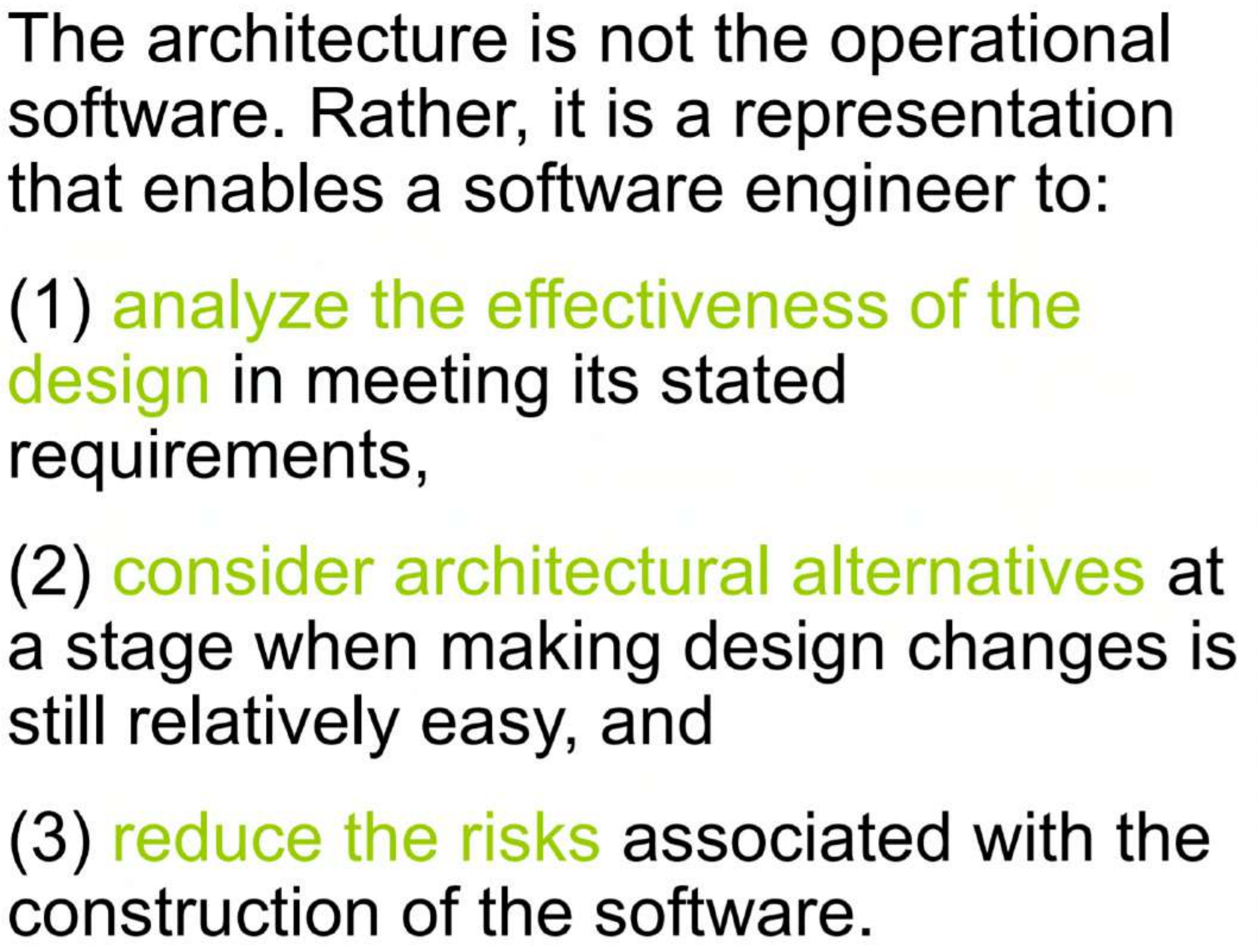
Software Systems Architecture

Introduction

Poștaru Andrei

What is 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.

