Fundamental Concepts

- Abstraction—data, procedure, control
- Architecture—the overall structure of the software
- Patterns—”conveys the essence” of a proven design solution
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Information Hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- Refinement—elaboration of detail for all abstractions
- Aspects—a mechanism for understanding how global requirements affect design
- Refactoring—a reorganization technique that simplifies the design
- OO design concepts—Appendix II
- Design Classes—provide design detail that will enable analysis classes to be implemented

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

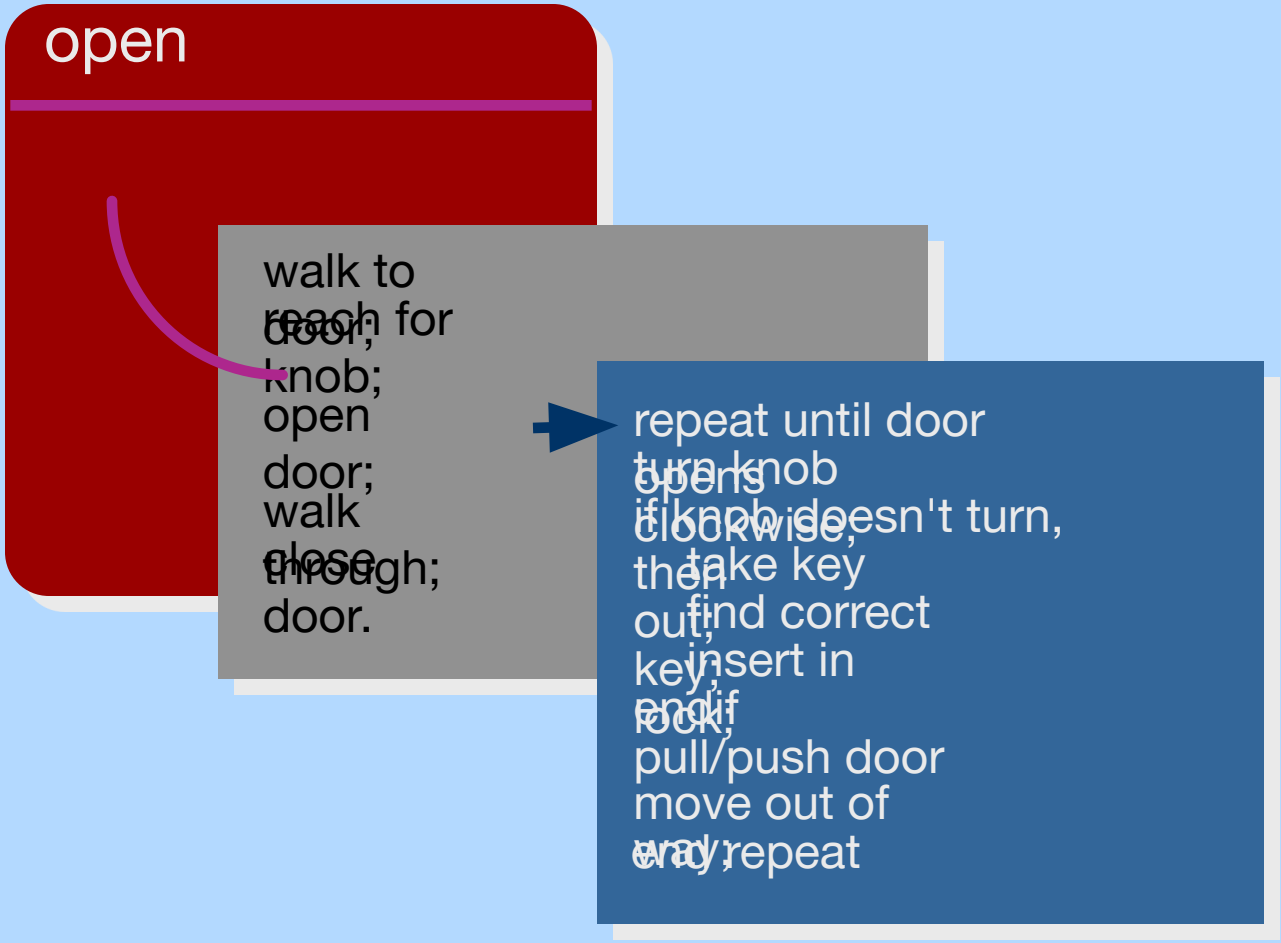
Refinement

Abstraction and refinement are complimentary concepts

- **Refinement means going into the details of the system via idea refinement**
- **Refine the requirement by decomposing it**
- **until it reaches to the state where it clearly elaborates the system.**
- **Abstraction shows the procedures and data whereas the refinement shows the implementation and low level details**

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Stepwise Refinement



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Architecture

- **Shows the overall structure of the software and the way in which the Structure provides conceptual integrity for a system.**
- **Architecture design = structure of data + program components needed for system**
- **Just a blue print of overall system, not a operational component**
- **Show dataflow, control flow, dependencies.**
- Conceptual integrity is the property of a system to work on a consistent set of concepts, such as entities (user, groups, posts) or operations (transfer money to an account or convert it in different currencies). If a railway system has three different websites for selling tickets (I know of at least one case**), each with different functionalities, its overall system lacks conceptual integrity.**
- Architectural design can be represented using following models:
  - Structural models ->show arch as organized collection of components
  - Framework models ->more abstract design , hiding the repeatable components
  - Dynamic models -> show how program structure may change because of an external event
  - Process models ->focus on functional requirements related to business process design and implementation
  - Functional models -> represent system functionality hierarchy wrt dependence

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Architecture design properties

- **Structural properties.** defines the:

- components of a system (e.g., modules, objects, filters) and
- the manner in which those components are packaged
- and interact with one another.
- For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods

