SE 211
Software Specification and Design II

Introduction to Design

About Design

- Design is ubiquitous in practical affairs
  - Architecture
  - Engineering (Automotive, Industrial, etc.)
  - …
  - Computing

- Design involves making something (where "making" is not from an existing plan).
- Different kinds of problems pose different demands.

About Design (cont'd)

"There are two ways of constructing a software design:

one way is to make it so simple that there are obviously no deficiencies;

the other is to make it so complicated that there are no obvious deficiencies."

C.A.R. Hoare (1985)

Elements of the Design Problem

- Five elements of design problems:
  - Goals
  - Constraints
  - Alternatives
  - Representations
  - Solutions

Design Solutions

- Characteristics of solutions:
  - The solution to a design problem is a description enabling system construction.
  - Design solutions are complex.
  - Most design problems have many acceptable solutions.
  - It is difficult to validate design solutions (consequence of complexity).

Software Design

- Software Architectural Design

- Software Detailed Design

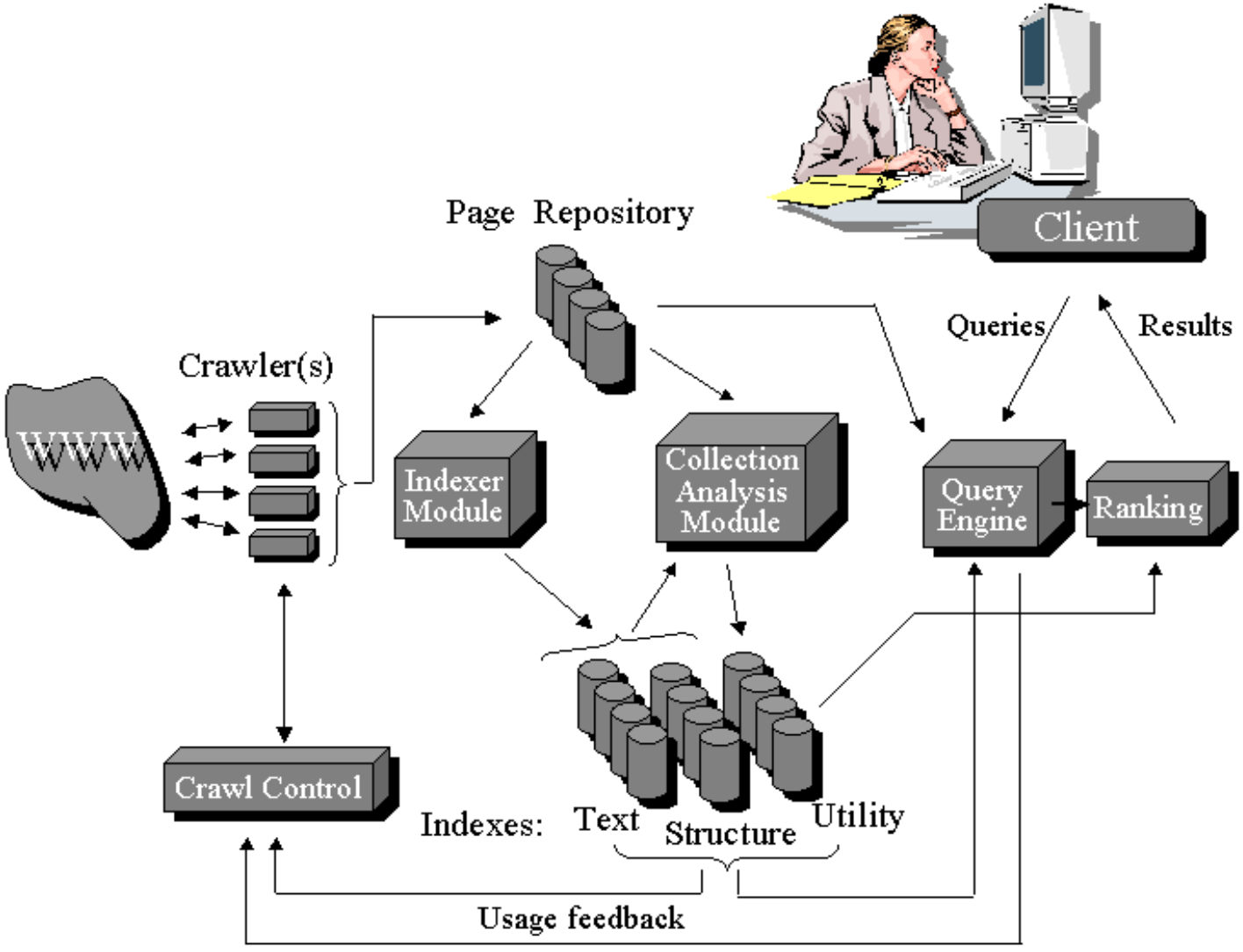
Software Design

- Software Architectural Design
  - The top-level structure and organization of the system is described and the various components are identified.
- Software Detailed Design
  - Each component is sufficiently designed to allow for its coding.

