Design Engineering: process, concepts, and model (I)

Design: a preface

- What is it?
  - Recall the 'modeling' in software process
- What is the goal?
  - Firmness (bug free)
  - Commodity (useful)
  - Delight (easy to use)
- How to accomplish the goal? (high level)
  - Diversification
  - Convergence

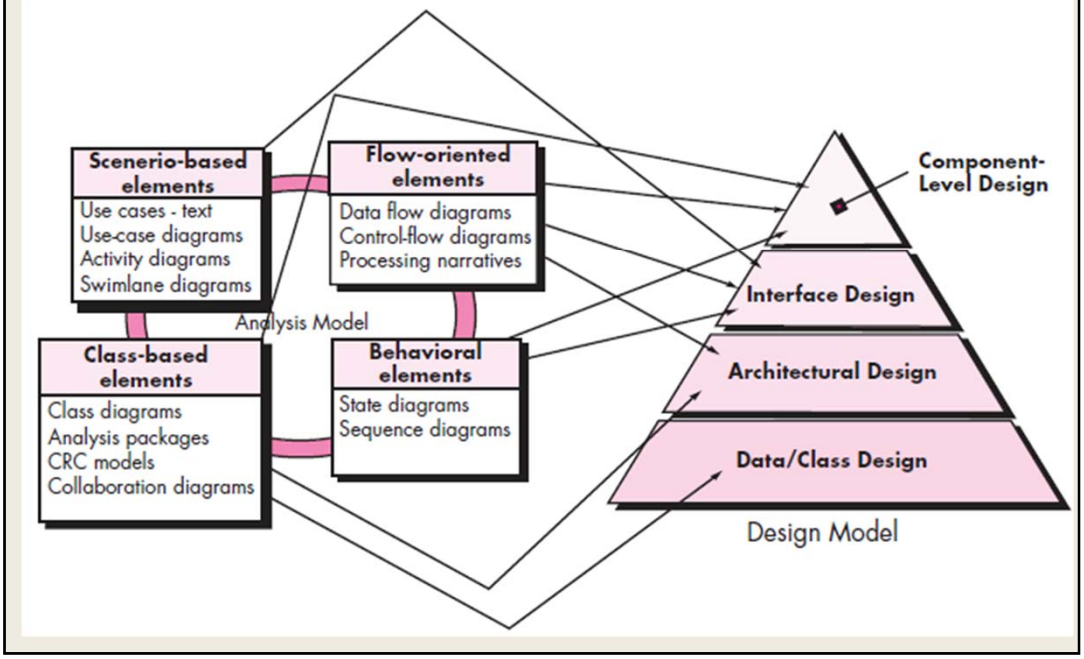
Software design is the last software engineering action within the **modeling activity** and sets the stage for **construction** (code generation and testing). Two kinds of modeling: analysis model and design model

The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight.

Firmness: A program should not have any bugs that inhibit its function.
Commodity: A program should be suitable for the purposes for which it was intended.
Delight: The experience of using the program should be a pleasurable one.

Diversification is the **acquisition** of a repertoire of alternatives, the raw material of design
components, component solutions, and knowledge, all contained in catalogs, textbooks, and the mind
You must pick and choose elements from the repertoire that meet the requirements defined by requirements engineering and the analysis model
Alternatives are considered and rejected and you **converge** on "one particular configuration of components, and thus the creation of the final product

Translation: analysis to design



Now you see why we need different elements in requirements modeling?
Each of the elements of the requirements model (Chapters 6 and 7) provides information that is necessary to create the four design models required for a complete specification of design.

Data/Class design: transforms analysis classes into implementation classes and data structures
Architectural design: defines relationships among the major structural elements of the software, the architectural styles and patterns
Interface design–defines how software elements, hardware elements, and end-users communicate. Usage scenarios and behavioral models are used
Component-level design–transforms structural elements into procedural descriptions of software components. Class-based models and behavioral models server as the basis

Only scenario-based elements serve just one design model

Design: the quality focus

- Design
  - provides software representations that can be *assessed* for quality
  - is where quality is fostered
- Where else did we have this focus?

The importance of software design can be stated with a single word—*quality*.

Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system.

Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process

The quality is also the focus on the entire software process: quality -> timeliness because of less rework

Design: the process

- Characteristics of a good design
  - The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
  - The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
  - The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Software design is an iterative process;

As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.

These can still be traced to requirements, but the connection is more subtle.