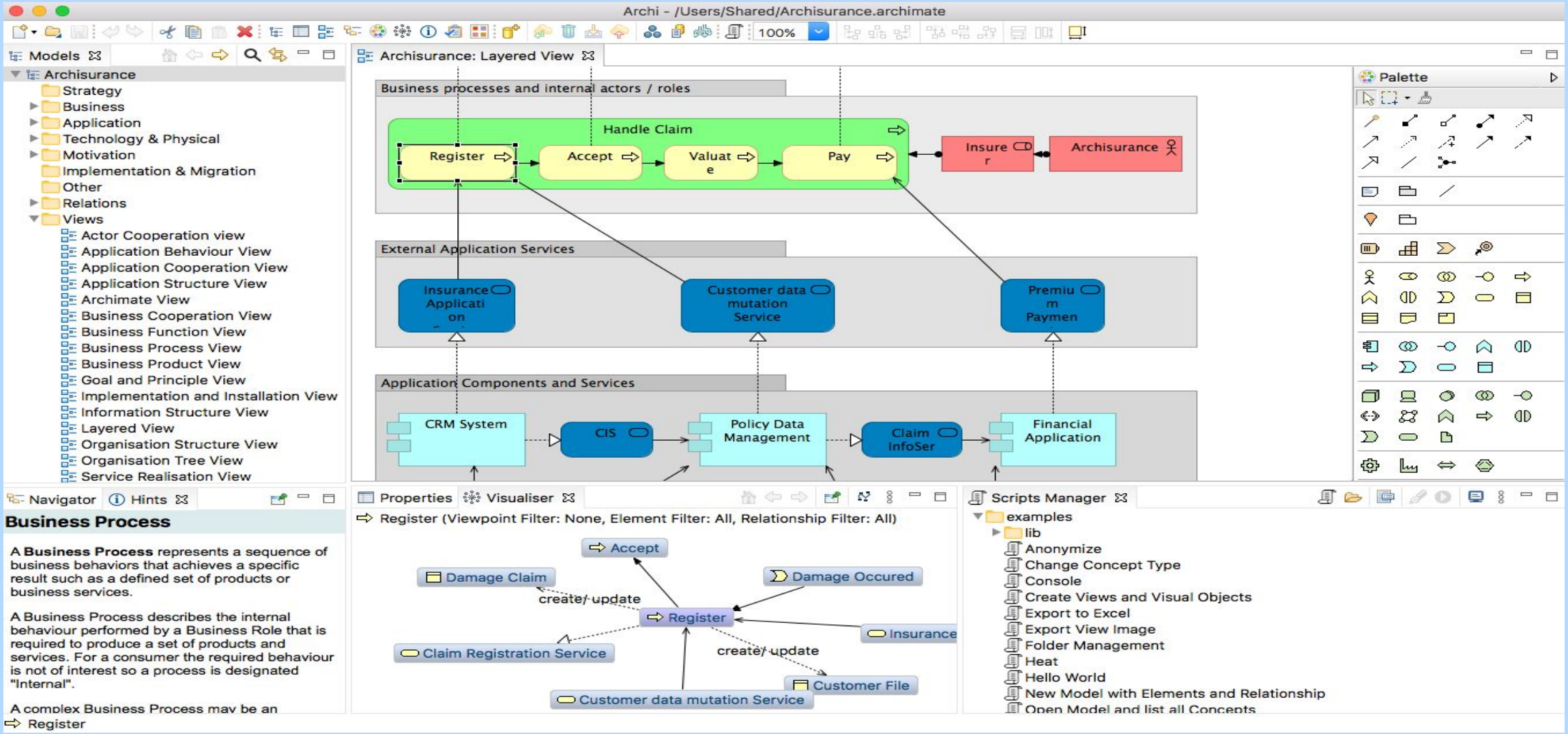
- **Extra-functional properties.**

- Show **how the design architecture achieves requirements for**:
- performance, capacity, reliability, security, adaptability, and other system characteristics.

- **Families of related systems.** The architectural design should **draw upon repeatable patterns that are commonly encountered** in the design of families of similar systems.
- In essence, **the design should have the ability to reuse architectural building blocks**.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Architecture Design developed using ArchiMate tool