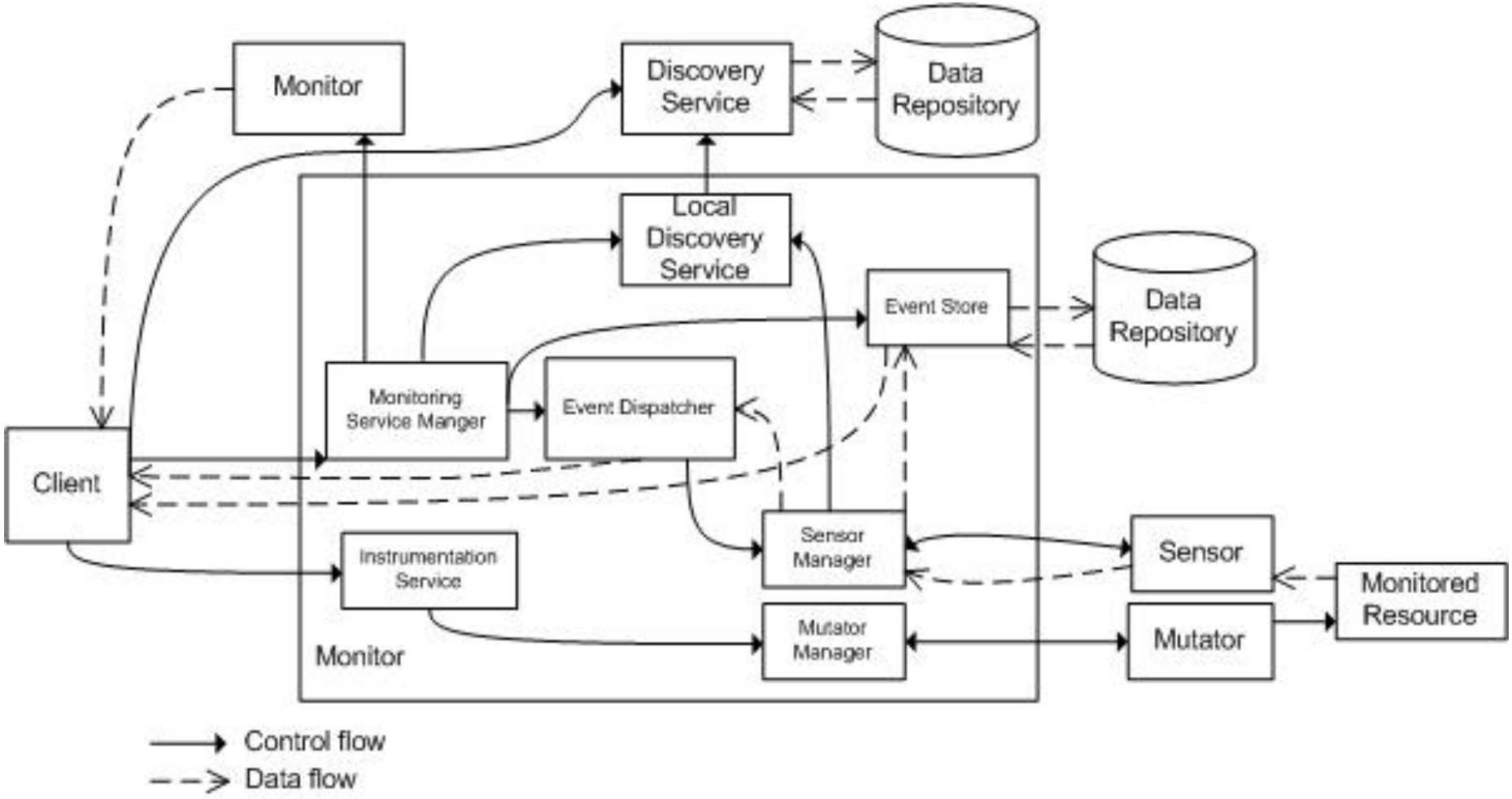
Software Design

- Software Architectural Design
  - The top-level structure and organization of the system is described and the various components are identified.
- Software Detailed Design
  - Each component is sufficiently designed to allow for its coding.
- UI design and Real-time design are “specialized” areas of Software Design.