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews; McGlaughlin [McG91] suggests
three characteristics that serve as a guide for the evaluation of a good design:

Design: the process

- Evaluate a design
  - A design should exhibit an architecture that,
    1. has been created using recognizable architectural styles or patterns,
    2. is composed of components that exhibit good design characteristics, and,
    3. can be implemented in an evolutionary fashion.
  - A design should be modular
    - The software should be logically partitioned into elements or subsystems
  - A design should contain distinct representations of data, architecture, interfaces, and components.
  - A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

to evaluate the quality of a design representation,
you and other members of the software team must establish technical criteria for good design.

Design: the process

- Evaluate a design
  - A design should lead to components that exhibit independent functional characteristics.
  - A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
  - A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
    - The correctness of engineering methods shall be exhibited by its repeatability.
  - A design should be represented using a notation that effectively communicates its meaning.

These design guidelines are not achieved by chance. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

Design: the process

- FURPS - Quality attributes of a design
  - Functionality
  - Usability
  - Reliability
  - Performance
  - Supportability

Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
Usability is assessed by considering human factors, overall aesthetics, consistency, and documentation
Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
Performance is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
Supportability combines the ability to extend the program (extensibility), adaptability, serviceability

Design: the process

- Design principles
  - The design process should not suffer from 'tunnel vision.'
    - Always maintain a world-view of system design.
  - The design should be traceable to the analysis model.
  - The design should not reinvent the wheel.
    - Deploy reusable patterns: design patterns etc…
  - The design should "minimize the intellectual distance" [DAV95] between the software and the problem as it exists in the real world.
    - Design will reflect the physical word it intends to model.
  - The design should exhibit uniformity and integration.
    - Design shall be inherently consistent, not spaghetti-like.

Design: the process

- Design principles
  - The design should be structured to accommodate change.
    - Improving functional independency for each module will make change management easier.
  - The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
    - Improve the robustness.
  - Design is not coding, coding is not design.
    - Design shall maintain a right degree of abstraction.
  - The design should be assessed for quality as it is being created, not after the fact.
    - Quality assurance is an umbrella activity, and shall be weaved into design engineering.
  - The design should be reviewed to minimize conceptual (semantic) errors.
    - Formal technical review is also an umbrella activity.

Design: the process

- Evolution of design
  - Structured programming
  - The translation of data flow or data structure
  - Object-oriented approach
  - Software architecture, Design pattern
  - Aspect-oriented methods, model-driven development, test-driven development

The evolution of software design is a continuing process that has now spanned almost six decades

Early design work: **Modular programs and methods** for refining software structures in a top down manner

Common characteristics of these design methodologies:
(1) a mechanism for the translation of the requirements model into a design representation,
(2) a notation for representing functional components and their interfaces
(3) heuristics for refinement and partitioning
(4) guidelines for quality assessment

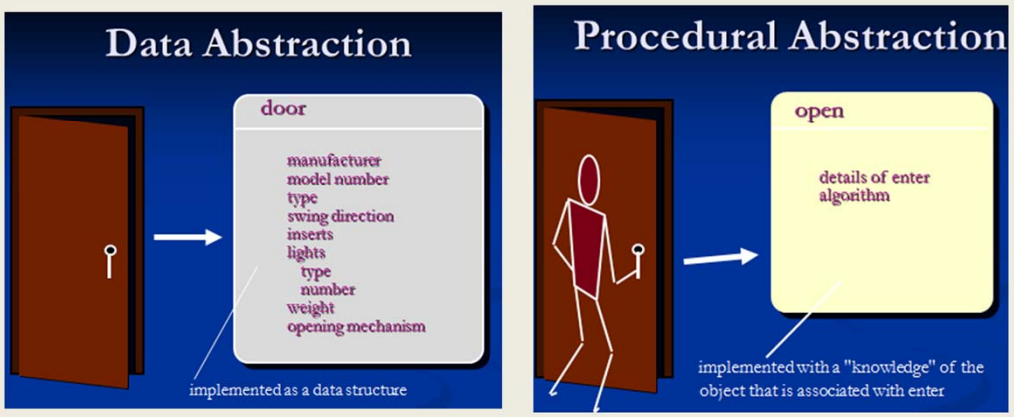
Design: fundamental concepts

- Abstraction—data, procedure, control
- Architecture—the overall structure of the software
- Patterns—"conveys the essence" of a proven design solution
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- Refinement—elaboration of detail for all abstractions
- Refactoring—a reorganization technique that simplifies the design

There are important software design concepts that span both traditional and object-oriented software development.

