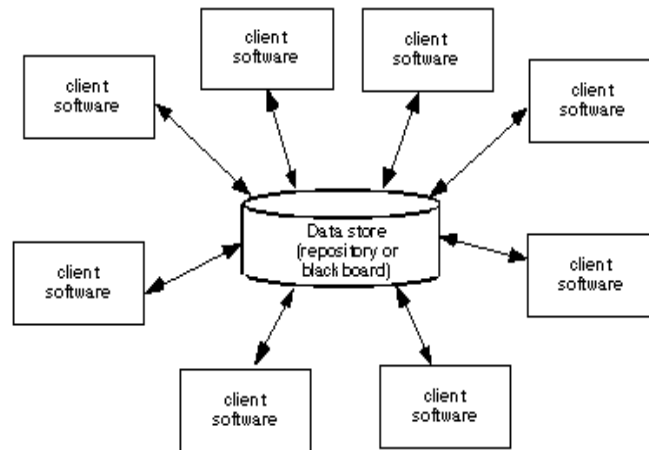
Architectural Styles

Each style describes a system category that encompasses: (1) a **set of components** (e.g., a database, computational modules) that perform a function required by a system, (2) a **set of connectors** that enable “communication, coordination and cooperation” among components, (3) **constraints** that define how components can be integrated to form the system, and (4) **semantic models** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

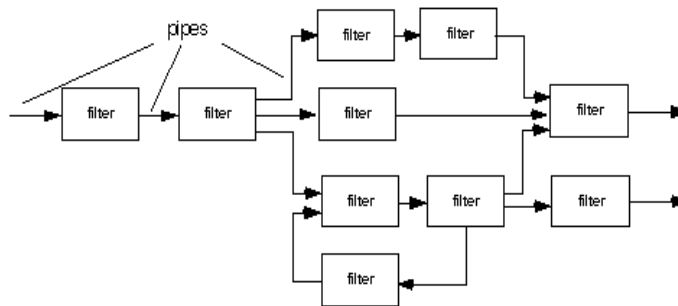
10

Data-Centered Architecture



11

Data Flow Architecture



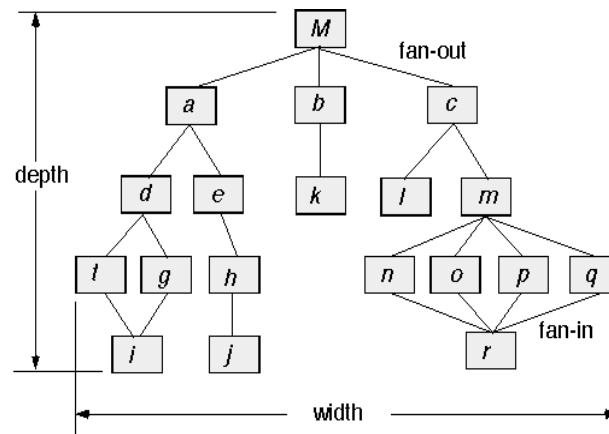
(a) pipes and filters



(b) batch sequential

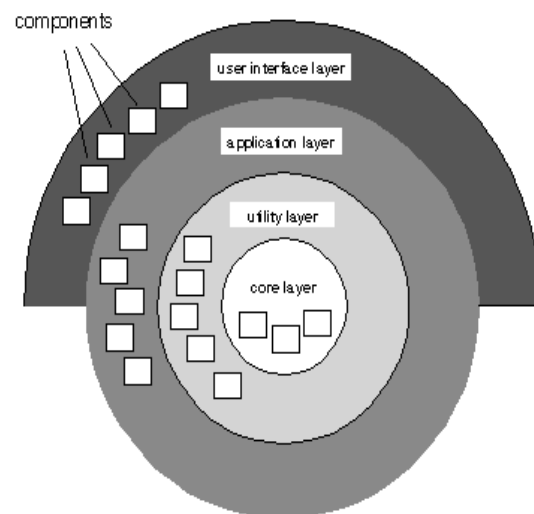
12

Call and Return Architecture



13

Layered Architecture



14

Architectural Patterns

- **Concurrency**—applications must handle multiple tasks in a manner that simulates parallelism
 - » *operating system process management* pattern
 - » *task scheduler* pattern
- **Persistence**—Data persists if it survives past the execution of the process that created it. Two patterns are common:
 - » a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture
 - » an *application level persistence* pattern that builds persistence features into the application architecture
- **Distribution**— the manner in which systems or components within systems communicate with one another in a distributed environment
 - » A *broker* acts as a 'middle-man' between the client component and a server component.

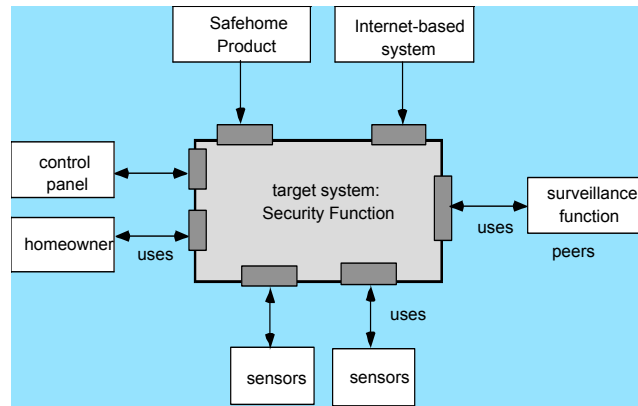
15

Architectural Design

- The software must be placed into context
 - » the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction
- A set of architectural archetypes should be identified
 - » An *archetype* is an abstraction (similar to a class) that represents one element of system behavior
- The designer specifies the structure of the system by defining and refining software components that implement each archetype

16

Architectural Context



17

Archetypes

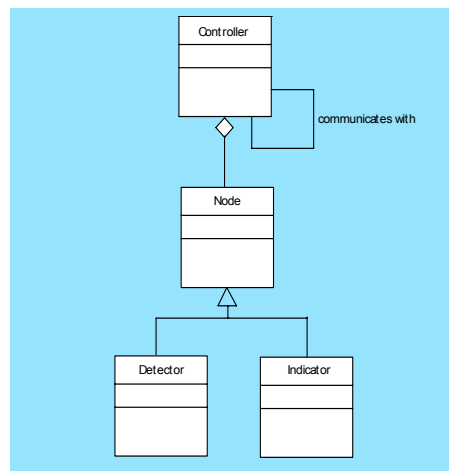
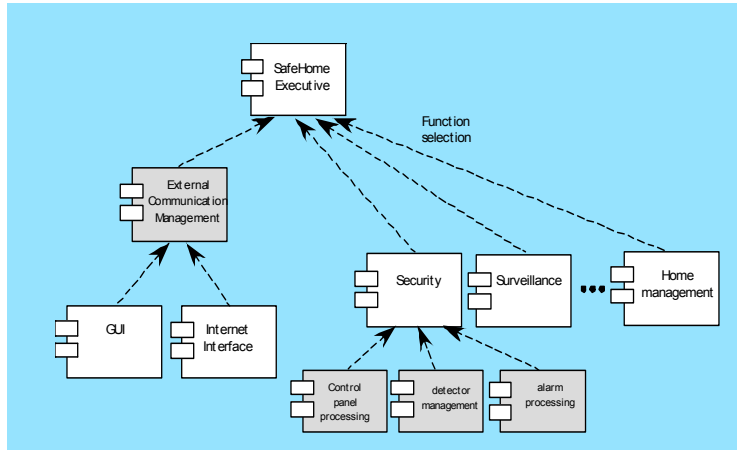


Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BO500])

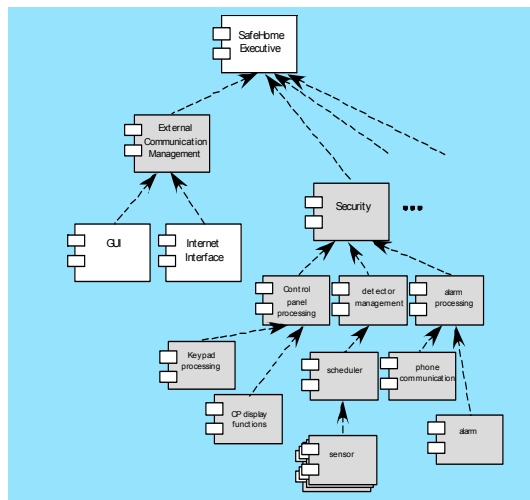
18

Component Structure



19

Refined Component Structure



20

Analyzing Architectural Design

1. Collect scenarios.
2. Elicit requirements, constraints, and environment description.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements:
 - module view
 - process view
 - data flow view
4. Evaluate quality attributes by considering each attribute in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.

21

Architectural Complexity

- the overall complexity of a proposed architecture is assessed by considering the **dependencies** between components within the architecture [Zha98]
 - » *Sharing dependencies* represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers.
 - » *Flow dependencies* represent dependence relationships between producers and consumers of resources.
 - » *Constrained dependencies* represent constraints on the relative flow of control among a set of activities.

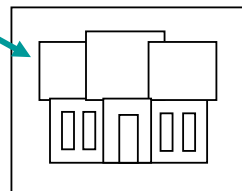
22

- *Architectural description language (ADL)* provides a semantics and syntax for describing a software architecture
- Provide the designer with the ability to:
 - » decompose architectural components
 - » compose individual components into larger architectural blocks and
 - » represent interfaces (connection mechanisms) between components.

An Architectural Design Method

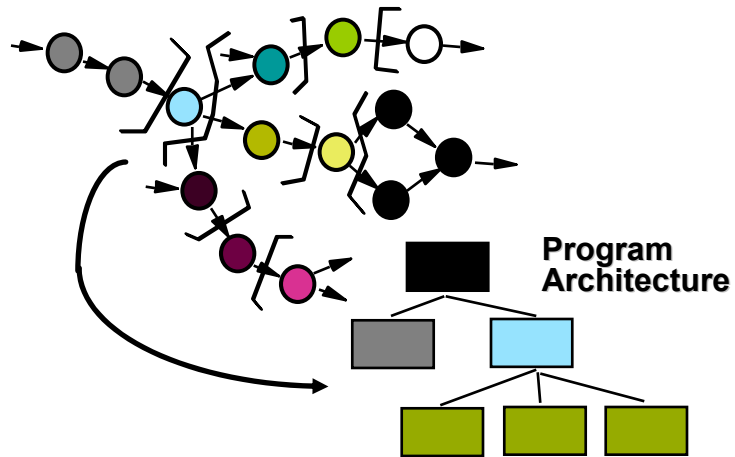
customer requirements

"four bedrooms, three baths,
lots of glass ..."



architectural design

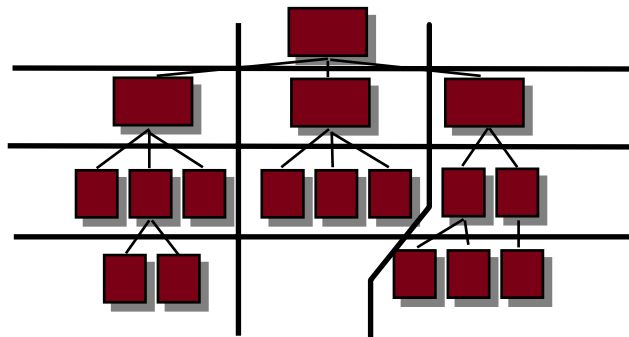
Deriving Program Architecture



25

Partitioning the Architecture

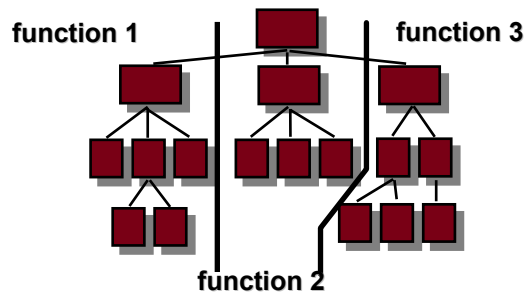
- “horizontal” and “vertical” partitioning are required



26

Horizontal Partitioning

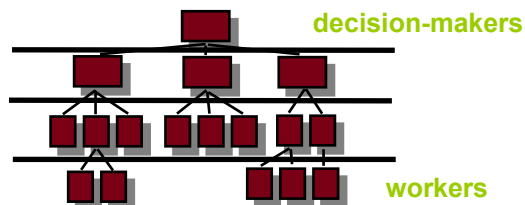
- define separate branches of the module hierarchy for each major function
- use control modules to coordinate communication between functions



27

Vertical Partitioning: Factoring

- design so that decision making and work are stratified
- decision making modules should reside at the top of the architecture



28

Why Partitioned Architecture?