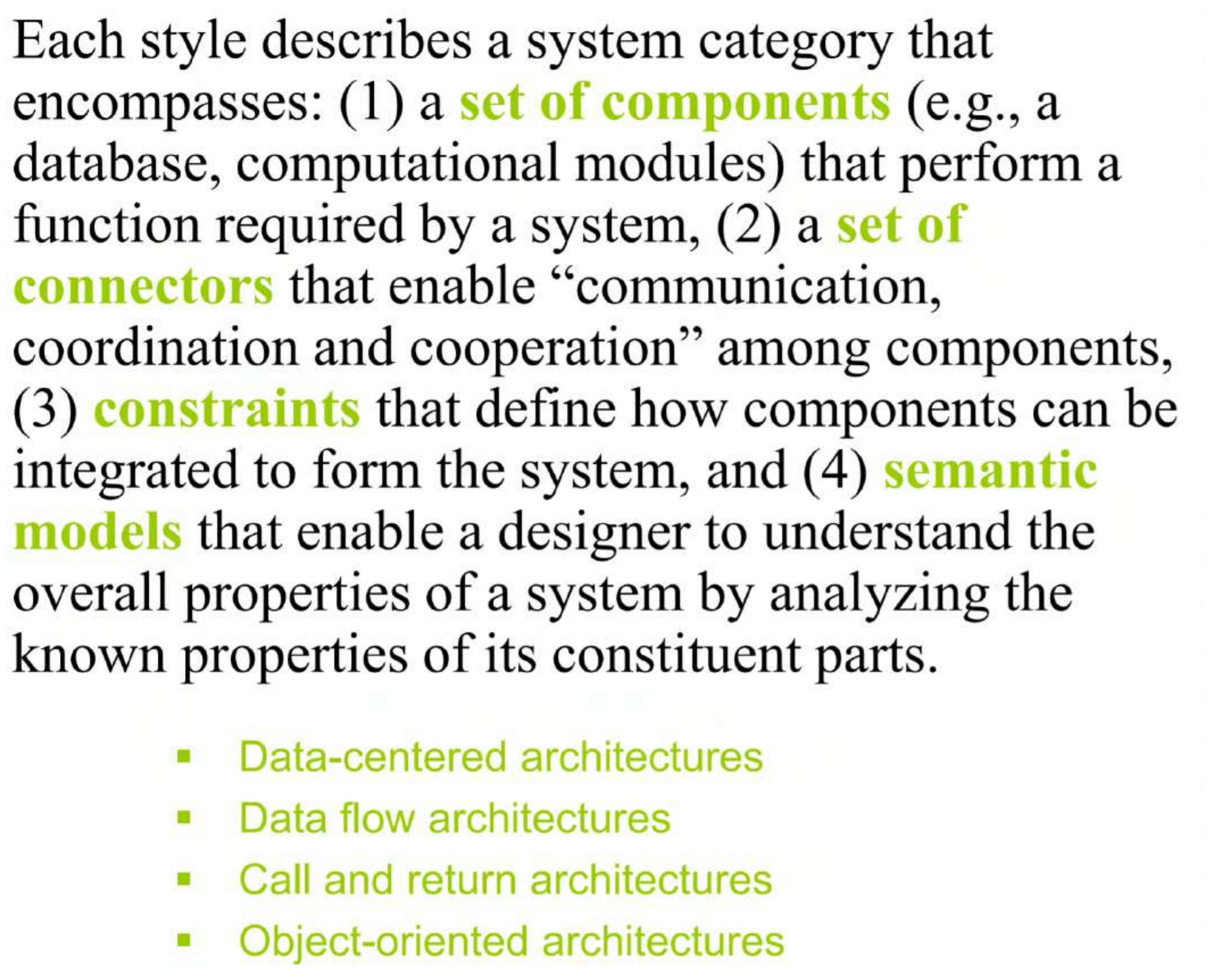
Why is Architecture Important?

- 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".

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.