Design: fundamental concepts

- Abstraction



A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction
implies these functions, but specific details are suppressed. An example: the word “play” for an mp3 file (there are many procedural steps underneath)

A *data abstraction* is a named collection of data that describes a data object. E.g., we can define a data abstraction called mp3 song; data abstraction for **song** would encompass a set of attributes that describe the song**.**

Design: fundamental concepts

- Architecture

  What is it?

  – the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.
    - Structural properties
    - Extra-functional properties
    - Families of related systems

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system
; serves as a framework from which more detailed design activities are conducted.

In a broader sense, however, components can be generalized to represent major system elements and their interactions.
One goal of software design is to derive an architectural rendering of a system.

Structural properties. This aspect of the architectural design representation 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. The architectural design description should address 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.

Design: fundamental concepts

- Architecture

  How to represent/describe it?

  - Structural models
  - Framework models
  - Dynamic models
  - Process models
  - Functional models

Architectural Design Language (ADL)is available to represent design models

Structural models
Represent architecture as an organized collection of program components
Framework models
Increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns)
Dynamic models
Address the behavioral aspects of the program architecture, indicating how the structure of system configuration may change as a function of external events
Process models (design process, not software process)
Focus on the design of the business or technical process that the system must accommodate
Functional models
Represents the functional hierarchy of a system

Design: fundamental concepts

- Pattern
  - named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns

a design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.

Design: fundamental concepts

- Pattern (Elements)
  - *Pattern name*—describes the essence of the pattern in a short but expressive name
  - *Intent*—describes the pattern and what it does
  - *Also-known-as*—lists any synonyms for the pattern
  - *Motivation*—provides an example of the problem
  - *Applicability*—notes specific design situations in which the pattern is applicable
  - *Structure*—describes the classes that are required to implement the pattern
  - *Participants*—describes the responsibilities of the classes that are required to implement the pattern
  - *Collaborations*—describes how the participants collaborate to carry out their responsibilities
  - *Consequences*—describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented
  - *Related patterns*—cross-references related design patterns

a design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

Design: fundamental concepts

- SC (Separation of Concerns)
  - Concern: a feature or behavior that is specified as part of the requirements model for the software
  - Why SC?
    - Less effort and time to solve a problem

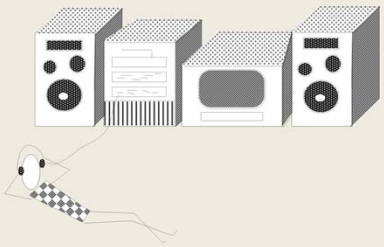
Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently

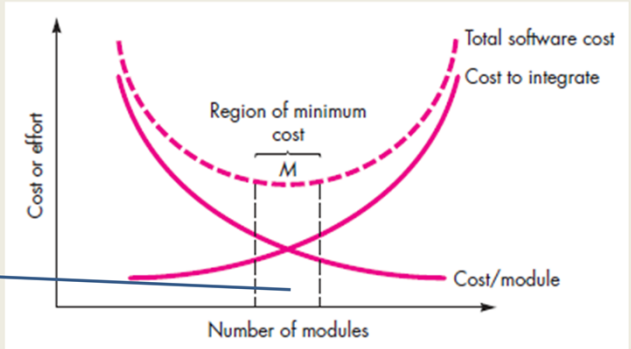
separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

The perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. this leads to a divide-and-conquer strategy—it's easier to solve a complex problem when you break it into manageable pieces.

Design: fundamental concepts

- Modularity
  - break the design into many modules
  - Why?
    - Easier to understand the software
    - Less expensive to build it





optimal

Modularity is the single attribute of software that allows a program to be intellectually manageable [Mye78].

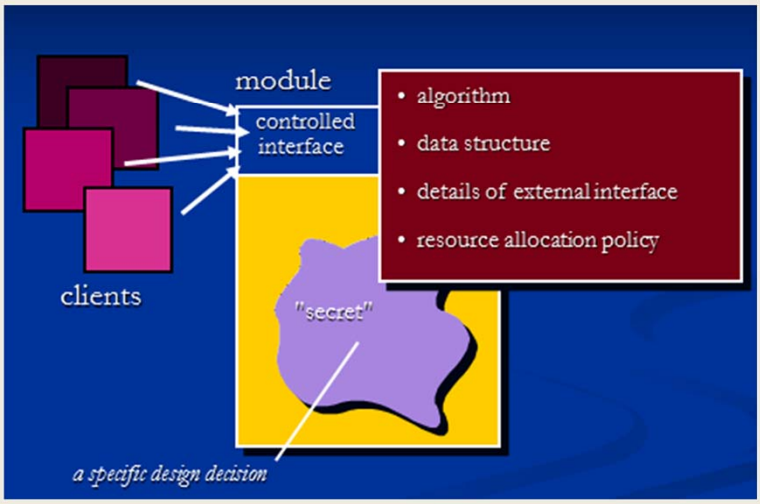
Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.

