SOFTWARE ENGINEERING

CHAPTER-12
PATTERN-BASED DESIGN

Software Engineering: A Practitioner's Approach, 7th edition

Originated by Roger S. Pressman

Software School, Fudan University
Spring Semester, 2016

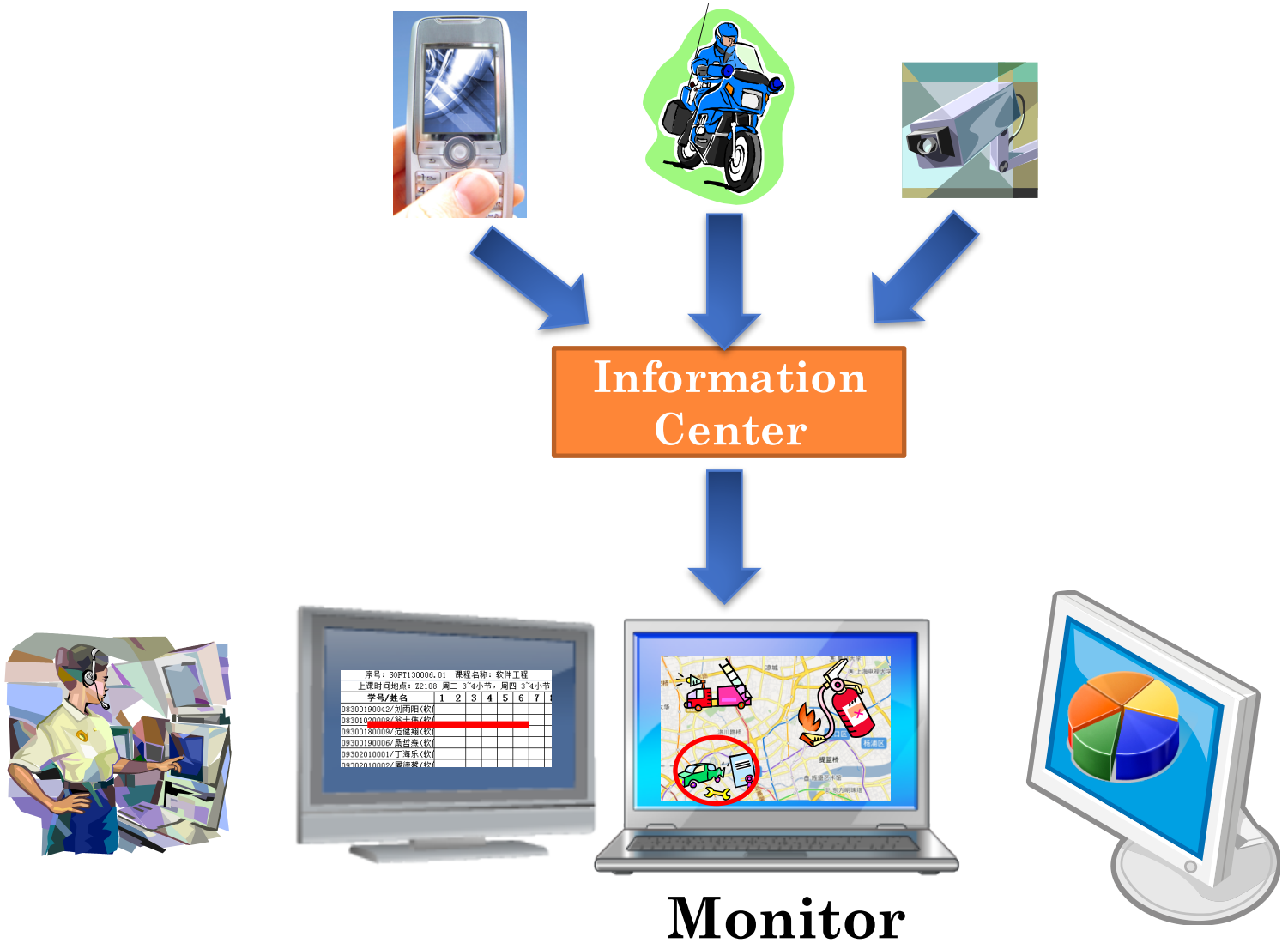
What to Reuse

- Component
- Pattern
- Framework

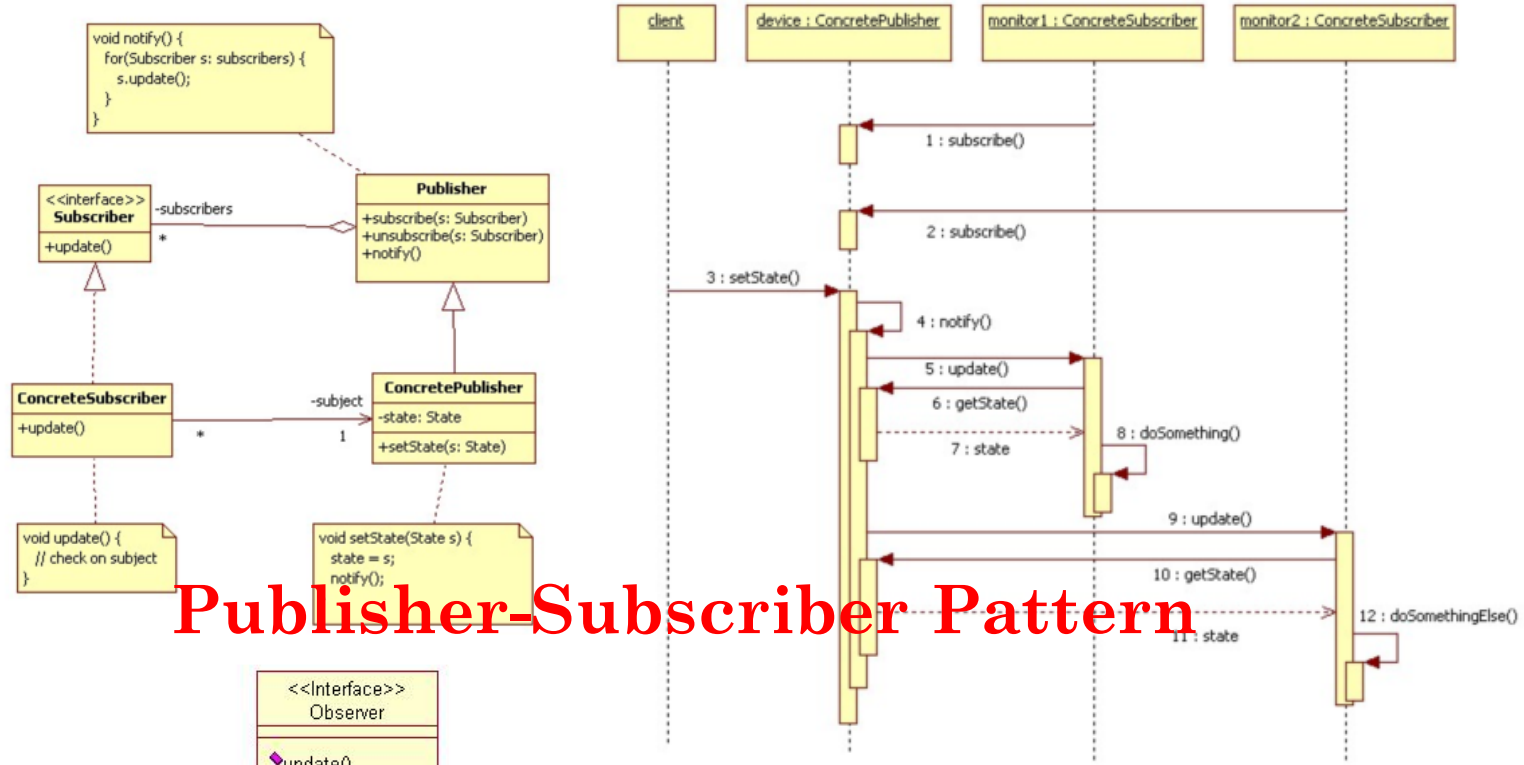
Design Patterns