- results in software that is easier to test
- leads to software that is easier to maintain
- results in propagation of fewer side effects
- results in software that is easier to extend

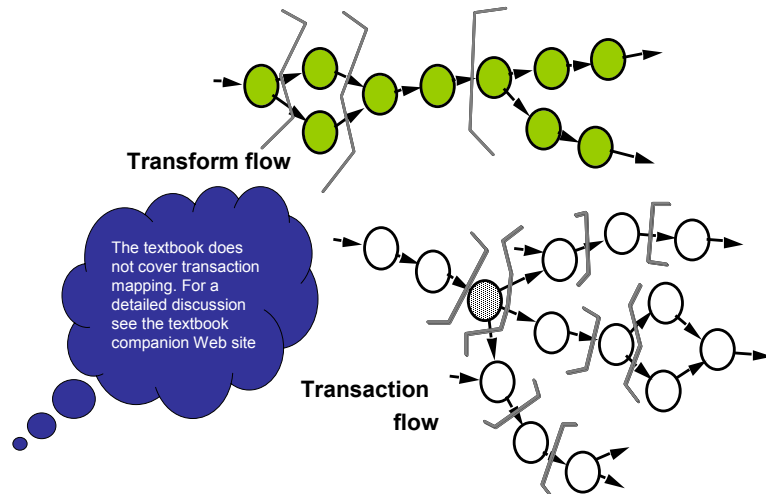
29

Structured Design

- objective: to derive a program architecture that is partitioned
- approach:
 - » a DFD is mapped into a program architecture
 - » the PSPEC and STD are used to indicate the content of each module
- notation: structure chart

30

Flow Characteristics



31

General Mapping Approach

- isolate incoming and outgoing flow boundaries; for transaction flows, isolate the transaction center
- working from the boundary outward, map DFD transforms into corresponding modules
- add control modules as required
- refine the resultant program structure using effective modularity concepts

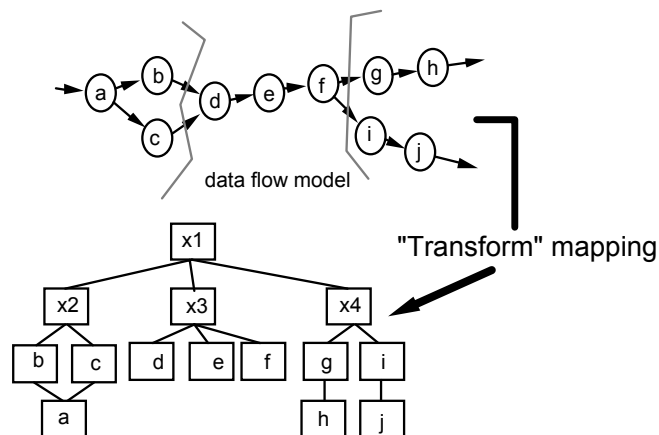
32

General Mapping Approach

- **Isolate the transform center by specifying incoming and outgoing flow boundaries**
- **Perform "first-level factoring."**
 - » The program architecture derived using this mapping results in a top-down distribution of control.
 - » *Factoring* leads to a program structure in which top-level components perform decision-making and low-level components perform most input, computation, and output work.
 - » Middle-level components perform some control and do moderate amounts of work.
- **Perform "second-level factoring."**

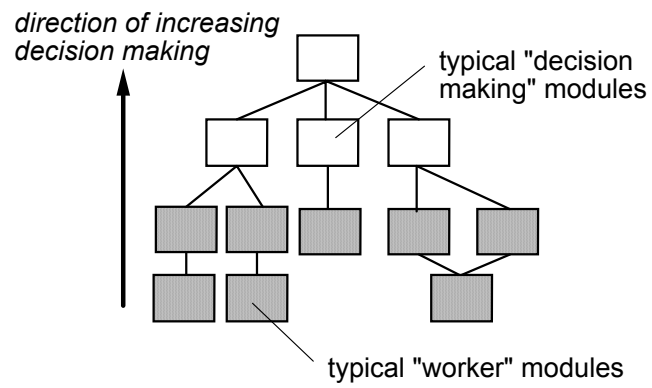
33

Transform Mapping



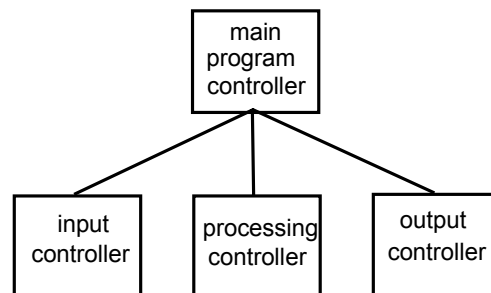
34

Factoring



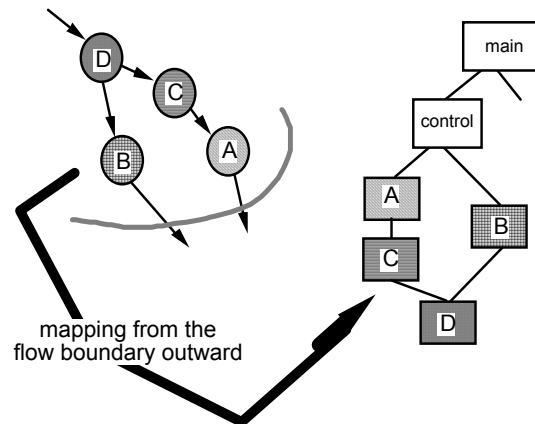
35

First Level Factoring



36

Second Level Mapping



37

Agenda

- 1 Introduction
- 2 Architectural Design
- ➔ 3 Component-Level Design
- 4 User Interface Design
- 5 Pattern-Based Design
- 6 Web Application Design
- 7 Summary and Conclusion

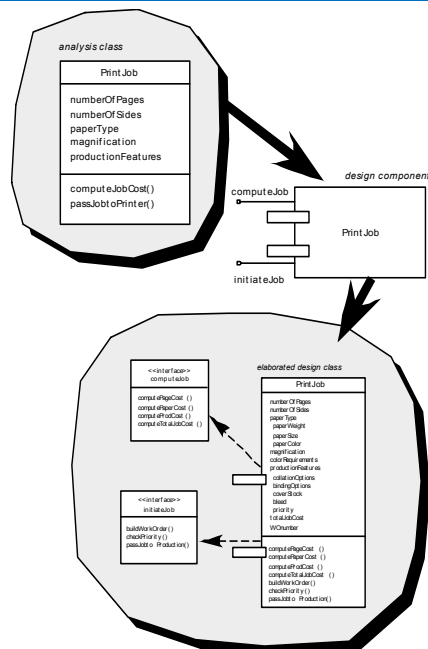
38

What is a Component?

- *OMG Unified Modeling Language Specification* [OMG01] defines a component as
 - » "... a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."
- *OO view*: a component contains a set of collaborating classes
- *Conventional view*: a component contains processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

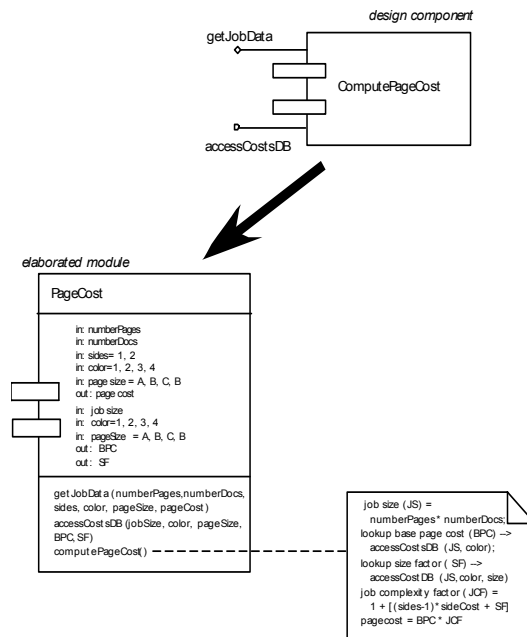
39

OO Component



40

Conventional Component



41

Basic Design Principles

- **The Open-Closed Principle (OCP).** "A module [component] should be open for extension but closed for modification."
- **The Liskov Substitution Principle (LSP).** "Subclasses should be substitutable for their base classes."
- **Dependency Inversion Principle (DIP).** "Depend on abstractions. Do not depend on concretions."
- **The Interface Segregation Principle (ISP).** "Many client-specific interfaces are better than one general purpose interface."
- **The Release Reuse Equivalency Principle (REP).** "The granule of reuse is the granule of release."
- **The Common Closure Principle (CCP).** "Classes that change together belong together."
- **The Common Reuse Principle (CRP).** "Classes that aren't reused together should not be grouped together."

Source: Martin, R., "Design Principles and Design Patterns," downloaded from <http://www.objectmentor.com>, 2000.

42

Design Guidelines

- **Components**
 - » Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
- **Interfaces**
 - » Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC)
- **Dependencies and Inheritance**
 - » it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

43

Cohesion

- **Conventional view:**
 - » the “single-mindedness” of a module
- **OO view:**
 - » *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- **Levels of cohesion**
 - » Functional
 - » Layer
 - » Communicational
 - » Sequential
 - » Procedural
 - » Temporal
 - » utility

44

Coupling

- Conventional view:
 - » The degree to which a component is connected to other components and to the external world
- OO view:
 - » a qualitative measure of the degree to which classes are connected to one another
- Level of coupling
 - » Content
 - » Common
 - » Control
 - » Stamp
 - » Data
 - » Routine call
 - » Type use
 - » Inclusion or import
 - » External

45

Component Level Design-I

- Step 1. Identify all design classes that correspond to the problem domain.
- Step 2. Identify all design classes that correspond to the infrastructure domain.
- Step 3. Elaborate all design classes that are not acquired as reusable components.
- Step 3a. Specify message details when classes or component collaborate.
- Step 3b. Identify appropriate interfaces for each component.