Architectural Design Sample Diagram #1



Architectural Design Sample Diagram #2



Architectural Design Sample Diagram #3

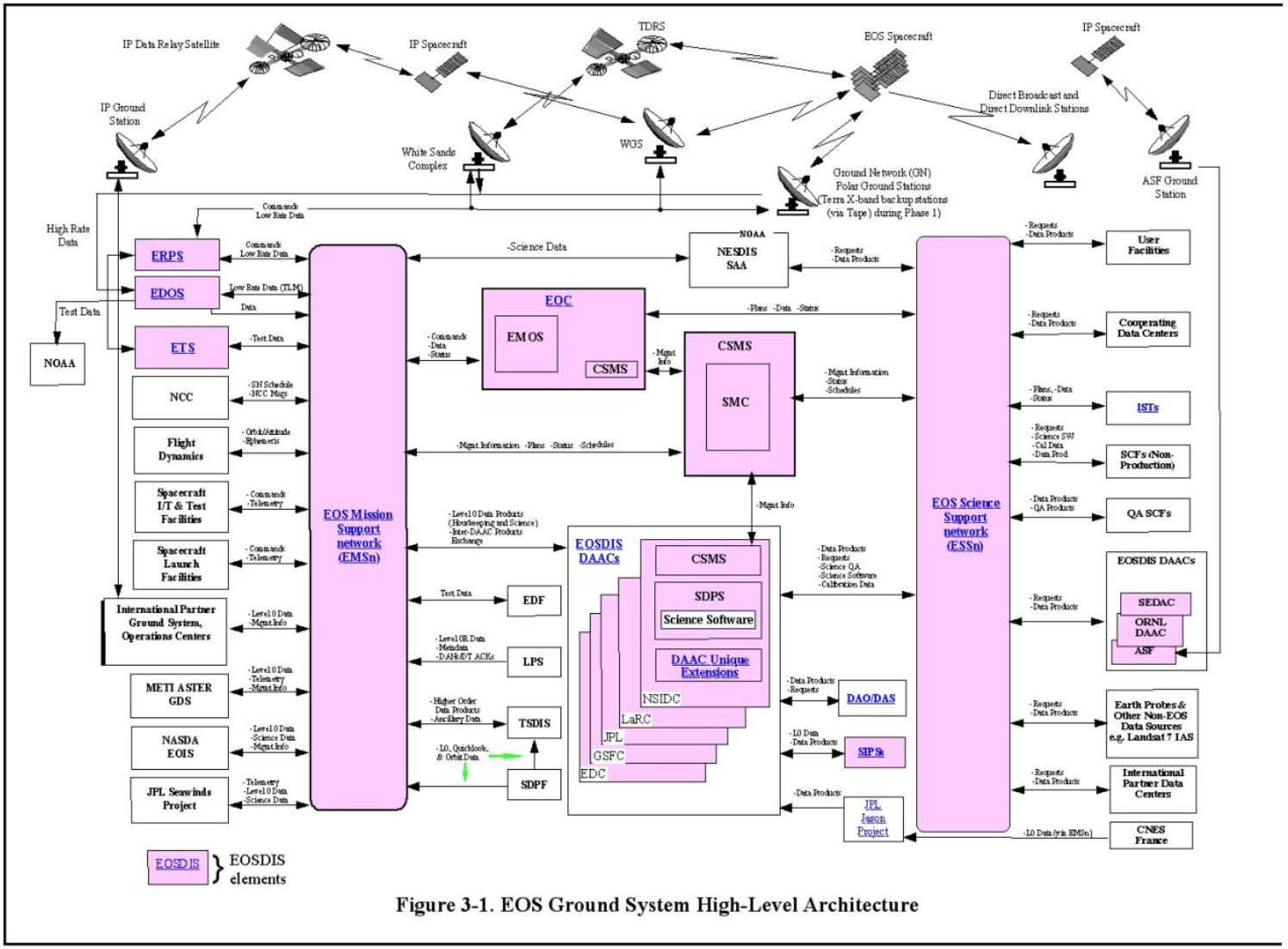
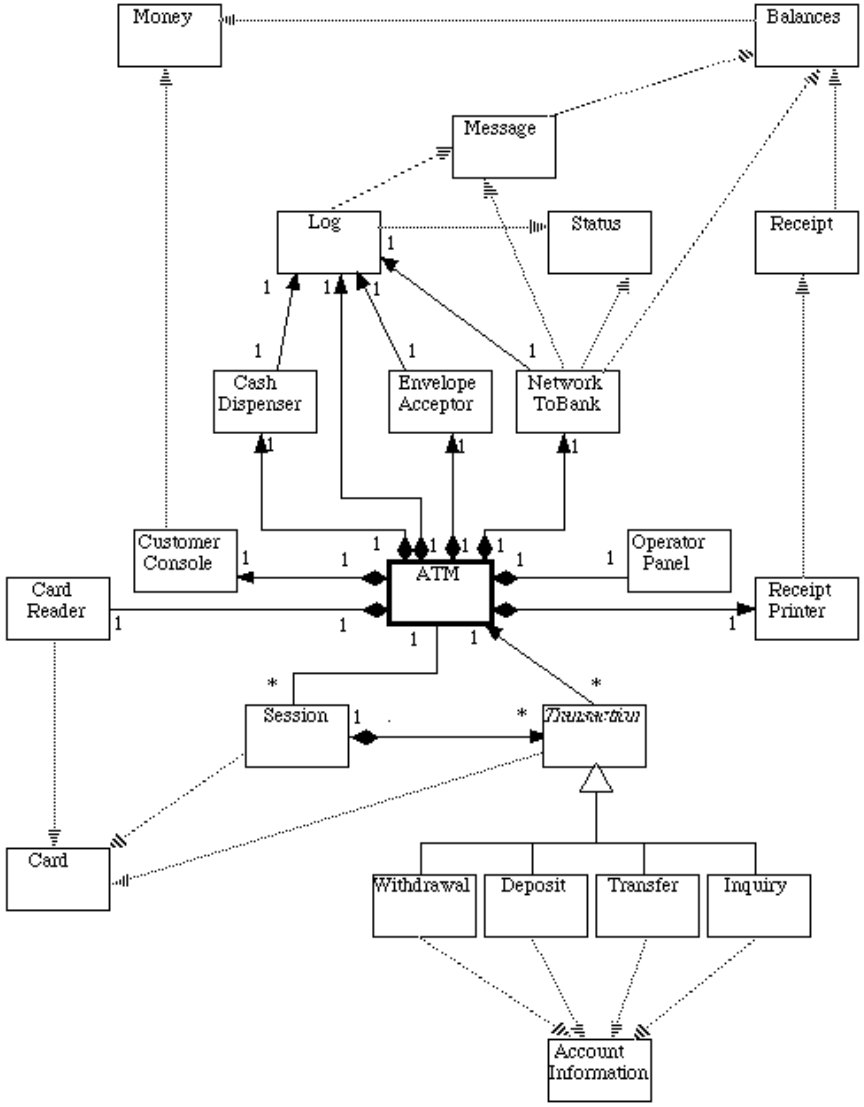