- Each of us has encountered a design problem and silently thought: *I wonder if anyone has developed a solution 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

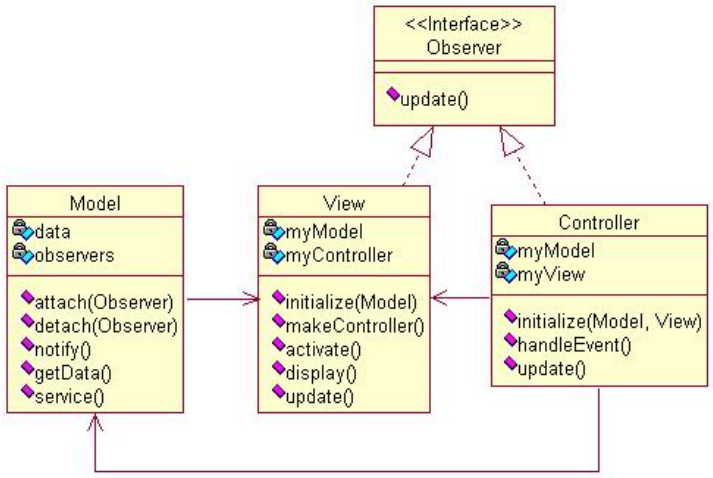
Consider a Design Problem



Related Patterns



Publisher-Subscriber Pattern



MVC Pattern

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."