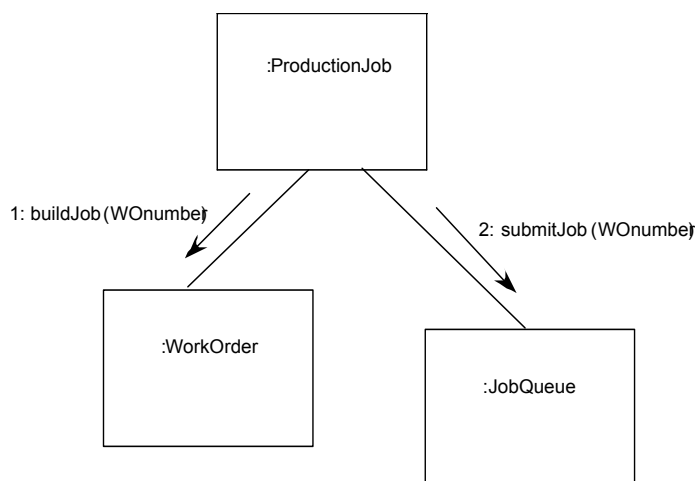
46

Component-Level Design-II

- Step 3c. Elaborate attributes and define data types and data structures required to implement them.
- Step 3d. Describe processing flow within each operation in detail.
- Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Step 5. Develop and elaborate behavioral representations for a class or component.
- Step 6. Elaborate deployment diagrams to provide additional implementation detail.
- Step 7. Factor every component-level design representation and always consider alternatives.

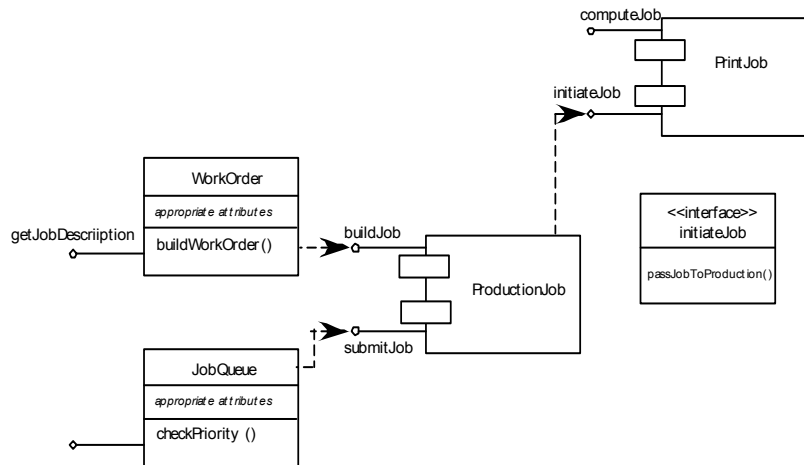
47

Collaboration Diagram



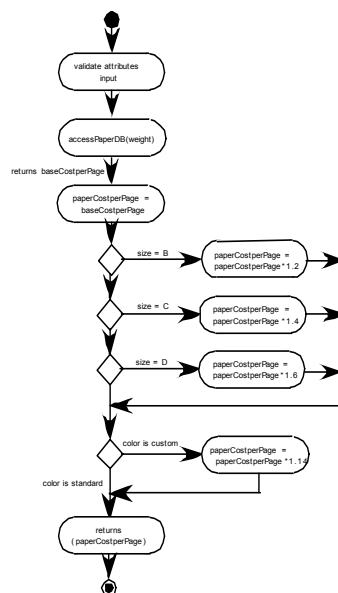
48

Refactoring



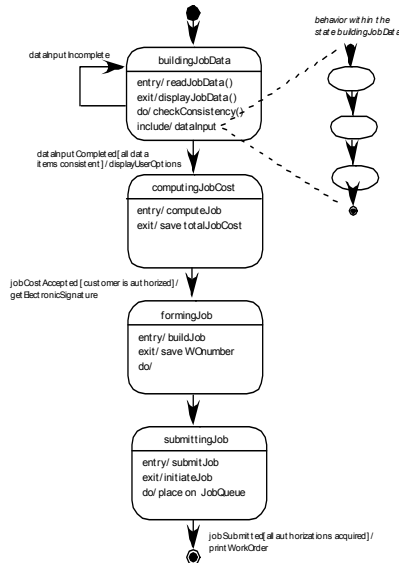
49

Activity Diagram



50

Statechart



51

Component Design for WebApps

- WebApp component is
 - » (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end-user, or
 - » (2) a cohesive package of content and functionality that provides end-user with some required capability.
- Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

52

Content Design for WebApps

- focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end-user
- consider a Web-based video surveillance capability within **SafeHomeAssured.com**
 - » potential content components can be defined for the video surveillance capability:
 - (1) the content objects that represent the space layout (the floor plan) with additional icons representing the location of sensors and video cameras;
 - (2) the collection of thumbnail video captures (each an separate data object), and
 - (3) the streaming video window for a specific camera.
 - » Each of these components can be separately named and manipulated as a package.

53

Functional Design for WebApps

- Modern Web applications deliver increasingly sophisticated processing functions that:
 - » (1) perform localized processing to generate content and navigation capability in a dynamic fashion;
 - » (2) provide computation or data processing capability that is appropriate for the WebApp's business domain;
 - » (3) provide sophisticated database query and access, or
 - » (4) establish data interfaces with external corporate systems.
- To achieve these (and many other) capabilities, you will design and construct WebApp functional components that are identical in form to software components for conventional software.

54

Designing Conventional Components

- The design of processing logic is governed by the basic principles of algorithm design and structured programming
- The design of data structures is defined by the data model developed for the system
- The design of interfaces is governed by the collaborations that a component must effect

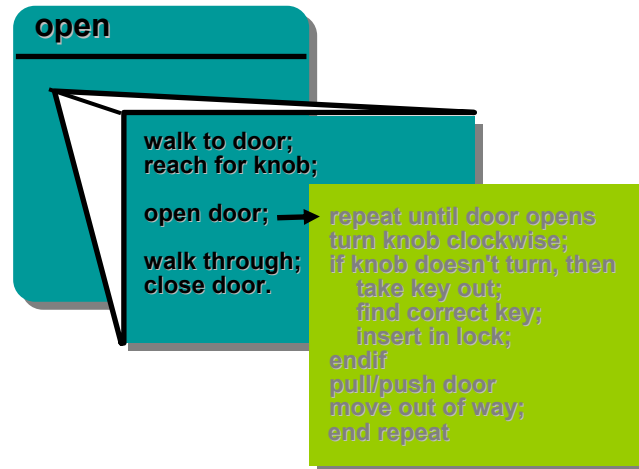
55

Algorithm Design

- the closest design activity to coding
- the approach:
 - » review the design description for the component
 - » use stepwise refinement to develop algorithm
 - » use structured programming to implement procedural logic
 - » use 'formal methods' to prove logic

56

Stepwise Refinement



57

Algorithm Design Model

- represents the algorithm at a level of detail that can be reviewed for quality
- options:
 - » graphical (e.g. flowchart, box diagram)
 - » pseudocode (e.g., PDL) ... choice of many
 - » programming language
 - » decision table

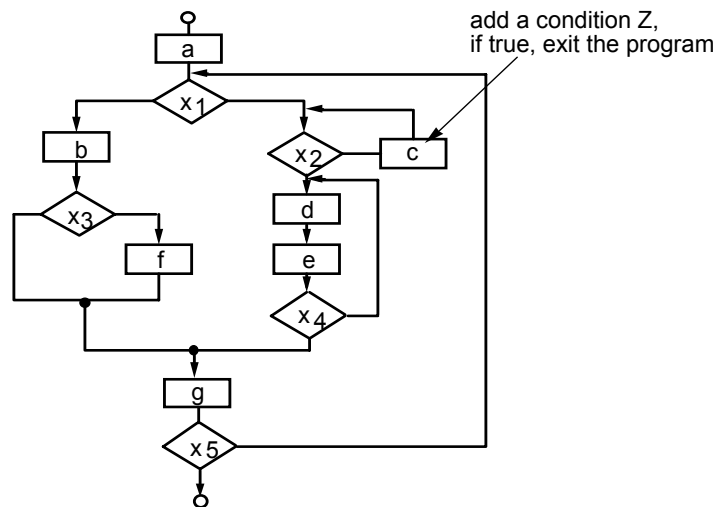
58

Structured Programming

- uses a limited set of logical constructs:
 - *sequence*
 - *conditional* — if-then-else, select-case
 - *loops* — do-while, repeat until
- leads to more readable, testable code
- can be used in conjunction with ‘proof of correctness’
- important for achieving high quality, but not enough

59

A Structured Procedural Design



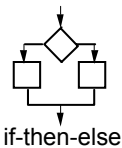
60

Decision Table

Conditions	Rules					
	1	2	3	4	5	6
regular customer	T	T				
silver customer			T	T		
gold customer					T	T
special discount	F	T	F	T	F	T
Rules						
no discount	✓					
apply 8 percent discount			✓	✓		
apply 15 percent discount					✓	✓
apply additional x percent discount		✓		✓		✓

61

Program Design Language (PDL)



if-then-else

```

if condition x
  then process a;
  else process b;
endif
  
```

PDL

- ☐ easy to combine with source code
- ☐ machine readable, no need for graphics input
- ☐ graphics can be generated from PDL
- ☐ enables declaration of data as well as procedure
- ☐ easier to maintain

62

Why Design Language?

- ☐ can be a derivative of the HOL of choice
e.g., Ada PDL
- ☐ machine readable and processable
- ☐ can be embedded with source code,
therefore easier to maintain
- ☐ can be represented in great detail, if
designer and coder are different
- ☐ easy to review

63

Component-Based Development

- When faced with the possibility of reuse, the software team asks:
 - » Are commercial off-the-shelf (COTS) components available to implement the requirement?
 - » Are internally-developed reusable components available to implement the requirement?
 - » Are the interfaces for available components compatible within the architecture of the system to be built?
- At the same time, they are faced with the following impediments to reuse ...

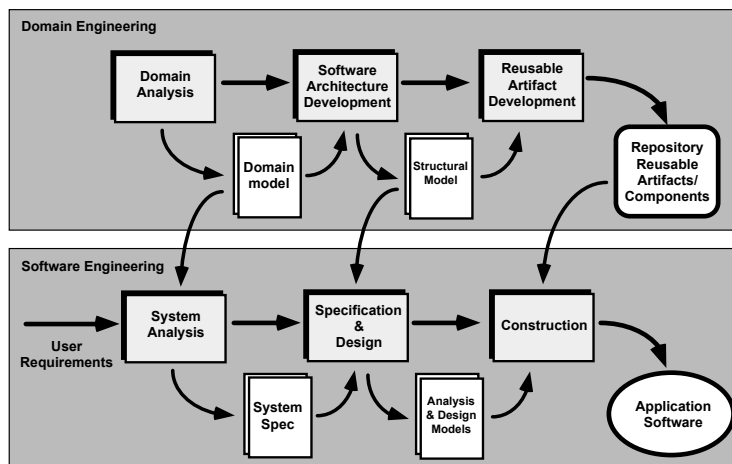
64

Impediments to Reuse

- Few companies and organizations have anything that even slightly resembles a comprehensive software reusability plan.
- Although an increasing number of software vendors currently sell tools or components that provide direct assistance for software reuse, the majority of software developers do not use them.
- Relatively little training is available to help software engineers and managers understand and apply reuse.
- Many software practitioners continue to believe that reuse is “more trouble than it’s worth.”
- Many companies continue to encourage software development methodologies which do not facilitate reuse
- Few companies provide an incentives to produce reusable program components.