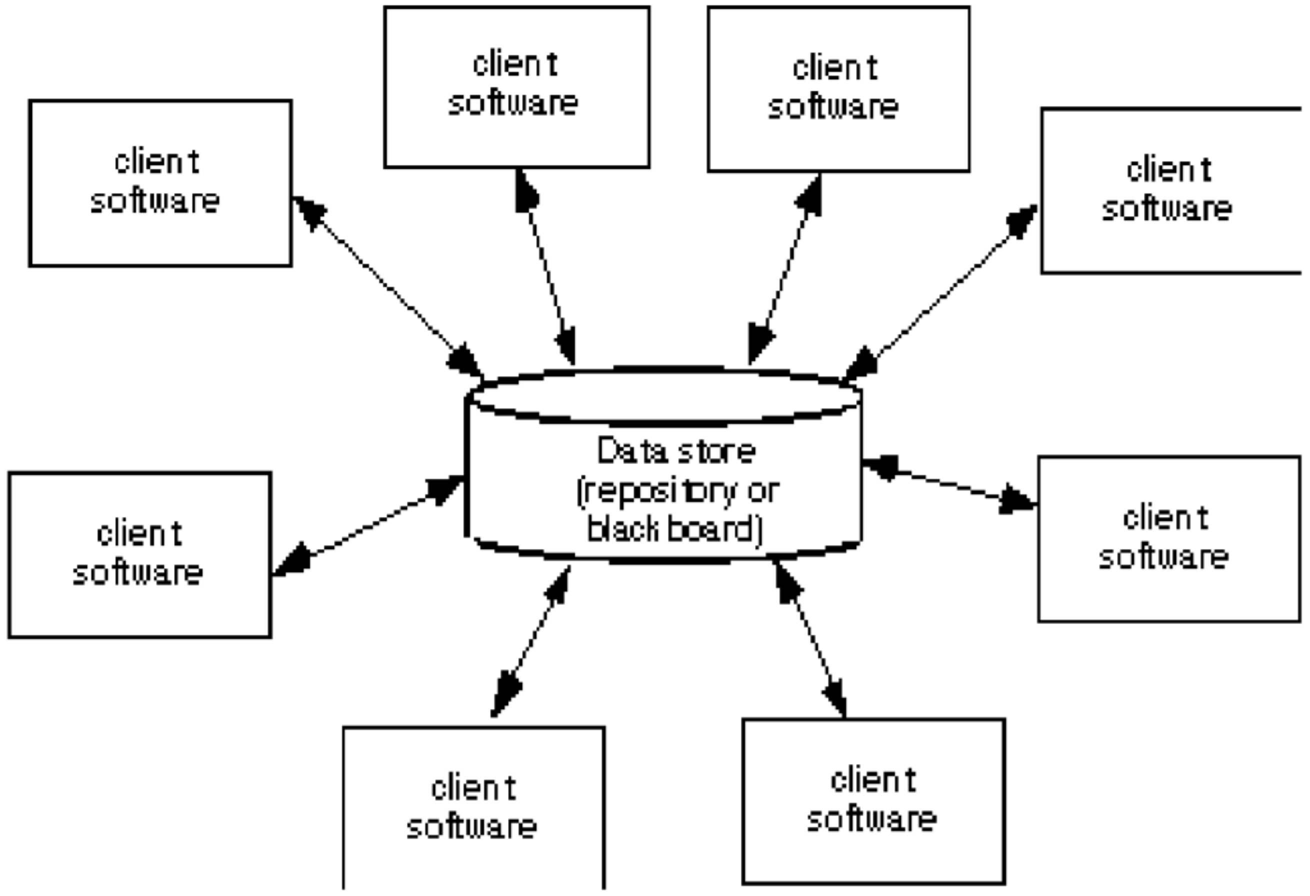
Architectural Styles



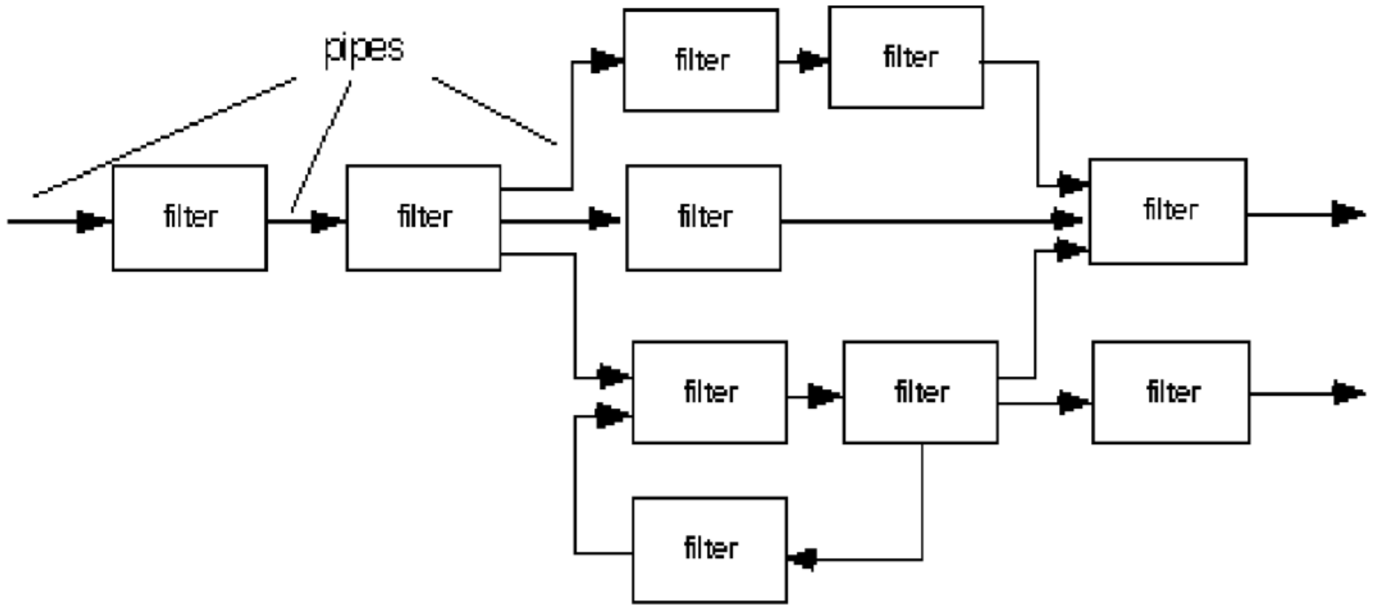
Each style describes a system category that encompasses: (1) a **set of components** (e.g., a database, computational modules) that perform a function required by a system, (2) a **set of connectors** that enable "communication, coordination and cooperation" among components, (3) **constraints** that 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