Figure 3-1. EOS Ground System High-Level Architecture

Low-level Design Class Diagram

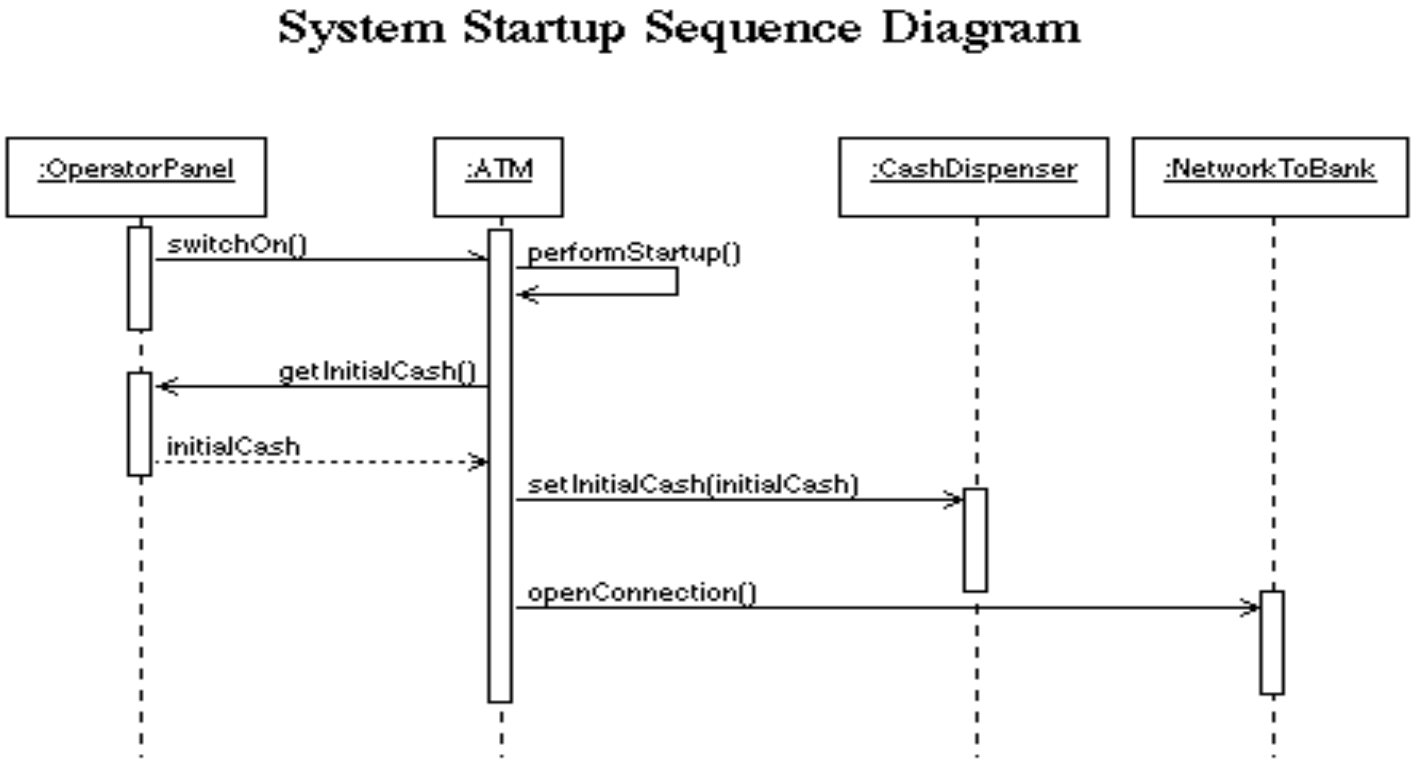


Low-level Design Class Specification

Withdrawal
- from: int - amount: Money
+ Withdrawal(atm: ATM, session: Session, card: Card, pin: int) # getSpecificsFromCustomer(): Message throws Cancelled # completeTransaction(): Receipt

Transfer
- from: int - to: int - amount: Money
+ Transfer(atm: ATM, session: Session, card: Card, pin: int) # getSpecificsFromCustomer(): Message throws Cancelled # completeTransaction(): Receipt

Low-level Design Sequence Diagram



Software Design in the Life Cycle

- Software requirements analysis

- System Architecture
- Software Detailed Design

- System Verification & Validation

Roles of Software Designers

- Systems Engineer
  - Designs systems using a holistic approach, which includes designing how software, hardware, people, etc. collaborate to achieve the system's goal.

- Software Architect
  - Design software systems using (for the most part) a black-box modeling approach; concern is placed on the external properties of software components that determine the system's quality and support the further design of functional requirements.

- Component Designer
  - Focuses on designing the internal structure and behavior of software components identified during the architecture phase; typically, these designers have strong programming skills, since they implement their designs in code.