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.

Design: fundamental concepts

- Information hiding
  - Recall the “wall”



The concept of modularity leads you to a fundamental question: “How do I decompose a software solution to obtain the best set of modules?”

The principle of information hiding [Par72] suggests that modules be “characterized by design decisions that (each) hides from all others.”

Abstraction helps to define the procedural (or informational) entities that make up the software.

Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module [Ros75].

Design: fundamental concepts

- Information hiding
  - Why?
    - Reduces the likelihood of "side effects"
      - The state of an object is decided by its attributes.
      - Class's implementation is responsible for its correctness.
    - Limits the global impact of local design decisions
    - Emphasizes communication through controlled interfaces
    - Discourages the use of global data
      - Using global data decreases the modularity of the code.
    - Leads to encapsulation
      - an attribute of high quality design
    - Results in higher quality software

The goal is to hide the details of data structures and procedural processing behind a module interface

Knowledge of the details need not be known by users of the module

Design: fundamental concepts

- Functional independence
  - How to measure

    COHESION - the degree to which a module performs one and only one function.

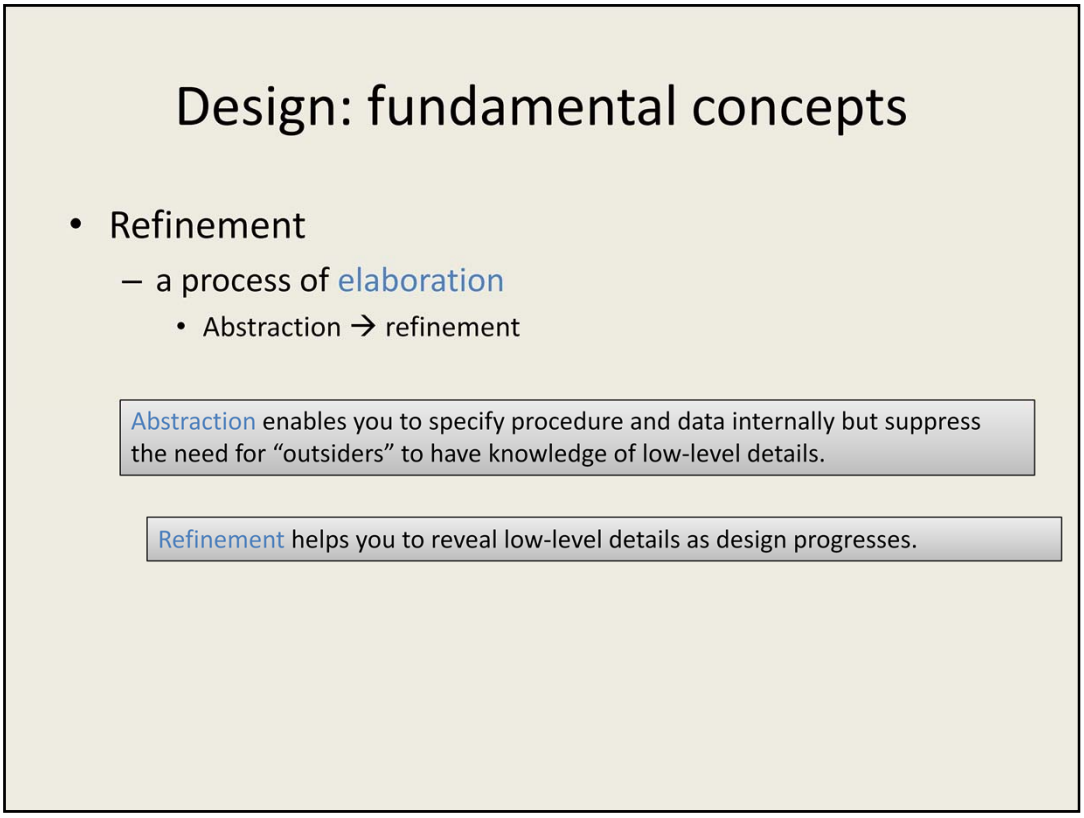
    COUPLING - the degree to which a module is "connected" to other modules in the system.
  - Rule of thumb
    - High cohesion
    - Low coupling

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.