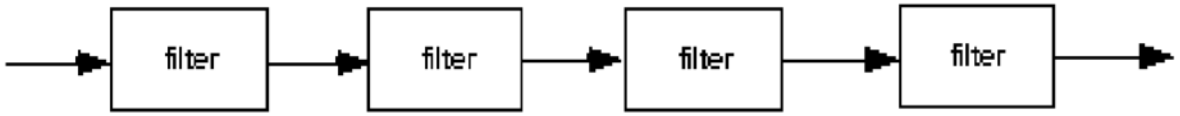
Data-Centered Architecture



Data Flow Architecture

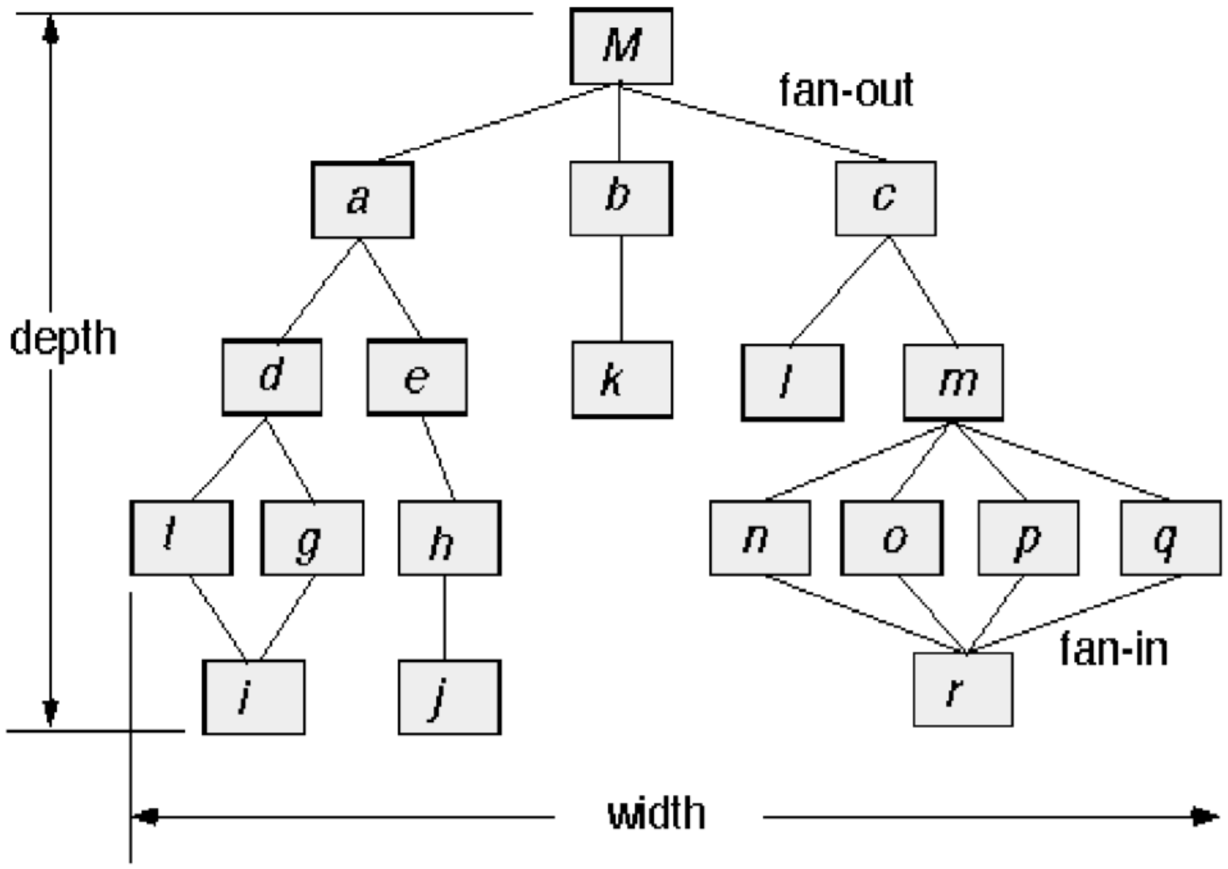


(a) pipes and filters

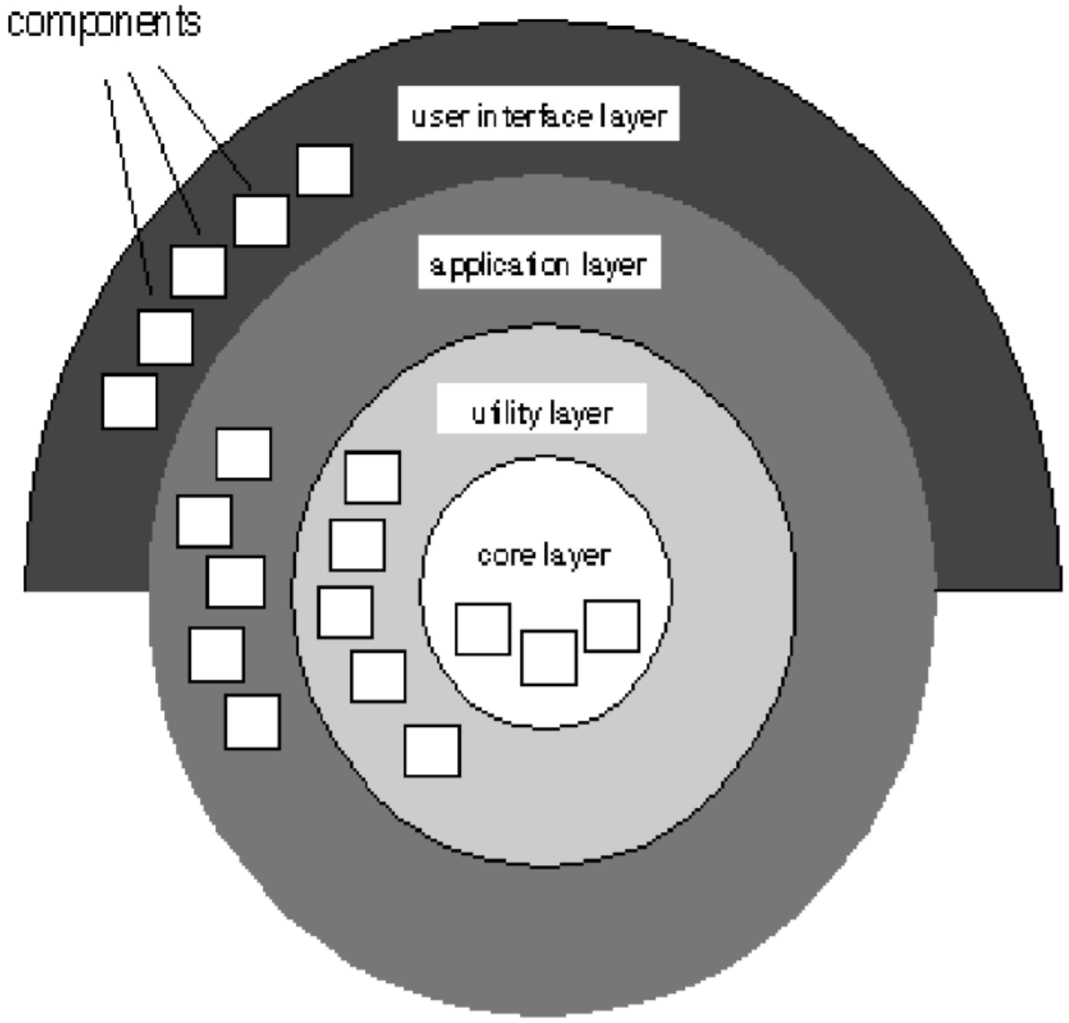


(b) batch sequential

Call and Return Architecture



Layered Architecture



Patterns…

- « Patterns help you build on the collective experience of skilled software engineers. »
- « They capture existing, well-proven experience in software development and help to promote good design practice »
- « Every pattern deals with a specific, recurring problem in the design or implementation of a software system »
- « Patterns can be used to construct software architectures with specific properties… »

Becoming a Software Designer Master

- First learn the rules
  - e.g., the algorithms, data structures and languages of software
- Then learn the principles
  - e.g., structured programming, modular programming, object oriented programming, generic programming, etc.
- However, to truly master software design, one must study the designs of other masters
  - These designs contain patterns must be understood, memorized, and applied repeatedly
- There are hundreds of these patterns

Software Architecture

- A software architecture is a description of the subsystems and components of a software system and the relationships between them.
- Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system.
- The software system is an artifact. It is the result of the software design activity.

Component

- A component is an encapsulated part of a software system. A component has an interface.
- Components serve as the building blocks for the structure of a system.
- At a programming-language level, components may be represented as modules, classes, objects or as a set of related functions.

Subsystems

- A subsystem is a set of collaborating components performing a given task. A subsystem is considered a separate entity within a software architecture.
- It performs its designated task by interacting with other subsystems and components…

Architectural Patterns

- An architectural Pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, their responsibilities, and includes rules and guidelines for organizing the relationships between them.

