COMP 354: Introduction to Software Engineering

Pattern-Based Design

Based on Chapter 14 of the textbook

Design Patterns

- Design pattern can be thought of as a three-part rule which expresses a relation between a certain context, a problem, and a solution.
- Context allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment.
- Requirements, including limitations and constraints, acts as a system of forces that influences how:
  - The problem can be interpreted within its context.
  - The solution can be effectively applied.

Effective Design Pattern

- Solves a problem: Patterns capture solutions, not just abstract principles or strategies.
- Proven concept: Patterns capture solutions with proven track records, not theories or speculation.
- Solution isn't obvious: The best patterns generate a solution to a problem indirectly--a necessary approach for the most difficult problems of design.
- Describes a relationship: Patterns don't just describe modules, but describe deeper system structures and mechanisms.
- Elegant in its approach and utility: describes simple solutions to specific problems; the best patterns explicitly appeal to aesthetics and usefulness.

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 (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.

Kinds of Patterns

- 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 incorporate many of other patterns categories.
- Mobile patterns describe solutions to problems commonly encountered when developing solutions for mobile platforms.

Kinds of Patterns

- Creational patterns focus on “creation, composition, and representation of objects”
  - Abstract factory pattern: centralize decision of what factory to instantiate.
  - Builder pattern: separates the construction of a complex object from its representation.
- Structural patterns focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure:
  - Adapter pattern: 'adapts' one interface for a class into one that a client expects.
  - Container pattern: create objects for the sole purpose of holding other objects and managing them.

Kinds of Patterns

- Behavioral patterns address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects:
  - 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.
  - You can select a reusable architecture that provides the generic structure and behavior for a family of software abstractions along with their context which specifies use within a given domain.
- A framework is not an architectural pattern, but rather a skeleton with a collection of “plug points” (hooks or slots) that enable it to be adapted to a specific problem domain.
  - Plug points enable you to integrate problem specific classes or functionality within the skeleton.
  - Collection of cooperating classes.

Patterns, Components and Frameworks

- Design patterns are abstract solutions that have been observed to work for concrete problems.
  - They define the conditions for the problem, the solution used, and any consequences that might result.
  - They lead to efficiency and reuse.
- Components are concrete implementations of design patterns.
  - They provide code you can copy and paste directly into a project.
  - Ideally they'll be flexible and offer a mechanism for modification.
  - There can be multiple components for a single design pattern.
- Frameworks combine components and package them together.
  - They provide APIs and tools for using the framework and tend to be specialized in nature.