Cohesion is an indication of the relative functional strength of a module.
Coupling is an indication of the relative interdependence among modules.

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.



Design steps: Abstraction -> Refinement

1) Begin with a statement of function (or description of information) that is defined at a high level of abstraction

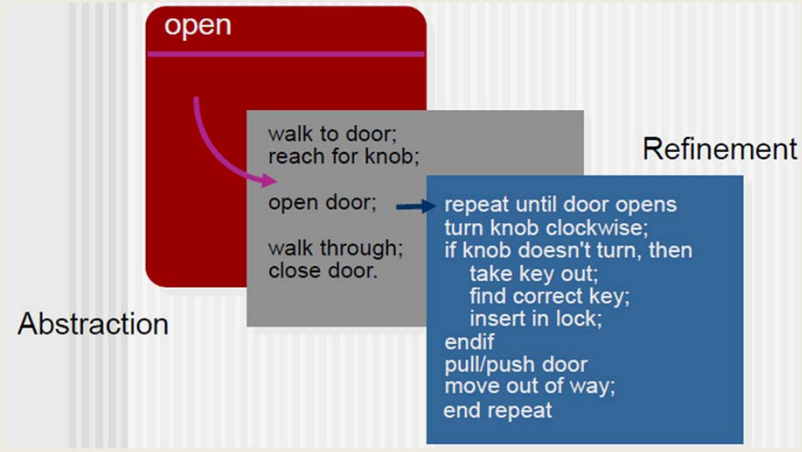
•the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information.

2) Elaborate then on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts; Both concepts allow you to create a complete design model as the design evolves.

Design: fundamental concepts

- Refinement
  - Step-wise refinement
    - **successively refining levels of procedural detail.**



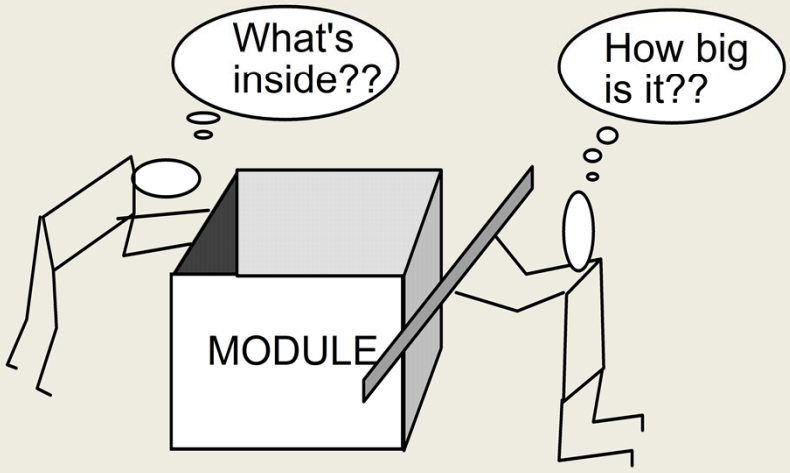
Stepwise refinement is a top-down design strategy

A program is developed by successively refining levels of procedural detail.

A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached

Design: fundamental concepts

- Refinement
  - Two views of module sizing



Design: fundamental concepts

- Aspects
  - representation of a cross-cutting concern
    - some characteristic of the system that applies across many different requirements
  - example?
    - SafeHome: ACS-DCV crosscuts user validation

In an ideal context, an aspect is implemented as a **separate module (component)** rather than as software fragments that are "scattered" or "tangled" throughout many components

A crosscutting concern is some characteristic of the system that applies across many different requirements.
The concern here is the design representation for a requirement.

Requirement *A crosscuts* requirement *B*, if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account.

Requirement *A* is described via the **ACS-DCV** use case; Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using* **SafeHome**

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* crosscuts A**.

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 app.*

Design: fundamental concepts

- Refactoring
  - A reorganization technique that
    - simplifies the design of component
    - without changing its function and behavior
  - After refactoring, check design against
    - 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.

Refactoring is the process of changing a software system in such a way that it does not alter the **external behavior** of the code [design] yet improves **its internal structure**."

Summary

- Analysis to design: translation
- Design process
  - Quality focus
  - Quality attributes and metrics
- Design concepts
  - The ten fundamental concepts