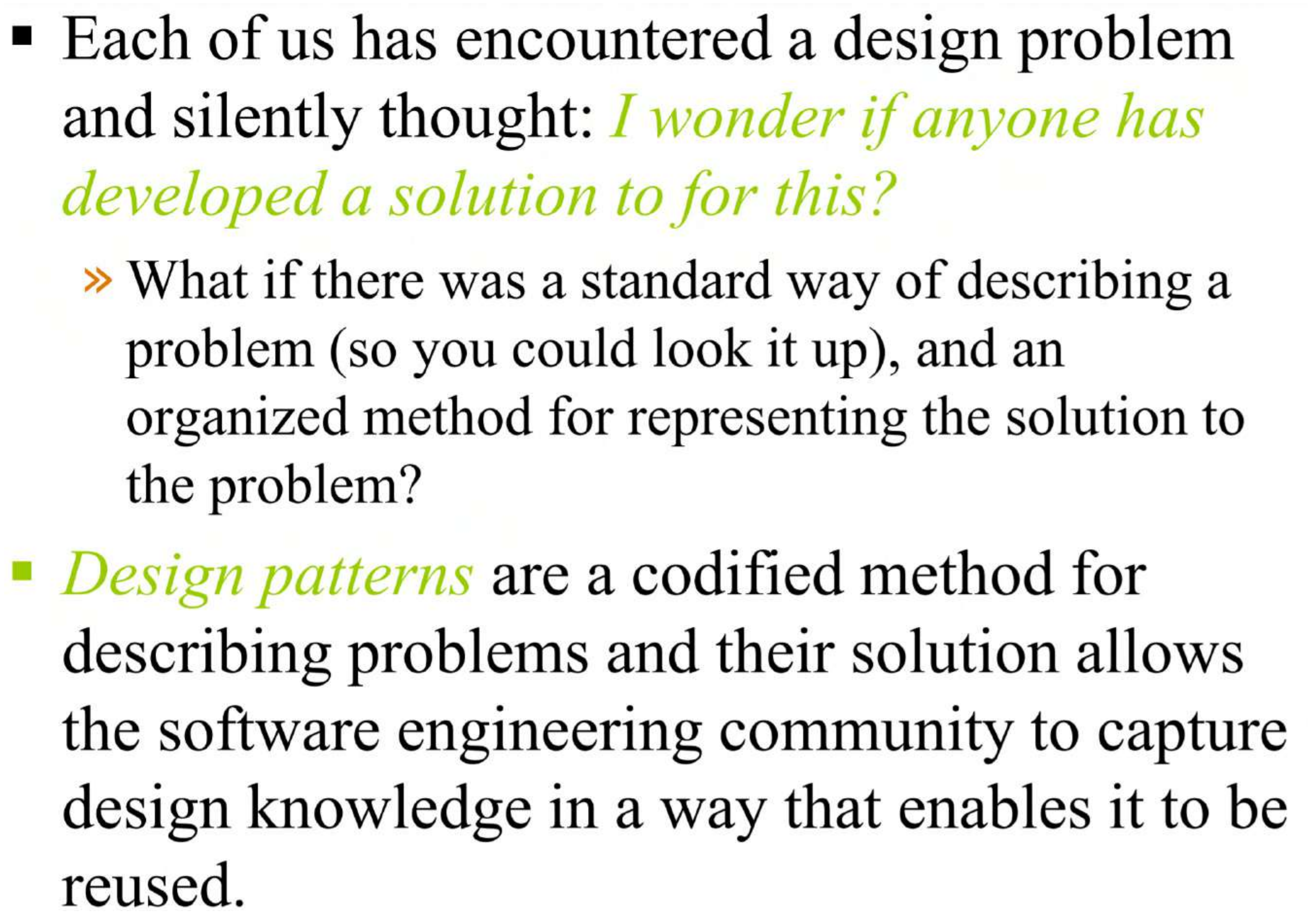
Design patterns

- A design pattern provides a scheme for refining the subsystems or components of a software system, or the relation ships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.

Framework

- A framework is a partially complete software (sub-) system that is intended to be instantiated. It defines the architecture for a family of (sub-) systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made.

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.