- User Interface Designer
  - Design the software's user interface; skilled in determining ways that increase the usability of the system

Software Design Fundamentals

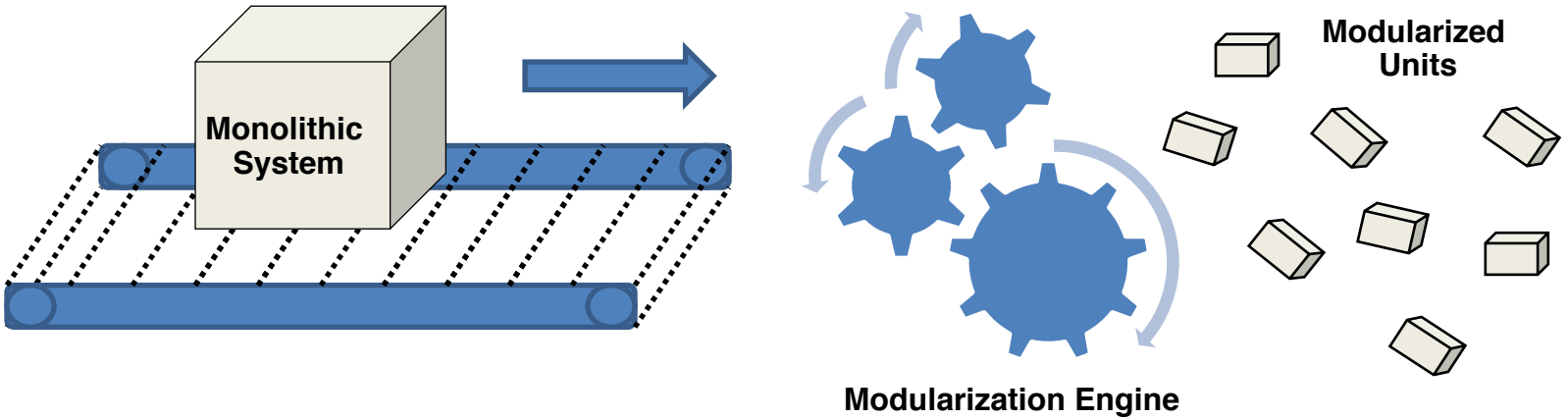
- Within the design process, many principles, strategies, and practical considerations exist to help designers execute the design process in effective and consistent manner.
- General design principles
  - Refer to knowledge matter that has been found effective throughout the years in multiple projects on different domains.
  - Serve as fundamental drivers for decision-making during the software design process.
    - They are used as basis for reasoning and serve as justification for design decisions.
  - Provide designers a foundation from which other design methods can be applied.
  - Are not specific to particular design strategies (e.g., object-oriented) or process.
    - The are fundamental to every design effort.
    - Can be employed during architecture, detailed design, construction design, etc.

Software Design Fundamentals

- Design Principle #1: *Modularization*
  - It is the principle that drives the continues decomposition of the software system until fine-grained components are created.
  - One of the most important design principle, since it allows software systems to be manageable at all phased of the development life-cycle.
  - When you modularize a design, you are also modularizing requirements, programming, test cases, etc.
  - Plays a key role during all design activities; when applied effectively, it provides a roadmap for software development starting from coarse-grained components that are further modularized into fine-grained components directly related to code.
  - Leads to designs that are easy to understand, resulting in systems that are easier to develop and maintain.

Software Design Fundamentals - Modularization

- Conceptual view of modularization:



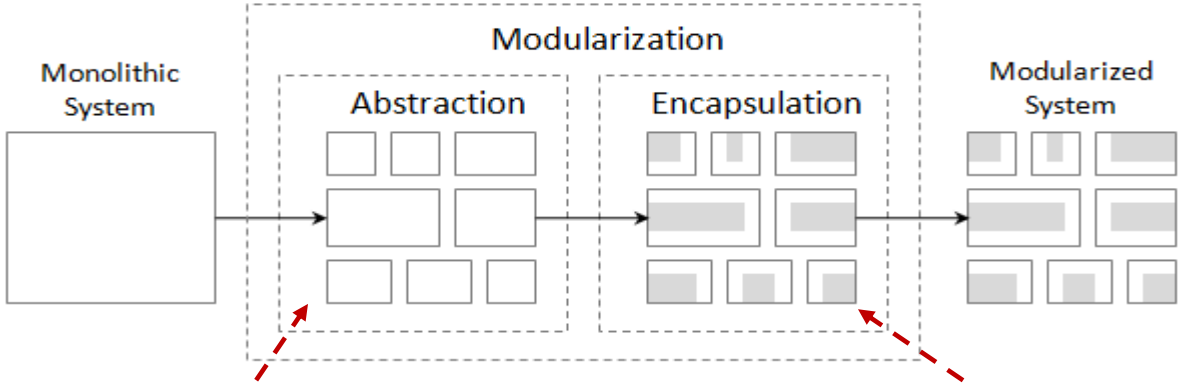
- Modularization is the process of continuous decomposition of the software system until fine-grained components are created. But how do we justify the "modularization engine"?
  - It turns out that two other principles can effectively guide designers during this process
    - Abstraction
    - Encapsulation