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.

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

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

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.

Design Patterns

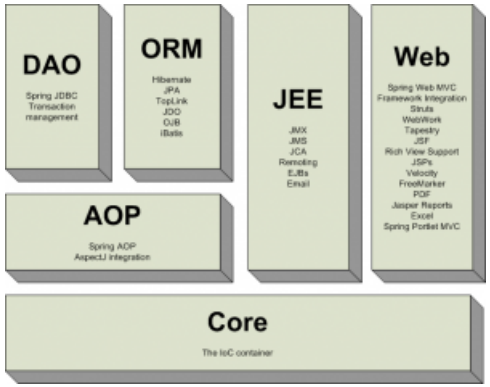
- ***Creational patterns*** focus on the "creation, composition, and representation of objects, e.g.,
  - **Abstract factory pattern:** centralize decision of what factory to instantiate
  - **Factory method pattern:** centralize creation of an object of a specific type choosing one of several implementations
- ***Structural patterns*** focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure, e.g.,
  - **Adaptor 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

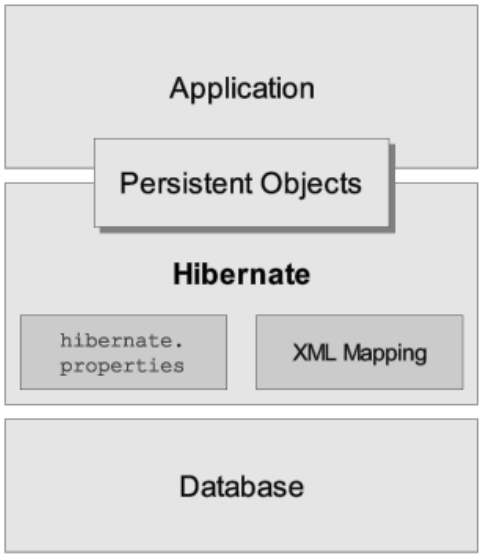
FRAMEWORKS

- 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.