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.”

Effective Patterns

- *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.

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.

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

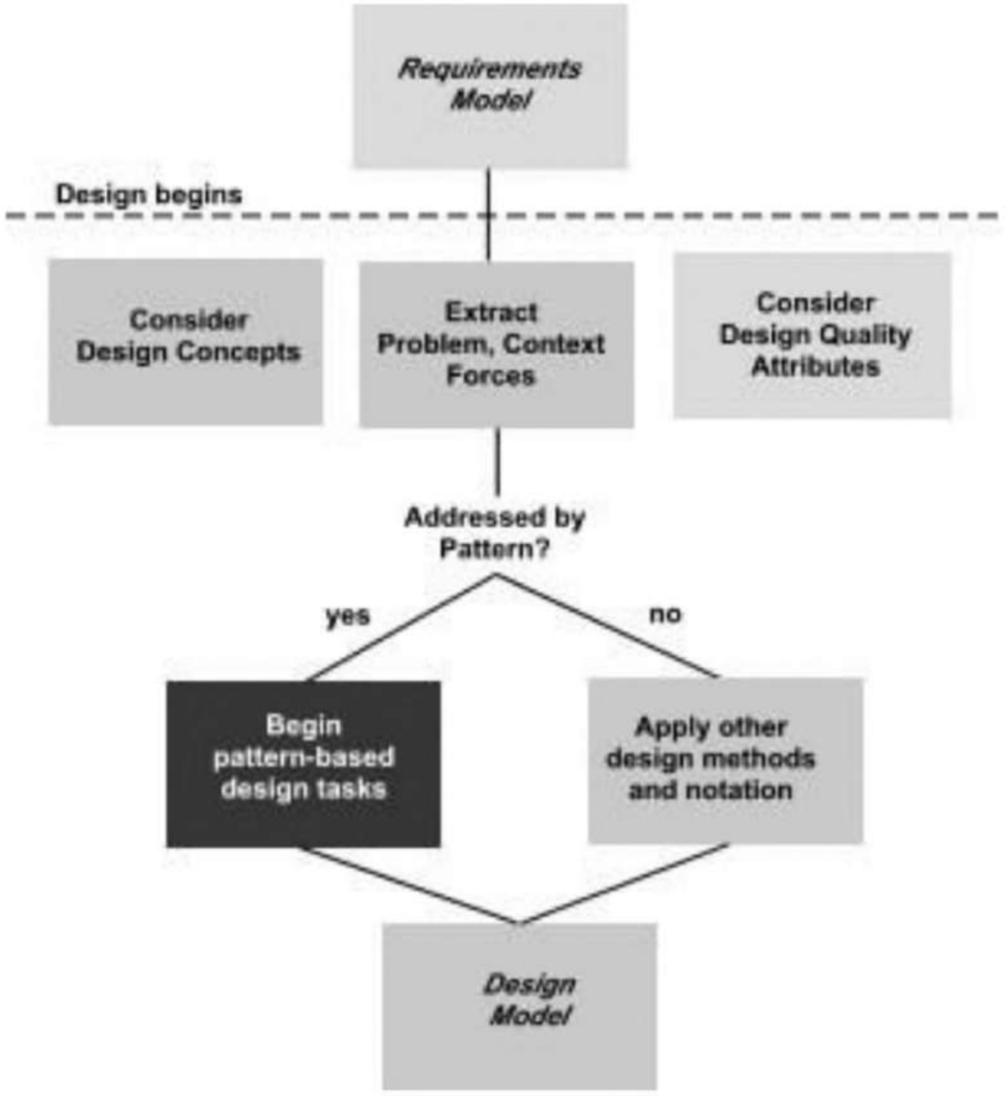
Frameworks

- 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.

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.

Pattern-Based Design


Requirements Model
Design begins
Consider Design Concepts
Extract Problem, Context Forces
Consider Design Quality Attributes
Addressed by Pattern?
yes
no
Begin pattern-based design tasks
Apply other design methods and notation
Design Model

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.

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.

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).