65

The CBSE Process



66

Domain Engineering

1. Define the domain to be investigated.
2. Categorize the items extracted from the domain.
3. Collect a representative sample of applications in the domain.
4. Analyze each application in the sample.
5. Develop an analysis model for the objects.

67

Identifying Reusable Components

- Is component functionality required on future implementations?
- How common is the component's function within the domain?
- Is there duplication of the component's function within the domain?
- Is the component hardware-dependent?
- Does the hardware remain unchanged between implementations?
- Can the hardware specifics be removed to another component?
- Is the design optimized enough for the next implementation?
- Can we parameterize a non-reusable component so that it becomes reusable?
- Is the component reusable in many implementations with only minor changes?
- Is reuse through modification feasible?
- Can a non-reusable component be decomposed to yield reusable components?
- How valid is component decomposition for reuse?

68

Component-Based SE

- a library of components must be available
- components should have a consistent structure
- a standard should exist, e.g.,
 - » OMG/CORBA
 - » Microsoft COM
 - » Sun JavaBeans

69

CBSE Activities

- Component qualification
- Component adaptation
- Component composition
- Component update

70

Qualification

Before a component can be used, you must consider:

- application programming interface (API)
- development and integration tools required by the component
- run-time requirements including resource usage (e.g., memory or storage), timing or speed, and network protocol
- service requirements including operating system interfaces and support from other components
- security features including access controls and authentication protocol
- embedded design assumptions including the use of specific numerical or non-numerical algorithms
- exception handling

71

Adaptation

The implication of “easy integration” is:

- (1) that consistent methods of resource management have been implemented for all components in the library;
- (2) that common activities such as data management exist for all components, and
- (3) that interfaces within the architecture and with the external environment have been implemented in a consistent manner.

72

Composition

- An infrastructure must be established to bind components together
- Architectural ingredients for composition include:
 - » Data exchange model
 - » Automation
 - » Structured storage
 - » Underlying object model

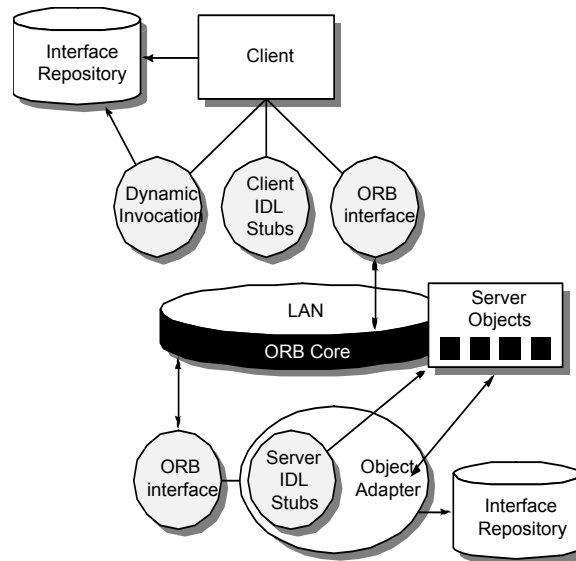
73

OMG/ CORBA

- The Object Management Group has published a *common object request broker architecture* (OMG/CORBA).
- An object request broker (ORB) provides services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.
- Integration of CORBA components (without modification) within a system is assured if an interface definition language (IDL) interface is created for every component.
- Objects within the client application request one or more services from the ORB server. Requests are made via an IDL or dynamically at run time.
- An interface repository contains all necessary information about the service's request and response formats.

74

ORB Architecture



75

Microsoft COM

- The *component object model* (COM) provides a specification for using components produced by various vendors within a single application running under the Windows operating system.
- COM encompasses two elements:
 - » COM interfaces (implemented as COM objects)
 - » a set of mechanisms for registering and passing messages between COM interfaces.

76

Sun JavaBeans

- The JavaBeans component system is a portable, platform independent CBSE infrastructure developed using the Java programming language.
- The JavaBeans component system encompasses a set of tools, called the *Bean Development Kit* (BDK), that allows developers to
 - » analyze how existing Beans (components) work
 - » customize their behavior and appearance
 - » establish mechanisms for coordination and communication
 - » develop custom Beans for use in a specific application
 - » test and evaluate Bean behavior.

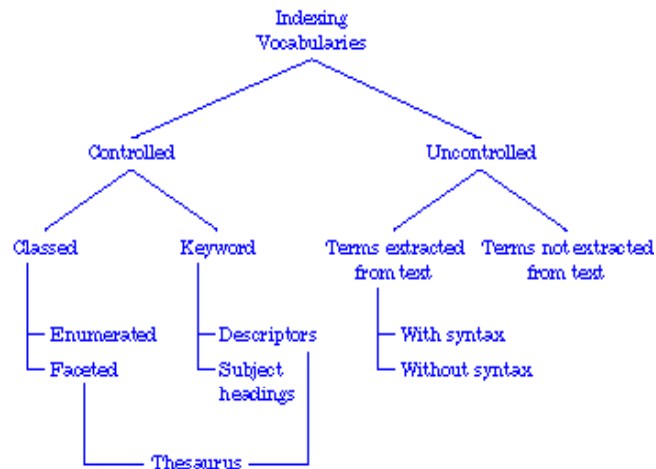
77

Classification

- **Enumerated classification**—components are described by defining a hierarchical structure in which classes and varying levels of subclasses of software components are defined
- **Faceted classification**—a domain area is analyzed and a set of basic descriptive features are identified
- **Attribute-value classification**—a set of attributes are defined for all components in a domain area

78

Indexing



79

The Reuse Environment

- A component database capable of storing software components and the classification information necessary to retrieve them.
- A library management system that provides access to the database.
- A software component retrieval system (e.g., an object request broker) that enables a client application to retrieve components and services from the library server.
- CBSE tools that support the integration of reused components into a new design or implementation.

80

Agenda

- 1 Introduction
- 2 Architectural Design
- 3 Component-Level Design
- 4 User Interface Design
- 5 Pattern-Based Design
- 6 Web Application Design
- 7 Summary and Conclusion

81

Interface Design

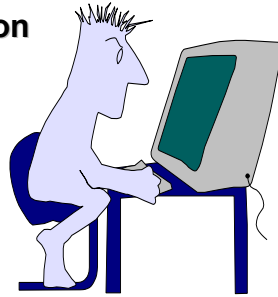
Easy to learn?
Easy to use?
Easy to understand?



82

Typical Design Errors

lack of consistency
too much memorization
no guidance / help
no context sensitivity
poor response
Arcane/unfriendly



Golden Rules

- Place the user in control
- Reduce the user's memory load
- Make the interface consistent

Place the User in Control

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction.
- Allow user interaction to be interruptible and undoable.
Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen.

85

Reduce the User's Memory Load

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real world metaphor.
- Disclose information in a progressive fashion.

86

Make the Interface Consistent

- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

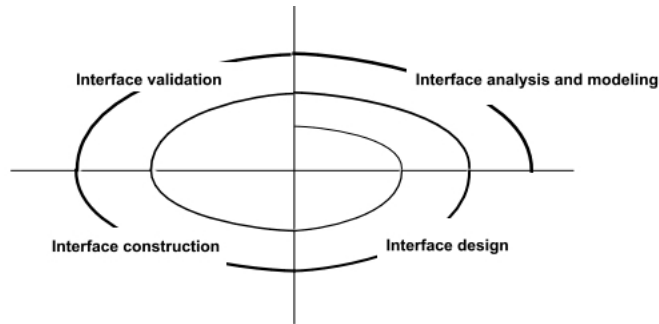
87

User Interface Design Models

- **User model** — a profile of all end users of the system
- **Design model** — a design realization of the user model
- **Mental model (system perception)** — the user's mental image of what the interface is
- **Implementation model** — the interface “look and feel” coupled with supporting information that describe interface syntax and semantics

88

User Interface Design Process



89

Interface Analysis

- Interface analysis means understanding
 - » (1) the people (end-users) who will interact with the system through the interface;
 - » (2) the tasks that end-users must perform to do their work,
 - » (3) the content that is presented as part of the interface
 - » (4) the environment in which these tasks will be conducted.

90

User Analysis

- Are users trained professionals, technician, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology the sits behind the interface?

91

Task Analysis and Modeling

- Answers the following questions ...
 - » What work will the user perform in specific circumstances?
 - » What tasks and subtasks will be performed as the user does the work?
 - » What specific problem domain objects will the user manipulate as work is performed?
 - » What is the sequence of work tasks—the workflow?
 - » What is the hierarchy of tasks?
- **Use-cases** define basic interaction
- **Task elaboration** refines interactive tasks
- **Object elaboration** identifies interface objects (classes)
- **Workflow analysis** defines how a work process is completed when several people (and roles) are involved

92

Swimlane Diagram

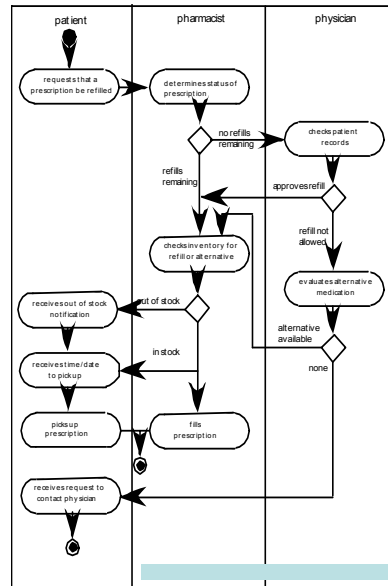


Figure 12.2 Swimlane diagram for prescription refill function

93

Analysis of Display Content

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data.
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color to be used to enhance understanding?
- How will error messages and warning be presented to the user?

94

Interface Design Steps

- Using information developed during interface analysis, **define interface objects and actions (operations)**.
- **Define events (user actions)** that will cause the state of the user interface to change. Model this behavior.
- **Depict each interface state** as it will actually look to the end-user.
- **Indicate how the user interprets the state of the system** from information provided through the interface.

95

Design Issues

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility
- Internationalization

96

WebApp Interface Design

- *Where am I?* The interface should
 - » provide an indication of the WebApp that has been accessed
 - » inform the user of her location in the content hierarchy.
- *What can I do now?* The interface should always help the user understand his current options
 - » what functions are available?
 - » what links are live?
 - » what content is relevant?
- *Where have I been, where am I going?* The interface must facilitate navigation.
 - » Provide a “map” (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the WebApp.