Credits: https://www.archimatetool.com/

Separation of Concerns

- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* -> feature /behavior that is specified as part of the requirements model for the software(something that is of interest or significance to a stakeholder or a group of stakeholders).
- By separating concerns into smaller, and therefore more manageable pieces, **a problem takes less effort and time to solve.**
- **shown in other design concepts like refinement, modularity, functional independence, aspects.**

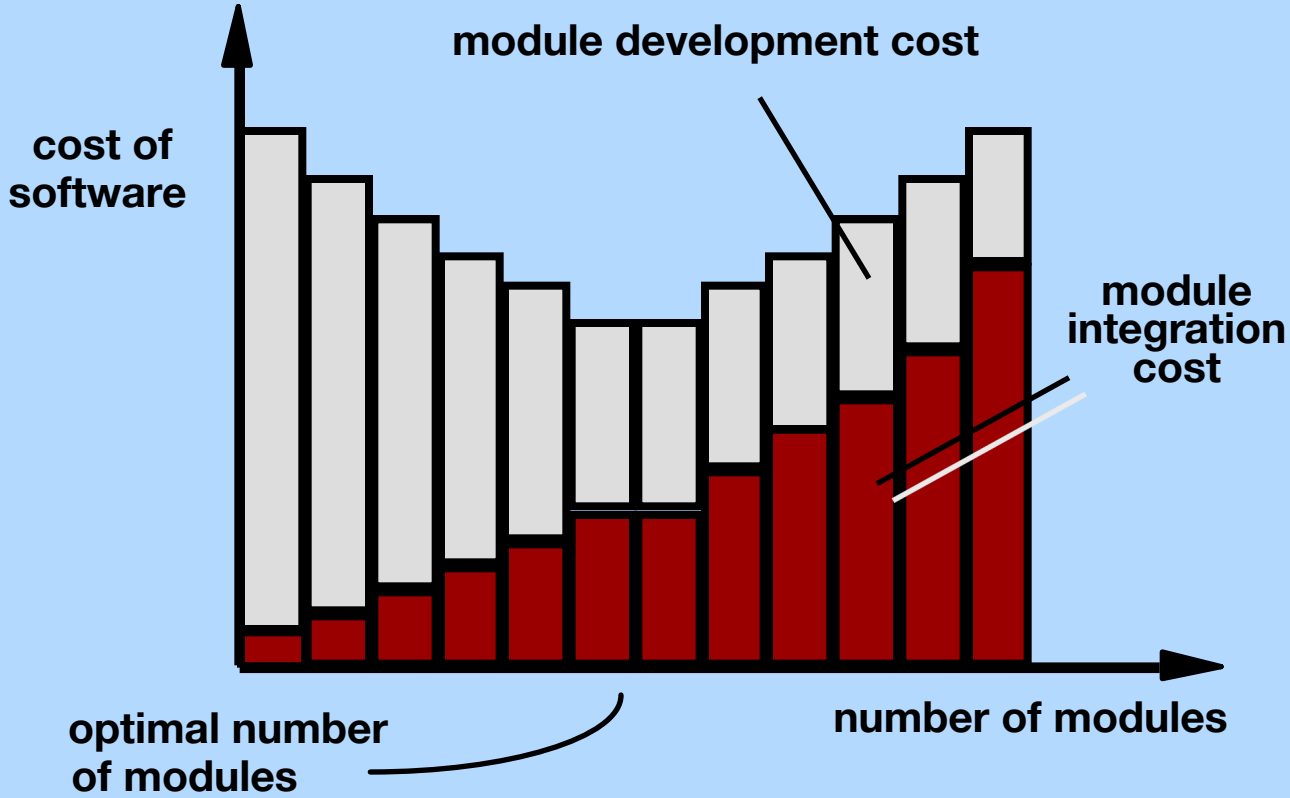
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Modularity