Pattern Description

- 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

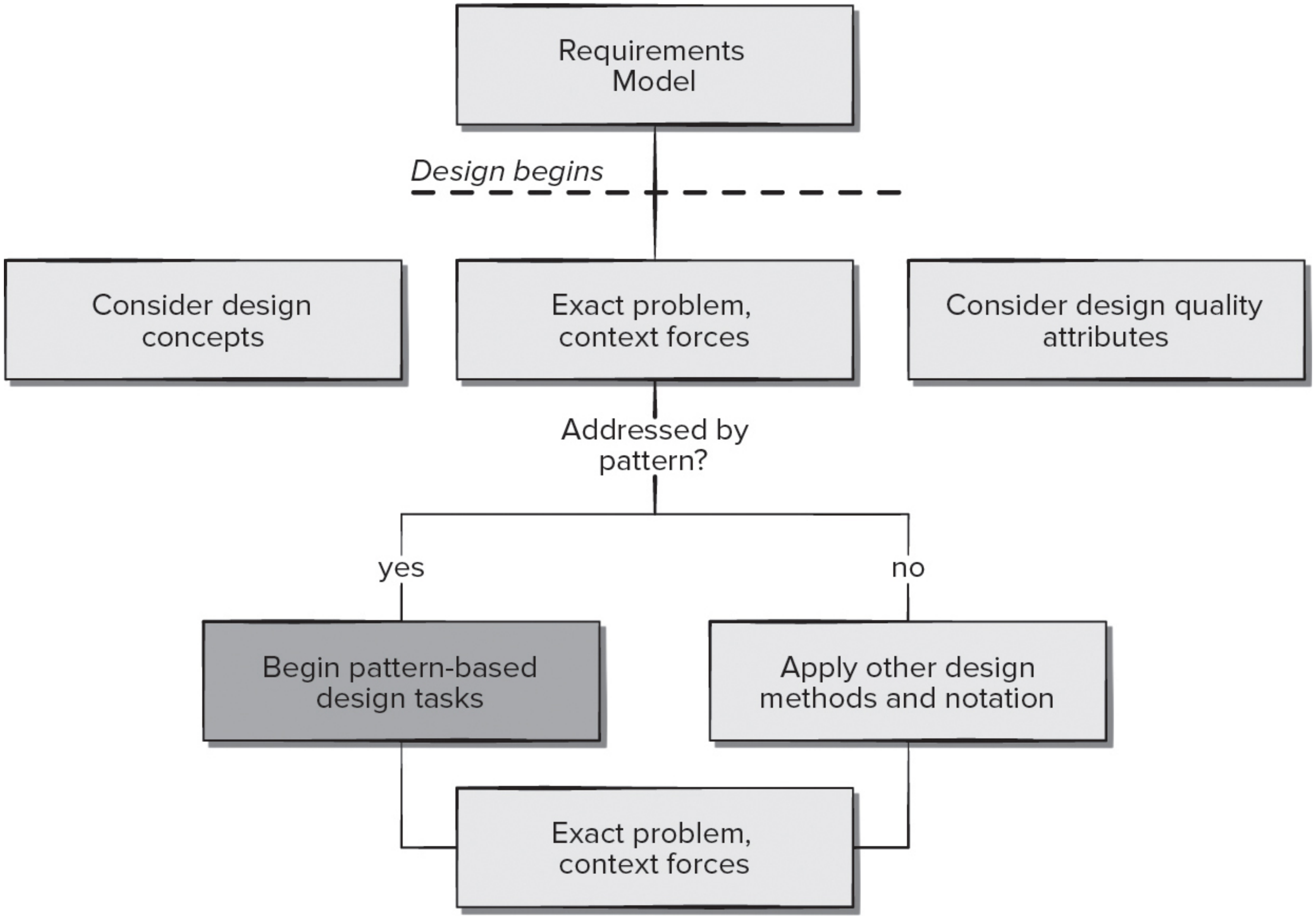
Pattern Description

- 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-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.
- If you discover you are faced with a problem, context, and system of forces that have solved before then use that solution.
- If it is a new problem then use the methods and modeling tools available for architectural, component-level, and interface design to create a new solution (pattern).

Pattern-Based Design in Context


Requirements Model
Design begins
Consider design concepts
Exact problem, context forces
Consider design quality attributes
Addressed by pattern?
yes
no
Begin pattern-based design tasks
Apply other design methods and notation
Exact problem, context forces

Thinking in Patterns

1. Be sure you understand big picture (requirements model) - the context in which the software to be built resides.
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, look for patterns at lower levels of abstraction that contribute to design solution.
5. Repeat steps 1 to 4 until complete design is fleshed out.
6. Refine the design by adapting each pattern to the specifics of the software you’re trying to build.