Software Design Fundamentals

- Design Principle #2: *Abstraction*
  - Abstraction is the principle that focuses on essential characteristics of entities—in their active context—while deferring unnecessary details.
  - While the principle of modularization specifies what needs to be done, the principle of abstraction provides guidance as to how it should be done. Modularizing systems in ad-hoc manner leads to designs that are incoherent, hard to understand, and hard to maintain.
  - Abstraction can be employed to extract essential characteristics of:
    - Procedures or behavior
    - Data
- *Procedural abstraction*
  - Specific type of abstraction that simplifies reasoning about behavioral operations containing a sequence of steps.
  - We use this all the time, e.g., consider the statement: "*Computer 1* ***SENDS*** *a message to server computer 2*"
    - Image if we had to say, e.g., "*Computer 1 retrieves the server's information, opens a TCP/IP connection, sends the message, waits for response, and closes the connection.*" Luckily, the procedural abstraction *SEND* helps simplify the operations so that we can reason about this operations more efficiently.
- *Data abstraction*
  - Specific type of abstraction that simplifies reasoning about structural composition of data objects.
    - In the previous example, *MESSAGE* is an example of the data abstraction; the details of a *MESSAGE* can be deferred to later stages of the design phase.

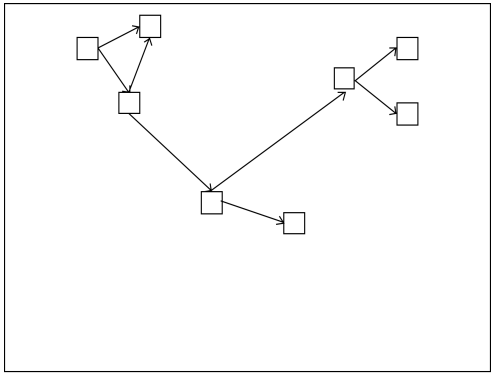
Software Design Fundamentals

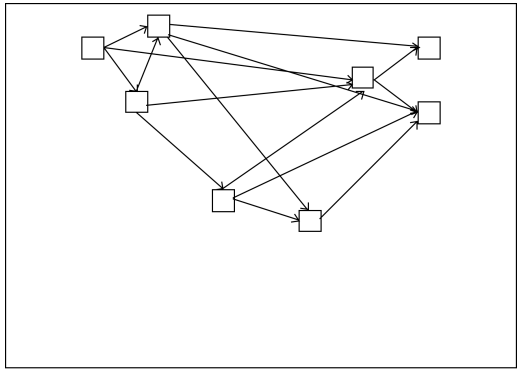
- Design Principle #3: *Encapsulation*
  - Principle that deals with providing access to the services of *abstracted entities* by exposing only the information that is essential to carry out such services while hiding details of how the services are carried out.
  - When applied to data, encapsulation provides access only to the necessary data of abstracted entities, no more, no less.
  - Encapsulation and abstraction go hand in hand.
    - When we do abstraction, we hide details…
    - When we do encapsulation, we revise our abstractions to enforce that abstracted entities only expose essential information, no more, no less.
    - Encapsulation forces us to create good abstractions!
- The principles of modularization, abstraction, and encapsulation can be summarized below.



Software Design Fundamentals

- Design Principle #4: *Coupling*
  - Refers to the manner and degree of interdependence between software modules.
  - Measurement of dependency between units. The higher the coupling, the higher the dependency and vice versa.