- Modularity is the implementation of separation of concerns
- Software is decomposed in to separately named addressable components called as modules
- They are integrated to fulfil the set of requirements
- Follow divide and conquer approach
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.
- **This module break down must be to an optimum level because more modules leads to more integration cost.**

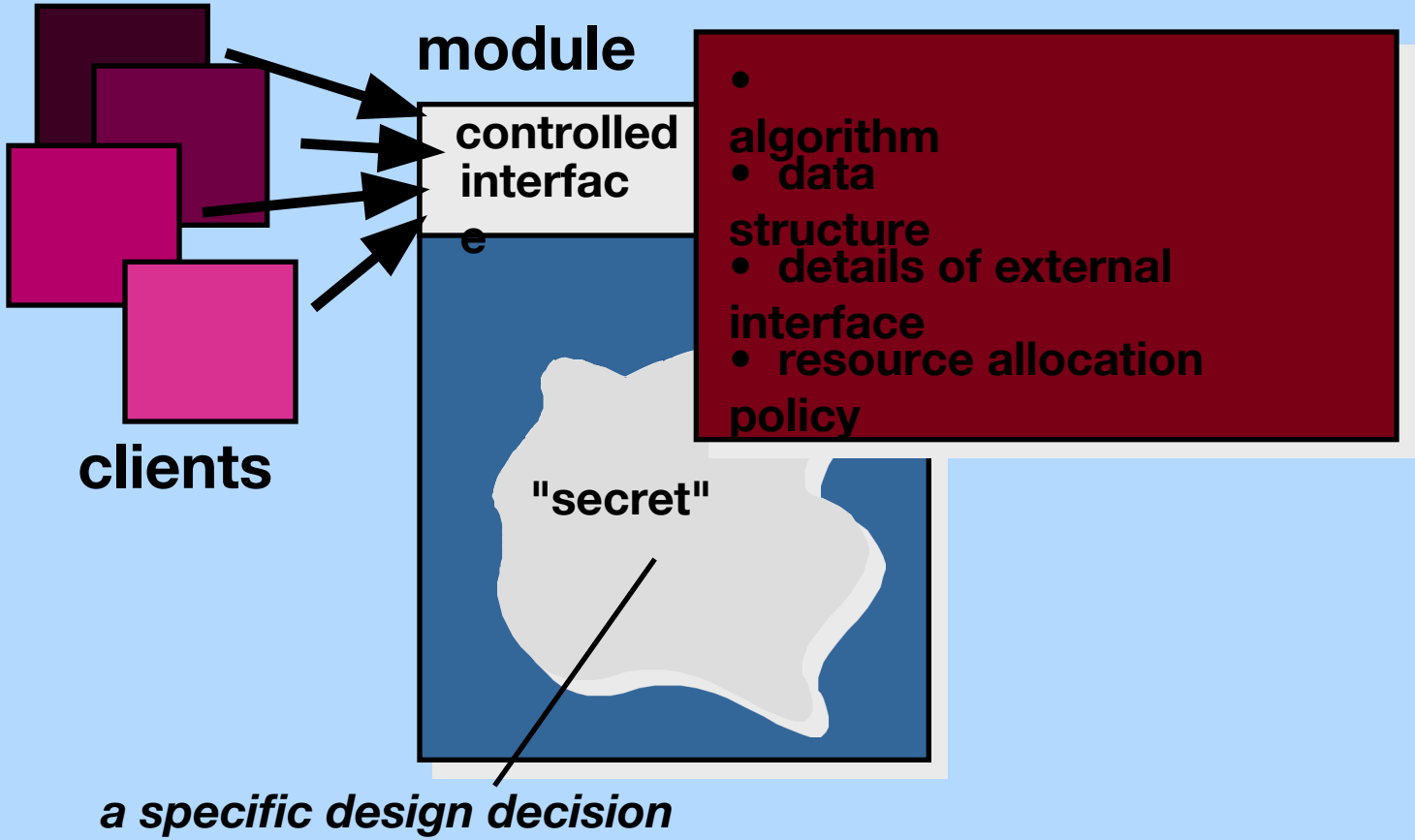
Modularity: Trade-offs

What is the "right" number of modules for a specific software design?



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Information Hiding



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Why Information Hiding?

- Modularity leads to access of information to particular entities.
- Information hiding is used to enforce the access constraints to a local details as well as the data structures used.
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software
- Modules are the ones that are not dependent and they don’t want their data to get shared

These slides are designed to accompany *Software Engineering: A Practitioner’s Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Refactoring

- Simplifies the design of a component without changing its behaviour by eliminating redundant codes and designs that might lead to system failures.
- So improves the internal structure of the system without disturbing the external behavior of the code

- When software is refactored, the existing design is examined for
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - or any other design failure that can be corrected to yield a better design.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Functional Independence

- Design a software in a way that can each module addresses a subset of requirements and it has an interface when viewed form other modules.