97

Effective WebApp Interfaces

- Bruce Tognozzi [TOG01] suggests...
 - » *Effective interfaces are visually apparent and forgiving*, instilling in their users a sense of control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.
 - » *Effective interfaces do not concern the user with the inner workings of the system.* Work is carefully and continuously saved, with full option for the user to undo any activity at any time.
 - » *Effective applications and services perform a maximum of work*, while requiring a minimum of information from users.

98

Interface Design Principles-I

- **Anticipation**—A WebApp should be designed so that it anticipates the use's next move.
- **Communication**—The interface should communicate the status of any activity initiated by the user
- **Consistency**—The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- **Controlled autonomy**—The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- **Efficiency**—The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client-server environment that executes it.

99

Interface Design Principles-II

- **Focus**—The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- **Fitt's Law**—"The time to acquire a target is a function of the distance to and size of the target."
- **Human interface objects**—A vast library of reusable human interface objects has been developed for WebApps.
- **Latency reduction**—The WebApp should use multi-tasking in a way that lets the user proceed with work as if the operation has been completed.
- **Learnability**— A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.

100

Interface Design Principles-III

- **Maintain work product integrity**—A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
- **Readability**—All information presented through the interface should be readable by young and old.
- **Track state**—When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
- **Visible navigation**—A well-designed WebApp interface provides “the illusion that users are in the same place, with the work brought to them.”

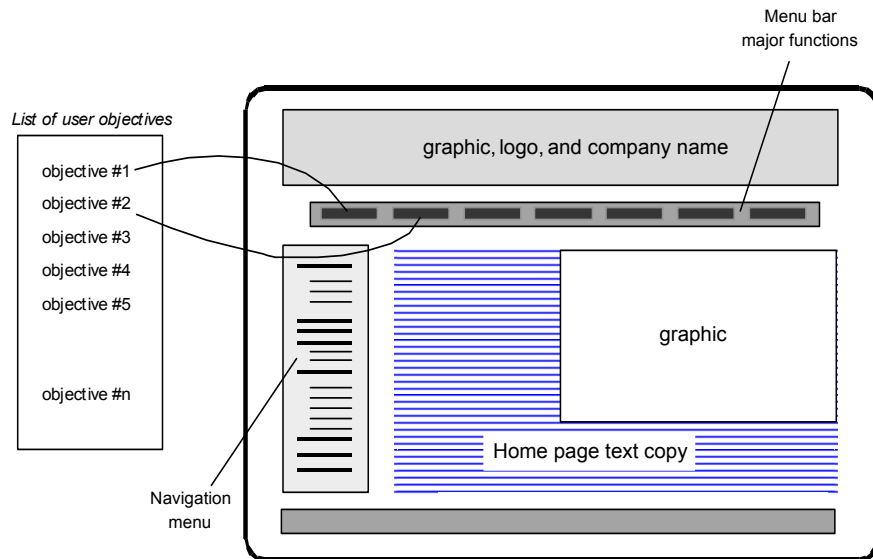
101

Interface Design Workflow-I

- Review information contained in the analysis model and refine as required.
- Develop a rough sketch of the WebApp interface layout.
- Map user objectives into specific interface actions.
- Define a set of user tasks that are associated with each action.
- Storyboard screen images for each interface action.
- Refine interface layout and storyboards using input from aesthetic design.

102

Mapping User Objectives



103

Interface Design Workflow-II

- Identify user interface objects that are required to implement the interface.
- Develop a procedural representation of the user's interaction with the interface.
- Develop a behavioral representation of the interface.
- Describe the interface layout for each state.
- Refine and review the interface design model.

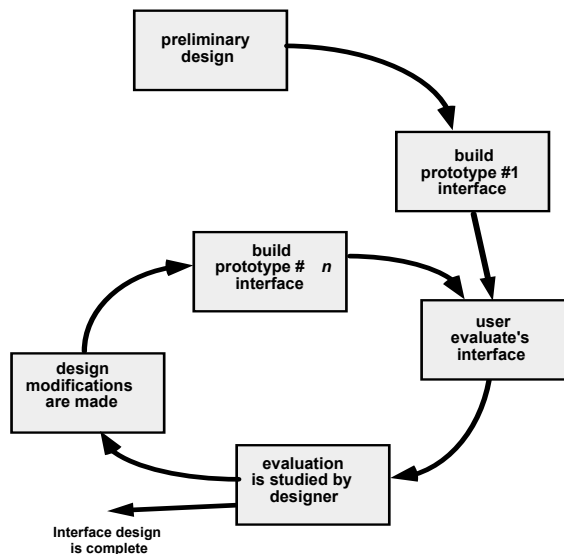
104

Aesthetic Design

- Don't be afraid of white space.
- Emphasize content.
- Organize layout elements from top-left to bottom right.
- Group navigation, content, and function geographically within the page.
- Don't extend your real estate with the scrolling bar.
- Consider resolution and browser window size when designing layout.

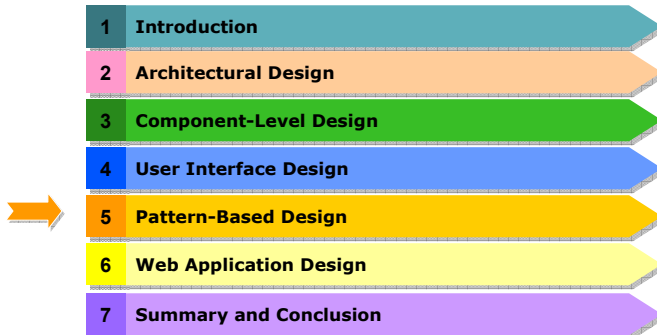
105

Design Evaluation Cycle



106

Agenda



107

Design Patterns

- Each of us has encountered a design problem and silently thought: *I wonder if anyone has developed a solution to for this?*
 - » What if there was a standard way of describing a problem (so you could look it up), and an organized method for representing the solution to the problem?
- *Design patterns* are a codified method for describing problems and their solution allows the software engineering community to capture design knowledge in a way that enables it to be reused.

108

Design Patterns

- *Each pattern describes a problem that 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 the solution a million times over without ever doing it the same way twice.*

—Christopher Alexander, 1977

- “a three-part rule which expresses a relation between a certain context, a problem, and a solution.”

109

Basic Concepts

- *Context* allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment.
- A set of requirements, including limitations and constraints, acts as a *system of forces* that influences how
 - » the problem can be interpreted within its context and
 - » how the solution can be effectively applied.

110

Effective Patterns

- Coplien [Cop05] characterizes an effective design pattern in the following way:
 - » *It solves a problem*: Patterns capture solutions, not just abstract principles or strategies.
 - » *It is a proven concept*: Patterns capture solutions with a track record, not theories or speculation.
 - » *The solution isn't obvious*: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns *generate* a solution to a problem indirectly--a necessary approach for the most difficult problems of design.
 - » *It describes a relationship*: Patterns don't just describe modules, but describe deeper system structures and mechanisms.
 - » *The pattern has a significant human component (minimize human intervention)*. All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

111

Generative Patterns

- *Generative patterns* describe an important and repeatable aspect of a system and then provide us with a way to build that aspect within a system of forces that are unique to a given context.
- A collection of generative design patterns could be used to “generate” an application or computer-based system whose architecture enables it to adapt to change.

112

Kinds of Patterns

- *Architectural patterns* describe broad-based design problems that are solved using a structural approach.
- *Data patterns* describe recurring data-oriented problems and the data modeling solutions that can be used to solve them.
- *Component patterns* (also referred to as *design patterns*) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture
- *Interface design patterns* describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end-users.
- *WebApp patterns* address a problem set that is encountered when building WebApps and often incorporates many of the other patterns categories just mentioned.

113

Kinds of Patterns

- *Creational patterns* focus on the “creation, composition, and representation of objects, e.g.,
 - » **Abstract factory pattern**: centralize decision of what [factory](#) to instantiate
 - » **Factory method pattern**: centralize creation of an object of a specific type choosing one of several implementations
- *Structural patterns* focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure, e.g.,
 - » **Adapter pattern**: 'adapts' one interface for a class into one that a client expects
 - » **Aggregate pattern**: a version of the [Composite pattern](#) with methods for aggregation of children
- *Behavioral patterns* address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects, e.g.,
 - » **Chain of responsibility pattern**: Command objects are handled or passed on to other objects by logic-containing processing objects
 - » **Command pattern**: Command objects encapsulate an action and its parameters

114

Frameworks

- Patterns themselves may not be sufficient to develop a complete design.
 - » In some cases it may be necessary to provide an implementation-specific skeletal infrastructure, called a *framework*, for design work.
 - » That is, you can select a “*reusable mini-architecture* that provides the generic structure and behavior for a family of software abstractions, along with a context ... which specifies their collaboration and use within a given domain.” [Amb98]
- A framework is not an architectural pattern, but rather a skeleton with a collection of “plug points” (also called *hooks* and *slots*) that enable it to be adapted to a specific problem domain.
 - » The plug points enable you to integrate problem specific classes or functionality within the skeleton.

115

Describing a Pattern

- *Pattern name*—describes the essence of the pattern in a short but expressive name
- *Problem*—describes the problem that the pattern addresses
- *Motivation*—provides an example of the problem
- *Context*—describes the environment in which the problem resides including application domain
- *Forces*—lists the system of forces that affect the manner in which the problem must be solved; includes a discussion of limitation and constraints that must be considered
- *Solution*—provides a detailed description of the solution proposed for the problem
- *Intent*—describes the pattern and what it does
- *Collaborations*—describes how other patterns contribute to the solution
- *Consequences*—describes the potential trade-offs that must be considered when the pattern is implemented and the consequences of using the pattern
- *Implementation*—identifies special issues that should be considered when implementing the pattern
- *Known uses*—provides examples of actual uses of the design pattern in real applications
- *Related patterns*—cross-references related design patterns

116

Pattern Languages

- A *pattern language* encompasses a collection of patterns
 - » each described using a standardized template (See section 12.1.3 of textbook) and
 - » interrelated to show how these patterns collaborate to solve problems across an application domain.
- a pattern language is analogous to a hypertext instruction manual for problem solving in a specific application domain.
 - » The problem domain under consideration is first described hierarchically, beginning with broad design problems associated with the domain and then refining each of the broad problems into lower levels of abstraction.

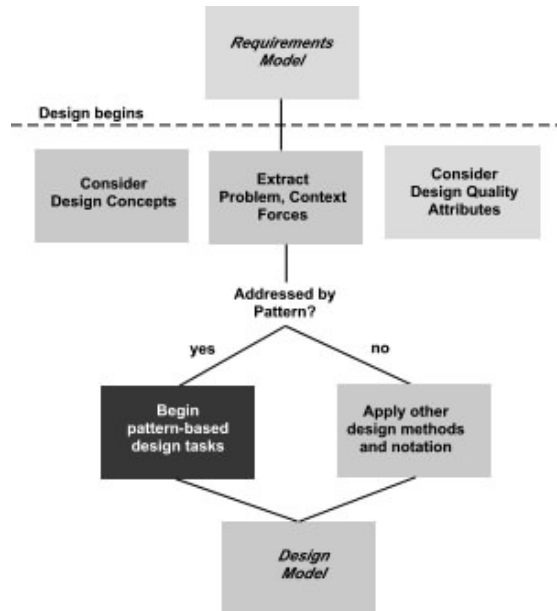
117

Pattern-Based Design

- A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system.
- The requirements model describes the problem set, establishes the context, and identifies the system of forces that hold sway.
- Then ...