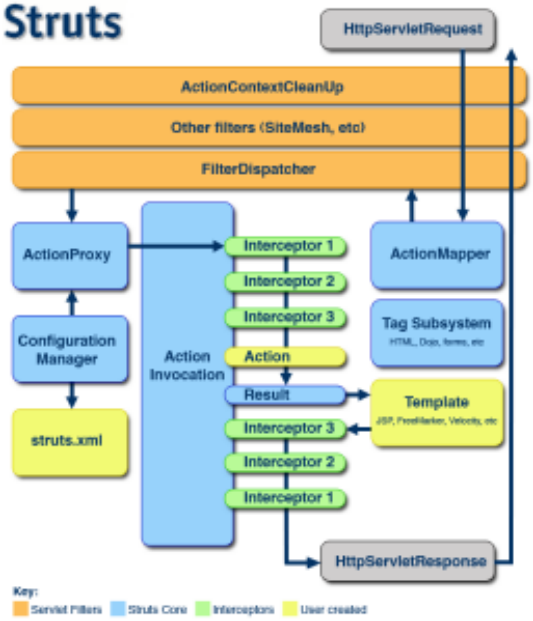
FRAMEWORKS

Spring

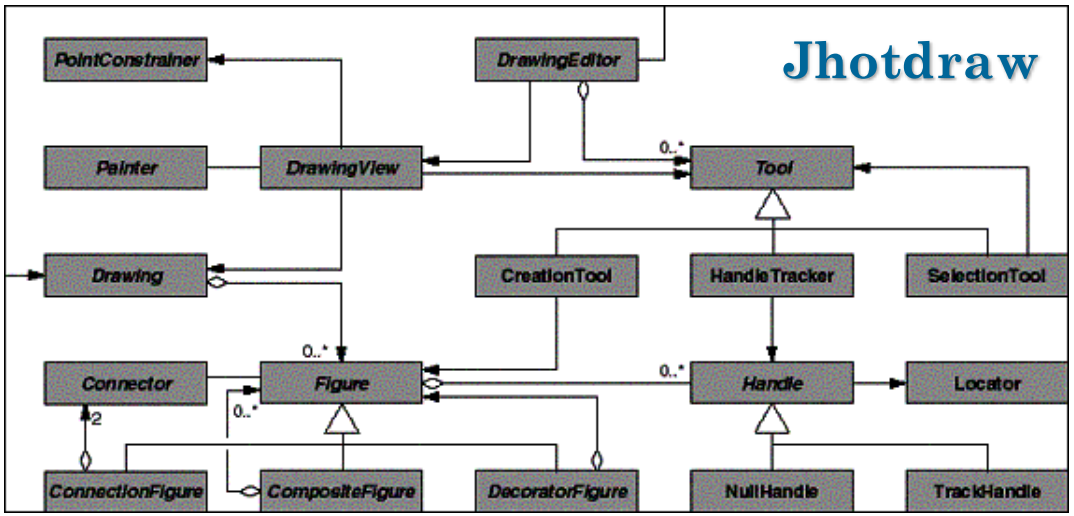




Struts



Jhotdraw



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

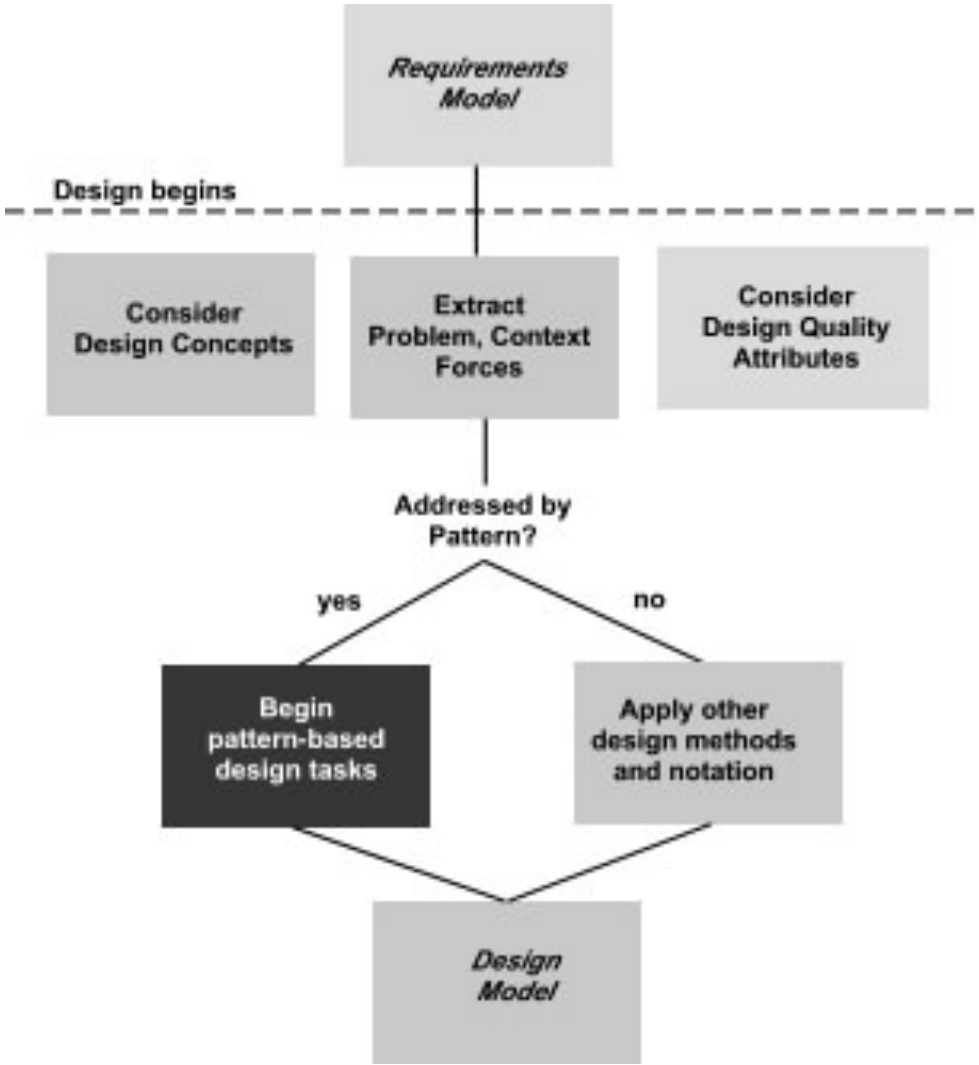
Pattern Languages

- A *pattern language* encompasses a collection of patterns
  - each described using a standardized template (Section 12.1.3) 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.

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 …

Pattern-Based Design



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.

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.

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.

Pattern Organizing Table

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

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.

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.

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 near Section 12.3

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?*

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?*

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

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.

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.

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:

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 books), navigation items (e.g. how might you format links?) or functional elements (e.g. specific algorithms).

End of Chapter 12