- Advantages:
  - Easy to develop
  - Easy task scheduling
  - Less error propagation chances
  - Reusable modules
- Independence is assessed using two criteria:

- *Cohesion* is an indication of the **relative functional strength of a module**.
  - A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- *Coupling* is an indication of the **relative interdependence among modules.**
  - Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Aspects

- Consider two requirements, *A* and *B*. Requirement *A* *crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account. [Ros04]
- An *aspect* is a representation of a cross-cutting concern.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Aspects—An Example

- Consider two requirements for the **SafeHomeAssured.com** WebApp.
- Requirement *A* is described via the use-case **Access camera surveillance via the Internet.** A design refinement would focus on those modules that would **enable a registered user to access video from cameras placed throughout a space.**
- Requirement *B* is a generic security requirement that states that ***a registered user must be validated prior to using* SafeHomeAssured.com.** This requirement is applicable for all functions that are available to registered *SafeHome* users.

- As design refinement occurs, ***A**** **is a design representation for requirement *A*** and ***B**** **is a design representation for requirement *B*.** Therefore, *A** and *B** are representations of concerns, and B* *cross-cuts* A*.

- An *aspect* is a representation of a cross-cutting concern. Therefore, the design representation, *B**, of the requirement, *a registered user must be validated prior to using* **SafeHomeAssured.com,** is an aspect of the *SafeHome* WebApp.