118

Pattern-Based Design



119

Thinking in Patterns

- Shalloway and Trott [Sha05] suggest the following approach that enables a designer to think in patterns:
 - » 1. Be sure you understand the big picture—the context in which the software to be built resides. The requirements model should communicate this to you.
 - » 2. Examining the big picture, extract the patterns that are present at that level of abstraction.
 - » 3. Begin your design with ‘big picture’ patterns that establish a context or skeleton for further design work.
 - » 4. “Work inward from the context” [Sha05] looking for patterns at lower levels of abstraction that contribute to the design solution.
 - » 5. Repeat steps 1 to 4 until the complete design is fleshed out.
 - » 6. Refine the design by adapting each pattern to the specifics of the software you’re trying to build.

120

Design Tasks—I

- Examine the requirements model and develop a problem hierarchy.
- Determine if a reliable pattern language has been developed for the problem domain.
- Beginning with a broad problem, determine whether one or more architectural patterns are available for it.
- Using the collaborations provided for the architectural pattern, examine subsystem or component level problems and search for appropriate patterns to address them.
- Repeat steps 2 through 5 until all broad problems have been addressed.

121

Design Tasks—II

- If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns.
- Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented.
- Be certain to refine the design as it is derived from patterns using design quality criteria as a guide.

122

Pattern Organizing Table

	Database	Application	Implementation	Infrastructure
<i>Data/Content</i>				
<i>Problem statement ...</i>	PatternName (s)		PatternName (s)	
<i>Problem statement ...</i>		PatternName (s)		PatternName (s)
<i>Problem statement ...</i>	PatternName (s)			PatternName (s)
<i>Architecture</i>				
<i>Problem statement ...</i>		PatternName (s)		
<i>Problem statement ...</i>		PatternName (s)		PatternName (s)
<i>Problem statement ...</i>				
<i>Component-level</i>				
<i>Problem statement ...</i>		PatternName (s)	PatternName (s)	
<i>Problem statement ...</i>				PatternName (s)
<i>Problem statement ...</i>		PatternName (s)	PatternName (s)	
<i>User Interface</i>				
<i>Problem statement ...</i>		PatternName (s)	PatternName (s)	
<i>Problem statement ...</i>		PatternName (s)	PatternName (s)	
<i>Problem statement ...</i>		PatternName (s)	PatternName (s)	

123

Common Design Mistakes

- Not enough time has been spent to understand the underlying problem, its context and forces, and as a consequence, you select a pattern that looks right, but is inappropriate for the solution required.
- Once the wrong pattern is selected, you refuse to see your error and force fit the pattern.
- In other cases, the problem has forces that are not considered by the pattern you've chosen, resulting in a poor or erroneous fit.
- Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented.

124

Architectural Patterns

- Example: every house (and every architectural style for houses) employs a **Kitchen** pattern.
- The **Kitchen** pattern and patterns it collaborates with address problems associated with the storage and preparation of food, the tools required to accomplish these tasks, and rules for placement of these tools relative to workflow in the room.
- In addition, the pattern might address problems associated with counter tops, lighting, wall switches, a central island, flooring, and so on.
- Obviously, there is more than a single design for a kitchen, often dictated by the context and system of forces. But every design can be conceived within the context of the 'solution' suggested by the **Kitchen** pattern.

125

Patterns Repositories

- There are many sources for design patterns available on the Web. Some patterns can be obtained from individually published pattern languages, while others are available as part of a patterns portal or patterns repository.
- A list of patterns repositories is presented in the sidebar (see section 12.3 of textbook)

126

Component-Level Patterns

- Component-level design patterns provide a proven solution that addresses one or more sub-problems extracted from the requirement model.
- In many cases, design patterns of this type focus on some functional element of a system.
- For example, the **SafeHomeAssured.com** application must address the following design sub-problem: *How can we get product specifications and related information for any SafeHome device?*

127

Component-Level Patterns

- Having enunciated the sub-problem that must be solved, consider context and the system of forces that affect the solution.
- Examining the appropriate requirements model use case, the specification for a *SafeHome* device (e.g., a security sensor or camera) is used for informational purposes by the consumer.
 - » However, other information that is related to the specification (e.g., pricing) may be used when e-commerce functionality is selected.
- The solution to the sub-problem involves a **search**. Since searching is a very common problem, it should come as no surprise that there are many search-related patterns.
- See Section 12.4 of textbook

128

User Interface (UI) Patterns

- **Whole UI.** Provide design guidance for top-level structure and navigation throughout the entire interface.
- **Page layout.** Address the general organization of pages (for Websites) or distinct screen displays (for interactive applications)
- **Forms and input.** Consider a variety of design techniques for completing form-level input.
- **Tables.** Provide design guidance for creating and manipulating tabular data of all kinds.
- **Direct data manipulation.** Address data editing, modification, and transformation.
- **Navigation.** Assist the user in navigating through hierarchical menus, Web pages, and interactive display screens.
- **Searching.** Enable content-specific searches through information maintained within a Web site or contained by persistent data stores that are accessible via an interactive application.
- **Page elements.** Implement specific elements of a Web page or display screen.
- **E-commerce.** Specific to Web sites, these patterns implement recurring elements of e-commerce applications.

129

WebApp Patterns

- **Information architecture patterns** relate to the overall structure of the information space, and the ways in which users will interact with the information.
- **Navigation patterns** define navigation link structures, such as hierarchies, rings, tours, and so on.
- **Interaction patterns** contribute to the design of the user interface. Patterns in this category address how the interface informs the user of the consequences of a specific action; how a user expands content based on usage context and user desires; how to best describe the destination that is implied by a link; how to inform the user about the status of an on-going interaction, and interface related issues.
- **Presentation patterns** assist in the presentation of content as it is presented to the user via the interface. Patterns in this category address how to organize user interface control functions for better usability; how to show the relationship between an interface action and the content objects it affects, and how to establish effective content hierarchies.
- **Functional patterns** define the workflows, behaviors, processing, communications, and other algorithmic elements within a WebApp.

130

Design Granularity

- When a problem involves “big picture” issues, attempt to develop solutions (and use relevant patterns) that focus on the big picture.
- Conversely, when the focus is very narrow (e.g., uniquely selecting one item from a small set of five or fewer items), the solution (and the corresponding pattern) is targeted quite narrowly.
- In terms of the level of granularity, patterns can be described at the following levels:

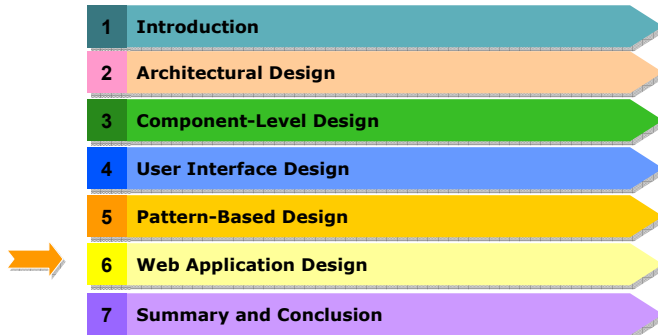
131

Design Granularity

- **Architectural patterns.** This level of abstraction will typically relate to patterns that define the overall structure of the WebApp, indicate the relationships among different components or increments, and define the rules for specifying relationships among the elements (pages, packages, components, subsystems) of the architecture.
- **Design patterns.** These address a specific element of the design such as an aggregation of components to solve some design problem, relationships among elements on a page, or the mechanisms for effecting component to component communication. An example might be the *Broadsheet* pattern for the layout of a WebApp homepage.
- **Component patterns.** This level of abstraction relates to individual small-scale elements of a WebApp. Examples include individual interaction elements (e.g. radio buttons, text boxes), navigation items (e.g. how might you format links?) or functional elements (e.g. specific algorithms).

132

Agenda



133

Design & WebApps

“There are essentially two basic approaches to design: the artistic ideal of expressing yourself and the engineering ideal of solving a problem for a customer.”

Jakob Nielsen

- *When should we emphasize WebApp design?*
 - » when content and function are complex
 - » when the size of the WebApp encompasses hundreds of content objects, functions, and analysis classes
 - » when the success of the WebApp will have a direct impact on the success of the business

134

Design & WebApp Quality

- **Security**
 - » Rebuff external attacks
 - » Exclude unauthorized access
 - » Ensure the privacy of users/customers
- **Availability**
 - » the measure of the percentage of time that a WebApp is available for use
- **Scalability**
 - » **Can** the WebApp and the systems with which it is interfaced handle significant variation in user or transaction volume
- **Time to Market**

135

Quality Dimensions for End-Users

- **Time**
 - » How much has a Web site changed since the last upgrade?
 - » How do you highlight the parts that have changed?
- **Structural**
 - » How well do all of the parts of the Web site hold together.
 - » Are all links inside and outside the Web site working?
 - » Do all of the images work?
 - » Are there parts of the Web site that are not connected?
- **Content**
 - » Does the content of critical pages match what is supposed to be there?
 - » Do key phrases exist continually in highly-changeable pages?
 - » Do critical pages maintain quality content from version to version?
 - » What about dynamically generated HTML pages?

136

Quality Dimensions for End-Users

- **Accuracy and Consistency**
 - » Are today's copies of the pages downloaded the same as yesterday's? Close enough?
 - » Is the data presented accurate enough? How do you know?
- **Response Time and Latency**
 - » Does the Web site server respond to a browser request within certain parameters?
 - » In an E-commerce context, how is the end to end response time after a SUBMIT?
 - » Are there parts of a site that are so slow the user declines to continue working on it?
- **Performance**
 - » Is the Browser-Web site-Web-Browser connection quick enough?
 - » How does the performance vary by time of day, by load and usage?
 - » Is performance adequate for E-commerce applications?

137

WebApp Design Goals

- **Consistency**
 - » **Content** should be constructed consistently
 - » **Graphic design (aesthetics)** should present a consistent look across all parts of the WebApp
 - » **Architectural design** should establish templates that lead to a consistent hypermedia structure
 - » **Interface design** should define consistent modes of interaction, navigation and content display
 - » **Navigation mechanisms** should be used consistently across all WebApp elements

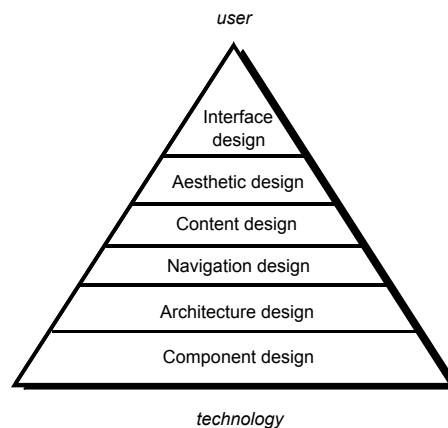
138

WebApp Design Goals

- **Identity**
 - » Establish an “identity” that is appropriate for the business purpose
- **Robustness**
 - » The user expects robust content and functions that are relevant to the user’s needs
- **Navigability**
 - » designed in a manner that is intuitive and predictable
- **Visual appeal**
 - » the look and feel of content, interface layout, color coordination, the balance of text, graphics and other media, navigation mechanisms must appeal to end-users
- **Compatibility**
 - » With all appropriate environments and configurations

139

WebE Design Pyramid



140

WebApp Interface Design

- *Where am I?* The interface should
 - » provide an indication of the WebApp that has been accessed
 - » inform the user of her location in the content hierarchy.
- *What can I do now?* The interface should always help the user understand his current options
 - » what functions are available?
 - » what links are live?
 - » what content is relevant?
- *Where have I been, where am I going?* The interface must facilitate navigation.
 - » Provide a “map” (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the WebApp.