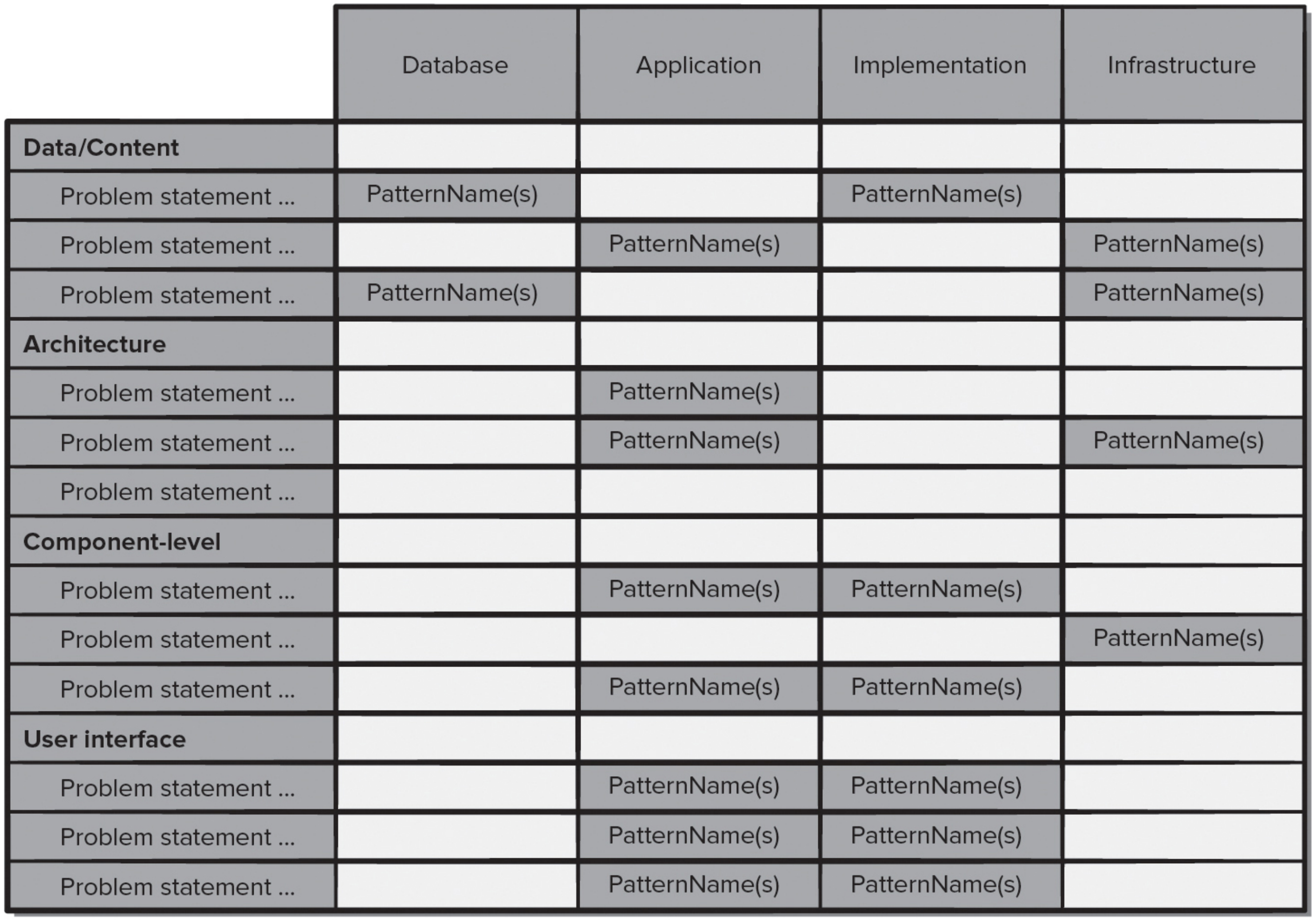
Design Tasks

1. Examine the requirements model and develop a problem hierarchy.
2. Determine if a reliable pattern language has been developed for the problem domain.
3. Beginning with a broad problem, determine whether one or more architectural patterns are available for it.
4. Using the collaborations provided for the architectural pattern, examine subsystem or component level problems, and search for patterns to address them.

Design Tasks

5. Repeat steps 2 through 5 until all broad problems have been addressed.
6. If user interface design problems have been isolated, search the user interface design pattern repositories for appropriate patterns.
7. 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.
8. 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)	

Source: Adapted from Microsoft, "Prescriptive Architecture: Integration and Patterns," MSDN, May 2004.

Common 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.
- Kitchen pattern and patterns it collaborates with address problems associated with storage and preparation of food, tools required to accomplish these tasks, and rules for placement of these tools relative to workflow in the room.
- 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, but every design can be conceived within the context and system forces of the 'solution' suggested by the Kitchen pattern.

Component-Level Design 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 Design 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 (for example: a security sensor or camera) is used for informational purposes by the consumer.
- However, other information that is related to the specification (for example: 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.

Search Patterns

- HelpWizard. Users need help on a certain topic related to the website or when they need to find a specific page within the site.
- SearchArea. Users must find a page.
- SearchTips. Users need to know how to control the search engine.
- SearchResults. Users must process a list of search results.
- SearchBox. Users must find an item or specific information.

Anti-Patterns

- Anti-patterns describe commonly used solutions to design problems that usually have negative effects on software quality.
- Anti-patterns can provide tools to help developers recognize when these problems exist and may provide detailed plans for reversing the underlying problem causes and implementing better solutions to these problems.
- Anti-patterns can provide guidance to developers looking for ways to refactor software to improve its quality.
- Anti-patterns can be used by technical reviewers to uncover areas of concern.

Selected Anti-Patterns

- Blob. Single class with large number of attributes, operators, or both.
- Stovepipe system. A barely maintainable assemblage of ill-related components.
- Boat anchor. Retaining a part of system that no longer has any use.
- Spaghetti code. Program whose structure is barely comprehensible, especially because of misuse of code structures.
- Copy and paste programming. Copying existing code several times rather than creating generic solutions.
- Silver bullet. Assume that a favorite technical solution will always solve a larger process or problem.
- Programming by permutation. Trying to approach a solution by successively modifying the code to see if it works.

User Interface (UI) Patterns

- Whole UI. Provide design guidance for top-level structure and navigation throughout the entire interface.
- Top-level Navigation. Provides a top-level menu, often coupled with a logo or identifying graphic, that enables direct navigation to any of the system’s major functions.
- Page layout. Address the general organization of pages (for Websites) or distinct screen displays (for interactive applications).
- E-commerce. Specific to Web sites, these patterns implement recurring elements of e-commerce applications.
- Shopping cart. Provides a list of items selected for purchase.

Mobility Design Patterns

- Check-in screens. How do I check in from a specific location, make a comment, and share comments with friends and followers on a social network?
- Maps. How do I display a map within the context of an app that addresses some other subject?
- Popovers. How do I represent a message or information that arises in real time or as the consequence of a user action?
- Sign-up flows. How do I provide a simple way to sign in or register for information or functionality?
- Custom tab navigation. How do I represent a variety of different content objects in a manner that enables user to select one?
- Invitations. How do I inform the user that he must participate in some action or dialog?