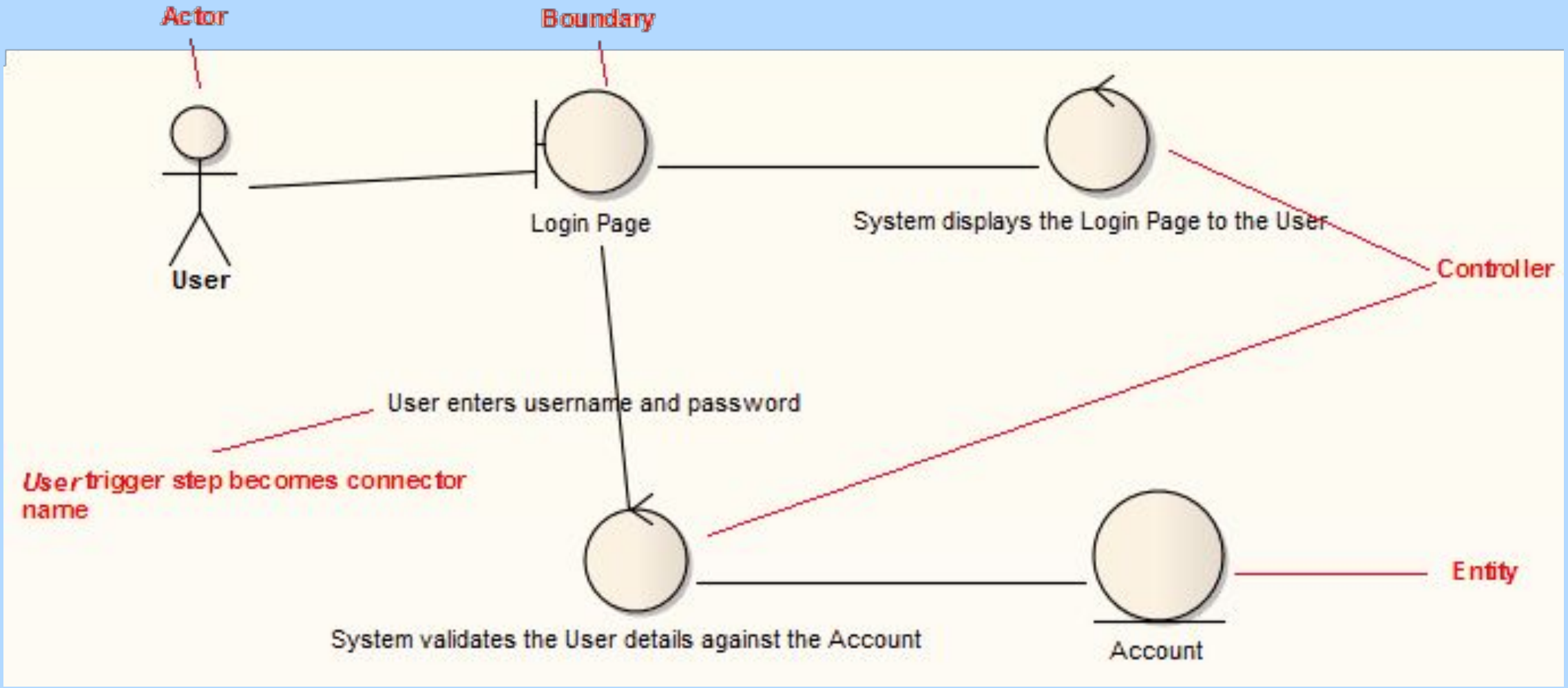
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

OO Design Concepts

- Design classes
  - Entity classes (represents system data)
  - Boundary classes (Objects that interface with system actors)
    - Windows, screens and menus are examples of boundaries that interface with users.
  - Controller classes (Objects that mediate between boundaries and entities.)
- Inheritance—all responsibilities of a superclass is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

OO Design Concepts



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Design Classes

- Analysis classes are refined during design to become entity classes
- Boundary classes are developed during design to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
  - Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- Controller classes are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

Design Model Elements

- Data elements
  - Data model --> data structures
  - Data model --> database architecture
- Architectural elements
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and “styles”
- Interface elements
  - the user interface (UI)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components.
- Component elements
- Deployment elements

These slides are designed to accompany *Software Engineering: A Practitioner’s Approach, 7/e* (McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.