141

Effective WebApp Interfaces

- Bruce Tognozzi [TOG01] suggests...
 - » *Effective interfaces are visually apparent and forgiving*, instilling in their users a sense of control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.
 - » *Effective interfaces do not concern the user with the inner workings of the system.* Work is carefully and continuously saved, with full option for the user to undo any activity at any time.
 - » *Effective applications and services perform a maximum of work*, while requiring a minimum of information from users.

142

Interface Design Principles-I

- **Anticipation**—A WebApp should be designed so that it anticipates the use's next move.
- **Communication**—The interface should communicate the status of any activity initiated by the user
- **Consistency**—The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- **Controlled autonomy**—The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- **Efficiency**—The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client-server environment that executes it.

143

Interface Design Principles-II

- **Focus**—The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- **Fitt's Law**—"The time to acquire a target is a function of the distance to and size of the target."
- **Human interface objects**—A vast library of reusable human interface objects has been developed for WebApps.
- **Latency reduction**—The WebApp should use multi-tasking in a way that lets the user proceed with work as if the operation has been completed.
- **Learnability**— A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.

144

Interface Design Principles-III

- **Maintain work product integrity**—A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
- **Readability**—All information presented through the interface should be readable by young and old.
- **Track state**—When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
- **Visible navigation**—A well-designed WebApp interface provides “the illusion that users are in the same place, with the work brought to them.”

145

Aesthetic Design

- Don't be afraid of white space.
- Emphasize content.
- Organize layout elements from top-left to bottom right.
- Group navigation, content, and function geographically within the page.
- Don't extend your real estate with the scrolling bar.
- Consider resolution and browser window size when designing layout.

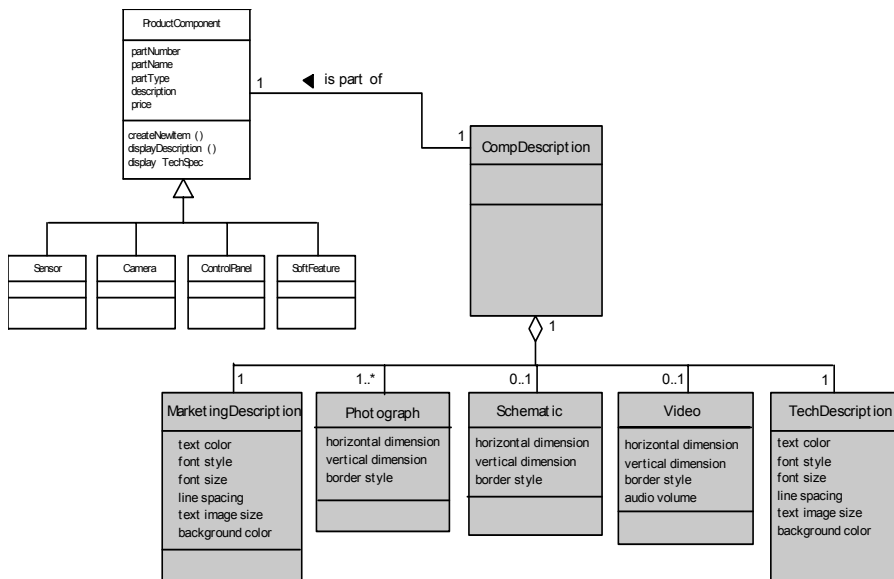
146

Content Design

- Develops a design representation for content objects
 - » For WebApps, a content object is more closely aligned with a data object for conventional software
- Represents the mechanisms required to instantiate their relationships to one another.
 - » analogous to the relationship between analysis classes and design components described in Chapter 11
- A content object has attributes that include content-specific information and implementation-specific attributes that are specified as part of design

147

Design of Content Objects



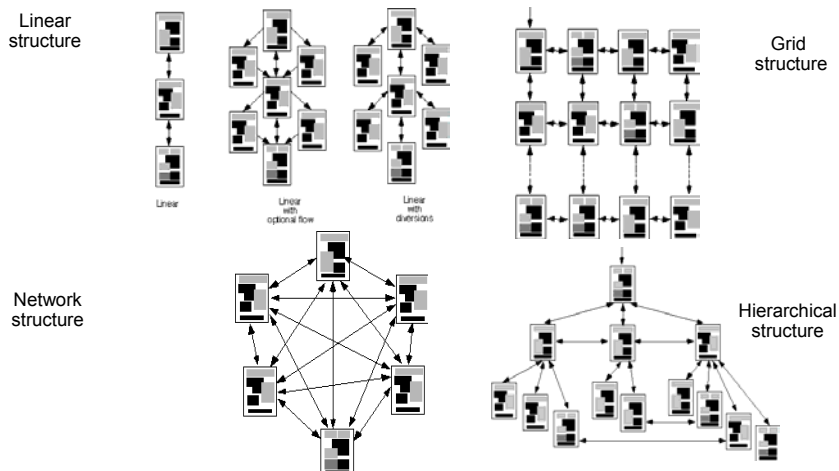
148

Architecture Design

- *Content architecture* focuses on the manner in which content objects (or composite objects such as Web pages) are structured for presentation and navigation.
 - » The term information architecture is also used to connote structures that lead to better organization, labeling, navigation, and searching of content objects.
- *WebApp architecture* addresses the manner in which the application is structured to manage user interaction, handle internal processing tasks, effect navigation, and present content.
- Architecture design is conducted in parallel with interface design, aesthetic design and content design.

149

Content Architecture



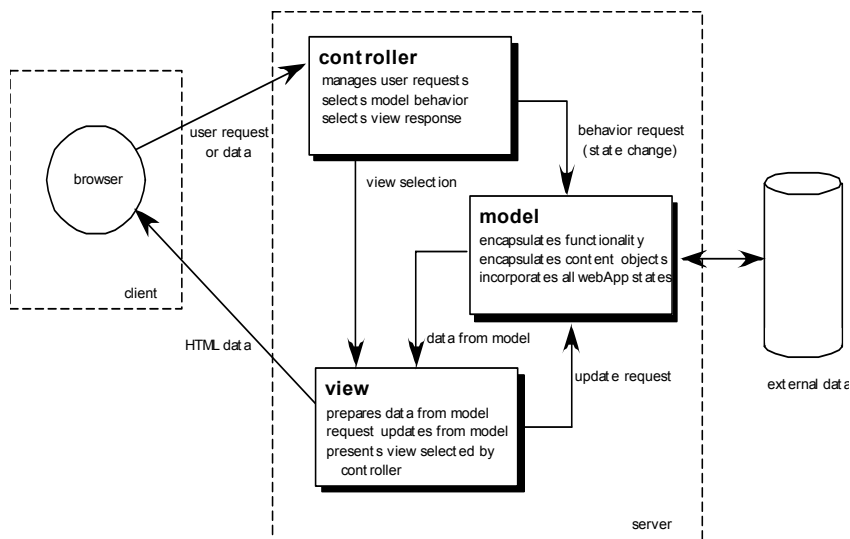
150

MVC Architecture

- The **model** contains all application specific content and processing logic, including
 - › all content objects
 - › access to external data/information sources,
 - › all processing functionality that are application specific
- The **view** contains all interface specific functions and enables
 - › the presentation of content and processing logic
 - › access to external data/information sources,
 - › all processing functionality required by the end-user.
- The **controller** manages access to the model and the view and coordinates the flow of data between them.

151

MVC Architecture



152

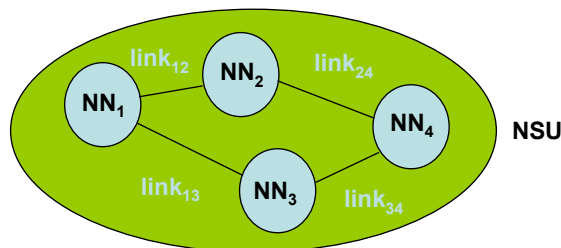
Navigation Design

- Begins with a consideration of the user hierarchy and related use-cases
 - » Each actor may use the WebApp somewhat differently and therefore have different navigation requirements
- As each user interacts with the WebApp, she encounters a series of *navigation semantic units* (NSUs)
 - » NSU—“a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements”

153

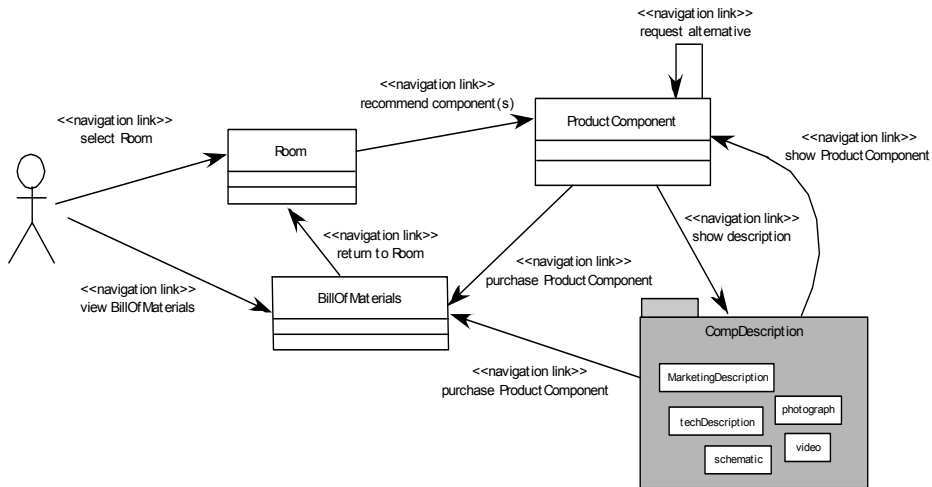
Navigation Semantic Units

- *Navigation semantic unit*
 - » *Ways of navigation (WoN)*—represents the best navigation way or path for users with certain profiles to achieve their desired goal or sub-goal. Composed of ...
 - *Navigation nodes (NN)* connected by *Navigation links*



154

Creating an NSU



155

Navigation Syntax

- **Individual navigation link**—text-based links, icons, buttons and switches, and graphical metaphors..
- **Horizontal navigation bar**—lists major content or functional categories in a bar containing appropriate links. In general, between 4 and 7 categories are listed.
- **Vertical navigation column**
 - » lists major content or functional categories
 - » lists virtually all major content objects within the WebApp.
- **Tabs**—a metaphor that is nothing more than a variation of the navigation bar or column, representing content or functional categories as tab sheets that are selected when a link is required.
- **Site maps**—provide an all-inclusive tab of contents for navigation to all content objects and functionality contained within the WebApp.