Software Design Fundamentals

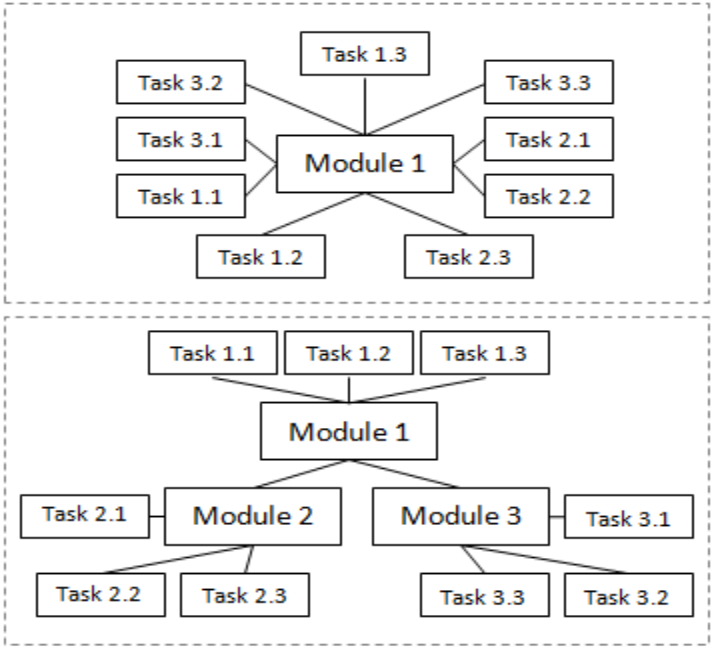
- Important types of coupling include:
  - Content coupling
    - The most severe type, since it refers to modules that modify and rely on internal information of other modules.
  - Common coupling
    - Refers to dependencies based on common access areas, e.g., global variables.
    - When this occurs, changes to the global area causes changes in all dependent modules.
    - Lesser severity than content coupling.
  - Data coupling
    - Dependency through data passed between modules, e.g., through function parameters.
    - Does not depend on other modules' internals or globally accessible data, therefore, design units are shielded from changes in other places.
- In all cases, a high degree of coupling gives rise to negative side effects.
  - Quality, in terms of reusability and maintainability, decrease.
  - When coupling increase, so does complexity of managing and maintaining design units.

Software Design Fundamentals

- Design Principle #5: *Cohesion*
  - The manner and degree to which the tasks performed by a single software module are related to one another.
  - Measures how well design units are put together for achieving a particular tasks.

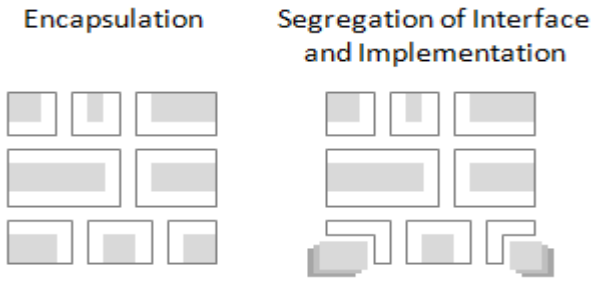
- Cohesion can be classified as:
  - Functional cohesion
  - Procedural (or sequential) cohesion
  - Temporal cohesion
  - Communication cohesion

- High cohesion good, low cohesion bad.



Software Design Fundamentals

- Design Principle #6: *Separation of Interface and Implementation*
  - Deals with creating modules in such way that a stable interface is identified and separated from its implementation.
  - Not the same thing as encapsulation!
  - While encapsulation dictates hiding the details of implementation, this principle dictates their separation, so that different implementations of the same interface can be swapped to provide modified or new behavior.



Software Design Fundamentals

- Design Principle #7: *Sufficiency*
  - Deals with capturing enough characteristics of the abstraction to permit meaningful interaction.
  - Must provide a full set of operations to allow a client proper interaction with the abstraction.
  - Implies minimal interface.

- Design Principle #8: *Completeness*
  - Deals with interface capturing all the essential characteristics of the abstraction.
  - Implies an interface general enough for any prospective client.
  - Completeness is subjective, and carried too far can have unwanted results.

Software Design Strategies

- Object-oriented design strategy
  - Design strategy in which a system or component is expressed in terms of objects and connections between those objects.
  - Focuses on object decomposition.
    - Objects have state
    - Objects have well-defined behavior
    - Objects have unique identity
  - Supports inheritance and polymorphism

- Structured (or Functional) design strategy
  - Design strategy in which a system or component is decomposed into single-purpose, independent modules, using an iterative top-down approach.
  - Focuses on
    - Functions that the system needs to provide,
    - the decomposition of these functions, and
    - The creation of modules that incorporate these functions.
  - Largely inappropriate for use with object-oriented programming languages.

Practical Software Design Considerations

- Design for minimizing complexity
  - Design is about minimizing complexity.
  - Every decision that is made during design must take into account reducing complexity.
  - When faced with competing design option, always choose the one that minimizes complexity

- Design for change
  - Software will change, design with extension in mind.
  - A variety of techniques can be employed through the design phase to achieve this.