156





Component-Level Design

- WebApp components implement the following functionality
 - » perform localized processing to generate content and navigation capability in a dynamic fashion
 - » provide computation or data processing capability that are appropriate for the WebApp's business domain
 - » provide sophisticated database query and access
 - » establish data interfaces with external corporate systems.

157

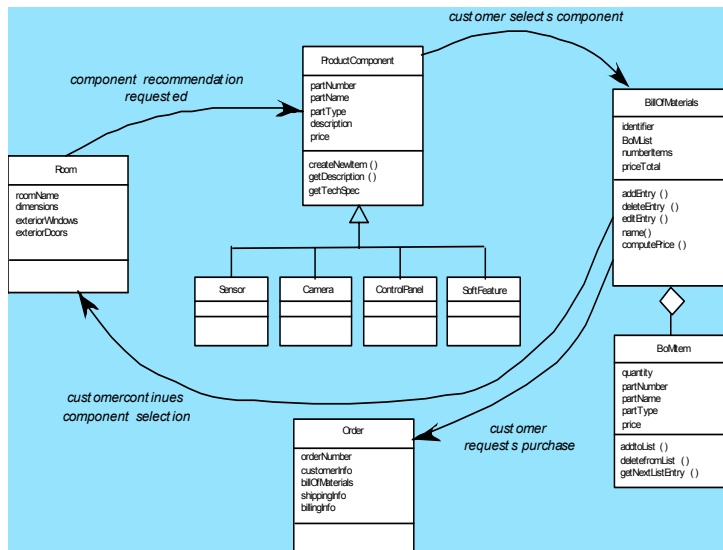
OOHDM

- *Object-Oriented Hypermedia Design Method (OOHDM)*

	 conceptual design	 navigational design	 abstract interface design	 implementation
work products	Classes, sub-systems, relationships, attributes	Nodes, links, access structures, navigational contexts, navigational transformations	Abstract interface objects, responses to external events, transformations	executable WebApp
design mechanisms	Classification, composition, aggregation, generalization, specialization	Mapping between conceptual and navigation objects	Mapping between navigation and perceptible objects	Resources provided by target environment
design concerns	Modeling semantics of the application domain	Takes into account user profile and task. Emphasis on cognitive aspects.	Modeling perceptible objects, implementing chosen metaphors. Describe interface for navigational objects	Correctness, Application performance, completeness

158

Conceptual Schema



159

Agenda

- 1 Introduction
- 2 Architectural Design
- 3 Component-Level Design
- 4 User Interface Design
- 5 Pattern-Based Design
- 6 Web Application Design
- ➔ 7 Summary and Conclusion

160



- All design work products must be traceable to software requirements and that all design work products must be reviewed for quality
- Software projects iterate through the analysis and design phases several times
- Pure separation of analysis and design may not always be possible or desirable
- There are many significant design concepts (abstraction, refinement, modularity, architecture, patterns, refactoring, functional independence, information hiding, and OO design concepts)
- Design changes are inevitable and that delaying component level design can reduce the impact of these changes



- The goal of the architectural model is to allow the software engineer to view and evaluate the system as a whole before moving to component design
- At the architecture level, data design is the process of creating a model of the information represented at a high level of abstraction (using the customer's view of data)
- At the component level, data design focuses on specific data structures required to realize the data objects to be manipulated by a component
- An architectural style is a transformation that is imposed on the design of an entire system



- Design classes in the problem domain are usually custom-designed, however, if an organization has encouraged design for reuse, there may be an existing component that fits the bill
- Design classes corresponding to the infrastructure domain can sometimes be often from existing class libraries
- A UML collaboration diagram provides an indication of message passing between components



- There are three principles of user interface design that should be followed to build software projects:
 - » The first is to place the user in control (which means have the computer interface support the user's understanding of a task and do not force the user to follow the computer's way of doing things)
 - » The second (reduce the user's memory load) means place all necessary information in the screen at the same time
 - » The third is consistency of form and behavior. It is sometimes good to bring in commercial software and try to see how well the interface designers seem to have followed these guidelines

Course Assignments



- Individual Assignments
 - Reports based on case studies / class presentations
- Project-Related Assignments
 - All assignments (other than the individual assessments) will correspond to milestones in the team project.
 - As the course progresses, students will be applying various methodologies to a project of their choice. The project and related software system should relate to a real-world scenario chosen by each team. The project will consist of inter-related deliverables which are due on a (bi-) weekly basis.
 - There will be only one submission per team per deliverable and all teams must demonstrate their projects to the course instructor.
 - A sample project description and additional details will be available under handouts on the course Web site

165

Team Project



- Project Logistics
 - Teams will pick their own projects, within certain constraints: for instance, all projects should involve multiple distributed subsystems (e.g., web-based electronic services projects including client, application server, and database tiers). Students will need to come up to speed on whatever programming languages and/or software technologies they choose for their projects - which will not necessarily be covered in class.
 - Students will be required to form themselves into "pairs" of exactly two (2) members each; if there is an odd number of students in the class, then one (1) team of three (3) members will be permitted. There may not be any "pairs" of only one member! The instructor and TA(s) will then assist the pairs in forming "teams", ideally each consisting of two (2) "pairs", possibly three (3) pairs if necessary due to enrollment, but students are encouraged to form their own 2-pair teams in advance. If some students drop the course, any remaining pair or team members may be arbitrarily reassigned to other pairs/teams at the discretion of the instructor (but are strongly encouraged to reform pairs/teams on their own). Students will develop and test their project code together with the other member of their programming pair.

166

Team Project Approach - Overall



- Document Transformation methodology driven approach
 - Strategy Alignment Elicitation
 - Equivalent to strategic planning
 - i.e., planning at the level of a project set
 - Strategy Alignment Execution
 - Equivalent to project planning + SDLC
 - i.e., planning a the level of individual projects + project implementation
- Build a methodology Wiki & partially implement the enablers
- Apply transformation methodology approach to a sample problem domain for which a business solution must be found
- Final product is a wiki/report that focuses on
 - Methodology / methodology implementation / sample business-driven problem solution

167

Team Project Approach – Initial Step



- Document sample problem domain and business-driven problem of interest
 - Problem description
 - High-level specification details
 - High-level implementation details
 - Proposed high-level timeline

168



- **Project Logistics**

- Teams will pick their own projects, within certain constraints: for instance, all projects should involve multiple distributed subsystems (e.g., web-based electronic services projects including client, application server, and database tiers). Students will need to come up to speed on whatever programming languages and/or software technologies they choose for their projects - which will not necessarily be covered in class.
- Students will be required to form themselves into "pairs" of exactly two (2) members each; if there is an odd number of students in the class, then one (1) team of three (3) members will be permitted. There may not be any "pairs" of only one member! The instructor and TA(s) will then assist the pairs in forming "teams", ideally each consisting of two (2) "pairs", possibly three (3) pairs if necessary due to enrollment, but students are encouraged to form their own 2-pair teams in advance. If some students drop the course, any remaining pair or team members may be arbitrarily reassigned to other pairs/teams at the discretion of the instructor (but are strongly encouraged to reform pairs/teams on their own). Students will develop and test their project code together with the other member of their programming pair.



- After teams formed, 1/2 week to Project Concept
- 1/2 week to Revised Project Concept
- 2 to 3 iterations
- For each iteration:
 - » 1/2 week to plan
 - » 1 week to iteration report and demo

Sample Project Methodology

Very eXtreme Programming (VXP) - (continued)



- Requirements: Your project focuses on two application services
- Planning: User stories and work breakdown
- Doing: Pair programming, write test cases before coding, automate testing
- Demoing: 5 minute presentation plus 15 minute demo
- Reporting: What got done, what didn't, what tests show
- 1st iteration: Any
- 2nd iteration: Use some component model framework
- 3rd iteration: Refactoring, do it right this time

171

Revised Project Concept (Tips)



1. Cover page (max 1 page)
2. Basic concept (max 3 pages): Briefly describe the system your team proposes to build. Write this description in the form of either user stories or use cases (your choice). Illustrations do not count towards page limits.
3. Controversies (max 1 page)

172

First Iteration Plan (Tips)



- Requirements (max 2 pages):
- Select user stories or use cases to implement in your first iteration, to produce a demo by the last week of class
- Assign priorities and points to each unit - A point should correspond to the amount of work you expect one pair to be able to accomplish within one week
- You may optionally include additional medium priority points to do “if you have time”
- It is acceptable to include fewer, more or different use cases or user stories than actually appeared in your Revised Project Concept

173

First Iteration Plan (Tips)



- Work Breakdown (max 3 pages):
- Refine as *engineering tasks* and assign to pairs
- Describe specifically what will need to be coded in order to complete each task
- Also describe what unit and integration tests will be implemented and performed
- You may need additional engineering tasks that do not match one-to-one with your user stories/use cases
- Map out a *schedule* for the next weeks
- Be realistic – demo has to be shown before the end of the semester

174

2nd Iteration Plan (Tips): Requirements



- Max 3 pages
- Redesign/reengineer your system to use a component framework (e.g., COM+, EJB, CCM, .NET or Web Services)
- Select the user stories to include in the new system
 - » Could be identical to those completed for your 1st Iteration
 - » Could be brand new (but explain how they fit)
- Aim to maintain project velocity from 1st iteration
- Consider what will require new coding vs. major rework vs. minor rework vs. can be reused “as is”

175

2nd Iteration Plan (Tips): Breakdown



- Max 4 pages
- Define engineering tasks, again try to maintain project velocity
- Describe new unit and integration testing
- Describe regression testing
 - » Can you reuse tests from 1st iteration?
 - » If not, how will you know you didn't break something that previously worked?
- 2nd iteration report and demo to be presented before the end of the semester

176



- Max 2 pages
- For each engineering task from your 2nd Iteration Plan, indicate whether it succeeded, partially succeeded (and to what extent), failed (and how so?), or was not attempted
- Estimate how many user story points were actually completed (these might be fractional)
- Discuss specifically your success, or lack thereof, in porting to or reengineering for your chosen component model framework(s)




- Max 3 pages
- Describe the general strategy you followed for unit testing, integration testing and regression testing
- Were you able to reuse unit and/or integration tests, with little or no change, from your 1st Iteration as regression tests?
- What was most difficult to test?
- Did using a component model framework help or hinder your testing?



- All Iterations Due
- Presentation slides (optional)



- Readings
 - Slides and Handouts posted on the course web site
 -  ▪ Textbook: Part Two-Chapters 6-8
- Individual Assignment (due)
 - See Session 5 Handout: "Assignment #2"
- Individual Assignment (assigned)
 - See Session 8 Handout: "Assignment #3"
- Team Project #1 (ongoing)
 - Team Project proposal (format TBD in class)
 - See Session 2 Handout: "Team Project Specification" (Part 1)
- Team Exercise #1 (ongoing)
 - Presentation topic proposal (format TBD in class)
- Project Frameworks Setup (ongoing)
 - As per reference provided on the course Web site

Any Questions?



181

Next Session: From Analysis and Design to Software Architecture (Part II)

182